TABLECODER: Table Extraction from Text via Reliable Code Generation

Haoyu Dong, Yue Hu, Huailiang Peng*, Yanan Cao
Institute of Information Engineering, Chinese Academy of Sciences
School of Cyber Security, University of Chinese Academy of Sciences

Abstract

This paper introduces a task aimed at extracting structured tables from text using natural language (NL) instructions. We present TABLECODER, an approach that leverages the symbolic nature of code to enhance the robustness of table structure construction and content extraction. TABLECODER first generates Python classes or SQL statements to explicitly construct table structures, capturing semantic ontology, computational dependencies, numerical properties, and format strings. This approach reliably mitigates issues such as structural errors, erroneous computations, and mismatched value types. Subsequently, TABLECODER proposes grounded content extraction, populating table cells sequentially and maintaining the exact order in which they are mentioned in the source text. By simulating a grounded "translation" from text to code, this method reduces the likelihood of omissions and hallucinations.

Experimental results demonstrate that TABLECODER significantly improves F1 scores and mitigates hallucination and computational errors, crucial for high-stakes applications like government data analytics and financial compliance reporting. Moreover, the code-generation-based method naturally integrates with standard SQL databases and Python workflows, ensuring seamless deployment in existing enterprise data pipelines.

User Instruction 1: Please extract a flat table from the following text about the top three most populous islands, including details such as "Ranking," "Island," "Population," "Area," "Density (/km^2)," "Country," and "Capital".

The top three most populous islands in the world are Java in Indonesia, Honshū in Japan, and Great Britain.

Java has an impressive population count of 148,756,685 and is home to Jakarta, the capital of its country. Its land spans 124,378 square kilometers, which equates to a dense population of 1,196 people per square kilometer.

Trailing behind Java is Honshū, Japan's largest island, home to 102,579,606 individuals. Honshū boasts a vast area of 227,954 square kilometers, noticeably larger than Java.

Answer by Assistant:

Ranking	Island	Population	Area	Density (/km2)	Country	Capital
1	Java	148.8 million	124,378 km2	1,196	Indonesia	Jakarta
2	Honshū	102.6 million	227,954 km^2	450	Japan	
3	Great Britain					

User Instruction 2: What information can be extracted from the text regarding the number of postgraduate degrees awarded in fields of Science and Engineering, specifically focusing on "Total", "Master's" ("All" and "Percent"), and "Doctoral" ("All" and "Percent")? Please organize it in a hierarchical table.

In the field of Science, Master's degrees are predominant, with 229,169 graduates, representing 55% of Science degrees. On the other hand, Doctoral degrees have 186,399 degrees awarded.

Shifting focus to Engineering, within this discipline, Master's degrees prevail, with 96,756 recipients accounting for 58% of all Engineering degrees. Meanwhile, Doctoral degrees hold a total of 68,825 degrees awarded.

Answer by Assistant:

Broad fields	Total	Master's		Doctoral	
		All	Percent	All	Percent
Science	415,568	229,169	55%	186,399	45%
Engineering	165,581	96,756	58%	68,825	42%

Figure 1: Examples of NL-TO-TABLE. Table schemas are flexibly defined by user instructions. Cells necessitating computation are highlighted in red.

1 Introduction

Structured table extraction from unstructured text is critical for automating data processing tasks across industries such as finance, government, and healthcare, where accuracy and reliability are paramount. As illustrated in Figure 1, relational tables enable automated processing and analysis through tools like SQL or Pandas, whereas hierarchical tables (Cheng et al., 2022a) intuitively present complex statistical data in government or financial reports. Considering that table extraction naturally involves diverse demands regarding "what information to extract" and "how to structure it," controllable table extraction tailored by NL user instructions is highly desirable for real-world deployments.

Pioneering works (Wu et al., 2022; Li et al., 2023b; Pietruszka et al., 2022; Jiao et al., 2023; Jain et al., 2024; Tang et al., 2023) have extracted tables from text. However, they neglect user intent and fail to tailor table structures for users, resulting in key-value pairs or simple relational tuples. Additionally, Reversing a "table-to-text" dataset to construct a "text-to-table" dataset may results in data quality issues. It includes excessive, missing, or unextractable cells, such as extracting "127,955 million" from text stating roughly "128.0 billion".

To address these challenges, we introduce NL-TO-TABLE, a human-labeled dataset for table extraction following NL instructions. Key features include: (1) We include a rigorous quality-control pipeline where human annotators carefully address issues like excessive, missing, or unextractable cells to guarantee dataset quality. (2) We perform fine-grained anno-

* Corresponding author

Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 6: Industry Track), pages 1399–1412
July 28-30, 2025

tations on ontology trees for semantic relationships, formulas for computational dependencies, and units and feasible ranges for numerical values. (3) NL-TO-TABLE introduces numerical reasoning as a key aspect of table extraction, which is in high demand in the financial and government domains, as illustrated in Figure 1—with red highlights. (4) Due to the equivalence of identical quantities expressed in various formats (Jiao et al., 2023), we annotate number format strings to facilitate automatic evaluation.

SQL and Python provide a robust framework for generating structured data, so we propose TABLECODER, a novel method to generate code that unravels the complexities involved in structure construction, data extraction, numerical computation, and number format representation. (1) TABLECODER employs Python classes and SQL `CREATE` statements to construct a comprehensive table structure with ontology trees, computational relationships, number units, feasible ranges, and number format strings. It facilitates a symbolic and reliable extraction process by defining cell placement, type and range validation, and automatic number computation. (2) TABLECODER extracts table contents in the order they appear in the source text, emulating a step-by-step "translation" from text to code to minimize omissions and hallucinations often caused by LLMs.

Existing automatic evaluation methods are challenged by different format expressions of the same content. To address this, we propose the Format Agnostic Evaluation (FORMATAGNOSTIC-EVAL) for automatic evaluation of table extraction. Experimental results show that FORMATAGNOSTIC-EVAL improves existing metrics, making them much closer to human evaluators' assessments. Notably, fine-tuned LLaMA-70B with the NL-TO-TABLE dataset remarkably mitigates hallucination and computational errors, outperforming few-shot GPT-4 by 11.4% to 19.2%, and fine-tuned Mistral-7B even outperforms GPT-4 by 5.7% to 12.3%.

We wrapped up TABLECODER as an API and deployed it on a server, enabling the storage of extracted tables using `openpyxl` [1].

2 Preliminaries

2.1 Task Formulation

The task is to extract a table from unstructured text, given human utterance to specify the table structure. The purpose of providing NL instructions as inputs is to meet specific and diverse user requirements concerning the structure of tables. Importantly, conditioning on NL instructions significantly reduces evaluation ambiguity associated with various potential structures (Jiao et al., 2023), such as opting for dual columns labeled "First name" and "Last name" as opposed to an alternative single "Name" column.

2.2 Semantic and Computational Relationships

Semantic Relationship Semantic relationships can be explicit hierarchies that are indicated by specific formats, such as merged cells (Wang et al., 2021; Cheng et al., 2022a), or implicit functional dependencies (Nan et al., 2020), as seen in the first example of Figure 1. Following both the explicit hierarchy (Cheng et al., 2022a) and implicit ontology (Nan et al., 2020), we identify the parent of each column header to construct a tree-structured ontology for each table, as illustrated in Figure 4 in the Appendix.

Computational Relationship A column may be derived from other columns via computations, as highlighted in Figure 4. They are implicit and require human reasoning, while only spreadsheets may have explicit formulas.

