# Auto-Suggest: A minimal Implementation

A Soft Implementation and Evaluation of “**Auto-Suggest: Learning-to-Recommend Data Preparation Steps Using Data Science Notebooks**”

## Contents

# Introduction

Data preparation, also known as data wrangling, refers to the process of transforming raw, often messy data from disparate sources into a clean and structured format suitable for business-intelligence analysis (BI) or machine learning modeling (ML). It is widely acknowledged as the most time-consuming phase in modern data workflows. Both academic studies and industry reports indicate that both data scientists and data analysts (often less technical users) can spend up to 80% of their time cleaning, joining, pivoting, and reshaping datasets before any modeling or reporting can begin. Automating this process has the potential to significantly improve user productivity, and to democratize modern BI and ML practices, making them more accessible to non-technical users. This growing demand for ease and speed in data preparation has given rise to a broader trend known as **self-service data preparation**, where the goal is to empower users, especially those without programming expertise to manage and transform data without relying on IT or engineering teams.

In response to these challenges, **Cong Yan and Yeye He** introduced **Auto-Suggest** in 2020, a novel system designed to recommend data preparation operations such as *Join, Pivot, Unpivot,* and *GroupBy* by learning from how data scientists manipulate data within real-world Jupyter notebooks. Their approach leveraged a large-scale crawl of over **4 million public notebooks on Github**, which were then replayed step-by-step to capture not only the exact sequence of operations but also the full input/output context of each transformation. These detailed resulting logs were used as a proxy for ground truth, enabling the training of machine learning models to suggest appropriate operations and parameters, much like search engines use click logs to improve result ranking relevance. In short, the system was able to learn directly from the diverse, real-world decisions made by data scientists. At the time the paper was published, existing commercial tools were primarily heuristic-based, which often failed to provide accurate recommendations, especially for complex operations like pivots or non-obvious joins. These heuristics typically relied on simple and static rules, such as value-overlap between columns or column name similarity, rather than learning from real user behavior. Auto-Suggest offered a data-driven alternative and formalized two core **recommendation tasks**:

**(1) single-operator prediction**, where the system recommends suitable parameters for a given operator (e.g., which columns to join or group by), and

**(2) next-operator prediction**, where the system predicts what transformation a user is likely to apply next, based on their workflow history and the current state of the dataset.

This report documents a **soft (minimal) implementation** of the Auto-Suggest system, focusing primarily on the **offline component** described in the original work. Instead of recreating the full-scale pipeline across millions of notebooks, we base our implementation on a **selected subset of operator examples** provided by the authors, which is sufficient to replicate the system's core insights. Our implementation includes feature extraction for selected operators, training and evaluation of models for both single-operator and next-operator recommendation tasks, and generation of basic model-based suggestions. In addition, we provide a lightweight demo pipeline that demonstrates how to crawl random GitHub repositories, extract notebooks, and perform notebook replay with operator logging and sequence tracking, mimicking the original replay infrastructure on a small scale.

We follow the same high-level structure as the original paper, describing our system architecture, data collection pipeline, operator-specific processing, and evaluation. We also detail **divergences** from the original work, including simplifications, constraints, and practical decisions that shaped our implementation.

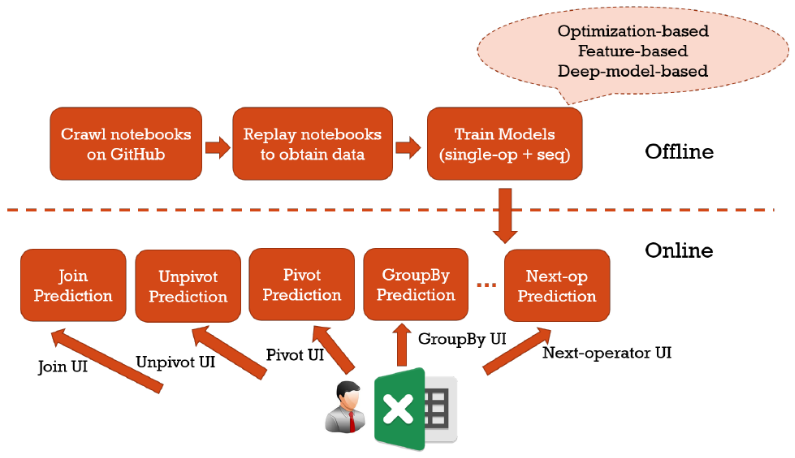
Our goals were:

1. To understand and replicate the technical mechanisms used in Auto-Suggest, including tracing and replaying notebook executions.
2. To construct a minimal dataset of operator traces and build basic predictive components for selected operations.
3. To reflect on the feasibility, robustness, and real-world applicability of Auto-Suggest's approach when implemented under realistic resource constraints.

This work serves as both a learning exercise in real data preparation and a practical reference for future efforts aiming to replicate or extend Auto-Suggest "idea" in academia or applications.

## System Architecture

The original Auto-Suggest system is designed as a full end-to-end architecture composed of two main components: an **offline component** that processes a large amount of Jupyter notebooks to extract operator behavior, and an **online component** that serves real-time operator recommendations to users within interactive environments.



**Original Design**

In the original design, the **offline component** begins with a **large-scale crawl** of Jupyter notebooks from Github. After syntactic **filtering** to identify notebooks containing relevant Pandas operations, each notebook is **replayed step-by-step** in a controlled environment. Using dynamic instrumentation, the system intercepts Pandas API calls and **logs** detailed execution data, including **input/output DataFrames**, **invoked operators** and **operator parameters**. This rich, low-level trace data serves as the foundation for training both **machine learning-based** and **optimization-based models** for parameter prediction (e.g., how to Join or Pivot). In addition, a **deep learning model** is trained to predict the next likely operator, using the operator sequences and characteristics of intermediate tables.

The **online component** integrates these trained models into an **interactive UI environment**. It supports both two key recommendation tasks.

**Soft Implementation Overview**

In our implementation, we focused exclusively on replicating the **offline component**, aiming to reproduce the essential ideas of Auto-Suggest in a minimal, but functional way. We did not implement the online recommendation interface, and we simplified many system-scale aspects. Our implementation consists of the following stages:

- **Notebook Ingestion (Demonstration Only)**
  To illustrate the pipeline conceptually, we created a small number of **dummy notebooks** that included Pandas operations such as merge, groupby, melt and pivot. These examples were used to validate instrumentation and trace logging, but were not used as training data for our models.
- **Notebook Replay and Instrumentation (Demonstration Only)**
  During notebook execution, a custom tracer used to capture invocations of key Pandas methods, along with arguments and DataFrame snapshots. We also experimented with handling issues such as missing packages and files, using the same logic described in the original implementation.
- **Operator Sequence Tracking (Demonstration Only)**
  To explore operator flow reconstruction, we tracked data dependencies across transformations within our dummy notebooks. This allowed us to build **dataflow graphs**, where nodes represent versioned DataFrames and edges represent transformation steps. While useful as a proof of concept, these sequences were not used for model training.
- **Operator Log Extraction (Used in Full Pipeline)**
  For the actual implementation, we used the **structured dataset provided by the authors**, which includes thousands of examples for selected operators. Each example contains:
    - Input tables (e.g. data.csv or left.csv/right.csv for merge)
    - Operation parameters (param.json)

We processed these files to create a consistent format for each operator and prepared them for training.

- **Feature Extraction and Modeling**
  For the first recommendation task **(single-operator prediction)**, we extracted features such as value overlap, distinct-value ratios, etc. These were used to train models for operators like Join, Groupby, Pivot and Unpivot. For the second recommendation task **(next-operator prediction)**, the dataset lacked operator sequences. To address this, we **synthesized artificial sequences**. These synthetic sequences, along with dataset's tables were combined and used to train a basic next-step prediction model.

**Simplifications from Original Architecture**

- We did not perform large-scale Github crawling. To demonstrate the replay process, we used dummy notebook examples. For training and evaluation, we reused operator data from the dataset provided by the authors.
- We omitted full-scale replay infrastructure and error handling.
- Operator sequences were not extracted from real notebook executions, but were artificially synthesized.

Despite these simplifications, our system captures the **core logic** of Auto-Suggest's offline pipeline: harvesting examples of real-world data preparation, learning from operator usage patterns, and generating data-driven recommendations for future transformations.

## Collect Data: Replay Notebooks

A key capability of the Auto-Suggest system is its ability to **learn data preparation behaviors from real-world Jupyter notebooks**. In the original work, the authors developed a large-scale infrastructure to **crawl, filter, and replay** over 4 million public notebooks from Github. This allowed them to extract, step-by-step, not only the **invoked Pandas operations** but also the **input/output tables**, the **parameters used** as well as the **operation sequences**. The resulting operator traces served as **training data** for various predictive components, forming the foundation of the entire system.

In our soft implementation, we take a **more constrained but focused approach**. Instead of performing large-scale crawling and full notebook replay, we rely primarily on a **small extracted dataset** released by the authors, which includes operator-level data examples for merge, groupby, pivot, melt. This extracted dataset includes 100 samples per operator.

