NL2Viz: Natural Language to Visualization via Constrained Syntax-Guided Synthesis

Zhengkai Wu
University of Illinois at Urbana-Champaign
USA

Vu Le
Microsoft
USA

Ashish Tiwari
Microsoft
USA

Sumit Gulwani
Microsoft
USA

Arjun Radhakrishna
Microsoft
USA

Ivan Radiček
Microsoft
USA

Gustavo Soares
Microsoft
USA

Xinyu Wang
University of Michigan, Ann Arbor
USA

Zhenwen Li
Peking University
China

Tao Xie
Peking University
China

ABSTRACT

Recent development in NL2Code (Natural Language to Code) research allows end-users, especially novice programmers to create a concrete implementation of their ideas such as data visualization by providing natural language (NL) instructions. An NL2Code system often fails to achieve its goal due to three major challenges: the user's words have contextual semantics, the user may not include all details needed for code generation, and the system results are imperfect and require further refinement. To address the aforementioned three challenges for NL to Visualization, we propose a new approach named NL2Viz with three salient features: (1) leveraging not only the user's NL input but also the data and code context that the NL query is upon, (2) using hard/soft constraints to reflect different confidence in the constraints retrieved from the user input and data/code context, and (3) providing support for result refinement and reuse. We implement a tool for NL2Viz in the Jupyter Notebook environment and evaluate NL2Viz on a real-world visualization benchmark and a public dataset to show the effectiveness of NL2Viz. We also conduct a user study involving 6 data scientist professionals to demonstrate the usability of NL2Viz, the readability of the generated code, and NL2Viz's effectiveness in helping users generate desired visualizations effectively and efficiently.

CCS CONCEPTS

• **Software and its engineering** → *Visual languages*; **Automatic programming**.

ESEC/FSE '22, November 14–18, 2022, Singapore, Singapore

ACM ISBN 978-1-4503-9413-0/22/11…$15.00
https://doi.org/10.1145/3540250.3549140

KEYWORDS

program synthesis, natural language to code, constraint

ACM Reference Format:
Zhengkai Wu, Vu Le, Ashish Tiwari, Sumit Gulwani, Arjun Radhakrishna, Ivan Radiček, Gustavo Soares, Xinyu Wang, Zhenwen Li, and Tao Xie. 2022. NL2Viz: Natural Language to Visualization via Constrained Syntax-Guided Synthesis. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '22), November 14–18, 2022, Singapore, Singapore.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3540250.3549140

1 INTRODUCTION

Recent development in Natural Language to Code (NL2Code) research allows end-users, especially novice programmers to create a concrete implementation of their ideas by providing natural language (NL) instructions. While code generation for general-purpose languages such as Python is still challenging [36], NL2Code for a domain-specific language (DSL) such as SQL [16, 37, 39] or NL2Code in a specific application domain such as competitive programming [18] has witnessed major advances. Given that data science has seen tremendous growth in recent years, data visualization has become a great application domain of NL2Code. The main reason is that data scientists need to frequently produce visualization to help them perform exploratory data analysis (EDA) to discover useful information from data and draw insights to support decision making. Yet it is quite a burden on data scientists as they have to memorize names of data visualization APIs and their many parameters [10]. Indeed, in our user study (Section 4.4), data scientists confirm that they could not memorize all the API options and have to look into API documentation frequently.

It is difficult for an NL2Code tool to achieve its goal due to three major challenges. First, in the stated NL instruction, users may use words whose semantics can be determined only in the context. For example, different NL2Sql approaches include different strategies to handle schema encoding to create a mapping from the user input

to the entities in the database, while few approaches have achieved good adaptability [16]. In visualization, it gets more complicated as users may already write some code to process the data and then use the Natural Language to Visualization (NL2Visualization) tool. So the tool also needs to understand the code context. Second, users may not include all details (needed for code generation) in the NL instruction. For example, when trying to produce a line plot, a user may not specify the name of the data column used for the x-axis especially when there is a data column with index or time value. Third, the results of NL2Code are often imperfect and thus require users' further interaction or update to fix issues in the results. Especially in data analysis, data scientists often need to make quick changes to visualization as data analysis is a data-driven process. The first two challenges on the user input may make the users' further interaction even more necessary.

To address the aforementioned three challenges, in this paper, we propose a new approach and its supporting tool named NL2Viz in the domain of NL2Visualization with three salient features. First, we leverage not only a user's NL input but also the contextual input, i.e., data and code context that the underlying query is upon. The data context includes the data tables that the user is working on and also the intermediate data vectors that the user has produced. The code context includes the previous code and existing plots that the user has produced (including the previous instruction-plot pairs produced by our NL2Viz tool). For example, when the user creates her own filtering function and refers to the function name in her instruction, we would be able to understand and use that in the generated plotting code by leveraging the code context. Second, to better fill the missing or ambiguous details in the NL input, we differentiate between hard constraints and soft constraints retrieved from the NL input and contextual input. The hard constraints are the ones that we have high confidence and could be explicitly specified by the user. For example, the user states that she wants a scatterplot, and then the plot type to be scatterplot would be a hard constraint. Meanwhile the soft constraints are the ones that we do not have high confidence and could be inferred from the context. For example, the user does not mention the data column for the x-axis but the column with time information would be the likely x-axis data. Third, we provide the user interface to allow iterative refinement for the user to further fix or change the results. We allow the user to give additional NL instructions to make changes to the visualization. Moreover, we are not only having the plot as the output but also the working code snippet that produces the plot. The user can also directly make changes on the code snippet; data scientists find making code changes convenient as shown in our user study (Section 4.4).

To better facilitate a user's requirements, we implement our approach as a tool named NL2Viz in the Jupyter Notebook environment [13]. NL2Viz is directly embedded into the user's daily workflow without the burden to switch between different environments. At a high level, NL2Viz first parses the NL instruction (given by the user) using semantic parsing [4, 5] into *symbolic constraints* that the target visualization program needs to satisfy. NL2Viz also generates such constraints after retrieving the data, program, and existing chart context in the current notebook. Next, NL2Viz uses a novel syntax-guided program synthesis algorithm to generate a complete visualization program from these hard/soft constraints. During this process, NL2Viz keeps multiple candidates at each synthesis state and assigns a heuristic fitness score to help prioritize the most likely structure. Finally, NL2Viz can take a further refinement NL instruction to change the generated visualization program or the user can choose to directly use or apply changes to the generated program.

We assess NL2Viz using four evaluations. First, we assess the synthesis accuracy in a one-shot scenario. Our benchmark contains 295 NL instructions that are collected from data scientists and online homework assignments. Overall, NL2Viz is able to achieve an overall accuracy of 74.6%. Second, we assess NL2Viz's accuracy in interactive scenarios. Given an initial plot and an instruction for describing a small change, NL2Viz achieves 62.5% accuracy in 40 scenarios. Third, we also assess NL2Viz in a public dataset [20]. NL2Viz outperforms the state of the art approach in easy to medium categories while achieving comparable overall accuracy of 55.0%. Last, we assess the usability of NL2Viz via a user study, where we ask 6 data scientist professionals to use NL2Viz to complete 5 visualization tasks. The participants are able to successfully complete 4.17 out of the 5 tasks on average. Most participants like NL2Viz and are willing to use it before writing actual visualization code.

This paper makes the following main contributions:

- We propose a novel NL2Code approach that aims to address challenges on the user input and interactions in the application domain of NL2Visualization by leveraging the data/code context, retrieving hard/soft constraints, and providing interactive refinement support.
- We present NL2Viz, an end-to-end synthesis tool implemented in the Jupyter Notebook environment for helping data scientists visualize their data using an NL interface. NL2Viz shows both the chart and the readable code snippet for generating that chart, allowing users to modify, extend, and reuse the code snippet.
- We evaluate our approach on a real-world visualization benchmark and a public dataset to show its applicability. We also conduct a user study with data scientist professionals on real world scenarios, finding that NL2Viz is easy to use and helpful for generating not only plots, but also readable code that could be extended and reused.

In the rest of the paper, Section 2 illustrates our overall approach using a motivating example. Section 3 discusses the implementation of NL2Viz. Section 4 presents our evaluation results. Section 5 discusses related work and Section 6 concludes.

2 OVERVIEW

This section provides a high-level overview of NL2Viz via a motivating example. In this example, a data scientist named Alice wants to study the trend of COVID-19 infection in Europe. She opens Jupyter Notebook [13], a popular platform among data scientists, to load a COVID-19 dataset[1] (Figure 1). Each row in the dataset reports the numbers of daily confirmed cases and deaths for a country in a certain day. It also includes the accumulated numbers of confirmed cases and deaths until that date.

