Can an LLM find its way around a Spreadsheet?

Cho-Ting Lee

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Science and Applications

May 2nd, 2024

Arlington, Virginia

Keywords: LLMs, data cleaning, end-user programming

Can an LLM find its way around a Spreadsheet?

Cho-Ting Lee

(ABSTRACT)

Spreadsheets are routinely used in business and scientific contexts, and one of the most vexing challenges data analysts face is performing data cleaning prior to analysis and evaluation. The ad-hoc and arbitrary nature of data cleaning problems, such as typos, inconsistent formatting, missing values, and a lack of standardization, often creates the need for highly specialized pipelines. We ask whether an LLM can find its way around a spreadsheet and how to support end-users in taking their free-form data processing requests to fruition. Just like RAG retrieves context to answer users' queries, we demonstrate how we can retrieve elements from a code library to compose data processing pipelines. Through comprehensive experiments, we demonstrate the quality of our system and how it is able to continuously augment its vocabulary by saving new codes and pipelines back to the code library for future retrieval.

Can an LLM find its way around a Spreadsheet?

Cho-Ting Lee

(GENERAL AUDIENCE ABSTRACT)

Spreadsheets are frequently utilized in both business and scientific settings, and one of the most challenging tasks that must be accomplished before analysis and evaluation can take place is the cleansing of the data. The ad-hoc and arbitrary nature of issues in data quality, such as typos, inconsistent formatting, missing values, and lack of standardization, often creates the need for highly specialized data cleaning pipelines. Within the scope of this thesis, we investigate whether a large language model (LLM) can navigate its way around a spreadsheet, as well as how to assist end-users in bringing their free-form data processing requests to fruition. Just like Retrieval-Augmented Generation (RAG) retrieves context to answer user queries, we demonstrate how we can retrieve elements from a Python code reference to compose data processing pipelines. Through comprehensive experiments, we showcase the quality of our system and how it is capable of continuously improving its code-writing ability by saving new codes and pipelines back to the code library for future retrieval.

Acknowledgments

I am sincerely grateful to Professor Naren Ramakrishnan, my advisor, for affording me the opportunity to engage in captivating projects and for his unwavering support, guidance, and patience throughout the research journey. His expertise, insightful feedback, and encouragement have been invaluable in shaping this thesis and enhancing its quality.

I would like to express my deep appreciation to my committee members, Professor Chang-Tien Lu and John Simeone, for their invaluable suggestions and meticulous review of my work. Their feedback has significantly improved the caliber of this thesis.

Furthermore, I would like to extend my gratitude to Shengzhe Xu for his invaluable guidance throughout my research. His insightful suggestions and constructive feedback have greatly enriched the quality of my work.

I also want to convey my gratitude for the financial support and valuable data provided by Jade Saunders and Marigold Norman from World Forest ID (WFID), which were instrumental in enabling this thesis.

Special appreciation is due to Jay Katyan, Patrick Cross, and Sharanya Pathakota for their assistance with my work. Their dedication and collaborative efforts have been essential to the completion of the project.

Lastly, heartfelt thanks to my grandparents, my parents, and my brother for their enduring love, encouragement, and unwavering belief in my capabilities. Their steadfast support has empowered me with newfound confidence and enabled achievements beyond my expectations.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Pre-trained large language models (LLMs) have been demonstrated to be adept at code generation from natural language prompts, ushering in a new era in software development [10, 12]. Modern IDEs such as Visual Studio Code and IntelliJ provide access to LLMs to support mistake correction, code recommendation, and the ability to converse in English to understand and navigate complex codebases [23].

1.1 Motivation

One of the vexing forms of code that programmers deal with pertains to processing tabular data, e.g., in the form of spreadsheets. Due to the substantial size of the data (Fig. 1.1, for example), they are often beset with issues such as typos, inconsistent formatting, missing information, and a lack of standardization. Not solving these issues beforehand can also cause several research problems in future ML analyses. For example, uncleaned and unformatted data that contains irrelevant information can lead to over-fitting the model, longer training times, and biased classification results. These problems state that

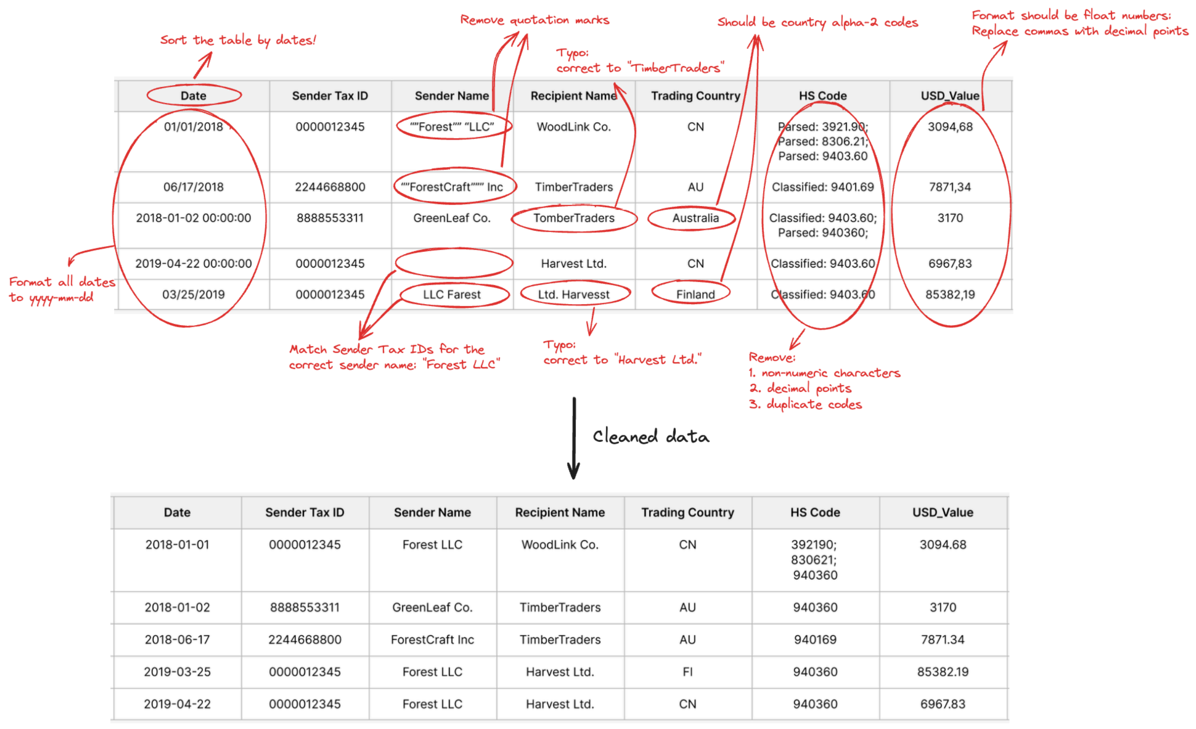
Table 1.1: Cleaning performance of our approach (TS: TradeSweep). Note the improvement from 9% to 98% for the Timber dataset.

	Teak	**Grain**	**Timber**
Init correct-rate	86.60%	85.94%	9.09%
TS correct-rate	**97.19%**	**97.61%**	**98.17%**

preprocessing data is a crucial step to avoid less accurate or less interpretable models, creating the need for highly specialized pipelines [1, 15] to preprocess (or clean, to be more specific) the data before proceeding with future analysis.

Data preprocessing involves identifying relevant and information-preserving columns in any given dataset, where a human analyst determines which categories have the potential to be utilized for downstream analysis. Once such relevant categories have been determined, the dataset is ready to undergo cleaning. Throughout the process, the analyst reviews the results and suggests modifications to the data. However, manually doing this for datasets demands significant effort, and automation often requires thorough domain knowledge and proficiency in programming. Furthermore, data cleaning is not simply a matter of resolving inconsistencies and can lead to significant shifts in results downstream. For example, the mere act of correcting typos in a dataset may inadvertently lead to over-clustering or under-clustering of the pertinent column; inaccurately resolving entities can distort the true distribution of data points. Thus, data cleaning needs to be approached with care and deliberation, with attention paid to sequential transformations.

It is clear that an automated procedure for data interaction and a more effective way to perform data preprocessing are urgently needed in place of direct human engagement. So in this thesis, we ask whether an LLM can find its way around a spreadsheet and how to support end-users in taking their free-form data processing requests to fruition. Our approach, TradeSweep (named for its focus on tabular trade datasets), takes English requests for data cleaning and generates code proposals that can be composed and applied to targeted datasets, achieving high performance as shown in Table 1.1. Just like RAG (retrieval-augmented generation) retrieves context to answer users' queries, we demonstrate how we can store and retrieve elements from a code library to compose complex pipelines.


