Improving Data Preparation for CSV Files with LLMs

Alexander van Renen
University of Technology Nuremberg
alexander.van.renen@utn.de

Moritz Rengert
University of Technology Nuremberg
moritz.rengert@utn.de

Macallyster Edmondson
University of Technology Nuremberg
macallyster.edmondson@utn.de

Andreas Kipf
University of Technology Nuremberg
andreas.kipf@utn.de

ABSTRACT

Data preparation and cleaning is a critical step in the data science process, often consuming a significant amount of time and effort, especially for less experienced engineers. With the recent advancements in Generative AI (GenAI), many tasks that previously required substantial manual effort can now be automated and, in fact, GenAI can take over the generation of the entire data pipeline. However, handing over complete control to a data science agent goes hand in hand with losing the ability to control, understand, and debug issues in a data pipeline. Therefore, we argue for using GenAI and agents as assistants to help data scientists with specific tasks, rather than replacing them outright.

In this work, we show how machine learning and GenAI can be used to assist with two specific tasks: First, when reading CSV files, it needs to be decided whether the first row is a header or not. While this is a simple task for humans, in text-only data an understanding of the semantics of the text is required to make this decision. Leveraging AI, we reduce the error rate to 0.5%, corresponding to a 5 to 20x improvement over Python's built-in CSV parser and DuckDB, respectively. Second, using our data pipeline system *Dataloom*, we demonstrate how data agents can support data cleaning while keeping all transformations understandable and verifiable by users. This is achieved by generating SQL code, explanations, as well as pre- and post-transformation data samples for each generated transformation, allowing users to inspect and verify the changes made to their data.

VLDB Workshop Reference Format:
Alexander van Renen, Moritz Rengert, Macallyster Edmondson, and Andreas Kipf. Improving Data Preparation for CSV Files with LLMs. VLDB 2026 Workshop: 15th International Workshop on Quality in Databases (QDB'26).

1 INTRODUCTION

Data Cleaning. It is widely known that data scientists spend most of their time on data preparation and cleaning [8, 11, 20]. In addition, the quality of machine learning models and data pipelines in general is highly dependent on the quality of the data used [27]. Further, decision making in companies is more and more data-driven, and thus the quality and, even more critically, the amount of time it takes from a given intent to the result is crucial. Therefore, data preparation and cleaning is still a very important issue and any improvements in this area can have a large impact.

Autonomous GenAI Agents. With the advent of generative AI (GenAI), large language models (LLMs), and data science agents, it becomes tempting to think that these tools can completely automate the data preparation and cleaning process. However, doing so comes with a lot of risks, such as hallucination, bias, cost, and privacy issues. While pasting a bunch of data into ChatGPT or Claude provides quick results, we argue that this is not a viable solution for business-critical decisions: It is not scalable and, even more important, these results are not transparent, repeatable, or explainable: the user has little insight into how the data was generated. All of these aspects are paramount for business-critical decisions.

Careful GenAI Integration. We argue that instead of removing the human from the data preparation and cleaning loop outright, a system needs to be carefully engineered to aid the user with specific tasks. By orchestrating many smaller agents and focusing their scope, we can provide guardrails to mitigate the risks of using GenAI. In addition, we propose code-generating agents that produce code that can be inspected and understood by users, thus providing transparency and repeatability for what is being changed in the data. Another advantage of generating code is runtime performance: for large datasets it is not feasible to feed all data to a GenAI model or even upload it to a GenAI provider.

Outline. In this paper, we discuss how machine learning models and generative AI can assist users with data preparation and cleaning. The focus is on (I) keeping the user in the loop, (II) providing transparency on what is being changed in the data, (III) speeding up repetitive tasks, and (IV) enabling non-expert users to perform data preparation and cleaning tasks. First, in Section 2, we show how text embeddings and GenAI can be used to further automate CSV file parsing. Existing systems are already doing extremely well in detecting the CSV dialect (separators, delimiters, etc.), but they still struggle with detecting whether a header row with column names exists in text-only data because their methods are primarily syntactic and lack semantic understanding. Leveraging machine learning techniques, the algorithm can gain an understanding of the data and thus significantly reduce the error rate, especially for text-only data. Second, we demonstrate a possible integration of GenAI into data processing pipelines by generating SQL-based data cleaning rules statically and dynamically (cf. Section 3). Afterwards, we discuss related work in Section 4 and conclude with a discussion of our findings, limitations, and future work in this area Section 5.

