CoWrangler: Recommender System for Data-Wrangling Scripts

Bhavya Chopra, Anna Fariha, Sumit Gulwani, Austin Z. Henley, Daniel Perelman, Mohammad Raza,
Sherry Shi, Danny Simmons, Ashish Tiwari
Microsoft
{t-bhchopra,annafariha,sumitg,austinhenley,danpere,moraza,shersh,dsimmons,astiwar}@microsoft.com

ABSTRACT

We present CoWrangler, a real-time data-wrangling recommender system, that can recommend the next-best data-wrangling operations along with the corresponding human-readable and efficient code snippets to expedite data exploration and wrangling efforts. A key feature of CoWrangler is that it provides explanations for the generated suggestions in the form of data insights, allowing the user to place confidence in the system. Under the hood, CoWrangler relies on intelligent generation of candidate suggestions using program synthesis techniques and ranking of a set of suggestions based on the notion of data quality improvement. We demonstrate how CoWrangler provides a human-in-the-loop data-wrangling experience, and helps users make informed data pre-processing decisions, while saving their time and effort.

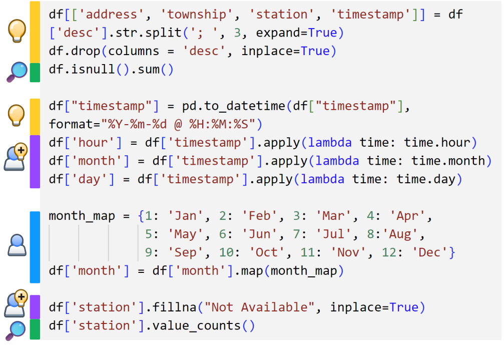
1 INTRODUCTION

Real-world data is often riddled with data quality issues, such as missing values, inconsistent/poor formatting, and duplicate entries. Data wrangling is an iterative process that involves data preparation for analysis by imputing missing values, performing appropriate typecasting and encoding, extracting features, removing duplicates, and so on. Data scientists spend up to 80% of their time in data extraction and preparation—continuously assessing and improving data quality by applying and validating chains of transformations—making wrangling a tedious and error-prone process [7].

Several libraries and interface designs allow effective data exploration and wrangling. However, the onus of writing correct code and learning specific tools still lies with the data scientist. With the advent of big data, organizations strive to be data-driven, and data analysis often needs to be performed by low-code or no-code users to propel business decisions. Several industry products enable visual data preparation with pre-built transformations and wrangling suggestions. However, the capabilities of such tools are limited due to the lack of human-readable code, which is essential to customize transformations, gain visibility, and replicate results across runs.

Example 1.1. Consider a 911 emergency calls dataset (Fig. 2), where the attribute `desc` describes call information—address, township, station, and timestamp—separated by semicolon (;). Liam, a data scientist wants to perform temporal analysis to identify reasons for emergency calls at different times of the day, month, and year. To prepare the data for analysis, Liam tries to split `desc` using semicolon as the delimiter using Python (pandas).

He believed he knew how to split a string column, but it takes 4 tries and a web search for Liam to find the correct API and parameters. Next, he casts the derived attribute `timestamp` to DateTime format, and subsequently derives attributes: `day`, `month`, and `hour`. Liam now checks for missing values and realizes that missing values in the derived attribute `station` must be imputed with the string “`Not Available`”. Fig. 1 displays a snapshot of the 11 lines of wrangling code authored by Liam in 38 minutes. □



```
df[['address', 'township', 'station', 'timestamp']] = df
['desc'].str.split('; ', 3, expand=True)
df.drop(columns = 'desc', inplace=True)
df.isnull().sum()

df["timestamp"] = pd.to_datetime(df["timestamp"],
format="%Y-%m-%d @ %H:%M:%S")
df['hour'] = df['timestamp'].apply(lambda time: time.hour)
df['month'] = df['timestamp'].apply(lambda time: time.month)
df['day'] = df['timestamp'].apply(lambda time: time.day)

month_map = {1: 'Jan', 2: 'Feb', 3: 'Mar', 4: 'Apr',
             5: 'May', 6: 'Jun', 7: 'Jul', 8:'Aug',
             9: 'Sep', 10: 'Oct', 11: 'Nov', 12: 'Dec'}
df['month'] = df['month'].map(month_map)

df['station'].fillna("Not Available", inplace=True)
df['station'].value_counts()
```

