Hasso Plattner Institute
University Potsdam, Germany
Information Systems Group

# Discovering Metadata in Data Files

**Dissertation**
**zur Erlangung des akademischen Grades**
**“Doktor der Naturwissenschaften”**
**(Dr. rer. nat.)**
**in der Wissenschaftsdisziplin “Informationssysteme”**

**eingereicht an der**
**Fakultät Digital Engineering**
**der Universität Potsdam**

**von**
**Lan Jiang**

**Dissertation, Universität Potsdam, 2021**

Hasso Plattner Institute
University of Potsdam
Information Systems Group

# Discovering Metadata in Data Files

**Dissertation**
**zur Erlangung des akademischen Grades**
**“Doktor der Naturwissenschaften”**
**(Dr. rer. nat.)**
**in der Wissenschaftsdisziplin “Informationssysteme”**

**eingereicht an der**
**Fakult¨at Digital Engineering**
**der Universit¨at Potsdam**

**von**
**Lan Jiang**

**Potsdam, den 14. Dezember 2021**

## Reviewers

Prof. Dr. Felix Naumann
Hasso Plattner Institute, University of Potsdam

Prof. Dr. Bernhard Mitschang
Institute of Parallel and Distributed Systems, University of Stuttgart

Prof. Dr. Renée Miller
Khoury College of Computer Sciences, Northeastern University

Published online on the
Publication Server of the University of Potsdam:
https://doi.org/10.25932/publishup-56620
https://nbn-resolving.org/urn:nbn:de:kobv:517-opus4-566204

# Abstract

It is estimated that data scientists spend up to 80% of the time exploring, cleaning, and transforming their data. A major reason for that expenditure is the lack of knowledge about the used data, which are often from different sources and have heterogeneous structures. As a means to describe various properties of data, *metadata* can help data scientists understand and prepare their data, saving time for innovative and valuable data analytics. However, metadata do not always exist: some data file formats are not capable of storing them; metadata were deleted for privacy concerns; legacy data may have been produced by systems that were not designed to store and handle metadata. As data are being produced at an unprecedentedly fast pace and stored in diverse formats, manually creating metadata is not only impractical but also error-prone, demanding automatic approaches for metadata detection.

In this thesis, we are focused on detecting metadata in CSV files – a type of plain-text file that, similar to spreadsheets, may contain different types of content at arbitrary positions. We propose a taxonomy of metadata in CSV files and specifically address the discovery of three different metadata: *line* and *cell type*, *aggregations*, and *primary keys* and *foreign keys*.

Data are organized in an ad-hoc manner in CSV files, and do not follow a fixed structure, which is assumed by common data processing tools. Detecting the structure of such files is a prerequisite of extracting information from them, which can be addressed by detecting the semantic type, such as header, data, derived, or footnote, of each line or each cell. We propose the supervised-learning approach Strudel to detect the type of lines and cells. CSV files may also include aggregations. An aggregation represents the arithmetic relationship between a numeric cell and a set of other numeric cells. Our proposed AggreCol algorithm is capable of detecting aggregations of five arithmetic functions in CSV files. Note that stylistic features, such as font style and cell background color, do not exist in CSV files. Our proposed algorithms address the respective problems by using only *content*, *contextual*, and *computational* features.

Storing a relational table is also a common usage of CSV files. Primary keys and foreign keys are important metadata for relational databases, which are usually not present for database instances dumped as plain-text files. We propose the HoPF algorithm to holistically detect both constraints in relational databases. Our approach is capable of distinguishing true primary and foreign keys from a great amount of spurious *unique column combinations* and *inclusion dependencies*, which can be detected by state-of-the-art data profiling algorithms.

# Zusammenfassung

Schätzungen zufolge verbringen Datenwissenschaftler bis zu 80% ihrer Zeit mit der Erkundung, Bereinigung und Umwandlung ihrer Daten. Ein Hauptgrund für diesen Aufwand ist das fehlende Wissen über die verwendeten Daten, die oft aus unterschiedlichen Quellen stammen und heterogene Strukturen aufweisen. Als Mittel zur Beschreibung verschiedener Dateneigenschaften können *Metadaten* Datenwissenschaftlern dabei helfen, ihre Daten zu verstehen und aufzubereiten, und so wertvolle Zeit die Datenanalysen selbst sparen. Metadaten sind jedoch nicht immer vorhanden: Zum Beispiel sind einige Dateiformate nicht in der Lage, sie zu speichern; Metadaten können aus Datenschutzgründen gelöscht worden sein; oder ältere Daten wurden möglicherweise von Systemen erzeugt, die nicht für die Speicherung und Verarbeitung von Metadaten konzipiert waren. Da Daten in einem noch nie dagewesenen Tempo produziert und in verschiedenen Formaten gespeichert werden, ist die manuelle Erstellung von Metadaten nicht nur unpraktisch, sondern auch fehleranfällig, so dass automatische Ansätze zur Metadatenerkennung erforderlich sind.

In dieser Arbeit konzentrieren wir uns auf die Erkennung von Metadaten in CSV-Dateien - einer Art von Klartextdateien, die, ähnlich wie Tabellenkalkulationen, verschiedene Arten von Inhalten an beliebigen Positionen enthalten können. Wir schlagen eine Taxonomie der Metadaten in CSV-Dateien vor und befassen uns speziell mit der Erkennung von drei verschiedenen Metadaten: *Zeile* und *Zellensemantischer Typ*, *Aggregationen* sowie *Primärschlüssel* und *Fremdschlüssel*.

Die Daten sind in CSV-Dateien ad-hoc organisiert und folgen keiner festen Struktur, wie sie von gängigen Datenverarbeitungsprogrammen angenommen wird. Die Erkennung der Struktur solcher Dateien ist eine Voraussetzung für die Extraktion von Informationen aus ihnen, die durch die Erkennung des semantischen Typs jeder Zeile oder jeder Zelle, wie z. B. Kopfzeile, Daten, abgeleitete Daten oder Fußnote, angegangen werden kann. Wir schlagen den Ansatz des überwachten Lernens, genannt "Strudel" vor, um den strukturellen Typ von Zeilen und Zellen zu klassifizieren. CSV-Dateien können auch Aggregationen enthalten. Eine Aggregation stellt die arithmetische Beziehung zwischen einer numerischen Zelle und einer Reihe anderer numerischer Zellen dar. Der von uns vorgeschlagene "AggreCol"-Algorithmus ist in der Lage, Aggregationen von fünf arithmetischen Funktionen in CSV-Dateien zu erkennen. Da stilistische Merkmale wie Schriftart und Zellhintergrundfarbe in CSV-Dateien nicht vorhanden sind, die von uns vorgeschlagenen Algorithmen die entsprechenden Probleme, indem sie nur die Merkmale *Inhalt*, *Kontext* und *Berechnungen* verwenden.

Die Speicherung einer relationalen Tabelle ist ebenfalls eine häufige Verwendung von CSV-Dateien. Primär- und Fremdschlüssel sind wichtige Metadaten für relationale Datenbanken, die bei Datenbankinstanzen, die als reine Textdateien gespeichert werden, normalerweise nicht vorhanden sind. Wir schlagen den "HoPF"-Algorithmus vor, um beide Constraints in relationalen Datenbanken ganzheitlich zu erkennen. Unser Ansatz ist in der Lage, echte Primär- und Fremdschlüssel von einer großen Menge an falschen *eindeutigen Spaltenkombinationen* und *Einschlussabhängigkeiten* zu unterscheiden, die von modernen Data-Profiling-Algorithmen erkannt werden können.

# Acknowledgments

Acquiring a Ph.D. title is by no means easy. It is a task that I could not have finished by myself without the supports and help from many others. Cordially, I would like to express my gratitude to all those that care about me. The very first person I want to thank is my doctoral advisor Prof. Dr. Felix Naumann who constantly supports me with my study and work. His profound knowledge in this discipline and honest attitude towards science have taught me a lot from thinking deep and performing rigorous research to writing down precise sentences in papers, and from holding a critical mindset to communicating and collaborating with others. These skills I learned from working with him have become treasures to me, and give me greater confidence to deal with new challenges in my future works. I would also like to thank all the teachers and professors who passed on knowledge to me.

I also want to thank all the excellent colleagues that have worked with me during my Ph.D. journey. I am grateful for their selfless efforts to support me. I want to especially thank the data preparation team. That we collaborated with and learned from each other are an unforgettable and valuable experience to me. May all my colleagues have a great and bright future.

Finally, I want to thank my friends and family, who always stood by my side to support and encourage me in every way possible, even though they are scattered all over the world. Their help and upholding let me make it through all the difficult time, especially during the lockdown time. Especially, thanks to my parents who have spent so many efforts raising me up and shaping me into a good person, and thanks to Tim Felgentreff and Felicia Flemming who helped me settle down in Germany and get used to living here.

Jumping out of the comfort zone needs courage, especially when it is about starting a Ph.D. in a new environment. Looking back to the past years as a doctoral student, I am very happy that I made this decision and proud of myself to get through it. This experience is certainly a highlight of my life. Again, thanks to all that care about and support me.

# Contents

# Chapter 1

# Metadata in Data Files

Data are being rapidly generated over the past years. It is estimated that the amount of new data created will reach 180 zettabytes ($1.8 \times 1^{23}$ bytes) in 2025, almost three times the volume of that in 2020 [Holst, 2021]. The COVID-19 pandemic has propelled the growth of new data produced due to home office and homeschooling, which have become a new routine to many more people since the initial outbreak of the disease. There is no doubt that the abundance of data could bring great opportunities to reinforce the productivity of various jobs, or to spark new technologies, such as assisting doctors in personalizing healthcare solutions to their patients [Ahuja, 2019; Dimitrov, 2016] and fighting global pandemic [Islam et al., 2020], supporting business decisions [Fountaine et al., 2019], new drug discovery [Chan et al., 2019; Smalley, 2017], urban traffic monitoring [Jain et al., 2019; Mandal et al., 2020], or autonomous driving [Yurtsever et al., 2020]. The frequently quoted phrase "data is the new oil" [Bhageshpur, 2019; Economist, 2017; Palmer, 2006] has revealed the ambitions of exploiting data in a great deal of scenarios.

This metaphor, however, also implies the challenge of consuming data. Just like oil is not useful until it is refined and transformed into various kinds of fuels, data themselves laying in the hardware are not valuable until they are refined, or more specifically, cleaned, processed, and transformed into proper digital assets. The necessity to refine data comes from the fact that said data are often unstructured, faulty, inconsistent, and heterogeneous [Oliveira et al., 2005; Rahm and Do, 2000]. Being usually not ready for consumption by downstream analytics, data need to be prepared first. Data preparation is by no means a trivial task. In reality, it is estimated that data scientists spend in their day-to-day work 50% to 80% of the time only on preparing their data [Dasu and Johnson, 2003; Kandel et al., 2012; Press, 2016], which typically includes gathering, loading, cleaning, and transforming data. Preparing data is not only time-consuming, but also unpleasant: according to a survey by Forbes, 76% of 80 interviewed data scientists considered data preparation as the least enjoyable part of their work [Press, 2016].

That data scientists spend a good deal of time preparing data is attributed largely to the lack of knowledge about data. Collecting, loading, cleaning, transforming, and visualizing data are all based on understanding the involved data. The data lake techniques rapidly developed in recent years [Giebler et al., 2019] enable access to a wide variety of data that are, however, often subject to a particular schema and stored in heterogeneous formats. Without domain-specific knowledge for comprehension, data scientists are often

reluctant to use, or not able to fully exploit the data [Foshay et al., 2007]. To prevent letting a data lake from becoming a data swamp, we must be able to describe the characteristics of data stored in it [Eichler et al., 2021; Halevy et al., 2016; Nargesian et al., 2019]. A great number of properties about data can help data scientists understand their data, e.g., the size, format, provenance of data files, the structure and semantic of content, the constraints on content, and the relationships amongst different pieces of data. These properties about data are commonly referred to as *metadata*. The amount and type of metadata can develop far beyond those introduced above. There may also be metadata specific to file format or data domain. Having the knowledge of metadata, data scientists can leverage particular tools or develop unique scripts to further process, visualize, and analyze their data.

Metadata are useful information throughout a data science lifecycle. Figure 1.1 demonstrates a typical workflow for data science projects. Given a dataset collected from for example a data lake, a data scientist first explores it with data exploration tools, such as Power BI and Tableau for tabular data, to gain an understanding of or spot errors in the dataset. With the knowledge of data characteristics and quality, the data scientist further prepares the dataset, aligning it to the models they want to build for particular applications. After that, they build and validate models based on the prepared data. With new data problems discovered during this phase, the data scientist may return to data exploration to gain more knowledge about the data, or data preparation to further polish the data, before serving the prepared data to the models again. All above operations require comprehending the data, which can be facilitated with various metadata.

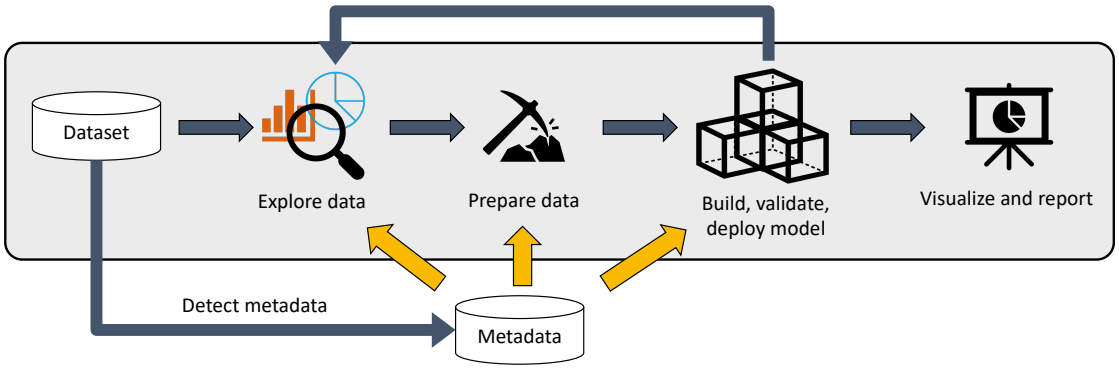



Figure 1.1: A typical workflow for data science projects.

Despite their usefulness, metadata are often not attached to the data they describe due to four main reasons. One reason is the inability of some file formats to hold metadata. For example, the widely used CSV format, according to RFC 4180 [Shafranovich, 2005], is designed to hold only tabular data. As a consequence, either there are no metadata to describe the data in such a file, or metadata are deposited in a separated file that is often not bundled with the data file. The second reason is the unavailability of metadata due to operational factors. For example, deleting complex metadata may reduce the size of data significantly, and therefore enable data loading on machines with limited hardware resources. Also, some legacy data may be produced by systems that were initially not designed to handle and store metadata. Last but not least, collecting

metadata is usually not a trivial task. On the one hand, some metadata are not obvious. For example, in a spreadsheet file with many tables scattered all over the place, counting the number of tables, recognizing the boundary of each table, and distinguishing among the different types of content in tables are all challenging tasks. On the other hand, the efforts to recognize metadata in one dataset may not always be useful to extract metadata from another dataset, where data may be organized in a different way. Data scientists often need to create a specific data process workflow to identify per-dataset metadata. As a consequence, manually scrutinizing and creating these metadata are not only time-consuming but also error-prone. The ever-growing amount of data aggravates the difficulty of the metadata extraction problem. Data files stored in heterogeneous formats are stored in different platforms, such as open-data portals, companies' on-premises data lakes. In practice, data scientists often need to query various sources to obtain data relevant to their analytics tasks. The volume of available data may be so large that manually gathering metadata of individual data files is not feasible.

While metadata detection is a general research problem for any type of data files, in this thesis, we focus on *verbose CSV files* – a type of plain-text data files that store semi-structured data. Different from a standard CSV file that, according to RFC 4180 [Shafranovich, 2005], contains an optional header line at the beginning of the file followed by a number of data lines, a verbose CSV file may include cells of heterogeneous classes, possibly with empty visual separators. Standard CSV files are a special case of verbose CSV files. The formal definition of verbose CSV files will be presented in Section 2.1.1. As of now, we demonstrate a typical real-world verbose CSV file with Figure 1.2, which includes three tables at different positions. A collection of metadata that can be derived from this file are marked in the figure.

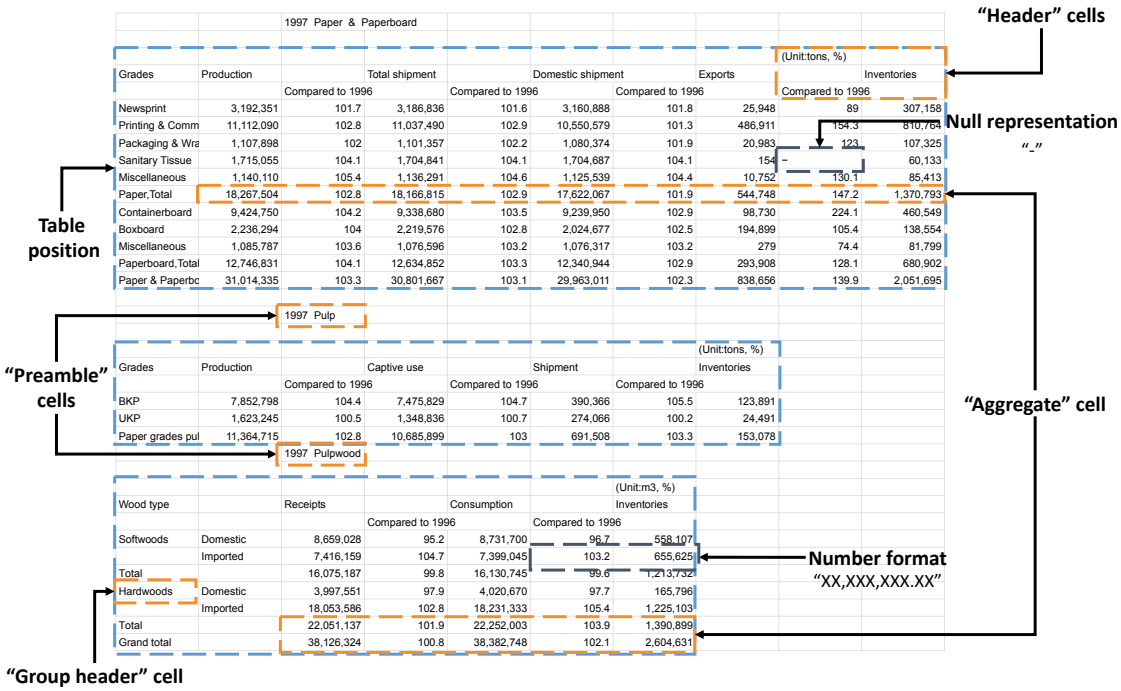



Figure 1.2: A real-world verbose CSV file that includes three tables. Samples of metadata are displayed.

Many more different types of metadata can be created for verbose CSV files. In the next section, we first introduce our proposed taxonomy of metadata in verbose CSV files and give several metadata examples. After that, we introduce the research challenges for metadata detection in data files and conclude this chapter by presenting the structure of the following chapters.

## 1.1 A Metadata Taxonomy

There are many ways to classify metadata that can be extracted from data files [Bilalli et al., 2016; Foshay et al., 2007; Varga et al., 2014]. In the context of verbose CSV files, we categorize metadata into six groups according to their purpose: (i) utility metadata; (ii) cell metadata; (iii) column metadata; (iv) row metadata; (v) table metadata; (vi) file metadata. The rest of this section elaborates on these categories.

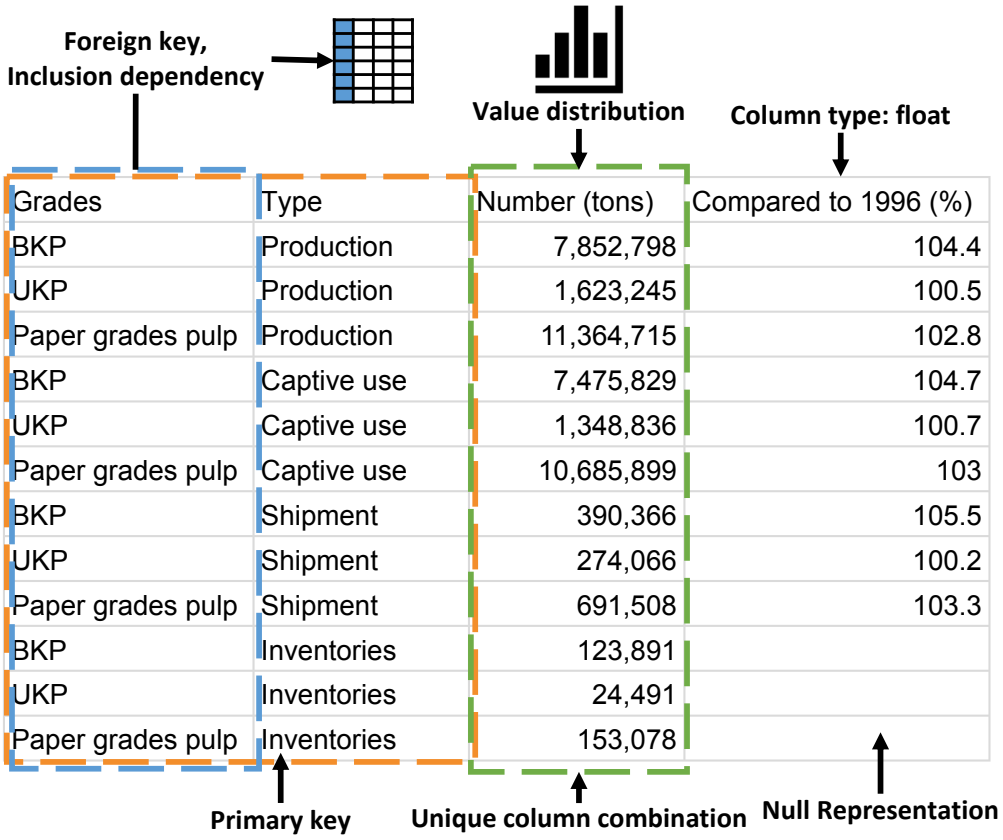
Utility metadata describe the form characters leveraged to interpret the structure of content, whereas the other five types of metadata (content metadata) depict properties of characters with real meaning in verbose CSV files, according to their scope. We discuss the five types of content metadata in both verbose CSV files with arbitrary layouts (e.g., Figure 1.2) and the ones following the RFC 4180 standard (e.g., Figure 1.3).

Figure 1.4 demonstrates our proposed metadata taxonomy and some examples in each group. In the following, we provide a detailed explanation of these six types along with the concrete metadata that fall into either group.

### 1.1.1 Utility metadata

Utility metadata describe how to interpret the structure of data files. In verbose CSV files, there are two types of characters: (i) *utility characters* that serve functional purposes to enable interpreting the structure of the content in files; (ii) *content characters* that represent the content with real meaning in data. Utility characters are a set of syntactic metadata that carry no real meaning, and should not be considered when depicting the content. Typical utility metadata in verbose CSV files include *file encoding*, *delimiter*, *quotation characters* of fields, *escape characters*, *line-break characters*, and so on. Figure 1.5 shows an example of three utility metadata used for the lines three to six in the file of Figure 1.2.

Detecting utility metadata is obviously the prerequisite of identifying the other five types of metadata in verbose CSV files, because utility metadata provide a way to separate characters with real meaning into correct fields. Utility metadata detection in verbose CSV files itself is an interesting and also challenging research problem. We have witnessed several research efforts to recognize the aforementioned utility metadata [Döhmen et al., 2017; Ge et al., 2019; Li and Momoi, 2001; Pinkas, 2014; van den Burg et al., 2019]. In this thesis, our focus is on identifying metadata about content, i.e., those of the other five types. Therefore, we do not address the utility metadata detection problem. Instead, we assume that these metadata have been correctly identified, and content in raw data files have been properly placed into respective cells.




| Grades | Type | Number (tons) | Compared to 1996 (%) |
|---|---|---|---|
| BKP | Production | 7,852,798 | 104.4 |
| UKP | Production | 1,623,245 | 100.5 |
| Paper grades pulp | Production | 11,364,715 | 102.8 |
| BKP | Captive use | 7,475,829 | 104.7 |
| UKP | Captive use | 1,348,836 | 100.7 |
| Paper grades pulp | Captive use | 10,685,899 | 103 |
| BKP | Shipment | 390,366 | 105.5 |
| UKP | Shipment | 274,066 | 100.2 |
| Paper grades pulp | Shipment | 691,508 | 103.3 |
| BKP | Inventories | 123,891 | |
| UKP | Inventories | 24,491 | |
| Paper grades pulp | Inventories | 153,078 | |

Figure 1.3: The relational table obtained by generating relational tuples from the content of the table in the middle in Figure 1.2. We define the primary key and a foreign key in this table for demonstration purposes. The foreign key shall reference the primary key of another table that is omitted.

### 1.1.2 Cell metadata

Cell metadata describe properties of cells – the smallest units – in verbose CSV files. Example cell metadata are cell length (the number of characters in the value of a cell), cell data type (e.g., integer, float, date, or string), date/number format (the pattern of the cell value that represents a date/number), cell type (e.g., preamble, header, aggregation, or footnote). Figure 1.2 demonstrates several cell metadata. Because verbose CSV files do not always follow a fixed row-wise or column-wise structure, cell-level metadata provide fine-grained details about the data, which could help recognize the presence of errors in data. Individual cell metadata may also be used to construct other types of content metadata. For example, the boundary of a table in a verbose CSV file may be determined by the positions of the data and header cells in it.

### 1.1.3 Column metadata

Column metadata depict characteristics of individual table columns in verbose CSV files. As a special case of verbose CSV files, standard CSV files contain one and only one

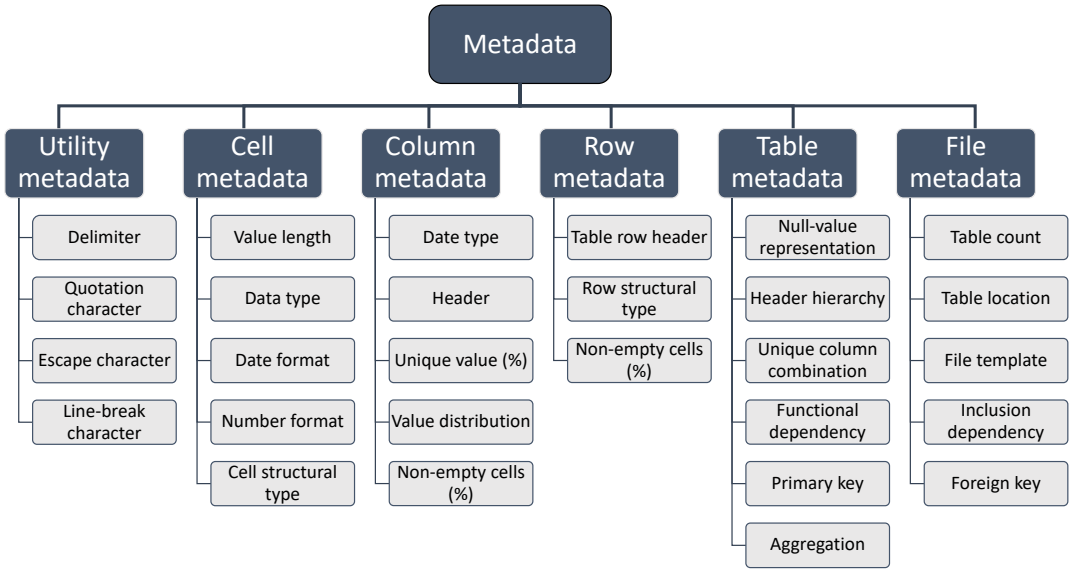



Figure 1.4: A taxonomy of metadata in verbose CSV files.

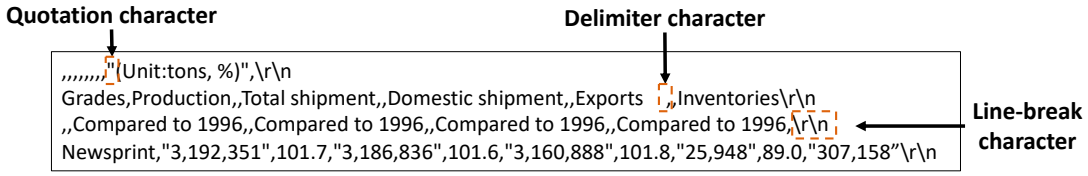



Figure 1.5: The raw character sequences of lines 3-6 in the verbose CSV file shown in Figure 1.2 with three utility metadata: comma as the delimiter, double-quote as the quotation character, and "`\r\n`" as the line-break character.

relational table. Therefore, a column therein includes all cells in the same vertical line. A number of column metadata can be extracted from standard CSV files, e.g., the column data type, format of its values, header of the column, percentage of unique values, and value distribution. In the case of verbose CSV files, such as those in Figure 1.2, columns may have a flexible arrangement of cells with different purposes, instead of an optional header cell at the top plus a set of data cells following. There are two types of column metadata in verbose CSV files: (i) *file column metadata* that describe the properties of all cells in the same vertical line of the file, such as the percentage of non-empty cells; (ii) *table column metadata* that depict the characteristics of all cells in one table of a presumable multi-table file, such as the scope of header/data, and the value distribution of the data part. Several previous works can be applied to verbose CSV files to determine the scope of tables, and the range of header and data parts in tables [Christodoulakis et al., 2020; Embley et al., 2016; Koci et al., 2017; Vitagliano et al., 2021], based on which we can identify table column metadata.

### 1.1.4 Row metadata

Similar to column metadata, we can divide row metadata into two groups: (i) file row metadata; (ii) table row metadata. A file row in a verbose CSV file is the set of cells in the same horizontal line. An example of row metadata about a file is the percentage of non-empty cells in a row. Another example row metadata is the row structural type, e.g., a data row, or a header row of a table, or a footnote row about a table. For individual tables, one example of row metadata is the table row header: tables in verbose CSV files may include row headers for all data cells in the whole row. For example, each cell in the first "Grades" column in the top table of Figure 1.2 is the row header for the corresponding table row. Other table row metadata include the number of header cells in a row, the data type of cells, and value distribution of cells in the data part, etc.

### 1.1.5 Table metadata

Table metadata depict the properties of a table in a data file. Such a table can either be with an arbitrary shape or a relational table following the RFC 4180 standard. A verbose CSV file may include multiple tables, for each we can derive different metadata. Examples of table metadata are the table-level Null-value representation [Qahtan et al., 2018], date/number value format, header hierarchy for tables with multi-layer headers, aggregations that correlate a numeric cell with a set of other numeric cells through an arithmetic relationship [Jiang et al., 2022]. For relational tables in standard CSV files, we may generate various profiling metadata, such as unique column combinations, denial constraints, and functional dependencies [Abedjan et al., 2015], and database constraints, such as primary keys [Jiang and Naumann, 2020].

### 1.1.6 File metadata

File metadata describes properties of a data file from the file level, e.g., the number of tables (table count) and their locations (table locations) in a single file. What is more, we may also define metadata across different files, e.g., the template used to create content in different files (file template [Vitagliano et al., 2021]). For relational tables in multiple standard CSV files, we may derive inclusion dependencies and foreign keys.

With the knowledge of various metadata, data scientists could understand their data more deeply, and conduct data preparation or data exploration more efficiently. For example, they may realize format inconsistency by checking the date format of all cells, or determine how to transform header regions of a table with the knowledge of header hierarchy, realize the structure of a file with the cell type, or understand the relationships amongst relational table columns with the foreign key metadata.

In the above, we present a taxonomy of six metadata types for verbose CSV files, and only several examples in each group. Note that the taxonomy is certainly not complete. It could be extended both horizontally to include more categories and vertically to incorporate more metadata. Given the uniqueness and complexity of file structure, detecting metadata in verbose CSV files is a valuable yet challenging task.

## 1.2 Challenges for metadata detection

While metadata are very useful information about data files, algorithms that automatically detect them have not been completely studied, leaving room for further research efforts. In this section, we present three major challenges about metadata detection in verbose CSV files, which we address in our works in the following chapters:

**(1) How to effectively detect line/cell types, aggregations, and PKs/FKs?**

Detecting the types of lines and cells, aggregations, and primary keys/foreign keys are all non-trivial tasks. Verbose CSV files have ad-hoc shapes and forms, and do not follow a fixed structure. Therefore, each line or cell may have a unique type. An effective approach to identify the type of each line/cell correctly must be able to handle all file structures. To detect aggregations, it must identify not only the cell whose value is derived from other cells, but also the said other cells and the used arithmetic function. As the set may have an arbitrary number of cells and multiple functions must be verified, the search space of aggregations can grow very large due to combinatorial explosion. Therefore, the key problem is to skip as many candidates as possible without losing true aggregations. Having a huge search space is also the issue for primary key and foreign key detection: many unique column combinations and inclusion dependencies – candidates of PKs and FKs – may exist even in a database of few tables with few columns.

**(2) How to detect metadata in data files without using stylistic features?**

Previous works about detecting line and cell types all focus on spreadsheets or web tables, where various stylistic features are available. However, being home to many valuable data, plain-text files, such as CSV files, cannot possess stylistic features, which have been proven in previous works to be very effective in classifying lines and cells. In order to achieve on-par or better detection performance than previous approaches, a challenge is to design an algorithm that relies only on characters in and positions of cells. The same issue applies to aggregation detection: only values and the arithmetic relationships amongst numeric cells are usable.

**(3) How to improve metadata detection with already detected metadata?**

Metadata often do not exist separately: The presence of one metadata may hint at or exclude that of another. For example, the detected type of a line in a verbose CSV file implies the type of the cells in the line; Having a relational table column predicted as the primary key, those inclusion dependencies that reference another column in this table cannot be valid foreign keys, because a foreign key must be definition reference a primary key. Previous works usually focus on the discovery of a single type of metadata, e.g., the type of lines, or cells. However, the quality performance of a discovery algorithm may be improved if the metadata it detects can be used to revamp the results of other types of metadata.

Besides the above major challenges, there are also others on detecting metadata, e.g., *how can we create distributed algorithms and enable scalable metadata detection? How to identify metadata from data files with errors? How to improve discovery performance with human feedback?* While these are all interesting questions amongst many others, our attention in this thesis is focused on the above three main challenges.

## 1.3 Structure and contributions

In previous sections, we have introduced the problem of metadata detection in data files and discussed the usefulness of metadata in exploring and preparing data, and building data-driven models in typical data science workflows. The following three chapters introduce our main contributions on detecting three types of metadata in verbose CSV files, which are abstracted in the following.

**Strudel: Line and cell classification in verbose CSV files**

In Chapter 2, we present our project Strudel, which detects the structure in verbose CSV files, and is based on our publication [Jiang et al., 2021]. The structure of a file is represented by the type of each line and cell therein. Strudel is based on a supervised-learning classifier that leverages a collection of useful features. Previous line or cell classification approaches all rely on stylistic features: information such as the font of values, the background color of cells, and thickness of cell frames. Therefore these approaches work only for spreadsheets or web tables from which such information can be extracted. Our algorithm drops the assumptions that input files are stylistic, and is capable of handling a more general data file format: CSV files. Strudel resorts to three types of non-stylistic features: (i) content features that describe the metadata of values in individual cells; (ii) contextual features that inspect individual lines or cells in the context of its neighbouring ones; (iii) computational features that indicate if the value of a numeric cell may be derived arithmetically from other cells. Additionally, the predicted types of lines are also encoded as a feature to serve the cell classification task. Our experiment results show that Strudel outperforms state-of-the-art line or cell classification tasks when no stylistic features are available. Strudel has been developed by Jiang, while Vitagliano and Naumann contributed valuable discussions.

**AggreCol: Aggregation detection in verbose CSV files**

Strudel utilizes computational features to help determine lines or cells of the aggregation type, which has been proven effective. However, the features consider only two types of aggregation functions: sum and average. In reality, data scientists summarize their data with a wide variety of functions. Additionally, Strudel assumes that cells can aggregate only adjacent cells in the same row or column. In fact, cells whose values are used to build an aggregation may appear at any arbitrary position in a file. To relax the aforementioned two constraints, we propose the AggreCol algorithm to detect aggregations of five different functions in verbose CSV files, which is based on our publication [Jiang et al., 2022] and presented in Chapter 3. AggreCol is a three-staged rule-based approach that is able to recognize aggregations of sum, difference, average, division, and relative changes. Our algorithm can detect not only the cell whose value is the aggregated value (aggregate), but also the cells whose values are used to calculate the aggregate value (ranges). AggreCol drops the assumption that the range of an aggregation must be adjacent to the corresponding aggregate. In other words, any numeric cells in the same row or column as the aggregate may be part of the range. Our experiment results demonstrate that the per-function precision scores of detected aggregations surpass those of an eager baseline approach, which iterates over range candidates of all possible cell combinations. In addition, having replaced the original aggregation detection technique used by Strudel with AggreCol, we have confirmed the improvement on the

cell classification task brought by using AggreCol. Jiang developed the AggreCol system, while Vitagliano, Hameed, and Naumann contributed valuable discussions.

**HoPF: Primary key and foreign key detection in relational tables**

Having recognized the type of each cell in a verbose CSV file, data scientists may transform the content of the tables in the file into relational tables that can be used for SQL-based queries [Barowy et al., 2015; Cafarella et al., 2008]. However, these "raw" relational tables lack primary keys and foreign keys, which are critical RDBMS constraints in various aspects, such as guaranteeing entity integrity and referential integrity, allowing for index creation, and enabling table joins. Previous works detect either primary keys or foreign keys. In contrast to that, Chapter 4 introduces our proposed algorithm HoPF to detect both primary keys and foreign keys in a collection of relational tables in a holistic manner, which is based on our publication [Jiang and Naumann, 2020]. HoPF takes unique column combinations (UCC) and inclusion dependencies (IND), for which primary keys and foreign keys are the respective special cases, as input. UCCs and INDs are well-known metadata that can be readily detected by previous data profiling algorithms [Abedjan et al., 2015], and therefore can be reasonably used as prepared input data. HoPF holistically selects a subset of UCCs and INDs as the predicted primary keys and foreign keys, according to their overall score w.r.t. a set of features. We prove the effectiveness of HoPF via a set of qualitative experiments on five relational database instances. Additionally, the comparison against two state-of-the-art approaches shows the superiority of our algorithm. HoPF was developed by Jiang, while Naumann contributed valuable discussions.

Finally, we summarize our work about detecting metadata in verbose CSV files in Chapter 5, based on which we suggest future directions for metadata detection with regards to both research and practical aspects. Putting all the works in a broader context of data preparation, we reason about the necessity of detecting various metadata for the success of automatic data preparation and self-service data preparation.

# Chapter 2

# Structure Detection in Verbose CSV Files

Data may be stored in a wide variety of file formats. While some data are persisted in well-defined formats, such as relational tables or as key-value pairs, that can be readily parsed by dedicated tools, a large quantity of other data are stored in documents with possibly unique structures, e.g., CSV files. CSV files are *character-separated values* documents that serve as data sources for various research and industrial problems, such as data profiling [Abedjan et al., 2015; Consonni et al., 2019; Schirmer et al., 2019], data curation [Nargesian et al., 2018; Thirumuruganathan et al., 2020; Zhang and Chakrabarti, 2013], data cleaning [Mahdavi et al., 2019; Rekatsinas et al., 2017], and information extraction [Chu et al., 2015; Gatterbauer et al., 2007]. Despite the broad usage of CSV files, the RFC 4180 CSV standard [Shafranovich, 2005] to depict a canonical way of persisting tabular data with such file format has not been proposed until 2005, 33 years after this format was first used to carry data by IBM [IBM Corporation, 1972]. What is more, this standard is not enforced on CSV files, leading data scientists and practitioners to frequently produce files with very unique structures, e.g., *verbose CSV files*. We give the formal definition of verbose CSV files in Section 2.1.1. In short, a verbose CSV file may include content serving different purposes, such as table header, aggregate, or footnotes, in different positions.