Proceedings of the VLDB Endowment. ISSN 2150-8097.

2 CSV HEADER DETECTION

Significance of CSV. CSV files are still omnipresent and widely used as a format for structured data exchange and storage: Vitagliano et al. [25] surveyed 17 governmental data portals and found that more than 30% of the datasets are in CSV format. Even in cloud database systems the problem remains relevant as a recent study has found that more than three quarters of the data scanned from S3 is stored in CSV files [23]. Therefore, it is important for data processing systems to enable seamless and efficient handling of CSV files.

Problem Definition. When loading data from a CSV file for processing, the first step is to determine the structure of the CSV file, i.e.: field separators and row delimiters. As the Pollock benchmark [9] has shown, systems like DuckDB [17] can solve this problem with a reasonably high accuracy already: DuckDB is placed first in the benchmark with a score of 9.961 out of 10. To find suitable separators and delimiters for a given file, all possible combinations are evaluated, and the best one is selected using a scoring mechanism [1]. The second step, before any data from a CSV file can actually be prepared, cleaned, or processed, is to determine whether the first row of the file contains column names (i.e.: a header) or actual data. We call this the *header detection* problem. Unlike the detection of separators and delimiters, existing systems still have a high error rate when detecting headers, as shown in Table 1: On two large public datasets, the error rate ranges between 2.2% and 40.5% for DuckDB and Python's CSV sniffer, respectively.

Outline. The benefits of reducing errors during header detection are twofold: (I) it prevents losing data (in case data is incorrectly classified as a header) and (II) it avoids data quality issues (in case a header is incorrectly classified as data). In this section, we first establish a baseline by explaining the existing algorithms for this problem as used in DuckDB and Python's built-in CSV sniffer. Next, we present our solution: a sequential rule-based classifier using text embedding and large language models (LLMs). Finally, we evaluate our algorithm on two datasets, breaking down the effectiveness of each rule and comparing it to the existing approaches.

2.1 Python's CSV Sniffer

Overview. Python's built-in csv module provides the `Sniffer` class for detecting the structure of CSV files, including header detection [6]. The algorithm for header detection is based on a voting mechanism: each column votes for (+1) or against (−1) the presence of a header row. If the sum of votes is positive (> 0), the algorithm assumes there is a header row; otherwise, it assumes there is no header row. A sample of the first 21 rows is used.

Algorithm Details [5]. The sniffer first analyses if the potential data rows (row 1 to 20) are of a consistent format. Two formats are considered: (I) either all numeric or (II) all strings of the same length. If none of these two formats is found in the potential data rows, the column abstains from voting. Otherwise, the header row (row 0) is compared to the found format: If it fits the format, it is assumed to be data as well (no header: −1). If it does not fit the format, it likely contains column names (has header: +1). Finally, the sum of votes determines the final decision on the presence of a header: a positive sum indicates the presence of a header, while a non-positive sum indicates its absence.

2.2 DuckDB

Overview. Another popular system for handling CSV files is the in-memory embedded database engine DuckDB. Using its Python integration, it can be directly plugged into any machine learning pipeline. As stated before, their CSV parser is state-of-the-art and also comes with an automatic header detection algorithm [3]. Being a full-fledged database system, DuckDB's header detection algorithm is tied more closely into the strict SQL type system.

Algorithm Details [4]. The DuckDB sniffer uses a sample size of 20 480 rows by default. First, it infers the most specific possible type for each column from the sampled data rows (excluding row 0). Then, it checks if the potential header row can be cast into the determined type for each column. If this is possible for *all* columns, the algorithm assumes that there is no header row. Otherwise, if at least one column cannot be cast into the determined type, it is assumed that the first row contains column names (has header). In addition, there are two corner cases: if all columns are null, or if all columns are of type *varchar* (i.e.: string/textual), the algorithm assumes that there is a header row.