Figure 1: Snapshot of code authored by Liam. CoWrangler can automate generation of the code marked by 💡, and eliminate the need for code marked by 🔎. CoWrangler adopts a human-in-the-loop approach for custom operations marked by 👤 and 👤+.

Building a tool to ease and accelerate the task of data preparation is challenging. First, the tool needs to generate human-readable and efficient code because the user would want to read, understand, and edit the code to achieve the intended transformation in their desired development environment. Moreover, readable code promotes trust and transparency in the tool's working. While it is often possible to generate working code, it is challenging to generate code that the user would have preferred writing themselves. Second, the space of all possible code snippets is enormous. For instance, the pandas library has a wide breadth of an order of 100 methods, where each method has an order of 10 arguments. Data scientists (as seen in Example 1.1) find it increasingly difficult to recall the correct API method and its relevant arguments. Machines find it difficult to search the space of all possible programs to find the one the user may want [1]. Finally, there are often many valid data wrangling steps that can be presented as recommendations to expedite the wrangling process, and it is challenging to pick the most relevant snippets to show.

Related Work. Visual data preparation services like Mito[1], Bamboolib[2] and Databricks[3] fit in data scientists' workflows to aid data exploration and wrangling. Commercial tools, such as Google Cloud DataPrep (Trifacta)[4], AWS Glue DataBrew[5], and Einstein Discovery[6], and other work [4, 6], attempt to alleviate data wrangling

[1] trymito.io
[2] bamboolib.8080labs.com
[3] databricks.com/product/data-lakehouse
[4] docs.trifacta.com/display/SS/Overview+of+Predictive+Transformation
[5] docs.aws.amazon.com/databrew/latest/dg/
[6] help.salesforce.com/s/articleView?id=sf.bi_edd_prep_terminology.htm

barriers with visual data preparation interfaces, and automated recommendations. These tools enable professionals and as low-code users (e.g., users in management roles) to prepare data for analysis. However, due to the absence of code (no-code) for the applied wrangling steps, these tools prevent data scientists from (1) customizing suggested transformations and their parameters and authoring custom code, which is necessary to achieve the desired transformation, (2) trusting suggestions as there is no explanation (transparency) towards why (how) the suggestion is proposed (executed), and (3) saving/replicating interleaved workflows in their desired development environment. Data scientists commonly use languages such as Python, R, and SQL to author wrangling code, making it difficult for no-code suggestions to match their complete intent.

The need to accelerate data wrangling is also illustrated by code-first recommendations for varied transformations, such as data reshaping (join, pivot, unpivot), extraction, and error detection [9]. Recent work attempts to capture the user's intent in natural language (NL) to synthesize code using large language models (LLMs) [5]. However, to provide NL prompts, the user must know the exact wrangling steps and must have sufficient knowledge of the data at hand, which requires additional effort.

These limitations motivate the development of CoWrangler, a tool that aims to leverage the best of the three dimensions identified in prior work: (1) visual data preparation interfaces, (2) automated wrangling suggestions, and (3) code-first paradigm. CoWrangler presents meaningful and most-relevant wrangling suggestions with explanations, that not only automate, but also inform the wrangling process with a human-in-the-loop approach.

They key properties of CoWrangler are: (1) it exploits the user's data context to automatically generate data-wrangling suggestions using predictive program synthesis, along with explanations, (2) it is a visual and code-first tool that generates human-readable and efficient code for each wrangling suggestion, and (3) it ranks suggestions based on the principle that the wrangling step that improves the data quality more is more desirable. Consequently, CoWrangler provides users flexibility to use the suggested code as is or edit the suggested code to customize it.