**Challenges of Crawling and Replaying Notebooks**

While Github offers access to millions of repositories, converting those into **replayable and structured pipelines** is a difficult task. Many repositories contain:

- **Missing data files** (e.g., CSVs referred to with local paths)
- **Missing or incompatible dependencies**
- Multiple notebooks and multiple datasets, often loosely linked

To make notebooks **fully replayable**, a significant preprocessing is required. The original paper describes three strategies to resolve **missing data**, which we also explored:

1. **Resolving via Kaggle API** – when notebook content matches known public competitions
2. **Scraping URLs** – from markdown cells or inline comments near failed reads
3. **Searching for matching filenames** – within the repository directory tree

Even when data files are found, a repository may contain **many notebooks and many files**, with no explicit links between them. To reconstruct meaningful pipelines, one must:

- Infer **which notebook uses which dataset**
- Handle cases where **a single dataset is used across multiple notebooks**, or vice versa
- Manually or programmatically reduce noisy repos to **clean, minimal notebook+data units** suitable for replay

These complexities help explain why we chose to **rely on the extracted dataset**, which isolates clean, single-step operator examples, and use it as our main source of training data.

**The "Real" End-to-End Notebook Processing Pipeline**

In the original Auto-Suggest implementation, it was observed that only about **10–15% of the downloaded notebooks** were well-structured and **successfully replayable**. This means that to obtain just **100 usable samples per operator**, the system would need to crawl and process approximately **1000 notebooks**, many of which include broken code, missing files, or outdated dependencies. Handling this at scale requires substantial compute resources, orchestration, and error handling, **beyond what a typical laptop setup can support**. While the original paper **does not describe the exact infrastructure used**, it clearly implies the need for a robust, automated processing pipeline capable of handling large-scale notebook execution and data extraction. The following section presents **our own proposed containerized implementation**, build with tools like **Apache Airflow, Celery, Redis** and **MongoDB**, designed to achieve safe crawling, replay, and structured data extraction at scale.

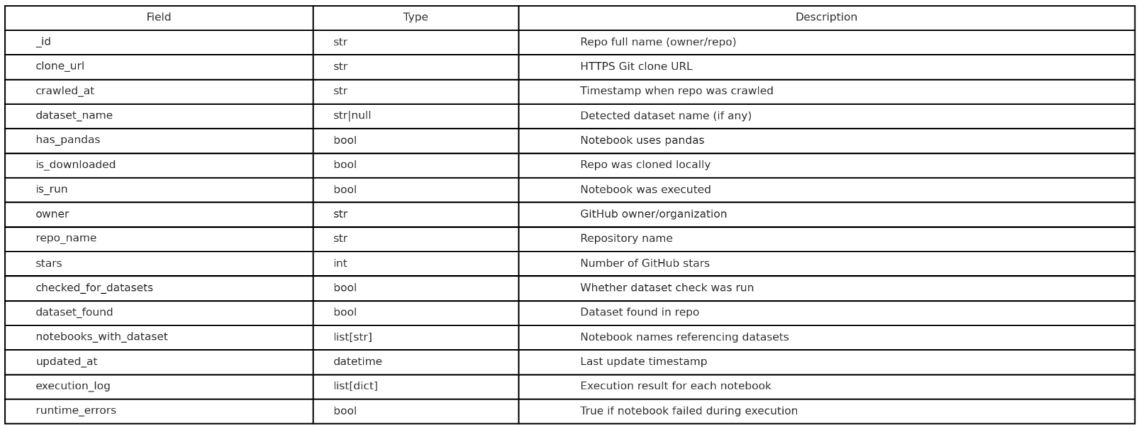
- **Architecture and Orchestration**

  The core architecture is implemented using the following components:

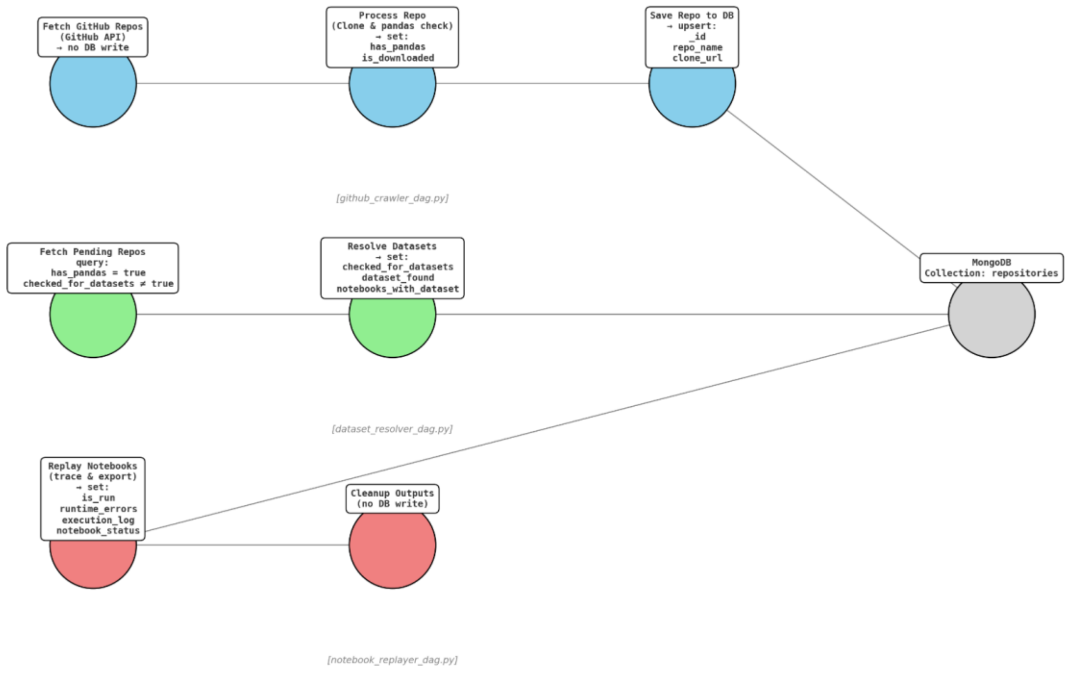
  - **Airflow Webserver:** Exposes the user interface for DAG control and monitoring
  - **Scheduler:** Triggers and schedules DAG executions
  - **Celery Workers:** Run individual Python tasks defined in the DAGs
  - **Redis:** Serves as a Celery message broker
  - **MongoDB:** Persists repository metadata and execution results
  - **Flower:** Provides a real-time dashboard to track Celery work activity

  These services are interconnected through a Docker environment within an isolated network, exposing only specific ports to the host machine. The necessary filesystem directories are mounted to ensure persistent storage of data and system state.

  The figures below illustrate the **MongoDB schema** used for tracking repository metadata, and the corresponding **Airflow DAG flowchart** that orchestrates the pipeline execution.



| Field | Type | Description |
|---|---|---|
| _id | str | Repo full name (owner/repo) |
| clone_url | str | HTTPS Git clone URL |
| crawled_at | str | Timestamp when repo was crawled |
| dataset_name | str\|null | Detected dataset name (if any) |
| has_pandas | bool | Notebook uses pandas |
| is_downloaded | bool | Repo was cloned locally |
| is_run | bool | Notebook was executed |
| owner | str | GitHub owner/organization |
| repo_name | str | Repository name |
| stars | int | Number of GitHub stars |
| checked_for_datasets | bool | Whether dataset check was run |
| dataset_found | bool | Dataset found in repo |
| notebooks_with_dataset | list[str] | Notebook names referencing datasets |
| updated_at | datetime | Last update timestamp |
| execution_log | list[dict] | Execution result for each notebook |
| runtime_errors | bool | True if notebook failed during execution |


Fetch GitHub Repos (GitHub API) → no DB write
Process Repo (Clone & pandas check) → set: has_pandas is_downloaded
Save Repo to DB → upsert: _id repo_name clone_url
[github_crawler_dag.py]
Fetch Pending Repos query: has_pandas = true checked_for_datasets ≠ true
Resolve Datasets → set: checked_for_datasets dataset_found notebooks_with_dataset
MongoDB Collection: repositories
[dataset_resolver_dag.py]
Replay Notebooks (trace & export) → set: is_run runtime_errors execution_log notebook_status
Cleanup Outputs (no DB write)
[notebook_replayer_dag.py]

- **github_crawler_dag.py** – *Repository Discovery and Filtering*

  This DAG is responsible for discovering and retrieving candidate GitHub repositories:

  - It uses GitHub's GraphQL API to find public repositories that match specific criteria (e.g. Jupyter Notebooks, sorted by stars in descending order) and retrieves them.
  - It paginates through results and uses a cursor to prevent downloading duplicate repositories across runs.
  - It clones the repositories in a temporary directory, checks them for **Pandas** presence, and those that have not included it are discarded.
  - Metadata such as repository name, URL, and Pandas presence is stored in **MongoDB** under the repositories collection.
  - Detailed **debug logs** track query cost, remaining API budget, and pagination cursor.

- **dataset_resolver_dag.py –** *Dataset Resolution and Validation*

  This DAG ensures that only notebooks with valid datasets are selected for further processing:

  - It recursively scans each repository's notebook to identify common data file formats (e.g. .csv, .json, .xls, .parquet).

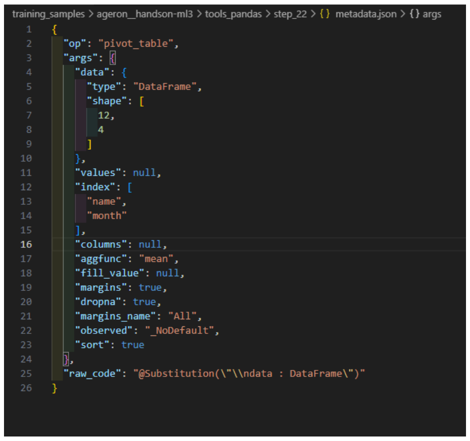
- If a dataset mentioned in a notebook is not found in the repository, the DAG automatically queries the Kaggle API to find it. If finally the dataset is found, it is downloaded and saved into the same corresponding repository's directory.
- When datasets are successfully resolved, the dataset_found field is set to true, and the repositories for which datasets where found are saved in MongoDB ("notebooks_with_dataset":[]).
- This flag is used as a condition in the replay stage.