Sort the table by dates!
Remove quotation marks
Typo:
correct to "TimberTraders"
Should be country alpha-2 codes
Format should be float numbers:
Replace commas with decimal points
Format all dates
to yyyy-mm-dd
Match Sender Tax IDs for the
correct sender name: "Forest LLC"
Typo:
correct to "Harvest Ltd."
Remove:
1. non-numeric characters
2. decimal points
3. duplicate codes
Cleaned data

Date	Sender Tax ID	Sender Name	Recipient Name	Trading Country	HS Code	USD_Value
01/01/2018	0000012345	""Forest"" "LLC"	WoodLink Co.	CN	Parsed: 3921.90; Parsed: 8306.21; Parsed: 9403.60	3094,68
06/17/2018	2244668800	""ForestCraft""" Inc	TimberTraders	AU	Classified: 9401.69	7871,34
2018-01-02 00:00:00	8888553311	GreenLeaf Co.	TomberTraders	Australia	Classified: 9403.60; Parsed: 940360;	3170
2019-04-22 00:00:00	0000012345		Harvest Ltd.	CN	Classified: 9403.60	6967,83
03/25/2019	0000012345	LLC Farest	Ltd. Harvesst	Finland	Classified: 9403.60	85382,19

Date	Sender Tax ID	Sender Name	Recipient Name	Trading Country	HS Code	USD_Value
2018-01-01	0000012345	Forest LLC	WoodLink Co.	CN	392190; 830621; 940360	3094.68
2018-01-02	8888553311	GreenLeaf Co.	TimberTraders	AU	940360	3170
2018-06-17	2244668800	ForestCraft Inc	TimberTraders	AU	940169	7871.34
2019-03-25	0000012345	Forest LLC	Harvest Ltd.	FI	940360	85382.19
2019-04-22	0000012345	Forest LLC	Harvest Ltd.	CN	940360	6967.83

Figure 1.1: Potential errors and issues in a "dirty" data.

1.2 Overview

This thesis is divided into five chapters. Chapter 2 surveys some related work focusing on different aspects of interactions with data and code generation by LLMs. Chapter 3 provides a detailed explanation of how TradeSweep works to automate data preprocessing tasks with self-enhancements. In Chapter 4, we explain our evaluation methodology on TradeSweep ; We design three baselines and compare their performance with TradeSweep on generating codes for data cleaning, performing an ablation study to further discuss several research questions. Last but not least, Chapter 5 concludes this thesis and proposes potential future directions for improving TradeSweep 's limitations.

Chapter 2

Review of Literature

2.1 Generating Formulas and SQL queries

Formulas and SQL queries are the lingua franca of spreadsheets. 'Formula Language Model for Excel' (FLAME) [14] is a T5-based model trained exclusively on Excel formulas that achieves competitive performance vs. LLMs (such as CodeT5) on facets such as formula repair, formula completion, and similarity-based formula retrieval. Results show that FLAME achieves the best results in 11 out of 16 evaluation settings and can successfully fix, complete, and retrieve formulas. However, in data analysis use cases, users may have different tabular formats (Excel, CSV, etc.) and hope to perform more complex tasks like correcting misspellings using a customized algorithm, filling in blanks by locating external documents, or even writing new code. This brings out the limitations of FLAME since this tool is restricted to using pre-defined formulas in Excel.

A substantial amount of research aims to convert natural language text into executable SQL queries. While LLMs like GPT-4 and Claude-2 demonstrate the capacity to accomplish text-to-SQL tasks, most current benchmarks concentrate on tiny databases with few rows, which is not the same as working on massive databases in real-world scenarios. In an effort to narrow the gap between experimental and practical scenarios, BIRD (Big Bench for Large-scale Databases) [18] is a text-to-SQL benchmark that is the first to provide more effective query techniques in the context of large and noisy databases. It also explores three challenges:

handling big and unclean databases, maximizing the effectiveness of SQL execution, and evaluating outside information sources. While BIRD made it possible for users to access and modify databases without the need for programming expertise, this approach is limited to using SQL; As data preprocessing is usually accomplished with Python, it is troublesome to write complex procedures in SQL.

2.2 LLMs with Information Retrieval

While LLMs are highly effective when fine-tuned for particular NLP tasks, they are inherently limited by their capability to access and precisely manipulate knowledge. Retrieval-augmented generation (RAG) [16] is a runaway success in how it combines retrieval and generation techniques, leading to numerous offshoots and variations in order to improve LLMs in understanding and producing human-like text. When compared to task-specific retrieve-and-extract architectures and parametric sequence-to-sequence models, RAG has shown superior performance, demonstrating a notable improvement in the factual correctness, specificity, and diversity of the generated language. This line of work led to a blurring of the lines between information retrieval and language generation.

The development of LLMs has caused a paradigm shift in human information acquisition, from using information retrieval to search for information, to generating information with them. As a result, information retrieval systems are now supporting LLMs rather than just humans. For instance, Qiaoyu Tang et al. proposed Self-Retrieval [24], an end-to-end information retrieval architecture driven by LLMs. In Self-Retrieval, an information document is internalized as a corpus into an LLM, and the retrieval process is redesigned as a process of document generation and self-evaluation carried out using the same model.

The advent of vector database systems such as Qdrant [22] and Pinecone [21] has contributed to the rapid adoption of RAG and LLM pipelines. With the vector database storing embedded vectors and data items, users can search the database using an embedded query vector and retrieve the closest matching data results. The database employs multiple distinct approximate-nearest-neighbor search algorithms and assembles them to retrieve the queried vector's neighbors quickly and precisely. As generative AI models advance, the importance of vector database systems continues to grow; By providing fast similarity searches and closest match queries, it enables LLMs to learn from large datasets.

In this thesis, we adapt the RAG approach in a code composition setting. Specifically, we retrieve only the most relevant codes and narrow the number of code candidates that are prompted into the LLM, reducing the prompt context length and allowing the LLM to learn with the most helpful information.

2.3 Data Preprocessing Pipelines

Data preprocessing is essential for enhancing the consistency and dependability of raw data, which often includes 'dirty' data containing inconsistent formats, typos, missing values, and outliers. In addition to requiring domain expertise, programmers often employ feature engineering to overcome deficiencies in data quality.

Fan et al. investigated and carried out a series of data preprocessing tasks on building operation data for additional analysis [7]. In order to ensure data compatibility with algorithm analysis, they proposed tasks for building operation data, including data transformation (encoding categorical columns to ensure data compatibility with algorithm analysis), data reduction (reducing data dimensions both row-wise and column-wise), data scaling (scaling data into similar ranges using max-min normalization and z-score standardization), and data partitioning (dividing data into subsets for in-depth analysis). In their study, they gave a thorough overview of both traditional and advanced data preprocessing duties. Furthermore, they addressed the necessity of automating these jobs to improve the efficiency of data analysis, which can be solved in this thesis.

Several network-based quantitative and qualitative strategies for timber data analysis were represented by Charvi Gopal in his work to fight against deforestation, especially illegal harvesting and over-logging of protected forests [9]. Following the computation of transaction data into adjacency matrices, he analyzed the export-import distribution as well as the primary trading partners of each export and import country. The research transformed the matrices into heatmaps to observe differences and qualitative trends between imports and exports, and to facilitate comparisons on smaller subsets of data. The study also computed correlation coefficient values from normalized matrices to observe the correlation between exports and imports, and tree cover loss. Subsequently, he proposed an interactive network-based visualization tool to provide estimates of illicit transactions and overreported trade data. The study explored effective techniques for data preprocessing and transaction data analysis.

2.4 LLMs and Code Generation

Inspired by implementing deep learning techniques for automatic code generation, Jia Li et al. proposed a sketch-based code generation approach, SkCoder, that can imitate engineers' code reuse behavior [17]. Upon receiving a natural language request by the user, SkCoder performs online scraping to search for a similar code snippet, then removes pertinent parts to create a 'code sketch' and modifies those parts to create the requested code. Although users without programming experience can successfully generate codes with SkCoder, the 'editing' feature is done automatically, which means the user is unable to provide feedback or steer the generated code and therefore must take care to provide a thorough request at the outset.

In order to enhance codes generated by pre-trained LLMs, Naman Jain et al. introduced Jigsaw, a tool designed to help language models understand program syntax and semantics and improve their performance based on user feedback [13]. By providing the LLM with a natural language string representing the user's request, along with samples of input-output data or test case examples, Jigsaw can generate a code snippet that solves the task. After the code is generated, Jigsaw then executes the code to ensure it passes all provided test cases and quality checks. The process is designed to correct frequent and recurrent errors in the code, including referencing errors, argument errors, and semantic errors. However, Jigsaw requires users to specify the exact column in the dataset to apply the generated codes, in addition to providing example output data. Thus, if a user makes accidental typos in column names or if the user request is imprecise, the resulting performance will degrade.

While LLMs are powerful at writing codes in response to user prompts, they still struggle with algorithmic challenges and typically require human verification. To tackle the issue, Kexun Zhang et al. presented ALGO, a framework that combines algorithmic programs with oracles generated to guide code generation and ensure accuracy [25]. After creating a reference oracle that consists of a list of correct but ponderous code programs, ALGO first asks the LLM to search through the oracle and select a code that solves the user request. Then, it employs a second LLM as a coder to produce a code proposal that performs faster (but possibly wrong). Both the coder's proposal and the oracle's search result go through the same test cases in order to compare outputs, and the coder further refines its code proposal if necessary. ALGO allows users to communicate with the LLM in English to generate codes, but they are not able to provide feedback or confirm whether the code can handle their request. This makes it challenging to customize generated codes, even if the user only desires to make minor adjustments.

Very recently (March 2024), Cognition unveiled Devin [4], their 'completely autonomous AI software engineer' that claims to manage the entire software development process. According to the company, Devin stands out with its ability to manage the entire software development process, from writing codes and addressing bugs to the final code execution. Cognition also claims that Devin is powerful in reading documents and extracting useful information, writing codes and test cases, executing codes, and fixing bugs automatically. This enables data analysts without programming experience to successfully generate data preprocessing codes and show example executions to the user. However, since Devin is not currently publicly accessible, we are unable to evaluate our approach against it. It is also unknown whether Devin can interact with massive datasets; For example, to automatically fix misspellings in a data column without the need to define a specific algorithm. Furthermore, it is also unclear whether Devin can be run locally; If not, then it might not be the best idea to give Devin crucial material since the information will not remain confidential.