2 SOLUTION SKETCH

CoWrangler generates wrangling suggestions for the user based on the data in their working set. We assume we have access to (a sample of) the dataset. We internally learn programs in a domain-specific language (DSL). Each program transforms tables (2D data grid) into new tables. Unlike program synthesis by example where users provide example output [2], we synthesize suggestions just based on the input table (without any user prompt), which is called *predictive program synthesis* [8]. The synthesized programs are internally executed over the input table, and then ranked based on their capability to improve the quality of the output table. Top-ranked programs are then translated to the user's target programming language and suggested as next possible wrangling steps.

2.1 Suggestion DSL

The suggestion DSL currently consists of a few popular data wrangling operators. However, CoWrangler is designed to be extensible and the DSL can be extended with new operators modularly. We analyzed 730 Python (Jupyter) notebooks from Kaggle to identify and include the most frequently occurring operators in the first version of CoWrangler:

(1) Drop column: This DSL operator indicates which columns to drop and the *reason* for dropping those columns; e.g., the column is (mostly) empty, has some constant value throughout, is a duplicate of another column, or is an index column that doesn't contribute to certain tasks.
(2) Drop row: This operator indicates whether to drop certain rows and the reasons. Two possible reasons are: the row is mostly empty, or it is a duplicate of another row.
(3) Fill missing values: This operator indicates the column with missing values, the markers for the missing values (e.g., NaN), and a replacement value (e.g., -1) or a mechanism to use to impute the missing values (e.g., mean, mode, or median).
(4) Encoding: The encoding operator indicates the column that can be either label-encoded or one-hot encoded.
(5) Split: This operator indicates the column whose string values can be split to create multiple new columns.
(6) Type cast: This operator indicates that the data in the identified column can be cast to the identified data type (datetime, integer, float, boolean, or categorical).

We focus on the above operators in this demonstration; however, we envision adding many more to CoWrangler.

2.2 Predictive Synthesis of Suggestions

Given an input table, the synthesis problem is to generate programs in the above DSL that denote meaningful data-wrangling steps, which can be applied on the input table. We solve this synthesis problem by creating separate *individual learners* for each of the top-level DSL operators described above. Each learner analyzes the input table and determines if the corresponding operation can be applied to the input table, and if so, it determines the best values for the various arguments of that operator. We finally collect programs learned by individual learners and rank them.

Certain individual learners are simple. For example, the drop row (or drop column) learner detects if the input table contains rows (columns) that are duplicates of another row (column). It similarly checks if the input table has mostly empty rows (columns). If either is true, then it generates the corresponding DSL program. We also have some complex learners. The "type-cast learner", for example, finds all possible types that are consistent with values in a column and then uses complex logic to disambiguate and pick a fixed type for that column. The "split learner" analyzes the strings in a column to find delimiting characters or strings that can be used to generate consistent splits across all column values [8].

2.3 Ranking using Data Quality Metric

Given a set of possible data wrangling operations that are learned in the above step, we next rank them to identify the most relevant operations to surface to the user. We rank by assigning a score to each operation. This score measures the improvement in terms of *data quality* of the output table obtained by applying that operation over the input table. When computing data quality, we penalize missing values and redundancy, while giving credit for uniformity in the form of data. Thus, data quality is high if the concrete data has high entropy (less redundancy), but its abstraction (as patterns or types) has low entropy (uniformity). While we use a certain data-quality metric in this work, this is a pluggable unit and CoWrangler can work with any user-defined notion of data-quality.

2.4 Translation

The top-ranked DSL programs are finally translated to a user-specified target language. Here we focus on Python (pandas), but CoWrangler can support translations to other languages such as pyspark and Power Query M. The translation generates the most performant code (using vector APIs wherever possible) and using comments and meaningful variables to improve code readability.