2.3 Our Approach

Overview. Both described algorithms use the idea of comparing the type of the potential header row to that of the remaining data rows. This approach already works well for many typical CSV files, as our evaluation has shown for DuckDB and Python's CSV sniffer (cf. Table 1). However, it can lead to wrong results when the header row can be cast into the same type as the data rows (e.g., when all values in the CSV file are of string types).

Algorithm Details. To improve upon this, we adopt the more flexible voting-based idea of Python's CSV sniffer and combine it with a number of additional rules (i.e.: heuristics). All of our rules are plugged into a sequential rule-based classifier: a list of decision rules, which are executed in a fixed order. Each rule can either classify the given CSV input as having a header, not having a header, or abstain from voting, thus allowing the next rule to make the decision. Hence, the first rule making a decision determines the final classification of the input. Compared to the existing approaches, our algorithm adds a couple of special case rules, which catch common corner cases in CSV files. However, the key advancements are (I) using a text embedding to compare the similarity between potential header and data rows; and (II) using an LLM on a sample to determine the presence of a header row. In the following, we describe the different rules in more detail.

Special Cases. First, we check for a number of special cases in the CSV structure.

- **Single row:** If the CSV file has only one row, we assume that there is no header row.
- **Null:** As in DuckDB, if all values in the potential header row are null, we assume that there is a header row. In addition, if only some values in the row are null, we assume that there is no header row, as column names are rarely null.
- **Length:** If any of the values in the potential header is longer than a threshold (e.g., 100 characters), we assume that there is no header row, as column names are rarely that long. In fact, PostgreSQL only allows column names of at most 63 characters.

- **Equality:** If the values in the potential header show a large overlap with those in subsequent rows, we assume that there is no header row. For instance, if the first three rows are identical, it is very likely that there is no header row, as column names are rarely identical to data values.

Type Check Rule. Next, we perform a type check similar to the one of DuckDB and Python's CSV sniffer, but combine the strength of both approaches: Similar to DuckDB we use a SQL-like type system, which has the advantage that cases where a year number is used as a column name (e.g., `2020`) and dates are used in the data rows (e.g., `2020-01-01`) can be correctly classified as having a header row, while Python's CSV sniffer would fail in this case. Similar to Python's CSV sniffer, we use a more nuanced decision-making process where a majority of columns need to be compatible with the data rows.

Text Embedding Rule. The previous rules only work on a syntactic level, however, when all values are strings, it is difficult to determine the presence of a header row. To address this, we propose using a text embedding model which captures the semantics of the individual values. For this, we create embeddings for each cell in the first four rows using the `all-MiniLM-L6-v2` sentence-transformer model [19, 21]. Let E denote this two-dimensional array where each *column*, *row* subscript holds the embedding of the corresponding cell in the CSV file (e.g., $E_{c,r}$ is the embedding of the cell in column c and row r). We then compute the average similarity between pairs of assumed data rows $\mathrm{drs}(c)$ as well as between header and data rows $\mathrm{hrs}(c)$ per column c:

$$\mathrm{hrs}(c) = \frac{1}{3} \cdot \big(\mathrm{sim}(E_{c,0}, E_{c,1}) + \mathrm{sim}(E_{c,0}, E_{c,2}) + \mathrm{sim}(E_{c,0}, E_{c,3})\big)$$

$$\mathrm{drs}(c) = \frac{1}{3} \cdot \big(\mathrm{sim}(E_{c,1}, E_{c,2}) + \mathrm{sim}(E_{c,1}, E_{c,3}) + \mathrm{sim}(E_{c,2}, E_{c,3})\big)$$

where $\mathrm{sim}(a, b)$ denotes the cosine similarity between embeddings a and b. We then compute the `mean`, `min`, and `max` of these per-column similarities. The resulting values are used as features in a logistic regression model to determine the presence of a header row. The model was trained with `scikit-learn` [16] on a sample of ≈4000 labeled CSV files from the *Data.gov* dataset.

LLM Rule. Lastly, if all previous rules proved indecisive, we send a sample of the first 20 rows to an LLM to determine the presence of a header row. In our experiments, we show results for the `gpt-5.2` model from OpenAI [15]. Self-hosted models such as gpt-oss-120b [13] provided similar results during testing if privacy is a concern.