2.3 Python and SQL for Table Generation

Existing works on LLMs for tabular data commonly use Markdown, HTML, LaTeX, or variants to encode tables, which are studied by (Singha et al., 2023; Sui et al., 2024). In this paper, we propose to leverage code for table generation.

SQL provides a robust framework for generating structured data. By using `CREATE` statements, users define tables with explicit schemas, ensuring that data is consistently structured and easy to query. This is crucial for LLM-based generation, which can produce corrupted tables with row or column misalignment. `INSERT` and `UPDATE` operations can add new data to existing tables in arbitrary orders without disrupting the overall structure. This kind of incremental data generation is essential for keeping the extracted table integral and up-to-date when processing long and complex unstructured text.

On the other hand, SQL's common practices may limit its flexibility for hierarchical tables. Python's inherent object-oriented paradigm is able to encode complex structured tables, and it facilitates automated data computations, e.g., `_update_density` in Program 1. However, despite LLMs' proficiency in Python (Li et al., 2023a), they are not fully proficient in generating hierarchical tables.

[1] https://openpyxl.readthedocs.io/en/stable/

Table 1: Dataset statistics of NL-TO-TABLE.

Labeled Data	Wikipedia	Statistical Reports
# User instructions	5,241	836
# Tokens in instruction	60.2	67.5
# Tables	5,241	836
# Mentioned columns	26,501	3,475
# Mentioned rows	38,572	2,510
# Mentioned cells	60,779	4,115
# Sentences in Text	31,802	3,012
% Complex ontology trees	48.1%	100.0%
% Number format cells	24.4%	74.5%
% Computed cells	1.9%	7.8%

2.4 Evaluation Metrics

We use Exact Match (EM), BERTScore (BERT), and Chrf metrics (Wu et al., 2022) to assess F1 scores, as detailed in Appendix B. But they are challenged by the flexible and equivalent formatting rules found in tables, e.g., "1.4 thousand dollars" and "$1,400", so we annotate the format string for each column consisting of quantities, e.g., f"{self.total:,.1f} thousand dollars". Thus, during the evaluation phase, we format quantities using the human-labeled format strings before comparing them with ground truth contents, enabling FORMATAGNOSTIC-EVAL. To cover all variations of format strings in our dataset, we first collected 58 built-in formats from Excel under categories like "Number," "Currency," "Accounting," "Date," "Percentage," etc. In addition, we labeled another 84 format strings that appeared in our dataset and produced 142 strings.

3 NL-TO-TABLE

We construct NL-TO-TABLE from Wikipedia articles (ToTTo (Parikh et al., 2020)) and statistical reports (HiTab (Cheng et al., 2022a)). Each dataset is rich in tables accompanied by corresponding textual descriptions, with highlighted cells linked to descriptive sentences. We only include tables that have at least four sentences and four mentioned cells. There are 5,241 tables from Wikipedia and 836 tables from statistical reports. Together, the two datasets present a comprehensive collection that spans various table structures.

We have designed a six-step annotation process to construct the first human-labeled dataset for generally structured table extraction following NL instructions, comprising a substantial amount of complex reasoning and fine-grained structure annotations, detailed in Appendix A.

As Table 1 shows, 48.1% of Wikipedia tables and 100.0% of statistical tables feature ontology trees with more than two layers. A significant portion (74.5%) of cells in statistical reports are quantities, and computed cells account for 7.8%, encompassing various types, including SUM (45.2%), AVG (5.6%), DIV (21.9%), DIFF (15.6%), and ADD (5.4%).

4 TABLECODER

Existing approaches commonly use Markdown, HTML, or their variants to encode tables for LLMs (Singha et al., 2023; Sui et al., 2024; Dong and Wang, 2024), as well as efficient JSON encoding (Dong et al., 2024). Unfortunately, when the task is table generation, they may produce structural corruption, row or column misalignment, erroneous value computation, missing or excessive information, etc. As depicted in Figure 2, TABLECODER first uses SQL or Python code to construct the table structure. It then extracts table contents following the order in the input text.

4.1 Symbolic Structure Construction

TABLECODER leverages LLMs to generate code to build the table, so that the generated results are ensured to be well structured, and inherent semantic/computational column relationships are explicitly reflected. Additionally, type constraints and computational dependencies can also be predefined to avoid obvious errors and inconsistent units in the following content extraction phase.

4.1.1 Type and range constraints

In SQL, value type and range constraints are well supported through `CREATE`, which is quite concise and useful. As shown in Program 2, properties of the column "Ranking" can be simply specified using "INT CHECK (Ranking > 0)".

4.1.2 Semantic dependencies

In SQL, we use the column corresponding to the root of the ontology tree as the primary key, with other columns as attributes. For tables featuring hierarchical ontology trees, generating multiple tables with SQL represents a promising direction for future work. Instead, we leverage Python's flexible object-oriented paradigm to encode both flat and hierarchical ontology trees in a unified manner. As shown in Program 1, we define classes for all parent nodes in the ontology tree, with dependencies established among multiple classes.

4.1.3 Computational dependencies

LLMs have difficulty reliably calculating numbers without an explicit executor (Gao et al., 2023b; Chen et al., 2022; Zhou et al., 2022). Fortunately, Program 1 showcases an example that "_update_density"

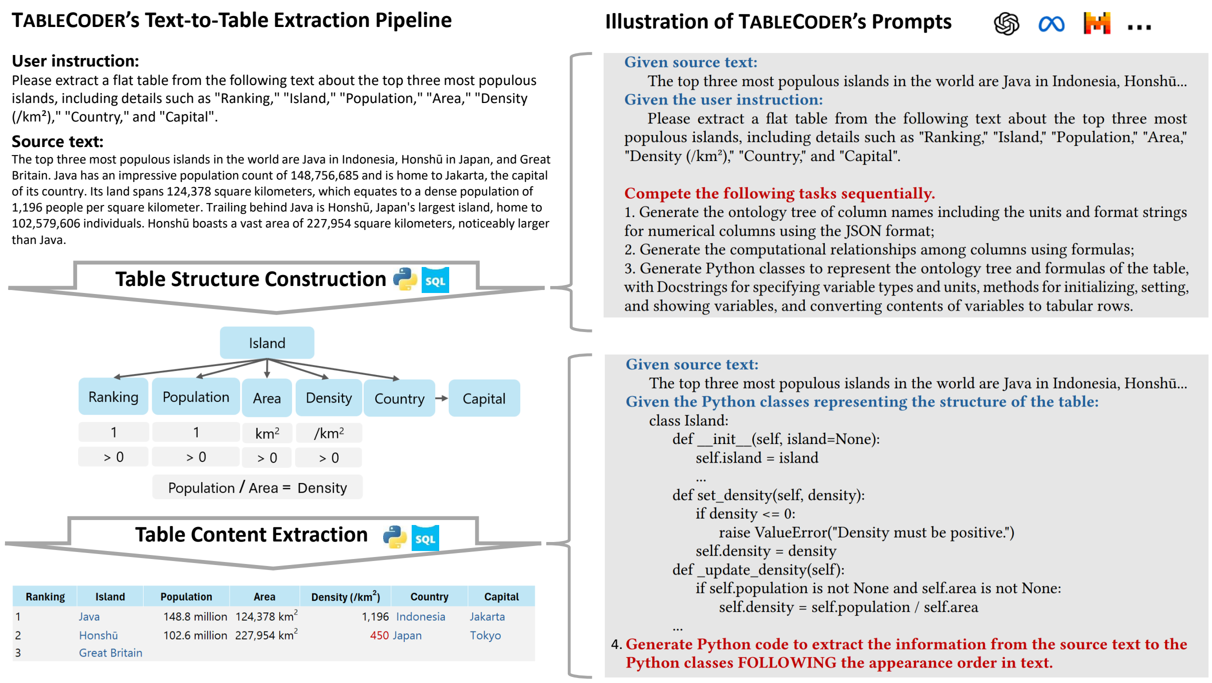

Figure 2: Architecture of TABLECODER. The left side illustrates a chain-of-thought pipeline of table extraction. The right side illustrates the prompt for LLMs, which is streamlined with four steps in a single run.

in TABLECODER automatically triggers a symbolic execution when "area" and "population" are set with values. As long as inherent computational relationships are discovered, TABLECODER generates methods like "_update_density" to ensure all derived cells are accurately calculated. This mechanism can also be implemented in SQL through `TRIGGER`.