A significant amount of CSV files are verbose. Researchers have found that only 22% of 200 randomly selected spreadsheets – a common source of verbose CSV files– can be directly converted to relational tables [Chen et al., 2013], meaning that 78% of them are verbose to some extent. A similar observation by Dong et al. states that less than 3% of spreadsheet tables are "machine-friendly" [Dong et al., 2019]. Note that spreadsheets are not the only source for verbose CSV files. Based on our own observation on a dataset crawled from the Mendeley data portal[1], in a random selection of 13,917 plain-text files that contain tables, 4,459 are verbose, accounting for around 32% of all inspected files. These verbose plain-text files can be easily transformed into CSV files by applying file-specific delimiters.

On the one hand, content in verbose CSV files is arranged to facilitate readability and understandability for humans. On the other hand, machines are designed to process

[1] https://data.mendeley.com

files with a certain data structure or schema, and can hardly handle these files that each may have a unique structure. To understand and extract information from verbose CSV files, an important preliminary task is *structure detection*, in particular *classifying lines or cells* by their purpose. As manual efforts are infeasible and error-prone for large files or large sets of files, automatic approaches are desirable.

File structure is a piece of important metadata in verbose CSV files for various applications, e.g., locating tables [Coletta et al., 2012; Dong et al., 2019; Vitagliano et al., 2021], removing the non-tabular parts of files, or cutting out redundant information serving as aggregates of other numeric data in tables. The structure of tables may also be used to compare files and discover file templates [Vitagliano et al., 2021] that guide the production of the same kinds of files.

This chapter introduces our approach STRUDEL for Structure Detection in Verbose CSV Files. Specifically, STRUDEL addresses both the *line* and the *cell* classification problems on verbose CSV files, based on our publication [Jiang et al., 2021]. In a nutshell, STRUDEL is a supervised learning approach built on a random forest classifier. By using a set of novel features, STRUDEL is capable of classifying lines and cells as one of six types: metadata, header, group header, data, aggregate, and notes.

The contributions of this work are summarized as follows:

(i) A supervised learning approach with three kinds of novel features to address the structure detection problem, namely line and cell classification problems, for verbose CSV files.
(ii) Two datasets with more than 101K annotated lines in 426 files, reforged labels of another three datasets with 936 files from related work based on our perspective on cell classes, and a dataset with 62 files transformed from plain-text files.
(iii) A series of qualitative experimental evaluations on the STRUDEL approach, and a comparison of our approach with baseline and state-of-the-art approaches.

The rest of this chapter is organized as follows: Section 2.1 first formally defines verbose CSV files, and presents the taxonomy of line and cell classes. Then, we highlight the difficulty of classifying lines and cells with a set of verbose CSV files with diverse structures. This section ends with the formal problem statement and a brief introduction of our proposed approach. Sections 2.2 and 2.3 describe the core idea of STRUDEL, followed by Section 2.4, where we present the results and analysis of our experiments. Section 2.5 summarizes the related work about line and cell classification tasks and other relevant areas. We conclude this work in Section 2.6.

## 2.1 Structure detection

In this section, we first introduce the formal definition of verbose CSV files and propose a taxonomy of six different classes for lines and cells. Then, we give the formal statement of the structure detection problem with regard to line and cell classification. After that, we present a gallery of eight randomly selected verbose CSV files to highlight the difficulty of detecting structures therein. Finally, we present the architecture of our STRUDEL approach.

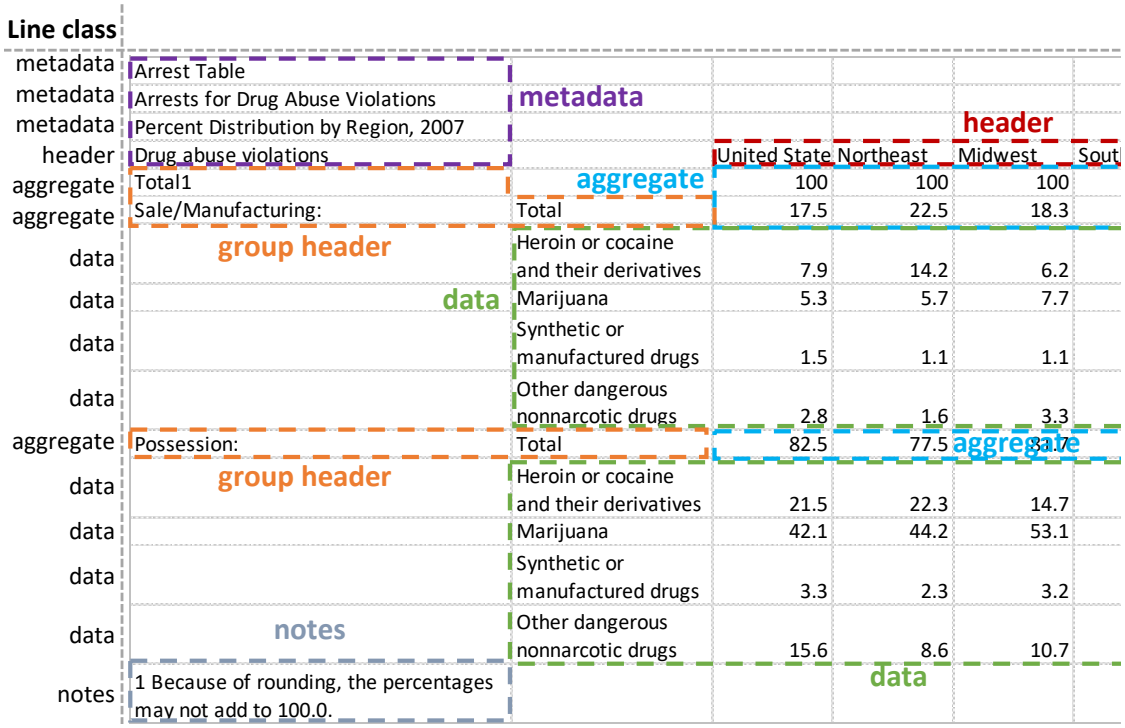



Figure 2.1: Excerpt of a real-world verbose CSV file with different cell-level and line-level content classes. Here, the class of each line is determined by the majority of its cell classes.

### 2.1.1 Verbose CSV file

A standard CSV file, according to RFC 4180, contains an optional header line at the beginning of the file, followed by a number of data lines. In contrast, a *verbose CSV file* may include elements of heterogeneous classes (introduced in Section 2.1.2), possibly with empty visual separators. Here, an *element* is either a non-empty cell or a line that includes at least one non-empty cell.

**Definition 1** (Verbose CSV file)**.** A *verbose CSV file* is a character-separated values file with values including one or more of metadata, header, group, data, aggregate, and notes at arbitrary positions. Each line of the file may be composed of cells of one or more classes. Empty cells may represent either missing values or serve layout purposes.

While a standard CSV file stores a single table that can be consumed by RDBMS or other tabular data processing systems, a verbose CSV file resembles a spreadsheet file that may include multiple tables, and make use of empty cells and cell types to improve human readability. An example of a real-world verbose CSV file from the "Crime In the US" (CIUS) dataset is shown in Figure 2.1, where cells with different purposes are highlighted. Information of different kinds may be organized as connected clusters of cells throughout the file: each such cluster may include information of a single type, such as data, metadata, or aggregate, and a table may be divided by blank visual separators into several table fractions.

### 2.1.2 Class taxonomy

We present a taxonomy that includes six classes, similar to that of [Adelfio and Samet, 2013], which addressed the line classification problem on web tables and spreadsheets. While in principle content of any class may appear at any location in a verbose CSV file, we enforce a few practical constraints on their possible position, reflecting the usual reading convention: from *left* to *right* and from *top* to *bottom*, assuming that tables are stacked only vertically. The following list describes each class in detail.

- **metadata.** Metadata are the descriptive text *above* a table. Such text may include the title of a table or additional information on the content of the table. A metadata area may span across one or more lines and columns.
- **header.** Headers are the column labels in the top area of a table (or table fraction). Headers may span multiple cells. In our definition, the header elements are located *above* the data area, and *below* any metadata block of the table.
- **group.** In verbose CSV files, tables are often split into several parts, each including data of a particular group. A group (a.k.a. group header) element serves as the label of such a part. Based on our observations on our datasets, group elements can appear both *above* and *below* header areas. Therefore, we allow both cases in our definition. Group cells may also serve as the leftmost string cells in an aggregate line, which will be described below.
- **data.** Data elements are the content of a table that cannot be derived from any other elements. Because they constitute the main body of a table, data elements of a section of a table are always *below* the header and group elements that indicate this section.
- **aggregate**[2] An aggregate cell can derive its values by applying a specific arithmetic function on the values of some other numeric cells in the same table. In verbose CSV files, aggregate cells are often organized as the top- or bottom-most lines, or the left- or right-most columns of the data area of a table.
- **notes.** Notes are descriptive text that follows a table. They may give explanations of particular parts of a table, explain the meaning of marks used in the table, or indicate the data source origin.

Any line or cell in a verbose CSV file can be associated with exactly one of the classes introduced above. The file in Figure 2.1 cannot be directly ingested by common RDBMS tools as it contains much additional information, aside from a table with its header and data rows, and its content with an ad-hoc layout does not follow a relational table schema.

### 2.1.3 Problem statement

With the above two concepts, we aim at *structure detection in verbose CSV files* by means of classifying file content in two granularities: *lines* and *cells*. Specifically, *given a verbose CSV file and the group of possible element classes, how can we determine the classes of all elements?* We consider elements of two natures addressing two sub-problems under the same problem definition: *lines* [Adelfio and Samet, 2013; Christodoulakis et al., 2020] that reflect the common top-down content arrangement within a file, and *cells* [Gol

[2]This class is named "derived" in the original paper. Here we rename it to "aggregate" so that it is consistent with the same concept used in Chapter 3.

et al., 2019; Koci et al., 2016], as the most fine-grained element of a structured file. The structure of a file is often reflected in the sequence of line classes, as data organized in verbose CSV files usually conform to the common top-to-bottom data presentation logic. For example, consider the line class labels of the example file in Figure 2.1. These classes show a natural logic of organizing information: metadata, such as captions, come first, followed by the main body of a table incorporating table headers, aggregate lines, and data lines, and finally, a few footnote lines conclude the file.

### 2.1.4 Challenges

The wide diversity of content layouts is an obvious challenge for detecting structures in verbose CSV files. While non-verbose CSV files all organize data in a relational table, verbose ones present content with unique structures that may be largely different from one another. Figure 2.2 demonstrates the thumbnails of eight verbose CSV files randomly selected from our datasets, where background colors indicate the content classes. This set of files adequately demonstrates the file structure diversity. For example, the size of metadata regions (purple) differ from one another; table headers (red) may span different numbers of rows; aggregate regions (blue) can be either below, or above, or in-between data regions (green). For STRUDEL, we propose a set of general features that aim at capturing different content layouts.

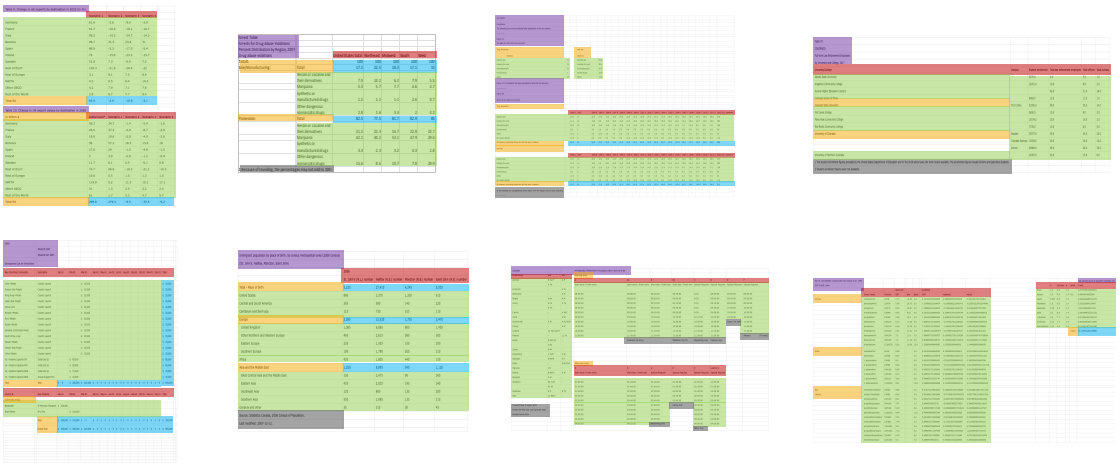


Figure 2.2: Thumbnails of eight verbose CSV files that are randomly selected from our datasets. Background colors of the regions indicate the classes of content. Specifically, purple/red/green/orange/blue/grey represent metadata/header/data/group/aggregate/notes regions, respectively.

Another challenge to identifying file structures comes from the lack of rich-text information. Verbose CSV files are favored to exchange data due to their generality over those with proprietary file types, such as spreadsheets, that are used by specific programs. However, as plain-text files, verbose CSV files are not able to preserve stylistic features, such as the font style of text, background colors of cells, or border style of a table, etc, which have been proven very useful features to classify lines and cells in spreadsheets and web tables by previous works [Adelfio and Samet, 2013; Dong et al.,

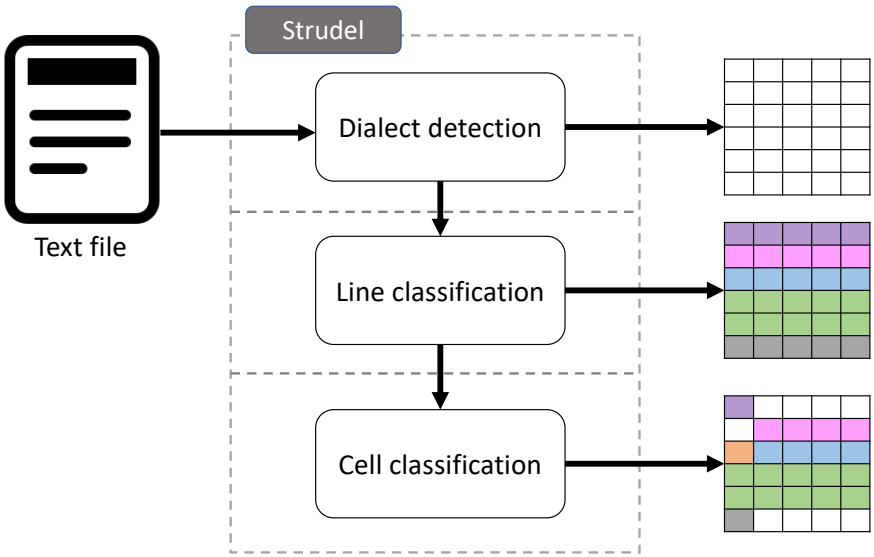



Figure 2.3: Architecture of the Strudel algorithm.

2019; Gol et al., 2019]. As for verbose CSV files, only the characters, and positions of lines or cells can be used for the tasks.

Obtaining datasets with line and cell annotations to evaluate an approach for the structure detection problem is not trivial, as ascertaining these classes in a verbose CSV file is a difficult task even for experienced practitioners. In our study, human annotators (i) took on average of two minutes to label the lines in a single file, because they needed to spend a lot of time understanding the unique structure of each file; (ii) at times disagreed with each other on the annotations of individual lines. We implemented a line and cell type annotation tool with a sophisticated graphical interface, which is introduced in Section 2.4, to support the manual labeling task.

### 2.1.5 Architecture of Strudel

To address the line and cell classification problems, we propose the Strudel approach, which is grounded on a multi-class random forest classifier. Figure 2.3 shows the architecture of the approach. Cells of different types are distinguished by colors. Strudel first detects the dialect of a text file, and creates a verbose CSV file from it, based on the dialect. Dialect detection is done using the approach proposed by van den Burg et al. [van den Burg et al., 2019]. In the next steps, Strudel first classifies lines and then cells in the files with the suggested feature sets. Section 2.2 and 2.3 describe Strudel's features for line and cell classification tasks, respectively. The features can be categorized into three groups: (i) *content features* that parse the values of lines or cells, such as the number of words and cell length; (ii) *contextual features* that compare the inspected line or cell with its neighboring ones regarding the properties such as the similarity of data types between lines and cells; (iii) *computational features* that seek to connect lines/cells with each other by inspecting arithmetic correlations between them.

## 2.2 Line Classification

In this section, we describe STRUDEL$^L$ – the STRUDEL approach to address the line classification problem. STRUDEL$^L$ is based on a multi-class random forest classifier. Its input includes a set of features extracted from the two-dimensional tabular data.

Various kinds of features have been proposed by previous works to address the line classification problem, including content features, contextual features, spreadsheet formula features, and stylistic features [Adelfio and Samet, 2013; Koci et al., 2016]. In our case, formula and stylistic features are not applicable, as CSV files do not preserve these rich-text information. Instead, we design a set of content features, contextual features, as well as novel computational features. We build the features of STRUDEL$^L$ on top of the applicable features from a previous work [Adelfio and Samet, 2013]. Table 2.1 lists our complete set of features, divided into three groups. The context features can include information from both the line above and the line below. Thus, the features marked by a star are applied twice – once for the line above and once for the line below. To distinguish aggregate cells from data cells, we propose novel *computational* features that check whether the value of a numerical cell can be calculated by applying a specific aggregation function on the numbers in the vicinity, i.e., the cells in the same row or column.

Table 2.1: Line classification features: '$*$' marks the contextual features applied to both lines above and below the inspected line; '†' marks the features adapted from [Adelfio and Samet, 2013].

| Category | Feature | Value |
|---|---|---|
| Content | EmptyCellRatio† | [0.0, 1.0] |
| | DiscountedCumulativeGain | [0.0, 1.0] |
| | AggregationWord† | 0/1 |
| | WordAmount | [0.0, 1.0] |
| | NumericalCellRatio† | [0.0, 1.0] |
| | StringCellRatio† | [0.0, 1.0] |
| | LinePosition† | [0.0, 1.0] |
| Contextual | DataTypeMatching* | [0.0, 1.0] |
| | EmptyNeighboringLines* | [0.0, 1.0] |
| | CellLengthDifference* | [0.0, 1.0] |
| Computational | AggregateCoverage | [0.0, 1.0] |

Here, we describe and explain only the novel features used in our approach and refer to related work for the others.

- DiscountedCumulativeGain (DCG) calculates the discounted cumulative gain on a vector created from the cells of a line. The vector has the same length as the number of cells in a line. An element is set to '1' if the corresponding cell in the line is non-empty, or '0' otherwise. This feature is used to model the pattern of empty cells. DCG gives more weight to left-more positions than to right-more positions, modeling typical user behavior that lays out data from left to right.
- AggregationWord checks whether a line contains any word that belongs to a pre-made

dictionary of terms associated with aggregation in tables (case-insensitive): `total`, `all`, `sum`, `average`, `avg`, and `mean`. The existence of any keyword gives '1' to this feature, otherwise '0'. Using a dictionary of such kinds of keywords proves to be effective [Koci et al., 2016].

- WordAmount calculates the number of words in all cells of a line. A word is a sequence of alphanumeric characters. The feature values are normalized per file by using a min-max normalization strategy.
- DataTypeMatching calculates the percentage of cells in a line whose data types match with those of the adjacent line (above or below). Note that some files insert an empty line between every pair of non-empty lines to visually highlight the content. However, comparing the data type of a line with an empty adjacent line does not carry much information. Therefore, an adjacent line refers to the closest non-empty line. Data and aggregate lines tend to have numerical cells while header lines usually contain alphanumeric values. Other functional lines, such as metadata, notes, and group lines tend to have many empty cells, because only the left-most cell in the line is usually non-empty.
- EmptyNeighboringLines calculates the percentage of empty lines in the five lines above or below the inspected line. Empty lines are often used as visual separators in verbose CSV files. Using such a separator between data lines *within* a table is uncommon, but placing them *between* two classes of lines, such as header-data and aggregate-notes is more common.
- CellLengthDifference calculates the cell value length difference between two adjacent lines by calculating the Bhattacharyya histogram difference [Bhattacharyya, 1946] on the sequences of cell value lengths of the two lines. When computing this feature, we compare only a line with its closest non-empty neighboring line, similar to that applied for the DataTypeMatching feature. While data lines tend to have similar cell-wise value lengths, as they usually describe the same property and thus draw values from the same domain and range, non-data lines may have values in natural language that are arbitrary long.
- AggregateCoverage counts the number of numeric cells recognized as aggregate cells by the aggregate cell detection algorithm introduced in Section 2.3. The feature is normalized by the number of numeric cells in this line.

Note that these are all local line classification features, i.e., they describe the characteristics of individual lines. We have tested a few global features that reflect properties of the entire file, e.g., percentage of empty lines in a file, width and length of a file, and the number of empty line blocks in a file. However, in our experiments, these features do not show a positive impact on the classification problem.

All features are normalized and passed to a random forest classifier that predicts one class for each line. When used as the LineClassProbability feature in the cell classifier of Strudel (Section 2.3.4), the output is a set of vectors, each of which stands for a probability vector of all classes for a line.

## 2.3 Cell Classification

Our cell classification approach Strudel$^C$ is, like Strudel$^L$, based on a multi-class random forest classifier. For the input of this classification task, we have again con-

structed a set of features that include both the effective ones from previous works and novel ones. The predictions for line classes are used as a set of features in Strudel$^C$. Therefore, the Strudel$^L$ approach is executed beforehand to obtain the line prediction probabilities that are then transformed into the features of Strudel$^C$. We leave the detailed description to Section 2.3.4.

### 2.3.1 Feature extraction

Previous works have proven the effectiveness of content features, stylistic features, spreadsheet formula features, and contextual features [Gol et al., 2019; Koci et al., 2016]. We ignore spreadsheet-specific formula features and stylistic features, as they cannot be constructed from verbose CSV files. Table 2.2 lists all features involved in our approach, which also fall into three groups: content, contextual, and computational features.

Table 2.2: Cell classification features; '$*$' marks contextual features applied to each of the eight surrounding cells of the inspected cell; '†' marks features from related work.

| Category | Feature | Value |
|---|---|---|
| Content | ValueLength† | [0.0, 1.0] |
| | DataType† | [0..4] |
| | HasAggregateKeywords† | 0/1 |
| | RowHasAggregateKeywords† | 0/1 |
| | ColumnHasAggregateKeywords† | 0/1 |
| | RowPosition† | [0.0, 1.0] |
| | ColumnPosition† | [0.0, 1.0] |
| | LineClassProbability | $(p_1, \ldots, p_6)$ |
| Contextual | IsEmptyRowBefore | 0/1 |
| | IsEmptyRowAfter | 0/1 |
| | IsEmptyColumnLeft | 0/1 |
| | IsEmptyColumnRight | 0/1 |
| | RowEmptyCellRatio† | [0.0, 1.0] |
| | ColumnEmptyCellRatio† | [0.0, 1.0] |
| | BlockSize | [0.0, 1.0] |
| | NeighborValueLength* | [0.0, 1.0] |
| | NeighborDataType* | [0..5] |
| Computational | IsAggregate | 0/1 |

The features marked with '†' in Table 2.2 are based on those used in [Gol et al., 2019; Koci et al., 2016]. Some of the original features are integrated into our feature set without modification, such as ValueLength and DataType, while others are adapted to a certain extent. For instance, a Boolean feature used to mark the existence of aggregate cell keywords is extended to a row or a column (RowHasAggregateKeywords and ColumnHasAggregateKeywords), i.e., whether the row or the column that contains the inspected cell contains any aggregate cell keywords. ValueLength counts the number of characters

in the value of a cell. DataType in this work has four possible values, corresponding to four data types: int, float, string, and date. Features without '†' are new. In the following, we explain the intuition and implementation of the five most sophisticated features, i.e., BlockSize, NeighborValueLength, NeighborDataType, LineClassProbability, and IsAggregation.

### 2.3.2 Block size

A verbose CSV file may contain multiple tables in various positions, rather than a single relational table. Apart from tables, a verbose CSV file may contain non-data regions composed of aggregate, notes, or metadata cells. In our datasets, non-data regions are usually spanned across several consecutive lines, each has few non-empty cells. Such lines as a whole often serve as metadata or footnotes of the tables after or before them. These non-data regions are usually smaller than tables.

To model this phenomenon, we create for each non-empty cell a BlockSize feature, which is calculated as *the size of the connected component that contains this cell*. A connected component is composed of a group of connected, non-empty cells. Two cells are connected if they are either vertically or horizontally adjacent to each other, or there is at least one connective path between them. Algorithm 1 describes how the value of this feature is calculated for each cell in a given verbose CSV file. It takes all non-empty cells in a table as input and outputs key-value pairs where keys are these non-empty cells and values are their respective block sizes. To obtain the block size for all non-empty cells, the algorithm employs a depth-first search strategy to iterate over all of them in a given file. It starts from a single cell block (lines 4-7), and continuously adds adjacent cells to expand the block until no more non-empty adjacent cells can be found (lines 8-13). The block size is normalized to $[0, 1]$ by the size of the file (line 14). The algorithm terminates once all cells have been processed.

Regarding the complexity of this algorithm, assume there are $n$ non-empty cells in a verbose CSV file. On the one hand, each cell will be visited once and only once, resulting in a $O(n)$ complexity. On the other hand, each of the 4-connected neighboring cells of a cell is checked once the cell is visited, leading to a $O(4n)$ complexity. Therefore, the overall algorithm complexity is $O(n) + O(4n) = O(n)$.

### 2.3.3 Neighbor profile

Cells of some classes may be likely to have particular kinds of neighboring cells. For example, to highlight group cells, users often separate them from other cells with empty cells, or they lay aggregate cells at the margin of a table, as a way to summarize data, or they place headers above data cells, which follows the top-to-bottom reading habit. These observations bring our focus to the adjacency context of a cell: for each cell, we gather the *data types* and *value lengths* of all 8-connected neighboring cells and present each as a single feature in the feature vector. The neighbor profile of a cell includes all these NeighborValueLength and NeighborDataType features. For the cells on the margins of a file, some adjacent cells do not exist. We set a default value for these non-existent adjacent cells, i.e., $-1$ for value length and data type.

**Algorithm 1:** Block size calculation

**Input:** The set of non-empty cells in a table $C$
**Output:** The set of key-value pairs $BS$ from cells to block size

$BS \leftarrow \{\}$;
$V \leftarrow \{\}$ // *visited cells*;
**while** $C - V \neq \emptyset$ **do**
  $c \leftarrow$ random cell in $C - V$;
  $bs \leftarrow 1$;
  $V \leftarrow V \cup c$;
  $B \leftarrow \{c\}$;
  **while** *there exist cells in* $C - V$ *adjacent to* $B$ **do**
    $c_{adj} \leftarrow$ an adjacent cell in $C$;
    $bs \leftarrow bs + 1$;
    $V \leftarrow V \cup c_{adj}$;
    $B \leftarrow B \cup \{c_{adj}\}$;
  **end**
  $bs \leftarrow$ normalize($bs$);
  **foreach** $c \in B$ **do**
    $BS \leftarrow BS \cup \{c : bs\}$;
  **end**
**end**
**return** $BS$

Table 2.3: Percentage of lines under different diversity degrees.

| Dataset | Diversity degree | | | | |
|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 |
| SAUS | 86.3% | 13.7% | 0% | 0% | 0% |
| CIUS | 88.7% | 11.2% | 0.1% | 0% | 0% |
| DeEx | 95.3% | 4.6% | 0.1% | 0% | 0% |

### 2.3.4 Line class probability

Despite the possible flexible layout, verbose CSV files are usually organized in some structurally meaningful way. Lines tend to organize mostly homogeneous types of cells to ease human understanding. For example, a data line contains mostly data cells, while a header line contains mostly header cells. Table 2.3 displays statistics about the *cell class diversity degree* of all lines in our datasets. The cell class diversity degree of a line is its number of distinct non-empty cell classes. We observe that most lines have a diversity degree of one: all cells in these lines are associated with the same class. In other words, for the cells in these lines, their classes can be trivially determined by the class of the line. Therefore, when determining the class of a cell, the class of the line it is located in is likely a useful feature. In fact, we use this feature alone as one of our baselines (denoted as $\textsc{Line}^C$ in Section 2.4).

To obtain the line class information, we first run $\textsc{Strudel}^L$ to obtain the predicted

class for each line. The result of this execution is, however, a probability vector of all classes, instead of a single predicted class. We interpret this probability vector as the classifier's confidence for these classes. Each element of the 6-dimensional vector accounts for a feature in the cell classification feature set.

### 2.3.5 Aggregate cell detection

If a cell is indeed a aggregate cell, it should be possible to generate its value by aggregating values of some other cells via a particular function. This fact has not been considered by previous work, possibly due to computational cost. We propose an aggregate cell detection algorithm that seeks to identify cells of such type by arithmetically correlating their values with other numeric cells. We made four observations while investigating the datasets: (i) an aggregate cell usually aggregates the values of cells from either its own row or its own column; (ii) an aggregate cell tends to aggregate values close to it; (iii) sum and average are the two most common aggregation functions used in verbose CSV files; (iv) indicative keywords such as "total" are likely to appear in a cell either in the same row or in the same column as an aggregate cell. We integrate these insights into Algorithm 2 that determines for each numeric cell in a file, if it is a sum or average of some other adjacent cells. For conciseness, the pseudo-code shows only the approach for sum cell detection.

The algorithm takes as input a file as a two-dimensional array, the aggregate keyword dictionary to look for anchoring cells, an aggregation delta $\delta$ to give some slack to aggregation results, and a coverage threshold $c$ that controls the generality of aggregation results. Executing the algorithm produces all detected aggregate cells.

For each numeric cell, any combination of other numeric cells is a candidate to calculate its value. Therefore, the combinatorial explosion leads it prohibitively expensive to traverse all aggregate cell candidates. Our algorithm applies a heuristic approach to skip most spurious candidates, according to the observation (iv) above. We mark those cells with any of our aggregate keywords (introduced in Section 2.2) as *anchoring* cells (line 2). Only numeric cells either in the same row or in the same column as an anchoring cell are treated as aggregate cell candidates (lines 6-8).

For the candidates in the same row as the anchoring cell, the algorithm looks both upwards and downwards for possible aggregating relationships (lines 9-19), whereas for the candidates in the same column as the anchoring cell, the algorithm looks leftwards and rightwards (lines 20-30). When looking upwards, the algorithm adds numeric values of a row each time to the sum vector correspondingly and inspects whether the current sum vector is element-wise close enough (according to $\delta$) to the candidates. If the coverage of the close enough elements in the sum vector surpasses $c$, the candidate is treated as an aggregate cell (lines 14-17). Due to our second observation, a row closer to the row where the candidates are is inspected earlier than a row farther away.

The proposed aggregate cell detection algorithm plays an important role in distinguishing aggregate cells from other types of cells, shown by the feature importance experiments in Section 2.4.3. Yet, it assumes the aggregate cells can be either sum or average of their adjacent cells, possibly limiting its generalizability. Sum and average may not always be the most common functions in general: an investigation into two datasets not considered during the algorithm design shows that both *division* and *relative change*

**Algorithm 2:** Aggregate cell detection

```
   Input: File F, keywords K, aggregation delta δ, coverage c
   Output: All detected aggregate cell C_D
 1 C_D ← {};
 2 A ←getAnchoringCells(F, K);
 3 if A is empty then
 4 |   return C_D;
 5 foreach a in A do
 6 |   i_a, j_a ← row index of a, column index of a;
 7 |   C_R, cc_ind ← the list of numeric cells in row i_a and their column indices;
 8 |   C_C, rc_ind ← the list of numeric cells in column j_a and their row indices;
   |   /* line 9-19 for upwards detection                                  */
 9 |   sum ← (0...0);
10 |   for i = 1 to ∞ do
11 |   |   if i_a − i < 0 then
12 |   |   |   break;
13 |   |   else
14 |   |   |   v_o ← numeric values at cc_ind in row (i_a − i);
15 |   |   |   sum ← sum + v_o;
16 |   |   |   if coverage of (C_R − sum < δ) > c then
17 |   |   |   |   C_D ← C_D ∪ C_R
18 |   |   end
19 |   end
   |   /* repeat lines 9-19 for downwards detection                        */
   |   /* line 20-30 for leftwards detection                               */
20 |   sum ← (0...0);
21 |   for i = 1 to ∞ do
22 |   |   if j_a − i < 0 then
23 |   |   |   break;
24 |   |   else
25 |   |   |   v_o ← numeric values at rc_ind in column (j_a − i);
26 |   |   |   sum ← sum + v_o;
27 |   |   |   if coverage of (C_C − sum < δ) > c then
28 |   |   |   |   C_D ← C_D ∪ C_C
29 |   |   end
30 |   end
   |   /* repeat lines 20-30 for rightwards detection                       */
31 end
32 return C_D;
```

were used in more than 5% of the files (Section 3.2.1). Additionally, our error analysis in Section 2.4.3 indicates that values of aggregate cells are not always calculated by adjacent cells. In light of the new observations, we propose a more advanced algorithm AggreCol introduced in Chapter 3 to detect not only each aggregate cell in a file, but also the set of other cells that calculate its value and the applied function.

## 2.4 Evaluation

In this section, we first introduce the datasets, our annotation tool, and all algorithms used in our experiments. After that, we present our experimental evaluation on Strudel, including its comparison with the referenced approaches, the performance of Strudel on an unseen dataset and a plain-text file dataset, analysis of feature importance, and an error study for both line and cell classification tasks.

### 2.4.1 Annotation, datasets, and experimental setup

In this section, we first describe the annotation method and the tool we designed to label our datasets. After that, we list the datasets used in our evaluations, and the preprocessing steps applied thereon. In practice, verbose CSV files may have unique dialects. The dialect of a file specifies the *delimiter*, the *quoting character*, and the *escape character*, which enable parsing the lines and cells correctly. Therefore, as a general preprocessing, we first applied dialect detection on each file with the approach of van den Burg et al. [van den Burg et al., 2019], which takes a text file as input, and produces its detected dialect. This approach manages to discover the correct dialects of all files in all datasets except for one that includes plain-text files. We discuss the handling of this exceptional dataset while describing it below. We applied the detected dialect to each file and determined the scope of each cell or line thereby. The third part of this section describes the list of baseline and competing approaches, our Strudel approach, and their respective configurations for evaluation.

#### Annotation tool

While a great number of verbose CSV files are accessible thanks to various open data portals[3], line and cell labels for those files, which are indispensable for structure detection approach evaluation, are scarce. Due to widely different structures of verbose CSV files, annotating lines and cells relies on manual efforts that are inefficient without a proper annotation tool. Therefore, we implemented a competent tool that allows data scientists to efficiently label line and cell classes in verbose CSV files. Our tool provides separate modules for both line and cell annotation tasks. The operations of the two modules are similar. In the following, we describe only the one for cell annotation for conciseness.

Figure 2.4 shows a screenshot of our running cell annotation tool, which was implemented in Java 8 with the Swing GUI framework. After a file is loaded into the tool, its cell values and positions are rendered in the bottom window. Data scientists can simply click a cell or a block of cells, and assign them a class label. Annotated cells are presented with the corresponding background color. Annotations can be overwritten by following the above process. Empty cells are by default not labeled even if they are included in a selected area. Nevertheless, holding the "shift" key during an annotation process allows to label the empty cells in the selected region. By selecting the "Line Type Annotation" tab at the top of the interface, one can switch to annotate lines, which follows a similar procedure as the cell annotation module.

[3] `https://data.gov.uk/`, `https://www.govdata.de/`, `https://data.europa.eu/euodp/en/data/`

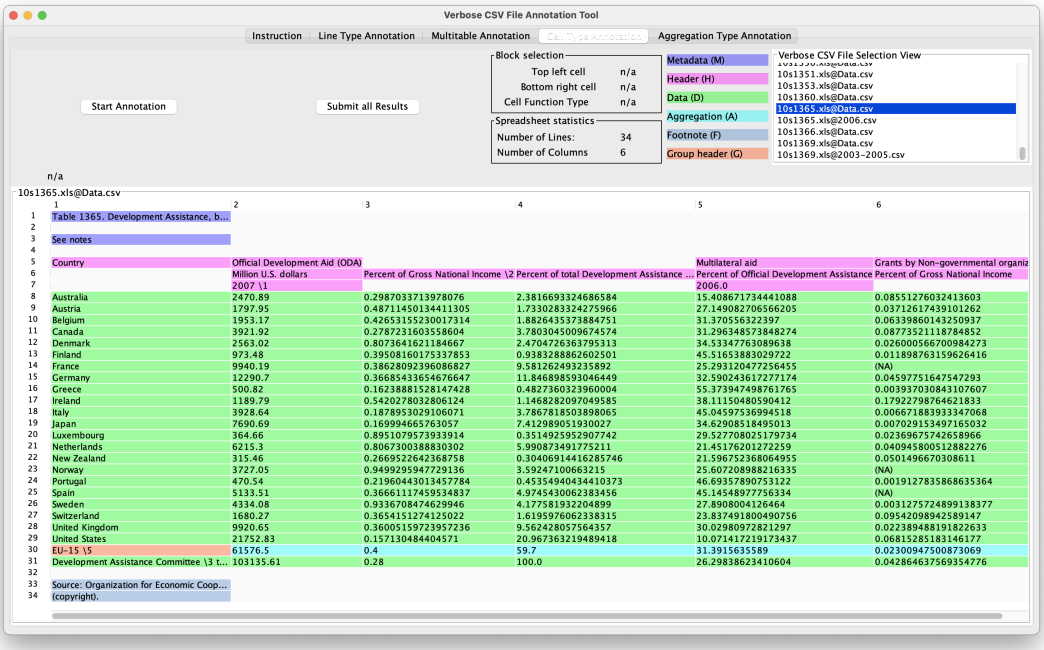



Figure 2.4: Screenshot of the cell annotation module of our annotation tool.

Provided with our annotation tool that has a concise graphical interface and easy-to-operate functions, we observed that human labelers took on average only two minutes to label all cells in one verbose CSV file, leading it realistic to annotate massive files for our evaluation. In addition, the visual annotation outcomes on the GUI make it easy to spot annotation mistakes and improve annotation quality.

### Datasets

Our datasets of verbose CSV files come from various sources. Table 2.4 shows some statistics of the datasets. Only non-empty lines and cells are counted. The content of files in our datasets is in the English language and follows a top-bottom / left-right organisation. Non-Western verbose CSV files may organize the content in a different fashion, and an exploration on classifying lines and cells in such files would be an interesting future work.

The `GovUK` dataset was created from all data files in Microsoft Excel format (both in .xls and .xlsx) we crawled from the open data portal of the government of the UK[4]. We randomly selected a subset of 300 files from the entire crawled file set and transformed them into corresponding CSV format with the Apache POI library[5]. When converting them to CSV files, we omitted the files that contain macros, were not otherwise processable by the library, or empty sheets. A set of 226 files were left after the filtering, for which we created a line-level ground truth. Each line of each file in the created dataset

[4] `https://data.gov.uk/`, last crawled on 13. August, 2019
[5] `http://poi.apache.org/index.html`

was annotated by three human experts. In case of disagreement, which affected only 1% of the annotated lines, we used majority-vote to determine the annotation. For the lines with complete disagreement (fewer than 250 lines in our dataset), we employed an independent fourth annotator to determine the final label. In the end, we obtained the ground truth for in total more than 95K annotated lines.