Table 2.1: Feature comparison of SOTA methods vs. TradeSweep

	SkCoder	**Jigsaw**	**ALGO**	**Devin**	**TradeSweep**
Fully-English communication	✓		✓	✓	✓
Simple request description				✓	✓
Improve code by feedback		✓		✓	✓
Novel code generation	✓	✓	✓	✓	✓
Ongoing code library update					✓
Visualized execution demo		✓			✓

While SkCoder, Jigsaw, ALGO, and Devin are all powerful code-generation tools, we have discovered certain limitations in each. Therefore, our goal is to develop an agent to solve these issues. Without requiring extensive programming knowledge or providing specific details like column names or algorithms in the request, TradeSweep helps users generate code and enhance it through English-language interactions. TradeSweep produces novel code functions and stores them in a code library, aiming for continuous optimization. Lastly, TradeSweep provides users with visualized execution results on sample data to confirm the validity of the code. Table 2.1 depicts a comparison between TradeSweep and the systems surveyed above.

Chapter 3

TradeSweep

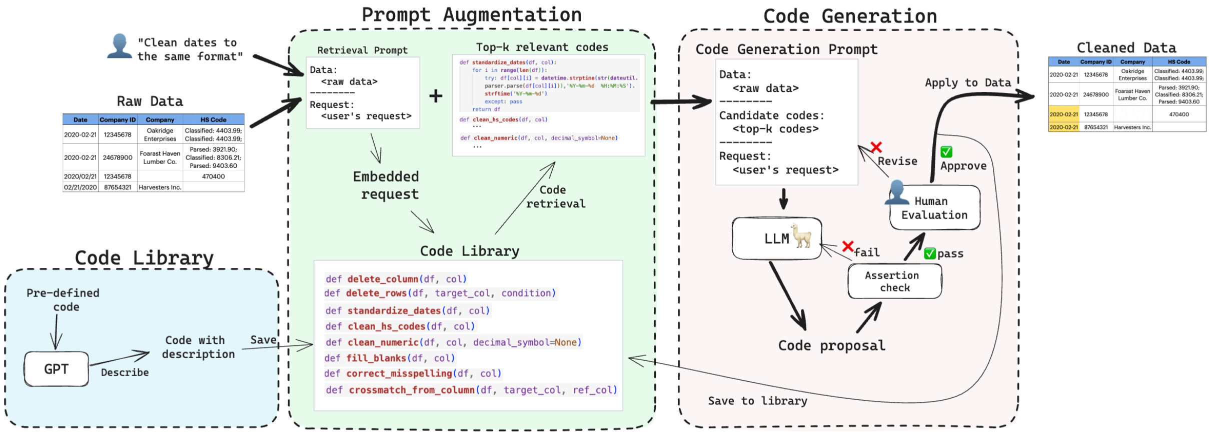
The goal of TradeSweep is to utilize LLM's code-writing abilities to produce executable programs for data preprocessing tasks. Users are only required to provide a dataset in either CSV or Excel format and enter a preprocessing request. As depicted in Fig. 3.1, TradeSweep consists of three primary components:

- **Prompt augmentation**: we employ information retrieval techniques to select the top-k relevant codes from the code library. (Chapter 3.2)
- **Code generation**: the LLM generates a code proposal along with visualized samples of execution results, awaiting user feedback. (Chapter 3.3)
- **Code library**: we build a reference document that includes classic Python scripts for data preprocessing tasks. (Chapter 3.4)

3.1 Problem Definition

Let $\mathcal{D}$ represent a table with n rows and m columns, with column names $c_1, c_2, ..., c_m$ (this set of column names is denoted as $\mathcal{C}$). Each element value x_{ij}, where $i \in \{1, ..., n\}$ and $j \in \{1, ..., m\}$, represents the value of the j-th feature of the i-th data record.

Typically, in practical applications, it is necessary for individuals to perform cleaning (e.g.,



Raw Data

Date	Company ID	Company	HS Code
2020-02-21	12345678	Oakridge Enterprises	Classified: 4403.99; Classified: 4403.99;
2020-02-21	24678900	Foarast Haven Lumber Co.	Parsed: 3921.90; Classified: 8306.21; Parsed: 9403.60
2020/02/21	12345678		470400
02/21/2020	87654321	Harvesters Inc.	

```
Data:
  <raw data>
--------
Request:
  <user's request>
```

```
def standardize_dates(df, col):
    for i in range(len(df)):
        try: df[col][i] = datetime.strptime(str(dateutil.
        parser.parse(df[col][i])),'%Y-%m-%d  %H:%M:%S').
        strftime('%Y-%m-%d')
        except: pass
    return df
def clean_hs_codes(df, col)
    ...
def clean_numeric(df, col, decimal_symbol=None)
    ...
```

```
def delete_column(df, col)
def delete_rows(df, target_col, condition)
def standardize_dates(df, col)
def clean_hs_codes(df, col)
def clean_numeric(df, col, decimal_symbol=None)
def fill_blanks(df, col)
def correct_misspelling(df, col)
def crossmatch_from_column(df, target_col, ref_col)
```

```
Data:
  <raw data>
--------
Candidate codes:
  <top-k codes>
--------
Request:
  <user's request>
```

Cleaned Data

Date	Company ID	Company	HS Code
2020-02-21	12345678	Oakridge Enterprises	Classified: 4403.99; Classified: 4403.99;
2020-02-21	24678900	Foarast Haven Lumber Co.	Parsed: 3921.90; Classified: 8306.21; Parsed: 9403.60
2020-02-21	12345678		470400
2020-02-21	87654321	Harvesters Inc.	

Figure 3.1: Overview of TradeSweep

inconsistent formats, outliers, missing values) and preprocessing (e.g., normalization, label encoding) on data $\mathcal{D}$ before applying an ML model. Let $\mathcal{A}$ be a subset of $\{c_1, c_2, ..., c_m\}$, which denotes the collection of columns that require some form of processing.

TradeSweep ($\mathcal{M}$) aims to receive a preliminary description r of the user's data preprocessing requirements in English and generate a Python code $\hat{f}$ and a processed dataset $\hat{\mathcal{D}}$ automatically.

Code Generation (Chapter 3.3): In order to achieve this goal, we begin by constructing a prompting curriculum denoted as $\mathcal{P} = [r, \mathcal{C}]$. This curriculum is then inputted into an LLM as a prompt to develop a Python code that satisfies the specified requirements r. Using the code created by TradeSweep (code proposal $\mathcal{G}$), we will additionally execute and evaluate $\mathcal{G}$ on a subset of $\mathcal{D}$ to obtain an execution outcome $\mathcal{O}$ that can be presented to the user so that they can validate the code proposal without any programming expertise. It should be noted that in the aforementioned process, the specific columns $\mathcal{A}$ that require modification are automatically determined by the model $\mathcal{M}$.

Avoiding LLM hallucination: In order to address hallucination within the context of automated data preprocessing, we expanded the existing curriculum $\mathcal{P} = [r, \mathcal{C}]$ into $\mathcal{P} = [r, \mathcal{C}, \mathcal{F}]$, where $\mathcal{F}$ denotes a finite set of closely interconnected fundamental functions that serve as a reference for LLM. It is important to recognize that this offers a dual advantage: 1) The occurrence of hallucinations is decreased; 2) LLMs can develop new code instead of relying solely on the fundamental reference function. Our solution utilizes an adaptable code library $\mathcal{L}$ to implement the aforementioned design (in Chapter 3.2). Initially, the request r will be transformed into an embedding representation emb_r. Then, a set of k relevant fundamental functions $\mathcal{F} = \{f_1, ..., f_k\}$ will be extracted from the code database $\langle \text{emb}_i, \text{function}_i \rangle$ based on the minimum cosine similarity $\min_{i \in L} \text{cosine}(\text{emb}_r, \text{emb}_i)$. Once a new code proposal $\mathcal{G}$ is successfully developed to meet complex requirements r, typically through interaction with humans, it is then saved in the library $\mathcal{L}$. This allows the library $\mathcal{L}$ to continually learn and provide more accurate and advanced reference functions in the future.

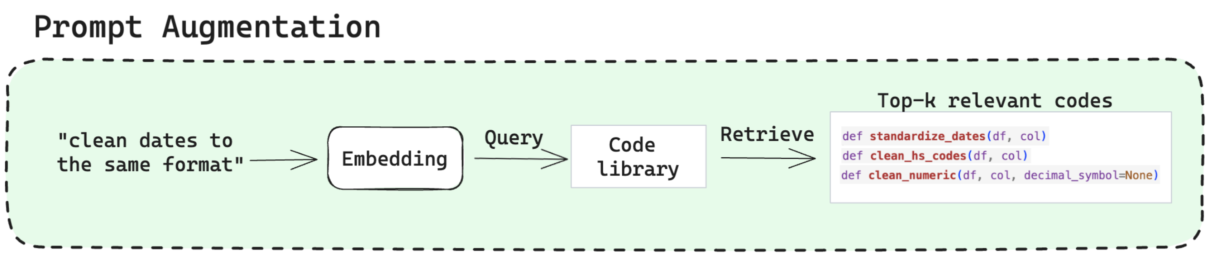
Human Feedback: Given the user's limited programming skills, we present the code $\mathcal{G}$ along with an execution result $\mathcal{O}$ to simplify the user's task of verifying the code. This allows the user to focus on determining if the processed data aligns with their domain knowledge of the data, rather than examining the code in detail. If the intended outcome is not achieved, TradeSweep will generate a continual conversation $\mathcal{P} = [r', \mathcal{C}, \mathcal{F}]$, where r' represents a revision need. In the Results (Chapter 4.4), we demonstrate how TradeSweep achieves a high probability of passing on all examined datasets on the first attempt and adapts faster than baselines when revisions are required.