Finally, if none of the rules provides a decision, we assume that there is no header row.

2.4 Evaluation

We evaluate our proposed algorithm against two competitors: DuckDB and Python's CSV Sniffer. The evaluation is performed on ≈100 k CSV files from two large open CSV data sources: *Data.gov* and *Wiki*. We first describe how these two datasets were prepared, then explain our experimental setup, and finally interpret the results.

2.4.1 Datasets. **Data.gov** is a cleaned sample from the open data portal of the US government [22], which offers more than half a million datasets. We randomly picked 10 k datasets from the portal and then filtered out invalid downloads (some links were broken), files with fewer than three rows, files with invalid encodings, and non-CSV formats (e.g., HTML, JSON). After these cleaning steps a total of 7943 CSV files remained.

Wiki is a dataset of CSV files extracted from Wikipedia tables by Vogel et al. [26]. We use the *10k version* of the dataset, which consists of 10 k databases, each of which contains multiple CSV files. We again removed files with fewer than three rows, which resulted in a total of 42 472 CSV files.

Methodology. For each dataset, we created two copies of the CSV files: one with the original header row, and one where we removed the first three rows, simulating a case where the header row is missing. This way, we can evaluate the header detection algorithms in both cases: files with and without header rows. We then ran the header detection algorithms of DuckDB, Python's CSV sniffer, and our approach, which is part of our Java-based data pipeline platform called *Dataloom* [24], and compared the results to the ground truth.

Algorithm Comparison Experiment. Table 1 shows the results of the header detection algorithms on the two datasets for all three approaches. We report true positives "TP" (i.e.: files with header correctly classified as having a header), true negatives "TN" (i.e.: files without header correctly classified as not having a header), false positives "FP" (i.e.: files without header incorrectly classified as having a header), and false negatives "FN" (i.e.: files with header incorrectly classified as not having a header). In the worst case, our approach achieves a 0.5% error rate, while DuckDB goes up to

Table 1: Header Detection: *Comparing our approach to DuckDB and Python's CSV Sniffer ("Sniffer") on the Data.gov and Wiki datasets.*

Dataset	Method	TP	FP	TN	FN	FP Rate	FN Rate
Data.gov	Ours	7903	36	7907	40	0.45%	0.50%
	DuckDB	7918	742	7201	25	9.34%	0.31%
	Sniffer	7514	419	7524	429	5.28%	5.40%
Wiki	Ours	42462	108	42364	10	0.25%	0.02%
	DuckDB	42471	17201	25271	1	40.50%	0.00%
	Sniffer	38246	939	41533	4226	2.21%	9.95%

Table 2: Header Detection: *Breakdown of the success rates of the individual rules of our proposed algorithm.*

	Data.gov				Wiki			
Rule	TP	FP	TN	FN	TP	FP	TN	FN
Single Row	0	0	201	0	0	0	0	0
Null	0	15	2735	0	0	0	11933	0
Length	0	0	347	0	0	0	0	0
Equality	0	0	1488	0	0	0	11715	0
Type	7226	8	0	0	25925	24	0	0
Embedding	463	13	2946	26	14609	84	17725	10
LLM	214	0	190	12	1928	0	991	0
Fallback	0	0	0	2	0	0	0	0
Total	7903	36	7907	40	42462	108	42364	10

≈40% false positives (i.e.: claims a header where there is none) and Python's CSV sniffer up to approximately 10% false negative rate (i.e.: claims no header where there is one).

Breakdown Experiment. To show the contribution of the different rules in our approach, we performed a breakdown experiment where we measured the impact of each rule. The results of this experiment are shown in Table 2. Each row in the table corresponds to a rule and shows the number of files that were correctly and incorrectly classified. If a particular rule abstained from voting, the next rule in the sequence was applied until a decision was made.