Table 2.4: The dataset overview.

| Dataset | # files | # lines | # cells |
|---|---|---|---|
| GovUK | 226 | 97,212 | 1,382,704 |
| SAUS | 223 | 11,598 | 157,767 |
| CIUS | 269 | 34,556 | 367,172 |
| DeEx | 444 | 77,852 | 784,229 |
| Mendeley | 62 | 195,598 | 1,359,810 |
| Troy | 200 | 4,348 | 23,077 |

Three other datasets, `SAUS`, `CIUS`, and `DeEx`, were created and annotated by Gol et al. for cell classification on spreadsheets and web tables [Gol et al., 2019]. The first two are administrative datasets, while the last one is a business dataset. More detailed descriptions of each dataset can be found in their original paper. The datasets were annotated by the original authors with a slightly different taxonomy. To reconcile their annotations to ours, we re-annotated their labels. In summary, they annotated all left-most headers of a table as attributes, while we consider them as data of their columns. In the example of Figure 2.1, they treat the cells that indicate the drug types in the second column as attributes, while we annotate them as data, as we model them as a data column of the table without a header. We also note that some obvious aggregate cells were marked as data: understandable errors due to their similarity, which we corrected. In many cases, aggregate cells form an entire line, with the exception of the leading cell, which is usually textual. This textual cell often includes keywords, such as 'Total', and is neither an aggregate cell nor a header cell. We treat it as a group cell in our system, because an aggregate line often serves as a section separator in a table. Table 2.5 presents the class distribution of these three datasets with the reforged annotations.

Our `Mendeley` data is a set of plain-text files collected from Mendeley's data-sharing platform[6] of experimental data. These data are stored in research projects that use them. We crawled all 2,214 projects whose data are stored on Mendeley's own server and that contain at least one plain-text file, i.e., whose MIME type is "text/*". This MIME type corresponds to a wide variety of actual file formats: not only files with table structures and verbose information, such as verbose CSV files, but also programming scripts, HTML pages, etc. We randomly selected 100 projects that include at least one verbose text file with at least one table, and obtain one such file from each project.

Given the intricate dialects of these plain-text files, the dialect detection approach of [van den Burg et al., 2019] cannot reliably discover the correct dialect for all files. A file is *parse-able* if the dialect for the table region (including header, data, group, and aggregate) is correct. For our experiments, we kept the 62 parse-able verbose CSV files.

[6]https://data.mendeley.com/, last crawled on 3. August 2020

Table 2.5: The number of lines or cells per class in the dataset SAUS, CIUS, and DeEx as a whole.

| class | # lines | # cells | avg. # cells per line |
|---|---|---|---|
| metadata | 2,213 | 2,479 | 1.12 |
| header | 2,232 | 19,047 | 8.53 |
| group | 1,767 | 6,143 | 3.48 |
| data | 114,354 | 1,202,058 | 10.51 |
| aggregate | 1,406 | 76,996 | 54.76 |
| notes | 2,036 | 2,445 | 1.20 |
| Overall | 124,006 | 1,309,168 | 10.56 |

Note that this dataset is used only to verify the performance of our approach on verbose plain-text files and is not part of the training set. We observe a high line-to-file and cell-to-file ratio, because the files of this dataset are mostly used to store data, e.g., experimental results, which are often very long, rather than presenting statistical tables.

The last dataset, Troy, contains 200 CSV files collected between 2009 and 2010 from various international statistic data portals, such as Statistics Finland and The World Bank [Nagy, 2010]. Embley et al. used this dataset in their work to convert different statistical tables to relational tuples [Embley et al., 2016]. The original data were stored in HTML and converted by the authors via Excel to verbose CSV files. We kept the dataset unseen during the design of Strudel to test the out-of-domain generalizability of our approach with this dataset.

In our data preparation process, we cropped each file by removing the marginal empty lines or columns, as some of our features are sensitive to the number of empty cells in the lines, and leading/trailing empty lines are trivial cases. Values of spanning cells in original spreadsheets are copied only to the top-left cell in the CSV file, instead of to all covered cells for two reasons: (i) the top-left is well-defined for all shapes of spanning cells and (ii) copying the values to all covered cells creates too many repeated characters, confusing the models that cause unnecessary over-fittings towards these values. To encourage future study on this topic, we publish all datasets and their annotations[7].

### Setup of experiments

The list below contains all algorithms used in the evaluation, along with their corresponding configurations. All algorithms were implemented in Python 3.7.7. We used the random forest classifier in the scikit-learn library [Pedregosa et al., 2011] for Strudel. The superscript in the name of an algorithm indicates the type of elements detected by this algorithm, i.e., '$L$' and '$C$' represent line and cell classification, respectively.

- Crf$^L$ is a conditional random field-based learning approach dedicated to line classification from Adelfio et al. [Adelfio and Samet, 2013] as the current state-of-the-art. We applied this approach with the logarithmic binning technique introduced by the authors, as this setting was reported to gain the best performance there.

[7] https://hpi.de/naumann/projects/data-preparation

- PYTHEAS$^L$ is a rule-based approach that discovers the locations of tables, and further classifies the lines in CSV files [Christodoulakis et al., 2020]. We use the parameter values introduced in the original paper for our experiments.
- STRUDEL$^L$ is our proposed approach for line classification. The underlying random forest classifier used the default settings in the scikit-learn library.
- LINE$^C$ is a baseline approach for cell classification. This approach simply extends the predicted class of a line from the result of a STRUDEL$^L$ run to each non-empty cell in this line.
- RNN$^C$ is based on the state-of-the-art approach by Gol et al., which classifies cell types with a recursive neural network using pre-trained cell embeddings [Gol et al., 2019]. For our experiment, we used the same settings as introduced in the original paper.
- STRUDEL$^C$ is our approach for cell classification. Again, we used the default settings of the random forest classifier in the scikit-learn library. In our experiment, we do not observe a substantial difference in the result with different values of the aggregation delta $d$ and coverage $c$. We set them to 0.1 and 0.5, respectively.

Apart from using content and spatial features, both original CRF$^L$ and RNN$^C$ applied stylistic or spreadsheet formula features. Because the input data in our use-case are styleless verbose CSV files, we leave out all stylistic features from the two approaches so as to conduct fair comparisons. Each algorithm is evaluated using group 10-fold cross-validation: when creating the folds, our process ensures that all elements from a single file appear in either the training or the test set. We repeat the cross-validation process ten times to reduce bias leaning to particular fold splits. The results of all repetitions are averaged to obtain the final score.

We have tested several classification algorithms for STRUDEL, including Naïve Bayes, KNN, SVM, and random forest. Random forest consistently outperformed the other candidate algorithms on our datasets for both classification tasks. Therefore, we chose it as the backbone algorithm of STRUDEL. The advantage of random forest over the other algorithms is that it reduces the risk of over-fitting by considering the results from multiple base classifiers, which is important for imbalanced datasets, such as types of lines and cells in verbose CSV files.

### 2.4.2 Comparative evaluation

This section presents the comparative evaluation results between STRUDEL and the referenced approaches on both line and cell classification tasks. We use the F1 measure to evaluate the classification correctness of each approach. When comparing the overall results amongst algorithms, we focus on the macro-average, which does not weigh the score of individual classes with the support (the number of instances) of these classes. Using macro-average avoids the bias from the number of per-class instances, which reveals the performance of an approach on minority classes better than the micro-average (weighted average).

#### Line classification

We compared STRUDEL$^L$ with CRF$^L$ and PYTHEAS$^L$. CRF$^L$ uses a set of features, including content features, contextual features, and stylistic features to train a conditional

random field based classifier on web tables and spreadsheets. PYTHEAS$^L$ uses a number of weighted rules to decide whether a line is data or non-data. The binary results are used to draw the table top/bottom boundaries, on top of which the approach utilizes some additional rules to determine the classes of lines.

Table 2.5 (top) reports the per-class and macro-average F1 scores, and accuracy for the three approaches. Note that PYTHEAS$^L$ can classify a line as one of only five classes that correspond respectively to ours, missing the aggregate class. Therefore, when calculating the measurements for this approach, we leave out the aggregate lines from our datasets. Overall, our approach leads on macro-average for all datasets. PYTHEAS$^L$ does not perform well in general on the minority classes in all but the SAUS datasets, as its proposed rules are not suitable for these datasets: they produce poor results already for the binary data/non-data classification, which disrupts the subsequent table discovery and line classification. Group lines are particularly difficult for PYTHEAS$^L$: the scope of group lines is constrained to lines between data lines and has only the leftmost cell non-empty. While the group lines in SAUS mostly follow this definition, those in the other datasets do not. Most header lines in both SAUS and CIUS are across few lines and with simple structures. Therefore, recognizing the headers correctly in these two datasets is simpler than those in GovUK and DeEx. Since the rule used to determine metadata lines is dependent only on the positions of headers, it is also easier to recognize metadata in these two datasets.

STRUDEL$^L$ outperforms CRF$^L$ for almost all classes across datasets. Both algorithms perform better on the CIUS dataset than on the other datasets, because many files in this dataset are essentially the reports from different years on the same themes with the same templates – the file structure diversity is small. Both approaches do not work well on aggregate lines in SAUS, because the dataset has many unanchored aggregate cells, i.e., headers of the rows or columns with aggregate cells often have no indicative keywords. GovUK and DeEx are difficult to both approaches.

In summary, STRUDEL$^L$ outperforms CRF$^L$ without using its stylistic features on our datasets, showing that our approach is more effective when fewer assumptions can be made about the input. STRUDEL$^L$ is also more flexible than rule-based approaches, such as PYTHEAS$^L$, in predicting cases that are not covered by the given rule set.

### Cell classification

For the cell classification task, we compare STRUDEL$^C$ with two aforementioned algorithms: (i) LINE$^C$ that provides a reasonable baseline, as most lines have homogeneous cells; (ii) RNN$^C$ that is based on an advanced deep learning architecture. The authors of RNN$^C$ evaluated their approach also without stylistic features, which allows a fair comparison to STRUDEL$^C$. We compared the F1-scores obtained by our re-implementation of their approach against the numbers reported in their paper and found that, for the three classes whose labels had not been revised by us, i.e., metadata, header, and notes, our results are very similar to theirs. The F1-scores on aggregate cells obtained by our implementation for all datasets are much lower than their reported ones, partly because our reforged annotations include more aggregate cells that were previously labeled as data cells – the difficult ones that cannot be easily found. Table 2.5 (bottom) summarizes the comparative result in terms of the per-class and macro-average F1-score, and accu-

racy. STRUDEL$^C$ surpasses both competitors. Meanwhile, the macro-average of RNN$^C$ shows an advantage against the baseline approach, although the per-class scores of the two approaches are on par with each other. Even though cell classification is a more imbalanced task than line classification, the performance of our STRUDEL$^C$ approach is comparable to its line counterpart.

Classifying group cells correctly is challenging for all approaches, as cells of this class are particularly rare. However, unlike other rare classes, such as metadata and notes, group cells are more likely to co-occur in the same line with data cells. LINE$^C$ reported a low F1-score particularly on group and aggregate cells across datasets. In fact, both group and aggregate cells often co-occur with other types in the same lines: some tables contain a group cell in a line with several aggregate cells; other tables have aggregate columns rather than lines, therefore causing the few aggregate cells in the lines with multiple data cells. Group and aggregate cells usually account for a minor amount in these cases.

LINE$^C$ applies a majority-take-all strategy to extend the line prediction result of a line to all its non-empty cells and therefore causes false negatives for group and aggregate cells in the above two cases. RNN$^C$ shows a low F1-score on the group class, which is not a class considered in the original paper [Gol et al., 2019], showing that the approach cannot be directly adapted to this class. The set of the reforged aggregate cells, many of which were misplaced in the original annotations from Gol et al., is also troublesome for RNN$^C$, which does not involve value calculation mechanisms to detect them.

### 2.4.3 STRUDEL performance evaluation

In this section, we present experimental results to gain insights on the following questions: (i) When does STRUDEL mis-classify an instance of a particular class, and which class is most likely to be considered? (ii) Do our approaches generalize to plain-text files that do not stem from spreadsheets? (iii) How do our used features affect performance? (iv) What are the typical reasons that cause these incorrect predictions?

#### Line classification

Table 2.5 has shown the per-class and overall F1 results of STRUDEL$^L$. In this section, we present our analysis of the classification results by using the confusion matrix. Figure 2.6 (top) shows the confusion matrix on executing STRUDEL$^L$ per dataset. The numbers in each confusion matrix are normalized row-wise to show the percentage of per-class instances misclassified as other classes. We leave out the confusion matrix for SAUS, which shows very similar results as that for GovUK. To create a confusion matrix with the repeated 10-fold cross-validation setting, we concatenate for each line in the files the predictions of all repetitions and construct an ensemble prediction for it with the majority voting strategy. To resolve possible ties, we stipulate that the fewer instances of a class included in the dataset, the more prior the class is.

Correctly identifying aggregate lines is the most challenging task across all datasets. These lines are mostly misclassified as data. The two main reasons for this are the lack of aggregate line training instances and the high similarity between aggregate and data lines, w.r.t. value types, and contextual characteristics. Around 11.4% of the aggregate lines are treated as headers for GovUK. We observed this to happen in many tables where

Figure 2.5: Per-class F1-score, and overall accuracy and F1-score on each dataset for *line* (top) and *cell* (bottom) classification.

| | | **metadata** | **header** | **group** | **data** | **aggregate** | **notes** | **accuracy** | **macro-avg** |
|---|---|---|---|---|---|---|---|---|---|
| GovUK | CRF$^L$ | **.789** | .379 | .898 | **.991** | .339 | .752 | **.979** | .733 |
| | PYTHEAS$^L$ | .446 | .444 | .172 | .986 | - | .545 | .970 | .518 |
| | STRUDEL$^L$ | .670 | **.774** | **.919** | .989 | **.361** | **.797** | .978 | **.751** |
| | *# lines* | *878* | *519* | *850* | *93,584* | *665* | *716* | - | - |
| SAUS | CRF$^L$ | .893 | .651 | .817 | .963 | .477 | .980 | .931 | .797 |
| | PYTHEAS$^L$ | .884 | .768 | .741 | .973 | - | .814 | .944 | .836 |
| | STRUDEL$^L$ | **.984** | **.960** | **.882** | **.987** | **.599** | **.984** | **.976** | **.899** |
| | *# lines* | *469* | *565* | *289* | *9,346* | *279* | *650* | - | - |
| CIUS | CRF$^L$ | **.994** | .961 | **.992** | **.996** | .749 | **.988** | .992 | .947 |
| | PYTHEAS$^L$ | .988 | .867 | .000 | .970 | - | .637 | .943 | .692 |
| | STRUDEL$^L$ | **.994** | **.972** | .984 | **.996** | **.834** | .978 | **.993** | **.960** |
| | *# lines* | *1,034* | *435* | *1,074* | *30,890* | *449* | *674* | - | - |
| DeEx | CRF$^L$ | .753 | .373 | .027 | .970 | .244 | .480 | .942 | .475 |
| | PYTHEAS$^L$ | .564 | .406 | .137 | .980 | - | .433 | .957 | .420 |
| | STRUDEL$^L$ | **.797** | **.807** | **.357** | **.989** | **.548** | **.761** | **.976** | **.710** |
| | *# lines* | *710* | *1,299* | *407* | *74,116* | *678* | *712* | - | - |

| | | **metadata** | **header** | **group** | **data** | **aggregate** | **notes** | **accuracy** | **macro-avg** |
|---|---|---|---|---|---|---|---|---|---|
| SAUS | LINE$^C$ | .963 | .915 | .451 | .970 | .332 | .888 | .930 | .753 |
| | RNN$^C$ | .977 | .925 | .466 | .956 | .345 | .902 | .919 | .762 |
| | STRUDEL$^C$ | **.987** | **.972** | **.752** | **.983** | **.689** | **.957** | **.968** | **.890** |
| | *# cells* | *469* | *4,769* | *825* | *142,301* | *8,708* | *695* | - | - |
| CIUS | LINE$^C$ | .991 | .973 | .361 | .929 | .156 | .937 | .824 | .725 |
| | RNN$^C$ | .987 | .976 | .679 | .904 | .443 | .963 | .850 | .825 |
| | STRUDEL$^C$ | **.993** | **.993** | **.916** | **.946** | **.465** | **.989** | **.895** | **.884** |
| | *# cells* | *1,035* | *3,838* | *4,228* | *310,354* | *47,043* | *674* | - | - |
| DeEx | LINE$^C$ | .630 | .625 | .155 | .981 | .258 | .520 | .955 | .528 |
| | RNN$^C$ | .623 | .772 | .347 | .952 | .244 | .413 | .930 | .559 |
| | STRUDEL$^C$ | **.689** | **.801** | **.444** | **.988** | **.683** | **.598** | **.977** | **.700** |
| | *# cells* | *975* | *10,314* | *1,216* | *749,403* | *21,245* | *1,076* | - | - |

aggregate lines are between header and data areas and separated from these two areas by empty lines. Note that when a line of a minority (non-data) class is misclassified, the wrong prediction tends to be data, which has much more instances than any other class. Apart from aggregate lines, header, group, and notes lines in DeEx also incorrectly lean towards the data class, because this dataset contains many tables of complicated structures. We discuss the kinds of mistakes in these categories in Section 2.4.3.

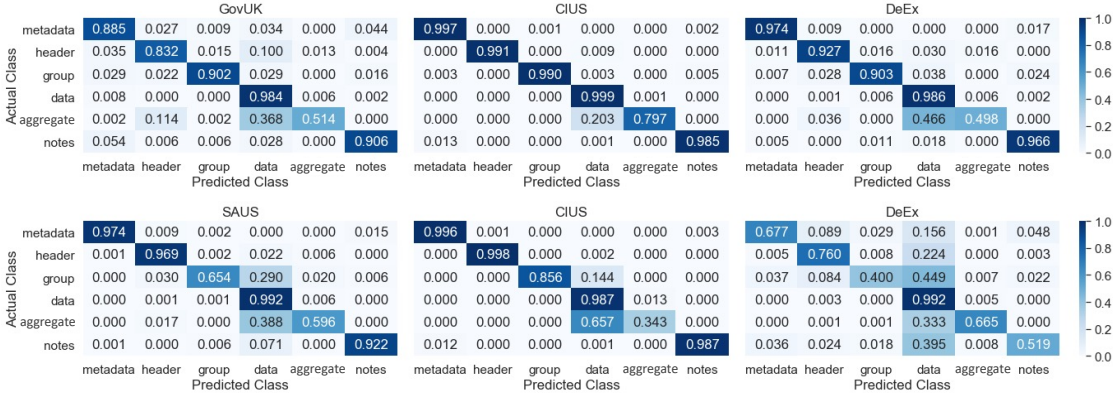



Figure 2.6: Confusion matrices to describe the pair-wise performance of STRUDEL$^L$ (top) and STRUDEL$^C$ (bottom) on individual datasets. The numbers are normalized by the number of instances per class.

### Cell classification

Figure 2.6 (bottom) depicts the per-dataset confusion matrix on executing STRUDEL$^C$. To create it, we applied the same procedure used to create the confusion matrix for STRUDEL$^L$.

Compared to the confusion matrix of STRUDEL$^L$, more classes have a higher misclassification ratio for all three datasets, showing that cell classification is a more challenging task than its line counterpart. On the one hand, the tendency to mark the instances of the minority classes as data is still prevalent. On the other hand, as the complexity of the problem increases, we do not observe many classification errors between two non-data classes, showing the effectiveness of our approach to distinguish between pairs of elements belonging to minority classes. About two-thirds of the aggregate cells are treated as data for CIUS. This is because a number of files share a fixed table schema that uses no keywords to indicate aggregate columns. Therefore, they are ignored by the aggregate cell detection component. For the other two datasets, around one-third of the aggregate cells are predicted as data. Overall, classifying aggregate cells correctly is not trivial. In the following, we discuss the reasons for such type of mis-classifications.

### Out-of-domain classification performance

To test the out-of-domain classification performance of STRUDEL, we kept the Troy dataset unseen during the algorithm design. We used a model trained on the collection of SAUS, CIUS, DeEx datasets to predict the classes of lines and cells in each file in the

Troy dataset.

The results in Table 2.6 show that, similar to the other datasets, group and aggregate cells are challenging for Strudel. After scrutinizing the input files, we found out that most of the aggregate cells lay in the lines that do not contain any indicative keywords, such as 'total', which determine the anchoring cells and therefore the aggregate cell candidates. A typical aggregate line contains few group cells (usually the left-most) and a number of numerical aggregate cells. Many of these aggregate lines are mis-classified as data, leading to the group cells therein also being mistaken.

Table 2.6: Per-class and overall F1-score on the Troy dataset.

| | Strudel$^L$ | # *lines* | Strudel$^C$ | # *cells* |
|---|---|---|---|---|
| metadata | .935 | 317 | .921 | 321 |
| header | .798 | 278 | .840 | 1,341 |
| group | .667 | 42 | .232 | 294 |
| data | .937 | 2,898 | .936 | 18,600 |
| aggregate | .070 | 239 | .216 | 1,935 |
| notes | .971 | 575 | .952 | 592 |
| macro-avg/num. | .730 | 4,349 | .683 | 23,083 |

**Feature analysis**

To understand which features exert more influence than others on particular classes, we calculated the feature importance for both Strudel$^L$ and Strudel$^C$ models. The mean decrease in the impurity mechanism [Dalton, 1920] has been used to calculate feature importance in the original random forest algorithm. However, this metric prefers continuous features and high-cardinality categorical features [Strobl et al., 2007, 2008]. There are a number of alternative techniques to calculate feature importance [Chandrashekar and Sahin, 2014]. Because many of our features are low-cardinality categorical features, we leveraged permutation feature importance, which has also been discussed by the original authors of the random forest algorithm [Breiman, 2001]. Permutation feature importance indicates the ability of one feature to distinguish instances of one class from those of another in a binary classification scenario. To adapt this metric to our multi-class classification problem, we trained a model for each class in a one-vs.-rest fashion, and use the permutation feature importance of each such binary classifier to represent the ability of our model to detect instances of that particular one class. The permutation of each feature was repeated five times and averaged.

Figure 2.7 illustrates the per-class feature importance for Strudel$^L$ (top) and Strudel$^C$ (bottom) with 100% stacked bars. The models are trained on the collection of SAUS, CIUS, and DeEx datasets. We grouped all neighbor profile features (Section 2.3.3) into neighbor value length and neighbor data type to reduce the complexity of the figure, because each individual feature has little importance on the cell classification task. Up to five most important features whose proportions are higher than 10% are highlighted.

The line type probability feature is the most important feature for notes, metadata, and header classes. The percentage of empty cells in the row is also quite useful for

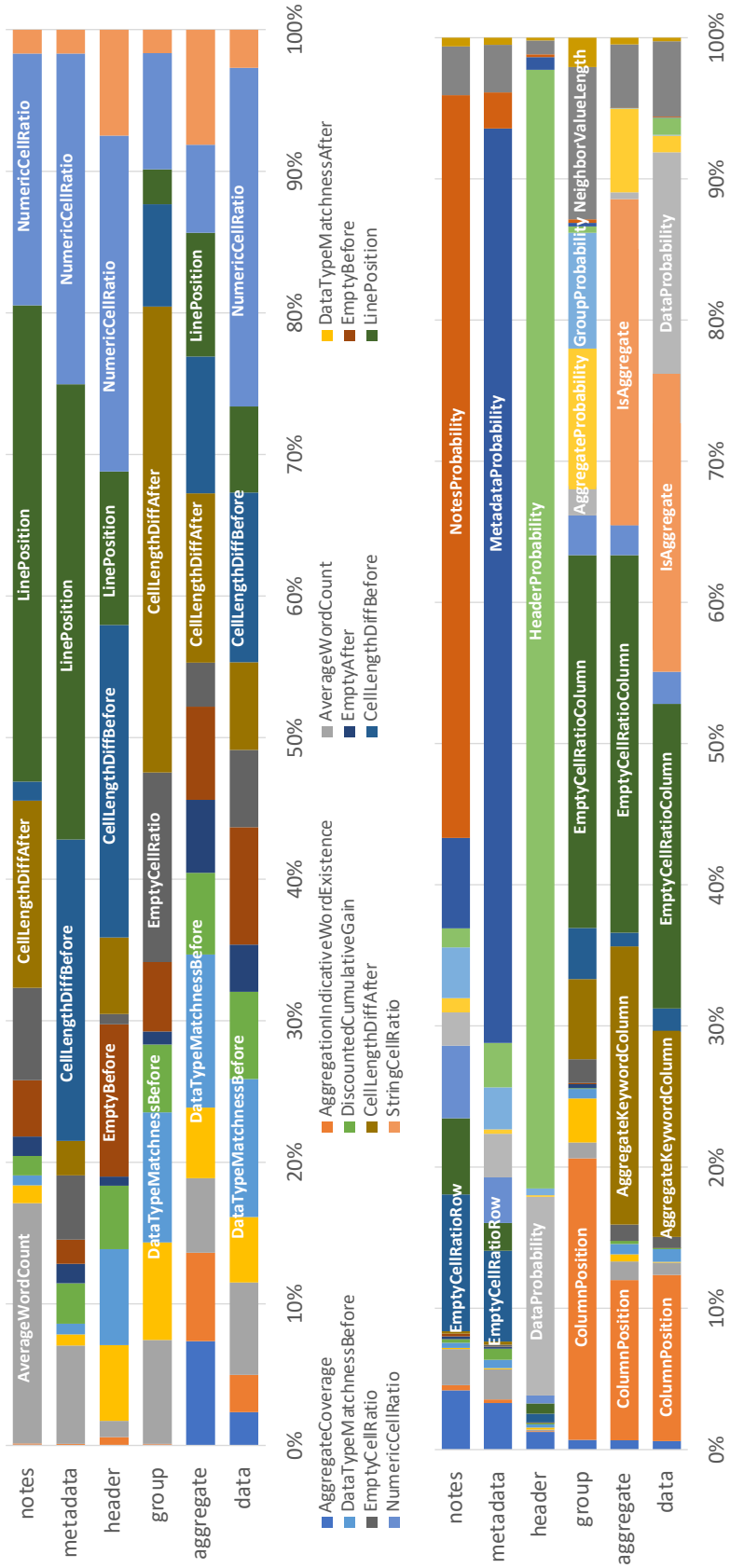



Figure 2.7: Feature importance of STRUDEL$^L$ (top) and STRUDEL$^C$ (bottom) trained on the collection of SAUS, CIUS, and DeEx. Several most important features for each class are highlighted.

notes and metadata. The percentage of empty cells in a column is most important to discover group cells: many group cells are in the left-most column (also indicated by the importance of the column position feature) of a file and span multiple rows. Neighbor profile features are useful for discovering group cells, proving that group cells tend to locate in specific places.

The novel feature signifying whether the value of a cell is the aggregation of other cells in the same line or column plays a great role in detecting aggregate cells, proving its effectiveness. Besides that, the existence of aggregate keywords in the same column is also important, indicating that users tend to use these words to mark the aggregate columns. However, the existence of aggregate keywords in the same line shows quite limited importance in our experiment, although we expected similar importance of it as its column counterpart.

### Performance on plain-text files

To test the performance of our Strudel algorithms on plain-text files that do not stem from spreadsheets, we tested the Strudel algorithm on the Mendeley dataset. We trained a model of our algorithm on the collection of SAUS, CIUS, and DeEx datasets, and used the whole Mendeley dataset for testing.

Table 2.7 displays the per-class and overall F1-score for this experiment. As mentioned above, files of the Mendeley dataset are mostly used to store (tabular) data. Therefore, the minority classes in the Mendeley dataset have very few instances.

While the overall F1-scores in this experiment are inferior to the respective ones shown in Table 2.5, they do show that even for such difficult files our approaches are well able to distinguish data from non-data. The values in Mendeley's plain-text files show properties different from traditional spreadsheets, e.g., the length of metadata and notes areas, the width of files. The second reason is that no data from Mendeley was included in the training phase. Therefore, dataset-specific properties are not learned properly by the classifiers. Also, different areas in a plain-text file might have their own delimiters. As the delimiter of the table areas is used across the file, it is possible to destroy the intrinsic structures of other areas, e.g., when using the comma character as the delimiter, the value of a note line is split across multiple cells.

Regarding the results of individual classes, our model treats quite a few metadata lines as data, as the delimiter of metadata and data areas are often different. At times, the delimiter dilemma also confuses our model of header lines in files where these lines are not split correctly. Out of the few aggregate cells, most are located in a single file, where the aggregate cells form a table by themselves, and aggregate on the values from another table, which is not recognizable by our model.

As the Mendeley dataset holds the biggest files across all our datasets, we also tested the scalability of our approach. The overall runtime on classifying cells of a file includes that for dialect detection, feature creation, and cell class prediction. Our experiments show that the overall runtime is linear to the number of non-empty cells. For a file of around 10MB, the whole procedure takes around 256s on a 1.4 GHz MacBook Pro with 16GB RAM. Most of the time is spent on creating the feature vectors. While we have few big files of such size, most files are only several kilobytes, probably because files with

verbose information are usually used to show limited information to readers, rather than store a big amount of data.

Table 2.7: Per-class and overall F1-score on the `Mendeley` dataset.

| | STRUDEL$^L$ | # *lines* | STRUDEL$^C$ | # *cells* |
|---|---|---|---|---|
| metadata | .623 | 604 | .245 | 2,152 |
| header | .406 | 86 | .629 | 769 |
| group | .263 | 27 | .303 | 44 |
| data | .999 | 194,786 | .999 | 1,356,635 |
| aggregate | .364 | 9 | .051 | 99 |
| notes | .448 | 86 | .380 | 111 |
| macro-avg/num. | .517 | 195,598 | .435 | 1,359810 |

**Analysis of difficult cases**

The confusion matrices shed light on which classes are most commonly mis-predicted for each class, either in the line or in the cell classification task. Here, we identify typical causes of those errors. The list below describes the pairs of common mis-classification cases (with $> 10\%$ incorrect classification in the class), e.g., mis-classifying 'aggregate as data', each followed by an error analysis based on our manual inspection of the results.

- **Aggregate as data.** In addition to the missing keywords introduced above, another reason that causes this type of mistakes is that some aggregate cells aggregate values from non-consecutive adjacent cells, which are ignored by our aggregate cell detection algorithm that assumes the value of an aggregate cell is calculated only by the values of the cells adjacent to it.
- **Header as data.** We found also two major reasons for this type of error: (i) a header line with a number of non-textual values adjacent to a data line may be mis-classified as part of the data area. Examples include numeric headers, such as year and date; (ii) in files with multiple vertically-stacked tables, headers of the tables towards the bottom of the stack have unusual line positions.
- **Notes as data.** First, tabular structures sometimes appear in notes lines, particularly in the `DeEx` dataset. Therefore, these tables of notes are likely to be treated as data. In some cases, authors place notes to the right of a table. Therefore, they are likely to be treated as data areas during cell classification.
- **Group as data.** One reason for this type of error is that some files have multi-level group columns, e.g., 'country-state-city', to the left of a table, followed by a number of data columns to the right of the table. As most tables have few group columns, the classifier may mis-interpret these rare cases as data. Another reason is that these group cells lay in the same lines as those aggregate cells, who are not captured by the aggregate cell detection algorithm, because there is no keyword in the same row or column. Therefore, the model tends to treat the whole row as a set of data cells.
- **Metadata as data.** Similar to the case in notes lines, metadata area may contain small tabular structures. Due to the tabular features of these metadata tables, STRUDEL tends to interpret them as data cells.

In summary, there are three aspects that mainly affect the correctness of our approach: (i) the geographical characteristic of vertically stacked multi-table files; (ii) the arithmetic calculation method for aggregate lines; (iii) the similarity between numeric header lines and data lines. These facts offer directions for improving our approach in future work. In Chapter 3, we present our work on an advanced aggregate cell detection algorithm AggreCol, which covers the sum and average functions. There, we present the improved results of Strudel by filling the IsAggregate feature values with the results of the AggreCol approach.

## 2.5 Related Work

Extracting information from semi-structured documents, such as verbose CSV files, spreadsheets, or web tables, has been a growing research topic in recent years. Relevant research questions include (i) how to locate tabular content in documents, such as PDF files [Liu et al., 2008], spreadsheets [Dong et al., 2019], or verbose CSV files [Vitagliano et al., 2021]; (ii) how to distinguish relational tables from non-relational tables [Cafarella et al., 2008; Wang and Hu, 2002]; (iii) how to extract relational tables from heterogeneous sources [Chen and Cafarella, 2013; Eberius et al., 2013; Elmeleegy et al., 2009; Shigarov and Mikhailov, 2017].

Prior to extracting information from a semi-structured document, understanding its structure is necessary. Some techniques have been proposed to address the structure detection problem on various types of documents, such as web tables and spreadsheets, which include tabular material and have a flexible layout. We summarize these works focusing on structure detection by classifying lines or cells, respectively. Image documents, such as PDF files and scanned documents, also commonly organize information with an arbitrary structure. A collection of research efforts have been made to extract information of different types from such files [Katti et al., 2018; Sarkhel and Nandi, 2019; Tata et al., 2021; Xu et al., 2020]. Due to the significant file format difference between verbose CSV files and image documents, we do not compare our work against these approaches. At the end of this section, we outline some works on converting tables with arbitrary shapes and forms into relational tables.

### 2.5.1 Line classification

Pinto et al. suggested a conditional random field (CRF) learning approach to predict the label for each line in plain-text documents crawled from a open data portal [Pinto et al., 2003]. For each document, a sequence of features is computed for its lines. The sequences of all documents are used by the CRF classifier to infer the label. This approach was later adopted to infer spreadsheet table schemata [Wachtel et al., 2016] and extract relational data from spreadsheets [Chen and Cafarella, 2013]. Moreover, it was extended by Adelfio et al. to recognize line classes in web tables and spreadsheets [Adelfio and Samet, 2013], which we marked as $\text{Crf}^L$ in our comparison. The authors suggested feature binning to generalize the training data and show the effectiveness of their approach on recognizing line classes in both HTML tables and spreadsheets crawled from several open data portals. However, the approach assumes the presence of stylistic features, such as font style, cell background color, or built-in spreadsheet formula features, which

are not available in verbose CSV files.

A recent work has proposed the rule-based approach Pytheas for CSV file line classification [Christodoulakis et al., 2020], marked as PYTHEAS$^L$ in our comparative experiment. To classify lines in a CSV file, the approach first determines for each line whether it is data or non-data with a set of fuzzy rules, whose weights have been learned beforehand with a training dataset. These binary results are then used to determine the top and bottom borders of tables in the file. Finally, the approach leverages additional class-specific rules on the discovered table/non-table areas to further ascertain the class of each line. The core of this approach is the design of the fuzzy rule set, which impacts the consequences of table border discovery, and also further line classification. However, such a fixed set of rules might fail to generalize to new circumstances in unseen data.

### 2.5.2 Cell classification

Finer-grained cell classification in data files has been the subject of academic research in recent years. Abraham et al. developed the UCheck framework that includes a component to detect "cell roles", such as header and data, in spreadsheets using several heuristics [Abraham and Erwig, 2007]. Cell roles are then used by the system to detect spreadsheet errors. The goal of their approach is to correlate cells in a table with their corresponding headers, thus they assume spreadsheets with only table regions as input data.

Gol et al. suggested a recursive neural network (RNN) architecture on two separately trained cell embeddings that capture the contextual and the stylistic semantics of cells, respectively [Gol et al., 2019]. We compared STRUDEL against this approach, which was marked as RNN$^C$ in our experiment. Even though the authors mentioned the contextual impact on a cell from both neighboring and distant cells, they considered only the former ones in their approach. They built the stylistic features upon those suggested by [Koci et al., 2016], which we dropped in our experiments for a fair comparison.

To discover the content layout in spreadsheets, Koci et al. proposed a supervised learning approach to classify cells, which employs a collection of features that fall into five types: *content*, *cell style*, *font*, *reference*, and *spatial*. The authors applied a post-processing component to repair classification errors [Koci et al., 2016]. The authors introduced five mis-classification patterns and suggested that the occurrence of them in the results hints at a mis-classification. Gol et al. have shown in their experiments that their RNN approach outperforms this algorithm w.r.t. F1-score. Therefore, we do not compare against this approach by Koci et al.

Chen et al. integrated an active learning technique into their spreadsheet cell classification approach [Chen et al., 2017]. In their iterative algorithm, a sheet selector presents the most uncertain spreadsheet to human labelers. The sheet is then labeled and included in a training set that is used to train a spreadsheet property classifier.

The above-introduced works all make use of stylistic features of their input. However, no such information can be obtained in verbose CSV files. In this work, we compare our approach only with the RNN-based approach of [Gol et al., 2019], as it was reported to outperform the other ones. In spite of using stylistic features to solve the task, the authors of the RNN$^C$ also reported the performance of their algorithm without their

usage, enabling a direct comparison to our approach.

Converting data in a table with an arbitrary shape into a relational table allows utilizing information in such ad-hoc tables with common query tools, such as SQL. Having associated a set of transformation rules with specific cell classes, Shigarov et al. proposed a rule-based approach to extract information from arbitrary tables into relational tables [Shigarov and Mikhailov, 2017]. FlashRelate addresses this problem with the programming-by-example technique [Barowy et al., 2015]. This approach accepts a few output relational tuple examples from users and constructs a program in its underlying language to obtain such output from the given table. With the assumption that tables are composed of top header rows, left header columns, and data regions, another type of works first recognizes the hierarchies of header rows and columns, and then allocate the headers for each cell in the data region of the table [Chen and Cafarella, 2013; Embley et al., 2011, 2016].

## 2.6 Conclusions

Often, valuable data are stored in semi-structured documents, such as verbose CSV files and spreadsheets, and cannot be directly extracted by common data management tools. Prior to extracting information from these files, it is necessary to understand their structure, by means of element classification, at either line or cell level. Previous works have addressed the line or cell classification problem for style-enriched documents, such as web tables or spreadsheets, making use of their stylistic features, such as font style, cell background color, cell margin style, and spreadsheet formulas. In this work, we proposed the Strudel approach to handle both tasks on verbose CSV files that, similar to spreadsheets, organize data in a flexible layout, yet lack stylistic features. We addressed the two classification problems separately and designed a set of features for each of them, including syntactic content features, contextual features, and computational features.

Our experimental results have shown that Strudel is able to achieve macro-average F1-scores that range from 0.71 to 0.96 on line classification for different datasets. For cell classification, the achieved F1-scores range from 0.70 to 0.89. These results were obtained across all semantic classes without using stylistic features. To conduct a fair comparison between Strudel and related work, we used only the non-stylistic features. We summarized reasons that cause common mis-classification cases, and recognized the effectiveness of computational features that are neglected by former studies, drawing key insights for further structure understanding research: (i) how to improve the prediction quality with semantic features; (ii) how can we extend the aggregate cell detection algorithm by recognizing more aggregation functions; (iii) whether column classification can help improve the classification quality. In the next chapter, we explore an advanced aggregate cell detection approach AggreCol. By replacing the values of the computational features used in this work with the results of AggreCol, we can improve the overall performance of the Strudel approach.

# Chapter 3

# Aggregation Detection in Verbose CSV Files

An *aggregation* represents an arithmetic relationship between a set of numbers and a single number: the latter (*aggregate*) can be calculated by applying an aggregation function on the former (*range*). A broad collection of functions can be used for arithmetic calculation, e.g., sum, difference, average, division, etc. Recall from Chapter 2 that tables in verbose CSV files often include aggregations to summarize data therein. There, we introduce the Strudel approach to classify each cell as one of six classes, including the "aggregate" type. Strudel makes two assumptions: (i) indicative keywords, such as "total" and "average", appear in some cells in the same line (either the same row or the same column) as an aggregate cell; (ii) the value of an aggregate cell must be derived from the same-line consecutive cells adjacent to them. According to the error analysis on the results of the Strudel approach, we found that neither of the assumptions is perfectly true. In fact, only around 60% of the sum aggregate cells in our datasets are in lines with indicative keywords. Also, the pattern of range described in (ii) above is just one of three possible cases summarized in Section 3.2.2 below. To obtain more correct aggregate detection results, and further improve the qualitative performance of Strudel or other cell classification approaches [Adelfio and Samet, 2013; Koci et al., 2016; Nargesian et al., 2018], a more advanced aggregate detection approach is needed. What is more, detected aggregations can also be used for enriching files with metadata, detecting and cleaning numeric errors, and serving as input for formula smell detection approaches, which we discuss in detail in Section 3.1.