4.1.4 Format application

Program 1 shows the example of the Python implementation. A "convert_to_tabular_row" method serializes each instance to a tabular row following user-specified column orders. Note that format strings are replaced with ground truth format strings during our format-agnostic evaluation.

```
class Country:
    ...
class Island:
    def __init__(self, island=None):
        ...
        self.area = None
        self.density = None
        self.country = Country()
    ...
    def set_density(self, density):
        if density <= 0:
            raise ValueError("density must be positive.")
        self.density = density
    def _update_density(self):
        if self.population is not None and self.area is not
             None:
            self.density = self.population / self.area
    def convert_to_tabular_row(self):
        return [
            self.show_ranking(), self.show_island(),
            f"{self.show_population() / 1_000_000:,.1f}
                 million" if self.show_population() else None
                 , ...
        ]
```

Program 1: Python for structure construction.

```
CREATE TABLE islands (
    Ranking INT CHECK (Ranking > 0),
    Island VARCHAR(255),
    Population BIGINT CHECK (Population > 0),
    Area BIGINT CHECK (Area > 0),
    Density DECIMAL(10, 2) CHECK (Density > 0),
    Country VARCHAR(255),
    Capital VARCHAR(255)
);

CREATE TRIGGER compute_density
BEFORE INSERT OR UPDATE ON islands

FOR EACH ROW
BEGIN
    IF NEW.Population IS NOT NULL AND NEW.Area IS NOT NULL
        THEN
        SET NEW.Density = NEW.Population / NEW.Area;
    END IF;
END;
```

Program 2: SQL for structure construction.

4.2 Grounded Content Extraction

Based on the constructed table structure, TABLECODER generates SQL statements or instantiates Python classes to establish infilling of tabular data.

As demonstrated in Figure 2, previous works sequentially generate "450" and the country name "Japan" due to their adjacency in the table's surface-level presentation (Wu et al., 2022; Li et al., 2023b; Pietruszka et al., 2022). However, these elements are significantly distant in the source text, appearing in the first and last sentences, respectively. This surface-level generation often disrupts logical coherence, leading to missing or hallucinated cell values.

As illustrated in Figure 3, we guide LLMs to extract table contents through symbolic and incremental code generation that strictly adheres to their order within the source text. For all i and j, if $i < j$, then T_i precedes T_j in the generated table, where T_i represents the content of the i-th cell in the original text, and $i < j$ means that cell i appears before cell

j in the original text. Composite quantities are generated right after the appearance of the last operand in the text. This method minimizes omissions and inconsistencies in the extraction process.

Python code to extract table contents	Input text for table content extraction
java = Island("Java") honshu = Island("Honshū") great_britain = Island("Great Britain") java.set_ranking(1) honshu.set_ranking(2) great_britain.set_ranking(3) indonesia = Country("Indonesia") japan = Country("Japan") java.set_country(indonesia) honshu.set_country(japan)	The top three most populous islands in the world are Java in Indonesia, Honshū in Japan, and Great Britain.
java.set_population(148_756_685) indonesia.set_capital("Jakarta")	Java has an impressive population count of 148,756,685 and is home to Jakarta, the capital of its country.
java.set_area(124_378) java.set_density(1196)	Its land spans 124,378 square kilometers, which equates to a dense population of 1,196 people per square kilometer.
honshu.set_population(102_579_606)	Trailing behind Java is Honshū, Japan's largest island, home to 102,579,606 individuals.
honshu.set_area(227_954)	Honshū boasts a vast area of 227,954 square kilometers, noticeably larger than Java.
print(java.convert_to_tabular_row()) print(honshu.convert_to_tabular_row()) print(great_britain.convert_to_tabular_row())	

Figure 3: An example to illustrate Python code generation for content extraction, grounded to their order within the source text to avoid frequent jumps in the logical flow.

5 Experiments

We examine the performance of TABLECODER based on open-source models such as Mistral-v2 (7B-Instruct-v0.2), LLaMA-2-7B, and LLaMA-2-70B-Instruct (Touvron et al., 2023), and closed-source GPT-3.5 (text-davinci-003) and GPT-4 (the 20230613 4k version) (Brown et al., 2020; OpenAI, 2023). Additionally, we evaluate SOTA baselines, such as the ODIE-DORECT method based on LLaMA-7B (Jiao et al., 2023) and Text-to-Table based on BART-Large (Lewis et al., 2019), and both are fine-tuned using NL-TO-TABLE. We present experiment results in three encoding settings: Markdown (MD) (Singha et al., 2023; Sui et al., 2024) and code (SQL and Python as introduced in Section 4). Ablation studies include:

w/o semantic dependencies The root column is designated as the primary key; others are attributes.

w/o computational dependencies Code for automatic value computation like "_update_density" is removed, but explicit computation is still allowed, e.g., "Honshu.set_density (102579606/227954)".

w/o type and range checking in code

w/o ordered and grounded cell infilling The table is generated row-by-row sequentially.

We experiment with two settings: (1) Few-shot setting: LLMs take the same six-shot examples. Few-shot examples are randomly sampled three times, and the average is used as the final result. (2) Fine-tuning setting: We use all labeled training samples for fine-tuning.

5.1 Implementation details

The text to be extracted is provided as a list of sentences. In markdown, we use "|" to separate cells in a row, and we flatten multiple header rows in our datasets if there are hierarchical headers to meet the markdown requirement. We fine-tune all parameters in BART-Large and partial parameters in LLaMA, and Mistral using LoRA (Hu et al., 2021). Fine-tuning takes 10 epochs for LLaMA and Mistral. For open source models set $lora\_rank$ to 32, $lora\_alpha$ to 64, and $lora\_dropout$ to 0.01 for efficiency, with $batch\_size$ set to 5 and $learning\_rate$ set to 0.00005. We utilize Nvidia A100 GPU nodes to fine-tune LLMs with LoRA. We fix the $temperature$ and top_p to 0 for all LLMs to ensure fair comparison. For ToTTo, we allocate 4,226 for training and 1,015 for testing. For the HiTab dataset, we allocate 667 for training and 169 for testing.

5.2 Experiment Result and Analysis

Table 2 presents the experiment results on table extraction. Experimental results show that LLM fine-tuning increases F1 scores by 8% to 15% compared to few-shot prompting LLMs, outperforming previous SOTA baselines on all datasets.