3.2 Prompt Augmentation

Following the submission of the user's request for data preprocessing, TradeSweep examines all functions contained within the code library to learn and utilize them as references (prompt lengths are minimized using information retrieval techniques).

The code library is made up of code functions that are commonly applied in various preprocessing tasks. In addition to the functions, data information is also prompted to the LLM to facilitate its comprehension of the dataset structure. However, inputting both the entire code library and data information could lead to a lengthy prompt containing redundant information, not only reducing the LLM's performance in accurately finding the most corresponding function, but also significantly delaying the response time of the LLM. Therefore, it is usually a good idea to shorten the input context. Instead of presenting all code functions in the prompt, we employ information retrieval techniques to perform a more accurate selection of code functions and provide the LLM with the most relevant and beneficial codes to learn from.

The approach of our prompt augmentation involves embedding the user request and querying it in the code library, which is represented as a vector database. As a result, the vector database compares the embedded request with its vectors, which are embedded descriptions of code functions. It then retrieves the top-k code functions that are most closely associated with fulfilling the user's request. Finally, these retrieved code functions are integrated with the user's request and data information to form a prompt for the LLM to generate code. This process not only effectively shortens the prompt length and LLM response time but also assists the LLM in concentrating on functions that are most related to achieving the user's request. (see Fig. 3.2)



```
def standardize_dates(df, col)
def clean_hs_codes(df, col)
def clean_numeric(df, col, decimal_symbol=None)
```

Figure 3.2: Prompt augmentation flowchart of TradeSweep

3.3 Code Generation

Once the top-k relevant codes have been retrieved as candidate codes, a prompt for the LLM is created by combining the candidate codes with data information and the user's request. The data information is collected based on the dataset provided by the user; It can be any type of information that aids the LLM in understanding the data structure, such as a list of column names or a limited number of data rows as samples.

The LLM examines the provided information and produces a code proposal that is expected to accomplish the requested task. If the LLM does not identify any candidate code that seems to align with the user's request, it produces a novel code function. In addition, the process of code generation will perform iterative code enhancements in the following scenarios after a code proposal is generated:

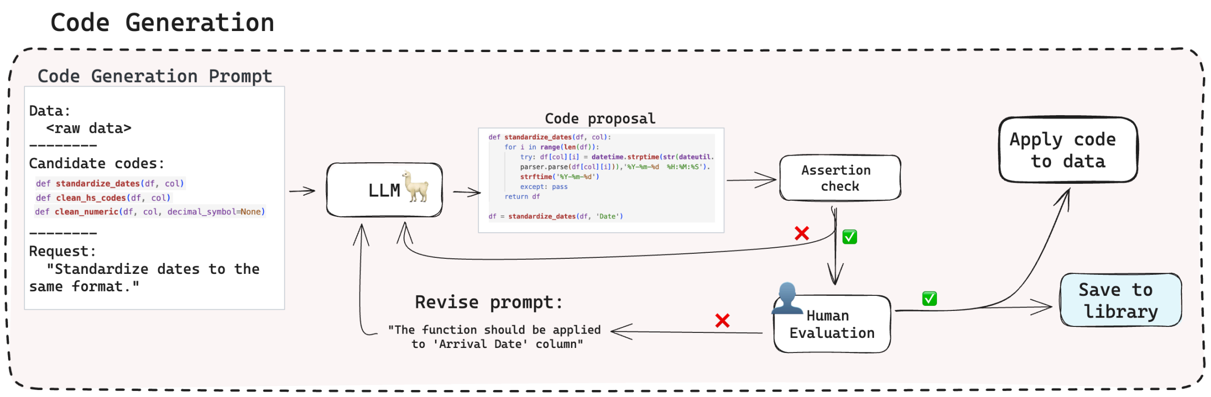
1. **Incorrect code proposal format**: We provided the LLM with a response template, asserting that the complete code proposal should consist of a Python code function that reads in a dataframe and returns it at the end, and a call function for applying the code to the data. If either one is not generated, the LLM automatically rolls back to re-generate.

2. **Execution error**: Once a code proposal is generated in the correct format, the codes

are tested on some sample data. If the execution fails for any reason, including bugs, syntax errors, exceptions, or other factors, both the code and its corresponding error message are prompted back to the LLM for a revision of the generated code.

3. **User's feedback**: In addition to the code proposal, users are also provided with a visualized execution result of sample data. This result displays some examples of input values and their corresponding output values if the code is applied. Following the user's observation of the code proposal and execution outcomes, the user can ask for code revision by giving feedback that describes a fix request to the LLM.

Finally, if the user suggests that both the code proposal and execution examples are correct, the code will be applied to the target dataset, as shown in Fig. 3.3. For situations in which the generated code is novel or when the user requests to save multiple functions into a single pipeline, we add this newly generated code or series of codes to our code library.



```
Data:
  <raw data>
--------
Candidate codes:
 def standardize_dates(df, col)
 def clean_hs_codes(df, col)
 def clean_numeric(df, col, decimal_symbol=None)
--------
Request:
  "Standardize dates to the
same format."
```

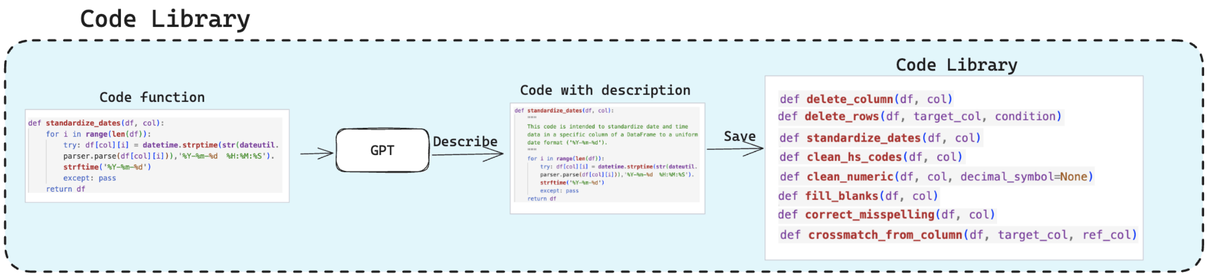
```
def standardize_dates(df, col):
    for i in range(len(df)):
        try: df[col][i] = datetime.strptime(str(dateutil.
        parser.parse(df[col][i])),'%Y-%m-%d  %H:%M:%S').
        strftime('%Y-%m-%d')
        except: pass
    return df

df = standardize_dates(df, 'Date')
```

"The function should be applied to 'Arrival Date' column"

Figure 3.3: Code generation flowchart of TradeSweep

3.4 Code Library

Due to complicated data preprocessing tasks, such as those that require specific algorithms or involve multiple datasets, a code library is beneficial to serve as a reference document for the LLM to learn and follow when generating code proposals. Within TradeSweep 's code library, each function is responsible for a certain data preprocessing task. Each code function also contains comments describing its usage.



```
def standardize_dates(df, col):
    for i in range(len(df)):
        try: df[col][i] = datetime.strptime(str(dateutil.
        parser.parse(df[col][i])),'%Y-%m-%d  %H:%M:%S').
        strftime('%Y-%m-%d')
        except: pass
    return df
```

```
def standardize_dates(df, col):
    """
    This code is intended to standardize date and time
    data in a specific column of a DataFrame to a uniform
    date format ('%Y-%m-%d').
    """
    for i in range(len(df)):
        try: df[col][i] = datetime.strptime(str(dateutil.
        parser.parse(df[col][i])),'%Y-%m-%d  %H:%M:%S').
        strftime('%Y-%m-%d')
        except: pass
    return df
```

```
def delete_column(df, col)
def delete_rows(df, target_col, condition)
def standardize_dates(df, col)
def clean_hs_codes(df, col)
def clean_numeric(df, col, decimal_symbol=None)
def fill_blanks(df, col)
def correct_misspelling(df, col)
def crossmatch_from_column(df, target_col, ref_col)
```

Figure 3.4: Code library maintenance of TradeSweep

To enhance the efficacy of TradeSweep , we also developed a code library capable of supporting the addition of new functions as interaction progresses. New functions shall be added to the library in the following circumstances:

1. **Novel code generation**: When the LLM does not identify any candidate code that corresponds to the user's request after analyzing the retrieved functions, it develops a novel code proposal. Given that the newly generated code is not included in the library, we incorporate it back into the library to enhance efficiency and accuracy for future code generations.

2. **Pipeline creation request**: After a series of code functions have been generated, the user can request that the entire procedure be stored as a pipeline. During this process, the several functions that were previously applied to the dataset are combined into a

single code function, which is then added to our code library.

As demonstrated in Fig. 3.4, in order to save a new code into the code library, we use the LLM to write a description of the code and add it as comments. Afterward, both the new code function and the description are added back into our vector database code library, where a new query vector is constructed using the function description, and its corresponding payload is generated by the code.

Chapter 4

Evaluation Methodology

In this chapter, we explain how we setup the environment for experiments, as well as some baselines designed to be compared and evaluated against TradeSweep .