2.5 Preliminary Evaluation

We performed an initial evaluation over 730 notebooks obtained from Kaggle, consisting of 2248 transformations using pandas APIs. We found CoWrangler's DSL vocabulary supports **33%** (742 *of* 2248) transformations and CoWrangler's suggestions accurately predict **53.4%** (396 *of* 742) of the supported transformations.

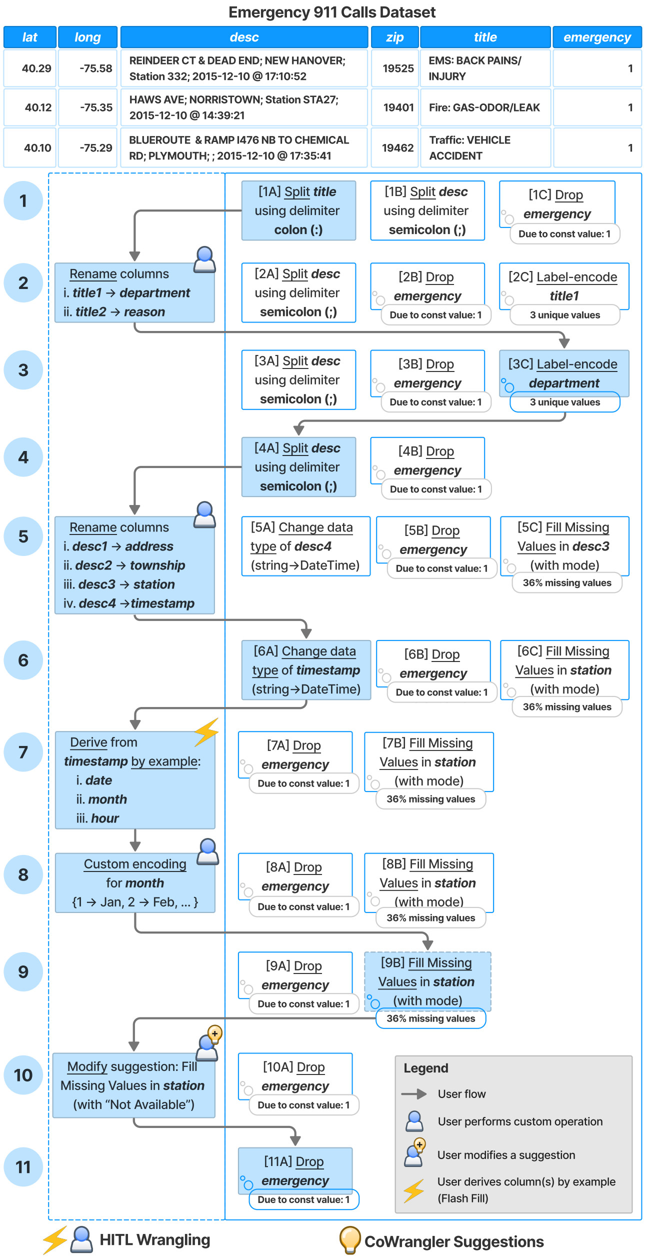
3 DEMONSTRATION

We will demonstrate CoWrangler on a real-world dataset of emergency 911 calls from Montgomery County, Pennsylvania[7]. It contains several attributes including `latitude`, `longitude`, `desc` (containing address, township, station, and timestamp of the call), `zip`, `title`, and `emergency`. Fig. 3 displays CoWrangler's user interface. Here, the user wants to pre-process the dataset to identify associations between variables, such as: reason of emergency, location, time, day of the week, and so on. Fig. 2 shows the user's data wrangling journey with CoWrangler. Below we describe the demonstration scenario based on Fig. 3:

• **Uploading the data:** The user first uploads a subset of the emergency 911 calls dataset Ⓐ, over which CoWrangler will generate wrangling suggestions.