Alice first wants to see the trend of confirmed cases for all countries in Europe. Alice understands that she needs to restrict the

[1] https://data.humdata.org/dataset/novel-coronavirus-2019-ncov-cases

	continent	country	date	daily_confirmed	total_confirmed	daily_death	total_death
915	Europe	Belgium	3/27/2020	1049	7284	69	289
832	Asia	Iran	3/21/2020	966	20610	123	1556
1328	North America	Canada	4/25/2020	678	19245	55	989
749	Europe	Germany	3/15/2020	1210	5795	2	11
801	Asia	Japan	3/19/2020	35	924	0	29
833	Europe	Germany	3/21/2020	2365	22213	17	84
429	Europe	Spain	2/21/2020	0	2	0	0
1378	Asia	Iran	4/29/2020	1073	93657	80	5957
492	Asia	South Korea	2/26/2020	284	1261	2	12
162	Europe	Italy	2/2/2020	0	2	0	0

Figure 1: The "COVID-19" dataset (sampled 10 rows).

`continent` column to "Europe". Because there are multiple countries in Europe, Alice also needs to group data by country before she can iterate and plot a line chart for each country. In the chart, the x-axis is the date sorted chronologically and the y-axis is the confirmed cases for that date. Figure 2a shows the desired chart and code.

The plotting code is non-trivial. Alice not only has to pick the right functions in `matplotlib` (e.g., `plot`), but also needs to process the data and construct the desired parameters for these functions. In our user study (Section 4.4), data scientists usually could not remember the usage of plotting functions and have to look up their documentation or search for similar code in help forums. This context switching breaks the data scientists' workflow and negatively affects their productivity.

In contrast, Alice can perform the same task in NL2Viz by typing *"%plot line showing total confirmed cases for countries in Europe"* (*%plot* is our magic command to invoke NL2Viz in Jupyter Notebook). Because our target audience is data scientists who have sufficient coding skills, NL2Viz shows both the chart and the code to produce it (Figure 2a). Having access to the code allows Alice to tune the chart if she wants. For instance, she can modify the code to change the x-axis tick labels from every 5 days to a different number. Alice can also reuse the code. For example, Alice can easily wrap the synthesized code inside a function that plots the total number of confirmed cases in any given continent, and then loops over the function to create plots for all continents.

Alice also has an option to interactively change the existing charts using NL. For instance, she may type *%plot change y label to "Total confirmed cases"* to update the y-axis label, or *"%plot change to Asia"* to change the chart to countries in Asia (Figure 2b). Doing so is feasible because NL2Viz also uses knowledge of existing charts when synthesizing plots from text.

Our approach. We next explain how our NL2Viz approach synthesizes the desired visualization program from the three modalities of specifications as shown below for the example from Figure 2:

- The program/data context, which includes the "COVID-19" dataset, as shown in Figure 1.
- The visualization context, which includes existing charts and their instructions.
- An NL instruction that describes the desired visualization task, *"%plot line showing total confirmed cases for countries in Europe"* provided by Alice as the instruction.

Given these inputs, NL2Viz synthesizes the desired program in two steps, as shown schematically in Figure 3. In the first *semantic parsing* phase, NL2Viz parses the three inputs into symbolic constraints. Then, in the second *program synthesis* phase, NL2Viz synthesizes a complete program that satisfies these constraints.

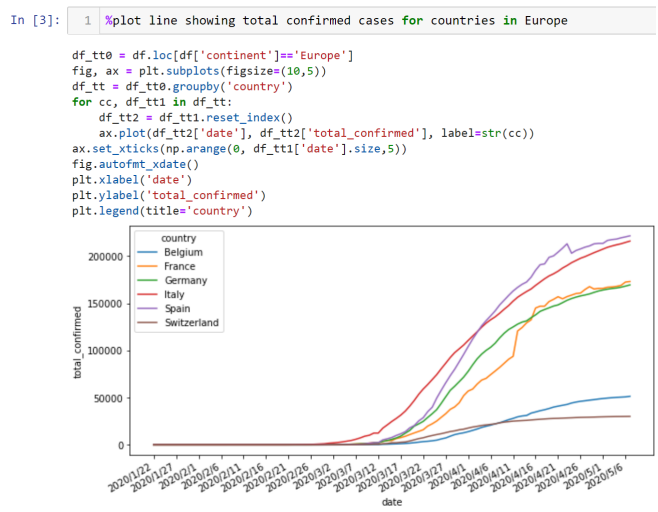
More specifically, given an NL instruction and the current data and program context, NL2Viz uses semantic parsing techniques [5] to generate a *ranked list* L of tuples, each of which T consists of two sets of constraints: a set $\mathcal{R}_{must}$ of *hard constraints* and a set $\mathcal{R}_{may}$ of *soft constraints* (Figure 4 shows a simplified version of our grammar for the semantic parser). The set $\mathcal{R}_{must}$ includes hard constraints that *must* be satisfied by the desired program P, whereas constraints in set $\mathcal{R}_{may}$ are soft, indicating that they *may* be satisfied by P. In NL2Viz, the constraints take the form of (a subset of) rules (of the grammar) used to generate plotting programs. For instance, our semantic parser generates the following tuple $(\mathcal{R}_{must}, \mathcal{R}_{may})$ (among possibly others) for the example from Figure 2. Note that although the NL instruction does not mention `continent`, the parser is able to include that column because it could derive a relationship between "Europe" and `continent` from the data context.

$$
\begin{aligned}
\mathcal{R}_{must} = \{ & \texttt{PlotType} \rightarrow \text{“LinePlot”}, \\
& \texttt{YAxis} \rightarrow \text{“total\_confirmed”}, \\
& \texttt{FilterColumn} \rightarrow \text{“continent”}, \\
& \texttt{GroupColumn} \rightarrow \text{“country”}, \\
& \texttt{FilterValue} \rightarrow \text{“Europe”} \ \} \\
\mathcal{R}_{may} = \{ & \texttt{XAxis} \rightarrow \text{“date”}, \\
& \texttt{DataFrame} \rightarrow \text{“df”} \ \}
\end{aligned}
$$

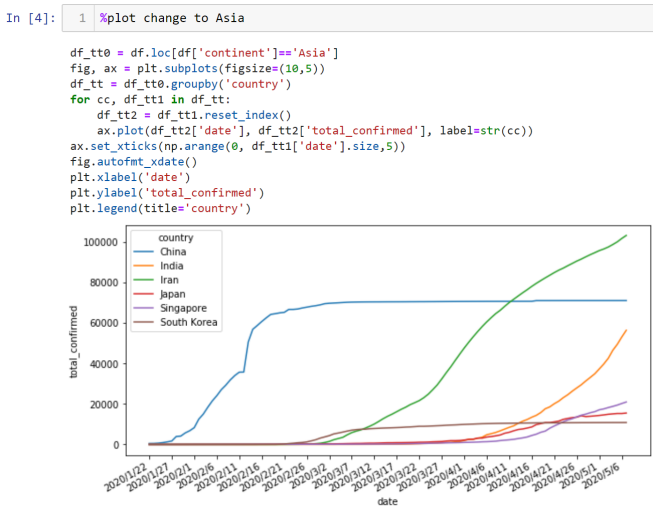
Here, the first rule `PlotType` $\rightarrow$ "LinePlot" in $\mathcal{R}_{must}$ is a hard constraint: when we synthesize the plotting program using the visualization domain-specific language's (DSL's) context-free grammar (CFG), the derivation in the synthesize process should use the rule `PlotType` $\rightarrow$ "LinePlot". The hard constraints in $\mathcal{R}_{must}$ are extracted from the English instruction that directly corresponds to the user's intent. In contrast, the second constraint `XAxis` $\rightarrow$ "date" in $\mathcal{R}_{may}$ is soft, indicating that the program *may* use the "date" column as the x-axis. These constraints in $\mathcal{R}_{may}$ are generated from analyzing the data/program context and existing charts. Since these inputs provide only contextual hints that may be useful for deriving the complete program, we treat $\mathcal{R}_{may}$ as soft constraints.

Once NL2Viz finishes generating hard and soft constraints from specifications, our second program synthesis phase synthesizes a complete visualization program from these constraints. NL2Viz synthesizes a program from the visualization CFG that uses all rules in $\mathcal{R}_{must}$ and avoids, as much as possible, using rules outside of $\mathcal{R}_{may}$. In a final step, NL2Viz translates the program in the DSL to the target language (Python). For instance, given the preceding tuple $(\mathcal{R}_{must}, \mathcal{R}_{may})$, our synthesizer is able to generate the desired program P in Figure 2a. Our synthesizer generates one program P_i for each tuple T_i in L and finally returns a program P that has the smallest cost among all P_i's.