In this chapter, we present our three-stage approach AggreCol to identify *aggregations* of five types, i.e., sum, difference, average, division, and relative change, in verbose CSV files. This work is based on our publication [Jiang et al., 2022]. The goal is to detect not only the aggregate cells, but also the set of cells to calculate these aggregates. In short, the first stage discovers aggregations of each function individually. The second stage collects the individual detection results and removes the spurious aggregations with a set of pruning rules. The last stage aims at recognizing non-adjacent aggregations by skipping the aggregates of detected aggregations. We evaluated our approach with two manually annotated datasets, showing that AggreCol is capable of achieving 0.95 precision and recall for 91.1% and 86.3% of the files, respectively. We obtained

similar results on an unseen test dataset, showing the generalizability of our proposed techniques.

Overall, this work makes the following contributions:

(i) We formalize the aggregation detection problem for verbose CSV files, and propose the three-stage approach AggreCol to address it for five aggregation functions.
(ii) We annotated two datasets that comprise 466 verbose CSV files of various domains, which include more than 26K aggregations, published on our website[1].
(iii) We conducted a series of comparative experiments to evaluate the effectiveness of AggreCol. Also, we verified that Strudel can achieve better cell classification performance using the results of AggreCol.

We organize the rest of this chapter as follows: we first discuss the need to identify aggregations with three typical use cases, and the demand for automated approaches in Section 3.1. In Section 3.2, we formalize the definitions of the used terms, and possible aggregation patterns, followed by the formal problem statement. After that, Section 3.3 describes the technical details of our approach. The experimental results along with a detection error analysis are presented in Section 3.4. Section 3.5 briefly discusses the related work. As a conclusion, we summarize this work and envision future work in Section 3.6.

## 3.1 Aggregation detection

As a useful tool to summarize data, aggregations are implemented as operators in many data analytics tools. For example, relational database systems support a collection of aggregation operators based on the underlying relational algebra, such as sum, average, count, and minimum/maximum, to enable data analysis on relational tables. Spreadsheet programs provide an extensive set of formulas to compute aggregations on values of arbitrary cells [Rahman et al., 2020]. Popular data analytic libraries, such as Pandas [development team, 2020] in Python and Apache Spark [Zaharia et al., 2016], also provide aggregation calculations on DataFrames. Various business intelligence and data analytics platforms, such as Tableau[2] and Trifacta[3], integrate aggregation calculation in their toolkits. Data scientists and practitioners frequently implement pipelines to prepare data or carry out data analytic tasks with the aforementioned tools. While building these pipelines, they often acquire insights on how to process or analyze data in the next step by looking up various kinds of aggregations amongst numbers. A common function shared by these tools is "Export to CSV", which produces plain-text CSV files that can be processed by a wide variety of applications. Many verbose CSV files with aggregations have been created with this function.

While aggregations are prevalent in verbose CSV files, locations of them (the aggregate and range cells) are not always obvious. One reason is the lack of proper metadata to describe aggregations. Because verbose CSV files cannot preserve any metadata, an aggregate appears to be just a normal numeric cell similar to any other numeric data

[1] https://hpi.de/naumann/projects/data-preparation/aggregation-detection-in-verbose-csv-files.html
[2] https://www.tableau.com/
[3] https://www.trifacta.com/

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 0 | Population by age 1875–2009 | | | | | | | |
| 1 | Year | Population | Age 0–14 | Age 15–64 | Age 65+ | 0–14 % | 15–64 % | 65+ % |
| 2 | 1875 | 1 912 647 | 659 267 | 1 178 113 | 75 267 | 34,5 | 61,6 | 3,9 |
| 3 | 1900 | 2 655 900 | 930 900 | 1 583 300 | 141 700 | 35,1 | 59,6 | 5,3 |
| 4 | 1925 | 3 322 100 | 1 031 700 | 2 090 000 | 200 400 | 31,1 | 62,9 | 6,0 |
| 5 | 1950 | 4 029 803 | 1 208 799 | 2 554 354 | 266 650 | 30,0 | 63,4 | 6,6 |
| 6 | 1975 | 4 720 492 | 1 030 544 | 3 181 376 | 508 572 | 21,8 | 67,4 | 10,8 |
| 7 | 2000 | 5 181 115 | 936 333 | 3 467 584 | 777 198 | 18,1 | 66,9 | 15,0 |
| 8 | 2009 | 5 351 427 | 888 323 | 3 552 663 | 910 441 | 16,6 | 66,4 | 17,0 |
| 9 | | | | | | | | |
| 10 | Source: Population Structure 2009, Statistics Finland | | | | | | | |
| 11 | Inquiries: Markus Rapo (09) 1734 3238, vaesto.tilasto@stat.fi | | | | | | | |
| 12 | Director in charge: Jari Tarkoma | | | | | | | |

Figure 3.1: A real-world verbose CSV file that contains two types of aggregations: sum (green) and division (blue).

cell in a table. Figure 3.1 illustrates a real-world verbose CSV file from the `Troy` dataset. Besides the title and footnotes at the top and bottom left, the file contains a table that clarifies the overall and per age-group population for several years in Finland. The headers contain no indicative keywords to hint at the presence of the aggregations. The three right-most columns account for the proportion of the per-group population against the total. However, Neither the total nor the single group population is adjacent to the corresponding ratio cell. As some verbose CSV files are exported from spreadsheet files, one might refer to the original spreadsheets, if available, which might contain the actual aggregation function as metadata. However, these metadata are not always present, even in spreadsheets, where users can copy and paste only the values derived by a formula. Based on our observation on the spreadsheet version of the `Troy` dataset, 150 of the 200 spreadsheet files have at least one aggregation, while none of them include the corresponding formulas.

With the lack of aggregation metadata in verbose CSV files, discovering them is beneficial for several downstream applications:

**Enriching files with metadata.** Metadata, despite being useful information to help understand data, cannot be embedded into the vanilla CSV file format that is commonly used to store data. Specifically, locations of aggregation cells can reveal the arithmetic relationships between numeric cells. In addition, detected aggregations can be used to improve cell classification algorithms that usually treat "aggregate" as a cell type [Gol et al., 2019; Jiang et al., 2021; Koci et al., 2016]. In Section 3.4.6, we show the improvement gain of our Strudel approach on cell classification by using the results of our aggregation detection algorithm proposed in this chapter to fill a binary feature that represents whether a cell is an aggregate or not.

**Numeric error detection and cleaning.** Aggregations in verbose CSV files often

include errors: the number in the aggregate cell does not precisely calculate the numbers in the cells of the range. A major reason is that numbers are rounded to preserve a certain amount of decimal digits. The calculated aggregation of a group of rounded numbers may deviate from the intended aggregate value – an effect we have observed in around 29% of the real-world aggregations, as described in Section 3.4.1. With our advanced approach to detect aggregations with an error, data scientists may realize the numeric data errors and mend them accordingly.

**Serving as input for formula smell detection.** Many verbose CSV files are exported from spreadsheets where the aggregations have been created via formulas. However, manual mis-operations may cause the absence or incorrectness of formulas in spreadsheets. For example, in a row of numeric cells that represent the per-column sum of numbers in other rows, the formula of one cell is missing, which can be identified by various "formula smell" detection approaches [Dou et al., 2014; Jansen and Hermans, 2015]. However, these approaches all assume the existence of some surrounding formulas, e.g., the other cells in the row with the sum formula in the above case. Detected aggregations can serve as the input for those formula smell detection approaches to recognize more smelly formulas.

With the usefulness of knowledge about aggregations and the lack of such information in verbose CSV files, being able to detect them is important. Manual work is infeasible, as locations of aggregations are not always apparent: any numeric cell could aggregate any set of other cells. When dealing with a large table, manual checking is error-prone and time-consuming due to many aggregation candidates. The existence of specific keywords in a cell, such as 'total' and 'average', might indicate the presence of aggregates in the same row or column. However, such a keyword-based approach could be unreliable, as keyword dictionaries can hardly cover all aggregations. In the table of Figure 3.1, for example, none of the aggregation cells can be identified by their headers. An investigation into our dataset shows that we could retrieve only 60.0% of the real sum aggregates by using a set of keywords including 'total', 'all', 'sum', 'subtotal', and 'overall'. In Section 3.4.4, we elaborate on the results of this keyword-based approach. In light of these challenges, automated aggregation detection approaches without relying on a limited set of indicative keywords are desirable.

## 3.2 Preliminaries

In the first part of this section, we provide definitions of all necessary concepts, including tables in verbose CSV files, the components of an aggregation (aggregate, range, and aggregation function), and the error level. In the next parts, we summarize the three observed aggregation patterns and give the formal problem statement.

### 3.2.1 Definitions

Recall that a file dialect incorporates all utility characters used to interpret its structure, such as field delimiter, quotation character. While comma and double-quote characters are widely used, no mandatory requirements are enforced for their selection. To interpret a file as a verbose CSV file, the corresponding file dialect must be recognized

first [Döhmen et al., 2017; Ge et al., 2019; van den Burg et al., 2019]. Here, we assume verbose CSV file dialects have been correctly detected, and the input files are delimited accordingly. As verbose CSV files allow a loose content layout, tables therein may have unique structures.

**Definition 2** (Table)**.** A table in a verbose CSV file is a group of cells that can form any shape. Each cell carries information of a particular purpose, such as data, row or column header, group header, aggregate, or empty visual separator. A table contains structured information about entities of a common type.

Stemming from spreadsheets, tables in verbose CSV files often contain sets of numbers and calculated statistics about them. The verbose CSV file in Figure 3.1 includes a table that spans rows 1 to 8, and columns 0 to 7. This table includes cells that serve as row header, column header, data, or aggregate. Tables of verbose CSV files often express the numeric value zero with an empty table cell. We follow this interpretation in this work.

An aggregation in a verbose CSV file describes an arithmetic relationship between a numeric cell and a set of other numeric cells in the same table. We refer to the former numeric cell as the *aggregate* and the latter numeric cell set as the *range*. We refer each component in a range as a *range element*. A table may include multiple aggregations.

**Definition 3** (Aggregate and Range)**.** An *aggregate* $r = c_{i_r,j_r}$, where $i_r = 0, \ldots, M-1$ and $j_r = 0, \ldots, N-1$, is a numeric cell whose value can be derived by applying a specific arithmetic function on a set of other numeric cells $E = \{c_{i,j} | i = 0, \ldots, M-1, j = 0, \ldots, N-1\}$, referred to as a *range*. We represent the list of row and column indices of the elements in $E$ with $i_E$ and $j_E$.

**Definition 4** (Aggregation function)**.** An aggregation function $f$ is an arithmetic operator that can be applied to the values of the elements in a range to determine the value of an aggregate.

While many different aggregation functions can be applied to numeric values, only few appear frequently in verbose CSV files based on our annotation of 385 files. Figure 3.2 illustrates the occurrence distribution of aggregation functions. In this work, we cover the aggregation functions that appear in more than 5% of the files, i.e., *sum*, *difference*, *average*, *division*, and *relative change*. Table 3.1 displays the specifications of these aggregation functions. For example, a sum function requires no less than one element in the range, and is a *cumulative* function, which means an aggregator derived by applying this function can be further used as a range element in other aggregations. In contrast, average is *non-cumulative*: averaging numbers that are themselves averages is not arithmetically meaningful. In fact, calculating the average over a collection of mean values based on sets of numbers should take the cardinality of each set into account, which cannot be uncovered only from these mean values. Relative change describes the change from $B$ to $C$ normalized by $B$, which is commonly used to show changes across a certain time period.

In this work, we treat only single-function aggregations, although we observe that a few real-world aggregations involve multiple aggregation functions. For example, the percentage of a population holding at least a university degree is the sum of populations with bachelor, master, and doctorate degrees divided by the total population. Detecting multi-function aggregations would be interesting future work.

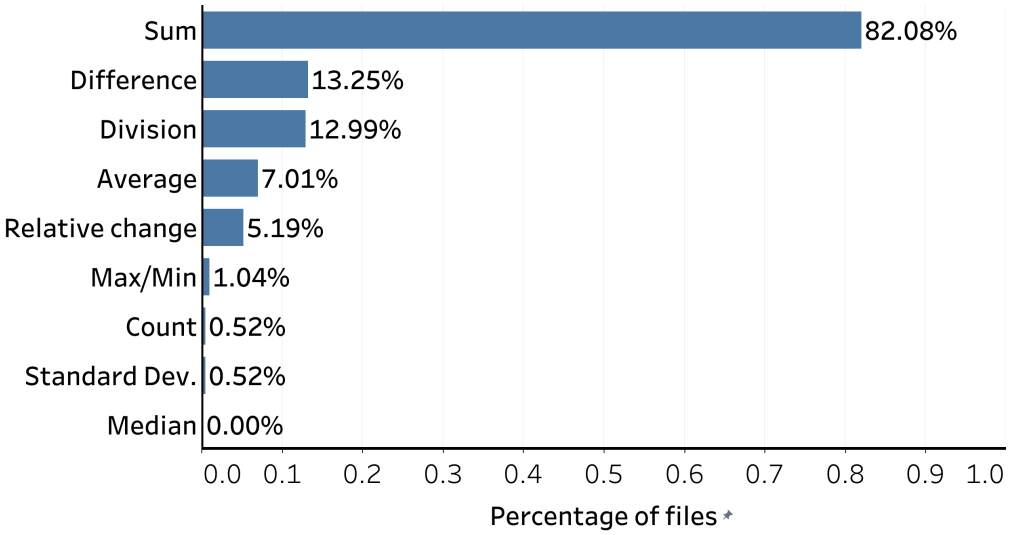



Figure 3.2: Percentage of 385 files that contain the individual aggregation functions.

Table 3.1: Overview of the supported aggregation functions.

| **Aggregation function** | **# Range elements** | **Formula** | **Cumulative** |
|---|---|---|---|
| Sum | $\geq 1$ | $A = \sum_{i=1}^{n} B_i$ | Yes |
| Difference | $= 2$ | $A = B - C$ | Yes |
| Average | $\geq 1$ | $A = (\sum_{i=1}^{n} B_i)/n$ | No |
| Division | $= 2$ | $A = B/C$ | No |
| Relative change | $= 2$ | $A = (C - B)/B$ | No |

Having defined all components of an aggregation, we can formalize the definition of the aggregation as follows.

**Definition 5** (Aggregation)**.** An aggregation $a$ is a tuple $(r \leftarrow E, f)$, where $r \notin E$ and the value of the cell $r$ can be derived by applying the arithmetic function $f$ on all values in $E$.

In principle, the value of $r$ should equal precisely the value calculated by applying $f$ on $E$. However in practice, this is often not true as numbers displayed in cells can be rounded. In fact, we observed that around 30% of all aggregations in our manually annotated files contain such errors. Numbers can be rounded to different significant figures, leading to a wide variety of error margins. As verbose CSV files do not preserve the metadata on how numbers were rounded, we model such a calculation deviation with an *error level* and detect aggregations for different levels.

**Definition 6** (Error level)**.** Given an aggregation $a$ $(r \leftarrow E, f)$, where $r$ is the observed aggregate value, and $r'$ the value determined by applying $f$ on $E$, the error level $e$ of $a$ describes the deviation factor from $r$ to $r'$: $e = |(r' - r)/r|$

We calculate the error level with the normalized absolute difference, because numeric

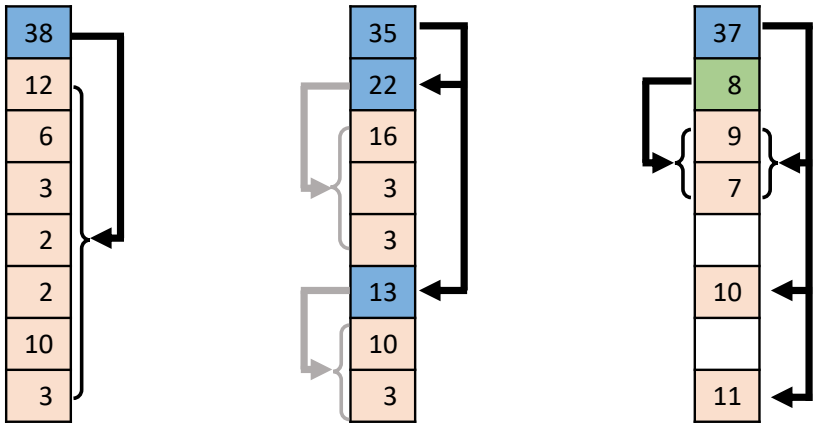
values of aggregates vary in their order of magnitude. Based on this formula, the highest observed error for a true aggregation was 37.5%. The error level is undefined if $r = 0$. Note that for the few such cases in our dataset, we calculate the error level as the absolute difference between $r$ and $r'$. We extend the notation of an aggregation to include the error level: $a\ (r \leftarrow E,\ f,\ e)$.

While in principle both $r$ and $E$ can be located at any arbitrary position in a verbose CSV file, in practice, most aggregations organize $r$ and $E$ in the same row (row-wise) or column (column-wise). We make this *same-line aggregation assumption*, because almost all single-function aggregations of the addressed function type follow this assumption according to our observation. Discovering non-same line aggregations can be an interesting extension, but is likely to be fraught with many false positive cases.

To express a row-wise aggregation, we use a slightly modified notation $(row{:}i, j_r \leftarrow j_E, f, e)$, where $i$ is the row index of all cells; $j_r$ and $j_E$ are column indices of the aggregate and the range, respectively; $f$ is the aggregation function; $e$ is the observed error level. For example, the green-shaded sum aggregation in Figure 3.1 can be represented as $(row{:}2, 1 \leftarrow \{2, 3, 4\}, Sum, 0)$. We call $j_r \leftarrow j_E$ the *pattern* of the aggregation that indicates the scope of the aggregate and the range. Column-wise aggregations can be represented analogously. For example, $(column{:}j, i_r \leftarrow i_E, f, e)$ states that in column $j$, the number in the cell with row index $i_r$ is calculated by applying $f$ on the numbers in the cells with row indices $i_E$.

### 3.2.2 Taxonomy of aggregation patterns

Given a row-wise or column-wise aggregation, range elements can be any subset of the numeric cells in the same column or same row. However, we observe that range elements are usually not randomly distributed. Authors of verbose CSV files tend to place aggregates close to their corresponding ranges. We summarize three aggregation patterns, shown in Figure 3.3, based on our examination of our dataset that includes 385 verbose CSV files.




(a) Adjacent (b) Cumulative (c) Interrupt

Figure 3.3: Aggregate-range patterns of three different aggregations (Blue and green cells are sum and average of the corresponding orange cells).

The simplest and the most common pattern is *adjacent*, where the contiguous range is next to the corresponding aggregate. Empty cells are allowed to appear in-between range elements without breaking the contiguity. Out of the examined verbose CSV files, 77.9% include adjacent aggregations. The high ratio is likely due to localized reading habits: humans tend to find the summary of a group of numbers in its direct proximity.

The second pattern is *cumulative*. As shown in Figure 3.3b, the value 35 is the sum of 22 and 13, which is not an adjacent aggregation. However, the latter two numbers are themselves aggregates of two different adjacent aggregations. For example, the global population equals to the sum of populations of northern and southern hemispheres, which, in turn, equals those of a number of countries therein. We have observed such a pattern in around 20% of the files. Note that to obtain a cumulative aggregation, the intermediate aggregates (22 and 13 in the figure) must be of cumulative nature: for the aggregations considered in this study, only sum and difference can behave cumulatively.

Lastly, 14.8% of our verbose CSV files contained *interrupt* aggregations. The range elements in an interrupt aggregation are scattered in the row or column. Interrupt aggregation happens when, for example, it is blocked by another non-cumulative aggregation (green shaded cell in Figure 3.3c is the average of the next two cells), an intermediate aggregation is actually not satisfied, or rows and columns are simply not organized in a localized manner.

### 3.2.3 Problem statement

The aforementioned three patterns summarize the appearance of aggregations, and helped shape our approach. We can now formally define the problem of *aggregation detection in verbose CSV files*: given a verbose CSV file $V = \{c_{i,j} | i = 0, \ldots, M-1, j = 0, \ldots, N-1\}$, a set of functions $F$, a given error level threshold $e'$, find all aggregations in $V$ using $F$ presented in Table 3.1. Each aggregation satisfies the same-line aggregation assumption, and has the appropriate number of range elements, according to Table 3.1, i.e., each detected aggregation $a$ ($r \leftarrow E$, $f$, $e$) must satisfy the following four requirements:

(i) $f \in F$;
(ii) $e \leq e'$;
(iii) either $\forall c_{i,j} \in E : i = i_r$ or $\forall c_{i,j} \in E$, $j = j_r$;
(iv) $|E| \geq 1$ if $f \in \{sum, average\}$, or
$|E| = 2$ if $f \in \{difference, division, relative\ change\}$.

## 3.3 The AggreCol Approach

In this section, we present the methods used by our three-stage algorithm AggreCol to detect aggregations in verbose CSV files. We start by describing the overall workflow of our approach, and then elaborate on the individual, collective, and supplemental aggregation detection stages in Sections 3.3.1-3.3.3.

Figure 3.4 illustrates the overview of AggreCol. Given a verbose CSV file, AggreCol first detects aggregations of different aggregation functions separately. Because some results of this step might be incidental, the results are then passed to the collec-

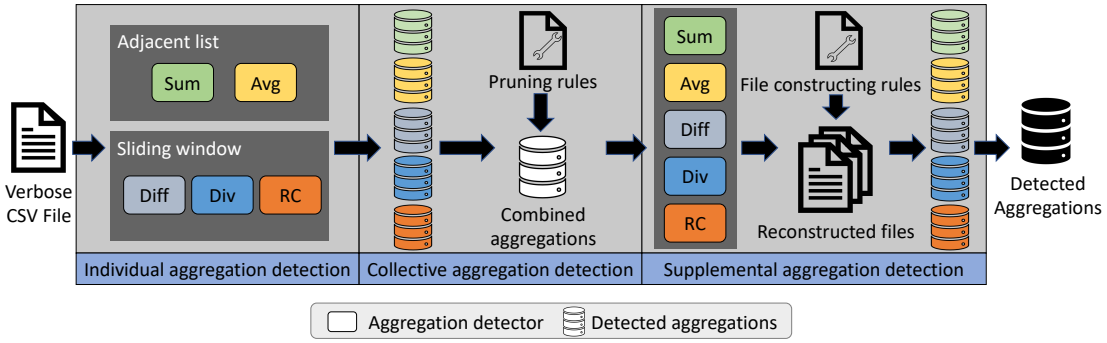



Figure 3.4: Overall workflow of our three-stage AggreCol approach. Round-cornered rectangles represent detectors for individual aggregation functions. The terms "Sum/Avg/Diff/Div/RC" stand for "Sum/Average/Difference/Division/Relative change", respectively. Each function is associated with a unique color.

tive aggregation detection stage, where we consider the results of all individual detectors as a whole and remove spurious aggregations with a set of pruning rules. To specifically handle interrupt aggregations as defined in Section 3.2.2, we introduce a final stage that constructs new files from the original verbose CSV file, on which we re-apply the individual aggregation detectors. The third stage outputs the final results.

According to the same-line aggregation assumption, the aggregate and the range of an aggregation are either in the same row or the same column. Our approach handles row-wise and column-wise aggregations equivalently in two respective passes over the data. Without loss of generality, the examples and terminology used in the rest of this section are focused on row-wise aggregation detection.

### 3.3.1 Individual aggregation detection

AggreCol expects a verbose CSV file as input. Prior to detecting individual aggregations, we must first understand how values in numeric cells in a file are formatted. Each file may use a unique number format. For example, one file may use dot as thousands separator and comma as decimal separator, while another may use space and comma for them, respectively. The interpretation of numbers may affect the results of aggregation detection. Section 3.4.2 clarifies our method to recognize the number format and interpret the underlying numbers in the file.

AggreCol detects aggregations of each aggregation function separately. Algorithm 3 outlines the procedure of the individual aggregation detection algorithm. It takes as input a verbose CSV file $V$, the function $f$ it detects aggregations of, the maximum tolerable error level $e'$, and the minimum coverage $cov$ of aggregates in a row or column. To help understand the procedure, we use Figure 3.5 as a working example: it shows a table with four row-wise aggregations whose formulas are shown below the table. Numbers in the table are modified to fit the example. We set $e' = 0$, and $cov = 0.7$.

The individual aggregation detection algorithm starts by recognizing, within each row, the aggregations whose range is adjacent to its aggregate (lines 4-7). This phase employs either an adjacency list strategy or a sliding window one, depending on whether

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | Year | Europe | Bulgaria | France | Germany | Poland | Portugal | Romania | Africa | Kenya | Ethiopia | Chile | Total pop. change | Kenya in Africa |
| 1 | 2013 | 3703 | 215 | 930 | 1278 | 1216 | 62 | 2 | 64 | 58 | 6 | 128 | 3895 | 0.91 |
| 2 | 2014 | 4038 | 546 | 959 | 1145 | 1388 | -243 | 243 | 22 | 6 | 16 | 78 | 4138 | 0.27 |
| 3 | 2015 | 3900 | 307 | 736 | 1573 | 1263 | 90 | -69 | 23 | 6 | 17 | 123 | 4046 | 0.26 |
| 4 | 2016 | 4830 | 279 | 1176 | 1683 | 135 | 1548 | 9 | 19 | 10 | 9 | 197 | 5046 | 0.53 |
| 5 | 2017 | 4944 | 378 | 1669 | 2897 | -305 | 228 | 77 | 21 | 8 | 14 | | 4965 | 0.38 |
| 6 | 2018 | 5791 | 900 | 2583 | 1148 | 1127 | 21 | 13 | 34 | 21 | 12 | | 5825 | 0.62 |
| 7 | 2019 | 8266 | 364 | 4155 | 3550 | 164 | 22 | 11 | 33 | 14 | 19 | | 8299 | 0.42 |

$a_1$: $C_1 = C_2 + C_3 + C_4 + C_5 + C_6 + C_7$; $a_2$: $C_8 = C_9 + C_{10}$;

$a_3$: $C_{12} = C_1 + C_8 + C_{11}$; $a_4$: $C_{13} = C_9/C_8$

Figure 3.5: Excerpt of a table that contains three sum (in green) and one division (in blue) aggregations. $C_i$ represents the $ith$ column.

the applied aggregation function is commutative. The two strategies are explained in the following.

**Adjacency list strategy.** This strategy recognizes an aggregation only if the set of all numeric elements in its range is adjacent to the aggregate, e.g., $a_1$ and $a_2$ in Figure 3.5. We employ this strategy for aggregation functions that are commutative. The commutative property guarantees that changing the order of the elements in a range has no impact on the aggregated result. For the functions addressed in this work, sum and average hold such property. The commutative property allows us to resort to a greedy approach: for each aggregate candidate $c_{i,j}$ and the numeric cells $E = \{c_{i,k}|k > j\}$ in its row, the approach iteratively moves the numeric cell in $E$ that is column-wise closest to $c_{i,j}$ into an adjacency list $L$. After each move, it checks if the error level between $c_{i,j}$ and the value calculated by applying $f$ on $L$ is smaller than the given $e'$. If such a list $L$ can be found, $c_{i,j}$ and $L$ compose a detected aggregation. The same process is applied also to the numeric cells $E = \{c_{i,k}|k < j\}$ to detect possible aggregations at the other side of the aggregate candidate. Although the range of a sum or average aggregation may have only one element, such single range element cases rarely appear in our datasets. However, allowing the detection of such aggregations would include massive false positive cases in the results, especially for files with many cells of identical numbers. Therefore, our approach requires at least two range elements for these two functions.

**Sliding window strategy.** For non-commutative aggregation functions, the order of range elements has an impact on the result. As the greedy adjacency list strategy is not applicable, we use a window to generate range candidates. For each aggregate candidate $c_{i,j}$, the approach creates one aggregation candidate for each permutation of size $n$ in the set of $w$ numeric cells that are closest to $c_{i,j}$, where $n$ is the number of possible range elements of the function specified in Table 3.1. Then, the approach checks every

**Algorithm 3:** Individual aggregation detection (row-wise)

**Input:** Verbose CSV file $V$, aggregation function $f$, error level $e'$, line aggregation coverage threshold *cov*
**Output:** A set of row-wise aggregations D

```
1  D ← {};
2  while true do
3  |  m ← length(V);
4  |  foreach i in {0,...,m} do
5  |  |  D_m ← DetectAdjacentAggregations(V_m, e');
6  |  |  D' ← D' ∪ D_m
7  |  end
8  |  D' ← MendAggregations(D');
9  |  if D' = ∅ then
10 |  |  break;
11 |  D' ← Prune(D', cov);
12 |  D ← D ∪ D';
13 |  if f is not cumulative then
14 |  |  break;
15 |  V ← RemoveColumns(V);
16 end
17 return D
```

aggregation candidate. Similar to the adjacency list strategy, this process is also applied to both sides of $c_{i,j}$ separately. We employ this strategy when detecting difference, division, and relative change aggregations.

Applying the greedy adjacency list strategy may yield incorrect aggregations in particular situations. For example, $(\textit{row:}2, 1 \leftarrow \{2,3,4,5,6,7\}, \textit{Sum}, 0)$ cannot be retrieved, as a previously detected aggregation with a shorter range $(\textit{row:}2, 1 \leftarrow \{2,3,4,5\}, \textit{Sum}, 0)$ terminates the search for this aggregate candidate. However, the detected aggregation is in fact not a true one. It occurs because the sum of $c_{2,6}$ and $c_{2,7}$ happens to be zero. We assume that, although such patterns might appear in few rows due to a computational coincidence, it is very unlikely that they repeat across all rows in the table. To discover the above true aggregation and also drop the spurious one, AggreCol mends the detected aggregations by checking whether candidates with the same patterns as the detected aggregations across rows are also valid aggregations (line 8). In the case of Figure 3.5, as $(\textit{row:}3, 1 \leftarrow \{2,3,4,5,6,7\}, \textit{Sum}, 0)$ is a valid aggregation, we check the candidates with the same pattern in the other rows and validate the candidates for rows 2 and 5. These two candidates could not be retrieved, because an aggregation with a shorter range in the respective row was detected. However, $(\textit{row:}1, 1 \leftarrow \{2,3,4,5\}, \textit{Sum}, 0)$ is not valid, even though it shares the same pattern as the detected $(\textit{row:}2, 1 \leftarrow \{2,3,4,5\}, \textit{Sum}, 0)$. Table 3.2 lists the detected aggregations with no errors after mending for the example in Figure 3.5.

Note that some rows are not compliant with specific column patterns. For example, $(\textit{row:}6, 1 \leftarrow \{2,3,4,5,6,7\}, \textit{Sum}, 0)$ cannot be detected as the numbers in columns 2-7 in row 6 add up to 5792, deviating slightly from the aggregate value "5791" in column 1.

Table 3.2: Detected row-wise sum aggregations after mending grouped by column patterns ($e' = 0$).

| | Column pattern | Compliant rows |
|---|---|---|
| 1 | $1 \leftarrow \{2, 3, 4, 5, 6, 7\}$ | 1, 2, 3, 4, 5, 7 |
| 2 | $4 \leftarrow \{5, 6\}$ | 1, 2, 3, 4, 6 |
| 3 | $8 \leftarrow \{6, 7\}$ | 1, 2, 6, 7 |
| 4 | $8 \leftarrow \{9, 10\}$ | 1, 2, 3, 5, 6, 7 |

If there are no detected aggregations in any row, the entire process is terminated (line 10). Otherwise, the algorithm removes spurious detected aggregations with a pruning step (line 11).

To perform the pruning step, all aggregations are first grouped by their patterns. Any group possessing an insufficient amount of aggregations is discarded. The sufficiency score is calculated as *the number of aggregations in the group* normalized by *the number of numeric cells in the same row*. Additionally, for all groups sharing the same aggregate, only the one with the highest sufficiency score is preserved. The same process is conducted also for the same-range groups. The surviving groups are organized in a ranked list, where a group with a higher rank has (i) the more detected aggregations; (ii) the smaller average actual error level of the aggregations. After that, AggreCol iterates over the list and prunes the lower-ranked groups whose patterns cannot co-exist with the pattern of the currently inspected group, according to the following three rules.

**Directional disagreement.** Two aggregation candidates may share the same aggregate, yet develop their ranges in the two different directions, e.g., $(row{:}3, 4 \leftarrow \{5, 6, 7\}, Sum, 0)$ and $(row{:}3, 4 \leftarrow \{2, 3\}, Sum, 0)$. If such two candidates use the same aggregation function, we allow their ranges to reside only at the same side of the aggregate, reflecting the typical structure of spreadsheets.

**Complete inclusion.** For two aggregation candidates, a complete inclusion scenario appears if both the aggregate and part of the range elements of one aggregation are contained in the range of the other aggregation. For example, $(row{:}1, 4 \leftarrow \{5, 6\}, Sum, 0)$ and $(row{:}1, 3 \leftarrow \{4, 5, 6, 7\}, Sum, 0)$ form a complete inclusion, because the aggregate and range elements of the former are included in the range of the latter. As all elements in a range should represent entities of the same semantic level, e.g., the average sales of all departments or the population of all states, any one of them cannot serve as an aggregate of any of its fellows.

**Mutual inclusion.** Two aggregation candidates are mutually included when the aggregate of each is included in the range of the other. For example, $(row{:}1, 4 \leftarrow \{5, 6\}, Sum, 0)$ and $(row{:}1, 5 \leftarrow \{3, 4\}, Sum, 0)$ are mutually inclusive. If one of them, say the former, is a true aggregation, then $c_{1,5}$ is part of the range elements that add up to $c_{1,4}$. Then, $c_{1,5}$ should not be the aggregate that sums up numbers including $c_{1,4}$. Two mutually included candidates cause circular calculations, which should not be correct semantically.

Given these pruning rules, in the example of Table 3.2, only the aggregations with the $1st$ and $4th$ column patterns are preserved. Six of the seven numeric rows, which is more than 0.7 given as the *cov* parameter, are compliant rows in both cases. Therefore,

all aggregations in these rows that have one of the patterns are included in the output of this phase.

Non-cumulative aggregation functions, according to Table 3.1, need no more iterations, because the detected aggregates cannot be used as ranges anymore (line 14 of Algorithm 3). For cumulative functions, the detected range columns are ignored in the next iteration (line 15), because they cannot be used as aggregates or ranges by any other aggregations that will be detected in the following iterations.

The individual aggregation detection phase outputs a set of row-wise and column-wise aggregation candidates for each aggregation function. The sets of results are consumed by the next phase that removes spurious candidates.

### 3.3.2 Collective aggregation detection

The previous phase identifies aggregations of individual functions with the three pruning rules. However, there might be a number of false positive results – mathematically correct but coincidental aggregations, which cannot be removed by the individual aggregation detection phase. Figure 3.6 demonstrates a fictitious example that has a true aggregation summing up the cost of heating, water, electricity, garbage disposal cost to the total cost. However, this table also includes a spurious average aggregation candidate, which calculates garbage disposal cost by taking the mean cost of the other three individual items. The coverage of this pattern is 3 of 4 (indicated by the orange shaded cells in the "Garbage disposal" column), surpassing the *cov* parameter. Therefore, the aggregation candidates with this pattern cannot be removed by the previous phase.

| | Total cost | Heating | Water | Electricity | Garbage disposal |
|---|---|---|---|---|---|
| Household A | 280 | 110 | 30 | 70 | 70 |
| Household B | 320 | 120 | 45 | 75 | 80 |
| Household C | 200 | 74 | 35 | 58 | 50 |
| Household D | 240 | 75 | 33 | 72 | 60 |

Figure 3.6: A fictitious example table with true sum aggregates (green solid fill) and spurious average aggregates (orange diagonal strip fill).

To remove the spurious candidates as such, we introduce the collective aggregation detection phase. By using pruning rules on the collection of results from all individual detectors of the previous stage, we refine these results: similar to the pruning step in the first phase, aggregation candidates with a particular pattern cannot co-exist with those of another particular pattern in the final results. However, in the first phase each detector refines the detected aggregations of its own function, whereas in this phase, AggreCol performs the refinement across all functions.

Specifically, the algorithm first groups the accepted aggregations from all individual detectors by their column patterns, similar to the process in the first phase, and ranks

the aggregation groups by two criteria: (i) the number of range elements in the group of a column pattern; (ii) the number of detected aggregations in a group. As the primary criterion, the more elements a range comprises, the higher the group's priority is, because we believe that an aggregation with fewer range elements is more likely to be a false positive. For the secondary criterion, we favor column patterns induced by a greater number of detected aggregations. For the case in Figure 3.6, the sum pattern has a higher score regarding both the above criteria than the average pattern, making the former a preferred choice by AggreCol.

Once AggreCol has ranked aggregation groups, it filters out groups contradicting the ones that have been validated, according to the complete inclusion and mutual inclusion rules suggested in Section 3.3.1. Besides that, AggreCol discards an aggregation group if the aggregate in its pattern is the same as that of a previously validated group, and the ranges of the two groups overlap: if a cell is the aggregate of a particular aggregation function, it should not act as the aggregate of another function while using (partly) the range of the former one. However, it is valid for a cell to serve as the aggregate for two aggregations with disjoint ranges. For example, the yearly net income of a company that equals the difference between the gross income and the expense can simultaneously be the sum of the net income of all quarters.

These rules are not applicable to the division function, because division aggregations can always be included in the final result. For example, given that $a_2$ in Figure 3.5 is validated, it is still reasonable to justify $a_4$, even though these two aggregations contradict each other according to the complete inclusion rule. This division records the percentage that a part (column 8) accounts for in the entirety (column 7) – a frequent kind of division usage in our observations.

### 3.3.3 Supplemental aggregation detection

While the collective aggregation detection phase can eliminate many spurious aggregations with the proposed pruning rules, difficult cases, such as interrupt aggregations, still cannot be retrieved. Take the interrupt aggregation shown in Figure 3.3c as an example: only the average aggregation can be retrieved by applying the first two phases. Because the sum detector in Phase 1 can identify only adjacent aggregations, it cannot recognize the interrupt sum aggregation in this case. We introduce the supplemental aggregation detection phase to detect such interrupt aggregations.

The general idea is to apply individual detectors proposed in Section 3.3.1 on a set of constructed verbose CSV files derived from the original file. Each of the new files is built by systematically removing certain aggregate cells that have been detected in the previous steps. As such "blocking" aggregates are removed from the file, some interrupt aggregations become detectable using the original individual detectors. Algorithm 4 describes the supplemental aggregation detection approach. It takes the original file $V$, the aggregation functions $F$, the detected aggregations $A$ from the previous phase, the error level threshold $e'$, and the aggregation row coverage threshold $cov$ as input, and outputs a set of detected aggregations.

First, individual detectors of all aggregation functions $F$ are pushed to a queue (line 2), so that each will be executed at least once. The algorithm constructs a set of files based on the original file $V$ by removing specific columns from $V$ (line 6). On

**Algorithm 4:** Supplemental aggregation detection (row-wise)

**Input:** Verbose CSV file $V$, aggregation functions $F$, detected aggregations $A$, error level $e'$, line aggregation coverage *cov*

**Output:** A set of row-wise supplemental aggregations D

```
1  detectors ← {d_f | f ∈ F};
2  q ← detectors // a queue to store detectors to be executed in the following;
3  D ← {};
4  while q ≠ ∅ do
5  |  d ← pop(q);
6  |  VS ← ConstructFiles(V, D);
7  |  res ← {};
8  |  foreach V' in VS do
9  |  |  res ← res ∪ d(V', e', cov);
10 |  end
11 |  if res ≠ ∅ then
12 |  |  D ← D ∪ res;
13 |  |  q ← {detectors \ d} ∪ q;
14 end
15 D ← Prune(D, cov);
16 return D
```

the one hand, all aggregate columns of detected non-cumulative aggregations should be excluded from the constructed files when detecting interrupt aggregations, because they cannot be used as range elements. On the other hand, each aggregate column of detected cumulative aggregations can be either excluded from or included in the constructed files, leading to multiple configurations to construct files. The algorithm constructs one file for each configuration and applies the individual detectors $d$ on all these files to recognize more aggregations (lines 7-10).