4.1 Experimental Setup

Data: For this study, we used shipment-level bill of lading data [8] that captures business-to-business international trade and highlight the complexity of supply chains that enable our globalized world to operate. The United States, like all countries, uses trade policy, tariffs, and trade sanctions to enforce objectives related to foreign policy and national security, as well as other strategic goals such as detecting and deterring trade in illegally harvested or produced products. The ability to identify specific shipments that may be in contravention of economic sanctions, have high tariff rates, or be consistent with suspicious or illegal activity depends largely on how well the data is initially cleaned and preprocessed. Due to the complexity of global trade and the incentives that companies face to reach destination markets and circumvent sanctions in the aforementioned transactions, there is potential for the involvement of third-party entities that may be located in neither the initial source nor the final destination country.

Specifically, we investigate three datasets that pertain to shipments involving imports and exports across multiple nations for commodities that have been subject to recent import prohibitions, high tariff rates, and sanctions, and all may contain risks associated with origin fraud [2, 3, 5]. The datasets are given as follows:

1. **Teak**: contains 69,134 teakwood transactions exporting from 116 countries to the United States, spanning from July 1, 2007, to August 10, 2023 (5,885 days in total). (According to the source from Panjiva [20])

2. **Grain**: consists of 145,217 grain transactions from Russia to 118 global destinations, starting from May 20, 2021, to November 30, 2022 (560 days in total). (According to sources ExportGenius [6] and ImportGenius [11])

3. **Timber**: contains 3,087,822 timber exports from Russia to 173 countries, commencing from October 20, 2021, to March 31, 2023 (528 days in total). (According to sources ExportGenius and ImportGenius)

These transaction data span a considerable amount of time and were manually entered, making them prone to errors such as typos, inconsistent formatting, missing information, and other potential issues. In this case, examining data becomes a laborious and time-consuming task for analysts, necessitating the urgency and importance of preprocessing the data.

Vector Database: Vector databases are a popular method of interacting with data representations, commonly referred to as vectors or embeddings. The utilization of these emerging databases is experiencing significant growth and is widely adopted in various applications, including semantic search and recommendation systems. In TradeSweep , we use Qdrant, a widely recognized and rapidly expanding vector database, to serve as the information retrieval system. Qdrant is a vector similarity search engine that offers users an API for storing, searching, and managing data. In comparison to other databases available on the market, it outperforms many of its competitors in terms of query speed and retrieval accuracy. In addition, it enables customers to deploy the system locally. Given our need to maintain the confidentiality of our transaction data, Qdrant is an ideal option for us.

Large Language Model: LLMs have recently shown remarkable proficiency in natural language processing tasks as well as in other domains. In addition, the development of code generation in LLMs is experiencing rapid growth. Nowadays, there is a vast selection of robust models that are capable of efficient code generation. Among the various kinds of models, we employed CodeLlama-Instruct [19] in our study and utilized it to learn from retrieved codes, generate executable Python functions, and modify codes in accordance with user requests. Unlike API-based models like ChatGPT, CodeLlama has the benefit of enabling customers to execute the model locally. Due to the fact that it guarantees the security of the data provided to the LLM, this aspect is quite important in our study.

Initial Code Library: We established our initial code library with a selection of 13 widely recognized and commonly used data preprocessing functions. Moreover, the functions we adopted are used for general data preprocessing, which means these functions can be applied to any area of data that the user inputs and are not limited to trade interactions.

Listed in Table 4.1, the functions we implemented in our initial code library include standardizing date formats, removing punctuation marks, correcting typos, filling in missing values based on another column, etc.

Hardware Environment: Our work is carried out using a Tesla P40 NVIDIA driver, which is equipped with 8 cores, 38 GB of RAM, and 500 GB of disk memory.

Table 4.1: Some of the functions included in our initial code library

Task	Explanation
Remove columns	Delete an unwanted column.
Remove rows	Delete rows that satisfy a certain condition.
Clean numbers	Remove non-numeric symbols and convert the value to numbers.
Clean strings	Remove punctuation marks, quotation marks, and any extra spaces.
Clean dates	Standardize all date values to a YYYY-mm-dd format.
Correct misspellings	Apply word-embedding and clustering to a column to cluster similar values. Then, in each cluster group, find the most frequent value and correct others to that.
Compare columns and clean	Between a to-clean column and a reference column, group the two columns. Then, for all to-clean values that have the same reference value, find the most frequent to-clean value and update others on this.
Lookup document	Given a target column in the dataset and an external CSV/Excel document, map the target values to a reference column in the document, then create a new column with the mapped values.

4.2 Baselines

In order to determine the efficiency of TradeSweep in generating meaningful codes, we developed three baselines to be used for comparison:

4.2.1 Baseline 1: State-of-the-Art (SOTA) simulation - Code generation purely with LLM

To compare the effectiveness of TradeSweep in achieving our primary objective, code generation, we have abstracted the code writing components of SOTA tools (SkCoder, Jigsaw, ALGO, etc.) and designated them as Baseline 1. This abstraction was needed in order to accommodate the various usages and features of these tools. During this stage, we solely rely on the LLM for code generation without applying other augmentations. In the absence of the code library as a reference document, the LLM is required to produce code independently. In this case, the user's description of the request is the most crucial component. It is possible that modifying the code proposal will require a considerable amount of effort, and the result code may not clean the data as successfully as expected.

4.2.2 Baseline 2: LLM is prompted with candidate codes and the user's request

In TradeSweep , we incorporate data information, the top-k candidate codes, and the user's request to be inputted into the LLM. By providing data information, the LLM can adjust its code according to the actual data. Otherwise, the LLM is entirely dependent on prompts and can only make guesses regarding the columns or data that the user is referring to. Therefore, we designed baseline 2, where the LLM is only given the candidate codes and the user's request. We assume that the absence of data information available for the LLM may in some way have an impact on its performance.

4.2.3 Baseline 3: Provide the entire code library to the LLM without code descriptions

As the candidate codes are provided to the LLM, they are accompanied by their function descriptions, formatted as comments. During the process of examining the descriptions, the LLM can gain a deeper understanding of the candidate codes, enabling it to generate a code proposal that employs a similar algorithm. So, in baseline 3, our objective is to investigate the impact of eliminating function descriptions on the code generation results. Additionally, since our vector database retrieves candidate codes by comparing the user's request to code descriptions, we have also eliminated the prompt augmentation step of retrieving relevant codes. Rather than giving k candidate codes, we now provide the entire code library (without descriptions) into the LLM as part of the prompt.

4.3 Evaluation

For each dataset, we consider various data preprocessing tasks depending on its content, which are outlined in Table 4.2.

Table 4.2: Preprocessing tasks performed in the three datasets.

	Teak	**Grain**	**Timber**
Remove columns	✓		✓
Clean dates	✓	✓	✓
Clean numbers	✓		✓
Clean strings		✓	✓
Correct misspellings	✓		✓
Compare columns and clean	✓	✓	✓
Lookup documents		✓	✓
Others (not in library)	✓	✓	✓

Each task is evaluated with each of the baselines as well as TradeSweep . To conduct evaluation, we input data preprocessing requests to each baseline and TradeSweep to produce codes for all three datasets. All user requests sent to the baselines and TradeSweep are identical when performed on the same data. We record the first and final versions (in the event of code revision) of generated codes, along with the amount of time it took to generate each code and the number of revisions that were carried out. After all codes have been generated, they are then applied to our initial uncleaned data. The execution outputs are subsequently compared to manually preprocessed data in order to assess the performance of each baseline.

A task could be applied to multiple columns in each dataset; In this case, we perform the same task repeatedly and ask the LLM to generate codes, each time using the same algorithm but adopting small changes, to tailor the code to certain columns in the dataset.

4.4 Results

From the codes generated by all baselines and TradeSweep , we consider the following research questions:

4.4.1 RQ1: Can TradeSweep automatically preprocess data in an accurate way?

In this chapter, we denote our initial uncleaned data as Init and the data that was manually preprocessed as Ground Truth (GT) for simplicity.

Following the generation of the codes by baselines and TradeSweep , the functions are then applied to the initial data for data preprocessing. As shown in Table 4.3, the execution results obtained from TradeSweep were the most similar to Ground Truth (with Baselines 2 and 3 yielding similar outcomes to TradeSweep as a result of referencing and learning from the code library for code generation). On the other hand, applying codes generated by Baseline 1 leads to outputs that differ significantly from GT; this disparity can be attributed to the LLM writing functions independently in the baseline, which ultimately employed different algorithms than the code library to accomplish the tasks.

After executing all generated codes by baselines and TradeSweep to the datasets, the overview of data preprocessing performances of Baseline 1, an abstracted version of SOTA methods, is represented in Table 4.4. Compared to Baseline 1, TradeSweep is fully capable of generating

Table 4.3: Overall data cleaning performance of TradeSweep vs. Baselines

		Init	GT	Baseline 1	Baseline 2	Baseline 3	TradeSweep
Teak	# of cols	127	26	26	26	26	26
	# of rows				69,134		
	NaNs (%)	48.77	7.57	10.94	7.09	7.09	**7.09**
	incorrect formats (%)	19.99	0	29.98	9.86	1.03	**1.03**
	typos (%)	9.29	0	25	3.92	3.92	**3.92**
Grain	# of cols	56	61	61	61	61	61
	# of rows				145,217		
	NaNs (%)	66.94	62.10	3.76	**63.07**	66.22	63.57
	incorrect formats (%)	2.84	0	9.68	0.09	0.08	**0.08**
	typos (%)	18.06	0	14.43	**2.37**	6.65	3.21
Timber	# of cols	29	23	23	23	23	23
	# of rows				3,087,822		
	NaNs (%)	3.19	4.55	8.68	4.75	8.24	**4.42**
	incorrect format (%)	87.93	0	64.86	0.63	12.81	**0.63**
	typos (%)	93.19	0	74.69	4.04	6.83	**2.76**

suitable and executable codes that clean the data to have a similar correct-value rate as Ground Truth.