Given the NL instruction in Figure 2b, the semantic parser returns the hard constraint set $\mathcal{R}_{must}$ = {`FilterValue` $\rightarrow$ "Asia", `FilterColumn` $\rightarrow$ "continent"}. The soft constraint set $\mathcal{R}_{may}$ now also includes the constraints of the previous chart. Given these constraints, NL2Viz is able to adapt the chart in Figure 2a to the chart in Figure 2b with minimal guidance.



(a) The user invokes NL2Viz to obtain a plot with code.



(b) The user creates another plot by adapting the existing one.

Figure 2: The screenshots of NL2Viz in Jupyter Notebook while working with the motivating example.

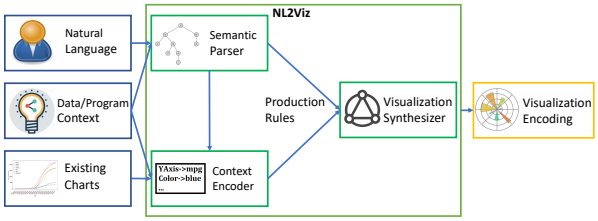


Figure 3: The workflow of NL2Viz

We next discuss the design and the implementation of NL2Viz.

3 NL2VIZ: NATURAL LANGUAGE TO VISUALIZATION

Figure 3 depicts our overall workflow for converting the given NL instruction to visualization code. First, we use a semantic parser to extract $\mathcal{R}_{must}$, the set of constraints that must be used, from the user-provided NL instruction. We then analyze the program and data context to extract a set of constraints that may be used (i.e., $\mathcal{R}_{may}$). Finally, our synthesis algorithm synthesizes the program in our visualization domain-specific language from the extracted constraints, and translates the program into Python.

3.1 Parsing NL Instruction to Constraints

Figure 4 shows a partial simplified version of our attribute grammar (NL grammar) used to parse an NL instruction to constraints. We design this grammar by analyzing online tutorials, visualization courses, and Jupyter Notebooks with high upvotes in Kaggle competitions [14]. We attach semantic rules in the form of S-attributes to the grammar. Each nonterminal in the NL grammar is associated with a list of attribute-value pairs, which corresponds to the semantics of this nonterminal. Given an NL instruction, we use a

```
Root ::= PlotElems
PlotElems ::= PlotElem PlotElems?
PlotElem ::= HistoElems | ··· | FilterElem | GroupElem
HistoElems ::= HistoElem HistoElems?
HistoElem ::= HistoType | Column | Bins | Stack | Log | Density
              ···
GroupElem ::= GroupType | GroupColumns | GroupOperator
```

Figure 4: Simplified version of the NL grammar.

```
Plot ::= plt(Data, Mappings, PlotType, Legends)
Data ::= DataSet | process(Data, Processor)
Processor ::= filter(...) | groupBy(...) | orderBy(...)
Mappings ::= nil | list(Mapping, Mappings)
Mapping ::= xaxis(XAxis, Scaling?, Options?) | yaxis(...) | ...
Scaling ::= range(Float, Float) | log(Scaling) |
            step(Scaling, Float) | ...
Options ::= stacking(Options) | transparent(Options) | ...)
PlotType ::= "Histogram" | "Scatter" | "Lineplot" | ...
Legends ::= legend(Title : String, Labels, ...)
Labels ::= label(XAxisLabel : String, YAxisLabel : String, ...)
```

Figure 5: Simplified visualization DSL

semantic parser [5] (which uses an enhanced CYK algorithm [15]) to parse the instruction into a list of attribute-value pairs. These pairs form our $\mathcal{R}_{must}$ set.

Figure 6 shows a simplified parse structure for the example in Section 2. We obtain the parse structure by making the following enhancements to the CYK algorithm.

Setting attribute values for terminals. We use *annotators* to initialize the attribute values. For example, for the nonterminal Column,

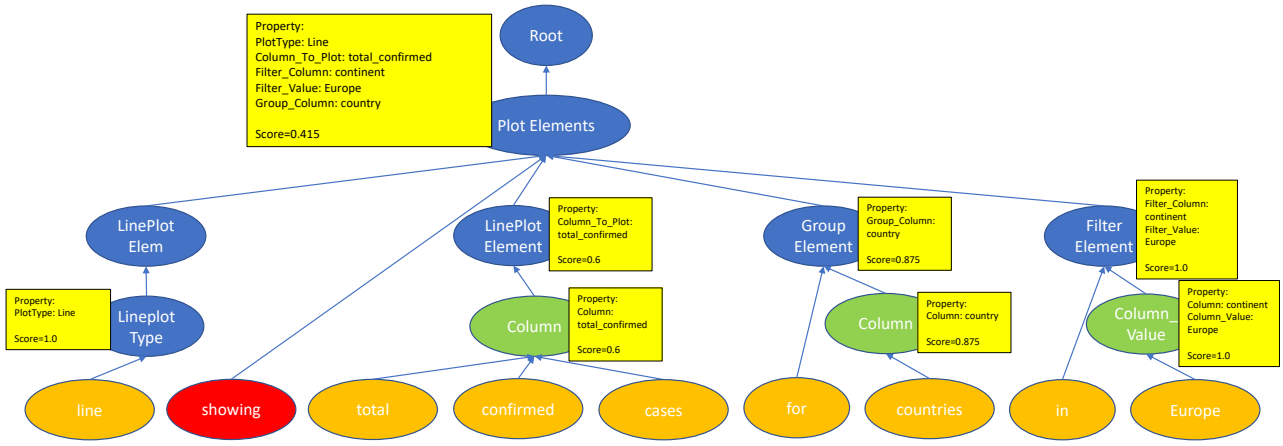


Figure 6: The parse structure for the motivating example.

we have an annotator *ColumnValue* that parses the token "europe" into a `ColumnValue` nonterminal symbol. The *ColumnValue* annotator maps one or multiple consecutive tokens to a value in some column in the given dataset. By using annotators, the semantic parser is able to parse tokens in a data-context-sensitive manner (e.g., parsing "Europe" as a `ColumnValue` and inferring "continent" as its `ColumnName`) and program-context-sensitive manner (e.g., parsing "foo" as a function name if a function of that name appears in the Jupyter Notebook code cell).

Setting attribute values for rule $N ::= N_1 N_2$**.** The attributes are propagated from children to parent following the semantic rules. Most semantic rules just propagate the lists of attributes from children to parent without change. However, in some cases, the semantic rules can change the attribute name in children's list. For example, in the `FilterElem` symbol, the two attribute-value pairs in the attribute of the child, `ColumnValue`, change their attribute name (e.g., `ColumnValue` becomes `FilterValue`).

Fitness score. Each nonterminal also has a fitness score in the range of $[0, 1]$ to represent the probability of producing that parse structure. For instance, the token "countries" does not exactly match the column name "country" in the dataset, hence its symbol "Column" is given a score of 0.87 based on the edit distance of the two strings. The score of the parent symbol is simply the product of the scores of its children.

Note that our NL grammar is inherently ambiguous to capture different interpretations of an NL instruction. From an NL instruction, our semantic parser produces multiple parse structures, each of which has a $\mathcal{R}_{must}$ set and a fitness score.

3.2 Using Program and Data Context to Construct May-use Constraints

Because the instruction usually does not contain all information necessary to synthesize the visualization (i.e., $\mathcal{R}_{must}$ is not complete), NL2Viz uses multiple heuristics to infer the potential omitted information (i.e., $\mathcal{R}_{may}$) from the program and data context . For example, when plotting a scatter plot, if the mapping of data columns to axes is not evident from the user's NL instruction, our NL2Viz approach prefers a categorical column to be on the x-axis.

Our heuristics in NL2Viz also capture popular *data preprocessing patterns*. For example, if the user wants to plot a line plot, but we find that there are multiple points on the same x-axis coordinate in the dataset, then it is likely that there is an inherent grouping step by the column on the x-axis before plotting. NL2Viz analyzes each column to determine (a) the *type* of values in that column, (b) whether the column is categorical, and (c) all the *distinct values* in that column. We use this information to create $\mathcal{R}_{may}$. For example, even if the instruction in Figure 2 does not mention "continent" in the text, NL2Viz infers that `AuxColumn` $\rightarrow$ "continent" in the `FilterElem` rule based on data insights, and adds it to $\mathcal{R}_{may}$.

3.3 Designing the Visualization Domain-Specific Language

Given the sets of $\mathcal{R}_{must}$ and $\mathcal{R}_{may}$, our synthesis algorithm synthesizes a visualization program in a domain-specific language (DSL). Figure 5 shows a simplified version of this visualization DSL (nonterminals start with uppercase, function symbols start with lowercase letters, and terminals are within quotes). Programs in this DSL are then translated to a target visualization library (such as `matplotlib` or `seaborn`) in the final translation step.