• **View wrangling suggestions:** Once the data is loaded Ⓑ, CoWrangler displays wrangling suggestions in natural language Ⓒ. CoWrangler generates 4 different types of suggestions: (1) & (2) split column using delimiter, (3) drop column, (4) fill missing values, and (5) label-encode. Upon observing the suggestions and their explanations, the user immediately identifies important characteristics of the data: (1) missing values in `zip`, (2) categorical nature of `title`, (3) common pattern across `desc` and `title`, and (4) `emergency` being a constant valued column. This provides data insights and a quick and informed assessment of suggestions, along with identification of other actionable wrangling steps.

• **Select and preview a suggestion:** The user finds the first suggestion, Split `title` using delimiter **colon (:)**, relevant for feature extraction and selects it. CoWrangler then generates code in the selected language (Python pandas) Ⓓ and presents a preview of the two new derived columns (`title1` and `title2`) denoted in green Ⓔ. Before accepting the suggestion provided by CoWrangler, the user inspects the generated code and to-be-transformed data preview to validate the suggested transformation.



Emergency 911 Calls Dataset

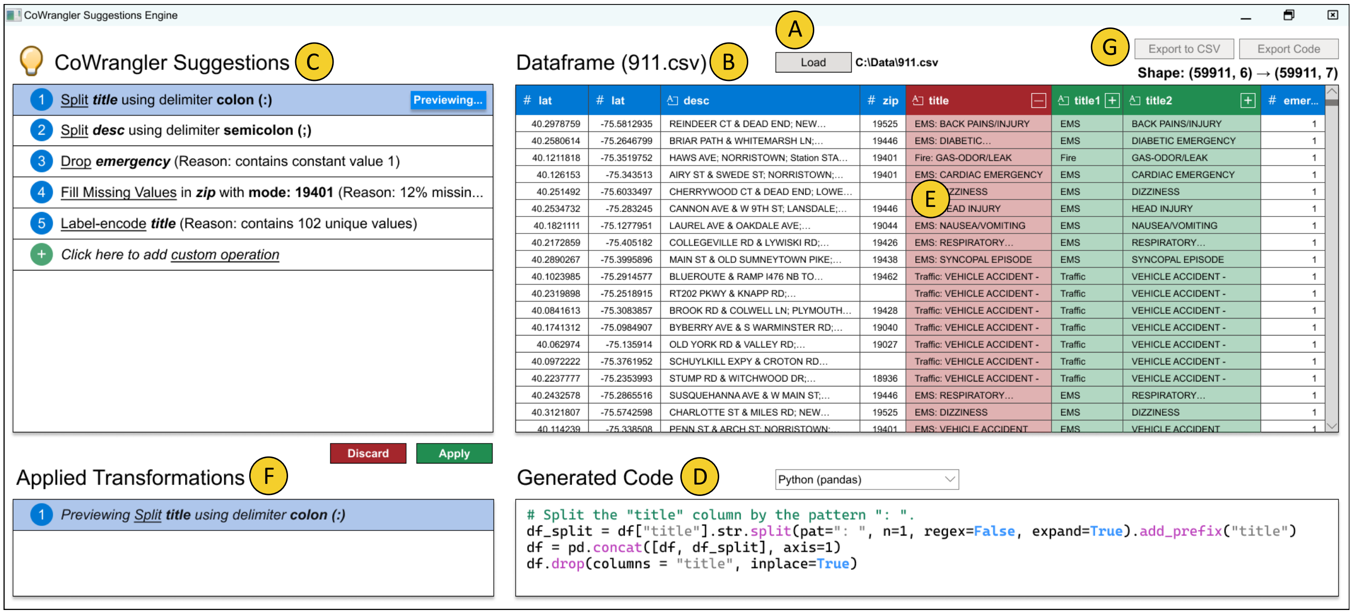
lat	long	desc	zip	title	emergency
40.29	-75.58	REINDEER CT & DEAD END; NEW HANOVER; Station 332; 2015-12-10 @ 17:10:52	19525	EMS: BACK PAINS/ INJURY	1
40.12	-75.35	HAWS AVE; NORRISTOWN; Station STA27; 2015-12-10 @ 14:39:21	19401	Fire: GAS-ODOR/LEAK	1
40.10	-75.29	BLUEROUTE & RAMP I476 NB TO CHEMICAL RD; PLYMOUTH; ; 2015-12-10 @ 17:35:41	19462	Traffic: VEHICLE ACCIDENT	1