4.4.2 RQ2: To what extent can TradeSweep independently generate valid code proposals?

In this chapter, we perform experiments to evaluate the ability of TradeSweep to produce valid codes without the need for user feedback.

This research question is evaluated by the user's acceptance of the code generated by the LLM on its first attempt. The rate of codes approved during LLM's first attempt is recorded in Table 4.5. Each value denotes the number of codes that were accepted by the user on the

Table 4.4: An overview of the data cleaning performance of TradeSweep vs. Baseline 1 (abstracted SOTA method).

		Teak	**Grain**	**Timber**
	Init correct-rate	86.60%	85.94%	9.09%
	GT correct-rate	100% (time-consuming)		
SOTA	correct-rate	73.09%	87.65%	29.58%
	first-attempt valid rate	41.66%	27.78%	62.5%
TS	correct-rate	**97.19%**	**97.61%**	**98.17%**
	first-attempt valid rate	83.33%	66.67%	75%

LLM's first try, divided by the total number of accepted codes.

TradeSweep successfully generated the most user-acceptable codes in all three datasets without requiring any further revision. The lowest acceptance rate was observed in Baseline 2 as a result of the absence of column name information inputted into the LLM. Thus, despite employing the identical algorithm as TradeSweep in the produced codes of Baseline 2, the functions were applied to incorrect columns, and users were compelled to request revisions and specify the desired column names.

Table 4.5: First-version code acceptance rate of TradeSweep vs. Baselines

	Baseline 1	Baseline 2	Baseline3	TradeSweep
Teak	5/12	1/12	9/12	**10/12**
Grain	5/18	5/18	12/18	**12/18**
Timber	15/24	5/24	17/24	**18/24**

4.4.3 RQ3: When user feedback becomes necessary, how many rounds of feedback are required to generate a valid code proposal?

If the generated code does not fulfill the user's requirements for the desired task, we invoke the LLM again to make additional modifications to the code. The average number of iterations that were required in these cases and the average duration (in seconds) needed are recorded in Table 4.6 and Table 4.7.

Table 4.6: The average number of iterations required to generate valid code proposal for TradeSweep vs. Baselines

	Baseline 1	Baseline 2	Baseline3	TradeSweep
Teak	1.92	2.08	1.42	**1.16**
Grain	2.11	2.17	1.39	**1.33**
Timber	1.5	2.13	1.42	**1.29**

Table 4.7: The average time (in seconds) required to generate a valid code proposal for TradeSweep vs. Baselines.

	Baseline 1	Baseline 2	Baseline3	TradeSweep
Teak	134.72	141.15	555.46	167.55
Grain	104.87	137.43	369.99	117.40
Timber	42.71	134.65	269.34	94.08

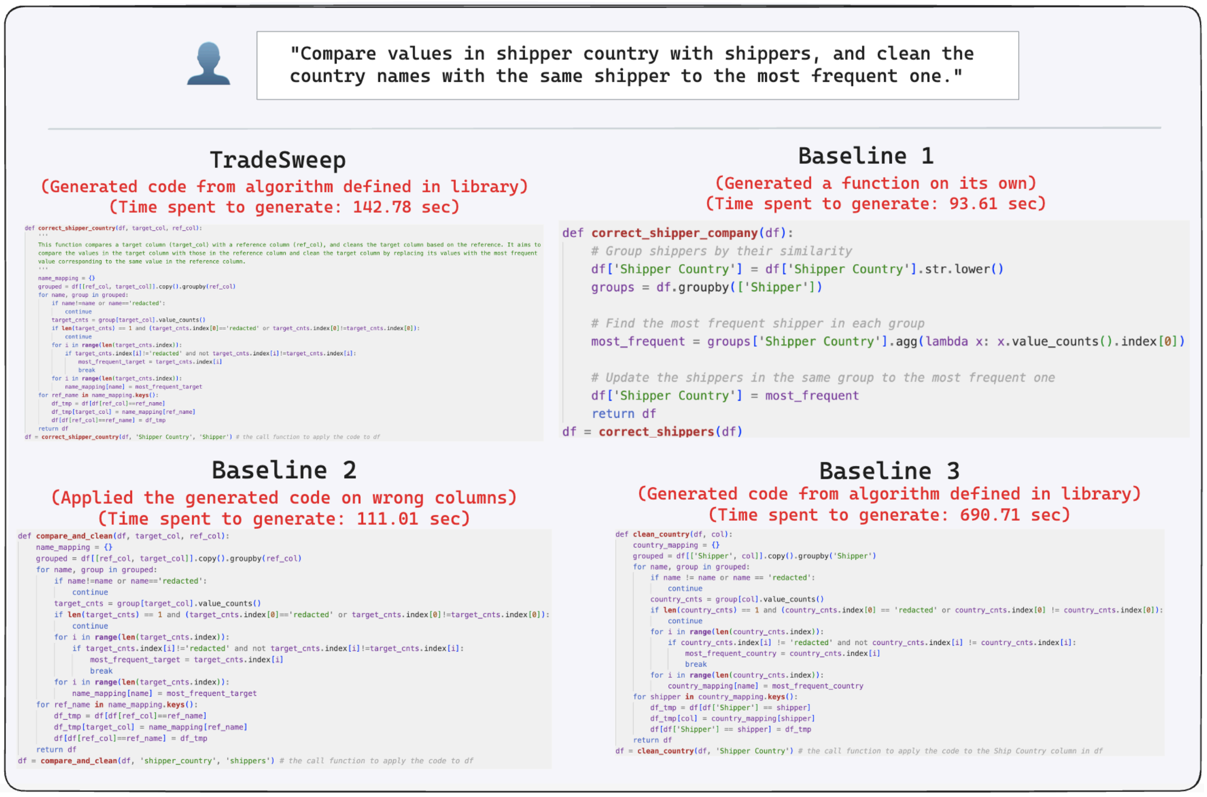
Based on the results, TradeSweep is capable of generating codes that successfully meet the user's demands at the LLM's initial attempt, achieved by utilizing knowledge from the code library. On the other hand, Baseline 1 produces unstable outcomes on the first attempt since the LLM generated codes independently; several codes were incapable of meeting the user's vague request (e.g., "Clean the numbers in net weight") and required the user to give a more explicit explanation (e.g., "Clean the numbers in net weights: remove commas, then convert the string value to float numbers"). Moreover, due to the absence of column names in Baseline 2, nearly all preprocessing tasks required revision; It was essential to specify accurate column names to ensure the functionality of the generated code (for example, the prompt should have read "Clean numbers in the NetWeight column" instead of "Clean numbers in weights").

While Baseline 3 could also generate suitable codes on the LLM's first attempt, it needed slightly more user involvement than TradeSweep . We observed that in this baseline, although the LLM selected the correct code to modify, the lack of function descriptions led the LLM to over- or under-modify the reference code, necessitating further revision requests. Moreover, this baseline took a considerably longer time to generate a code compared to the other two baselines and TradeSweep . This was due to providing all codes in the library to the LLM rather than inputting the most relevant ones. The length of the prompt sent to the LLM exceeded that of other baselines, resulting in a much slower process and a greater expenditure of time.

4.4.4 Case Study: TradeSweep vs. Baselines

To gain a more comprehensive understanding of the disparity in code generation performance between various baselines and TradeSweep , we analyze and compare the initial code proposals produced (before the user provides any feedback) in each scenario while ensuring all baselines were given the same user request input for each preprocessing task.

Using the example of cleaning the column "Shipper Country", Fig. 4.1 illustrates the code proposal generated by the three baselines and TradeSweep . In this experiment, the user's request was inputted as "Compare values in shipper country with shippers, and clean country names with the same shipper to the most frequent value." The ideal procedure is to enable the LLM to employ the "compare_and_clean" function from the code library and apply it to the "Shipper Country" column while comparing with the "Shipper" column.



"Compare values in shipper country with shippers, and clean the country names with the same shipper to the most frequent one."