Code generation significantly enhances the performance of LLMs, whether in few-shot or fine-tuning settings. For example, LLaMA-70B equipped with code generation significantly outperforms the Markdown format of fine-tuned LLaMA-70B by large margins in the F1-EM score, ranging from 12% to 32% for textual cells, single quantities, and composite quantities that are required to be calculated during extraction. Composite cell extraction poses the biggest challenge to existing models, while the accuracy gain of using code generation is the biggest (over 30% for fine-tuned LLaMA-70B in Wikipedia and statistical reports).

SQL performs better than Python in the few-shot learning setting, showing the naturalness of using SQL code for this task, while Python performs better than SQL in the fine-tuning setting, showing the adaptability and flexibility of Python code.

Ablation studies show that utilizing semantic dependencies, type and range checking, and consistently ordered cell infilling greatly improve TABLECODER over vanilla code generation. Leveraging computational relationships highly improves the performance of composite cells (10% on average).

Table 2: Results on NL-to-Table, distinguishing Textual cells (T), Single Quantities (SQ) that do not need computation, and Composite Quantities (CQ) requiring multi-quantity computation.

Cell-level F1-score %	Wikipedia			Reports	
EM with FormatAgnostic-Eval	**SQ**	**CQ**	**T**	**SQ**	**CQ**
Baselines					
Text-to-Table (Bart-Large, Fine-tune)	35.9	15.1	39.2	43.0	20.9
ODIE (LLaMA-7B, Fine-tune)	41.3	18.2	55.6	48.5	25.5
Table Extraction via Markdown					
GPT-3.5, Six-shot	38.8	15.8	45.3	39.2	26.1
GPT-4, Six-shot	47.8	24.2	56.8	45.2	31.3
BART-Large, Fine-tune	33.4	14.5	35.2	39.1	19.7
Mistral-v2-7B, Fine-tune	48.7	22.7	55.4	48.7	30.6
LLaMA-70B, Fine-tune	55.1	21.0	57.3	52.3	31.2
Table Extraction via SQL					
GPT-4, Six-shot	57.5	40.6	63.0	55.9	43.9
—— w/o computation dependencies	57.4	30.4	62.7	55.9	34.1
—— w/o type and range checking	53.3	36.6	62.8	51.7	39.8
—— w/o ordered cell infilling	53.6	37.1	59.5	52.4	41.4
Mistral-v2-7B, Fine-tune	60.4	42.1	65.1	60.4	52.0
—— w/o computation dependencies	60.3	30.3	65.2	60.5	41.0
—— w/o type and range checking	59.8	40.8	64.8	59.9	50.7
—— w/o ordered cell infilling	56.6	38.0	61.7	56.7	49.7
CodeLLaMA-70B, Fine-tune	64.2	47.2	69.2	64.4	57.1
—— w/o computation dependencies	64.4	36.9	68.9	64.6	47.1
—— w/o type and range checking	64.0	45.9	68.6	63.4	55.7
—— w/o ordered cell infilling	60.6	43.3	65.7	61.1	54.4
Table Extraction via Python Code					
GPT-4, Six-shot	55.2	40.0	60.3	53.5	43.3
—— w/o semantic dependencies	52.3	37.9	57.9	50.8	40.3
—— w/o computation dependencies	55.0	29.8	60.4	53.2	33.7
—— w/o type and range checking	50.7	36.2	60.1	49.4	39.2
—— w/o ordered cell infilling	50.9	36.7	56.9	49.7	41.0
Mistral-v2- -7B, Fine-tune	62.0	46.0	66.0	63.3	55.6
—— w/o semantic dependencies	59.0	43.4	64.0	60.4	52.5
—— w/o computation dependencies	61.8	32.0	66.0	62.9	41.3
—— w/o type and range checking	59.8	44.1	65.0	61.3	53.7
—— w/o ordered cell infilling	58.0	42.5	61.6	59.0	53.0
CodeLLaMA-70B, Fine-tune	67.7	52.8	71.7	69.1	62.5
—— w/o semantic dependencies	64.9	50.3	69.7	66.1	59.3
—— w/o computation dependencies	67.4	42.8	72.1	68.9	53.0
—— w/o type and range checking	65.4	51.0	71.4	66.7	60.5
—— w/o ordered cell infilling	63.8	49.4	68.2	65.1	60.2

5.3 Case Study

We manually investigated 100 tables (1,180 cells) from Wikipedia and 100 tables (545 cells) produced by few-shot GPT-4 integrated with Python code generation to analyze their errors. We categorize bad cases into the following types:

(1) Incorrect positions, particularly for tables with complex ontology trees or column names with vague and default information, e.g., "Total", "Master's All" and "Doctoral All" have similar meaning of the sum aggregation in Example 2 of Figure 1, and in Figure 5, two cells are using the cell string"4" but have different meanings. Fortunately, our dataset has provided detailed cell-sentence alignment to enhance model capabilities.

(2) Missing cells caused by computations, especially for those requiring complex numerical reasoning, such as "15" in Figure 7 and "539" in Figure 8.

(3) Incorrect values often stem from complex ontology trees. In Wikipedia.

(4) Incorrect values caused by neglecting the unit conversion, as shown in Figure 6. Although both "billion" and "million" are well understood, LLMs still find it challenging to convert them.

(5) Generated code that is not executable.

(6) Correct semantics but inconsistent formats, e.g., extracting "3rd" from "bronze medal" as shown in Figure 9 and adding the unit ("Km2") of column "Area" to the cell string in Example 1 of Figure 1.

(7) Excessive cells caused by LLMs' internal knowledge or incorrect hallucinations that are not mentioned in the source text, e.g., generating "Tokyo" that is not mentioned in the text as shown in Figure 1. This is undesirable in our task since it's hard to evaluate the correctness without labeling external knowledge beyond the text input. After being augmented with TableCoder's code generation, these cases are much less.

5.4 Evaluation on Complex Tables

In this section, we further investigate TableCoder's scalability regarding complicated table structures.

Our dataset uniquely contains many complex structures, and TableCoder shows significant advancements in handling these compared to baseline methods. We present detailed experimental results of TableCoder for tables with different depths of ontology trees (levels 2, 3, and >3). We group the results by Single Quantities and Composite Quantities. As shown in Table 3, the more complex the structure (i.e., the deeper the ontology tree), the greater the improvement in extraction accuracy by the Semantic Dependency module.

Table 3: Scalability of TableCoder for different ontology depths. "Depth 2" indicates tables with an ontology tree of depth 2; similarly for depth 3 and >3. We highlight the EM F1 (%) values.

EM F1 %	Single Quantity			Composite Quantities		
(Higher is better)	**Depth 2**	**Depth 3**	>3	**Depth 2**	**Depth 3**	>3
Llama 2 (CodeLLaMA 70B)						
TableCoder	74.3	69.8	64.2	68.8	63.3	56.5
- w/o Semantic Dependency	74.1	66.8	59.2	68.5	59.8	51.7
Mistral-v2 (7B)						
TableCoder	72.6	62.7	57.5	66.6	56.0	46.8
- w/o Semantic Dependency	72.1	59.7	50.4	66.3	53.0	42.4

Experimental results indicate that the more complex the structure, the greater the improvement in extraction accuracy provided by the Semantic Dependency module. TableCoder exhibits robust performance in these challenging settings, which demonstrates its capacity to handle real-world data extraction scenarios with deeply nested table ontologies.