1
[1A] Split title using delimiter colon (:)
[1B] Split desc using delimiter semicolon (;)
[1C] Drop emergency
Due to const value: 1
2
Rename columns i. title1 → department ii. title2 → reason
[2A] Split desc using delimiter semicolon (;)
[2B] Drop emergency
Due to const value: 1
[2C] Label-encode title1
3 unique values
3
[3A] Split desc using delimiter semicolon (;)
[3B] Drop emergency
Due to const value: 1
[3C] Label-encode department
3 unique values
4
[4A] Split desc using delimiter semicolon (;)
[4B] Drop emergency
Due to const value: 1
5
Rename columns i. desc1 → address ii. desc2 → township iii. desc3 → station iv. desc4 →timestamp
[5A] Change data type of desc4 (string→DateTime)
[5B] Drop emergency
Due to const value: 1
[5C] Fill Missing Values in desc3 (with mode)
36% missing values
6
[6A] Change data type of timestamp (string→DateTime)
[6B] Drop emergency
Due to const value: 1
[6C] Fill Missing Values in station (with mode)
36% missing values
7
Derive from timestamp by example: i. date ii. month iii. hour
[7A] Drop emergency
Due to const value: 1
[7B] Fill Missing Values in station (with mode)
36% missing values
8
Custom encoding for month {1 → Jan, 2 → Feb, ... }
[8A] Drop emergency
Due to const value: 1
[8B] Fill Missing Values in station (with mode)
36% missing values
9
[9A] Drop emergency
Due to const value: 1
[9B] Fill Missing Values in station (with mode)
36% missing values
10
Modify suggestion: Fill Missing Values in station (with "Not Available")
[10A] Drop emergency
Due to const value: 1
11
[11A] Drop emergency
Due to const value: 1
Legend
User flow
User performs custom operation
User modifies a suggestion
User derives column(s) by example (Flash Fill)
HITL Wrangling
CoWrangler Suggestions

Figure 2: A user's 11 step journey with CoWrangler to prepare their dataset for visualization tasks. Arrows indicate the user flow, with the applied transformations highlighted in blue. For each step, the first column shows HITL actions (if any), while the second column shows an ordered list (from left to right) of CoWrangler suggestions in each row.

• **Accept a suggestion:** Once convinced, the user clicks 'Apply' to accept the suggested transformation, which then reflects in the 'Applied Transformations' pane Ⓕ. CoWrangler then refreshes the data preview Ⓑ and the suggestions pane Ⓒ (not visible in Fig. 3), populating it with suggestions for the updated data.

[7] https://www.kaggle.com/datasets/mchirico/montcoalert



CoWrangler Suggestions

1. Split *title* using delimiter **colon (:)**
2. Split *desc* using delimiter **semicolon (;)**
3. Drop *emergency* (Reason: contains constant value 1)
4. Fill Missing Values in *zip* with **mode: 19401** (Reason: 12% missin...
5. Label-encode *title* (Reason: contains 102 unique values)

+ *Click here to add custom operation*

Dataframe (911.csv)

C:\Data\911.csv

Shape: (59911, 6) → (59911, 7)