○ **notebook_replaayer_dag.py –** *Notebook Execution and Operator Trace Extraction*

The final DAG is responsible for replaying Jupyter notebooks to extract the **sequence of Pandas API calls** and observe how DataFrames evolve during execution:

- It queries **MongoDB** for repositories where dataset_found == true.
- Each notebook is executed cell-by-cell using a tracing engine.
- It captures transformations such as "groupby", "merge", "pivot", "pivot_table", "melt", "explode", "drop_duplicates", "sort_values", "dropna" and "fillna".
- For every detected operator, two files are saved:
  - data.csv: A snapshot of the input/output DataFrames at the time of invocation (left.csv/right.csv for merge).
  - param.json: A structured JSON file describing the parameters used in the operation.
- MongoDB is updated with:
  - is_run = true if the notebook executes without critical failure.
  - runtime_errors = true if execution errors are encountered.
- Temporary folders are cleaned at the end of each run to prevent disk overflow.

The figure below shows an example of a **param.json** file, containing the captured parameters for each **Pandas API call** during notebook execution.


training_samples > ageron__handson-ml3 > tools_pandas > step_22 > {} metadata.json > {} args

```
{
  "op": "pivot_table",
  "args": {
    "data": {
      "type": "DataFrame",
      "shape": [
        12,
        4
      ]
    },
    "values": null,
    "index": [
      "name",
      "month"
    ],
    "columns": null,
    "aggfunc": "mean",
    "fill_value": null,
    "margins": true,
    "dropna": true,
    "margins_name": "All",
    "observed": "_NoDefault",
    "sort": true
  },
  "raw_code": "@Substitution(\"\\ndata : DataFrame\")"
}
```

The preprocessing pipeline combines **Airflow, Celery, Redis, MongoDB** and custom Python scripts to systematically transform raw GitHub notebooks into structured learning samples. Its modular design and trace-based execution strategy allow high-throughput processing while maintaining complete visibility over data lineage and operator context. This forms the backbone of the original (full) Auto-Suggest implementation model training phase.

For our **soft implementation**, we ultimately relied on the **authors' released dataset**, making **artificial adjustments where necessary** to simulate execution traces and operator sequences.

**Synthesizing Operator Sequences**

One key limitation of the extracted dataset is that it contains only **isolated operator invocations**. It lacks the **execution trace or sequence** of operations that would occur during a real notebook run. To support **next-operator prediction**, we needed sequences.

To address this, we applied a **naive random strategy** to synthesize 1000 operator sequences. These sequences are not semantically meaningful, but they allowed us to create training samples for sequence-based learning models (N-Gram and RNN).

**Combining Data for MLP Model Training**

The second prediction task, **next-operator prediction**, requires both the **history of applied operations** and the **characteristics of the current table** as input. To approximate this setup, we combined two resources: the **artificially generated operator sequences** and the **structured tables** from the dataset used in the first prediction task (parameter prediction for individual operators). To construct the training dataset for this task, we followed a **simple, but constrained strategy**. For each synthetic operator sequence, we appended one table, selected from the extracted operator examples (e.g., input to a groupby), to serve as the table context. This yielded a total of approximately **400 combined examples**, each consisting of an operator sequence and a table snapshot, paired with a label indicating whether a specific operator is the next one to apply.

It is important to note that the tables used in this dataset do **not reflect the actual state** of the intermediate data after the preceding operations, since we did not execute the full synthetic sequences. The table context is simply sampled from the extracted dataset and paired with a candidate operator, meaning the structural features of the data may not fully align with the operation history. Using this method also means that we limited the scope of prediction to **four core operations**: join, groupby, pivot and Unpivot, unlike the original implementation, which supported more operators including dropna, fillna, and concat.

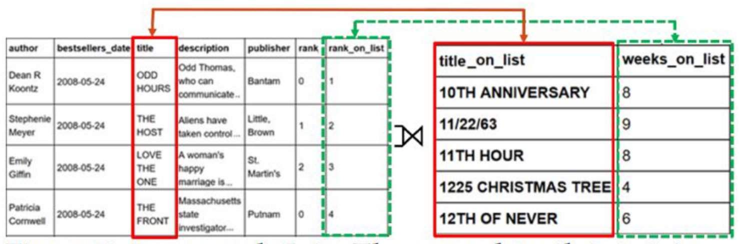
Despite these limitations, this setup allowed us to train a final **MLP model** that successfully combines operator-level trained models from first recommendation task and sequence information (approximated via **a RNN model**), using a reasonably **descent dataset**.

## Predict Single Operators

A key ability of Auto-Suggest is to recommend **how** to apply a specific data preparation operation, once a user has provided the input table(s). For example, this includes suggesting which columns to group by, which keys to use for a join, or how to structure a pivot table. Join and Groupby operations are relatively straightforward and can be modeled as simple feature-based machine learning problems, where the system learns patterns from prior usage to infer the most appropriate parameters for a given input. In contrast, operations of Pivot and Unpivot are considerably more complex and are formulated as optimization problems requiring more specialized, and custom, algorithms to be solved.

### Join Predictions

The Join operation is one of the most fundamental and frequently used steps in data preparation for combining data from multiple sources. In Pandas, it is implemented using the pd.merge() function and typically involves specifying three key parameters: **left_on** and **right_on**: columns to join on from the left and right tables (join keys), and **how**: the join type (e.g. inner, left, right, or outer). The challenge lies in accurately identifying both the join keys and the join type, as users often rely on intuition or heuristics, which can easily lead to incorrect matches based on superficial similarities like value overlap. The figure below illustrates this issue, showing a case where the correct join should be based on book titles, but heuristic-based methods incorrectly favor columns with higher value overlap.



| author | bestsellers_date | title | description | publisher | rank | rank_on_list |
|---|---|---|---|---|---|---|
| Dean R Koontz | 2008-05-24 | ODD HOURS | Odd Thomas, who can communicate... | Bantam | 0 | 1 |
| Stephenie Meyer | 2008-05-24 | THE HOST | Aliens have taken control... | Little, Brown | 1 | 2 |
| Emily Giffin | 2008-05-24 | LOVE THE ONE | A woman's happy marriage is... | St. Martin's | 2 | 3 |
| Patricia Cornwell | 2008-05-24 | THE FRONT | Massachusetts state investigator... | Putnam | 0 | 4 |

$\bowtie$

| title_on_list | weeks_on_list |
|---|---|
| 10TH ANNIVERSARY | 8 |
| 11/22/63 | 9 |
| 11TH HOUR | 8 |
| 1225 CHRISTMAS TREE | 4 |
| 12TH OF NEVER | 6 |

**Figure 5: An example Join: The ground-truth is to join using book-titles (in solid red boxes). Existing methods using heuristics tend to incorrectly pick columns in dashed-boxes that have a higher value overlap.**

In the original Auto-Suggest paper, join prediction is divided into two subtasks:

- **Join Column Prediction**: Identifying the most appropriate column(s) from each table to use as join keys.
- **Join Type Prediction**: Recommending the correct join type: inner, left, right, or outer.

To approach both tasks, we used a subset from the authors' provided dataset. Each example included left.csv, right.csv, and a param.json file specifying the ground-truth join parameters (left_on, right_on, how). To prepare data for training, we first generated **candidate column pairs** by matching columns with compatible types (e.g., numeric with numeric, string with string). We then extracted a set of features similar to those described in the original paper:

- **Distinct value ratio**: Ratio of unique values to total rows in a column.
- **Value overlap**: Measured using **Jaccard similarity** and **containment** between column value sets (e.g. what fraction of values in the left column appear in the right column).
- **Value-range-overlap**: For numeric columns, computed as the intersection over union of value ranges.
- **Column types**: Only compatible types (e.g. numeric with numeric, str with str) are considered.
- **Column position (Left-ness)**: Columns appearing earlier (toward the left) in a table are more likely to be used as join keys.
- **Sorted-ness**: Whether the column is sorted, which increases the likelihood of it being a key.
- **Single-column vs. multi-column candidates**: We allowed up to 3-column combinations; a binary feature flags whether a candidate is single-column.
- **Table size statistics**: Number of rows in each table and their ratio.

These features were used to train a **gradient boosting decision tree model** as a **binary classification** task, predicting whether each candidate pair corresponds to the ground truth. The model ranks all join key candidates based on their predicted probability.

For the **Join Type Prediction** task, the original paper does not specify the exact features used for join type prediction. For our implementation, we designed a feature set based on observations from the paper and data behavior:

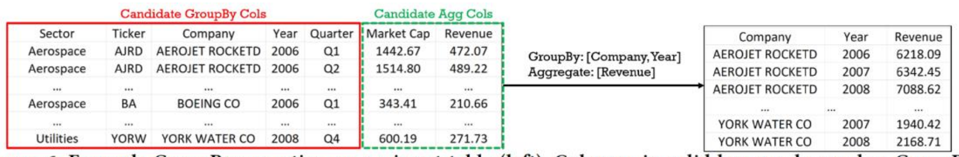
- **Table size features**: Number of rows and columns in each table.
- **Size ratios**: Features capturing whether the right table is significantly smaller or larger than the left, which can indicate whether an outer or inner join is preferred.
- **Small-column right table flag**: A heuristic based on the observation that when the right table has very few columns (e.g., ≤3), the join may act more like a filtering step, making inner joins more likely.
- **Value overlap**: Similar to the join key task, both Jaccard and containment metrics were used.
- **Sorted-ness**: Included as an additional signal.