The approach runs sequentially: individual detectors are executed one after another. Once a detector discovers new aggregations, the queue reloads detectors of other aggregation functions that have already been processed (line 13). Newly detected aggregations may expose interrupt ones, leading to the necessity to re-check the other functions. Therefore, some detectors might be executed multiple times. The whole process terminates when no aggregation detector reports new aggregations. Finally, the algorithm applies the pruning rules used in the individual aggregation detection phase again to filter out spurious supplemental aggregations (line 15).

To summarize AggreCol, this approach employs a three-stage approach to detect aggregations, where the first stage recognizes simple and cumulative occurrences of individual aggregation functions. The second stage combines individual aggregation results into one collection and aims at removing spurious ones. To extend the ability of AggreCol on discovering interrupt aggregations, we apply the third stage that utilizes the individual detectors from the first phase on a set of reconstructed files.

## 3.4 Experimental Evaluation

This section includes the description of our evaluation datasets and the results of our in-depth qualitative evaluation of AggreCol including a comparison with a baseline and a detailed error analysis. Also, we present the improvement gain on cell classification after replacing Strudel's vanilla aggregate detection algorithm with AggreCol.

### 3.4.1 Datasets

We used real-world verbose CSV files from a variety of sources and constructed two datasets from them to evaluate our approach on detecting aggregations in verbose CSV files. Most tables in these files have fewer than 100 rows and 50 columns, whereas the longest and the widest tables contain 601 rows and 97 columns, respectively.

The first dataset combines files from `Troy` and `EUSES`. The `Troy` dataset [Nagy, 2010] with 200 verbose CSV files has been used in Chapter 2 to evaluate line and cell classification approaches. The `EUSES` dataset [Fisher and Rothermel, 2005] includes a collection of 1,352 spreadsheet files from diverse domains, such as data management, education, and finance. The dataset is widely used [Abraham and Erwig, 2007; Gonsior et al., 2020; Hofer et al., 2013] for spreadsheet cell classification, error detection, etc. We randomly sampled 200 files from this dataset and converted them to CSV format with the Apache POI library[4]. Dropping the files that could not be processed by the library left us with 185 verbose CSV files. As both datasets incorporate files from diverse domains, we merged them into a single dataset referred to as `Validation`, as we used it to validate the effectiveness of AggreCol while designing our approach.

The second dataset comprises verbose CSV files from three different datasets: the Statistical Abstract of the United States (`SAUS`), the Criminal In the US (`CIUS`), and the administrative spreadsheet files on an open data portal of the UK. These datasets include in total 3,053 files and have been used by related works in classifying lines or cells [Gol et al., 2019; Jiang et al., 2021]. We randomly sampled and annotated 100 files and created our `Unseen` dataset with those 81 files that contain aggregations. This dataset served purely as an unseen test set to assess the generalizability of our approach.

Naturally, verbose CSV files do not contain aggregation information. We use the "Aggregation Annotation" module in the annotation tool introduced in Section 2.4 to label our datasets. According to Definition 5, a valid aggregation annotation must include all the following three components: a single cell that acts as the aggregate, a set of other cells as the range, and the aggregation function. We have published the tool with detailed instruction to help researchers or data scientists annotate aggregations in their own datasets[5].

We annotated all aggregations regardless of their shapes in files: apart from the typical purely numeric aggregations, we observed that verbose CSV files may include aggregations that comprise non-numeric cells. A typical example is the usage of 'x' or '-' in cells to represent zero. Another unusual case represents the number '1.4' with the string '+1.4 Points'. Our annotations cover all cases as such, to better reflect the true

[4] `https://poi.apache.org/`

[5] `https://hpi.de/naumann/projects/data-preparation.html`

Table 3.3: Statistics of datasets.

| Observations | Dataset | |
|---|---|---|
| | `Validation` | `Unseen` |
| Number of files with | 385 | 81 |
| No aggregations | 50 | 0 |
| Aggregations of one type | 259 | 62 |
| Aggregations of two types | 71 | 17 |
| Aggregations of three types | 5 | 2 |
| Aggregations of all types | 0 | 0 |
| Number of aggregations | 20,280 | 5,854 |
| Sum | 14,070 | 4,399 |
| Average | 858 | 33 |
| Division | 4,800 | 1,097 |
| Relative change | 552 | 325 |
| Number of aggregations with | 20,280 | 5,854 |
| error = 0 | 14,479 | 4,020 |
| error > 0 | 5,801 | 1,834 |
| Min. per-file aggregation count | 1 | 1 |
| Max. per-file aggregation count | 1,651 | 490 |

aggregations in the datasets, whereas our AggreCol approach is not designed to handle numbers represented in such unusual ways, which we deem as an interesting extension. Note that aggregations may be subject to rounding errors. Even when an aggregate could not be precisely derived from its range, we labeled an aggregation based only on its semantics as indicated context, including file and column names, surrounding cell value, and our general understanding of the respective domain.

Table 3.3 displays some basic statistics of our datasets showing the high complexity of aggregations in data files. Around 20% of the files in both datasets include aggregations of more than one type. Sum is the most frequently used aggregation function, accounting for about 70% aggregations in both datasets. Aggregations in real-world data files often have errors – their aggregate does not precisely aggregate the range. This is the case for about 29% of all aggregations in our datasets.

### 3.4.2 Number format transformation

Numbers can be formatted in different ways: the decimal separator may vary depending on the cultural background of file authors, and a thousand separator can differ or be absent entirely. One file may use '12 345,67' and another '12,345.67' to represent the same underlying number '12345.67'. Cell values in verbose CSV files commonly preserve these formats of numbers. As for aggregations, an incorrect interpretation of the numbers in a file might cause incorrect calculation results. For instance, '1.000' may or may not be a correct aggregate for $700 + 300$ if the thousands separator character is treated as

Table 3.4: Overview of valid number formats and their respective occurrence in the 200 files from `Troy`.

| Digit group separator | Decimal separator | Example | Occurrences |
|---|---|---|---|
| Space | Comma | 12 345,67 | 24.5% |
| Space | Dot | 12 345.67 | 6% |
| Comma | Dot | 12,345.67 | 66.5% |
| None | Comma | 12345,67 | 1.5% |
| None | Dot | 12345.67 | 1.5% |

the decimal separator.

Before detecting aggregations in verbose CSV files, we first apply a pre-processing step to identify and normalize the number format. An investigation into the `Troy` dataset indicates five valid number formats, shown in Table 3.4. We propose a number format transformer that converts values of numeric cells in a file into a normalized format. A normalized format uses no thousands separator and the dot as the decimal separator. We created a regular expression for each valid number format. For each cell in the file, we tried to match it with every regular expression. As a consequence, numeric cells might be matched to one or multiple number formats, while non-numeric cells do not match any. We selected the number format that matches most cells in the file and performed the appropriate transformation. In case of ties, we chose the number format that has a higher occurrence ratio according to Table 3.4. For all files in our datasets, this process misjudges the number format of only eight files. These files all use 4-digit year strings in table headers, leading number patterns using no digit group separator to match more cells than the respective correct number patterns. With the incorrectly normalized numbers, AggreCol misses the true aggregations in these files.

### 3.4.3 Quality evaluation

We conducted a series of qualitative experiments to test the ability of AggreCol on recognizing aggregations with a variety of patterns in the datasets. First, we introduce the metrics used in our experiments. Then, we present the evaluation results on the `Validation` dataset at both aggregation- and file-level. After that, we investigate the generalizability of AggreCol by applying it to the `Unseen` dataset that remained unseen while designing the approach.

#### Metrics

According to Definition 5, two aggregations match only if their respective aggregates, ranges, and aggregation functions all match. We refer to a detected aggregation as *correct* if it matches some true aggregation in the ground truth, and *incorrect* if not. A true aggregation in the ground truth is *missed* if no detected aggregation matches it. With these interpretations, we apply the commonly used *precision* ($P$) and *recall* ($R$) metrics

to measure the detection quality of our approach:

$$P = \frac{|correct|}{|correct + incorrect|} \quad R = \frac{|correct|}{|correct + missed|} \tag{3.1}$$

Precision counts the number of correctly detected aggregations amongst all detected ones, while recall counts the number of correctly detected aggregations amongst all in the ground truth. The F1-score is the harmonic mean of these two measures. Note that precision is undefined when no result is returned by an approach, similarly for recall if a dataset contains no true aggregations. As usual, we set the score to 1 in both cases.

**Aggregation-level effectiveness**

The first experiment evaluates the effectiveness of AggreCol at the aggregation level, considering all aggregations from all verbose CSV files in a dataset equally, regardless of the files they belong to.

Three parameters have an impact on the performance of our approach: (i) the given error level parameter $e'$, which specifies how much calculation error does AggreCol tolerate; (ii) the line aggregation coverage $cov$, which indicates the minimum percentage of numeric cells in a row or column recognized as aggregates of aggregations with the same pattern; (iii) the window size used by the sliding window strategy, which specifies how many numeric cells in the proximity of an aggregate candidate should be checked.

Here, we fix the window size at 10 to cover the majority of the difference, division, and relative change aggregations, and set the coverage value $cov = 0.7$, because our experiment shows that the average F1-score across aggregation functions is highest with this value. Then, we select the best error level for each function with the results of individual detectors proposed in Section 3.3.1. Figure 3.7 illustrates the precision, recall, and F1-score our approach achieves for individual aggregation functions with the aforementioned parameter setting, under different error levels. Note that a difference aggregation can be trivially transformed to a sum aggregation by moving the minuend to the aggregate side. Therefore, we merge the ground truth of sum and difference, and the evaluations on them as well.

We observed a common trend of the change of F1-score against the increasing error level: it first increases as small increments of the error level allow more true positives to be retrieved and are still not large enough to include many spurious cases. As the error level becomes sufficiently large, e.g., 5% for division, the F1-score starts to decrease, because the algorithm incorporates many occasional false positive cases in the prediction and sharply decreases the precision.

While precision keeps decreasing with the rise of error level as expected, recall may not keep increasing with the error level, which we explain with the fictitious file in Figure 3.8. The left-most table shows four true aggregations with the pattern $F \leftarrow \{A, B, C, D, E\}$ based on the semantics of the table, although some of them are actually subject to an error. When setting the error level as zero, three column-wise aggregations are detected: two with the pattern $F \leftarrow \{A, B, C, D, E\}$ and the other with the pattern $D \leftarrow \{A, B, C\}$, shown in green in the middle table. The one with the latter pattern is filtered out as its pattern contradicts the one used by the other two, leading to a recall of 0.5. As a higher error level is used, six aggregations can be detected. Four have the pattern

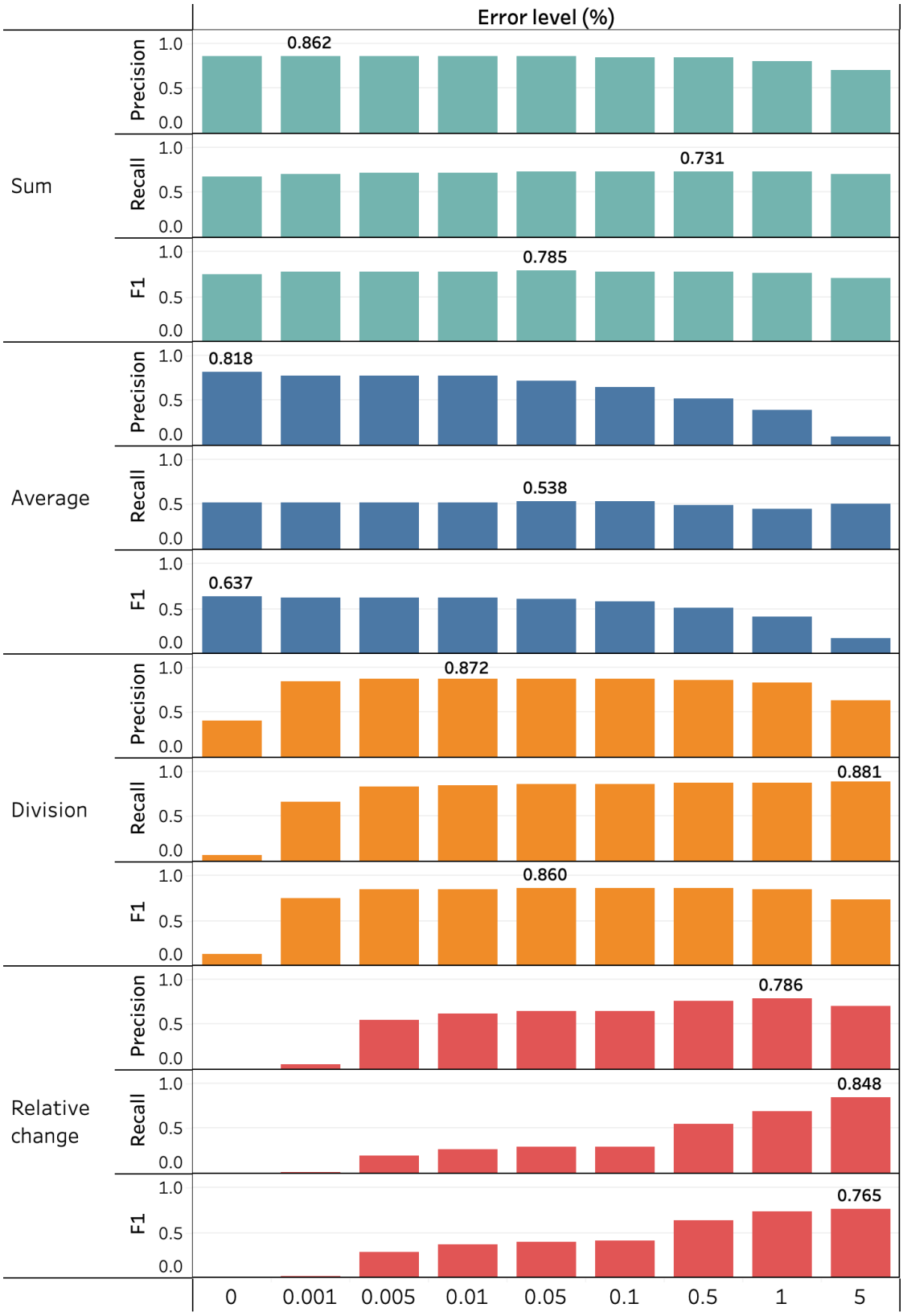



Figure 3.7: Per-function recall and F1-scores under different error levels. Line aggregation coverage is set as 0.7.

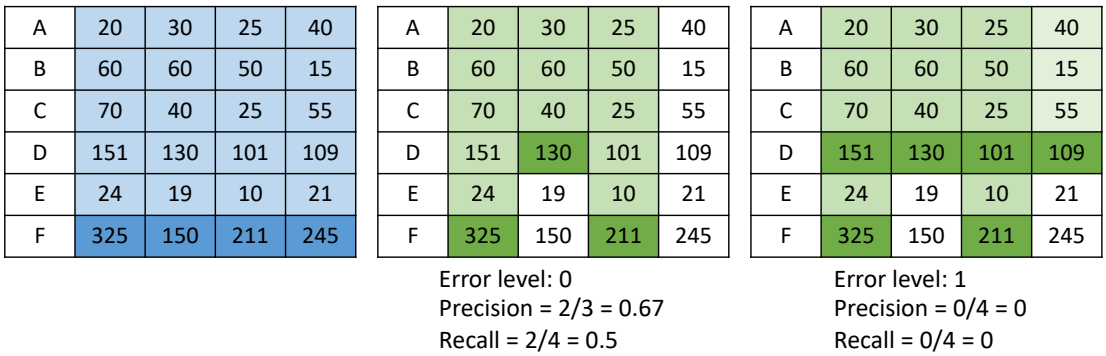



Figure 3.8: Example on how recall could decrease with increasing error level. The file on the left shows four aggregations. While the two files in the middle and on the right show the detected results under different error levels.

$D \leftarrow \{A, B, C\}$, while the other two have the pattern $F \leftarrow \{A, B, C, D, E\}$. AggreCol chooses the four candidates with the former pattern, because there are more candidates with this pattern, and therefore prunes the two predictions with the other pattern, leading no true positive cases to be retrieved. This usually happens under large error levels, and therefore many spurious candidates are generated.

Overall, the F1-score reaches its maximum at different error levels for different aggregation functions. We identify and select the respective optimal error levels for the aggregation functions for the following experiments.

Given the selected values for both per-function error level and line aggregation coverage parameters, we explored the effectiveness of different stages of AggreCol. Figure 3.9 demonstrates the precision, recall, and F1-scores for each aggregation function. The letters 'I' 'C', and 'S' in the x-axis represent the individual, collective, and supplemental aggregation detection stages, respectively. As each stage depends on the result of the previous stage, the metric score of a particular stage indicates the results obtained by applying AggreCol until this stage.

On the one hand, applying the collective aggregation detection stage increases precision across all aggregation functions, because this stage removes spurious candidates detected in the first stage with a set of pruning rules. Although the pruning could also drop some correct candidates, our experiment shows no or only a very minor drop of recall. On the other hand, adding a supplemental aggregation detection phase to the workflow achieves better recall, as expected. An investigation into the results indeed shows the detection of some interrupt aggregations. Overall, employing all phases yields a better F1-score across all aggregation functions.

### File-level effectiveness

Verbose CSV files may have different numbers of aggregations: the file with most aggregations contains 1,651 cases, while the one with fewest only one case in the `Validation` dataset. The aggregation-level results of the previous section may be biased in favor of long and wide files with many aggregations. To deal with this bias, we conducted an additional file-level evaluation on the `Validation` dataset. For each verbose CSV file, we

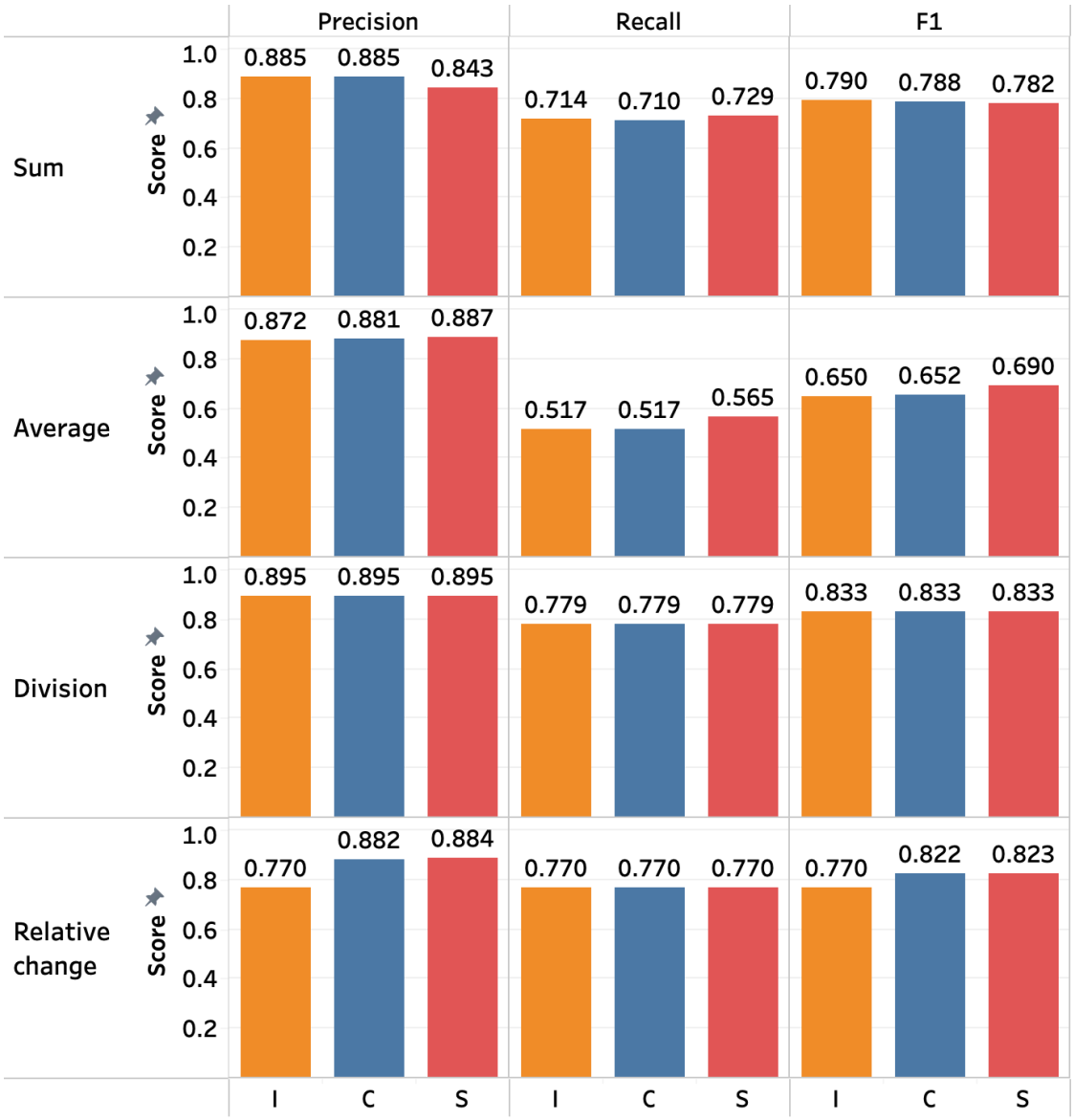



Figure 3.9: Precision, recall, and F1-scores for each aggregation function obtained by AggreCol at different stages. The individual, collective, and supplemental stages are indicated as "I", "C", and "S" in the x-axis.

compared the entire set of aggregations detected by our approach against the collection of real aggregations in it and determined which percentage of files meets certain minimum precision and recall values. Figure 3.10 presents the results, where the y-axis represents the percentage of the files on which AggreCol achieves the score in the range given by the x-axis. The score range between zero and one is divided into 20 bins, each spanning 0.05. As AggreCol achieves medium-range precision or recall (between 0.05 and 0.95) on very few files, we group the bins in this scope into three larger groups, each stretching over 0.3 on the score.

At file-level, AggreCol achieves greater than 0.95 precision and recall for more than

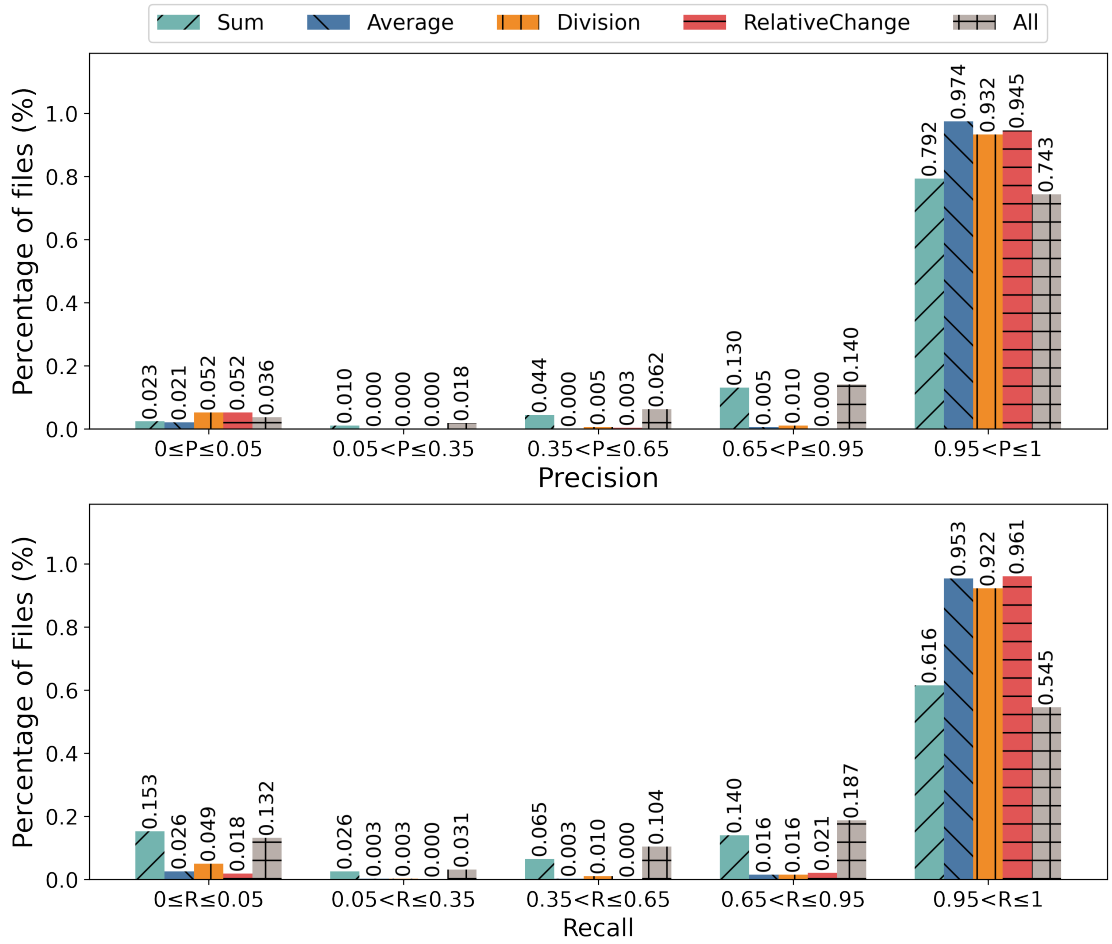



Figure 3.10: File-level precision and recall obtained by AggreCol on the `Validation` dataset.

90% of the files with regard to average, division, and relative change. The corresponding scores for aggregation-level results are all below 0.9, and even below 0.6 for the recall of average detection, indicating that most false negative and false positive aggregations are concentrated in few files. Compared to recognizing the above three aggregation functions, correctly detecting sum aggregations is more challenging – our approach achieves more than 0.95 precision and recall for 79.2% and 61.6% of the files, respectively. As around 70% of the aggregations in our ground truth are sum, each file includes on average more sum aggregations than other types, increasing the difficulty to discover all of them correctly. The grey bars demonstrate the overall precision and recall across all aggregation functions. For some cases, it is lower than the precision or recall of any single function, due to undefined precision or recall values that are adjusted to 1.

Overall, the file-level evaluation results indicate that false negative and false positive cases yielded by our approach tend to occur only in few files.

### Test on an unseen dataset

Due to ad-hoc shapes and forms of verbose CSV files, a set of 385 files can hardly cover all aggregation patterns. Therefore, we tested the generalizability of our approach with a second dataset after completing the algorithm design. Figure 3.11 demonstrates the file-level precision and recall achieved by AggreCol on this dataset.

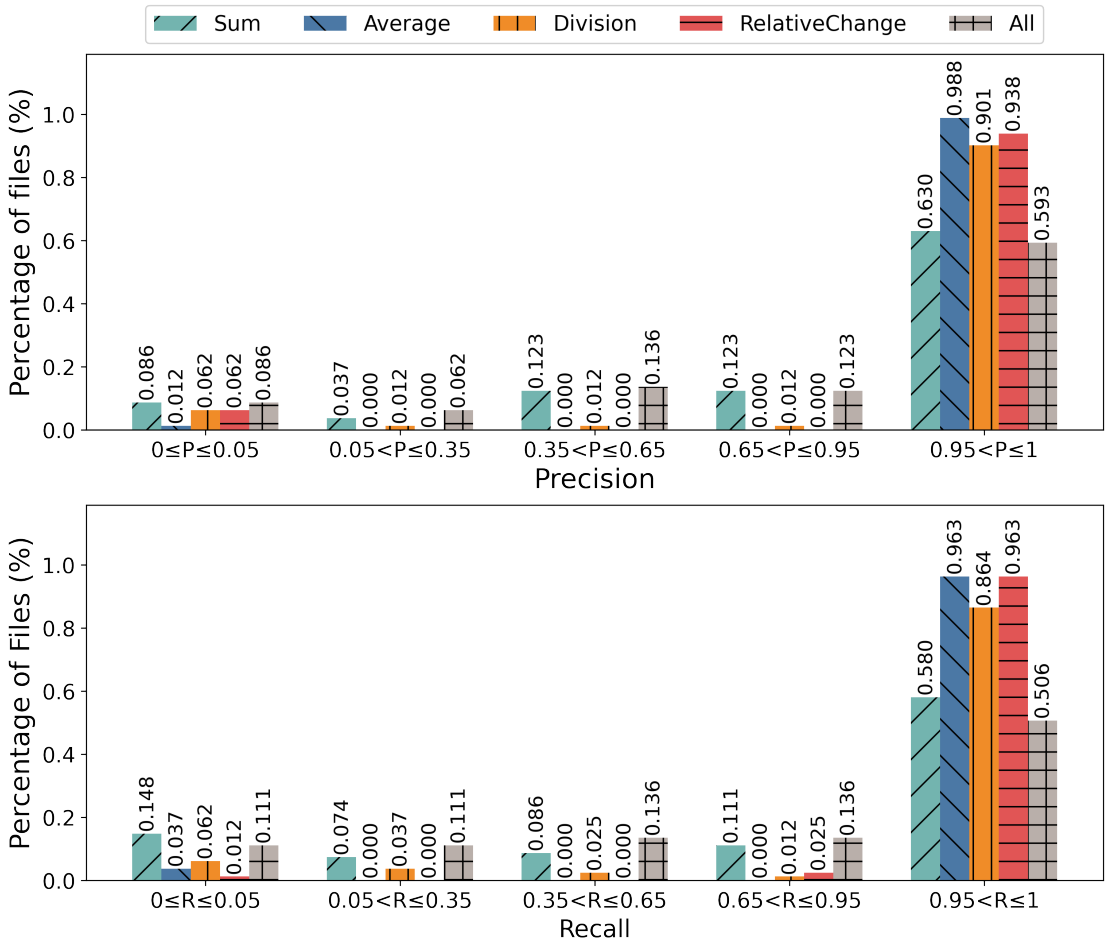



Figure 3.11: File-level precision and recall obtained by AggreCol on the `Unseen` dataset.

Similar to our approach's performance on the `Validation` dataset, precision and recall are higher than 0.95 for the majority of the files, and sum detection is the most challenging task on this dataset as well. The similar results on the two datasets indicate that our solution is not tailored for the `Validation` dataset, but, rather, can perform well on general verbose CSV files. A noticeable difference of sum detection precision in the results of the two datasets appears in the right-most group ($0.95 < P \leq 1$), where our approach obtains only 0.630 on the `Unseen` dataset. One reason for it is the prevalence of zero-valued cells in the many unseen files.

### 3.4.4 Comparison to baseline

There is no previous work specifically to detect and specify aggregations in verbose CSV files automatically. Therefore, we compared AggreCol with a baseline approach that eagerly checks all possible range element combinations: for each numeric cell in a verbose CSV file, the baseline traverses the permutations of all numeric cells in the same row or column, treating each permutation as a range candidate. Given a verbose CSV file with $n$ columns, the complexity of the traversal is in $\mathcal{O}(n*2^{n-1})$ for aggregation functions that involve at least two range elements, and $\mathcal{O}(n^3)$ for those having only two range elements. The computational burden is thus infeasible in general, and we set a 5-minute timeout for each file. Within that limit, this approach managed to process only 202, 203, 379, and 380 files regarding sum, average, division, and relative change detection, respectively, out of 385 verbose CSV files in the `Validation` dataset. The baseline was run on a Mac Pro 2019 using 4 cores. We also tested AggreCol on the same hardware with the same time limit. Our proposed approach is able to finish within 5 minutes per-file for 381 files, which cover all those that can be handled by the baseline. Therefore, only the baseline processable files are used for the following analysis. Note that even with a 20 minutes time limit, the baseline still cannot finish all files. In contrast, the longest time taken by AggreCol for any single file is about 599 seconds. Because the individual detectors of AggreCol process each aggregation candidate independently, they can be easily implemented in parallel to improve efficiency. Phase 3 costs on average 85% of the runtime in the entire workflow.

Not only is the baseline very time-consuming, but this approach also achieves poorer results than AggreCol. Figure 3.12 demonstrates the precision, recall, and F1-score achieved by AggreCol and the baseline approach. To obtain fair comparisons, we use

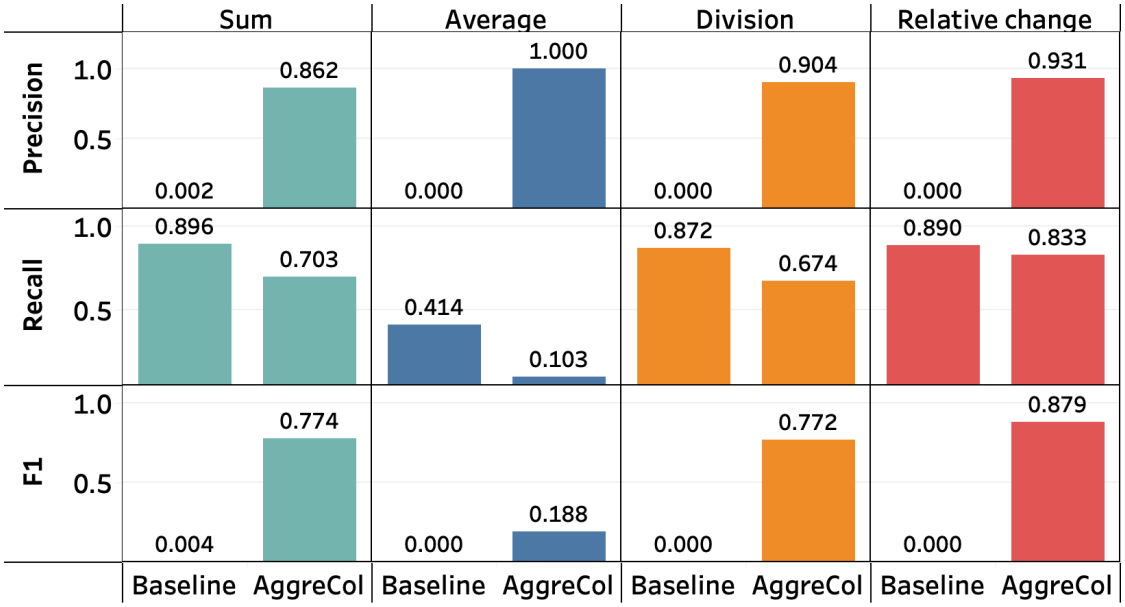



Figure 3.12: Precision, recall, and F-1 score comparison between the eager baseline approach and AggreCol. The baseline approach achieves very low precision and F-1 scores.

the same error level setting suggested in Section 3.4.3 for both. The baseline approach retrieves more correct aggregations than AggreCol, as expected, because for each aggregate candidate, it checks every possible range element permutation. However, this approach achieves this recall at the cost of almost zero precision, due to the huge amount of spurious detected aggregations in few files. For example, a file with many zeros or ones includes many false positive sum cases.

To further explore the comparison between the baseline and AggreCol on individual files, we present the file-level F-1 score obtained by the two approaches in Figure 3.13. For all functions, AggreCol achieves more than 0.95 on F-1 for more than 60% of the files, whereas the baseline obtains such an F-1 score for only up to 35% of the files. For the majority of the files, the baseline achieves less than 0.05 on F-1, which is caused mainly due to poor precision of the baseline. Overall, the eager baseline approach is not an ideal solution to obtain high F-1 scores.

Table headers in verbose CSV files could indicate the presence of aggregates in the same row or column. For example, if a header cell includes the word 'total', it might

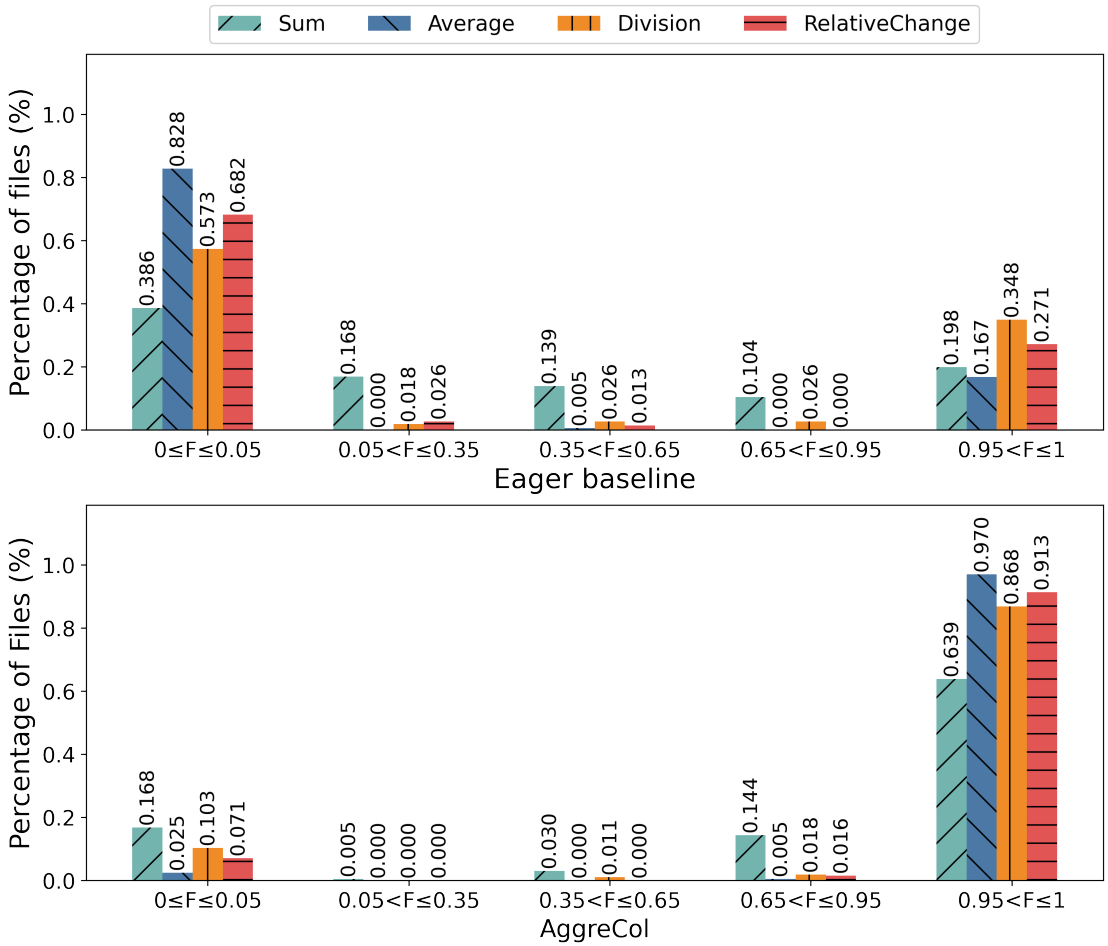



Figure 3.13: File-level F1-score comparison between the baseline approach and AggreCol.

imply that the numeric cells in its row or column might be the sum of several other numeric cells. However, our investigation into the `Validation` dataset shows that only 60.0% of the real sum aggregates use one of the keywords 'total', 'all', 'sum', 'subtotal', and 'overall' in their header. The corresponding ratios for average, division, and relative change are 86.6%, 53.1%, and 92.4%, with a unique keyword dictionary for the respective aggregation functions. What is more, these keywords are often used in rows or columns without aggregates. Our experiments show that the precision scores of detecting aggregates are 0.565, 0.256, 0.458, and 0.038 for sum, average, division, and relative change, respectively.