TradeSweep

(Generated code from algorithm defined in library)
(Time spent to generate: 142.78 sec)

```python
def correct_shipper_country(df, target_col, ref_col):
    '''
    This function compares a target column (target_col) with a reference column (ref_col), and cleans the target column based on the reference. It aims to
    compare the values in the target column with those in the reference column and clean the target column by replacing its values with the most frequent
    value corresponding to the same value in the reference column.
    '''
    name_mapping = {}
    grouped = df[[ref_col, target_col]].copy().groupby(ref_col)
    for name, group in grouped:
        if name!=name or name=='redacted':
            continue
        target_cnts = group[target_col].value_counts()
        if len(target_cnts) == 1 and (target_cnts.index[0]=='redacted' or target_cnts.index[0]!=target_cnts.index[0]):
            continue
        for i in range(len(target_cnts.index)):
            if target_cnts.index[i]!='redacted' and not target_cnts.index[i]!=target_cnts.index[i]:
                most_frequent_target = target_cnts.index[i]
                break
        for i in range(len(target_cnts.index)):
            name_mapping[name] = most_frequent_target
    for ref_name in name_mapping.keys():
        df_tmp = df[df[ref_col]==ref_name]
        df_tmp[target_col] = name_mapping[ref_name]
        df[df[ref_col]==ref_name] = df_tmp
    return df
df = correct_shipper_country(df, 'Shipper Country', 'Shipper') # the call function to apply the code to df
```

Baseline 1

(Generated a function on its own)
(Time spent to generate: 93.61 sec)

```python
def correct_shipper_company(df):
    # Group shippers by their similarity
    df['Shipper Country'] = df['Shipper Country'].str.lower()
    groups = df.groupby(['Shipper'])

    # Find the most frequent shipper in each group
    most_frequent = groups['Shipper Country'].agg(lambda x: x.value_counts().index[0])

    # Update the shippers in the same group to the most frequent one
    df['Shipper Country'] = most_frequent
    return df
df = correct_shippers(df)
```

Baseline 2

(Applied the generated code on wrong columns)
(Time spent to generate: 111.01 sec)

```python
def compare_and_clean(df, target_col, ref_col):
    name_mapping = {}
    grouped = df[[ref_col, target_col]].copy().groupby(ref_col)
    for name, group in grouped:
        if name!=name or name=='redacted':
            continue
        target_cnts = group[target_col].value_counts()
        if len(target_cnts) == 1 and (target_cnts.index[0]=='redacted' or target_cnts.index[0]!=target_cnts.index[0]):
            continue
        for i in range(len(target_cnts.index)):
            if target_cnts.index[i]!='redacted' and not target_cnts.index[i]!=target_cnts.index[i]:
                most_frequent_target = target_cnts.index[i]
                break
        for i in range(len(target_cnts.index)):
            name_mapping[name] = most_frequent_target
    for ref_name in name_mapping.keys():
        df_tmp = df[df[ref_col]==ref_name]
        df_tmp[target_col] = name_mapping[ref_name]
        df[df[ref_col]==ref_name] = df_tmp
    return df
df = compare_and_clean(df, 'shipper_country', 'shippers') # the call function to apply the code to df
```

Baseline 3

(Generated code from algorithm defined in library)
(Time spent to generate: 690.71 sec)

```python
def clean_country(df, col):
    country_mapping = {}
    grouped = df[['Shipper', col]].copy().groupby('Shipper')
    for name, group in grouped:
        if name != name or name == 'redacted':
            continue
        country_cnts = group[col].value_counts()
        if len(country_cnts) == 1 and (country_cnts.index[0] == 'redacted' or country_cnts.index[0] != country_cnts.index[0]):
            continue
        for i in range(len(country_cnts.index)):
            if country_cnts.index[i] != 'redacted' and not country_cnts.index[i] != country_cnts.index[i]:
                most_frequent_country = country_cnts.index[i]
                break
        for i in range(len(country_cnts.index)):
            country_mapping[name] = most_frequent_country
    for shipper in country_mapping.keys():
        df_tmp = df[df['Shipper'] == shipper]
        df_tmp[col] = country_mapping[shipper]
        df[df['Shipper'] == shipper] = df_tmp
    return df
df = clean_country(df, 'Shipper Country') # the call function to apply the code to the Ship Country column in df
```

Figure 4.1: Comparison of an accepted code proposal of TradeSweep and Baselines

TradeSweep: In TradeSweep, the top 3 relevant functions were retrieved from the code library and inputted into the LLM. The code proposal demonstrates that the LLM effectively learned from the "compare_and_clean" function in the code library. Furthermore, by inputting the data column names into the LLM, TradeSweep is able to determine the precise columns that the user wishes to apply the function to, even when not providing accurate names. In this scenario, the desired columns are "Shipper" and "Shipper Country". In the proposed code, the function groups the data by the "Shipper" column. Then, within each group, it identifies the most frequent value of "Shipper Country", and modifies the other shipper countries so that they reflect this value.

Baseline 1: While the user's request remains consistent across all baselines, Baseline 1, which lacks knowledge of the code library, assumed the user's intended meaning of ”compare and clean”. Although the code proposed by Baseline 1 successfully grouped the columns and determined the most frequent shipper country, it erred by converting all strings to lowercase and failing to account for potential scenarios where the most frequent value found was NaN. Consequently, the resulting cleaned data exhibited a higher prevalence of NaNs and wrongly formatted country names. On the other hand, TradeSweep was able to successfully acquire the function defined in the code library; This function identifies the most frequent non-NaN country value, thereby filling in missing values in Shipper Country without altering its string format.

Baseline 2: In Baseline 2, the candidate codes and the user's request are provided to the LLM, while data information is not inputted. The generated code proposal shows that Baseline 2 successfully obtained the ”compare_and_clean” function from the code library and performed adjustments following the user's request. Nevertheless, the function was unable to comprehend the particular columns that the user intended to apply due to the lack of data information. Instead of using actual column names, the LLM was only able to depend on the user's request and make assumptions about the column names, which ended up using ”shippers” and ”shipper_country”. In contrast, when data information is provided, like in TradeSweep , the LLM considers the column description provided by the user and selects the most suitable columns to apply the function to. In this case, the correct columns ”Shipper” and ”Shipper Country” were selected.

Baseline 3: Instead of providing only the top 3 relevant functions to the LLM, the entire code library is inputted in Baseline 3, but without function descriptions. By observing the generated results from this baseline, the LLM is capable of comprehending the usage of each given function, despite the absence of their descriptions. The LLM successfully utilized the

”compare_and_clean” function from the code library, which was then modified and applied to the appropriate columns based on understanding the user’s request. However, as a result of inputting all functions in the code library into the LLM, the prompt grew excessively long and contained redundant information. Consequently, Baseline 3 required a total of 698 seconds to read the given information and generate codes, which is significantly slower in comparison to other baselines. This further indicates that by minimizing the number of functions offered to the LLM, as in TradeSweep , we can effectively shorten the prompt and therefore expedite the process.

Chapter 5

Conclusions

The contributions of this thesis, TradeSweep , are as follows:

1. TradeSweep utilizes English conversations to understand and respond to users' requests with Python code encompassing preprocessing functions. This supports three main capabilities: 1) It produces new code when requested for a completely new task; 2) It can precisely modify its proposed code based on users' feedback in English; 3) The proposed code is automatically tailored to the target data, including accurate column names and suitable algorithms.

2. TradeSweep develops and continuously grows a library of fundamental data preprocessing codes by storing executable functions that have been successfully deployed previously. Augmentation of the code library supports the composition and creation of elaborate data pipelines.

3. To incorporate feedback from users who lack programming expertise, TradeSweep offers both code proposal and execution results on example data. This allows users to determine whether to accept TradeSweep 's proposal or to make modifications based on output data visualizations, instead of focusing on code specifics.

4. We perform extensive experiments on three trade datasets. Results show that TradeSweep is capable of generating executable and efficient codes for data preprocessing, and also significantly reduces time and effort for data analysts.

5.1 Summary of Results

The rise of automation and programming support by LLMs has rapidly decreased the turnaround time for end-users in processing massive spreadsheets. TradeSweep can be viewed as an LLM-driven data preprocessing agent that retrieves suitable functions from a code library and modifies them to achieve the data preprocessing task. Our results have demonstrated the effectiveness of TradeSweep in practical data transformation contexts. Our study conducted experiments on three transaction datasets to perform data cleaning, and the results indicate that TradeSweep can efficiently perform the requested task using minimal user interaction. The ablation study also demonstrates that the utilization of a vector database for information retrieval contributes to the enhancement of the speed of code generation. Furthermore, providing data information and function descriptions to the LLM facilitates the generation of more accurate codes without requiring user feedback for modification, enabling users with less programming proficiency to be effective.

5.2 Limitations

Though TradeSweep shows promising results in code generation performance compared to SOTA tools and baselines, there are also some limitations that await resolution.

First, when the user is performing a data preprocessing task where TradeSweep cannot find a related function in the code library, it generates a novel code. In this case, TradeSweep performs in the same way as SOTA tools and generates codes purely by the LLM, requiring the user to give more iterations of feedback and modification in order for the LLM to generate valid code functions.

Also, while TradeSweep responds with a code proposal faster and more accurately than other baselines, it still requires some time when the desired data preprocessing task is relatively complicated. Apart from changing hardware requirements, finding a solution to optimize the response time is important to enhance the user experience.

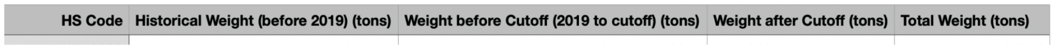
Furthermore, the sample execution might not be sufficient for users to evaluate the code when performing some tasks. For example, correcting misspellings in company names might lead to over- or under-correcting, and by observing a limited number of correction examples, users are not able to confirm whether the code successfully captures all corner cases. In this scenario, it is helpful to perform an apply-test on the target data and generate a comparison of names before and after correction (just like find-replace in word processing tools). However, applying the code to the full dataset takes much more time than only applying it to sample data; It is also important to speed up this process and provide much faster results for users.

5.3 Future Directions

In addition to continuing to improve TradeSweep 's performance and resolve the limitations mentioned above, in this chapter, we discuss what can be accomplished in the future for TradeSweep to provide a more user-friendly, efficient, and accurate data preprocessing experience.