We trained a **multi-class gradient boosting classifier** using these features. Due to class imbalance in our dataset (e.g. only one sample for right join), we limited prediction to the three most common classes: **inner, left, and outer**.

Despite the small size of the dataset, our models performed well. As we will show later, the results confirmed that even with limited data, a feature-rich, data-driven approach can outperform, or at least match, simple heuristics, which typically rely on rigid rules derived from a limited subset of these same features.

**Groupby Predictions**

The **GroupBy** operation is another fundamental data preparation step, commonly used in both SQL and Pandas workflows. In Pandas, it is implemented using the .groupby() method along with an aggregation function (e.g. sum, mean). It typically requires specifying two parameters: **by**: columns to group by (dimensions) and **value**: columns to aggregate (measures). The task here involves identifying which columns should be used as **grouping columns** (dimensions), but **not** which columns should be aggregated (measures), since the most common aggregation functions, such as sum and average, are typically equally applicable. Users may rely on intuition or simplistic heuristics, which can lead to suboptimal choices. The figure below illustrates this task, where the input table is split into **candidate GroupBy columns** (in solid red boxes) and **candidate aggregation columns** (in green dashed boxes). The output is generated by grouping on Company and Year, and aggregating Revenue.



| Sector | Ticker | Company | Year | Quarter | Market Cap | Revenue |
|---|---|---|---|---|---|---|
| Aerospace | AJRD | AEROJET ROCKETD | 2006 | Q1 | 1442.67 | 472.07 |
| Aerospace | AJRD | AEROJET ROCKETD | 2006 | Q2 | 1514.80 | 489.22 |
| ... | ... | ... | ... | ... | ... | ... |
| Aerospace | BA | BOEING CO | 2006 | Q1 | 343.41 | 210.66 |
| ... | ... | ... | ... | ... | ... | ... |
| Utilities | YORW | YORK WATER CO | 2008 | Q4 | 600.19 | 271.73 |

| Company | Year | Revenue |
|---|---|---|
| AEROJET ROCKETD | 2006 | 6218.09 |
| AEROJET ROCKETD | 2007 | 6342.45 |
| AEROJET ROCKETD | 2008 | 7088.62 |
| ... | ... | ... |
| YORK WATER CO | 2007 | 1940.42 |
| YORK WATER CO | 2008 | 2168.71 |

**Figure 6: Example GroupBy operation on an input table (left). Columns in solid box can be used as GroupBy columns (dimensions), while columns in dotted box can be used for Aggregation (measures).**

**using book-titles (in solid red boxes). Existing meth-
ods using heuristics tend to incorrectly pick columns
in dashed-boxes that have a higher value overlap.**

In the original Auto-Suggest paper, the GroupBy prediction task focuses specifically on detecting the appropriate **dimension columns**. To approach this, we used a subset of examples from the authors' dataset. Each sample consists of an input table (data.csv) and a parameter file (param.json) indicating the ground-truth grouping columns (via the "by" field).

To prepare the data for training, we first generate **candidate grouping columns**, and then extract features for each, similar to the strategy used for Join prediction:

- **Distinct value count and ratio**: Grouping columns usually have low cardinality.
- **Column data types**: String columns are more likely to serve as grouping columns.
- **Column position (Left-ness)**: Columns appearing toward the left of the table are more likely to be selected.
- **Emptiness**: Grouping columns typically have few or no missing values.

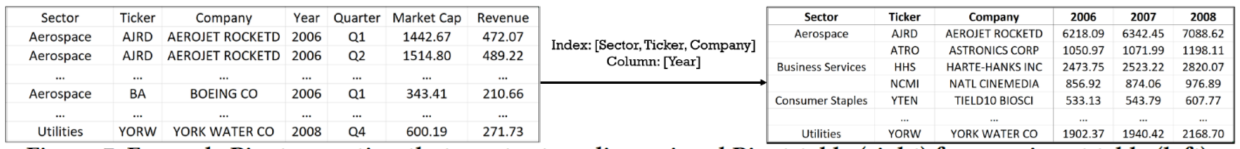
- **Value range**: For numeric columns, grouping columns tend to have small ranges (e.g. year, month).
- **Column name patterns**: Instead of looking for the most common names at the training set (as in the original implementation), we manually defined a small set of **common dimension-like names** (e.g., category, class, year) as name-based signals.

These features were used to train a **gradient boosting decision tree model** as a **binary classification task**, predicting whether each column is a valid GroupBy (dimension) column. The model ranks candidate columns based on their predicted probability of being selected. We also experimented with a model that attempted to **jointly predict both dimension and measure columns**, but this yielded poor results and was ultimately abandoned.

Despite the relatively simple setup and small dataset, our predictions performed **at least as well as** heuristic-based methods, confirming again that a data-driven approach, even with limited training data, can match or surpass traditional rule-based techniques (see Section 6 for evaluation details).

**Pivot Predictions**

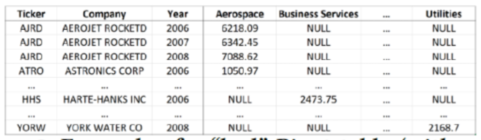
Pivot is one of the more complex data preparation tasks, as it transforms a **flat table** into a **two-dimensional layout**. In Pandas, this is implemented via the pivot_table() method and requires four key parameters: **index** (the row identifiers), **header/columns** (the pivoted columns), **values** (the aggregation columns) and **aggfunc** (the aggregation function, e.g. sum, avg). For non-technical users, this configuration is often unintuitive, and it usually takes **multiple trials and errors** in vendor tools to produce the desired result. The below Figures coming from the original paper illustrate this challenge:



| Sector | Ticker | Company | Year | Quarter | Market Cap | Revenue |
|---|---|---|---|---|---|---|
| Aerospace | AJRD | AEROJET ROCKETD | 2006 | Q1 | 1442.67 | 472.07 |
| Aerospace | AJRD | AEROJET ROCKETD | 2006 | Q2 | 1514.80 | 489.22 |
| ... | ... | ... | ... | ... | ... | ... |
| Aerospace | BA | BOEING CO | 2006 | Q1 | 343.41 | 210.66 |
| ... | ... | ... | ... | ... | ... | ... |
| Utilities | YORW | YORK WATER CO | 2008 | Q4 | 600.19 | 271.73 |

| Sector | Ticker | Company | 2006 | 2007 | 2008 |
|---|---|---|---|---|---|
| Aerospace | AJRD | AEROJET ROCKETD | 6218.09 | 6342.45 | 7088.62 |
| | ATRO | ASTRONICS CORP | 1050.97 | 1071.99 | 1198.11 |
| Business Services | HHS | HARTE-HANKS INC | 2473.75 | 2523.22 | 2820.07 |
| | NCMI | NATL CINEMEDIA | 856.92 | 874.06 | 976.89 |
| Consumer Staples | YTEN | TIELD10 BIOSCI | 533.13 | 543.79 | 607.77 |
| | ... | ... | ... | ... | ... |
| Utilities | YORW | YORK WATER CO | 1902.37 | 1940.42 | 2168.70 |

Figure 7: Example Pivot operation that creates two-dimensional Pivot-table (right) from an input table (left).



| Ticker | Company | Year | Aerospace | Business Services | ... | Utilities |
|---|---|---|---|---|---|---|
| AJRD | AEROJET ROCKETD | 2006 | 6218.09 | NULL | ... | NULL |
| AJRD | AEROJET ROCKETD | 2007 | 6342.45 | NULL | ... | NULL |
| AJRD | AEROJET ROCKETD | 2008 | 7088.62 | NULL | ... | NULL |
| ATRO | ASTRONICS CORP | 2006 | 1050.97 | NULL | ... | NULL |
| ... | ... | ... | ... | ... | ... | ... |
| HHS | HARTE-HANKS INC | 2006 | NULL | 2473.75 | ... | NULL |
| ... | ... | ... | ... | ... | ... | ... |
| YORW | YORK WATER CO | 2008 | NULL | NULL | ... | 2168.7 |

**Figure 8: Example of a "bad" Pivot-table (with many NULLs) that uses the same dimensions as Figure 7.**

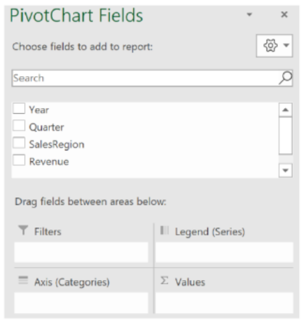


**Figure 9: UI Wizard to create Pivot-table in Excel. It requires users to drag suitable columns into 4 possible buckets (shown at bottom) to properly configure Pivot, which typically takes many trials to get right. Creating Pivot in other systems is similarly complex.**

To address this challenge, the authors of Auto-Suggest proposed a data-driven solution that learns from real-world Pivot usage in notebooks to automatically infer the appropriate configuration. Specifically, Auto-Suggest formalizes pivot recommendation as two sub-steps:

- **Predict Index/Header vs. Aggregation Columns**
  The first step is to determine which **columns serve as dimensions** (index/header) and which **as measures** (values). Intuitively, this is similar to predicting GroupBy columns, versus aggregation columns. Each training sample includes a data.csv file (the flat input table) and a param.json file that specifies the ground-truth pivot parameters: "index", "columns", "values", and "aggfunc". To simplify this step, we reuse our **GroupBy prediction model** (see Section 4.2) to identify likely **dimension columns**. These are then passed forward as candidates for the next step of determining how to split them between index and header.
- **Splitting Index vs. Header (AMPT Formulation)**
  Once the set of dimension columns is known, the next step is to split them into index and header such that the resulting pivot **minimizes sparsity** (empty/NULL cells) and maximizes conceptual closeness between columns, improving in this way table readability. This is very challenging for users to configure manually and often requires guesswork. Auto-Suggest addresses it using an **optimization-based approach** that relies on pairwise **affinity scores** between columns (columns that are semantically or structurally similar are grouped together).