To design this DSL, we first perform a preliminary study on the Jupyter Notebook dataset released by Felipe et al. [25]. Based on the stats of different plots used in the dataset and the *documentation* of popular visualization libraries, such as `mathplotlib`, `seaborn`, and `ggplot2`, we include the frequently-used plot types and parameters including the column to be plotted and the size/color/style of the visualization element. Additional grammar rules in $\mathcal{R}$ link these parameters with the corresponding plot type.

Based on the analysis of the plotting code fragments collected from the Jupyter Notebook dataset, we also include rules (in $\mathcal{R}$) that perform *data preprocessing operations*. For example, we observe that three commonly used typical patterns of data preprocessing are filtering, grouping and ordering; hence, we extend the grammar rules $\mathcal{R}$ to include rules that perform these steps.

3.4 Constrained Syntax-Guided Synthesis

Having defined the NL grammar and the visualization DSL, we next illustrate our main synthesis algorithm (Algorithm 1). The algorithms takes a grammar $\mathcal{G}$ = (terminal set $\mathcal{T}$, nonterminal set $\mathcal{N}$, production rules $\mathcal{R}$, start symbol `Plot`), and a pair $(\mathcal{R}_{must}, \mathcal{R}_{may})$ of must-use and may-use constraints (used for constraining derivations) as input; informally a derivation is application of a grammar rule toward generating the target program. It returns a program (generated by $\mathcal{G}$), generating which undergoes a derivation that satisfies $\mathcal{R}_{must}$ and minimizes the cost function.

In the visualization DSL, we begin with the start symbol `Plot` and perform a series of derivations to further extend to a complete program. More formally, a derivation is a sequence of terms that start with the start symbol `Plot` and each subsequent term is obtained from the previous term by applying a production rule in $\mathcal{G}$. If we use $P_1 \rightarrow_{r_1} P_2$ to denote that P_2 is derived from P_1 by applying r_1, then a derivation of P, denoted by d, can be written as

$$\texttt{Plot} \rightarrow_{r_0} P_1 \rightarrow_{r_1} P_2 \rightarrow_{r_2} \cdot\cdot P_k \rightarrow_{r_k} P$$

A program is *incomplete* if it contains a non-terminal symbol. A *complete* program contains only functions and terminal symbols.

Example 1. *The derivation for the first program shown in Section 2 is shown in parts below with some simplification:*

```
Plot → plt(Data, Mappings, PlotType, Legends)
Data →3 process(process(DataSet, Processor), Processor)
DataSet → "df"
Processor → filter(FilterColumn, FilterValue)
    →2 [filter("continent", "Europe"), group("country")]
Processor → group(GroupColumn) → group("country")
Mappings →2 list(Mapping, list(Mapping, nil))
Mapping → xaxis(XAxis) → x("date")
Mapping → yxis(YAxis) → y("total_confirmed")
PlotType → "Lineplot"
Legends → Labels → labels(XAxisLabel, YAxisLabel)
    →2 labels("date", "total_confirmed")6
```

The algorithm works by maintaining a worklist that consists of tuples (P, d, c), where P is a (potentially incomplete) program generated by derivation d whose cost is c. In each iteration, the algorithm works by picking an element (P, d, c) from the worklist. If the program P cannot be completed to a program that satisfies $\mathcal{R}_{must}$ (determined using a subroutine `feas`) or the current cost c is already more than the best cost found so far, we just prune this search branch and continue with the next iteration (Line 7). If not, then we further process this tuple (P, d, c). We first check whether P is already a complete program (Line 9), and if so, we update the best solution found so far and continue to the next iteration (Lines 10-12). If P is not complete, we apply all possible single-step rewrites to P and add new items to our worklist (Lines 15-17).

We next describe the subroutine $\texttt{feas}(d, \mathcal{R}_{must})$ that checks whether derivation d satisfies the constraint $\mathcal{R}_{must}$. If d is a complete derivation (generating a complete program), then $\texttt{feas}(d, \mathcal{R}_{must})$ returns "true" iff all rules in $\mathcal{R}_{must}$ are included in derivation d.

Since we aim to satisfy all constraints in $\mathcal{R}_{must}$ and as many constraints in $\mathcal{R}_{may}$ as possible, for each derivation we define its cost to be equal to the number of the production rules (used in this derivation) that do not belong to the set $\mathcal{R}_{may}$. In the definition

Algorithm 1: Branch-and-bound for cSyGuS.

Inputs : A CFG $\mathcal{G} := (\mathcal{N}, \mathcal{T}, \mathcal{R}, S)$, a pair $(\mathcal{R}_{must}, \mathcal{R}_{may})$ of constraints on $\mathcal{G}$

Output : A program P whose derivation d in $\mathcal{G}$ satisfies $\mathcal{R}_{must}$ and has minimum cost under $\texttt{cost}(d, \mathcal{R}_{may})$

```
1  P* ← Null ;                          // best program found so far
2  c* ← ∞ ;          // cost of the best program found so far
3  Q ← {(S, ⟨S⟩, 0)} ;                          // worklist queue
4  while Q ≠ {} do
5      (P, d, c) ← Remove an element from Q;
6      if feas(d, R_must) is false or c > c* then
7          continue
8      end
9      if P has no nonterminals then
10         P* ← P;
11         c* ← c;
12         continue
13     end
14     R ← all rules in R applicable on P;
15     foreach r ∈ R do
16         add (P', ⟨d, r, P'⟩, c') to Q where P →r P' and
             c' = c + cost_e(r|d, R_may)
17     end
18 end
19 return P*
```

of the cost function below, we use the notation $d|_i$ to denote the subderivation $(S, r_0, P_1, \cdot\cdot, r_{i-1}, P_i)$ of the derivation d consisting of the first i rule applications. If d has k rule applications, $d|_k = d$. Given the may-use constraint $\mathcal{R}_{may}$, the cost of a derivation d is defined as follows:

$$\texttt{cost}(d, \mathcal{R}_{may}) = \sum_{i=1}^{k} \texttt{cost}_e(r_i \mid d|_i, \mathcal{R}_{may}) \quad (1)$$

where the elementary cost function $\texttt{cost}_e$ is defined as

$$\texttt{cost}_e(r \mid d, \mathcal{R}_{may}) = \begin{cases} 0 & \text{if } r \in \mathcal{R}_{may} \\ 1 & \text{otherwise} \end{cases} \quad (2)$$

The cost of a derivation is simply the number of rules (in the derivation) that are not included in $\mathcal{R}_{may}$. Note that $\texttt{cost}(d, \mathcal{R}_{may}) = 0$ iff every production used in d lies in $\mathcal{R}_{may}$.

3.5 Extension to An Interactive System

We have implemented our NL2Viz approach with a supporting interactive tool. After NL2Viz synthesizes the first Python program and shows the generated plot, if the user is not satisfied with it, then the user can give another NL instruction to refine the plot. In the subsequent re-synthesis runs, NL2Viz uses additional information from the *program context* – the production rules used to generate the previous programs are included in the set $\mathcal{R}_{may}$ (as may-use production rules) to help the synthesizer prefer programs that are similar to the previously generated programs.

mpg	cylinders	displacement	horsepower	weight	acceleration	model_year	origin	name
12.0	8	400.0	167	4906	12.5	73	usa	ford country
23.8	4	151.0	85	2855	17.6	78	usa	oldsmobile starfire sx
26.0	4	79.0	67	1963	15.5	74	europe	volkswagen dasher
23.5	6	173.0	110	2725	12.6	81	usa	chevrolet citation
32.2	4	108.0	75	2265	15.2	80	japan	toyota corolla
26.4	4	140.0	88	2870	18.1	80	usa	ford fairmont
19.2	8	267.0	125	3605	15.0	79	usa	chevrolet malibu classic (sw)
17.5	6	250.0	110	3520	16.4	77	usa	chevrolet concours
31.0	4	76.0	52	1649	16.5	74	japan	toyota corona
15.0	6	250.0	72	3158	19.5	75	usa	ford maverick

Figure 7: The "Auto-MPG" dataset (sampled 10 rows)

4 EVALUATION

We implement NL2Viz as a Python package that registers a *magic ipython command* [12] `plot` in the popular Jupyter Notebook environment [13]. Hence, a user can input "`plot a histogram of cylinders`" to obtain an appropriate plot (see Figure 2). The Jupyter interface for NL2Viz also supports rudimentary auto-complete, suggesting column names, keywords, and pre-processing function names. The data for plotting is assumed to be in the form of a dataframe object from the widely used Pandas library [24]. We choose Matplotlib and Seaborn as the target plotting libraries for the generated code.