### 3.4.5 Analysis of detection errors

Depending on how the authors organize their data, aggregations in verbose CSV files have all sorts of patterns. Due to this diversity, recognizing aggregations in all possible situations is challenging. Here, we analyze the mistakes produced by AggreCol on the `dataset` and summarize the major causes that lead to false positive and false negative cases.

A general reason that causes both false positive and false negative cases is the selection of $e'$. As we used a fixed error level for each function determined by the aggregation-level F1-score in our experiment, it may not be resilient to different orders of magnitude of aggregate values. A difference between $r$ and $r'$ normalized by a smaller $r$ is larger than one normalized by a larger $r$ according to Definition 6. Therefore, the fixed error level might be either too large for big aggregate numbers or too small for small numbers. The former case yields incorrect results, while the latter causes missed cases.

#### False positive cases

A particular reason that causes many of the false sum predictions is the small number of different numbers in table cells. For example, when marking absence as '0' and presence as '1', a table for the student roster of a course might include only these two numbers. Therefore, it is very likely that many spurious sum candidates, such as '$1 = 0 + 1$' are generated. In our experiment, most mistakes involved many '0'-valued cells. We make similar observations for the division detection results that involve many '1'-valued cells.

Empty or almost empty lines and columns can be misinterpreted as contributing the number zero to the range. We observe such effects in only very few cases and a corresponding preprocessing might lead to more false negatives.

#### False negative cases

We have recognized five major reasons that cause false negative cases. First, AggreCol cannot recognize aggregations upon incorrectly formatted numbers, which involves the cells in only eight of 466 files in our experiments. Second, ranges of some aggregations in the ground truth are both of the interrupt kind and not in the proximity of their aggregates. Although the supplemental aggregation detection phase retrieves some interrupt aggregations, it cannot deal with cases whose ranges are not interrupted by aggregates of other detected aggregations. For example, consider a table that has columns for trade

volume, import volume, and export volume of a country with other countries in a whole year and for every month. The aggregation that sums up the per-month import volume to the yearly one cannot be detected, because the per-month export volume columns, which are not detected aggregates, cannot be removed when constructing files in the third phase. The third reason is the inflexibility of a fixed window size, which cannot cover the whole ground truth. Additionally, we observed that very few pairs of true aggregations break the 'directional agreement' rule proposed in Section 3.3.1. However, our approach could retrieve these cases by dismissing this rule at the cost of introducing many false positives into the results. Finally, ranges whose last cells are '0'-valued could be missed when detecting sum, because our adjacency list strategy stops when encountering zeros. If every sum with the same pattern has the same number of tailing zero cells, an aggregation with fewer range elements is detected, rather than the true one.

We tested our approach with different error levels, yet none of them can completely eliminate detection errors caused by the selection of this parameter value. We did not use particular techniques tailored to address the aforementioned special cases, as they appear in only few files. Resolving them could lead to many more detection errors overall.

### 3.4.6 Cell classification improvement

Cell classification aims at classifying each cell in a CSV file as one of several pre-defined semantic types, among which "aggregation" is a common one [Gol et al., 2019; Jiang et al., 2021]. AggreCol can contribute to the improvement of overall cell classification performance, as we show for one example: our supervised-learning based cell classification approach Strudel uses a binary feature to represent whether a cell is an aggregate (sum or average) [Jiang et al., 2021]. Recall from Chapter 2 that Strudel uses an approach similar to the adjacency list strategy of AggreCol on detecting aggregates in CSV files. We replaced the values of that feature with the results of AggreCol on the `SAUS` and `Troy` datasets[6] used there and conducted the same cross-validation experiments described in that work. Table 3.5 shows the per-type F1-score of the original Strudel algorithm and the version using AggreCol's results.

Table 3.5: Per-type F1-score of the cell classification approach Strudel by using its original aggregation cell detection approach ($Strudel^O$) or AggreCol ($Strudel^A$).

| | SAUS | | Troy | |
|---|---|---|---|---|
| Cell type | $Strudel^O$ | $Strudel^A$ | $Strudel^O$ | $Strudel^A$ |
| metadata | 0.987 | **0.989** | 0.975 | **0.976** |
| header | 0.976 | **0.978** | 0.959 | **0.963** |
| group | 0.731 | **0.806** | 0.638 | **0.740** |
| data | 0.967 | **0.976** | 0.950 | **0.962** |
| aggregate | 0.677 | **0.786** | 0.615 | **0.740** |
| notes | **0.957** | 0.956 | **0.968** | 0.966 |

For both datasets, the F1-score of the aggregation type increases significantly, as ex-

[6]Annotations have been revised due to some label errors.

pected. When using AggreCol's output in the feature of Strudel, more cells previously classified as one of the other types are predicted correctly as aggregation, increasing the precision score, and also the F1-score, of most other types.

## 3.5 Related work

Two previous efforts have been made to directly address aggregation detection in plain-text files such as verbose CSV files. Long et al. introduced a keyword-based approach to recognize only summation in tables of plain-text files [Long, 2010]. They suggested using two categories of keywords: direct aggregation keywords, such as 'total' and 'sum', and complementary keyword pairs in two rows or columns, such as 'student' and 'non-student'. Our experiment has shown that purely using the first type of keywords is not effective in detecting aggregations. The complementary keyword pairs used in this work require external knowledge about the semantics of content in files, which are not available in our input data.

Our Strudel approach incorporates an aggregate detection algorithm to determine if a cell is the sum or average of numeric cells adjacent to it [Jiang et al., 2021]. The result was then encoded in a binary feature to detect classes of lines and cells in verbose CSV files. This approach concerned only adjacent aggregations depicted in Figure 3.3, missing any cumulative and interrupt cases. While both works exploited keywords, these information are not always reliable in locating aggregations. A limited number of keywords is not general enough to cover all cases. Meanwhile, having keywords in headers does not always imply the existence of aggregations.

After we finished this project, we realized a very recent work that addressed the formula prediction problem on spreadsheets with a supervised learning approach [Chen et al., 2021]. To predict the formula of a cell, this approach considers (i) the context within a bounded range, i.e., the closest 10 rows and columns to the cell in both directions; (ii) the header row that is always the first row in the file. A BERT encoder [Devlin et al., 2018] is used to represent the values in the context, and an LSTM decoder [Hochreiter and Schmidhuber, 1997] is used to generate the sketch (aggregation function) and the range (range) in the prediction. Unlike our AggreCol approach, this work does not take the arithmetic aspect into account: it does not encode the relationship between an aggregate and its corresponding range via calculation.

Despite the lack of previous work on aggregation detection, there is related work on downstream tasks that require aggregation information as input. In the rest of this section, we discuss related work on these use cases.

### 3.5.1 Structure detection

Information in data files, such as spreadsheets and verbose CSV files, are often organized in an ad-hoc manner: authors treat files like a canvas and drop information at arbitrary positions therein, which makes it difficult to process data in these files automatically by a machine. Before we can extract information from these files, it is important to understand their overall structure (not just aggregations) by recognizing the types of content in different regions.

Identifying types of lines or cells in data files is a typical way to tackle the structure detection task. Various previous works have been proposed for this purpose [Adelfio and Samet, 2013; Christodoulakis et al., 2020; Gol et al., 2019; Jiang et al., 2021; Koci et al., 2016]. Common content types include data, header, metadata, aggregation, and so on. Distinguishing aggregation content from data content is usually challenging, as the content of both types are often numbers that are similar to each other. An advanced aggregation detection approach helps locate lines or cells of this type more precisely by resolving the above challenge and therefore can improve the structure detection performance. Detected structures in data files can help extract and transform information from data files [Barowy et al., 2015; Chen and Cafarella, 2013, 2014].

### 3.5.2 Formula smell detection and repair

Spreadsheet formulas often contain "smells". The term formula smell is inspired by source code smells that indicate violations of programming principles, such as mysterious variable names and overly large classes [Fowler, 2018]. Such smells may cause poor code readability and error-prone code refactoring. A formula smell in a spreadsheet indicates a wide variety of poor usage of the formula functions, such as inconsistent formulas in a single row or column, missing formulas, and complex formulas that are caused by sloppy cell copy-and-paste, abuse of formula functions, etc. The existence of such smells may significantly increase the difficulty of spreadsheet comprehension [Hermans et al., 2012].

Many research efforts have been dedicated to detecting and repairing smelly formulas [Barowy et al., 2018; Dou et al., 2014, 2018; Hermans et al., 2012; Jansen and Hermans, 2015]. However, they all assume the existence of some formulas in the files. Unfortunately, verbose CSV files do not preserve such information, and not every spreadsheet keeps formulas in its cells. Aggregation detection approaches can supply the necessary input to formula smell detection and repair tasks.

## 3.6 Conclusion and Future Work

Aggregations represent arithmetic relationships between a set of numbers (range) and a single number (aggregate), and are common not only in spreadsheets but also in tables of verbose CSV files. A variety of applications depend on the existence of aggregations. Identifying aggregations helps understand the structure of tables and provides insights on how raw data can be extracted from such files. In addition, data errors might be cleaned with the knowledge of the true aggregations.

We formalize the problem of recognizing aggregations in verbose CSV files and recognize three patterns of aggregations in data files. We propose the three-stage approach AggreCol to address this problem. Our approach can detect aggregations of five types in verbose CSV files: sum, difference, average, division, and relative change. To evaluate the performance of our approach, we annotated 466 real-world verbose CSV files and conducted a series of qualitative experiments to show the effectiveness of our approach. For the validation dataset encompassing 385 data files, AggreCol achieves on average 0.795 F1-score in the aggregation-level evaluation, and more than 0.8 precision and recall for 93.5% and 88.7% of the files, respectively. A test on an unseen

test data shows similar performance. Besides that, we also compared AggreCol with a brute-force baseline approach that retrieved eagerly too many spurious aggregations. Our study reveals that real-world verbose CSV files incorporate surprisingly many errors in aggregations. Besides that, we also compared AggreCol with a brute-force baseline approach that retrieved eagerly too many spurious aggregations.

AggreCol does have limitations. In this work, we address aggregations whose aggregate and range are either in the same row or in the same column, which might not always be the case. For example, we have observed average and relative change aggregations whose range includes cells from both the same row and the same column of the aggregate. An extension of AggreCol should relax the same-line aggregation assumption. Another observation we made is that some aggregations involve more than one aggregation function in the calculation. For example, students' final scores for a course may be weighted by the importance of different modules (attendance, homework, exams, etc.). In this case, both sum and division calculations are involved. Therefore, another future work could support multi-functional aggregation detection.

# Chapter 4

# Holistic Primary Key and Foreign Key Detection

*Primary keys* (a.k.a. *keys*) and *foreign keys* are two of the most important constraints for relational databases, indicating the entity integrity and referential integrity that databases need to follow. Both constraints are ubiquitously used in databases. While in principle, primary keys (PKs) and foreign keys (FKs) should be explicitly assigned for tables by database designers, in practice, these constraints are either incomplete or missing, making it difficult to understand the structure of the database schemata.

The reasons for the lack of primary keys and foreign keys are manifold. One major reason is to avoid performance decline. If the primary key and foreign keys are defined on a relational table, applying any data manipulation operation, such as insertion, deletion, and modification, leads the database system to update the underlying key index, and re-validate the referential integrity for the foreign keys, which might severely impair query performance. As a consequence, a common practice is to control entity integrity and referential integrity in the application layer, instead of defining primary and foreign keys in the database system. Enabling to load legacy data into relational databases is another reason to exclude primary keys and foreign keys. Legacy data might include duplicated records that violate entity integrity, or erroneous records that violate referential integrity. As a compromise, data practitioners need to drop the primary key and foreign key constraints to allow data loading. Additionally, many relational tables are dumped in plain-text files, such as CSV files, which are suitable for exchanging data across systems. However, all constraint definitions may be lost when migrating databases, as plain-text files usually do not carry metadata. Finally, many data are stored in verbose CSV file tables, each of which may have its unique structure. By using the techniques outlined in Section 2.5, we can extract relational tuples from these tables. However, relational tables constructed from these tuples usually include no metadata.

This chapter introduces our proposed HoPF (Holistic Primary Key and Foreign Key Detection) approach that automatically discovers both primary keys and foreign keys in the tables of a database in a holistic manner, which is based on our publication [Jiang and Naumann, 2020]. HoPF takes as input the sets of a database's unique column combinations (UCCs) and inclusion dependencies (INDs), which are the superset of the true primary and foreign keys. The algorithm ranks all UCCs and INDs by the score

calculated from a set of proposed PK and FK features and leverages several pruning rules to remove poor PK/FK candidates. Detected primary keys are used to reduce the search space of foreign keys, whose detection, in turn, guides the improvement of primary key results. In this way, HoPF addresses the detection of primary keys and foreign keys as a whole. The algorithm outputs a refined set of UCCs and INDs as the recognized PKs and FKs. The resulting sets include on average 88% and 91% of the true PKs and FKs across the tested datasets.

The contributions of this chapter are the following:

(i) We present HoPF – the first algorithm to detect primary keys and foreign keys in relational tables holistically, in particular removing the assumption that primary keys are known and present. We propose advanced pruning rules to filter out spurious unique column combinations and inclusion dependencies.

(ii) We conducted a series of experiments on five different datasets to show the effectiveness of our holistic algorithm. Foreign keys are also generated without assigning primary keys to show that the absence of primary keys worsens the performance of foreign key detection.

(iii) We compared HoPF with the state-of-the-art methods for both primary key and foreign key detection, and showed the performance of our approach is on par with or better than the other ones for most cases.

The rest of this chapter is organized as follows: In Section 4.1 we discuss the various applications of primary keys and foreign keys first, followed by the introduction of the concepts INDs and UCCs used as input of our algorithm. After that, we formalize the primary key and foreign key detection problem. Related works are introduced in Section 4.2. Section 4.3 lists the features used for the primary key and foreign key detection algorithm, and Section 4.4 describes pruning strategies for primary and foreign key candidates. The overall holistic algorithm is explained in Section 4.5. Experimental results are shown and compared to related works in Section 4.6. Finally, we conclude this work in Section 4.7.

## 4.1 Structuring Schemata

Knowledge of primary keys and foreign keys is essential for various applications, such as *data cleansing*, *reverse engineering*, *query optimization*, and *data integration*. Although for databases with simple schemata, missing keys and foreign keys can still be manually labeled by domain experts, it could be excessively time-consuming or even infeasible to do so for databases with complex schemata, e.g., databases with hundreds of tables that have hundreds of foreign keys. Several research efforts have been made for foreign key detection in relational tables [Rostin et al., 2009; Zhang et al., 2010], which assume the presence of primary keys. Normally, primary keys exist for tables stored in RDBMS platforms, because these tools explicitly request users to specify a primary key for each table. However, this does not always apply to databases compiled from plain-text files or web sources, because such constraint information would need to be stored in a separate file. In our experience, such associated information is often not present. The lack of primary keys motivates us to detect primary keys as well and design an approach to solve the two inter-dependent problems in a holistic fashion.

Primary keys and foreign keys, which in general can cover multiple attributes, are essentially the particular cases of *Unique Column Combinations* (UCCs) and *Inclusion dependencies* (INDs). Before formalizing the primary key and foreign key discovery problem, we briefly introduce these two concepts based on the notations introduced in [Papenbrock, 2017]. Let us denote a relational schema as $\mathbb{R}$, which may contain multiple relations. A relation, denoted as $R$, is a named non-empty set of attributes. An attribute of a relation is denoted as $A \in R$ with its domain $dom(A)$. The cross-product of the domains of all attributes $\bigcup_{A \in R} dom(A)$ forms the data domain. An instance $r$ of the relation (a.k.a. a table) $R$ is a set of tuples following the schema thereof. For each attribute $A$, every tuple $t$ in the instance takes the value $t[A] \in dom(A)$. For a subset $X$ of the attributes in $R$, we use $t[X]$ to denote the projection on $X$ for $t$. and $r[X]$ the projection on $X$ for all tuples in the instance of the relation.

### 4.1.1 Types of dependencies

There are many different dependencies to describe the characteristics of a table or relationships between tables. We use only unique column combinations and inclusion dependencies, because they are the supersets of primary keys and foreign keys, respectively. We give the definitions of these dependencies and their relationships to primary and foreign keys in the following.

**Definition 1** (Unique Column Combination)**.** *Given the instance $r$ of a relation $R$, a* unique column combination *(UCC) is a set of attributes $X \subseteq R$ whose projection contains only unique, non-null value entries on $r$, i.e., $\forall t_i, t_j \in r, i \neq j : t_i[X] \neq t_j[X]$ and $\forall t \in R, A \in X : t[A] \neq \perp$.*

A unique column combination is *minimal* if none of its not-null subsets is a valid UCC. In turn, all not-null supersets of a UCC are also valid UCCs. A UCC may contain one or multiple columns, as primary keys can be composed of one or more attributes. In real-world cases, database designers tend to define primary keys with few attributes. A study on the SAP ERP system of a Fortune 500 company, which includes 23,886 relational tables, reported that all tables have a primary key with no more than 16 attributes. Among the primary keys, more than 95% have fewer than six attributes [Faust et al., 2014]. Although a single instance of a database system cannot cover the entire database universe, we believe these valuable observations can be acquired on a considerable amount of relational database instances.

In principle, each relation $R$ should have one and only one primary key, while in practice, a primary key might not be definable for some tables. For example, no primary key can be defined on a table with duplicated records. In this work, we assume that each table is duplicate-free, and therefore at least one UCC exists.

Each table in an instance of a schema $\mathbb{R}$ may contain a number of UCCs. We denote the set of UCCs in a table as $U$. Only at most one UCC in $U$ is the true primary key. Picking the true primary key from $U$ of each table in the schema, we can constitute a list of UCCs representing the primary keys of $\mathbb{R}$ and denote it with $PK_{\mathbb{R}}$. Note that we refer to the schema $\mathbb{R}$ itself, instead of any schema instance, when describing the primary keys. This is because the primary key of every relation is instance-independent – it is determined by the schema.

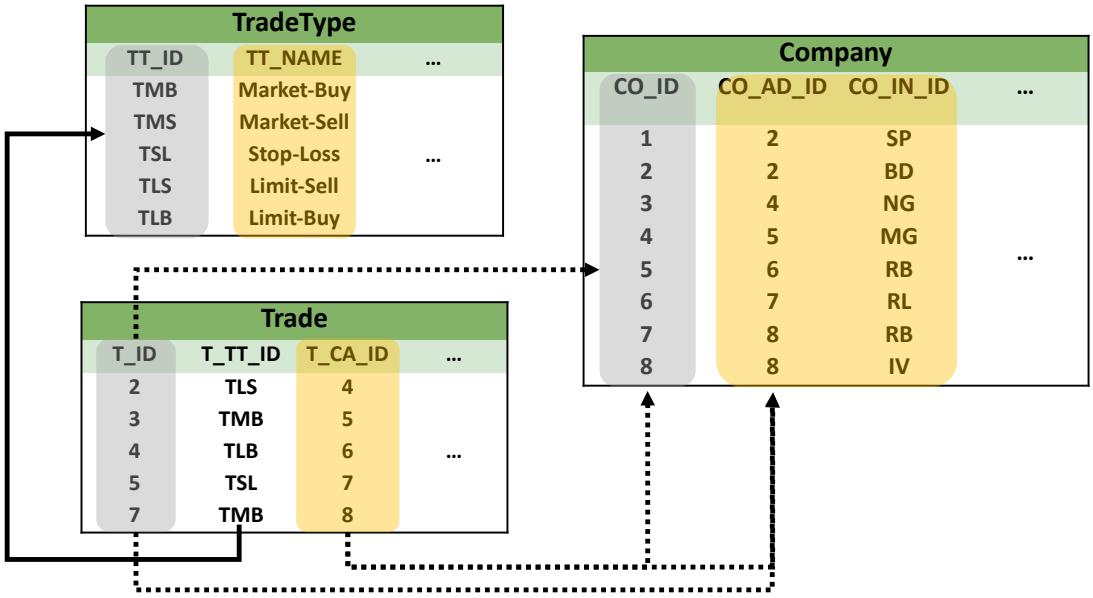



Figure 4.1: A subset of tables of the `TPC-E` schema. For simplicity, some columns are omitted. True primary keys are indicated by unframed grey boxes, whereas spurious UCCs by framed grey boxes. The true foreign key is indicated by the solid arrow, while spurious INDs by dashed arrows.

**Definition 2** (Inclusion Dependency)**.** *Given the instances $r_i$, $r_j$ of two relations $R_i$, $R_j \in \mathbb{R}$, an* inclusion dependency *(IND), denoted as $r_i[X] \subseteq r_j[Y]$ with $X \subseteq R_i$ and $Y \subseteq R_j$, states that all the value entries in the column combination $X$ in $r_i$ are also contained in the column combination $Y$ in $r_j$, i.e., $\forall t_m \in r_i, \exists t_n \in r_j : t_m[X] = t_n[Y]$.*

We call the dependent part $r_i[X]$ the *left-hand side* (LHS) and the referenced part $r_j[Y]$ the *right-hand side* (RHS). The notation of the defined IND can be expressed as $X \subseteq Y$ for short. Note that the definition of IND implies $|X| = |Y|$. When $|X| = |Y| = 1$, we call the IND *unary*, otherwise *n-ary*. We denote the set of INDs in an instance of the schema $\mathbb{R}$ as $I$. A foreign key must be an IND, because by definition, each value appearing in the LHS of a foreign key must be included in the value set of its RHS. Similar to the primary keys, foreign keys are also independent of the instances of the schema. We denote the set of true foreign keys of $\mathbb{R}$ as $FK_{\mathbb{R}}$.

Both UCC and IND detection tasks are difficult due to the combinatorial explosion of column combinations. Fortunately, many algorithms have been proposed in previous works [Abedjan et al., 2015] to discover these two dependencies, making them well accessible. However, those algorithms commonly generate huge amounts of UCCs and INDs – many more than the numbers of real primary keys and foreign keys. Therefore, the tasks of primary key and foreign key detection are to distinguish true primary keys and foreign keys from spurious UCCs and INDs, respectively. We present a motivating example of three relational table excerpts in Figure 4.1 to show the difficulty of primary key and foreign key detection.

The potentially very many unique column combinations for one table are all valid

candidates for the primary key of that table. For example, both the two shown columns {T_ID} and {T_CA_ID} in the Trade table in Figure 4.1 are unique column combinations. While the combination of the two columns {T_ID,T_CA_ID} is also a unique column combination, we regard only minimal ones for simplicity (details discussed in Section 4.1.2). According to the schema documentation for this table, {T_ID} is the true primary key, whereas the other one is a spurious candidate. In real-world cases, there might be many more spurious candidates, making it challenging to recognize only the true primary keys.

Similarly, we can expect many spurious inclusion dependencies besides true foreign keys. The IND Trade.T_TT_ID $\subseteq$ TradeType.TT_ID in Figure 4.1, for example, is a true foreign key, while the IND Trade.T_ID $\subseteq$ Company.CO_ID becomes a foreign key candidate only because the containment is satisfied. As is shown in Figure 4.1, there may be much more spurious INDs than true foreign keys.

In other cases that are not shown in the figure, the values of two columns might be included in one another. For example, in the TPC-E benchmark database, the column {T_D} in the table *Trade* and the column {SE_T_ID} in the table *Settlement* are the primary keys for the respective tables, and the two columns happen to contain the same set of values. As a consequence, both INDs Trade.T_ID $\subseteq$ Settlement.SE_T_ID and Settlement.SE_T_ID $\subseteq$ Trade.T_ID are valid foreign key candidates, whereas only the latter one is a true foreign key.

By examining data, intuitive rules can be found to distinguish keys from spurious UCCs, which makes the automatic detection of keys feasible. The selection of foreign keys is also constrained to the predicted primary keys. For example, a foreign key must reference a primary key according to its definition. Therefore, INDs that do not reference detected primary keys should be excluded from the predicted foreign key set. In Section 4.4, we describe the proposed spurious candidate pruning techniques in detail.

### 4.1.2 Problem statement

With all the terms defined above, we formalize the problem of *detecting primary keys and foreign keys in relational tables* as follows: *given a database instance with schema* $\mathbb{R}$*, the sets of the minimal unique column combinations* $U$ *and the inclusion dependencies* $I$ *of this instance, find the set of primary keys* $PK_{\mathbb{R}}$ *and the set of foreign keys* $FK_{\mathbb{R}}$*, where* $PK_{\mathbb{R}} \subseteq U$ *and* $FK_{\mathbb{R}} \subseteq I$*.*

The dependency between primary key and foreign key is obvious, because the right-hand side of a foreign key must be a primary key. Although each table may have multiple alternative keys, only one of them is the true PK. In this work, we use only minimal unique column combinations for two reasons: (i) the proposed PK features (see details in Section 4.3) always prefer the minimal UCCs than their non-minimal supersets; (ii) the complete set of UCCs (including both minimal and non-minimal UCCs) could contain exponentially more UCCs than only the minimal UCC set, because each superset of a minimal unique column combination is also a valid UCC. However, we are aware of a disadvantage of this choice: when searching for primary keys on tables where any subset of the real primary key is also a UCC, the true primary keys will not be detected. We discuss it further in Section 4.6.

Note that real-world data might be erroneous. Therefore, exact UCC and IND de-

tection algorithms may discover a subset of the true cases and also introduce spurious results. Approximate detection algorithms address this issue and discover UCCs and INDs that might contain violations [Caruccio et al., 2015; Kruse and Naumann, 2018; Kruse et al., 2017]. In this work, we consider only exact UCCs and INDs as input and assume the datasets do not contain errors.

## 4.2 Related Work

The proposed HoPF algorithm requires unique column combinations (UCCs) and inclusion dependencies (INDs) as its input. We first briefly summarize the latest techniques on unique column combination and inclusion dependency detection, even though they are not the focus of this chapter. After that, we present a discussion of previous research efforts on primary and foreign key discovery.

### 4.2.1 Metadata discovery

Efficiently discovering all UCCs and INDs for a given set of tables is a challenge due to the combinatorial explosion of column combinations. The number of unique column combinations candidates and inclusion dependencies grows exponentially with the number of attributes. Given a table with $n$ attributes, there might be $2^n - 1$ unique column combinations. Up to $\binom{|n|}{|n|/2}$ column combinations can be minimal UCCs [Abedjan and Naumann, 2011]. On the other hand, the search space for n-ary inclusion dependency candidates is $2^n * n!$ [Liu et al., 2012]. The complexity of both detecting all minimal UCCs and detecting all maximal n-ary INDs is NP-complete [Kantola et al., 1992; Lucchesi and Osborn, 1978]. Fortunately, there are already quite a few algorithms designed to discover unique column combinations and inclusion dependencies [Abedjan et al., 2015], which are efficient in practice. However, the amount of detected UCCs or INDs can be formidable. It is not unusual to discover hundreds or even thousands of UCCs and INDs even in tables with only tens of columns. To enable primary key and foreign key detection with UCCs and INDs, the challenge is to recognize the spurious candidates from such large candidate sets.

### 4.2.2 Primary key and foreign key discovery

Surprisingly, not many efforts have been made for primary key detection. A set of simple heuristic features was proposed to differentiate true primary keys from spurious unique column combinations for the purpose of decomposing a relation into Boyce-Codd normal form [Papenbrock and Naumann, 2017]. The authors calculate and add up scores for their features for each UCC. Lacking a human expert to view the ranked results, the UCC with the top score for a table is assumed to be its primary key.

In contrast to the lack of previous efforts on primary key detection, there are some works dedicated to discovering foreign keys [Chen et al., 2014; Lopes et al., 2002; Marchi et al., 2009; Memari et al., 2015; Rostin et al., 2009; Zhang et al., 2010]. We briefly introduce three representative works among them. Similar to the aforementioned primary key discovery idea, intuitive features can also be applied to foreign key detection. Rostin

et al. proposed ten features for machine learning methods to automatically detect foreign keys from various datasets [Rostin et al., 2009]. However, the method detects only single-column foreign keys. The authors list several situations in which the classifier makes mistakes, including transitive foreign keys and one-to-one relationships. A transitive foreign key represents the situation, where a primary key referenced by a foreign key is also a foreign key. One-to-one relationships are paraphrased as "PK $\subseteq$ PK" in this paper and explained in Section 4.4.2. Overall, this machine learning-based method is not able to address these two situations.

Assuming that the data of a foreign key should well represent a sample from the key column it references, a state-of-art method introduces the *randomness* metric to discover both single-column and multi-column foreign keys [Zhang et al., 2010]. The authors used the *earth-mover distance* (EMD) to measure the data distribution similarity between LHS and RHS of foreign key candidates: it is the minimum cost of transforming one distribution into the other by moving counts of values among buckets within a distribution [Rubner et al., 1998]. The data distributions are created by choosing a fixed, same number of buckets for both LHS and RHS, and counting the number of corresponding values for each bucket. The closer the data distributions are, the smaller the distance is. It then ranks all foreign key candidates with regard to their earth-mover distance and reports the performance based on the top X% of the result. This approach is reported to outperform the one described above [Rostin et al., 2009] on both precision and recall. Therefore, we compare HoPF only with this randomness-based approach, referred to as Randomness in the following sections.

Chen et al. proposed to combine heuristic features with different pruning rules to detect foreign keys, which is most similar to our algorithm [Chen et al., 2014]. However, they assume that each table pair can hold only one foreign key, which is a too strong restriction for real-world scenarios. We refer to this approach as FastFK and compare it with our HoPF approach.

We note that *all* the aforementioned approaches assume that primary keys are already known and base their heuristics on them. We drop this assumption of prior knowledge and propose an approach that is better suited for many real-world scenarios. Nevertheless, we compare our approach with the last two of the described works on foreign key discovery, and our experiments show that HoPF tops these approaches in both precision and recall in most cases.

## 4.3 Features for Primary Key and Foreign Key Discovery

In this section, we introduce the features used in our algorithm to score and differentiate true primary keys and foreign keys from the respective spurious UCCs and INDs. Previous works have already proposed some useful features to identify either primary keys [Papenbrock and Naumann, 2017] or foreign keys [Rostin et al., 2009; Zhang et al., 2010]. We adopt some of them as well as suggest new features to score each primary key or foreign key candidate.

### 4.3.1 Primary key features

Several useful heuristic features have been explored to distinguish true primary keys from spurious UCCs [Papenbrock and Naumann, 2017], including the cardinality of a UCC, average value length, and positions of its attributes in the schema. They have been proven by the authors to be effective in discovering primary keys for their application of normalizing relation table schemata. We reuse the score functions for these three features in our task. Based on our observation, in many cases, headers of primary key columns follow a different pattern than that of spurious candidates. Therefore, we introduce an additional name-based feature. We elaborate on the features in the following.

**Cardinality.** The cardinality of a UCC stands for the number of attributes therein. Because primary keys are often used as foreign keys that reference other tables, executing join operation across tables that involve primary keys is a common practice. For maintainability purposes, it is common to include few attributes in primary keys. Therefore, we prefer primary keys with smaller cardinality. We define the cardinality score as $\frac{1}{|X|}$, where $X$ is the column set of the UCC. That is to say, the more columns are involved in a UCC, the lower its chance is to be a true primary key.

**Value length.** When performing join operation across tables, values in the joined attributes are compared with each other. The efficiency of value comparison can be highly dependent on the type of values. As comparing strings to one another is cumbersome, a good practice is to join on attributes with short, integer values, such as automatically incremented integers. Therefore, we expect columns used as primary keys to have short value length. The score function used for this feature is $\frac{1}{\max(1,|\max(X)|-l)}$, where $|\max(X)|$ is the average length of the longest values of each column in $X$, and $l$ represents the parameter to penalize long values. It can be adjusted according to the datasets. In our experiments we choose $l = 8$, a reasonable choice for one of the following primary key types: (i) commonly used surrogate keys that are usually compact data type, such as 4-digits or 8-digits integers; (ii) auto-incremental integer keys that use at most eight digits in tables with less than 100 million records; (iii) date string keys that require up to eight characters – four for the year part, and two for the month and the day parts. In our datasets, the primary key length of the longest table is eight. Although our choice cannot cover all real-world primary key cases, we believe it can be useful for a great number of tables. For multi-column UCCs, we calculate the score for each column and use the average as its overall score.

**Position.** In principle, attributes are unordered in a relation, while in practice, there is an implicit order of attribute positions when defining the schema. For single-column primary keys, we observed that in most cases, primary keys appear in the left-most position of the column list. A similar observation was made for web tables [Venetis et al., 2011]. In addition, for multi-column primary keys, we expect no (or very few) non-key columns in between the key columns, apart from that all keys should be close to the left-most position in the column list. The column position score is calculated as $\frac{1}{2}(\frac{1}{|\mathit{left}(X)|+1} + \frac{1}{|\mathit{between}(X)|+1})$, where $\mathit{left}(X)$ and $\mathit{between}(X)$ represent the number of columns left of the first column of $X$ and the number of non-key columns between the first and last columns of $X$, respectively.

**Name suffix.** Database designers often highlight primary key columns with particular

suffix in the column name. We notice in all datasets used for experiments that primary key columns are often indicated by their column suffix name, e.g., "id" and "key". Here we choose "id", "key", "nr", and "no" as our suffix set. Clearly, this list can be extended, for instance, to include foreign language schemata. For a given UCC, we apply the score function as $\frac{|suffix(X)|}{|X|}$, while $|suffix(X)|$ counts the number of columns in the UCC whose name contains either suffix mentioned above. The count is normalized by the cardinality of the UCC.

We denote the unweighted average of the above feature scores as the primary key candidate scores. Note that many relational database systems allow users to export tables without column headers to plain-text files. Therefore, column headers of tables stored in plain-text files are often missing. To test the effectiveness of our features without knowing header names, we also perform experiments with this feature turned off.

### 4.3.2 Foreign key features

Rostin et al. proposed ten heuristic features to discover foreign keys among INDs [Rostin et al., 2009]. Zhang et al. suggested a *randomness* feature for data value distribution that subsumes a variety of those features except column names [Zhang et al., 2010]. However, the computation of this feature is very time-costly. We describe in the following our improved version, namely the *data distribution* feature. Column name similarity also plays an influential role in discovering foreign keys, which is not covered by the data distribution features. However, Rostin et al. only checked the exact match or complete containment of each pair of column names. We propose an improved *column name* feature with a fuzzy matching approach [Chaudhuri et al., 2003]. Overall, we employ two features for foreign key discovery, which are column name and data distribution. We elaborate on these two features in the following paragraphs.

**Column Name.** In many cases, for the purpose of better schema understanding and database maintenance, database designers tend to avoid giving arbitrary names to related tables or columns. An inclusion dependency `R.A` $\subseteq$ `S.B` is likely to represent a true foreign key if the names of the table $R$ and the column combination $A$ are similar to those of $S$ and $B$, respectively. Therefore, we present a column name feature that leverages the string similarity between the LHS and the RHS of INDs. We tested several string similarity functions and found the fuzzy similarity function, which has been initially proposed to solve the fuzzy matching between records [Chaudhuri et al., 2003], to be most suitable for our task. To calculate similarity using this metric, each string in the labels of the components $R$, $A$, $S$, and $B$ is tokenized first by delimiters. A valid delimiter can be either "_" or "-". In the case where a label string has no delimiter characters but is presented in camel case, we split the string at the positions before the uppercase letters. After the tokens for each component in a foreign key candidate is generated, we combine the token sets of table name and column name for both LHS and RHS. For example, given an IND `Trade.T_S_SYMB` $\subseteq$ `LastTrade.LT_S_SYMB`, the merged token set of the LHS and the RHS are $T_{LHS}$ = {Trade,T, S, SYMB} and $T_{RHS}$ = {Last, Trade, LT, S, SYMB}, respectively. To calculate the similarity between the LHS and the RHS, we map each token in $T_{RHS}$ to the most similar unmapped token from $T_{LHS}$. The similarity between two tokens is computed by the Levenshtein distance [Levenshtein et al., 1966]. Overall, the similarity of an inclusion dependency is calculated according to the following

formula [Chaudhuri et al., 2003]:

$$sim(LHS, RHS) = \frac{\sum_{j=1}^{n}(sim_j \times ln(\frac{1}{f_j}))}{\sum_{j=1}^{n} ln\frac{1}{f_j}}$$

where $sim_j$ is the token similarity between the $j$th token in $T_{RHS}$ and the most similar one to it in $T_{LHS}$. Note that if $T_{LHS}$ contains fewer tokens than $T_{RHS}$, some tokens in $T_{RHS}$ cannot find a map. For each such token $j$, we set the $sim_j$ as zero. $f_j$ represents the frequency of the $j$th token in the collection of all table and column names. $ln(\frac{1}{f_j})$ is the weight of the token in $T_{RHS}$, which measures the rarity of the token. We prefer rare tokens to common ones. Intuitively, a rare token is more useful to indicate the similarity between two strings, therefore should receive a higher weight. The similarity is asymmetric, which means that, given two INDs `R.A` $\subseteq$ `S.B` and `S.B` $\subseteq$ `R.A`, the similarity of the former is different from that of the latter. This is useful to recognize the true foreign key from two INDs whose column-set values are included in one another and thus their data distribution scores are identical.

**Data Distribution.** In many cases, values in the LHS of a true foreign key approximately follow an even sampling of the values in the RHS. This approximation becomes more precise when the number of values in the LHS increases. For example, values of the course_id column in the table that deposits student course registration information reference the values of the course_id column in the table that stores the information of all courses. As the number of course registrations increases, each course tends to receive a similar amount of registrations. Based on the aforementioned uniform sampling assumption, Zhang et al. proposed to measure the similarity between the LHS and the RHS of an IND with the data distribution similarity of the two sides, by using the earth-mover distance [Zhang et al., 2010]. When selecting true foreign keys from all IND candidates, they prefer those with high data distribution similarity to the ones with low.

The data distribution of the participating columns is a good indicator to distinguish foreign keys from spurious INDs. We verify the effectiveness of this assumption while observing the time overhead to construct this measure is large. Therefore, we propose a simpler yet effective histogram difference to represent the cost: Given an IND $r_i[X] \subseteq r_j[Y]$, we create a set of buckets according to the value range of each column $Y_i$ in $r_j[Y]$ and put each value into the corresponding bucket. We choose twenty as the number of buckets by default, which is a good choice we experimentally show in Section 4.6.2. The buckets form a histogram, which is denoted as $Hist(Y_i)$, and for each column $X_i$ in $X$ we place its values into the buckets created for $Y_i$. The overall data distribution score is the average of the histogram difference score in each dimension. While there are some other alternatives for histogram difference calculation, such as the $L_1$ norm and histogram intersection, we use the Bhattacharyya coefficient [Bhattacharyya, 1946] – a known good solution to measure the similarity, thus the difference in turn, between two histograms.

## 4.4 Pruning PK and FK candidates

Enumerating and scoring all combinations of UCCs and INDs in their corresponding search space is the most naive approach to discovering PKs and FKs. However, this

approach is excessively time-consuming due to the potentially exponential number of valid candidates. Consider a schema with only 20 tables: even if each table contains only two UCCs and the whole schema instance includes only 50 INDs, there are $2^{20}$ primary key candidate combinations and $2^{50}$ candidate combinations of foreign keys in the search space. And in practice, tables contain many more UCCs and typical databases contain thousands of INDs, leading the naive traversal approach impractical. Therefore, advanced pruning techniques are needed to reduce the search space.