## ❖ Affinity Score between Columns

To estimate how "conceptually close" two columns are (e.g. how likely they should appear on the same side of the Pivot), Auto-Suggest defines an **affinity score** based on two features:

- Emptiness Reduction Ratio (ERR): Measures how much sparsity (empty space) can be avoided by placing two columns on the same side.
    - ERR = Theoretical number of combinations / Actual observed combinations.

  Example: If col1 has 3 unique values [A, B, C] and col2 has 2 [1, 2, 1], the theoretical cross-product is 6. If only 3 pairs appear in the data, for example (A, 1), (B, 2), (C, 1), then ERR = 6 / 3 = 2.

  A higher ERR suggests the columns should be on the same side to avoid generating many empty cells.

- Column Position Difference: Measures how close the two columns appear in the original table (under the assumption that nearby columns are more likely to be related).
    - Position Difference = |position(column1)-position(column 2)| / total number of columns

These two features are combined using a **regression model** trained on real Pivot examples from the dataset to produce a **pairwise affinity score** between columns.

## ❖ AMPT Optimization

Using the affinity scores, Auto-Suggests builds an affinity **graph** such as the figure below, where each node represents a dimension column, and edges are weighted by affinity scores. The goal is to partition this graph into two groups: index/intra (e.g. Ticker, Company, and Sector) and header/inter (e.g., Year) that maximize internal similarity and minimize cross-group affinity.

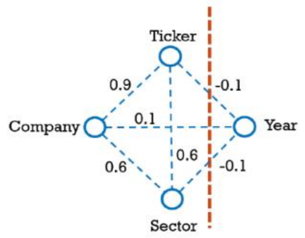


**Figure 10: Example graph with affinity-scores.**

This leads to the **Affinity-Maximizing Pivot Table (AMPT)** formulation:

$$
\begin{aligned}
\text{(AMPT)} \quad \max \sum_{c_i, c_j \in C} a(c_i, c_j) + \sum_{c_i, c_j \in \overline{C}} a(c_i, c_j) & \\
- \sum_{c_i \in C, c_j \in \overline{C}} a(c_i, c_j) & \qquad (1) \\
\text{s.t. } C \cup \overline{C} = C & \qquad (2) \\
C \cap \overline{C} = \emptyset & \qquad (3) \\
C \neq \emptyset, \overline{C} \neq \emptyset & \qquad (4)
\end{aligned}
$$

Where the first two terms of equation (1) represent the sum of affinities in index and header groups and third term represents the sum of cross-group affinities.

This problem is equivalent to **minimizing the affinity across the split**:

$$\sum_{c_i \in C, c_j \in \overline{C}} a(c_i, c_j)$$

This is formally equivalent to a **minimum graph cut** problem. Auto-Suggest solves it using the **Stoer-Wagner algorithm**, which efficiently finds the optimal cut for undirected graphs with non-negative weights. In rare cases where the algorithm fails (e.g. due to negative weights), a **greedy fallback strategy** is applied.

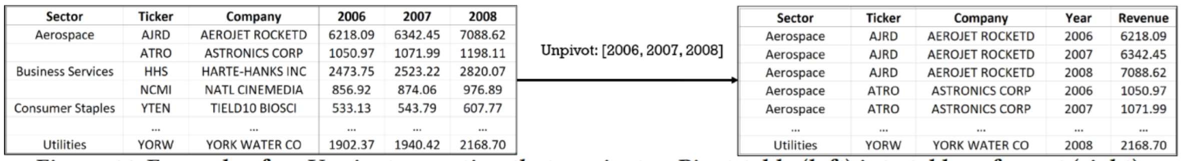
For example, suppose we have columns: ["Year", "Quarter", "Region", "Product"], and their affinities show: Year and Quarter are highly related, and Region and Product are also closely related, the system will likely split them as: index columns = ["Year", "Quarter"], and header columns = ["Region", "Product"]. This produces a clean Pivot table where rows represent time, and columns represent product-region combinations.

Our implementation follows exactly the same approach. It combines a) the Groupby model to detect dimension columns, b) a regression-based affinity model and c) the AMPT optimization formulation for clean column index/header splitting, to effectively replicating the original Pivot recommendation pipeline.

Despite the small training dataset, this combined data-driven and optimization-based approach significantly outperforms heuristic methods, especially in avoiding sparse or semantically awkward Pivot configurations.

**Unpivot Predictions**

The Unpivot operation is the inverse of Pivot as it reshapes a two-dimensional table back into a **flat tabular format**. In Pandas, this is implemented using the melt() function and typically requires two key parameters: **id_vars**: columns to retain as identifier variables (they remain unchanged), and **value_vars**: columns to collapse (i.e., columns whose names become values in a new key column, and whose values become values in a new value column). Correctly choosing which columns to unpivot is essential for preserving data semantics and improving downstream usability. For non-expert users, this decision is often difficult, especially in wide tables where dozens or hundreds of columns might be potential candidates. The below figure coming from the original paper shows how a wide pivot table is transformed into a long-form representation.



| Sector | Ticker | Company | 2006 | 2007 | 2008 |
|---|---|---|---|---|---|
| Aerospace | AJRD | AEROJET ROCKETD | 6218.09 | 6342.45 | 7088.62 |
| | ATRO | ASTRONICS CORP | 1050.97 | 1071.99 | 1198.11 |
| Business Services | HHS | HARTE-HANKS INC | 2473.75 | 2523.22 | 2820.07 |
| | NCMI | NATL CINEMEDIA | 856.92 | 874.06 | 976.89 |
| Consumer Staples | YTEN | TIELD10 BIOSCI | 533.13 | 543.79 | 607.77 |
| | ... | ... | ... | ... | ... |
| Utilities | YORW | YORK WATER CO | 1902.37 | 1940.42 | 2168.70 |

Unpivot: [2006, 2007, 2008]

| Sector | Ticker | Company | Year | Revenue |
|---|---|---|---|---|
| Aerospace | AJRD | AEROJET ROCKETD | 2006 | 6218.09 |
| Aerospace | AJRD | AEROJET ROCKETD | 2007 | 6342.45 |
| Aerospace | AJRD | AEROJET ROCKETD | 2008 | 7088.62 |
| Aerospace | ATRO | ASTRONICS CORP | 2006 | 1050.97 |
| Aerospace | ATRO | ASTRONICS CORP | 2007 | 1071.99 |
| ... | ... | ... | ... | ... |
| Utilities | YORW | YORK WATER CO | 2008 | 2168.70 |

**Figure 11: Example of an Unpivot operation that unpivots a Pivot-table (left) into tabluar format (right).**

To support accurate and automated Unpivot suggestions, Auto-Suggest frames the prediction task as identifying the most appropriate subset of columns to collapse (value_vars). Each training instance consists of a data.csv file which is the wide-form input table before Unpivoting, and a param.json that specifies the ground-truth value_vars (columns to collapse) and id_vars (columns to retain).

- **CMUT Optimization**

Similar to the Pivot case, Auto-Suggest computes pairwise scores to assess whether two columns are suitable to be collapsed together. These scores are referred to as **compatibility scores** (conceptually similar to affinity). They are calculated using the same features as in Pivot case (ERR and Column Position Difference) that they are combined together by using a regression model trained on Unpivot examples. So, a matrix is computed, which is represented as the below graph:

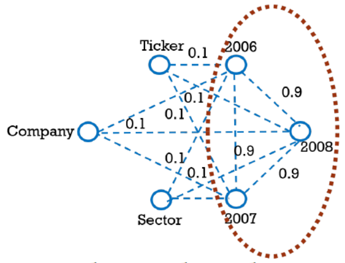


**Figure 12: Example graph with compatibility-scores for Unpivot (some edges are omitted to reduce clutter).**

Using the compatibility matrix, Auto-Suggest formulates the **Compatibility-Maximizing Unpivot Table (CMUT)** formulation:

$$\text{(CMUT)}\quad \max \underset{c_i, c_j \in C}{\text{avg}} a(c_i, c_j) - \underset{c_i \in C, c_j \in \boldsymbol{C} \setminus C}{\text{avg}} a(c_i, c_j) \tag{5}$$

$$\text{s.t. } C \subset \boldsymbol{C} \tag{6}$$

$$|C| \geq 2 \tag{7}$$

Unlike the AMPT problem formulated for Pivot, the CMUT introduces two key differences:

- It focuses only on maximizing compatibility **within** the selected group of collapsing columns (the CMUT set), rather than also considering the compatibility of the remaining columns.

- It uses average compatibility scores instead of sums, in order to avoid bias toward large clusters. Using sums would favor selecting more columns regardless of their actual compatibility.