Another advancement of TABLECODER is its scalability with respect to input size. TABLECODER generates incremental code that completes the output table step-by-step by adhering to the order within the source text. Unlike existing works that generate output tables row-by-row (e.g., via Markdown), TABLECODER allows a cell in a row to appear at the beginning of a long document and another cell in the *same row* to appear at the end of the long document. **This incremental approach naturally handles large input texts by sequentially dividing the input and filling the table cell-by-cell, rather than row-by-row**. We would like to explore large table extraction in future work.

5.5 FORMATAGNOSTIC-EVAL Effectiveness

We further employ annotators of this dataset as human evaluators to check if the extraction results are correct. Each sample has three annotators to label it, and we use the majority vote as the human evaluation result. Table 4 compares evaluation metrics on 200 randomly selected single-quantity test samples from statistical reports. This reveals that EM, Chrf, and BERT underestimate the performance of models on quantity cells by about 14%, and FORMATAGNOSTIC-EVAL successfully mitigates the gap and reduces it to about 3%. In future work, we would like to explore LLM-based evaluation.

Table 4: Comparison of classic evaluation methods, FORMATAGNOSTIC-EVAL, and human evaluation.

Cell-level F1	Default Evaluation			FORMATAGNOSTIC			Human
%	EM	Chrf	BERT	EM	Chrf	BERT	
GPT-4, Six-shot	41.2	43.8	45.0	53.5	54.0	54.7	58.2
Mistral-v2-7B, Fine-tune	51.3	52.6	54.5	63.3	63.9	64.9	68.4
— W/O semantic	48.3	49.5	50.8	60.4	61.4	62.3	64.0
— W/O computational	51.5	53.0	54.2	62.9	62.9	63.8	66.0
— W/O type and range	50.1	50.9	52.0	61.3	61.9	63.0	65.8
— W/O sequential order	48.3	49.0	50.7	59.0	59.4	60.2	63.8

6 Related Work

Table extraction The "text-to-table" task, as introduced by (Wu et al., 2022; Li et al., 2023b; Pietruszka et al., 2022; Deng et al., 2024; Wang et al., 2024; Singh et al., 2024; Jiao et al., 2023; Jain et al., 2024), represents a pioneering effort in extracting tables from textual content. However, they only involve simple and static key-value pairs or relational tuples without controllable NL instruction. (Huang et al., 2023; Singh et al., 2022; Ma et al., 2024) propose interactive table manipulation from semi-structured data for visualization purposes. To automatically evaluate different column organizations (Ramu et al., 2024; Jiao et al., 2023), (Ramu et al., 2024) break down a table into a list of atomic statements and then measure the statement entailment. Fortunately, the NL instruction in our dataset has provided sufficient details for column organization, so we directly use the column corresponding to the root node of the ontology tree as the index for rows.

Code for table generation Recent studies focused on tasks where tables are inputs (Gao et al., 2023a; Wu et al., 2024; Cheng et al., 2022b; Gao et al., 2023b; Chen et al., 2022; Li et al., 2024; Dong and Wang, 2024) rather than generating structured tables as outputs. As far as we know, the only work to extract tables using code is (Arora et al., 2023), deriving relational tuples from HTML pages and PDFs with tags. However, it targets parsing code to use string processing functions and regular expressions based on tags.

7 Conclusion

We propose TABLECODER, a novel code generation framework for symbolic structure construction and grounded content extraction. To enable training and evaluation, this paper provides a human-labeled dataset targeting generally structured table extraction from text following NL instructions, presenting a unique challenge in this area.

Experimental results show that TABLECODER substantially reduces structure issues and content inaccuracies, which is essential for industrial applications requiring high reliability. Moreover, the code-generation-based method naturally facilitates seamless deployment in existing enterprise data pipelines.

8 Acknowledgments

This work was supported by the Key Project of the Joint Fund for General Technology of the National Natural Science Foundation of China (No. U2336202) and the National Natural Science Foundation of China (No. U21B2009).

References

Simran Arora, Brandon Yang, Sabri Eyuboglu, Avanika Narayan, Andrew Hojel, Immanuel Trummer, and Christopher Ré. 2023. Language models enable simple systems for generating structured views of heterogeneous data lakes. *Proceedings of the VLDB Endowment*, 17(2):92–105.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*.

Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W Cohen. 2022. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *arXiv preprint arXiv:2211.12588*.

Zhoujun Cheng, Haoyu Dong, Zhiruo Wang, Ran Jia, Jiaqi Guo, Yan Gao, Shi Han, Jian-Guang Lou, and Dongmei Zhang. 2022a. Hitab: A hierarchical table dataset for question answering and natural language generation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1094–1110.

Zhoujun Cheng, Tianbao Xie, Peng Shi, Chengzu Li, Rahul Nadkarni, Yushi Hu, Caiming Xiong, Dragomir Radev, Mari Ostendorf, Luke Zettlemoyer, et al. 2022b. Binding language models in symbolic languages. *arXiv preprint arXiv:2210.02875*.

Zheye Deng, Chunkit Chan, Weiqi Wang, Yuxi Sun, Wei Fan, Tianshi Zheng, Yauwai Yim, and Yangqiu Song. 2024. Text-tuple-table: Towards information integration in text-to-table generation via global tuple extraction. *arXiv preprint arXiv:2404.14215*.

Haoyu Dong and Zhiruo Wang. 2024. Large language models for tabular data: Progresses and future directions. In *Proceedings of the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 2997–3000.

Haoyu Dong, Jianbo Zhao, Yuzhang Tian, Junyu Xiong, Shiyu Xia, Mengyu Zhou, Yun Lin, José Cambronero, Yeye He, Shi Han, et al. 2024. Spreadsheetllm: Encoding spreadsheets for large language models. *arXiv preprint arXiv:2407.09025*.

Dawei Gao, Haibin Wang, Yaliang Li, Xiuyu Sun, Yichen Qian, Bolin Ding, and Jingren Zhou. 2023a. Text-to-sql empowered by large language models: A benchmark evaluation. *arXiv preprint arXiv:2308.15363*.

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. 2023b. Pal: Program-aided language models. In *International Conference on Machine Learning*, pages 10764–10799. PMLR.

Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*.

Yanwei Huang, Yunfan Zhou, Ran Chen, Changhao Pan, Xinhuan Shu, Di Weng, and Yingcai Wu. 2023. Interactive table synthesis with natural language. *IEEE Transactions on Visualization and Computer Graphics*.

Parag Jain, Andreea Marzoca, and Francesco Piccinno. 2024. Structsum generation for faster text comprehension. *arXiv preprint arXiv:2401.06837*.

Yizhu Jiao, Ming Zhong, Sha Li, Ruining Zhao, Siru Ouyang, Heng Ji, and Jiawei Han. 2023. Instruct and extract: Instruction tuning for on-demand information extraction. *arXiv preprint arXiv:2310.16040*.

J Richard Landis and Gary G Koch. 1977. The measurement of observer agreement for categorical data. *biometrics*, pages 159–174.

Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Ves Stoyanov, and Luke Zettlemoyer. 2019. Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. *arXiv preprint arXiv:1910.13461*.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023a. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*.

Tong Li, Zhihao Wang, Liangying Shao, Xuling Zheng, Xiaoli Wang, and Jinsong Su. 2023b. A sequence-to-sequence&set model for text-to-table generation. *arXiv preprint arXiv:2306.00137*.

Zixuan Li, Yutao Zeng, Yuxin Zuo, Weicheng Ren, Wenxuan Liu, Miao Su, Yucan Guo, Yantao Liu, Xiang Li, Zhilei Hu, et al. 2024. Knowcoder: Coding structured knowledge into llms for universal information extraction. *arXiv preprint arXiv:2403.07969*.