Note that foreign keys are defined to reference a primary key. Therefore, once the primary key is chosen for a table, all the foreign key candidates that reference other UCCs in the table can be filtered out. This motivates us to combine the discovery of primary keys and foreign keys in a two-step approach. In particular, a combination of UCCs is selected from the search space, each of which is considered as the predicted primary key for the corresponding table. Thereafter, we determine the foreign keys from the INDs whose RHS are among these primary keys. The PK's score is summed up with the FK's score for this selection. After enumerating all combinations of UCCs and their corresponding INDs, the selection of UCCs and INDs with the highest overall score is passed to the primary key reduction process, which we describe in Section 4.5, to improve primary key detection results with discovered foreign keys.

In order to avoid checking every UCC and IND, we suggest filtering techniques for the detection of both primary keys and foreign keys to help the algorithm skip unnecessary checks. These techniques include foreign key candidate prefiltering, $\texttt{PK} \subseteq \texttt{PK}$ filtering, primary key candidate pruning, and foreign key candidate pruning. We elaborate on each of them in the rest of this section.

### 4.4.1 FK candidate prefiltering

The number of INDs typically grows quadratically with the number of tables and columns [Tschirschnitz et al., 2017]. However, not all valid INDs are good candidates for foreign keys. The prefiltering step, which leverages the following two rules, aims at keeping only a small suitable portion of the INDs as foreign key candidates.

**RHS Uniqueness.** By definition, a primary key contains neither duplicate values nor Null-values. As a foreign key must reference a primary key, the RHS of the foreign key is a UCC and does not contain Null-values. Therefore, we prune all INDs whose RHS is not a UCC, and thus not a primary key candidate.

**Non-Null Column Combinations.** We observed that columns with only *null* values exist, especially in real-world datasets. In principle, these columns can be seen as included in any other column. However, for the purpose of deriving foreign keys, they are not useful and thus we ignore them.

Table 4.1 displays the number of INDs before and after prefiltering on each used dataset. We introduce the datasets in more detail in Section 4.6.1. Applying the two rules reduces the number of INDs to be considered in the following steps.

Table 4.1: INDs before and after FK candidate prefiltering.

| Dataset | Before | After |
|---|---|---|
| TPC-H | 90 | 33 |
| TPC-E | 511 | 175 |
| Adwork | 19,602 | 2,047 |
| SCOP | 6,450 | 2,062 |
| MusicBrainz | 236,151 | 28,722 |

### 4.4.2 PK ⊆ PK Filtering

Auto-incremented integers are often used in unary primary keys for tables. When using such an approach, primary key values of a table often subsume those of another, when the former table contains fewer tuples than the latter, which shapes a valid IND whose LHS and RHS are the primary key columns of the former and the latter tables, respectively. We denote this type of INDs as PK ⊆ PK. In the case where two tables have similar numbers of tuples, the PK ⊆ PK IND seems a good foreign key candidate, because most values in the LHS are covered in the value set of the RHS, and the occurrences of the covered values are close to one another.

However, PK ⊆ PK inclusion dependencies are usually not true foreign keys. To prune out such candidates, we propose a *PK ⊆ PK filtering*: given an IND A ⊆ B, we discard it if the ordered values in $A$ form a consecutive subsequence of the ordered values of $B$ – in such cases it is more likely that $A$ is a key in its own right, and its inclusion in $B$ is spurious.

In the TPC-H database, for example, all tables have such an auto-incremented integer primary key. The table REGION contains only five tuples, whose primary key values are $1 \ldots 5$, whereas the NATION table contains 24 tuples, whose primary key values are $1 \ldots 24$. Therefore, REGION.REGIONKEY ⊆ NATION.NATIONKEY is a valid IND. However, such a PK ⊆ PK IND is meaningless and thus should be filtered out, because the set of ordered values in the LHS is a consecutive subsequence of that in the RHS.

The FastFK approach [Chen et al., 2014] does not take such PK ⊆ PK candidates into account on foreign key discovery. Motivated by the fact that many tables contain auto-incremental integer primary keys, the Randomness approach [Zhang et al., 2010] suggests a consecutive prefix or suffix check between LHS and RHS as a naive approach to detect foreign keys. Our HoPF algorithm relaxes this restriction by considering any consecutive subsequence.

### 4.4.3 Primary key candidate pruning

Each table may possess multiple UCCs. To obtain the optimal result, we need to consider all valid UCC combinations, i.e., the Cartesian product of the UCCs by the tables they belong to. In practice, the search space is still quite large, even after we restrict only one out of all the UCCs for each table to be considered as the primary key. For example, we can see from Table 4.2 that for the TPC-H schema, there are more than 76 million different $PK_{\mathbb{R}}$ candidates. We denote a primary key combination candidate as *PKcc*.

Traversing the whole PKcc search space is impractical. To reduce the number of candidates that need to check, we first group the UCCs by the table they belong to, and then rank them within each group in a descending order of their scores calculated based on the primary key features mentioned in Section 4.3.1, with the goal of pruning poor candidates that obtain low scores.

Figure 4.2 shows the score for each primary key candidate of the table `Trade` in the `TPC-E` schema, as an example. The x-axis represents the rank of candidates with regard to their scores, while the y-axis shows the primary key score for each candidate. The score for the top-ranked candidate, which is the true primary key for this table, is much higher than those of all other candidates.

Inspired by this observation, we propose a metric *cliff* to help select the subset of a table's all UCCs that is likely to include the true primary key of the table. To calculate the cliff of a set of UCC candidates, let us denote $S_i$ as the score for the top-$i$th candidate. For each pair of neighboring candidates $S_i$ and $S_{i+1}$, the *score-difference* equals $SD_i = S_i - S_{i+1}$. In analogy to the notion of a *knee* [Terrell, 1913] for continuous curves, we define a cliff for our discrete case:

**Definition 3** (Cliff)**.** *Given the sorted score list of the primary key candidates of a table* $S = \{S_1, S_2, ..., S_n\}$ *and the corresponding score-difference list* $SD = \{SD_1, SD_2, ..., SD_{n-1}\}$*, the cliff of the table is the pair of neighboring candidates* $S_i$ *and* $S_{i+1}$ *with the largest SD score.*

The candidates of each table can be separated into two subgroups at the position of the cliff, namely *Upper*, which includes all the candidates before the cliff $(S_1, \ldots, S_i)$ including $S_i$, and *Lower*, which includes the remaining candidates $(S_{i+1}, \ldots, S_n)$, including $S_{i+1}$. For example, the cliff of the `Trade` table is the pair of the top two primary key candidates due to the largest *SD* score between them. In this case, the upper part contains only the top candidate, whereas the lower part contains the remaining 15 ones.

We calculated the Upper and the Lower for each table in all datasets to see into which part the true primary keys fall. Table 4.2 shows the result. For example, 31 out of 32 primary keys in the `TPC-E` database fall into the Upper part, while only 1 of 32 primary keys fall into the Lower part of the candidates for that table. The statistics in Table 4.2 support this observation also for the other datasets. To reduce the search space of $PK_{\mathbb{R}}$, we simply drop the Lower part from contention for the primary key, which means the primary key of only one table in `TPC-E` gets lost. This shrinks the primary key search space drastically. For instance, the search space for the `TPC-E` database is reduced from $9.03 \times 10^{13}$ candidates to only 768, as indicated by the numbers of primary key combination candidates before (# PKcc before) and after (# PKcc after) pruning in Table 4.2. Reducing the primary key search space with the cliff metric loses very few primary keys for only the TPC-E and MusicBrainz databases. We verified the effectiveness of filtering out poor candidates with the cliff metric, by using which there is no need to include a parameter to control the number of filtered out candidates.

### 4.4.4 Foreign key candidate pruning

Inclusion dependency discovery algorithms, which produce the foreign key candidates, regard only value containment between columns. However, treating every IND as a

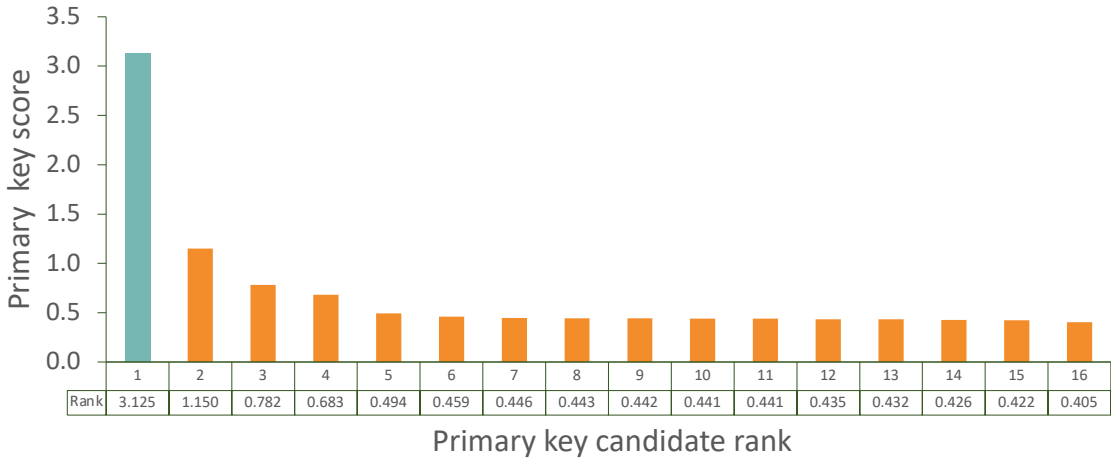



Figure 4.2: Scores of primary key candidates of tables `Trade` in the `TPC-E` database. The candidates are ranked in descending order of their score. The green and orange shaded regions are the Upper and Lower, respectively. The *cliff* appears between the first and second candidates.

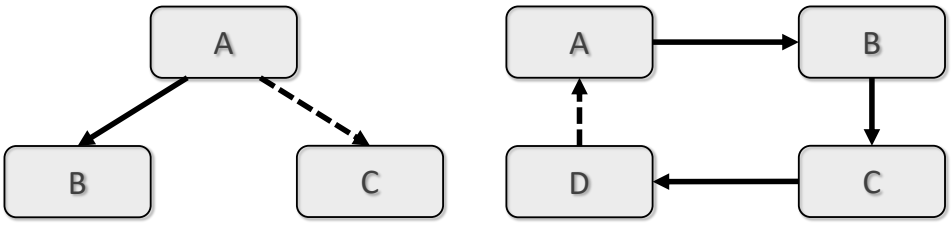
Table 4.2: Evaluation datasets with the number of primary key candidate combinations *before* and *after* pruning Lower. "# PKs U" and "# PKs L" indicate the number of true primary keys falling into Upper or Lower. Thus, "# PKs L" is the difference of the number of tables and "# PKs U".

| Dataset | # Tables | # PKs U | # PKs L | # PKcc before | # PKcc after |
|---|---|---|---|---|---|
| TPC-H | 8 | 8 | 0 | $7.67 \times 10^7$ | 1 |
| TPC-E | 32 | 31 | 1 | $9.03 \times 10^{13}$ | 768 |
| AdWork | 27 | 27 | 0 | $6.23 \times 10^{21}$ | 256 |
| SCOP | 42 | 42 | 0 | $7.25 \times 10^{11}$ | 2 |
| MusicBrainz | 124 | 122 | 2 | $3.73 \times 10^{25}$ | 576 |

foreign key may yield a huge amount of spurious results. To reduce the size of the foreign key candidate set, specific pruning techniques are needed. We propose two candidate selection rules that leverage the mutual exclusive nature amongst foreign key candidates, i.e., *uniqueness of foreign keys* and *non-cyclic reference*. Figure 4.3 visualizes the rules.

**Uniqueness of foreign keys.** Each foreign key can reference only one primary key in a schema, while the values of the LHS of an IND might be contained in multiple different RHSs. If one of these INDs is a true foreign key (shown as the solid arrow in Figure 4.3a), it is clear that all others are spurious (shown as the dashed arrow).

**Non-cyclic reference.** Figure 4.3b demonstrates a situation where a set of INDs forms a cyclic reference. If a cyclic reference exists in the schema, all the involved column combinations contain the same values, which we consider as not semantically meaningful. Therefore, we do not predict a foreign key that causes a cyclic reference in the schema graph. The dashed arrow in Figure 4.3b introduces a cyclic reference and should not be predicted as a foreign key if the other three solid arrows are already




(a) Only one referenced RHS (b) Non-cyclic reference.

Figure 4.3: Dashed INDs are invalid foreign keys.

classified as foreign keys. In our datasets, it is more common that two IND candidates form a cyclic reference, because the underlying tables have the same number of records.

To remove foreign key candidates that contradict the ones already predicted as foreign keys, we first ranked the foreign key candidates in descending order according to their foreign key scores calculated by the features introduced in Section 4.3.2. An empty prediction set is created to keep all predicted foreign keys. Then, we iteratively moved the top-ranked candidate in the ranked list to the prediction set and removed the ones in the ranked list, which contradict this predicted foreign key according to the above rules. We observed that there may exist still many valid foreign key candidates with low scores after we applied the above two rules. To avoid suggesting too many false foreign keys, our approach stops the iteration as long as the whole schema is connected. Consider the schema as an undirected graph, where each node is a table in the schema and each edge represents a predicted foreign key between the two connected nodes, the schema is connected when every node is reachable from every other node. Before the graph is fully connected, there might be multiple predicted foreign keys between a single pair of tables. In this case, we only keep one edge between the nodes of these tables. The connectivity of all the tables is a good indicator that the majority of true foreign keys have been found, as shown by the recall in Figure 4.5. To this end, HoPF checks whether the schema connectivity has been fulfilled each time a new foreign key is predicted.

## 4.5 Holistic algorithm HoPF

Figure 4.4 presents the overall workflow of our proposed algorithm HoPF. Given the UCCs of individual tables, INDs across the whole database, and basic statistics of each column, it first refines the set of UCCs and INDs, decreasing the number of primary key and foreign key candidates needed to be processed in the next step. For all surviving candidates, HoPF enumerates all primary key candidate sets and their corresponding INDs, scoring each of them with regard to the features proposed in Section 4.3 and selecting the proposed true primary keys and foreign keys. The special $\texttt{PK} \subseteq \texttt{PK}$ candidates are removed during this procedure. Conflict checking is integrated into the process of foreign key detection to remove further spurious INDs. Before producing the final result, a primary key reduction step simplifies the predicted primary keys and amends the corresponding foreign keys. The used candidate pruning techniques have been described in Section 4.4.

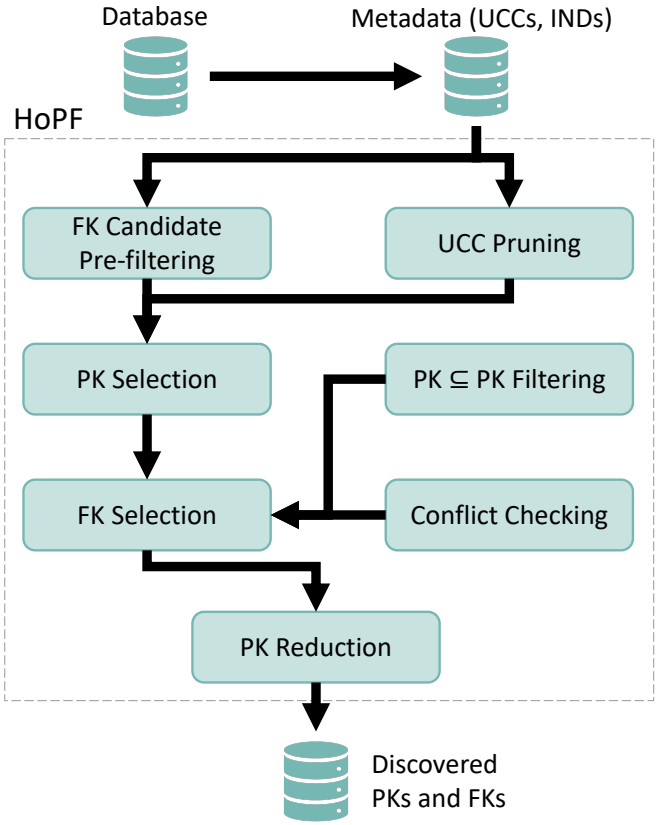



Figure 4.4: Overview of the HoPF algorithm.

The general algorithm to holistically determine primary keys and foreign keys from UCCs and INDs is displayed in Algorithm 5, which in turn calls various methods that are shown in the following separately. In line 5.2 (line 2 of Algorithm 5), the search space of primary key combinations is generated and saved in $PKcc$ (Primary Key combination candidates). The loop from line 5.3 to line 5.15 calculates the scores of these primary key combinations along with their corresponding foreign keys, determining the one with the highest score as the predicted primary key set and foreign key set. In line 5.4 only the foreign key candidates that reference a predicted primary key are retained in $FKCands$. We restrict the minimal number of FKs to be $|PKc| - 1$ in line 5.5, otherwise it is certain to break the rule of schema connectivity. In line 5.6 the algorithm searches for the INDs to form the foreign keys given the detected primary key combination $PKc$. Here, $D$ represents the discarded inclusion dependencies that are used to restore some foreign keys later in line 5.7. The lines from 5.8 to 5.13 simply select the primary keys and their corresponding foreign keys with the highest overall score as the final prediction.

Algorithm 6 displays the procedure of finding all suitable primary key combination candidates. It first groups all UCCs by the tables they belong to, and filters out poor primary key candidates according to the cliff technique described in Section 4.4.3. To compose a primary key combination, only one candidate from each table's leftover UCC set is selected.

Given the primary key combination, the procedure of generating detected foreign

**Algorithm 5:** Holistic Primary Key and Foreign Key Detection (HoPF)

**Input:** UCC set $U$, IND set $I$, schema $\mathbb{R}$
**Output:** Predicted primary keys $P$, Predicted foreign keys $F$

```
1  P ← {}, F ← {}, Q ← 0;
2  PKcc ← GetPKCombinations(U, ℝ);
3  foreach PKc in PKcc do
4  |  FKCands ← I \ PruneCands(I, PKc);
5  |  if |FKCands| ≥ |PKc| − 1 then
6  |  |  (FKCands, D) ← GetFKCandidates(FKCands);
7  |  |  (PKc, FKs) ← PKReduce(PKc, FKCands, D);
8  |  |  score ← score_PKc + score_FKs;
9  |  |  if Q < score then
10 |  |  |  P ← PKc;
11 |  |  |  F ← FKs;
12 |  |  |  Q ← score;
13 |  |  end
14 |  end
15 end
16 return P, F
```

**Algorithm 6:** GetPKCombinations()

**Input:** UCC set $U$, schema $\mathbb{R}$
**Output:** primary key combinations $PKcc$

```
1  PKcc ← {};
2  foreach R_i in ℝ do
3  |  g_i ← {u | u ∈ U ∧ R_u = R_i};
4  |  CliffPrune(g_i);
5  end
6  foreach PKc ∈ g_1 × ... × g_|R| do
7  |  PKcc ← PKcc ∪ {PKc};
8  end
9  return PKcc
```

keys is shown in Algorithm 7. All input foreign key candidates are first ranked by their score in line 7.2. In line 7.3 we initialize a graph to represent the tables of a schema and the predicted foreign keys among them. Each node represents a single table. An edge is added between the two corresponding nodes if there is a foreign key connecting two tables. From line 7.4 to line 7.18, qualified INDs are greedily added to the graph as predicted foreign keys from top to bottom with regard to their scores. Each time a new foreign key is predicted, the algorithm checks whether the schema connectivity is met and stops if the graph is connected. The non-cyclic reference requirement is checked in line 7.6, followed by the $\texttt{PK} \subseteq \texttt{PK}$ check. Candidates failing the $\texttt{PK} \subseteq \texttt{PK}$ check are added into the discard set $D$ and used in the primary key reduction step. Once a foreign key is predicted, all other candidates that share the same LHS are excluded from the existing

**Algorithm 7:** GetFKCCandidates()

```
   Input: Inclusion dependencies I
   Output: Foreign key candidates FKCands, Discard set D
 1 FKCands ← {}, D ← {};
 2 I' ← SortByScoreDescending(I);
 3 G(I') ← G(N ← T, E ← {});
 4 while I' is not empty do
 5 |   FKCand ← FirstElement(I');
 6 |   if FKCand causes no circle reference then
 7 |   |   if FKCand is not a PK ⊆ PK then
 8 |   |   |   FKCands ← FKCands ∪ {FKCand};
 9 |   |   |   E ← E∪ Edge(FKCand);
10 |   |   |   if G(I') is connected then
11 |   |   |   |   break;
12 |   |   |   else
13 |   |   |   |   D ← D ∪ {FKCand};
14 |   |   |   end
15 |   |   end
16 |   end
17 |   FKCand+ ← {FKCand'|LHS(FKCand') = LHS(FKCand) ∧ FKCand' ⊆ I'};
18 |   I' ← I' \ FKCand+;
19 end
20 return FKCands, D
```

foreign key candidate set (lines 7.17 and 7.18).

In principle, each table should define a primary key to keep entity integrity. In practice, however, primary keys are not always defined, especially for so-called join tables, which represent many-to-many relationships between two or more tables. Although an indexed primary key helps to query more efficiently, it becomes an encumbrance under frequent data modifications, leading schema designers to avoid defining primary keys for such tables. We observe that the two largest datasets used in our experiments, i.e., `SCOP` and `Musicbrainz` both contain a few join tables without primary key definitions. Because we leverage only minimal UCCs as input to generate primary key candidates, a true multi-column primary key of a join table cannot be included in the candidate set if a subset of these columns is already a UCC. If our algorithm treats the smaller UCC as the primary key for such a table, it is a false positive. A true foreign key, whose LHS is such a false positive primary key, may be considered as spurious IND with the `PK` $\subseteq$ `PK` filtering described above.

A naive solution to avoid this loss of true foreign keys is to use the set of full UCCs as input to produce primary keys. However, this approach has two disadvantages. First, a full set of UCCs contains exponentially more UCCs than its minimal set counterpart, because each column combination subsuming a UCC is also a valid UCC. Second, using the full UCC set may yield wrong primary keys that are non-minimal UCCs, while true PKs are usually minimal ones. Therefore, this approach may predict many wrong primary keys and further undermine the effectiveness of foreign key detection. To improve

**Algorithm 8:** PKReduce()

**Input:** Primary key combination *PKC*, Predicted foreign keys *FKCands*, Discarded foreign key candidates *D*
**Output:** Updated primary keys *PKC*, Updated foreign keys *FKs*

```
1  FKs ← {};
2  score ← score_PKC + score_FKCands;
3  foreach PK in PKC do
4  |   PKC' ← PKC \ PK;
5  |   FKCands' ← UpdateFKCands(PKC', FKCands, D);
6  |   if score < score_PKC' + score_FKCands' then
7  |   |   PKC ← PKC';
8  |   |   FKs ← FKCands';
9  |   |   score ← score_PKC + score_FKs;
10 |   end
11 |   PKC ← PKC \ P;
12 end
13 return PKC, FKs
```

the recall of foreign keys, we propose the primary key reduction step as a post-process technique as shown in Algorithm 8. We observe that by removing inappropriate primary keys from the predicted PK set, we can rediscover some true foreign keys that were discarded when iterating the FK candidates. For each predicted PK, we exclude it from the primary key combination and update the corresponding foreign key candidates. We assume that a predicted PK is not true (and should be discarded) if the overall score rises with its absence. This process consequently leads to two advantages. On the one hand, some false positive primary keys are ultimately removed and previously discarded $\texttt{PK} \subseteq \texttt{PK}$ candidates referencing one of these PKs are restored, because their LHS is no longer primary keys. On the other hand, the false positive foreign keys referencing one of these discarded primary keys are not conceived as valid foreign keys anymore.

Finally, the HoPF algorithm selects a subset of UCCs and INDs as the predicted primary keys and foreign keys, respectively. The result primary key and foreign key sets have much fewer elements than the original input UCC and IND sets. Therefore, using HoPF yields highly correct PKs and FKs yet avoids presenting users with too many spurious ones.

## 4.6 Experiments and Analysis

This section demonstrates the experimental results that show the effectiveness of the proposed HoPF algorithm. After introducing the setup of the experiments, we first present the qualitative evaluation results that HoPF achieves on various datasets. The next part discusses the results of an analysis on incurred errors, including three different types of errors, i.e., incorrect primary key, empty LHS column, and $\texttt{PK} \subseteq \texttt{PK}$. Next, we explore and report the influence on F1-score with different sizes of buckets used to calculate the data distribution foreign key detection feature. After that, we explore the

performance of foreign key discovery without assigning primary keys, showing the necessity of detecting primary keys. Finally, we conclude this section with the performance comparison between HoPF and two state-of-the-art algorithms [Chen et al., 2014; Zhang et al., 2010].

### 4.6.1 Experimental setup

Because HoPF takes UCCs and INDs as input, we assume that the datasets at hand have already been profiled so that these dependency information are available. Given many efficient profiling algorithms [Abedjan et al., 2015], we can readily acquire these dependencies, making this assumption reasonable. We extracted all these metadata with *Metanome*[1], a data profiling platform that allows executing a wide variety of algorithms that detect different kinds of metadata, such as unique column combinations, inclusion dependencies, function dependencies [Papenbrock et al., 2015]. We used the HyUCC [Papenbrock and Naumann, 2017] and Binder [Papenbrock et al., 2015] algorithms to acquire UCCs and INDs of our datasets, respectively.

We conducted our experiments on five different datasets, including three synthetic ones: `TPC-H` and `TPC-E` from the TPC benchmark system[2], and `AdventureWorksDW`[3] from Microsoft's SQL Server database sample pool, as well as two real-world datasets `SCOP`[4] and `MusicBrainz`[5]. All datasets contain complete key and foreign key specifications, which we used as the ground truth for the qualitative evaluation. Also, all tables in these datasets contain header names for each column. Table 4.3 displays detailed statistics of these datasets. As our algorithm is data-driven, we removed empty tables, which appeared only in `MusicBrainz`, reducing the number of tables there from 206 to 124. Both `TPC-H` and `TPC-E` have several parameters to control their scales. We set the same parameter values as those in [Zhang et al., 2010].

Table 4.3: Datasets and their statistics.

| Name | # Tables | # Columns | # UCCs | # PKs | # INDs | # FKs |
|---|---|---|---|---|---|---|
| TPC-H | 8 | 61 | 435 | 8 | 90 | 8 |
| TPC-E | 32 | 185 | 167 | 32 | 411 | 45 |
| AdvWorks | 27 | 321 | 1,434 | 27 | 4,300 | 45 |
| SCOP | 65 | 282 | 120 | 42 | 5,244 | 90 |
| MusicBrainz | 124 | 682 | 252 | 124 | 236,151 | 168 |

The number of inclusion dependencies includes both unary and n-ary ones. HoPF attempts to discover both single-column and multi-column foreign keys out of them, respectively. Note that only 42 true primary keys are defined in `SCOP`, whereas the schema contains 65 tables. All the tables without defined primary keys are join tables, proving that real-world relational tables do not necessarily hold primary keys. To avoid

[1] `www.metanome.de`
[2] `http://www.tpc.org`
[3] `https://github.com/Microsoft/sql-server-samples/releases/tag/adventureworks`
[4] `http://scop.berkeley.edu`
[5] `https://musicbrainz.org`

predicting a key for such tables, we proposed a key reduction technique that attempts to remove these from our results, which has been described in Section 4.6.2.

### 4.6.2 Qualitative evaluation

We present the results of qualitative evaluation on HoPF in this section. The algorithm takes unique column combinations, inclusion dependencies, and various basic statistics of an entire database instance $\mathbb{R}$ as input, and outputs $PK_{\mathbb{R}}$ and $FK_{\mathbb{R}}$, a subset of UCCs and INDs as the respective predicted primary keys and foreign keys. We compared the output sets with the ground truth of each dataset. HoPF predicts at most one primary key for each table. Therefore, we measure the primary key detection quality with the number of correctly predicted primary keys in that of all keys in the ground truth. Because a foreign key may involve two tables in a schema, we evaluated foreign key detection results on schema-level with *precision* and *recall*. Precision is the number of true foreign keys in the $FK_{\mathbb{R}}$ divided by the number of predictions in $FK_{\mathbb{R}}$, whereas recall is the amount of true foreign keys in $FK_{\mathbb{R}}$ divided by the number of foreign keys in the ground truth. While we consider both precision and recall measures in our evaluation, our focus is on recall, because we believe it is easier for users to remove false positives from a small result than to discover false negatives in the large candidate set.

For each predicted FK, HoPF yields a score calculated as the average of the foreign key feature scores. The higher the overall score is, the more confident HoPF is to call this candidate a foreign key. Ideally, the true foreign keys in the predicted ones should obtain higher scores than the spurious ones. Therefore, we ranked the list of the predicted foreign keys in the descending order of their scores, and measure the precision and recall for the top X predicted foreign keys, where the value of X can be between one and the number of all predicted FKs. Figure 4.5 illustrates the precision and recall curves on foreign key discovery for each dataset for top X predicted foreign keys. As we can see from the figure, recall increases gradually, whereas precision does not see an apparent drop. That is to say, for each dataset, the true foreign keys usually appear in the top part of the predicted list, which means they have higher scores than spurious foreign key candidates. And when checking more predicted foreign keys at more bottom positions in the list, we see fewer true foreign keys while more spurious INDs.

Table 4.4 lists the number of true primary keys and true foreign keys discovered by HoPF. It discovers most true primary keys for all datasets. After inspecting the sets of predicted primary keys, we found that errors occur for two reasons. First, the primary key features do not cover those erroneous cases well. For example, the algorithm failed to find the true primary key for the `Financial` table in `TPC-E`, because the table contains a three-column primary key and several spurious unary UCCs. Second, the names of the true primary key columns do not follow the suffix name rule we employ. These two factors cause the missing of the primary key for this table. However, this error accounts for only a small portion. We do not find out other missed PKs for the same reason, indicating that the features we employ for primary key discovery are in general effective. More errors stem from the primary key reduction step. We discuss the influence of this strategy on the effectiveness of both PK and FK discovery below.

We also studied the actual false negative and false positive cases of predicted foreign keys caused by HoPF, and found they fall into the following three categories:

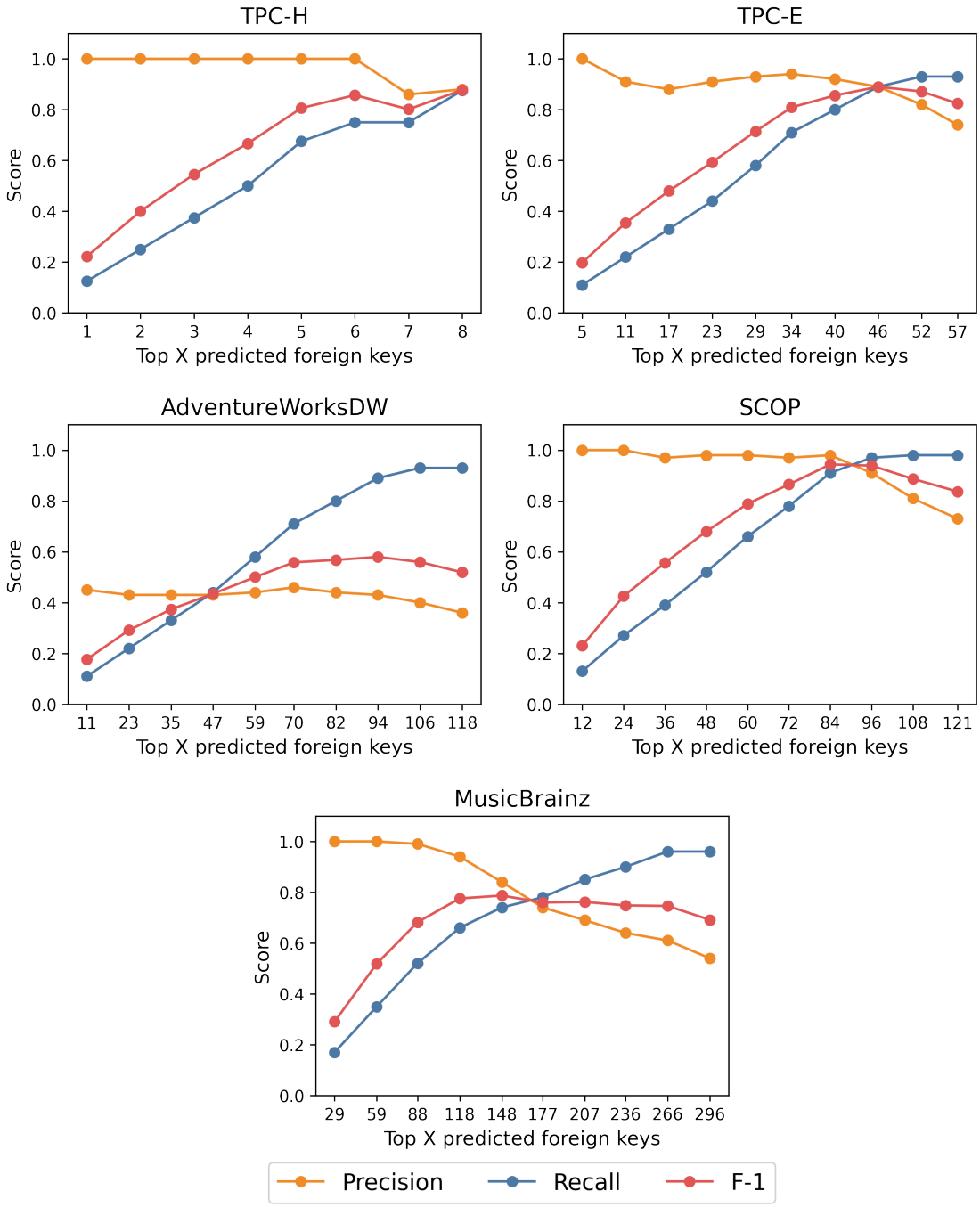



Figure 4.5: Effectiveness of HoPF on the five datasets. The x-axis reflects choosing the top X predicted foreign keys.

**Incorrect primary key.** Because the FKs predicted by HoPF depend on the recognized PKs, an incorrectly predicted primary key could hurt the foreign key discovery results. Therefore, once HoPF obtains a false positive/negative primary key, it might possibly predict multiple false positive/negative foreign keys. In practice, however, we did not witness many errors caused by incorrect primary keys. After inspecting the

Table 4.4: Number of PKs and FKs in the ground truth and number of those discovered by HoPF, marked as 'true' and 'disc.', respectively. 'undoc.' represents the undocumented foreign keys discovered by HoPF.

| Datasets | true PKs | disc. PKs | true FKs | disc. FKs | undoc. FKs |
|---|---|---|---|---|---|
| TPC-H | 8 | 8 | 8 | 7 | 0 |
| TPC-E | 32 | 27 | 45 | 42 | 0 |
| AdvWorks | 27 | 25 | 45 | 40 | 2 |
| SCOP | 42 | 35 | 90 | 88 | 2 |
| MusicBrainz | 124 | 101 | 168 | 161 | 0 |

results, we found out that the spurious primary key candidates usually have low scores.

**Empty LHS column.** If the left-hand side of an IND is empty, all attributes in the LHS contain only Null-values. An IND with an empty LHS can still be a valid foreign key, especially in real-world datasets. However, our data distribution feature is not able to establish an evaluation of such an inclusion dependency. Therefore, our approach filters them out in an early stage. For example, in the `MusicBrainz` dataset, `artist_type.parent` $\subseteq$ `artist_type.id` is a true foreign key that cannot be detected, because the LHS is an empty column. We observed 24 other foreign keys with an empty LHS only in `MusicBrainz`, which reduces recall on this dataset largely.

**`PK` $\subseteq$ `PK`.** Although we employ the `PK` $\subseteq$ `PK` filter in an early stage to remove these candidates, some of them may be added back to the predicted result when the algorithm removes redundant primary keys with primary key reduction. In `MusicBrainz`, for example, `place_alias_type.id` $\subseteq$ `label_alias_type.id` is reconsidered as a foreign key after the algorithm no longer holds {`place_alias_type.id`} as a primary key. Nevertheless, as we can see from Table 4.5, because primary key reduction works well in most cases, HoPF is still able to block most `PK` $\subseteq$ `PK` candidates.

By removing those primary keys that are referenced by some predicted foreign keys with a low score, the discovered foreign keys receive a higher overall score. Therefore, HoPF considers them as non-primary keys. Nevertheless, the loss of the performance of primary key discovery improves the performance of foreign key discovery, as displayed and explained below. It brings back a group of true foreign keys that were considered as `PK` $\subseteq$ `PK` in the previous step.

Table 4.5 displays the number of true primary keys and true foreign keys predicted by HoPF with and without applying the primary key reduction technique. For example, HoPF retrieves 35 primary keys and 88 foreign keys for `SCOP` with primary key reduction. The respective numbers are 41 and 77 for the same dataset without primary key reduction. As we can see from the results, employing primary key reduction helps to retrieve more foreign keys at the price of losing some true primary keys for all datasets, except for `AdvWorks` where two more true foreign keys were removed from predictions after primary key reduction. An inspection of the results of this dataset suggested that primary key reduction falsely removed the true primary key of a table that has a unary primary key, which is referenced by the two missing foreign keys. Therefore, we suggest employing the primary key reduction strategy for datasets containing join tables without

primary key definitions. In these cases, this strategy helps discover more foreign keys at the price of a small loss of true primary keys. We believe it is easier for users to notice the missing primary keys compared to finding the missing foreign keys, which necessitates collectively considering multiple tables. In our experiments, we applied the primary key reduction technique for all datasets.

Table 4.5: True primary keys and true foreign keys predicted with and w/o primary key reduction

| Dataset | # disc. PKs | | # disc. FKs | |
|---|---|---|---|---|
| | w/ PK redu. | w/o PK redu. | w/ PK redu. | w/o PK redu. |
| TPC-H | 8 | 8 | 7 | 7 |
| TPC-E | 27 | 30 | 42 | 40 |
| AdvWorks | 25 | 25 | 40 | 42 |
| SCOP | 35 | 41 | 88 | 77 |
| MusicBrainz | 93 | 109 | 161 | 137 |

We explored the impact of using different bucket sizes on our histogram difference feature. The top figure in Figure 4.6 demonstrates the F1-score for each dataset using different sizes of buckets, and the bottom figure the time expenses for the different parameter values. The F1-scores fluctuate a little when we choose bucket sizes smaller than 20, but stay stable in general. However, the time overhead to construct this data structure increases significantly when using more buckets. For all participant datasets, the optimal bucket size is between 10 and 20, and we choose 20 as our default value.

Using only minimal UCCs as the input of HoPF to produce primary key candidates can possibly miss true n-ary primary keys, if a subset of an n-ary primary key is already a UCC. This problem and the problem it causes on join tables has been discussed early in the section, for which we proposed the primary key reduction strategy. In addition, the lack of complete UCC sets also causes a similar consequence for very small datasets with only few records per relation. In these cases, relations can more readily have unary UCCs, thus shadowing larger UCCs that are the true PKs as we regard only minimal UCCs as our input.

To gauge this effect, we apply HoPF on a `TPC-H` database instance acquired by setting the scale factor to 0.001. In this case, all relations in the dataset contain only dozens of records. The result confirms our suspicion: HoPF is not able to discover the true primary keys of table `lineitem` and `partsupp` who have n-ary primary keys, because both of them also have unary UCCs that are part of the true PK. Unfortunately, our algorithm cannot solve this very-small-table dilemma without changing the prerequisite of only consuming minimal UCCs. As mentioned in Section 4.1.2, changing the prerequisite so as to use also non-minimal UCCs does not solve this dilemma, because the proposed PK features favor UCC candidates with small cardinality, and thus HoPF is more liable to refer to them as true primary keys.