2.4.2 Results and Discussion. **DuckDB's FP Rate.** Overall, it has a much higher false positive rate (40.5% and 9.34%) than false negative rate (≈0.3% and ≈0%) on the two datasets. This is caused by files consisting of only string type columns, where DuckDB assumes a header row. These cases are particularly hard to solve with purely syntactic rules (e.g., type checks) and require some insight into the semantics of the string values, which a text embedding model can provide. The effectiveness of the text embedding model in these cases can be seen in Table 2: in the Wiki dataset, our algorithm successfully classifies ≈17 k files with the text embedding rule. In addition, our null and equality rules help immensely with ≈11 k files correctly classified. Unlike DuckDB's null check, which only checks if all columns are null, our null heuristic also rejects potential header rows which have some null values, as column names are rarely null.

Python's CSV Sniffer's FN Rate. In contrast to DuckDB, Python's CSV sniffer algorithm does not heavily favor false positives, but yields more balanced results. In the Wiki dataset, it has a false negative rate of around 10%. This can, again, be explained with the abundance of text-only columns in the dataset: If the strings in one column are not all of the same length, the sniffer will not consider them in the vote. This often leads to zero votes for the potential header row, which results in a false negative classification.

Embedding Performance. While our embedding-based rule is very effective in correctly classifying files with only string type columns, it is also much more expensive to evaluate compared to the simpler syntactic rules. To quantify this, we measured the embedding performance in a micro experiment where we embedded 10 k random strings of 1 to 20 characters. Using the batched API, our test platform (MacBook Pro with M5 CPU) can embed around 2000 individual strings per second, with a batch size of 20. Embeddings are computed with `all-MiniLM-L6-v2` sentence-transformer model [19, 21] via DJL's PyTorch engine, backed by native LibTorch/PyTorch CPU execution. Thus embedding the first four rows of a CSV file with 10 columns (i.e.: 40 strings) takes around 20 ms. In comparison, DuckDB only requires around 1 ms to 2 ms to read most of the CSV files in the benchmark. However, the embedding-based approach could be accelerated by using a GPU, on-CPU accelerators, or smaller embedding models.

LLM Concerns. Another concern in terms of runtime, and also data privacy, is the use of the LLM for the final rule. However, this rule can be seen as optional and only handles a small fraction of cases; even when removing it, our approach already provides a significant improvement over the existing ones. In that case, we could tune the logistic regression model in our embedding-based rule to be more decisive and abstain from voting less often. While this would reduce its accuracy, it would also reduce the number of cases where the LLM needs to be applied, thus improving the overall runtime of our approach. If privacy is a concern, self-hosted models such as gpt-oss-120b [13] could be used instead of cloud-based ones.

Table 3: Data Preparation Issues: *A sample from one of the Eurostat [2] datasets with several data preparation issues: (A), (B), and (C).*

freq,unit,age,geo\TIME_PERIOD	2012	2023	2022
A,NR,TOTAL,AL01	: (B)	:	1.44192
A,NR,TOTAL,AL03	:	: (C)	1.03786
A,NR,TOTAL,PL21	1.32	1.25080 ep	1.34531 b
A,NR,TOTAL,PL22 (A)	1.27	1.13619 ep	1.21641 b

3 DATA PREPARATION ENHANCEMENTS

Before data can be effectively used in any data pipeline, it needs to be prepared and cleaned [7]. With the advent of large language models (LLMs) and data science agents, it is tempting to think that these tools can completely automate the data preparation process. However, even if agents worked perfectly, we argue that many decisions during data preparation and cleaning are not clear-cut and can be very domain-specific. We therefore do not see the data science agent as an outright replacement, but rather as an assistant. Thus, the question becomes how we can design a system where agent-based assistants work hand in hand with human users.

As an assistant, the agent can help experienced data scientists become more efficient. In addition, it enables domain experts with little or no data science experience to solve tasks. This removes the requirement for time-consuming back-and-forth communication across company departments with data scientists. In this section, we present our vision on how data science agents can be integrated into the data preparation process. In our system, we show how agents can be used for specific tasks, making the agent's changes transparent and understandable for users of all skill levels.