Zeyao Ma, Bohan Zhang, Jing Zhang, Jifan Yu, Xiaokang Zhang, Xiaohan Zhang, Sijia Luo, Xi Wang, and Jie Tang. 2024. Spreadsheetbench: Towards challenging real world spreadsheet manipulation. *arXiv preprint arXiv:2406.14991*.

Linyong Nan, Dragomir Radev, Rui Zhang, Amrit Rau, Abhinand Sivaprasad, Chiachun Hsieh, Xiangru Tang, Aadit Vyas, Neha Verma, Pranav Krishna, et al. 2020. Dart: Open-domain structured data record to text generation. *arXiv preprint arXiv:2007.02871*.

OpenAI. 2023. GPT-4 technical report. *CoRR*, abs/2303.08774.

Ankur P Parikh, Xuezhi Wang, Sebastian Gehrmann, Manaal Faruqui, Bhuwan Dhingra, Diyi Yang, and Dipanjan Das. 2020. Totto: A controlled table-to-text generation dataset. *arXiv preprint arXiv:2004.14373*.

Michał Pietruszka, Michał Turski, Łukasz Borchmann, Tomasz Dwojak, Gabriela Pałka, Karolina Szyndler, Dawid Jurkiewicz, and Łukasz Garncarek. 2022. Stable: Table generation framework for encoder-decoder models. *arXiv preprint arXiv:2206.04045*.

Maja Popović. 2015. chrf: character n-gram f-score for automatic mt evaluation. In *Proceedings of the tenth workshop on statistical machine translation*, pages 392–395.

Pritika Ramu, Aparna Garimella, and Sambaran Bandyopadhyay. 2024. Is this a bad table? a closer look at the evaluation of table generation from text. *arXiv preprint arXiv:2406.14829*.

Mukul Singh, José Cambronero, Sumit Gulwani, Vu Le, Carina Negreanu, Mohammad Raza, and Gust Verbruggen. 2022. Cornet: Learning table formatting rules by example. *arXiv preprint arXiv:2208.06032*.

Mukul Singh, Gust Verbruggen, Vu Le, and Sumit Gulwani. 2024. Tabularis revilio: Converting text to tables. In *Proceedings of the 33rd ACM International Conference on Information and Knowledge Management*, pages 4056–4060.

Ananya Singha, José Cambronero, Sumit Gulwani, Vu Le, and Chris Parnin. 2023. Tabular representation, noisy operators, and impacts on table structure understanding tasks in llms. *arXiv preprint arXiv:2310.10358*.

Yuan Sui, Mengyu Zhou, Mingjie Zhou, Shi Han, and Dongmei Zhang. 2024. Table meets llm: Can large language models understand structured table data? a benchmark and empirical study. In *Proceedings of the 17th ACM International Conference on Web Search and Data Mining*, pages 645–654.

Xiangru Tang, Yiming Zong, Yilun Zhao, Arman Cohan, and Mark Gerstein. 2023. Struc-bench: Are large language models really good at generating complex structured data? *arXiv preprint arXiv:2309.08963*.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurélien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. Llama: Open and efficient foundation language models. *CoRR*, abs/2302.13971.

Haochen Wang, Kai Hu, Haoyu Dong, and Liangcai Gao. 2024. Doctabqa: Answering questions from long documents using tables. In *International Conference on Document Analysis and Recognition*, pages 470–487. Springer.

Zhiruo Wang, Haoyu Dong, Ran Jia, Jia Li, Zhiyi Fu, Shi Han, and Dongmei Zhang. 2021. Tuta: Tree-based transformers for generally structured table pre-training. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, pages 1780–1790.

Jack Williams, Carina Negreanu, Andrew D Gordon, and Advait Sarkar. 2020. Understanding and inferring units in spreadsheets. In *2020 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 1–9. IEEE.

Xueqing Wu, Jiacheng Zhang, and Hang Li. 2022. Text-to-table: A new way of information extraction. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2518–2533.

Yang Wu, Yao Wan, Hongyu Zhang, Yulei Sui, Wucai Wei, Wei Zhao, Guandong Xu, and Hai Jin. 2024. Automated data visualization from natural language via large language models: An exploratory study. *Proceedings of the ACM on Management of Data*, 2(3):1–28.

Tianyi Zhang, Varsha Kishore, Felix Wu, Kilian Q Weinberger, and Yoav Artzi. 2019. Bertscore: Evaluating text generation with bert. *arXiv preprint arXiv:1904.09675*.

Fan Zhou, Haoyu Dong, Qian Liu, Zhoujun Cheng, Shi Han, and Dongmei Zhang. 2022. Reflection of thought: Inversely eliciting numerical reasoning in language models via solving linear systems. *arXiv preprint arXiv:2210.05075*.

A NL-to-Table Dataset Construction

We design an annotation process with six steps. Through a reliable and publicly listed data service vendor company, we recruited 38 students or graduates (16 women and 22 men) who are majoring in computer science from top universities to correct quality issues of table content extraction, label column properties, and relationships, and annotate format strings. Labeling costs 1,120 working hours. Comprehensive online training, documents, and QA are provided to annotators to ensure their consistent understanding of the labeling requirements.

A.1 User Instruction in Natural Language

We utilize GPT-4 to create an initial set of instructions based on the input table using the following instructions.

```
Suppose you are a human and want to ask
GPT-4 to extract a table from the
following text: <TEXT>

Imagine that your desired table is as
follows: <TABLE>

How should you ask GPT-4 using an
instruction? This instruction describes
the content and structure of the table
you want in natural language.
```

Column names should be consistent with the target table to facilitate evaluation, and the table structure, whether flat or hierarchical, is also required to be described. We encourage various forms of expression to simulate different habits of users. So we set GPT-4's temperature to 1 and encourage annotators to adapt the prompt and guide GPT in generating queries with diverse styles. Finally, instructions are manually refined to ensure clarity and alignment.

A.2 Column Property and Relationship

Ontology tree and computational dependency Column relationships, as detailed in Section 2, are finely labeled with JSON format, employing ontology trees for semantic relationships and spreadsheet formulas for computational relationships.

Unit and feasible range Annotators label the unit (Williams et al., 2020) and feasible range of each number column, and we use rules to infer types such as `INT` and `DECIMAL` based on cell text.

Format string Each column that contains numbers, dates, and times is annotated with an f-string, a Python feature for string formatting. For example, Figure 9 in Appendix presents a complex number string, we label it using an f-string f'%d%s' % (n, 'th' if 4 <= n % 100 <= 20 else {1: 'st', 2: 'nd', 3: 'rd'}.get(n % 10, 'th')). The f-string in Figure 6 is labeled to be f'{n / 1_000_000:.0f} million'.

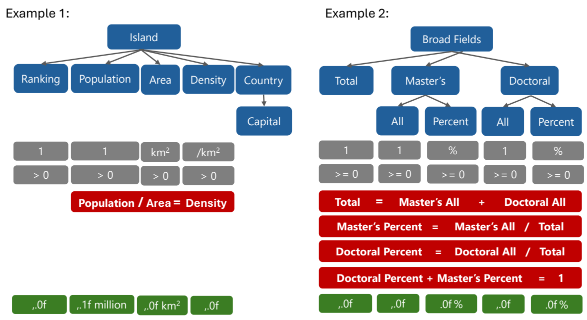

Figure 4: Examples illustrating column relationships through ontology trees (blue), number units and feasible ranges (gray), formulas (red), and number format strings (green).