Since CMUT is **NP-complete**, as it involves evaluating a very large number of column subsets, the authors designed a **greedy algorithm** to solve it:

- Initialization: Select the pair of columns with the highest compatibility score to initialize the CMUT set. The remaining columns form the second group.
- Iteration: At each step, add the column with the highest incremental compatibility gain to the CMUT set.
- Objective Recalculation: After each addition, re-compute the CMUT objective function (Equation 5).
- Termination: The process continues until **no further improvement** is observed. The best-scoring CMUT set encountered is returned as the final output. (this process is illustrated in **Example 7** from the original paper).

As with the Pivot case, we followed the same methodology in our implementation. We began by using the **GroupBy model** to identify candidate **dimension columns** in the input table. Then, using **extracted Unpivot samples**, we trained a **regression model** to combine the two compatibility features (Emptiness Reduction Ratio and Column Position Difference) specific to the Unpivot context. Finally, we applied the **greedy CMUT algorithm** over the compatibility matrix to identify the most likely set of columns to be unpivoted.

Despite the complexity of the task, especially given that some input tables had over 100 columns, our method achieves strong predictive performance, reaching at least the performance of based heuristics. The resulting recommendations provide an effective starting point for users to quickly normalize wide tables into a clean, long-form layout.

## Predict Next Operator

The second major recommendation task in the Auto-Suggest is to predict the **next most likely data preparation step** a user will take at time step *t+1*, given:

1. The **sequence of previously applied operations** up to time *t*, and
2. The **characteristics of the current DataFrame** at time *t*.

This prediction task captures real-world workflows where operators frequently follow recognizable patterns. For example, a pipeline containing dropna → merge is often followed by groupby, if the merged table contains grouping-friendly attributes.

## Model Architecture

To combine both these sources of signals, **operator sequence history** and **input table characteristics**, Auto-Suggest employs a **two-part model architecture**, shown in the below figure:

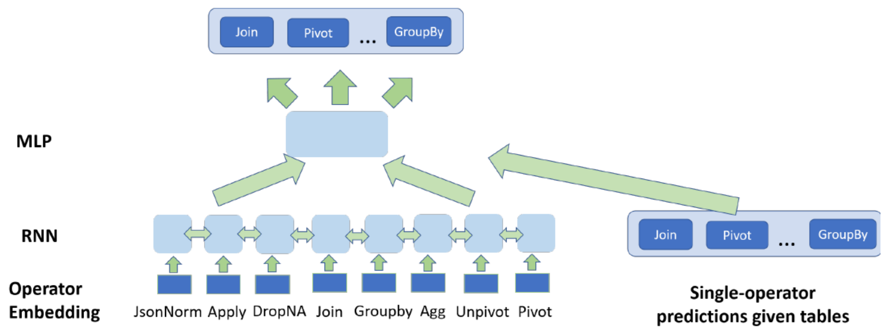


**Figure 13: Model architecture to predict next operator.**

The first component is a Recurrent Neural Network (RNN) trained on operation sequences extracted from replayed Jupyter notebooks. Given a sequence prefix such as dropna → merge, the RNN outputs a hidden representation that captures the latent structure of the pipeline and produces a ranked list of predicted next operations. The authors also experimented with an **n-gram model**, but found that the RNN substantially outperformed it in both precision and recall. The second component uses the current DataFrame to compute operator-specific scores by querying the four operator models introduced earlier:

- **Join** and **GroupBy**: Machine learning classifiers trained on annotated examples,
- **Pivot (AMPT)** and **Unpivot (CMUT)**: Optimization-based models producing objective scores.

Each of these models independently evaluates the current DataFrame and outputs a **confidence score** indicating the likelihood that its corresponding operation is applicable next. These five signals, one from the RNN and four from the individual operator models, are **concatenated into a single feature vector**, which serves as input to a final **multi-layer perceptron (MLP)**. The MLP outputs the final prediction: the most probable next operator, selected from a fixed set (e.g. groupby, pivot, unpivot, etc.).

**Implementation Notes**

Our implementation follows the same architecture as the original Auto-Suggest design. However, while the original system used a high-quality, manually filtered dataset generated via robust notebook replay (as described in Section 3 of the paper), we instead trained our model on a **combined dataset** generated using the procedure previously outlined. As a result, our prediction space is limited to **four operators**: merge, groupby, pivot, and unpivot, rather than the larger operator set used in the original implementation.

While our dataset include some noise or incomplete coverage, it effectively captures the essential sequential and structural patterns observed in real-world notebooks. As a result, our implementation successfully reproduces the core insight of the original Auto-Suggest approach: **combining the history of operations with the characteristics of the current table yields significantly more accurate predictions than using either source alone.**

# Experiments

In this section, we evaluate the performance of our lightweight implementation of the Auto-Suggest system across its two core tasks: **single-operator parameter prediction** and **next-operator prediction**. While our system operates at a smaller scale than the original, it enables controlled experimentation using a subset dataset of the original one and synthesized operator sequences. Our primary objective is to validate whether meaningful recommendations can be learned from a limited subset of real notebook data, using lightweight models and partial operator traces.

**Dataset Summary**

We conducted experiments using a subset of the dataset provided by the original Auto-Suggest authors. This dataset consists of isolated operator examples (e.g. merge, groupby, pivot, melt), each paired with input tables and ground-truth parameters. To enable **next-operator prediction**, we constructed **synthetic sequences** by logically grouping related examples into applicable pipelines. The final datasets included:

- **100 examples** per operator (merge, groupby, pivot, melt) that used for training single-prediction models
- A **synthesized dataset of 1000 operator sequences** for training sequence models (RNN, n-gram)
- **400 combined samples**, created by pairing the individual operator examples with their corresponding synthetic sequences, used for training the final **MLP-based next-operator prediction model**.

For machine learning tasks (e.g. join key prediction and groupby column selection), we used an 80% training / 10% validation / 10% test split. Validation was added to explore different model configurations, despite the limited data volume. For the final multi-layer perceptron (MLP) model used in next-operator prediction, we applied an 80/20 train-test split.

## Evaluation Metrics

To assess performance, we adopted standard **ranking and classification metrics** aligned with those used in the original Auto-Suggest paper:

- **Precision@k**: Measures the fraction of correct items (e.g., join columns) among the top *k* predictions.
- **NDCG@k (Normalized Discounted Cumulative Gain)**: Evaluates the ranking quality by rewarding correct items ranked higher.

Where applicable (e.g. join and groupby predictions), we computed both Precision@k and NDCG@k over candidate sets. For optimization-based tasks (pivot and unpivot), only top-1 predictions were available. In the next-operator prediction task, we did not generate ranked candidate lists. Instead, each example was framed as a classification problem over a fixed set of four possible operators (merge, groupby, pivot, unpivot). Accordingly, we report standard classification metrics such as precision and recall for evaluation.

## Experimental Setup

All models were implemented in Python using **scikit-learn** and **Keras**. Our setup included:

- **Gradient Boosting Classifier** for the ML-based models (join and groupby predictions)
- **Stoer-Wagner min-cut via networkx** and **greedy heuristics** for optimization-based Pivot (AMPT) and Unpivot (CMUT)
- **Keras layers** (e.g. Embedding, LSTM, Dense, Dropout) for the sequence model in next-operator prediction
- **Multi-Layer Perceptron (MLP)** for combining RNN and operator model outputs in the final prediction pipeline

## Results Overview

We evaluated all implemented prediction tasks and compared their performance against relevant baselines, following the approach in the original Auto-Suggest paper. Baselines include methods from prior literature, re-implemented and re-evaluated on our dataset, as well as vendor systems, for which we report the original scores directly as published in the paper.

**Predict Join Columns and Types**

To evaluate our join column prediction task, we compared our Auto-Suggest model against several baseline methods from the literature, as well as vendor systems reported in the original paper. The baselines include:

- **ML-FK**: A machine learning–based approach that generates join key candidates and extracts three features, **Jaccard similarity**, **containment** (fraction of left values appearing on the right), and **uniqueness**, to train a random forest classifier.
- **PowerPivot**: A heuristic-based method using **Jaccard similarity**, **containment** and **column name similarity**.
- **Multi-baseline**: Combines heuristically **containment** with **distributional similarity** between columns.
- **Holistic**: Uses multiple heuristics, including **left-to-right containment**, **name similarity**, **type matching** and **Jaccard overlap**.
- **Max-Overlap**: A simple baseline based solely on **Jaccard similarity**.

We also include results from commercial systems (**Vendor A, B, C**) as reported in the original Auto-Suggest paper.

All models were evaluated using a small test set of 8 join cases. We report standard ranking metrics: **Precision@1**, **Precision@2**, **NDCG@1**, and **NDCG@2**.

| **method** | **prec@1** | **prec@2** | **ndcg@1** | **ndcg@2** |
|---|---|---|---|---|
| Auto-Suggest | 0.88 | 0.88 | 0.88 | 0.88 |
| ML-FK | 0.88 | 0.88 | 0.88 | 0.88 |
| PowerPivot | 0.50 | 0.62 | 0.50 | 0.58 |
| Multi | 0.25 | 0.88 | 0.25 | 0.64 |
| Holistic | 0.38 | 0.88 | 0.38 | 0.69 |
| Max-overlap | 0.25 | 0.88 | 0.25 | 0.64 |
| Vendor-A | 0.76 | - | 0.76 | - |
| Vendor-C | 0.42 | - | 0.42 | - |
| Vendor-B | 0.33 | - | 0.33 | - |

Given the limited size of our dataset, the trained join column model is not expected to match the full-scale performance of the original Auto-Suggest system. Nonetheless, our results show that **ML-based methods (Auto-Suggest and ML-FK) outperform all heuristic-based baselines** (at least for prec@1), even on this small test set. This aligns with the findings from the original paper. Interestingly, **Auto-Suggest and ML-FK achieve identical scores** in this setting. This is expected, as ML-FK employs strong, overlapping features that are also central to Auto-Suggest. With more data, the original authors showed that Auto-Suggest slightly outperforms ML-FK.