3.1 Integrating Data Science Agents into the Data Preparation Process

Dataloom [24] is our data science agent research system, where we explore ways to improve the data preparation process with the help of AI. For this particular case, let us consider the table in Table 3, which is an excerpt from a table from the *Eurostat* dataset [2]. We can see three particular data preparation issues in this table:

- (A) The first column actually contains multiple columns. Here the issue was that the column was escaped with double quotes in the CSV file, which caused the CSV parser to ignore the commas as separators: `"freq,unit,age,geo\TIME_PERIOD"`.
- (B) Missing values in otherwise numeric columns are represented using a colon symbol (':').
- (C) Some numeric values contain textual suffixes such as `b` or `ep`.

None of these issues are trivial to resolve in BI tools (e.g., Power BI or Excel) or in SQL for non-expert users. Even for expert users, the respective functionality likely has to be looked up and is verbose to implement. For instance, here is the SQL code to resolve the first issue (A):

Table 4: Date Parsing Benchmark: *Success rates for parsing* 50 *different date and timestamp formats.*

Method	Passed Absolute	Pass Rate
PyArrow	6	12.0%
Pandas	8	16.0%
SQLite[a,b]	10	20.0%
MySQL[a]	11	22.0%
Polars	15	30.0%
DuckDB[a]	18	36.0%
Postgres[a]	26	52.0%
Dataloom (ours)	50	100.0%

[a] We hard-coded the expected type here.
[b] Due to SQLite's type affinity system, we called date parsing functions explicitly here.

```
SELECT
    split_part("freq,unit,age,geo\TIME_PERIOD", ',', 1) AS freq,
    split_part("freq,unit,age,geo\TIME_PERIOD", ',', 2) AS unit,
    split_part("freq,unit,age,geo\TIME_PERIOD", ',', 3) AS age,
    split_part("freq,unit,age,geo\TIME_PERIOD", ',', 4)
        AS "geo\TIME_PERIOD",
    * EXCLUDE ("freq,unit,age,geo\TIME_PERIOD")
FROM eurostat;
```

Obviously, BI tools can implement rules to resolve these three specific issues and in fact our tool, Dataloom, initially did exactly that. However, we quickly realized that these three issues are only very specific instances of data preparation issues. Solving all would be impossible; solving many requires vast amounts of manual work to implement, test, and maintain rules for each specific issue.

3.2 Statically Generated Rules

To mitigate this problem, we can use coding agents to help generate a large number of specific rules for common data preparation tasks. A good example is date or timestamp parsing, which is a common data preparation issue. Writing specific parsers for all kinds of formats manually is cumbersome and very error-prone. However, generating these rules with the help of a coding agent is much easier and is a kind of problem that can be tested effectively by using (generated) examples.

As an example, we used a total of 50 different date and timestamp formats and tested various systems on their ability to parse these formats correctly. We then used a coding agent (in this case CODEX [14]) to generate parsing rules for these formats for our system, Dataloom. The results of this experiment are shown in Table 4.

Overall, the tested systems were able to parse between 12% and 52% of the formats correctly. Given that the generated rules saw the input data, they were able to parse 100% of the formats correctly. However, the generated rules are static, meaning that they only work for the specific formats they were generated for. And while we can generate a large number of rules for common data cleaning issues, specific datasets and use cases will always require specific rules that are not covered.

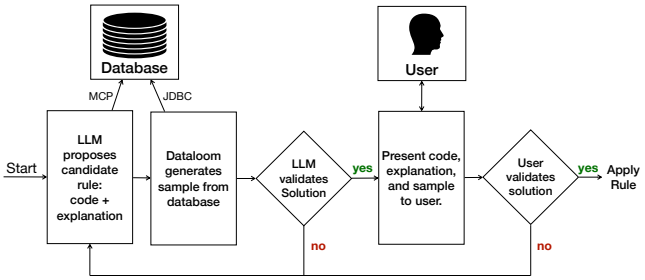

Figure 1: Data Cleaning Agent: *Data cleaning rules cycle with human verification of rules and their application.*

3.3 Dynamically Generated Rules

To solve the non-generality of the statically generated rules, we can generate rules dynamically. In Dataloom, we deploy an agent for this in two ways: (I) As a background task during data loading. (II) As a user-invoked task where the user can ask the agent to fix a specific issue. The process is illustrated in Figure 1. In both cases, the agent is prompted to generate SQL code (projections) that is applied to the specific column, thus providing strict guardrails on what the agent can change in the data. The agent is able to inspect the specific part of the SQL database (usually a column) and is allowed to self-validate its generated code before presenting it to the user. The resulting code, an explanation of it, and samples are presented to the user, thus allowing them to understand how the data is being changed before approving or rejecting it. The generated sample includes, if possible, modified and unmodified values, thus making it easier to understand the effect of the generated rule.