Our evaluation aims to answer four specific research questions:

- **RQ1: One-shot accuracy.** How accurately can NL2Viz produce the target plot from a single NL instruction? How effectively can the data/program context help NL2Viz resolve ambiguities in the user's NL instruction?
- **RQ2: Plot-and-change accuracy.** How accurately can NL2Viz create a new chart from an NL change instruction? How much does the user benefit from NL2Viz in this scenario in terms of the instruction length reduction?
- **RQ3: Comparison with the state of the art.** How does NL2Viz compare with other related state-of-the-art tools for visualization synthesis?
- **RQ4: Usability.** How usable and accurate is NL2Viz in a real setting?

Benchmark. We collect 303 NL instructions for 54 plots from two sources. **Auto-MPG plot descriptions.** In this source, there are 267 manually written NL instructions for 18 plots selected from online tutorials that use the "Auto-MPG" dataset[2], which contains technical specifications of 398 cars as shown in Figure 7. These NL instructions are provided by 15 professionals with experiences in data science. **Homework and COVID-19 assignments.** We also collect 36 scenarios from homework assignments and Jupyter Notebooks that use COVID-19 datasets in Github. In these scenarios, we use the problem statements as the NL instructions and the plots as the expected results. We exclude 5 instructions from these scenarios in which the plot types are not supported by NL2Viz.

NL2VIS dataset. Luo et al. [20] publish a NL2Visuaization dataset named NL2VIS. The NL2VIS dataset is generated by applying a neural network NL2SQL-to-NL2VIS model on a popular NL2SQL dataset named Spider [38]. Although the NL2VIS dataset has an impressive number of 25,750 (NL, VIS) pairs, we find that the dataset mainly focuses on the data preprocessing steps as most visualizations in the dataset are a direct presentation of the output by the SQL query from the Spider dataset without the plot options such as formats and legends. So we evaluate NL2Viz on the NL2VIS dataset in only RQ3 to compare with the results reported in their paper [20].

[2]https://www.kaggle.com/uciml/autompg-dataset

4.1 RQ1 Results: One-Shot Accuracy

Correctness. We categorize the output plots of NL2Viz into 4 separate categories based on how well it matches the ground truth:

- **Exact Match.** In this category, the output of NL2Viz exactly matches the ground truth.
- **Functionally Equivalent.** In this case, the output plot is functionally equivalent to the ground truth plot, but differs in a minor, often visual, detail. For example, if the instruction does not specify that a histogram should have normalized frequency on the y-axis, but the ground truth does use frequencies, NL2Viz produces a histogram using counts instead of frequencies. However, the two plots are functionally equivalent for most purposes. Therefore, we also consider this category to be correct.
- **Functionally Different.** In this case, the output plot differs from the ground truth in significant details. For example, the output plot has an axis plotted using a linear scale, while the ground truth uses a logarithmic scale.
- **No match.** Here, the produced plot is completely different from the ground truth. We differentiate "functionally different" from "no match" because in a "no match" case, often the time the semantic parser is not able to capture the intention from the NL instruction, while in a "functionally different" case, usually the synthesizer is unable to synthesize a reasonable program indicating that we should extract more may-use constraints with higher accuracy.

For the accuracy numbers, we consider the exact match and functionally equivalent categories to be correct, and functionally different and no match to be incorrect.

Results. The results of the evaluation are summarized in Table 1. Overall, NL2Viz is able to produce the correct output in 74.6% of the cases (with 58.6% being an exact match, and 15.9% of the cases being functionally equivalent). Of the remaining, 12.5% of the cases are functionally different.

To further analyze the results, we separate the Auto-MPG instances into two categories, *hard* and *easy*, based on whether the plot requires or not additional data preprocessing steps and additional parameters not in the input or dataframe. The idea to differentiate hard and easy plots is due to the observation that in an easy scenario, the synthesizer should be able to generate the correct program using only or mostly must-use constraints; while in a hard scenario, some information of the plot is often missing or vague in the NL instruction such that the synthesizer requires enough may-use constraints to generate the correct program. Of the 18 plots, 8 are categorized as easy and 10 as hard. As can be seen from the table, the system achieves an accuracy of 85.7% and

Category	#	Correct			Incorrect		
		em	fe	Tot.	fd	nm	Tot.
Total	295	173	47	220	37	38	75
Homework	31	19	2	21	5	5	10
Auto-MPG Total	264	154	45	199	32	33	65
Auto-MPG - Easy	119	86	16	102	15	2	17
Auto-MPG - Hard	145	68	29	97	17	31	48

Table 1: Accuracy of NL2Viz. The columns em, fe, fd, and nm denote exact match, functionally equivalent, functionally different, and no match, respectively.

Category	#	Correct			Incorrect		
		em	fe	Tot.	fd	nm	Tot.
Total	295	84	21	105	50	140	190
Homework	31	7	2	9	12	10	22
Auto-MPG Total	264	77	19	96	38	130	168
Auto-MPG - Easy	119	77	16	93	21	5	26
Auto-MPG - Hard	145	0	3	3	17	125	142

Table 2: Accuracy of the baseline approach (NL2Viz without context and data understanding)

70% in the easy and hard cases, respectively. We discuss the failure cases below.

Analysis of Failure Cases. For the Homework category, the main reason for failures is missing context – the homework assignments often contain relevant data in previous discussions or problems. In the two "no match" cases in the "Auto-MPG - Easy" category, the semantic parser interpretes the parameter representing point sizes with a group by column. For example, for the input *"scatter plot of mpg and acceleration with point size by cylinder"*, the semantic parser interprets cylinder as a group by column instead of a parameter for the point size. Most "functionally different" cases in both the "Auto-MPG-Easy" and "Auto-MPG-Hard" categories involve bin sizes in histograms. For example, for the input *"plot a bar graph that shows me the number of rows with MPG in range 5 to 10, 10 to 15, and so on"*, NL2Viz is unable to identify the bin sizes due to the limitations of its DSL grammar.

For the "Auto-MPG-Hard" category, the limitation of NL2Viz is in the parsing of the filtering or group by clauses. For example, the input fragment *yearly* or *annual* represents the "group by model_year" clause in the "Auto-MPG" dataset. However, NL2Viz is unable to do these translations in a fraction of the cases. As mentioned in Section 3.1, we choose to use a context free grammar to represent the NL instruction because the grammar would cover most cases. However, for words such as *yearly* or *annual* that are functionally equivalent to the column name *model_year*, we are unable to enumerate and cover all equivalent words in the grammar. Recent developments in the field of detecting equivalence via extrapolation [21] present a potential solution, and we believe that these scenarios can be correctly handled with better NL understanding.

Effect of Context and Data Understanding. We also run a baseline synthesizer whose results can be seen in Table 2. The baseline uses only NL information (extracted using the semantic parser described in Section 3.1), and does not use the data and program context. As expected, the code generated by the baseline approach in certain cases is incomplete. For example, in the instruction *"plot scatter average mpg by cylinder"*, there is an inherent group operator since it is required to calculate the aggregation function average for the "mpg" column. However, the "cylinder" column cannot both function as the column for the x-axis and group column, resulting in a missing group operation. Tables 1 and 2 show that using the data/program context substantially improves the results, especially in the "Auto-MPG-Hard" category. In particular, no cases of the "Auto-MPG-Hard" category can get exact match results because all the plots in this category require information from the data context.

Performance. We measure the performance of NL2Viz by the execution time. It turns out that NL2Viz is quite efficient, and we set a timeout bound to be 30 seconds. The average execution time for an instruction is around 3 seconds, and 95% of the instructions finish within 5 seconds. There are 3 instructions causing timeout, all of which are in the "Auto-MPG-Hard" category. They all have a length of more than 150 characters with the longest one being 340 characters, resulting in timeout in the semantic parsing phase.

4.2 RQ2 Results: Plot-and-Change Accuracy

This section evaluates NL2Viz with respect to RQ2, i.e., in the setting where NL2Viz is provided with an already existing plot and a change instruction. For example, the initial plot could have been generated by the instruction *"scatter plot of mpg and acceleration grouped by cylinders"* and the change instruction can be *"change to average mpg and accleration"*. The initial instruction generates an initial plot where each car is a single point colored based on the number of cylinders. The change instruction should generate a plot where each point corresponds to a group of cars with the same number of cylinders, with the coordinates given by average mpg and average acceleration.