# lat	# lat	desc	# zip	title	title1	title2	# emer...
40.2978759	-75.5812935	REINDEER CT & DEAD END; NEW…	19525	EMS: BACK PAINS/INJURY	EMS	BACK PAINS/INJURY	1
40.2580614	-75.2646799	BRIAR PATH & WHITEMARSH LN;…	19446	EMS: DIABETIC…	EMS	DIABETIC EMERGENCY	1
40.1211818	-75.3519752	HAWS AVE; NORRISTOWN; Station STA…	19401	Fire: GAS-ODOR/LEAK	Fire	GAS-ODOR/LEAK	1
40.126153	-75.343513	AIRY ST & SWEDE ST; NORRISTOWN;…	19401	EMS: CARDIAC EMERGENCY	EMS	CARDIAC EMERGENCY	1
40.251492	-75.6033497	CHERRYWOOD CT & DEAD END; LOWE…		ZZINESS	EMS	DIZZINESS	1
40.2534732	-75.283245	CANNON AVE & W 9TH ST; LANSDALE;…	19446	EAD INJURY	EMS	HEAD INJURY	1
40.1821111	-75.1277951	LAUREL AVE & OAKDALE AVE;…	19044	EMS: NAUSEA/VOMITING	EMS	NAUSEA/VOMITING	1
40.2172859	-75.405182	COLLEGEVILLE RD & LYWISKI RD;…	19426	EMS: RESPIRATORY…	EMS	RESPIRATORY…	1
40.2890267	-75.3995896	MAIN ST & OLD SUMNEYTOWN PIKE;…	19438	EMS: SYNCOPAL EPISODE	EMS	SYNCOPAL EPISODE	1
40.1023985	-75.2914577	BLUEROUTE & RAMP I476 NB TO…	19462	Traffic: VEHICLE ACCIDENT -	Traffic	VEHICLE ACCIDENT -	1
40.2319898	-75.2518915	RT202 PKWY & KNAPP RD;…		Traffic: VEHICLE ACCIDENT -	Traffic	VEHICLE ACCIDENT -	1
40.0841613	-75.3083857	BROOK RD & COLWELL LN; PLYMOUTH…	19428	Traffic: VEHICLE ACCIDENT -	Traffic	VEHICLE ACCIDENT -	1
40.1741312	-75.0984907	BYBERRY AVE & S WARMINSTER RD;…	19040	Traffic: VEHICLE ACCIDENT -	Traffic	VEHICLE ACCIDENT -	1
40.062974	-75.135914	OLD YORK RD & VALLEY RD;…	19027	Traffic: VEHICLE ACCIDENT -	Traffic	VEHICLE ACCIDENT -	1
40.0972222	-75.3761952	SCHUYLKILL EXPY & CROTON RD…		Traffic: VEHICLE ACCIDENT -	Traffic	VEHICLE ACCIDENT -	1
40.2237777	-75.2353993	STUMP RD & WITCHWOOD DR;…	18936	Traffic: VEHICLE ACCIDENT -	Traffic	VEHICLE ACCIDENT -	1
40.2432578	-75.2865516	SUSQUEHANNA AVE & W MAIN ST;…	19446	EMS: RESPIRATORY…	EMS	RESPIRATORY…	1
40.3121807	-75.5742598	CHARLOTTE ST & MILES RD; NEW…	19525	EMS: DIZZINESS	EMS	DIZZINESS	1
40.114239	-75.338508	PENN ST & ARCH ST; NORRISTOWN;…	19401	EMS: VEHICLE ACCIDENT	EMS	VEHICLE ACCIDENT	1

Applied Transformations

1. *Previewing Split **title** using delimiter **colon (:)***

Generated Code

```
# Split the "title" column by the pattern ": ".
df_split = df["title"].str.split(pat=": ", n=1, regex=False, expand=True).add_prefix("title")
df = pd.concat([df, df_split], axis=1)
df.drop(columns = "title", inplace=True)
```

Figure 3: Demonstrating CoWrangler: (A) Upload reference data; (B) 911 calls data is loaded into the data preview panel; (C) CoWrangler populates the suggestions pane with wrangling suggestions. The user selects the first suggestion, highlighted in blue; (D) Human-readable pandas code appears in the 'Generated Code' panel; (E) Data preview panel highlights changes that will be made if the suggestion is accepted; (F) Wrangling history is tracked in the 'Applied Transformations' panel; (G) Export the resulting CSV file and code after wrangling.