4 RELATED WORK

CSV Parsing. Robust CSV loading is a long-standing problem due to the loosely defined nature of the format. Döhmen et al. [1] detects the structure of a file (separators, delimiters, and quoting) by scoring candidate configurations, an approach also reflected in DuckDB's CSV parser [3, 17]. The Pollock benchmark [25] systematically quantifies how well such systems cope with non-standard CSV files. While existing systems rely on syntactic patterns, we isolate the header detection problem and show that purely syntactic signals are insufficient for text-only files, and semantic understanding provided by text embeddings and LLMs is needed to reach low error rates.

Data Preparation and Cleaning. Data preparation and cleaning has been studied extensively, and a wide range of commercial tools exists [7]. Interactive systems aim to keep the user in control: Potter's Wheel [18] lets users construct transformations incrementally while the system flags discrepancies, and Wrangler [10] (later commercialized as Trifacta) infers candidate transforms from direct manipulation and renders them as a readable, auditable script. Our system, *Dataloom*, shares the goal of keeping users in the loop, but instead of offering fixed transforms, it uses agents to generate SQL transformations on demand, presenting the generated code, an explanation, and before/after samples so that even non-expert users can verify each change.

LLMs for Data Preparation. More recently, large language models have been applied directly to data preparation tasks. Narayan et

al. [12] show that foundation models, prompted directly, achieve strong results on data cleaning and integration. In contrast, we use LLMs to generate inspectable code rather than to transform the data themselves: the model sees only a small sample, the generated SQL executes natively over the full dataset, and every change remains transparent and repeatable.

5 CONCLUSION AND FUTURE WORK

In this work, we have presented two ML- and AI-based techniques to improve automation in the data cleaning process while keeping the human in the loop. First, we proposed a carefully engineered CSV header detection algorithm that leverages machine learning and generative AI to achieve high accuracy in identifying header rows. Second, we have shown how data cleaning can be automated with AI while remaining understandable even by non-expert users by supplying code, explanations, and demonstrative examples of the changes being made in combination with focusing the task on a single column at a time. In the future, we plan to explore further techniques on how AI can be leveraged to automate data cleaning in a transparent and understandable way.

REFERENCES

[1] Till Döhmen, Hannes Mühleisen, and Peter Boncz. 2017. Multi-Hypothesis CSV Parsing. In *Proceedings of the 29th International Conference on Scientific and Statistical Database Management, Chicago, IL, USA, June 27-29, 2017.*

[2] European Commission, Eurostat. 2026. Eurostat Database. https://ec.europa.eu/eurostat/data/database. Accessed: 2026-05-14.

[3] DuckDB Software Foundation. [n.d.]. DuckDB CSV Sniffer Blog. https://duckdb.org/2023/10/27/csv-sniffer#header-detection [Accessed 13-05-2026].

[4] DuckDB Software Foundation. [n.d.]. DuckDB CSV Sniffer Code. https://github.com/duckdb/duckdb/blob/5ba4285f97fbe2488d0ca5aa8908cfb449513b88/src/execution/operator/csv_scanner/sniffer/header_detection.cpp#L348 [Accessed 13-05-2026].

[5] Python Software Foundation. [n.d.]. Python's CSV Sniffer Code. https://github.com/python/cpython/blob/d6b6fd70c4afce0210372945bca8e0bc7aa8cabf/Lib/csv.py#L452 [Accessed 13-05-2026].

[6] Python Software Foundation. [n.d.]. Python's CSV Sniffer Documentation. https://docs.python.org/3/library/csv.html#csv.Sniffer [Accessed 13-05-2026].