Table 3 shows the information about the 40 tests. We separate the plots into "Easy" and "Hard" group based on whether it is needed contextual information to generate the plot. Further, we group the change instructions into *Replace* and *Add* categories: *Replace* change instructions replace the value of some plot aspect with another, while *Add* change instructions add a new aspect to a plot. We do not consider the delete category: in most practical scenarios, users start with a simple plot and add more complexity over time.

Accuracy Results. Table 3 shows the performance of NL2Viz on the change experiments. We find that NL2Viz performs well in general, achieving 67.5% accuracy (25 correct out of 40 total). Our tool is more effective in processing *Add* change instructions than *Replace* instructions. *Replace* instructions change an existing, correct aspect in the plot. Hence, NL2Viz needs to both locate the aspect to be replaced and parse the new aspect correctly. While in *Add* instructions, because the new aspect usually does not interfere with existing rules, NL2Viz just needs to add new aspect.

Instruction length results. We also evaluate how much instruction length is saved by the change instruction. For each initial instruction and change instruction pair, we also generate a *combined* instruction from which NL2Viz can produce the intended plot in one attempt. In terms of absolute numbers, we can see that

Plot Type	Change Type	Total	Correct	Avg. InitLen.	Avg. ChgLen.	Avg. FinLen.	Avg. Red.
Easy	Add	5	5	37.2	13.6	46.4	68%
	Replace	5	3	36.4	16.4	38.2	57%
Hard	Add	10	8	55.6	22.5	73.0	64%
	Replace	10	5	54.3	29.1	61.7	54%
	Add+Replace	10	4	57.2	37.0	69.9	49%

Table 3: Results of change experiments. The columns Avg. InitLen., Avg. ChgLen., Avg. FinLen. and Avg. Red. stand for Average Initial Instruction Length, Average Change Instruction Length, Average Final Instruction Length and Average Instruction Length Reduction in Correct cases respectively. Instruction length is in number of characters

the average length of *Replace* instruction is higher than *Add*, due to need of specifying both the component to replace and the replacement. On the other hand, the average *combined* instruction length is higher for *Add* instructions, due to the additional information added by the instruction.

We also measure the average instruction length reduction, given by $1 - \frac{Length(change\ instruction)}{Length(combined\ instruction)}$. We evaluate the setting where the user has already used the initial instruction to create a plot, and later wants to update it: either by using the change instruction, or directly using the *combined* instruction. We find that in general we achieve 58% instruction length reduction via the interaction enabled by the change instructions. We achieve higher reduction in *Add* category changes since the combined instruction text has to specify both the old and the new aspects, while the change instruction has to specify only the new aspects.

Analysis of Failure Cases. We note that *Add* change has a higher accuracy than *Replace* change due to that the aspect user wants to replace is often implicit in the change instruction. In the example in Section 2, the change instruction is "change to Asia". In this instruction, by the help of data context it is easy to infer that the aspect needs to be replaced is "Europe". However, in some cases, it's unclear whether the instruction is to change existing aspect or to add a new aspect. For example, when the initial instruction is "plot scatter of mpg versus model year", which would produce a scatter plot with each point representing a car and its mpg (y-axis) versus model year (x-axis). The change instruction is "change to average mpg for all cylinders". The idea of this change is to produce a final instruction which is "plot scatter of average mpg for all cylinders". The final plot is a scatter plot which x-axis represents cylinder number and y-axis represents average mpg. In this case, the "for all cylinders" in the instruction text represents a *Replace* change. However it can also be the case that the final instruction is "plot scatter of average mpg versus model year group by cylinders". In this case, the final plot a scatter plot which x-axis represents model year and y-axis represents average mpg with different cylinders have points with different colors on the plot.

4.3 RQ3 Results: Comparison with the State of the Art

Luo et al. [20] reports the accuracy of their neural-translation-model-based tool SEQ2VIS along with two other rule-based and semantic-parser-based tools DeepEye [19] and NL4DV [23] on a test set containing 3990 (NL, VIS) pairs from their NL2VIS dataset.

	DeepEye	NL4DV	SEQ2VIS	NL2VIZ
Easy	9.5%	11.5%	67.4%	83.9%
Medium	15.4%	22.5%	69.6%	74.2%
Hard	1.4%	7.6%	60.5%	41.5%
Extra Hard	6.1%	4.1%	61.8%	13.3%
Overall	9.1%	13.7%	65.7%	58.8%

Table 4: Comparison with the state of the art.

Due to the different target libraries of NL2Viz and NL2VIS dataset (Matplotlib vs. Vega-Lite), though it is possible to translate one to another, we find it not reasonable to directly compare the visualization program as different programs may lead to same or essentially same plots. Also we notice that in Luo's paper, they measure the "tree matching accuracy" to compare with other tools. The "tree matching accuracy" measures whether the flow of data transformation for each data column and its corresponding data shown on the axis is correst. Therefore for each test case, we need to manually examine whether our synthesized visualization program is equivalent to their program which is the labeled output. We are able to manually verify the 500 (NL, VIS) pairs sampled from their 3990-pairs test set. We treat our output to be correct if it's an Exact Match or Functionally Equivalent as defined in Section 4.1.

Table 4 shows our accuracy against other state of the art tools on the NL2VIS dataset. Out of the 500 pairs sampled, there are 32 visualizations that are currently not supported by NL2Viz. First we can see that NL2Viz outperforms two other rule-based tools by a large margin in all difficulty groups. The reason is that our tool supports the synthesis of data preprocessing steps while the other two tools don't or only have a very limited support. When compared to the neural-translation-model-based approach SEQ2VIS. We find that in the "Easy" and "Medium" category, our tool outperforms SEQ2VIS. The possible reason is that the "Easy" cases here are similar to the "Easy" cases in Section 4.1 and the "Medium" cases here are similar to the "Hard" cases, for both categories we see similar accuracy as in Section 4.1, which are 83.9% vs. 85.1%, 74.2% vs. 70%. In these two categories, our NL grammar and Visualization DSL can cover the target visualization program, therefore we are able to achieve higher accuracy than a learning-based approach by using a constrained syntax-guided synthesis. However, in the "Hard" and "Extra Hard" cases, the accuracy of NL2Viz declines drastically. The reason is that the NL instructions in these two categories are usually generated from nested SQL queries especially for the "Extra Hard" category. Our grammar and DSL don't cover

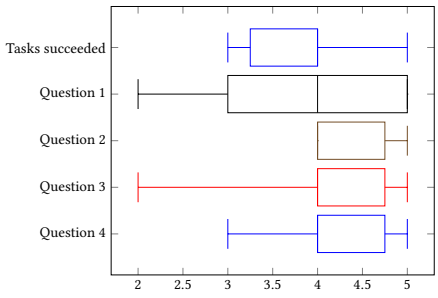


Figure 8: The distribution of succeeded tasks and scores among 6 participants.

such data processing steps, therefore it's impossible for NL2Viz to produce the correct visualization.

Overall, NL2Viz achieve lower but comparable accuracy compared with the SEQ2VIS tool while being able to achieve higher accuracy in "Easy" to "Medium" cases without the need for training data. NL2Viz also outperforms existing rule-based approaches by a large margin by leveraging the data/program context.

4.4 RQ4 Results: Usability and Interaction

To evaluate the usability of the system, we ask volunteers to complete 5 plotting tasks using NL2Viz, with an average of 7.4 years of experience in data science. Among the 5 plots in the tasks, there are 2 histogram/bar chart, 1 scatter plot and 2 line plots. Each participant is first asked multiple background questions, and is then given an explanation of the "Auto-mpg" dataset. Next we ask the participants to complete the 5 tasks in order until they are satisfied or are not willing to try NL2Viz any more. Each participant is given 20 minutes to finish all 5 tasks. Then, the participants are asked to rate NL2Viz on a scale of 1 (least positive) to 5 (most positive) on the following aspects:

- **Question 1** Do you find NL2Viz easy to use?
- **Question 2** Do you find it easy to interpret the code generated by NL2Viz and change for future usage?
- **Question 3** Do you want to use NL2Viz before you do visualization in future?
- **Question 4** Is NL2Viz able to understand your input?

On average, the participants successfully complete 4.13 out of 5 tasks. Figure 8 shows that the participants are generally in favour of NL2Viz, with an average of 4 for all questions. The high variance for question 3 is due to participants with higher expertise who are very familiar with writing visualization code strongly not preferring to use NL2Viz.

Suggestions from the participants highlight two major issues. Three participants (S2, S4, S6) suggest that NL2Viz should have built-in "highlighting" to show which part of their natural language corresponds to which part of the generated code. This would help users change their input when NL2Viz misinterprets their intention. Another suggestion is to modify NL2Viz to produce multiple candidate plots for the same input (S1, S2, S3). It is fairly straightforward to modify our semantic parser and synthesizer to produce multiple candidate, and we intend to make this modification. Other suggestions include allowing for click selecting columns as an input modality (S3), displaying a confidence score for the generated plot (S6), and a separate cleaning step before visualization (S5).