To understand what drives our model's decisions, just like authors did in the original implementation, we analyzed feature importance in our gradient boosting classifier. Despite the limited data, we observed that the most influential features: **value overlap (Jaccard)**, **distinct value ratio**, and **leftness** are consistent with those identified in the original implementation. However, due to data sparsity, **value overlap** alone accounted for 93% of the importance, reflecting its dominance in this small-scale setup.

We also evaluated join type prediction (e.g. predicting whether the join should be inner, left or outer) using a 3-class classifier on the same limited test set (8 samples). Our model achieved **50% accuracy**, compared to **78%** reported for **Vendor-A** in the original paper. While this is a reasonable result given the data constraints, it suggests that join type prediction is more sensitive to data volume and class imbalance.

**Predict Groupby Columns**

As with join prediction, we evaluated our GroupBy column prediction model against both literature-based baselines and vendor systems reported in the original Auto-Suggest paper. The task involves identifying the most appropriate dimension columns for grouping operations, based on the structure and semantics of the input table. The baselines from prior work include the following:

- **SQL-History**: Simulates learning from column usage frequency in prior SQL queries. For demonstration, we approximated this using a fixed list of commonly used dimension-like column names (e.g. type, class, state, city, etc.).
- **Coarse-Grained Types**: A heuristic that treats columns with categorical types or numeric columns with low cardinality as likely grouping candidates. Thresholds are chosen naively.
- **Fine-Grained Types**: An improved version of the above, assigning higher scores to columns that match known dimension patterns (e.g. date, region, ID).
- **Min-Cardinality**: A simple baseline that selects columns with a unique-to-total value ratio (cardinality) below a threshold (e.g. unique/total < 0.2).

We also include **Vendor-B** and **Vendor-C** results, using the scores reported in the original paper. We evaluated on a small test set and computed **Precision@k** and **NDCG@k** to assess ranking quality.

| method | prec@1 | prec@2 | ndcg@1 | ndcg@2 |
|---|---|---|---|---|
| Auto-Suggest | 0.50 | 0.70 | 0.50 | 0.56 |
| SQL-history | 0.40 | 0.70 | 0.40 | 0.56 |
| Coarse-grained-types | 0.40 | 0.60 | 0.40 | 0.53 |
| Fine-grained-types | 0.60 | 0.80 | 0.60 | 0.69 |
| Min-Cardinality | 0.60 | 0.60 | 0.60 | 0.60 |
| Vendor-B | 0.56 | 0.71 | 0.56 | 0.75 |
| Vendor-C | 0.71 | 0.82 | 0.71 | 0.85 |

Despite the small sample size (test set = 8 samples), **Auto-Suggest performs competitively**, particularly when compared to simple heuristics. The Min-Cardinality and Fine-Grained Types baselines perform surprisingly well, which is expected since they rely on strong priors that often align with human intuition (e.g. grouping by year, city, etc.).

To better understand the behavior of our GroupBy model, we analyzed feature importances from the gradient boosting classifier. The top contributing features were **distinct value ratio** (columns with lower cardinality are more likely to be used for grouping) and **leftness** (columns appearing earlier in the table tend to be selected more often). Surprisingly, **column type** such as whether a column is a string or date, which often indicate dimensions, did not emerge as a strong signal in our results.

These findings are in line with the original Auto-Suggest paper, confirming that even in low-data regimes, meaningful GroupBy recommendations can be extracted using well-engineered features.

**Predict Pivot: Index/header split**

As with previous tasks, we evaluated our Auto-Suggest implementation against several baseline methods drawn exclusively from the literature in this case:

- **Affinity**: A heuristic-based method that groups attributes with hierarchical relationships. It computes average cardinalities and uses a greedy algorithm to assign columns to either index or header based on their relationships.
- **Type-Rules**: Applies a rule-based strategy where columns are classified based on data types and cardinality. For example, date/time and numeric columns with high cardinality are more likely to be assigned to the index.
- **Min-Emptiness**: Evaluates all possible split combinations (e.g. all-index, all-header, one index & rest header, etc.) based on emptiness ratio (e.g. fewest NULLs in the resulting pivot) and selects the configuration with the lowest one. This is conceptually similar to our **Emptiness Reduction Ratio (ERR)**.

- **Balanced-Split**: A simple heuristic that splits the columns in half, 50% to index and 50% to header, regardless of semantics.

Unlike Join or GroupBy, Pivot split prediction does not involve candidate ranking, so metrics like prec@k and ndcg@k do not apply. Instead, we evaluate:

- **Full Accuracy**: Measures whether the predicted split exactly matches the ground-truth split.
- **Rand Index**: A clustering similarity metric that evaluates how closely the predicted split aligns with the true split. It considers all pairs of dimension columns and rewards agreement on whether each pair belongs to the same group (either both index or both header).

| **method** | **full_accuracy** | **rand_index** |
|---|---|---|
| Auto-Suggest | 0.90 | 0.98 |
| Affinitty | 0.90 | 0.94 |
| Type-Rules | 0.92 | 0.95 |
| Min-Emptiness | 0.96 | 0.97 |
| Balanced-Split | 0.89 | 0.93 |

All methods perform reasonably well in this task, with Min-Emptiness achieving the highest full accuracy (96%). This confirms that minimizing sparsity (e.g. avoiding NULLs) is a strong indicator for producing good pivot splits. However, **Auto-Suggest achieves the highest Rand Index (0.98)**, meaning that even when its predicted split doesn't perfectly match the ground truth, it is often very close, correctly placing most column pairs in the same or different groups.

These results demonstrate that our optimization-based AMPT formulation, effectively captures the underlying structure of pivot layouts despite our small dataset.

**Predict Columns to Unpivot**

To evaluate the prediction of columns for unpivoting, we compare our Auto-Suggest model against several heuristic baselines adapted solely from the literature, similar to the approach used in the Pivot task. The baselines here are:

- **Pattern-Similarity**: Groups columns with similar naming patterns by identifying shared prefixes (e.g. year_2018, year_2019 → year). Ignores matches where the prefix equals the full column name.
- **Column Name Similarity**: Computes pairwise **Jaccard similarity** between column names to group related columns for unpivoting.

- **Data-Type**: Selects columns with matching data types, under the assumption that measure-like columns (e.g. to be unpivoted) typically share the same type and are not key-identifier columns.
- **Contiguous-Type**: An enhancement over the data-type baseline that adds the constraint that selected columns must also be **consecutive** in the table.

Given that unpivot prediction involves selecting subsets of columns from a wide table, we report standard metrics of **accuracy** (fraction of samples where the predicted set matches the ground truth exactly), **precision** (fraction of predicted columns that are correct), **recall** (fraction of actual unpivot columns that were correctly predicted) and **F1 score** (harmonic mean of precision and recall, providing a balanced performance measure).

| method | accuracy | precision | recall | F1 score |
|---|---|---|---|---|
| Auto-Suggest | 0.21 | 0.68 | 0.47 | 0.50 |
| Pattern-similarity | 0.28 | 0.68 | 0.52 | 0.53 |
| Col-name-similarity | 0.20 | 0.65 | 0.58 | 0.57 |
| Data-type | 0.27 | 0.60 | 0.74 | 0.64 |
| Contiguous-type | 0.30 | 0.65 | 0.71 | 0.65 |

Overall, **Auto-Suggest performs comparably** to the best heuristics, despite the difficulty of the task and limited training data. While its exact match **accuracy** (0.21) is slightly lower than some heuristics, its **precision (0.68)** remains high, indicating that most of the predicted columns are valid. However, slightly lower **recall** suggests that Auto-Suggest tends to be more conservative, missing some valid unpivot columns in favor of higher precision. The Contiguous-Type baseline performs best overall, likely due to the strong structural assumption that columns intended for unpivoting are often placed together and share similar types. Still, this assumption may not generalize well across all datasets.

Our results underscore the inherent difficulty of the unpivot prediction task, particularly when working with a limited dataset. One likely reason for this challenge is the way the regression model learns how features interact to estimate compatibility between columns, an information that later is used to construct the affinity (compatibility) matrix. This process requires a sufficiently large and diverse dataset to capture meaningful patterns of feature combinations. Expanding the dataset would likely enhance the model's ability to learn these relationships, ultimately leading to more accurate affinity scores and improved unpivot predictions.