5.3.1 Summary Tables

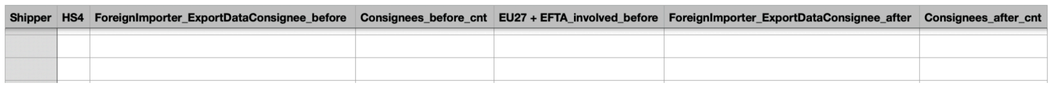
After preprocessing the data, we could use the LLM to write codes that generate summary documents based on some condition. For example, Fig. 5.1 shows an example of a weight summary table that records the transaction net weight of different HS codes during a specific period of time.



HS Code	Historical Weight (before 2019) (tons)	Weight before Cutoff (2019 to cutoff) (tons)	Weight after Cutoff (tons)	Total Weight (tons)

Figure 5.1: An example table of weight summary for different HS codes.

We can also use the LLM to generate tables for emerging trends. As shown in Fig. 5.2, the generated table will show information for each shipper and HS code. For each entity, we list consignee companies that interacted with the shipper before and after a specific date, then highlight those that appeared in both time frames.



Shipper	HS4	ForeignImporter_ExportDataConsignee_before	Consignees_before_cnt	EU27 + EFTA_involved_before	ForeignImporter_ExportDataConsignee_after	Consignees_after_cnt

Figure 5.2: An example table of emerging trends of shipper companies.

5.3.2 Data Visualization

Besides generating summaries and tables, providing visualization plots or figures for these results is often helpful for data analysts to examine the data more efficiently. Fig. 5.3 shows an example of distribution analysis; the plot represents the top 10 countries with the highest transaction amount (weight) for each month in the dataset. In Fig. 5.4, we draw a world map and connect two countries if the difference in their transaction amount (weight) with Russia is under a specific δ value.

5.3.3 Anomaly Detection

Finally, since one eventual outcome of trade analytics is to detect suspicious trades, it is also desirable to implement machine learning technology into TradeSweep . It would be beneficial to train ML models to detect and highlight anomaly entities, helping flag shipments for further scrutiny.

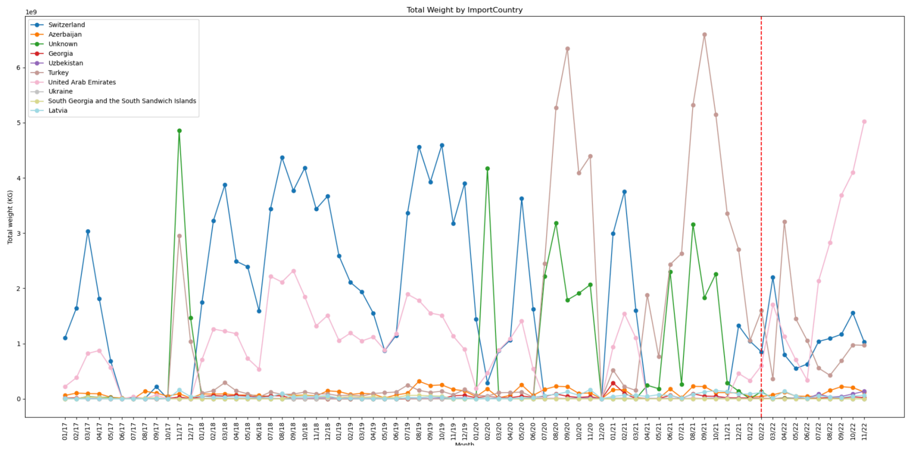


Figure 5.3: An example of distribution analysis.

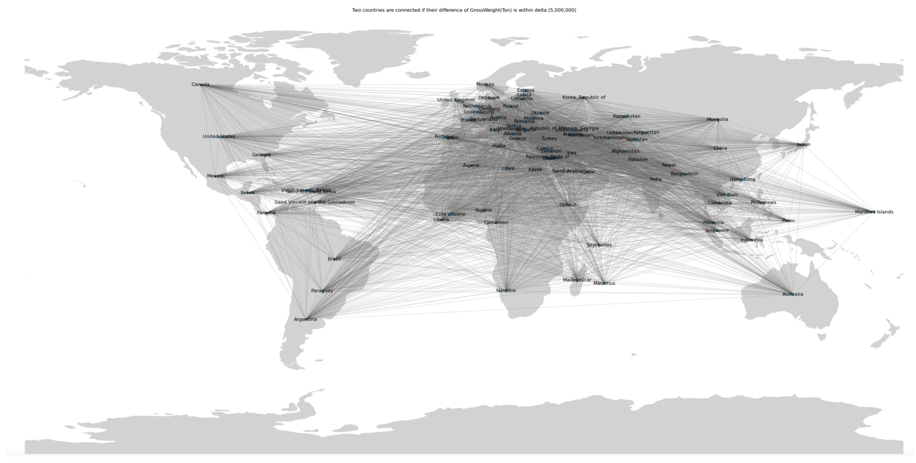


Figure 5.4: An example of network analysis.

Bibliography

[1] Sadhvi Anunaya. Data preprocessing in data mining: A hands on guide. URL https://www.analyticsvidhya.com/blog/2021/08/data-preprocessing-in-data-mining-a-hands-on-guide/.

[2] AP News. Russia smuggling Ukrainian grain to help pay for Putin's war. URL https://apnews.com/article/russia-ukraine-putin-business-lebanon-syria-87c3b6fea3f4c326003123b21aa78099.

[3] Lauren Aratani. US imports of 'blood teak' from Myanmar continue despite sanctions. *The Guardian*, May 2023. ISSN 0261-3077. URL https://www.theguardian.com/world/2023/may/16/myanmar-teak-wood-import-sanctions.

[4] Devin. Introducing devin, the first ai software engineer. URL https://www.cognition-labs.com/introducing-devin.

[5] European Parliament. Trade restrictions / import ban on Russian and Belarusian wood and timber products, 2022. URL https://www.europarl.europa.eu/doceo/document/P-9-2022-000973_EN.html.

[6] ExportGenius. Exportgenius: Trade intelligence online platform. URL https://www.exportgenius.in/.

[7] C. Fan, M. Chen, X. Wang, J. Wang, and B. Huang. A review on data preprocessing techniques toward efficient and reliable knowledge discovery from building operational data, 2021. URL https://doi.org/10.3389/fenrg.2021.652801.

[8] Aaron B. Flaaen, Flora Haberkorn, Logan T. Lewis, Anderson Monken, Justin R. Pierce, Rosemary Rhodes, and Madeleine Yi. Bill of Lading Data in International Trade Research with an Application to the COVID-19 Pandemic. Finance and Economics Discussion Series 2021-066, Board of Governors of the Federal Reserve System (U.S.), October 2021. URL https://www.federalreserve.gov/econres/feds/files/2021066pap.pdf.

[9] Charvi Gopal. Network visualization and anomaly detection in international timber trade flows, 2021.

[10] Martin Heller. Large language models and the rise of the ai code generators. URL https://www.infoworld.com/article/3696970/llms-and-the-rise-of-the-ai-code-generators.html.

[11] ImportGenius. Importgenius: Global trade data. URL https://www.importgenius.com/.

[12] Prathamesh Ingle. Top artificial intelligence (ai) tools that can generate code to help programmers (2024). URL https://www.marktechpost.com/2024/03/14/top-artificial-intelligence-ai-tools-that-can-generate-code-to-help-programmers/.

[13] Naman Jain, Skanda Vaidyanath, Arun Iyer, Nagarajan Natarajan, Suresh Parthasarathy, Sriram Rajamani, and Rahul Sharma. Jigsaw: Large language models meet program synthesis, 2021.

[14] Harshit Joshi, Abishai Ebenezer, José Cambronero, Sumit Gulwani, Aditya Kanade, Vu Le, Ivan Radiček, and Gust Verbruggen. Flame: A small language model for spreadsheet formulas, 2023.

[15] Abdullah Alka Kandilli. How is dirty data handled in

data analytics? URL https://medium.com/@sweephy/how-is-dirty-data-handled-in-data-analytics-1767fb998e37.

[16] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive NLP tasks, 2021.

[17] Jia Li, Yongmin Li, Ge Li, Zhi Jin, Yiyang Hao, and Xing Hu. Skcoder: A sketch-based approach for automatic code generation, 2023.

[18] Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Rongyu Cao, Ruiying Geng, Nan Huo, Xuanhe Zhou, Chenhao Ma, Guoliang Li, Kevin C. C. Chang, Fei Huang, Reynold Cheng, and Yongbin Li. Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls, 2023.

[19] Meta. Code-llama. URL https://github.com/facebookresearch/codellama.

[20] Panjiva. Panjiva supply chain intelligence: Data-driven insights for the global trade community. URL https://panjiva.com/.

[21] Pinecone. Pinecone. URL https://www.pinecone.io/.

[22] Qdrant. Qdrant. URL https://qdrant.tech/.

[23] David Ramel. Top 10 ai ’copilot’ tools for visual studio code. URL https://visualstudiomagazine.com/articles/2023/06/30/vs-code-copilots.aspx.

[24] Qiaoyu Tang, Jiawei Chen, Bowen Yu, Yaojie Lu, Cheng Fu, Haiyang Yu, Hongyu Lin, Fei Huang, Ben He, Xianpei Han, Le Sun, and Yongbin Li. Self-retrieval: Building an information retrieval system with one large language model, 2024.

[25] Kexun Zhang, Danqing Wang, Jingtao Xia, William Yang Wang, and Lei Li. Algo: Synthesizing algorithmic programs with llm-generated oracle verifiers, 2023.