5 RELATED WORK

Natural Language to Visualization The idea of using natural language (NL) as a query interface for visualization is getting popular as the development in NL2Code.

Tong et al. [9] presents, DataTone, a mixed-initiative approach to address the ambiguity problem in NL interfaces for visualization. Unfortunately, because DataTone is not publicly available, we could not perform a direct comparison with NL2Viz. Zhang et al [7] proposes Text-to-Viz. The usage scenario of Text-to-Viz is quite different from ours. Text-to-Viz is a visualization recommendation tool that focuses on data exploration. It does not support precise NL instructions to a specific visualization. Instead, the user's input works as a guide to explore charting options on certain columns or combination of columns. We find it not fair to compare Text-to-Viz's accuracy on our dataset as it is not designed to produce visualization with the NL instruction provided. Similarly, Sun et al. [32] proposes Articulate, a two-step process to generate visualization from NL instructions. First, it parses the NL instruction into commands using supervised learning. It then generates visualizations for the commands using heuristics. Articulate is also focused on data exploration instead of synthesizing precise visualization according to the NL instruction input.

Narechania et al. [23] proposes NL4DV, which has similar functionality as NL2Viz. It is alsointegrated into the Jupyter Notebook environment while producing the results in Vega-Lite format [28]. However, NL4DV relies only on the NL instruction to generate the visualization. It only checks the data to identify the database entities in the NL instruction without leveraging other contextual information from data/program. Similarly, it also lacks the ability to create the necessary data preprocessing steps. Luo et al. [20] publishs a public dataset NL2VIS consisting of 25750 (NL, VIS) pairs. They propose a NL2SQL-to-NL2VIS model to translate the (NL, SQL) pairs in the popular spider [38] dataset to the (NL, VIS) pairs. In this paper, they also propose a learning-based approach SEQ2VIS based on SEQ2SEQ model [33] used in NL2SQL tasks. They evaluate their approach on the dataset comparing with the other two approaches NL4DV and DeepEye [19], which is a keyword-based approach previously proposed by them too. They find their approach largely outperforms the other two approaches. However, since the spider dataset is a NL2SQL dataset. It only focuses on how the output is calculated using the data transformations defined in the SQL query. The NL instructions in NL2VIS dataset completely ignore important options of visualizations like formats and legends. Despite the limitations, we also evaluate NL2Viz on this dataset in Section 4.3.

It's worth noting that unlike NL2Viz, the above systems don't support further refinement on the result, which limits the ability of users to further modify or reuse the results later in other tasks.

Rong et al. [27] proposes CodeMend, which uses neural network to infer the correspondence between NL query and function/parameter in visualization program. Similarly, Setlur et al. [29] proposes Evizas which allows users to refine existing visualizations by asking questions or direct manipulation. However, both tools lack the ability to generate complete visualization code and also

cannot generate the necessary data preprocessing steps. They can be seen as complementary with NL2Viz. Their approaches can be combined with the interactive approach used by NL2Viz to provide better user experience after the first-shot query.

Visualization Recommendation Visualization recommendation focuses on producing the recommended visualization encoding based on design domain knowledge [35].

Dominik et al. [22] presents Draco, which represents a visualization as a set of logical facts and thus converts visualization design patterns into a set of constraints. It then uses constraint solving to recommend the best visualization scheme based on the collection of domain knowledge. Ding et al. [8] presents QuickInsights to discover interesting patterns from multi-dimensional datasets by formalizing the notion of interesting pattern (insights) and present them as visualizations. Siddiqui et al. [30, 31] proposes an interactive visual analytic platform Zenvisage to find desired visual patterns from large datesets. It extends the previous work Vispedia proposed by Chan et al. [6] which only performs a keyword-query of collected graphs.

While the output of our tool is also a visualization, the focus is different. Visualization recommendation tries to follow visualization design patterns. We focus on eliminating the ambiguity in natural language instructions by bringing insights from data. Our approach is also extensible and can be integrated with existing visualization recommendation tools.

Syntax Guided Synthesis The constrained syntax-guided synthesis problem is an extension of the syntax-guided synthesis (SyGuS) problem first introduced by Alur el al [1]. Our cSyGuS problem asks for a program that is not only generated by a given grammar, but also uses specific rules and non-terminals of the grammar. Successful solution strategies for SyGuS are based on bottom-up enumeration [2, 3, 34], model based quantifier instantiation [26], and top-down search over the grammar [17]. Hu et al [11] considers QSyGuS, a variant of the SyGuS problem where a cost model given by a weighted tree automaton assigns costs to programs, and task is to generate the minimal cost program that satisfies the semantic constraint. The solution they used is however infeasible in our setting due to the the presence of derivation constraints

6 CONCLUSION

In this paper, we have presented a novel NL2Visualization approach named NL2Viz and its supporting tool for automatically synthesizing visualization programs from a user's NL instruction. The key idea underlying our approach is to leverage not only the NL instruction, but also the other contexual information (namely data context and program context) and then convert the different kinds of specifications provided by the user into *symbolic constraints*, which can be used to generate the desired visualization programs using syntax-guided program synthesis. Moreover, NL2Viz includes an interactive interface for allowing the user to further refine and reuse the resulting visualization. We evaluate our approach on a real-world visualization benchmark and a public dataset to show the effectiveness of NL2Viz. We also perform a user study involving 6 data scientist professionals to demonstrate the usability of NL2Viz, the readability of the generated code, and NL2Viz's effectiveness in helping users generate desired visualizations effectively and efficiently.

ACKNOWLEDGMENTS

This work was partially supported by National Natural Science Foundation of China under Grant No.: 62161146003.

REFERENCES

[1] Rajeev Alur, Rastislav Bodík, Eric Dallal, Dana Fisman, Pranav Garg, Garvit Juniwal, Hadas Kress-Gazit, P. Madhusudan, Milo M. K. Martin, Mukund Raghothaman, Shambwaditya Saha, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2015. *Syntax-guided synthesis.* https://doi.org/10.3233/978-1-61499-495-4-1

[2] Rajeev Alur, Pavol Cerný, and Arjun Radhakrishna. 2015. Synthesis through unification. In *Proceedings the 27th International Conference on Computer Aided Verification* (San Francisco, CA, USA) *(CAV '15)*. Springer, Cham, 163–179. https://doi.org/10.1007/978-3-319-21668-3_10

[3] Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. 2017. Scaling enumerative program synthesis via divide and conquer. In *Proceedings of the 23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Uppsala, Sweden) *(TACAS '17)*. 319–336. https://doi.org/10.1007/978-3-662-54577-5_18

[4] Yoav Artzi. 2016. Cornell SPF: Cornell semantic parsing framework. https://doi.org/10.48550/arXiv.1311.3011 arXiv:arXiv:1311.3011

[5] Jonathan Berant, Andrew Chou, Roy Frostig, and Percy Liang. 2013. Semantic parsing on Freebase from question-Answer pairs. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing* (Seattle, WA, USA) *(EMNLP '13)*. ACL, 1533–1544. https://aclanthology.org/D13-1160

[6] Bryan Chan, Justin Talbot, Leslie Wu, Nathan Sakunkoo, Mike Cammarano, and Pat Hanrahan. 2009. Vispedia: On-demand data integration for interactive visualization and exploration. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data* (Providence, Rhode Island, USA) *(SIGMOD '09)*. ACM, New York, NY, USA, 1139–1142. https://doi.org/10.1145/1559845.1560003

[7] Weiwei Cui, Xiaoyu Zhang, Yun Wang, He Huang, Bei Chen, Lei Fang, Haidong Zhang, Jian-Guan Lou, and Dongmei Zhang. 2019. Text-to-Viz: Automatic generation of infographics from proportion-related natural language statements. *IEEE transactions on visualization and computer graphics* 26, 1 (2019), 906–916. https://doi.org/10.1109/tvcg.2019.2934785

[8] Rui Ding, Shi Han, Yong Xu, Haidong Zhang, and Dongmei Zhang. 2019. QuickInsights: Quick and automatic discovery of insights from multi-dimensional data. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. ACM, New York, NY, USA, 317–332. https://doi.org/10.1145/3299869.3314037

[9] Tong Gao, Mira Dontcheva, Eytan Adar, Zhicheng Liu, and Karrie G. Karahalios. 2015. DataTone: Managing ambiguity in natural language interfaces for data visualization. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology* (Charlotte, NC, USA) *(UIST '15)*. ACM, New York, NY, USA, 489–500. https://doi.org/10.1145/2807442.2807478

[10] Lars Grammel, Melanie Tory, and Margaret-Anne Storey. 2010. How information visualization novices construct visualizations. *IEEE transactions on visualization and computer graphics* 16, 6 (2010), 943–952. https://doi.org/10.1109/TVCG.2010.164

[11] Qinheping Hu and Loris D'Antoni. 2018. Syntax-guided synthesis with quantitative syntactic objectives. In *Proceedings the 30th International Conference on Computer Aided Verification* (Oxford, UK) *(CAV '15)*. Springer, Cham, 386–403. https://doi.org/10.1007/978-3-319-96145-3_21

[12] IPython. 2020. IPython magic commands. https://ipython.readthedocs.io/en/stable/interactive/magics.html. Accessed: 2020-05-15.