[7] Mazhar Hameed and Felix Naumann. 2020. Data Preparation: A Survey of Commercial Tools. *SIGMOD Rec.* 49, 3 (2020), 18–29. https://doi.org/10.1145/3444831.3444835

[8] Joseph M. Hellerstein, Jeffrey Heer, and Sean Kandel. 2018. Self-Service Data Preparation: Research to Practice. *IEEE Data Eng. Bull.* 41, 2 (2018), 23–34.

[9] HPI Information Systems Group. 2026. Pollock: A Benchmark for Data Loading on Character-Delimited Files. https://github.com/HPI-Information-Systems/Pollock. GitHub repository, accessed: 2026-05-14.

[10] Sean Kandel, Andreas Paepcke, Joseph M. Hellerstein, and Jeffrey Heer. 2011. Wrangler: interactive visual specification of data transformation scripts. In *CHI*. ACM, 3363–3372.

[11] Steve Lohr. 2014. For Big-Data Scientists, 'Janitor Work' Is Key Hurdle to Insights. *The New York Times* (17 Aug. 2014). https://www.nytimes.com/2014/08/18/technology/for-big-data-scientists-hurdle-to-insights-is-janitor-work.html

[12] Avanika Narayan, Ines Chami, Laurel J. Orr, and Christopher Ré. 2022. Can Foundation Models Wrangle Your Data? *Proc. VLDB Endow.* 16, 4 (2022), 738–746.

[13] OpenAI. 2025. gpt-oss-120b & gpt-oss-20b Model Card. arXiv:2508.10925 [cs.CL] https://arxiv.org/abs/2508.10925

[14] OpenAI. 2025. Introducing Codex. https://openai.com/index/introducing-codex/. Accessed: 2026-05-15.

[15] OpenAI. 2025. Introducing GPT-5.2. https://openai.com/index/introducing-gpt-5-2/ Accessed: 2026-05-13.

[16] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011).

[17] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: an Embeddable Analytical Database. In *SIGMOD*.

[18] Vijayshankar Raman and Joseph M. Hellerstein. 2001. Potter's Wheel: An Interactive Data Cleaning System. In *VLDB*. Morgan Kaufmann, 381–390.

[19] Nils Reimers and Iryna Gurevych. 2019. Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics. https://aclanthology.org/D19-1410/

[20] Ignacio G. Terrizzano, Peter M. Schwarz, Mary Roth, and John E. Colino. 2015. Data Wrangling: The Challenging Journey from the Wild to the Lake. In *CIDR*.

[21] Sentence Transformers. 2024. sentence-transformers/all-MiniLM-L6-v2. https://huggingface.co/sentence-transformers/all-MiniLM-L6-v2. Hugging Face model repository.

[22] U.S. General Services Administration. 2026. Data.gov. https://www.data.gov/. Accessed: 2026-05-13.

[23] Alexander van Renen, Dominik Horn, Pascal Pfeil, Kapil Vaidya, Wenjian Dong, Murali Narayanaswamy, Zhengchun Liu, Gaurav Saxena, Andreas Kipf, and Tim Kraska. 2024. Why TPC Is Not Enough: An Analysis of the Amazon Redshift Fleet. *VLDB* (2024).

[24] Alexander van Renen, Mihail Stoian, and Andreas Kipf. 2024. DataLoom: Simplifying Data Loading with LLMs. *VLDB Demo* (2024). https://www.vldb.org/pvldb/vol17/p4449-renen.pdf

[25] Gerardo Vitagliano, Mazhar Hameed, Lan Jiang, Lucas Reisener, Eugene Wu, and Felix Naumann. 2023. Pollock: A Data Loading Benchmark. *VLDB* (2023).

[26] Liane Vogel, Jan-Micha Bodensohn, and Carsten Binnig. 2024. WikiDBs: A Large-Scale Corpus Of Relational Databases From Wikidata. In *The Thirty-eight Conference on Neural Information Processing Systems Datasets and Benchmarks Track*. https://openreview.net/pdf?id=abXaOcvujs

[27] Wikipedia contributors. 2026. Garbage in, garbage out. https://en.wikipedia.org/wiki/Garbage_in,_garbage_out. Wikipedia, accessed: 2026-05-15.