- **Human-in-the-loop wrangling:** The user now wants to rename the newly derived columns: `title1` to `department`, and `title2` to `reason`. To achieve this, the user selects 'Click here to add custom operation' from the suggestions pane Ⓒ and writes the desired code using the pandas `rename` API in the code editor Ⓓ. The user then applies their custom operation to the dataset, which gets appended to the 'Applied Transformations' pane Ⓕ.

Users can collaborate with CoWrangler in three different modes to customize wrangling transformations as desired (Fig. 2): (1) editing the generated code while previewing (steps 9–10), (2) expressing intent by providing examples with Flash Fill [3] (step 7), and (3) authoring code from scratch (steps 2, 5, and 8).

- **Select and preview subsequent suggestion:**When the data preview and suggestions get refreshed Ⓑ & Ⓒ, a new suggestion—Label-encode `department` (REASON: contains 3 unique values)—appears and the user selects this suggestion. They validate the transformation by previewing the data and generated code.
- **Accept subsequent suggestion:** The user accepts the label-encode suggestion, which is appended to the 'Applied Transformations' pane Ⓕ. CoWrangler again refreshes the data preview Ⓑ and wrangling suggestions Ⓒ and the above processes go on.
- **Export generated code and wrangled data:** Once the user has made the desired transformations, they can export the transformed data to a CSV file and export the code to a notebook Ⓖ.

Demonstration engagement. Following our demonstration, participants can load their own datasets into CoWrangler to receive automated wrangling suggestions depending on their data context. They can then preview the effects of different suggestions and make modifications to the generated code to fullfil their requirements.

4 CONCLUSIONS AND FUTURE WORK

This demonstration is our attempt to inform and accelerate the data wrangling process by (1) recommending the most meaningful wrangling transformations with explanations in real-time, (2) generating human-readable and efficient wrangling code, and (3) enabling flexibility with human-in-the-loop interactions for intent expression. In future, we aim to extend CoWrangler's vocabulary to increase the coverage of suggestions, and add capabilities to understand data semantics for more contextualized and smarter suggestions.

REFERENCES

[1] R Bavishi, C Lemieux, R Fox, K Sen, and I Stoica. 2019. AutoPandas: Neural-Backed Generators for Program Synthesis. *Proc. ACM Program. Lang.* OOPSLA, Article 168 (oct 2019), 27 pages.

[2] A Fariha and A Meliou. 2019. Example-Driven Query Intent Discovery: Abductive Reasoning using Semantic Similarity. *PVLDB* 12, 11 (2019), 1262–1275.

[3] S Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. In *POPL*. 317–330.

[4] P J Guo, S Kandel, J Hellerstein, and J Heer. 2011. Proactive Wrangling: Mixed-Initiative End-User Programming of Data Transformation Scripts. In *UIST*. 65–74.

[5] N Jain, S Vaidyanath, A Iyer, N Natarajan, S Parthasarathy, S Rajamani, and R Sharma. 2022. Jigsaw: Large Language Models Meet Program Synthesis. In *ICSE*. 1219–1231.

[6] S Kandel, A Paepcke, J Hellerstein, and J Heer. 2011. Wrangler: Interactive Visual Specification of Data Transformation Scripts. In *CHI*. 3363–3372.

[7] G Press. 2016. Cleaning Big Data. https://www.forbes.com/sites/gilpress/2016/03/23/data-preparation-most-time-consuming-least-enjoyable-data-science-task-survey-says/

[8] M Raza and S Gulwani. 2017. Automated data extraction using predictive program synthesis. In *AAAI*, Vol. 31.

[9] C Yan and Y He. 2020. Auto-Suggest: Learning-to-Recommend Data Preparation Steps Using Data Science Notebooks. In *SIGMOD*. 1539–1554.