[13] Jupyter. 2020. Project Jupyter. https://jupyter.org/. Accessed: 2020-05-15.

[14] Kaggle. 2020. Kaggle competitions. https://www.kaggle.com/competitions. Accessed: 2020-05-15.

[15] Tadao Kasami. 1966. An efficient recognition and syntax-analysis algorithm for context-free languages. *Coordinated Science Laboratory Report no. R-257* (1966).

[16] Hyeonji Kim, Byeong-Hoon So, Wook-Shin Han, and Hongrae Lee. 2020. Natural language to SQL: Where are we today? *Proceedings of the Very Large Data Base Endowment.* 13, 10, 1737–1750. https://doi.org/10.14778/3401960.3401970

[17] Woosuk Lee, Kihong Heo, Rajeev Alur, and Mayur Naik. 2018. Accelerating search-based program synthesis using learned probabilistic models. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI '18)*. ACM, New York, NY, USA, 436–449. https://doi.org/10.1145/3192366.3192410

[18] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago,

et al. 2022. Competition-level code generation with alphacode. *arXiv preprint arXiv:2203.07814* (2022). https://doi.org/10.48550/arXiv.2203.07814

[19] Yuyu Luo, Xuedi Qin, Nan Tang, Guoliang Li, and Xinran Wang. 2018. DeepEye: Creating good data visualizations by keyword search. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. ACM, New York, NY, USA, 1733–1736. https://doi.org/10.1145/3183713.3193545

[20] Yuyu Luo, Nan Tang, Guoliang Li, Chengliang Chai, Wenbo Li, and Xuedi Qin. 2021. Synthesizing natural language to visualization (NL2VIS) benchmarks from NL2SQL benchmarks. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. ACM, New York, NY, USA, 1235–1247. https://doi.org/10.1145/3448016.3457261

[21] Jeff Mitchell, Pontus Stenetorp, Pasquale Minervini, and Sebastian Riedel. 2018. Extrapolation in NLP. In *Proceedings of the Workshop on Generalization in the Age of Deep Learning* (New Orleans, LA, USA). ACL, 28–33. https://doi.org/10.18653/v1/W18-1005

[22] Dominik Moritz, Chenglong Wang, Greg L Nelson, Halden Lin, Adam M Smith, Bill Howe, and Jeffrey Heer. 2018. Formalizing visualization design knowledge as constraints: Actionable and extensible models in Draco. *IEEE transactions on visualization and computer graphics* 25, 1 (2018), 438–448. https://doi.org/10.1109/TVCG.2018.2865240

[23] Arpit Narechania, Arjun Srinivasan, and John Stasko. 2020. NL4DV: A toolkit for generating analytic specifications for data visualization from natural language queries. *IEEE Transactions on Visualization and Computer Graphics* 27, 2, 369–379. https://doi.org/10.1109/tvcg.2020.3030378

[24] Pandas. 2019. pandas: Python data analysis library. https://pandas.pydata.org/. Accessed: 2019-11-20.

[25] João Felipe Pimentel, Leonardo Murta, Vanessa Braganholo, and Juliana Freire. 2019. A large-scale study about quality and reproducibility of jupyter notebooks. In *Proceedings of the 16th International Conference on Mining Software Repositories*. IEEE Press, 507–517. https://doi.org/10.1109/MSR.2019.00077

[26] Andrew Reynolds, Morgan Deters, Viktor Kuncak, Cesare Tinelli, and Clark W. Barrett. 2015. Counterexample-guided quantifier instantiation for synthesis in SMT. In *Proceedings the 27th International Conference on Computer Aided Verification* (San Francisco, CA, USA) *(CAV '15)*. Springer, Cham, 198–216. https://doi.org/10.1007/978-3-319-21668-3_12

[27] Xin Rong, Shiyan Yan, Stephen Oney, Mira Dontcheva, and Eytan Adar. 2016. CodeMend: Assisting interactive programming with bimodal embedding. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology* (Tokyo, Japan) *(UIST '16)*. ACM, New York, NY, USA, 247–258. https://doi.org/10.1145/2984511.2984544

[28] Arvind Satyanarayan, Dominik Moritz, Kanit Wongsuphasawat, and Jeffrey Heer. 2016. Vega-lite: A grammar of interactive graphics. *IEEE transactions on visualization and computer graphics* 23, 1 (2016), 341–350. https://doi.org/10.1109/tvcg.2016.2599030

[29] Vidya Setlur, Sarah E. Battersby, Melanie Tory, Rich Gossweiler, and Angel X. Chang. 2016. Eviza: A natural language interface for visual analysis. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology* (Tokyo, Japan) *(UIST '16)*. ACM, New York, NY, USA, 365–377. https://doi.org/10.1145/2984511.2984588

[30] Tarique Siddiqui, Albert Kim, John Lee, Karrie Karahalios, and Aditya Parameswaran. 2016. Effortless data exploration with Zenvisage: An expressive and interactive visual analytics system. *Proceedings of the Very Large Data Base Endowment.* 10, 4, 457–468. https://doi.org/10.14778/3025111.3025126

[31] Tarique Siddiqui, John Lee, Albert Kim, Edward Xue, Xiaofo Yu, Sean Zou, Lijin Guo, Changfeng Liu, Chaoran Wang, Karrie Karahalios, and Aditya G. Parameswaran. 2017. Fast-forwarding to desired visualizations with zenvisage. In *8th Biennial Conference on Innovative Data Systems Research, 2017, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2017/papers/p43-siddiqui-cidr17.pdf

[32] Yiwen Sun, Jason Leigh, Andrew Johnson, and Sangyoon Lee. 2010. Articulate: A semi-automated model for translating natural language queries into meaningful visualizations. In *Proceedings of the 10th International Conference on Smart Graphics*. Springer-Verlag, 184–195. https://doi.org/10.1007/978-3-642-13544-6_18

[33] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. 2014. Sequence to sequence learning with neural networks. In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2* (Montreal, Canada) *(NIPS '14)*. MIT Press, Cambridge, MA, USA, 3104–3112. https://doi.org/10.5555/2969033.2969173

[34] Abhishek Udupa, Arun Raghavan, Jyotirmoy V. Deshmukh, Sela Mador-Haim, Milo M.K. Martin, and Rajeev Alur. 2013. TRANSIT: Specifying protocols with concolic snippets. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. ACM, New York, NY, USA, 287–296. https://doi.org/10.1145/2491956.2462174

[35] Kanit Wongsuphasawat, Dominik Moritz, Anushka Anand, Jock Mackinlay, Bill Howe, and Jeffrey Heer. 2016. Towards a general-purpose query language for visualization recommendation. In *Proceedings of the Workshop on Human-In-the-Loop Data Analytics* (San Francisco, California) *(HILDA '16)*. ACM, New York, NY, USA, Article 4, 6 pages. https://doi.org/10.1145/2939502.2939506

[36] Frank F. Xu, Zhengbao Jiang, Pengcheng Yin, Bogdan Vasilescu, and Graham Neubig. 2020. Incorporating external knowledge through pre-training for natural language to code generation. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics* (Virtual Event) *(ACL '20)*. ACL. https://doi.org/10.18653/v1/2020.acl-main.538

[37] Navid Yaghmazadeh, Yuepeng Wang, Isil Dillig, and Thomas Dillig. 2017. SQLizer: Query synthesis from natural language. *Proceedings of the ACM Programming Language* 1, OOPSLA, Article 63 (oct 2017), 26 pages. https://doi.org/10.1145/3133887

[38] Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev. 2018. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL task. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing* (Brussels, Belgium) *(EMNLP '18)*. ACL. https://doi.org/10.18653/v1/d18-1425

[39] Victor Zhong, Caiming Xiong, and Richard Socher. 2017. Seq2SQL: Generating structured queries from natural language using reinforcement learning. *arXiv preprint arXiv:1709.00103* (2017). https://doi.org/10.48550/arXiv.1709.00103