**Predict Next Operator**

To evaluate our **next-operator prediction model**, we compared the full Auto-Suggest architecture against several baseline approaches, each representing a partial view of the task. The objective is to predict the most likely next data preparation step, given the sequence of operations performed so far and the characteristics of the current DataFrame. Methods to compare with are:

- **RNN Model**: A recurrent neural network trained solely on operation sequences. It predicts the next operator based on historical patterns in the workflow, without considering the input table.
- **N-gram**: A statistical language model using trigrams (*n=3*), which predicts the next operator based on frequency patterns in observed operator (artificial synthesized) sequences.
- **Single-Operators**: A model that uses only the predictions from the four individual operator models (Join, GroupBy, Pivot, Unpivot). It selects the operator with the highest score based solely on the current table's features, ignoring any operator history.
- **Random**: A naive baseline that randomly selects one of the four possible next operators.

| **method** | **precision** | **recall** |
|---|---|---|
| Auto-Suggest | 0.68 | 0.91 |
| RNN | 0.60 | 0.73 |
| N-gram | 0.47 | 0.72 |
| Single-Operators | 0.28 | 0.47 |
| Random | 0.19 | 0.32 |

The results clearly demonstrate that the **combined model (Auto-Suggest)**, which integrates sequence modeling via RNN with operator-specific table features, **outperforms all other methods**. While both the RNN and N-gram models capture sequential patterns reasonably well, they lack contextual awareness of the table's structure. Conversely, the Single-Operators baseline leverages table characteristics but ignores the sequence context.

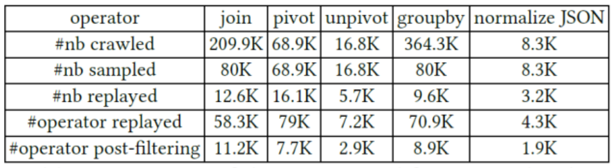
By fusing both types of signals, Auto-Suggest achieves the **highest precision and recall**, even when trained on a relatively small, artificially constructed dataset. This highlights the strength of **combining sequential context with feature-based operator predictions**, and validates the design of the hybrid model architecture proposed in the original paper.

## Divergences from the Original Paper

While our soft implementation of Auto-Suggest closely follows the overall structure and core methodology of the original work, several important divergences were introduced, some by design, others due to practical constraints related to scope, resources, and data availability. Below, we summarize the key differences between our approach and that of the original paper.

**Dataset Sources**

- **Original**: The authors crawled over **4 million Jupyter notebooks** from GitHub and **replayed thousands** using a dynamic execution engine. Their replay system included support for timeout management, dependency installation, and dataset resolution, enabling extraction of over **32600 high-quality operator examples**, complete with full input/output tables and execution sequences.



| operator | join | pivot | unpivot | groupby | normalize JSON |
|---|---|---|---|---|---|
| #nb crawled | 209.9K | 68.9K | 16.8K | 364.3K | 8.3K |
| #nb sampled | 80K | 68.9K | 16.8K | 80K | 8.3K |
| #nb replayed | 12.6K | 16.1K | 5.7K | 9.6K | 3.2K |
| #operator replayed | 58.3K | 79K | 7.2K | 70.9K | 4.3K |
| #operator post-filtering | 11.2K | 7.7K | 2.9K | 8.9K | 1.9K |

**Table 2: Statistics of data extracted from Notebooks.**

- **Ours**: We relied on a **pre-extracted dataset** released by the authors, using **100 examples per operator** (e.g. merge, groupby, pivot, melt). These samples are **isolated** and not linked through execution traces or real operator pipelines. No large-scale replay system or crawling infrastructure was developed. Instead, to better understand the instrumentation process, we manually created and simulated **10 dummy notebooks**, mimicking the replay logic described in Section 3 of the paper. For next-operator prediction, we **synthetically generated 1000 operator sequences** and combined these with the extracted operator examples to form **400 training samples**. However, this approach does **not reflect the true state transitions** between tables that would result from actual notebook execution.

**Baseline Simplifications**

- **Original**: The original paper compared Auto-Suggest against a combination of **literature-based** and **commercial vendor** baselines. These included well-tuned heuristics (and a machine learning-based model) with carefully engineered features and weighting.
- **Ours**: We recreated several **literature-based baselines**, but made **simplifying assumptions**:
    - For the groupby task, we defined common grouping column names (e.g. type, class, state) manually rather than deriving them statistically from the training set (Column-names feature). Similarly, for SQL-history baseline, we used again

frequency-based logic with these manually selected and fixed names rather than analyzing true column usage frequencies.

  - For baseline feature scoring, we combined signals using **naïve weighting** rather than tuned coefficients or learned combinations.
  - No vendor systems were used, **all vendor results are taken directly from the original paper** for comparison.

**Evaluation Strategy**

- **Original**: The evaluation was performed on a **large-scale dataset** with thousands of annotated examples across tasks. Comparisons included both academic and commercial baselines, and used a rich set of metrics including **Precision@k**, **NDCG@k**, **Full Accuracy**, and **Rand Index**.
- **Ours**: We evaluated on a **small test set**, using the same metrics wherever applicable. For example to test join and groupby column models, we used only 8 examples. Additionally, due to the limited dataset and the nature of our synthetic sequences, some metrics (e.g. Precision@k for next-operator prediction) were **not computable**.

**Prediction Scope and Output Format**

- **Original**: The full system supports **top-k prediction and ranking** for the next-operator task, with evaluation based on **Precision@k**, **NDCG@k**, and other ranking metrics.
- **Ours**: Due to the synthetic nature of our dataset and the way combined samples were constructed, we **only predict the top-1 next operation**. We do not generate ranked candidates, so metrics like Precision@k or NDCG@k are not applicable. Additionally, we restrict our next-operator predictions to **only four operators**: merge, groupby, pivot, and unpivot.

**Online Recommendation and User Interaction**

- **Original**: Auto-Suggest includes an **online component** that integrates with a front-end UI to provide interactive, real-time recommendations based on live table states and user behavior.
- **Ours**: Our system is entirely **offline**. It does not include a user interface or interactive feedback loop. All training and evaluation information is logged in CSV and JSON format, and predictions are conducted offline using predefined test tables.

Despite these divergences, our implementation remains **faithful to the core principles** of the Auto-Suggest paper. By selectively replicating key components, particularly those involving operator-specific modeling and the integration of sequence-based and table-based signals, we demonstrate that even a **minimal, offline implementation**, when guided by structured features (even if noisy), can yield **interpretable and effective recommendations** for data preparation workflows.

# Conclusion and Future Directions

This report presented a minimal yet functional reimplementation of the **Auto-Suggest** system that originally proposed by Yan and He (SIGMOD 2020), which learns to recommend data preparation steps from Jupyter notebooks. Despite working under resource and data limitations, we demonstrated that key components of the Auto-Suggest pipeline, **single operator prediction**, and prediction of **next-operator** can be successfully reproduced in a simplified setting using thoughtfully designed features and a modest dataset.

**Key Findings**

- Using a subset of the dataset released by the original authors, we were able to extract meaningful patterns for common operators such as merge, groupby, pivot, and melt and get a descent performance.
- Even a lightweight MLP model trained on a modest, combined dataset achieved strong performance in predicting the most likely next operator.

**Future Research**

This project offers a foundation for several promising research and application paths. Notably, **Auto-Pipeline** (VLDB 2021), a direct successor of Auto-Suggest by Yeye He, expands the vision by automating **entire data science pipelines**. Beyond this, the Auto-Suggest paradigm could have a strong practical potential in other environments where data science is actively practiced, including:

- **Platforms like Kaggle or Hugging Face**, instead of GitHub, where notebooks and datasets are shared at scale. An Auto-Suggest-like system could enhance user productivity by learning from shared workflows.
- **Business Intelligence (BI)** tools: Instead of learning from notebooks, Auto-Suggest could use BI projects to learn. One of the next moves of Yeye He was **Auto-Prep**, which focuses on predicting full data preparation steps for self-service BI users.
- **Spreadsheet environments**: Similar principles are applied in **Auto-Formula**, which recommends formulas in Excel using contrastive learning on table representations. This shows how Auto-Suggest-style learning can be applied to a wide range of **tabular data manipulation contexts** (for more, see Microsoft's Yeye He **publications**).
- **Domain-specific applications**: Instead of focusing on data manipulation, the general "idea" of Auto-Suggest could be applied into other domain-specific areas. For example, in the restaurant industry, a system that logs and analyzes customer orders could learn to predict the **next likely item** to sell based on previous selections and current context, enhancing both **user experience** and **sales performance**.

Across all these settings, the goal is unified: **to enable self-service**, empowering non-technical users to carry out complex data transformations that would typically require expert knowledge, or even to **accelerate expert workflows** through intelligent recommendations.

**Future work for the current implementation:**

Building on our soft prototype, the following directions could significantly enhance system capability and accuracy:

- Re-implement a full notebook replay engine with robust error handling and dependency management
- Scale operator extraction and model training using a much larger corpus of notebooks
- Extend support to additional operators such as concat, fillna, and json_normalize just like in the full version of the original paper
- Incorporate real-time user interfaces and interactive feedback loops
- Integrate with Auto-Pipeline to move beyond isolated operator recommendations toward **end-to-end pipeline automation**

In conclusion, even a minimal implementation of Auto-Suggest, when driven by structured data, engineered features, and clear modeling principles, can produce meaningful insights and highlight the path toward fully intelligent, self-service data preparation systems.

## Project's Repository

The complete implementation of this **soft version of Auto-Suggest** is available on GitHub:

**https://github.com/GiX007/auto-suggest-offline/tree/main**

The repository contains all code, detailed explanations, and extensive comments that document the reasoning behind each design choice and simplification. It serves as a comprehensive reference for anyone looking to **understand, reproduce, or extend** this minimal offline version of the Auto-Suggest system.

In addition, the **proposed pipeline** for the **full preprocessing phase,** including crawling, replaying and processing notebooks to extract structured training data is available at:

**https://github.com/MARUMDRS/airflow_pipeline/tree/data_resolver**