Below is an example JSON structure of Column Property and Relationship Annotation for the example in Figure 1:

```
{
  "Root of the Ontology Tree": {
    "Children": [
      {
        "Island": {
          "Children": [
            {"Ranking": {"Unit": 1, "
                Range": ">0", "
                FormatString": "\{self.
                ranking:,0f\}", "Children
                ": []}},
            {"Population": {"Unit": 1, "
                Range": ">0", "
                FormatString": "\{self.
                population / 1_000_000:,1
                f\} million", "Children":
                 []}},
            {"Area": {"Unit": "km2", "
                Range": ">0", "
                FormatString": "\{self.
                area:,0f\} km2", "
                Children": []}},
            {"Density": {"Unit": "/km2",
                "Range": ">0", "
                FormatString": "\{self.
                density:,0f\}", "Children
                ": []}},
            {
              "Country": {
                "Children": [
                  {"Capital": { "Children
                      ": []}}]
                }}]}}]},
  "Formulas for Computational
      Relationships": [
    "[@Population] / [@Area] =  [@Density
         (/km2)]"]
}
```

A.3 Table Extraction

Based on collected table-text paired data, annotators are instructed to be careful with deleting unmentioned cells, adding missing cells, and revising inconsistent cells in the table. For any omitted cells, annotators must accurately record the corresponding sentences, adhering to the methodologies employed by ToTTo and HiTab, ensuring all extracted cells are linked to corresponding sentences. Crucially, human annotators jointly utilize values and format strings to accurately record quantity cells.

Approximation Given that a quantity expressed in the text may be an approximate value, annotators receive careful training to label them with precision. Figure 6 presents a representative example that the cell value ("127,955") is too precise to be extracted from the text mention ("128.0 billion"), showing that merely reversing the table-to-text dataset can produce lots of overprecise errors, so we label the cell to be 128,000 to ensure the information is extractable.

Same value in different positions Annotators are required to be careful about distinguishing different cells with the same value. Figure 5 showcases an example of a labeling error by (Parikh et al., 2020), the red box surrounds the annotated cell for text generation, but the correct one is the cell surrounded by the green box.

Numerical reasoning inside the table Figure 7 demonstrates that the extracted cell ("15%") is calculated by "33%" and "18%", which can be easily omitted by human annotators. Figure 8 also shows a cell that needs calculation in Wikipedia. These cases require numerical reasoning, which is a core capability needed to meet the key demands and pain points of table extraction in financial and audit domains. Gladly, with the annotation on computational dependency, these cases can be labeled in high quality.

Numerical reasoning outside the table There is another kind of challenging case in which annotators need to verify if a cell can be inferred through numerical using the text information. Figure 8 shows an example where the cell "539" is calculated from "2,146" and "1,607" in Wikipedia pages.

A.4 Converting Annotations to Code

Writing code is challenging for annotators. Instead, we propose a rule-based system to construct code based on human-labeled table structure and content. (1) It converts the table structure—comprising ontology trees, formulas, number units, and format strings in JSON format—into Python classes and SQL statements, as depicted in Section 4. In Python, the implementation involves building one or more classes based on ontology trees. Each class contains various properties and methods for value setting, checking, computation, and formatting. In SQL, to avoid splitting the target table into multiple tables, we use the column corresponding to the root of the ontology tree as the primary key, and other columns as attribute columns. (2) It transforms table content annotations into Python code and SQL `INSERT` and `UPDATE` statements, preserving the sequence of code snippets to reflect the order in which cells appear in the source text, as shown in the example in Figure 3. We refine this rule-based system until all generated code can be executed flawlessly and produce corresponding table contents.

A.5 Regular Inspections and the Final Review

Due to the complexity of the labeling task, we have designated our two most experienced annotators to conduct regular inspections and the final review. (1) During the labeling process, they periodically review a sample of annotations (about 3%) from all annotators to provide timely feedback on any issues. (2) In the final step, they review all annotations to correct any errors. The agreement between the two annotators was evaluated by comparing annotations by all annotators (who are randomly paired) on a randomly selected sample of 200 tables. Table-level Fleiss Kappa (Landis and Koch, 1977) are 0.89 for table content extraction, 0.82 for column relationship and property labeling, and 0.94 for format string annotation, which is regarded as "almost perfect agreement" (Landis and Koch, 1977). And 98.5% instructions are considered accurate and high-quality by the counterpart.

B Existing Evaluation Metrics

Exact match (Popović, 2015) determines if two texts are the same. Chrf (Popović, 2015) calculates character-level n-gram similarity between two texts, useful for assessing similarity in a more granular manner. BERTScore (Zhang et al., 2019) measures the similarity of BERT embeddings between two texts, providing a neural semantic similarity metric.

Existing evaluation methods use the left-most column to distinguish rows (Wu et al., 2022). However, the left-most column does not always distinctly index rows in real tables. Instead, NL-TO-TABLE leverages the annotations of the ontology tree and uses the column corresponding to the root node of the ontology tree as the index for rows. The flattened row headers are used to index columns. Therefore, the evaluation metric is agnostic to the order of columns and rows.

2016 season [edit]

In Week 2, Amendola caught four passes for 48 yards and a career-high two touchdowns from Jimmy Garoppolo in a 31–24 win over the Miami Dolphins.[52] In Week 13 against his former team, the St. Louis Rams, he suffered a high ankle sprain that sidelined him for the rest of the regular season, but he returned for the playoffs.[53] The Patriots reached Super Bowl LI, where Amendola had eight catches for 78 yards in the Patriots' historic 34–28 overtime comeback victory over the Atlanta Falcons.[54] Amendola scored the Patriots' first touchdown of the fourth quarter to narrow what had been a 25-point Falcons lead down to 28–18 and a two-point conversion with less than a minute to go to tie the game at 28–28.[55] His Super Bowl LI touchdown was his second Super Bowl receiving touchdown. He became the 27th player in NFL history to have at least two career receiving touchdowns in the Super Bowl.[56] Amendola finished the season with 23 receptions on 29 targets for 243 yards and four touchdowns in 2016.[57] His 79.3% catch rate was the best of his career.[58]

Regular season [edit]

Year	Team	Games		Receiving					Rushing					Kickoff ret		
		GP	GS	Rec	Yds	Avg	Lng	TD	Att	Yds	Avg	Lng	TD	Ret	Yds	Avg
2009	STL	14	2	43	326	7.6	25	1	3	-2	-0.7	8	0	**66**	**1,618**	24.5
2010	STL	**16**	6	**85**	**689**	8.1	36	3	**7**	**81**	**11.6**	**30**	0	50	1,142	22.8
2011	STL	1	1	5	45	9.0	18	0	—	—	—	—	—	—	—	—
2012	STL	11	8	63	666	10.6	56	3	2	8	4.0	6	0	2	16	8.0
2013	NE	12	6	54	633	11.7	**57**	2	1	1	1.0	1	0	—	—	—
2014	NE	**16**	4	27	200	7.4	21	1	—	—	—	—	—	20	482	24.1
2015	NE	14	7	65	648	10.0	41	3	2	11	5.5	8	0	8	172	21.5
2016	NE	12	4	23	243	10.6	32	**4**	—	—	—	—	—	5	129	**25.8**
2017	NE	15	8	61	659	10.8	27	2	—	—	—	—	—	1	16	16.0
2018	MIA	15	**15**	59	575	9.7	39	1	1	-2	-2.0	-2	0	—	—	—
2019	DET	15	9	62	678	10.9	47	1	—	—	—	—	—	—	—	—
2020	DET	14	5	46	602	**13.1**	50	0	1	2	2.0	2	0	—	—	—
2021	HOU	8	0	24	248	10.3	39	3	—	—	—	—	—	1	15	15.0

https://en.wikipedia.org/wiki/Danny_Amendola

Figure 5: Example of the position challenge.

Current-dollar federal obligations[2] for research and development and R&D plant decreased 1% from FY 2014 to FY 2015, from $132.5 billion to $131.4 billion. Within this total, funding for research increased 1% to $63.6 billion while development funding fell 4% to $64.9 billion. R&D plant funding increased substantially (by 27%) to $2.8 billion (table 1). Federal agencies estimated an 8% total increase in FY 2016 obligations for R&D and R&D plant, to $142.6 billion, and projected a 2% increase in FY 2017 to $145.4 billion. After adjusting for inflation, total federal R&D and R&D plant obligations decreased 2% to $119.6 billion from FY 2014 to FY 2015. Constant-dollar obligations were estimated to increase 7% to $127.7 billion in FY 2016 and were projected to remain essentially flat at $128.0 billion in FY 2017 (table 1).

TABLE 1. Federal obligations for research and development and R&D plant, by type of R&D: FYs 2013–17

Type of R&D	Current $millions					Constant 2009 $millions				
	2013	2014	2015	2016 preliminary	2017 projected	2013	2014	2015	2016 preliminary	2017 projected
All R&D and R&D plant	127,291	132,496	131,398	142,555	145,408	119,399	122,195	119,561	127,692	127,955
R&D	125,386	130,279	128,573	140,070	142,608	117,612	120,150	116,991	125,466	125,491
Research	59,198	62,909	63,645	67,761	69,744	55,528	58,018	57,912	60,696	61,373
Basic	29,779	31,588	31,527	33,227	34,323	27,933	29,132	28,687	29,763	30,203
Applied	29,419	31,321	32,118	34,533	35,421	27,595	28,886	29,225	30,932	31,169
Development	66,188	67,370	64,928	72,309	72,865	62,084	62,132	59,079	64,770	64,119
Science and technology	13,471	14,313	15,279	16,339	16,311	12,636	13,200	13,903	14,635	14,353
Major systems[a]	52,717	53,057	49,649	55,971	56,554	49,448	48,932	45,177	50,135	49,766
R&D plant	1,905	2,218	2,825	2,485	2,799	1,787	2,046	2,571	2,226	2,463

https://www.nsf.gov/statistics/2017/nsf17316/overview.htm

Figure 6: Example of the unit conversion challenge.

The proportion of women with a university degree in both types of families has increased over time, however at a slower pace for female lone parents. The proportion of female lone parents with a university degree more than doubled between 1991 and 2011 to 20% (a difference of 11 percentage points). The proportion of female parents in couples with a university degree also doubled in that time period to 33% (a difference of 18 percentage points). The gap in education levels between female lone parents and female parents in couples may be partly explained by the tendency for female lone parents to have had their children at a younger age. 47

Table 9
Percentage of Highest certificate, diploma or degree of female lone parents and female parents in couples, aged 25 to 54 with children aged 15 and under in 1991, 2001 and 2011, Canada

Highest certificate, diploma or degree	Female lone parents				Female parent in couples			
	1991	2001	2011	Difference (2011 - 1991)	1991	2001	2011	Difference (2011 - 1991)
	Percent							
Total	100	100	100	...	100	100	100	...
No certificate, diploma or degree	34	20	13	**-21**	24	13	8	**-16**
High school diploma or equivalency	30	28	25	**-5**	32	28	21	**-11**
Postsecondary certificate below the bachelor's level	26	39	42	**16**	29	36	38	**9**
University degree at the bachelor's level or above	9	13	20	**11**	15	23	33	**18**

https://www150.statcan.gc.ca/n1/pub/89-503-x/2015001/article/14640-eng.htm

Figure 7: Example of the computation challenge.

the last drive of the game with 23 yards on 6 rushes. The Eagles won 24–22 and earned a playoff spot – the third seed in the NFC at 10–6.[115][116] McCoy rushed for 77 yards and one touchdown in the Eagles' Wild Card Round game against the 11–5 New Orleans Saints, but the team lost 26–24 after a last-second field goal.[117]

For the 2013 season, McCoy rushed for 1,607 yards and was also the all-purpose yards leader at 2,146.[118][119][120]

Regular season [edit]

Year	Team	Games		Rushing					Receiving					Fumbles	
		GP	GS	Att	Yds	Avg	Lng	TD	Rec	Yds	Avg	Lng	TD	Fum	Lost
2009	PHI	**16**	4	155	637	4.1	66T	4	40	308	7.7	45	0	2	1
2010	PHI	15	13	207	1,080	5.2	62	7	**78**	**592**	7.6	40	2	2	1
2011	PHI	15	15	273	1,309	4.8	60	**17**	48	315	6.6	26	**3**	1	1
2012	PHI	12	12	200	840	4.2	34	2	54	373	6.9	36	**3**	**4**	**3**
2013	PHI	**16**	**16**	**314**	**1,607**	5.1	57T	9	52	539	**10.4**	**70**	2	1	1
2014	PHI	**16**	**16**	312	1,319	4.2	53	5	28	155	5.5	18	0	**4**	**3**
2015	BUF	12	12	203	895	4.4	48T	3	32	292	9.1	22	2	2	1
2016	BUF	15	15	234	1,267	**5.4**	**75T**	13	50	356	7.1	41	1	3	0
2017	BUF	**16**	**16**	287	1,138	4.0	48T	6	59	448	7.6	39	2	3	1
2018	BUF	14	13	161	514	3.2	28T	3	34	238	7.0	24	0	0	0
2019	KC	13	9	101	465	4.6	39	4	28	181	6.5	23	1	3	2
2020	TB	10	0	10	31	3.1	14	0	15	101	6.7	15	0	0	0

https://en.wikipedia.org/wiki/LeSean_McCoy

Figure 8: Example of the computation challenge.

Her bronze medal time, behind a pair of young Kenyans, at the 2014 Commonwealth Games of 15:08.96 bettered the listed W40 World Record by almost 12 seconds, however Pavey ran an even better time of 15:04.87 at the Golden Gala two months earlier.[42] The Commonwealth Games race was probably one of the most exciting races of her career. In the closing four laps Pavey battled the Kenyans refusing to give up the lead. She went to the front, after being overtaken on three occasions. On the final bend the Kenyan runners had all gone past her again and opened a small gap but Pavey battled back again down the home straight overtaking one of the Kenyan athletes and narrowly missing the Silver medal by 6/100th of a second.

Jo Pavey at the 2014 Commonwealth Games in Glasgow

International competitions [edit]

Year	Competition	Venue	Position	Event	Notes
Representing England					
2002	Commonwealth Games	Manchester, United Kingdom	5th	5000 m	15:19.91
2006	Commonwealth Games	Melbourne, Australia	2nd	5000 m	14:59.08
2014	Commonwealth Games	Glasgow, United Kingdom	3rd	5000 m	15:08.96

https://en.wikipedia.org/wiki/Jo_Pavey

Figure 9: Example of format evaluation challenge.