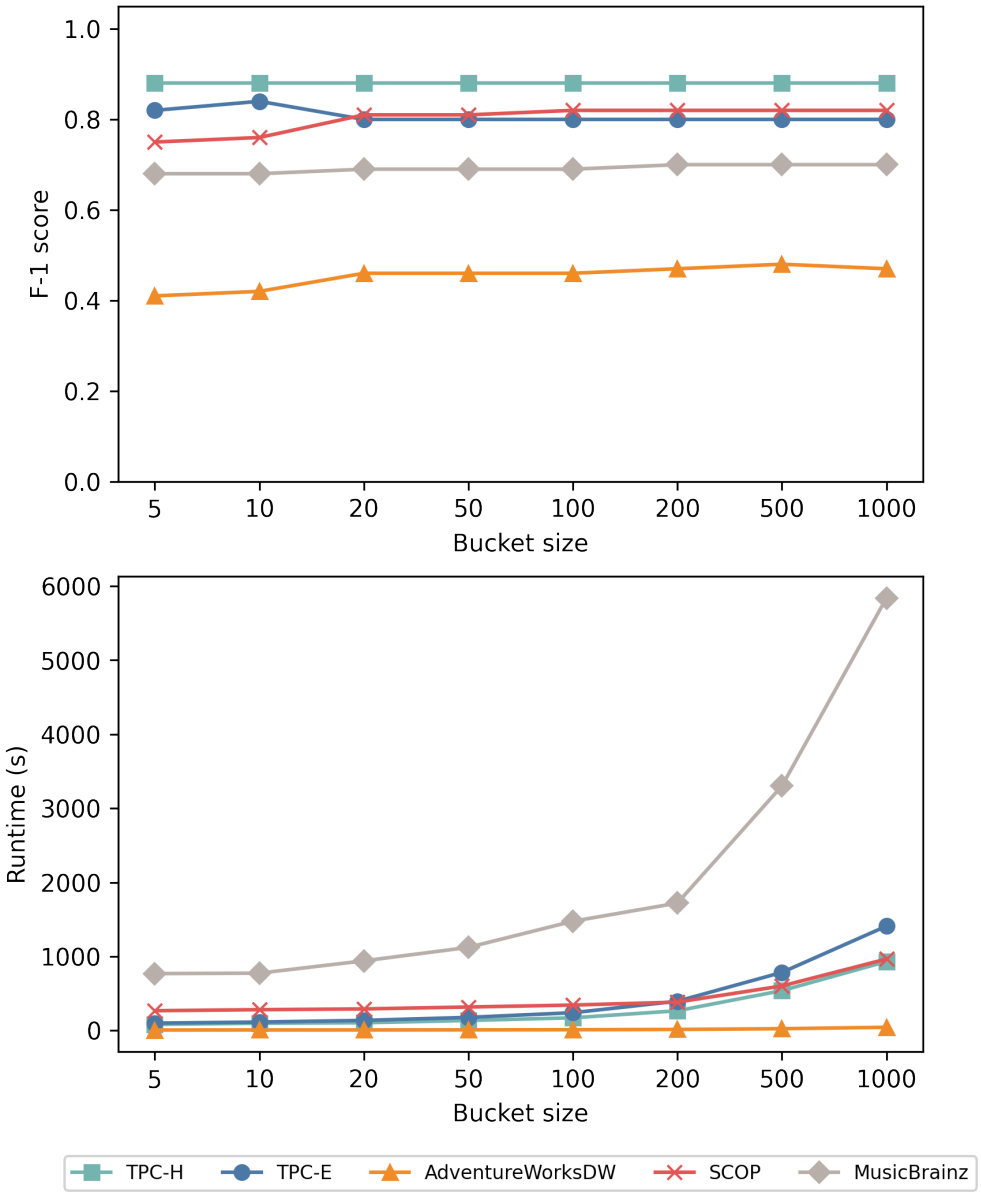



Figure 4.6: The F1-score with different size of buckets and their corresponding time expense for all datasets.

### 4.6.3 Foreign key detection without primary keys

Previous works all assume that primary keys are at present. However, this assumption is too optimistic for many databases, especially for those dumped in plain-text files, where these constraints might not be tightly attached to data themselves. In these cases, foreign key discovery algorithms based on this assumption might break or predict possibly many erroneous foreign keys. Here, we use HoPF to explore the influence on the quality of discovered foreign keys without the presence of primary keys. Fortunately, even without discovering the primary keys, our method can still rank the list of INDs by their foreign

key score. Instead of predicting primary keys holistically with foreign keys, we let HoPF output a predicted foreign key, as long as its RHS is a UCC.

Table 4.6 displays the detected foreign keys with and without knowledge of primary keys. Consistent with our assumption, without the primary keys, HoPF obtains fewer true positive foreign keys even with a bigger foreign key candidate search space. For example, if we ignore the primary keys and rank only the foreign key candidates by their score, we obtain 141 true foreign keys out of 1079 candidates in `Musicbrainz`, whereas we obtain 161 true foreign keys out of only 296 candidates if we detect the primary keys first and use them as the input for foreign key detection.

Table 4.6: Detected foreign keys without and with knowledge of primary keys. "# Fk cand." is the number of INDs considered to detect foreign keys, whereas "# disc. FKs" is the number of the detected foreign keys.

| Dataset | # FKs | without PKs | | with PKs | |
|---|---|---|---|---|---|
| | | # disc. FKs | # FK cand. | # disc. FKs | # FK cand. |
| TPC-H | 8 | 2 | 18 | 7 | 8 |
| TPC-E | 45 | 41 | 77 | 42 | 57 |
| AdvWorks | 45 | 39 | 369 | 40 | 118 |
| SCOP | 90 | 88 | 167 | 88 | 121 |
| MusicBrainz | 168 | 141 | 1,079 | 161 | 296 |

In general, we consider as foreign key candidates only those inclusion dependencies whose RHS is among the primary key set. Without knowledge of primary keys, HoPF would consider each IND whose RHS is a UCC as a foreign key candidate. This explains why the number of foreign key candidates grows without considering primary keys; each table typically contains many UCCs, and each of them may contribute several valid INDs into the candidate set.

Restricted by the *uniqueness of foreign key* rule, a true foreign key sharing the same LHS with another foreign key candidate is excluded if the latter one has a higher score and is predicted as a foreign key. However, if an incorrect foreign key is predicted, adding a true foreign key to the final result may cause a circle-reference, and thus be rejected. This explains why fewer true positive foreign keys are acquired without the knowledge of primary keys.

### 4.6.4 Undocumented foreign key discovery

In addition to discovering documented foreign keys, further potential foreign keys were found in `AdventureWorksDW` and `SCOP` in spite of their absence in the ground truth. These uncertain foreign keys fall into two categories: missing ones and erroneous ones. For instance, `pdb_release_author.pdb_author_id` $\subseteq$ `pdb_author.id` in `SCOP` was counted as a false positive while we believe it is in fact a true foreign key. On the other hand, `cdd.release_id` $\subseteq$ `pfam_release.id` is a documented foreign key while we believe it to be incorrectly documented; the correct foreign key should likely be `cdd.release_id` $\subseteq$ `cdd_release.id`, which was predicted by HoPF.

We can imagine several reasons for these undocumented foreign keys, e.g., loss during data migration or removal by schema designers for query efficiency. The number of undocumented but true foreign keys predicted by HoPF for each dataset is shown in the last column in Table 4.4. The discovery of undocumented foreign keys enriches this metadata, and therefore may provide us with insight for further database application such as data integration.

### 4.6.5 Comparison

We have witnessed a few previous works on detecting foreign keys in relational tables. We re-implemented two state-of-the-art algorithms proposed in [Zhang et al., 2010] and [Chen et al., 2014], which we dub Randomness and FastFK, respectively, and compared their performances with HoPF. As explained in more detail in Section 4.2, the Randomness algorithm measures the data distribution between LHS and RHS with a so-called *randomness* measure. FastFK, which assumes that there exist only single-attribute foreign keys, employs a few features to score foreign key candidates, as well as some pruning rules to decrease the search space. For Randomness, we applied $\theta = 0.9$, and bottom 256 sketches, 256 and 16 quantiles for unary and n-ary foreign keys candidates, respectively, – the sweet spots determined by the original authors. As both previous works assume that primary keys are present and known, we provide them with true primary keys in this experimental setting.

To improve results, the Randomness algorithm also considers column names, keeping only candidates with exactly matched names. The authors apply this technique only to `TPC-H` and `TPC-E`, relying on external documentation to guide a manual trimming of column labels before the matching. We compared their results with HoPF and FastFK when switching on the *column name* features. We also compared the results of Randomness without conducting this post-processing with the other two when switching off the column name features. Table 4.7 displays the details of this comparison.

As seen in the table, without knowledge of column names, all three algorithms experience a drop in precision and recall, proving that column names are indicative in recognizing foreign keys. While using the column names, the performances of the three algorithms are quite similar on synthetic datasets (`TPC-H`, `TPC-E`, and `AdventureWorks`), while HoPF outperforms the two compared approaches in the real-world datasets (`SCOP` and `MusicBrainz`). We also notice that HoPF produces smaller predicted foreign key sets for all the datasets compared to the other two approaches, making any post-processing by a human expert easier.

## 4.7 Conclusions

Primary keys and foreign keys are important integrity constraints to keep databases consistent. However, data stored as plain-text files or dumps do not always carry these constraint definitions and in many cases, it is up to the user of the data to identify them and thus understand the data better and enforce its quality. As schemata can be quite large and complex, automatic discovery of primary and foreign keys is a relevant (and challenging) research topic.

Figure 4.7: Comparison of foreign key detection among HoPF and other two previous work [Chen et al., 2014; Zhang et al., 2010]. In HoPF and [Chen et al., 2014], column name similarity between LHS and RHS of a foreign key candidate is set as a foreign key feature while non-matching column names of LHS and RHS is used as a postprocessing [Zhang et al., 2010]. We use four metrics to measure the performances: precision (P), recall (R), F1-score, and the number of predicted foreign keys (Predicted FKs)

| Dataset | True FKs | Algorithm | with column names | | | | without column names | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | Precision | Recall | F1 | Predicted FKs | Precision | Recall | F1 | Predicted FKs |
| TPC-H | 8 | FastFK | 0.56 | 0.90 | 0.69 | 16 | 0.56 | 0.90 | 0.69 | 16 |
| | | Randomness | **1.00** | **1.00** | **1.00** | 8 | 0.21 | **1.00** | 0.35 | 39 |
| | | HoPF | 0.88 | 0.88 | 0.88 | 8 | **0.88** | 0.88 | **0.88** | 8 |
| TPC-E | 45 | FastFK | 0.72 | **0.95** | 0.82 | 59 | 0.59 | 0.78 | 0.67 | 59 |
| | | Randomness | **1.00** | 0.89 | **0.94** | 45 | 0.57 | **0.82** | 0.67 | 308 |
| | | HoPF | 0.74 | 0.93 | 0.82 | 57 | **0.64** | **0.82** | **0.72** | 57 |
| AdvWorks | 45 | FastFK | 0.32 | **0.97** | 0.49 | 131 | 0.24 | **0.72** | 0.37 | 131 |
| | | Randomness | **0.41** | 0.90 | **0.56** | 122 | 0.21 | 0.58 | 0.31 | 122 |
| | | HoPF | 0.34 | 0.89 | 0.49 | 118 | **0.27** | **0.72** | **0.39** | 120 |
| SCOP | 90 | FastFK | 0.57 | **0.94** | 0.71 | 149 | 0.53 | **0.87** | 0.66 | 149 |
| | | Randomness | 0.36 | 0.61 | 0.45 | 151 | 0.36 | 0.61 | 0.45 | 151 |
| | | HoPF | **0.73** | **0.98** | **0.84** | 121 | **0.63** | 0.82 | **0.71** | 118 |
| MusicBrainz | 168 | FastFK | 0.33 | 0.74 | 0.46 | 368 | **0.28** | **0.62** | **0.39** | 367 |
| | | Randomness | 0.24 | 0.49 | 0.32 | 341 | 0.24 | 0.49 | 0.32 | 341 |
| | | HoPF | **0.54** | **0.96** | **0.69** | 296 | **0.28** | 0.50 | 0.36 | 289 |

Previous efforts were made to discover foreign keys and primary keys separately. In this work, we have proposed the HoPF algorithm to integrate primary key and foreign key detection in a holistic fashion. We employ a set of carefully designed features to score and distinguish both the true primary keys and foreign keys from the spurious UCCs and INDs. We employ several useful pruning rules to effectively reduce the search spaces of both PKs and FKs.

In performance experiments on five diverse datasets, our algorithm reaches an average recall of 88% and 91% in primary keys and foreign keys discovery, respectively. We show with an experiment that without knowledge of primary keys (which is assumed in related work), the performance of foreign key discovery is much worse, indicating the necessity to discover primary keys in advance or simultaneously. We compared precision and recall with the state-of-art algorithms.

We assume data are clean: values are well-formatted and follow the data type defined on the column. However, this is often not true in reality. For example, for a date column with a defined "YYYY-MM-DD" format, there might be few records with values in "DD-MM-YYYY" format on this column. Primary keys and foreign keys are defined on the schema regardless of the relation instances. Discovering PKs and FKs in relational tables with data errors would be an interesting future work.

# Chapter 5

# Conclusion and Outlook

CSV files are a type of plain-text data files that store tabular and textual data. Each file has a unique content structure: information with particular meanings can appear in arbitrary cells. Such files represent a great amount of data resources that are extensively used for data analytics. Due to the ad-hoc structure of CSV files, it is usually difficult to extract information from them. In many cases, even loading them into databases is not trivial. Metadata are useful properties to describe CSV files, which help data scientists to understand and process their data. Overall, the presence of metadata enables various downstream applications, such as data cleaning [Mahdavi and Abedjan, 2020; Rekatsinas et al., 2017], self-service data preparation [Hellerstein et al., 2018], data preparation suggestion and automation [Guo et al., 2011; Jiang et al., 2019; Yan and He, 2020; Yang et al., 2021], and visualization [Hansen and Johnson, 2011].

However, these meta-information are not always bundled with data or do not even exist. The detection of metadata is an indispensable work before data preparation and the usage of the content in these files for innovative and valuable data analytics. In this thesis, we introduce a taxonomy of six types of metadata that can be extracted from CSV files (Chapter 1), and describe our original algorithms for the discovery of five specific metadata: Strudel for the *type of lines and cells* (Chapter 2), AggreCol for *aggregations* (Chapter 3) in CSV files with unique content structure, and HoPF for *primary keys* and *foreign keys* (Chapter 4) in CSV files containing a relational table. CSV files often preserve data in cells similar to spreadsheet files w.r.t. the layout of content. However, the former can keep only values and positions of data, whereas the latter may incorporate various stylistic features, such as background color and font style of cells, and formulas. While the usefulness of rich-text features on detecting metadata has been proven by previous works, one of the major objectives of our algorithms is to address the metadata detection problem *without using stylistic features*, and therefore generalize metadata detection to plain-text files. Another target of most of our studies is to explore whether the presence of some metadata may affect the discovery results of other metadata. For example, whether we can achieve better foreign key detection results with the knowledge of primary keys, or cell classification results with the presence of line classes.

Besides the above major contributions to metadata detection, I have also created

various corpora of annotated data files and publicized all the data and the code[1]. I have participated in developing other related works during my Ph.D. study, which are not presented in this thesis because of its limited scope. These works include MONDRIAN that identifies the positions of tables and layout templates in complex multi-table data files [Vitagliano et al., 2021], SURAGH that distinguishes ill-formed records from well-formed ones in CSV files, and EXTRACTABLE that detects the boundaries of tables in plain-text files [Hübscher, 2021].

The problem of detecting metadata in data files is far from being solved. While the discovery of each metadata introduced in our taxonomy (Chapter 1) is not trivial and deserves dedicated research efforts, we discuss some future works that are most related to our main contributions in the following.

**Metadata detection in data with errors**

The algorithms introduced in this thesis mostly assume that the used datasets are clean. However, real-world data often have diverse quality issues. In a relational table, for example, for a date column with a defined "YYYY-MM-DD" format, there might be few records with values in "DD-MM-YYYY" format on this column, or the primary key column happens to include duplicate values. Key and foreign key constraints cannot be correctly discovered with algorithms that assume the datasets are clean. An erroneous number may also affect the results of aggregation detection: computational features may fail to recognize the arithmetic relationships amongst numbers. In light of the above examples, a general interesting future work is to discover metadata from data with quality problems. One way to cope with this type of problems could be detecting approximate metadata that are valid on a subset of data.

**Use of semantic features**

Our algorithms do not leverage semantic features for the discovery of the respective metadata. However, semantics of the content might be very useful in such tasks. Take aggregation detection as an example, we have found that purely using keywords, such as "total", cannot reliably identify aggregations that do not use these keywords in table headers: the number in a row with the header "Europe" should sum up the numbers in the rows with the headers of individual European countries. For line and cell class detection, a cell with only a currency symbol often indicates the unit of the numbers in the same row or column. Such information about data should not be classified as data. We envision that incorporating the semantic meaning of content into models is likely to improve the metadata detection results.

**Metadata-specific future work**

There are also interesting future research directions for the discovery of specific metadata. Assuming that content is mostly organized from top to bottom in these files, our STRUDEL approach classifies the lines in CSV data files. However, columns, i.e., vertical lines in CSV files, may also carry useful information to help determine the class of cells. An interesting extension of the STRUDEL algorithm should explore the impact of column type on the cell classification task. Regarding our aggregation detection approach AGGRECOL, there is room for improvement by dropping the assumption, which states that the aggregate and the range of an aggregation must be either in the same row or in the

[1] https://hpi.de/naumann/projects/data-preparation.html

same column. Also, real-world aggregations may involve multiple aggregation functions, which have not been addressed by AggreCol. What is more, an improved algorithm may address more functions, such as min/max, median, and standard deviation.

Although data are being created at an unprecedentedly fast pace, many of them are stored in formats that cannot be directly consumed by dedicated data analysis tools. As a consequence, data preparation is a significant preceding process to enabling downstream data analytic tasks. CSV files persist tremendous data in ad-hoc shapes and forms, which need to be prepared first. Data preparation is notoriously known to be time-consuming, partly because data scientists spend a lot of time comprehending and exploring their data, which can be more efficient with the knowledge of metadata. In this thesis, we put our efforts into discovering metadata in CSV files, which is the cornerstone of many data-driven applications.

# References


[1] Ziawasch Abedjan and Felix Naumann. Advancing the discovery of unique column combinations. In *Proceedings of the International Conference on Information and Knowledge Management (CIKM)*, pages 1565–1570, 2011.

[2] Ziawasch Abedjan, Lukasz Golab, and Felix Naumann. Profiling relational data: a survey. *VLDB Journal*, 24(4):557–581, 2015.

[3] Robin Abraham and Martin Erwig. GoalDebug: A spreadsheet debugger for end users. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 251–260, 2007.

[4] Robin Abraham and Martin Erwig. UCheck: A spreadsheet type checker for end users. *Journal of Visual Languages & Computing*, 18(1):71–95, 2007.

[5] Marco D Adelfio and Hanan Samet. Schema extraction for tabular data on the web. *PVLDB*, 6(6):421–432, 2013.

[6] Abhimanyu S Ahuja. The impact of artificial intelligence in medicine on the future role of the physician. *PeerJ*, 7:e7702, 2019.

[7] Daniel W Barowy, Sumit Gulwani, Ted Hart, and Benjamin Zorn. FlashRelate: extracting relational data from semi-structured spreadsheets using examples. *ACM SIGPLAN Notices*, 50(6):218–228, 2015.

[8] Daniel W Barowy, Emery D Berger, and Benjamin Zorn. ExceLint: automatically finding spreadsheet formula errors. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–26, 2018.

[9] Kiran Bhageshpur. Data Is The New Oil – And That's A Good Thing. `https://www.forbes.com/sites/forbestechcouncil/2019/11/15/data-is-the-new-oil-and-thats-a-good-thing/?sh=591495857304`, 2019. Accessed: 2021-09-16.

[10] Anil Bhattacharyya. On a measure of divergence between two multinomial populations. *Sankhyā: the Indian Journal of Statistics*, pages 401–406, 1946.

[11] Besim Bilalli, Alberto Abelló, Tomàs Aluja-Banet, and Robert Wrembel. Towards intelligent data analysis: The metadata challenge. In *Proceedings of the International Conference on Internet of Things and Big Data*, pages 331–338, 2016.

[12] Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[13] Michael J Cafarella, Alon Halevy, Daisy Zhe Wang, Eugene Wu, and Yang Zhang. WebTables: Exploring the power of tables on the web. *PVLDB*, 1(1):538–549, 2008.

[14] Michael J Cafarella, Alon Y Halevy, Yang Zhang, Daisy Zhe Wang, and Eugene Wu. Uncovering the relational web. In *Proceedings of the ACM SIGMOD Workshop on the Web and Databases (WebDB)*, 2008.

[15] Loredana Caruccio, Vincenzo Deufemia, and Giuseppe Polese. Relaxed functional dependencies — a survey of approaches. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 28(1):147–165, 2015.

[16] HC Stephen Chan, Hanbin Shan, Thamani Dahoun, Horst Vogel, and Shuguang Yuan. Advancing drug discovery via artificial intelligence. *Trends in Pharmacological Sciences*, 40(8):592–604, 2019.

[17] Girish Chandrashekar and Ferat Sahin. A survey on feature selection methods. *Computers & Electrical Engineering*, 40(1):16–28, 2014.

[18] Surajit Chaudhuri, Kris Ganjam, Venkatesh Ganti, and Rajeev Motwani. Robust and efficient fuzzy match for online data cleaning. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 313–324, 2003.

[19] Xinyun Chen, Petros Maniatis, Rishabh Singh, Charles Sutton, Hanjun Dai, Max Lin, and Denny Zhou. Spreadsheetcoder: Formula prediction from semi-structured context. In *Proceedings of the International Conference on Machine Learning*, pages 1661–1672, 2021.

[20] Zhe Chen and Michael Cafarella. Automatic web spreadsheet data extraction. In *Proceedings of the International Workshop on Semantic Search over the Web*, pages 1–8, 2013.

[21] Zhe Chen and Michael Cafarella. Integrating spreadsheet data via accurate and low-effort extraction. In *Proceedings of the International Conference on Knowledge discovery and data mining (SIGKDD)*, pages 1126–1135, 2014.

[22] Zhe Chen, Mike Cafarella, Jun Chen, Daniel Prevo, and Junfeng Zhuang. Senbazuru: A prototype spreadsheet database management system. *PVLDB*, 6(12): 1202–1205, 2013.

[23] Zhe Chen, Sasha Dadiomov, Richard Wesley, Gang Xiao, Daniel Cory, Michael Cafarella, and Jock Mackinlay. Spreadsheet property detection with rule-assisted active learning. In *Proceedings of the International Conference on Information and Knowledge Management (CIKM)*, pages 999–1008, 2017.

[24] Zhimin Chen, Vivek R. Narasayya, and Surajit Chaudhuri. Fast foreign-key detection in Microsoft SQL server powerpivot for excel. *PVLDB*, 7(13):1417–1428, 2014.

[25] Christina Christodoulakis, Eric B Munson, Moshe Gabel, Angela Demke Brown, and Renée J Miller. Pytheas: Pattern-based table discovery in csv files. *PVLDB*, 13(11):2075–2089, 2020.

[26] Xu Chu, Yeye He, Kaushik Chakrabarti, and Kris Ganjam. Tegra: Table extraction by global record alignment. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 1713–1728, 2015.

[27] Remi Coletta, Emmanuel Castanier, Patrick Valduriez, Christian Frisch, DuyHoa Ngo, and Zohra Bellahsene. Public data integration with websmatch. In *Proceedings of the International Workshop on Open Data (WOD)*, pages 5–12, 2012.

[28] Cristian Consonni, Paolo Sottovia, Alberto Montresor, and Yannis Velegrakis. Discovering order dependencies through order compatibility. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 409–420, 2019.

[29] Hugh Dalton. The measurement of the inequality of incomes. *The Economic Journal*, 30(119):348–361, 1920.

[30] Tamraparni Dasu and Theodore Johnson. *Exploratory data mining and data cleaning*, volume 479. John Wiley & Sons, 2003.

[31] Pandas development team. pandas-dev/pandas: Pandas, February 2020. URL `https://doi.org/10.5281/zenodo.3509134`.

[32] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. 2018.

[33] Dimiter V Dimitrov. Medical internet of things and big data in healthcare. *Healthcare Informatics Research*, 22(3):156–163, 2016.

[34] Till Döhmen, Hannes Mühleisen, and Peter Boncz. Multi-hypothesis CSV parsing. In *Proceedings of the International Conference on Scientific and Statistical Database Management (SSDBM)*, pages 1–12, 2017.

[35] Haoyu Dong, Shijie Liu, Shi Han, Zhouyu Fu, and Dongmei Zhang. TableSense: Spreadsheet table detection with convolutional neural networks. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, volume 33, pages 69–76, 2019.

[36] Wensheng Dou, Shing-Chi Cheung, and Jun Wei. Is spreadsheet ambiguity harmful? Detecting and repairing spreadsheet smells due to ambiguous computation. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 848–858, 2014.

[37] Wensheng Dou, Shi Han, Liang Xu, Dongmei Zhang, and Jun Wei. Expandable group identification in spreadsheets. In *Proceedings of the ACM/IEEE International Conference on Automated Software Engineering*, pages 498–508, 2018.

[38] Julian Eberius, Christoper Werner, Maik Thiele, Katrin Braunschweig, Lars Dannecker, and Wolfgang Lehner. DeExcelerator: a framework for extracting relational data from partially structured documents. In *Proceedings of the International Conference on Information and Knowledge Management (CIKM)*, pages 2477–2480, 2013.

[39] The Economist. The world's most valuable resource is no longer oil, but data. `https://www.economist.com/leaders/2017/05/06/the-worlds-most-valuable-resource-is-no-longer-oil-but-data`, 2017. Accessed: 2021-09-15.

[40] Rebecca Eichler, Corinna Giebler, Christoph Gröger, Holger Schwarz, and Bernhard Mitschang. Modeling metadata in data lakes—a generic model. *Data & Knowledge Engineering*, 136:101931, 2021.

[41] Hazem Elmeleegy, Jayant Madhavan, and Alon Halevy. Harvesting relational tables from lists on the web. *PVLDB*, 2(1):1078–1089, 2009.

[42] David W Embley, Mukkai Krishnamoorthy, George Nagy, and Sharad Seth. Factoring web tables. In *International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems*, pages 253–263, 2011.

[43] David W Embley, Mukkai S Krishnamoorthy, George Nagy, and Sharad Seth. Converting heterogeneous statistical tables on the web to searchable databases. *International Journal on Document Analysis and Recognition (IJDAR)*, 19(2):119–138, 2016.

[44] Martin Faust, David Schwalb, and Hasso Plattner. Composite group-keys – space-efficient indexing of multiple columns for compressed in-memory column stores. In *Proceedings of the International Workshop on In Memory Data Management and Analytics, IMDM*, pages 139–150, 2014.

[45] Marc Fisher and Gregg Rothermel. The EUSES spreadsheet corpus: a shared resource for supporting experimentation with spreadsheet dependability mechanisms. In *Proceedings of the First Workshop on End-user Software Engineering*, pages 1–5, 2005.

[46] Neil Foshay, Avinandan Mukherjee, and Andrew Taylor. Does data warehouse end-user metadata add value? *Communications of the ACM*, 50(11):70–77, 2007.

[47] Tim Fountaine, Brian McCarthy, and Tamim Saleh. Building the AI-powered organization. *Harvard Business Review*, 97(4):62–73, 2019.

[48] Martin Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.

[49] Wolfgang Gatterbauer, Paul Bohunsky, Marcus Herzog, Bernhard Krüpl, and Bernhard Pollak. Towards domain-independent information extraction from web tables. In *Proceedings of the International World Wide Web Conference (WWW)*, pages 71–80, 2007.

[50] Chang Ge, Yinan Li, Eric Eilebrecht, Badrish Chandramouli, and Donald Kossmann. Speculative distributed CSV data parsing for big data analytics. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 883–899, 2019.

[51] Corinna Giebler, Christoph Gröger, Eva Hoos, Holger Schwarz, and Bernhard Mitschang. Leveraging the data lake: Current state and challenges. In *International Conference on Big Data Analytics and Knowledge Discovery*, pages 179–188. Springer, 2019.

[52] Majid Ghasemi Gol, Jay Pujara, and Pedro Szekely. Tabular cell classification using pre-trained cell embeddings. In *Proceedings of the International Conference on Data Mining (ICDM)*, pages 230–239, 2019.

[53] Julius Gonsior, Josephine Rehak, Maik Thiele, Elvis Koci, Michael Günther, and Wolfgang Lehner. Active learning for spreadsheet cell classification. In *Proceedings of the Workshop on Search, Exploration, and Analysis in Heterogeneous Datastores*, 2020.

[54] Philip J Guo, Sean Kandel, Joseph M Hellerstein, and Jeffrey Heer. Proactive wrangling: Mixed-initiative end-user programming of data transformation scripts. In *Proceedings of the annual ACM symposium on User interface software and technology*, pages 65–74, 2011.

[55] Alon Y Halevy, Flip Korn, Natalya Fridman Noy, Christopher Olston, Neoklis Polyzotis, Sudip Roy, and Steven Euijong Whang. Managing Google's data lake: an overview of the Goods system. *IEEE Data Engineering Bulletin*, 39(3):5–14, 2016.

[56] Charles D Hansen and Chris R Johnson. *Visualization handbook*. Elsevier, 2011.

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

[58] F Hermans, M Pinzger, and A van Deursen. Measuring spreadsheet formula understandability. *Proceedings European Spreadsheet Risks Interest Group (EuSpRIG)*, pages 77–96, 2012.

[59] Felienne Hermans, Martin Pinzger, and Arie Van Deursen. Detecting code smells in spreadsheet formulas. In *International Conference on Software Maintenance (ICSM)*, pages 409–418. IEEE, 2012.

[60] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[61] Birgit Hofer, André Riboira, Franz Wotawa, Rui Abreu, and Elisabeth Getzner. On the empirical evaluation of fault localization techniques for spreadsheets. In *International Conference on Fundamental Approaches to Software Engineering*, pages 68–82. Springer, 2013.

[62] Arne Holst. Volume of data/information created, captured, copied, and consumed worldwide from 2010 to 2025. `https://www.statista.com/statistics/871513/worldwide-data-created/#statisticContainer`, 2021. Accessed: 2021-09-12.

[63] Leonardo Hübscher. ExtracTable: Extracting tables from plain text, 2021.

[64] IBM Corporation. IBM FORTRAN Program Products for OS and the CMS Component of VM/370 General Information. page 17, 1972.

[65] Muhammad Nazrul Islam, Toki Tahmid Inan, Suzzana Rafi, Syeda Sabrina Akter, Iqbal H Sarker, and AKM Islam. A survey on the use of AI and ML for fighting the covid-19 pandemic. *arXiv preprint arXiv:2008.07449*, 2020.

[66] Neeraj Kumar Jain, RK Saini, and Preeti Mittal. A review on traffic monitoring system techniques. In *Soft Computing: Theories and Applications*, pages 569–577. Springer, 2019.

[67] Bas Jansen and Felienne Hermans. Code smells in spreadsheet formulas revisited on an industrial dataset. In *International Conference on Software Maintenance and Evolution (ICSME)*, pages 372–380, 2015.

[68] Lan Jiang and Felix Naumann. Holistic primary key and foreign key detection. *Journal of Intelligent Information Systems (JIIS)*, 54(3):439–461, 2020.

[69] Lan Jiang, Gerardo Vitagliano, and Felix Naumann. A scoring-based approach for data preparator suggestion. In *LWDA*, pages 6–9, 2019.

[70] Lan Jiang, Gerardo Vitagliano, and Felix Naumann. Structure detection in verbose CSV files. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 193–204, 2021.

[71] Lan Jiang, Gerardo Vitagliano, Mazhar Hameed, and Felix Naumann. Aggregation detection in verbose CSV files. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, 2022. (accepted).

[72] Sean Kandel, Andreas Paepcke, Joseph M Hellerstein, and Jeffrey Heer. Enterprise data analysis and visualization: An interview study. *IEEE Transactions on Visualization and Computer Graphics*, 18(12):2917–2926, 2012.

[73] Martti Kantola, Heikki Mannila, Kari-Jouko Räihä, and Harri Siirtola. Discovering functional and inclusion dependencies in relational databases. *International journal of intelligent systems*, 7(7):591–607, 1992.

[74] Anoop R Katti, Christian Reisswig, Cordula Guder, Sebastian Brarda, Steffen Bickel, Johannes Höhne, and Jean Baptiste Faddoul. Chargrid: Towards understanding 2d documents. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, pages 4459–4469, 2018.

[75] Elvis Koci, Maik Thiele, Oscar Romero, and Wolfgang Lehner. Cell classification for layout recognition in spreadsheets. In *International Joint Conference on Knowledge Discovery, Knowledge Engineering, and Knowledge Management (IC3K)*, pages 78–100. Springer, 2016.

[76] Elvis Koci, Maik Thiele, Óscar Romero Moral, and Wolfgang Lehner. A machine learning approach for layout inference in spreadsheets. In *International Joint Conference on Knowledge Discovery, Knowledge Engineering, and Knowledge Management (IC3K)*, pages 77–88, 2016.

[77] Elvis Koci, Maik Thiele, Oscar Romero, and Wolfgang Lehner. Table identification and reconstruction in spreadsheets. In *International Conference on Advanced Information Systems Engineering*, pages 527–541. Springer, 2017.

[78] Sebastian Kruse and Felix Naumann. Efficient discovery of approximate dependencies. *PVLDB*, 11(7):759–772, 2018.

[79] Sebastian Kruse, Thorsten Papenbrock, Christian Dullweber, Moritz Finke, Manuel Hegner, Martin Zabel, Christian Zöllner, and Felix Naumann. Fast approximate discovery of inclusion dependencies. *Proceedings of the Conference Datenbanksysteme in Büro, Technik und Wissenschaft (BTW)*, 2017.

[80] Vladimir I Levenshtein et al. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet Physics Doklady*, volume 10, pages 707–710, 1966.

[81] Shanjian Li and Katsuhiko Momoi. A composite approach to language/encoding detection. In *Proceedings of the International Unicode Conference*, pages 1–14, 2001.

[82] Jixue Liu, Jiuyong Li, Chengfei Liu, and Yongfeng Chen. Discover dependencies from data - A review. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 24(2):251–264, 2012.

[83] Ying Liu, Prasenjit Mitra, and C Lee Giles. Identifying table boundaries in digital documents via sparse line detection. In *Proceedings of the International Conference on Information and Knowledge Management (CIKM)*, pages 1311–1320, 2008.

[84] Vanessa Long. *An agent-based approach to table recognition and interpretation.* PhD thesis, Macquarie University Sydney, NSW, Australia, 2010.

[85] Stéphane Lopes, Jean-Marc Petit, and Farouk Toumani. Discovering interesting inclusion dependencies: application to logical database tuning. *Information Systems (IS)*, 27(1):1–19, 2002.

[86] Claudio L. Lucchesi and Sylvia L. Osborn. Candidate keys for relations. *Journal of Computer and System Sciences*, 17(2):270–279, 1978.

[87] Mohammad Mahdavi and Ziawasch Abedjan. Baran: Effective error correction via a unified context representation and transfer learning. *PVLDB*, 13(12):1948–1961, 2020.

[88] Mohammad Mahdavi, Ziawasch Abedjan, Raul Castro Fernandez, Samuel Madden, Mourad Ouzzani, Michael Stonebraker, and Nan Tang. Raha: A configuration-free error detection system. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 865–882, 2019.

[89] Vishal Mandal, Abdul Rashid Mussah, Peng Jin, Yaw Adu-Gyamfi, et al. Artificial intelligence-enabled traffic monitoring system. *Sustainability*, 12(21):1–21, 2020.

[90] Fabien De Marchi, Stéphane Lopes, and Jean-Marc Petit. Unary and n-ary inclusion dependency discovery in relational databases. *Journal of Intelligent Information Systems*, 32(1):53–73, 2009.

[91] Mozhgan Memari, Sebastian Link, and Gillian Dobbie. SQL data profiling of foreign keys. In *Proceedings of the International Conference on Conceptual Modeling (ER)*, pages 229–243, 2015.

[92] George Nagy. TANGO-DocLab web tables from international statistical sites (Troy_200). `http://tc11.cvc.uab.es/datasets/Troy_200_1`, 2010. Accessed: 2021-07-20.

[93] Fatemeh Nargesian, Erkang Zhu, Ken Q Pu, and Renée J Miller. Table union search on open data. *PVLDB*, 11(7):813–825, 2018.

[94] Fatemeh Nargesian, Erkang Zhu, Renée J Miller, Ken Q Pu, and Patricia C Arocena. Data lake management: challenges and opportunities. *PVLDB*, 12(12): 1986–1989, 2019.

[95] Paulo Oliveira, Fátima Rodrigues, Pedro Henriques, and Helena Galhardas. A taxonomy of data quality problems. In *International Workshop on Data and Information Quality*, pages 219–233, 2005.

[96] Michael Palmer. Data is the New Oil. `https://ana.blogs.com/maestros/2006/11/data_is_the_new.html`, 2006. Accessed: 2021-09-16.

[97] Thorsten Papenbrock. *Data profiling - efficient discovery of dependencies.* doctoralthesis, Universität Potsdam, 2017.

[98] Thorsten Papenbrock and Felix Naumann. Data-driven schema normalization. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 342–353, 2017.

[99] Thorsten Papenbrock and Felix Naumann. A hybrid approach for efficient unique column combination discovery. *Proceedings of the Conference Datenbanksysteme in Büro, Technik und Wissenschaft (BTW)*, 2017.

[100] Thorsten Papenbrock, Tanja Bergmann, Moritz Finke, Jakob Zwiener, and Felix Naumann. Data profiling with metanome. *PVLDB*, 8(12):1860–1863, 2015.

[101] Thorsten Papenbrock, Sebastian Kruse, Jorge-Arnulfo Quiané-Ruiz, and Felix Naumann. Divide & conquer-based inclusion dependency discovery. *PVLDB*, 8 (7):774–785, 2015.

[102] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[103] Otakar Pinkas. Automatic encoding and language detection in the GSDL. *Journal of Systems Integration*, 5(4):47–57, 2014.

[104] David Pinto, Andrew McCallum, Xing Wei, and W Bruce Croft. Table extraction using conditional random fields. In *Proceedings of the International Conference on Information retrieval (SIGIR)*, pages 235–242, 2003.

[105] Gil Press. Cleaning Big Data: Most Time-Consuming, Least Enjoyable Data Science Task, Survey Says. https://www.forbes.com/sites/gilpress/2016/03/23/data-preparation-most-time-consuming-least-enjoyable-data-science-task-survey-says/?sh=5cd029046f63, 2016. Accessed: 2021-06-18.

[106] Abdulhakim A Qahtan, Ahmed Elmagarmid, Raul Castro Fernandez, Mourad Ouzzani, and Nan Tang. FAHES: A robust disguised missing values detector. In *Proceedings of the International Conference on Knowledge discovery and data mining (SIGKDD)*, pages 2100–2109, 2018.

[107] Erhard Rahm and Hong Hai Do. Data cleaning: Problems and current approaches. *IEEE Data Engineering Bulletin*, 23(4):3–13, 2000.

[108] Sajjadur Rahman, Kelly Mack, Mangesh Bendre, Ruilin Zhang, Karrie Karahalios, and Aditya Parameswaran. Benchmarking spreadsheet systems. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 1589–1599, 2020.

[109] Theodoros Rekatsinas, Xu Chu, Ihab F Ilyas, and Christopher Ré. Holoclean: holistic data repairs with probabilistic inference. *PVLDB*, 10(11):1190–1201, 2017.

[110] Alexandra Rostin, Oliver Albrecht, Jana Bauckmann, Felix Naumann, and Ulf Leser. A machine learning approach to foreign key discovery. In *Proceedings of the ACM SIGMOD Workshop on the Web and Databases (WebDB)*, 2009.

[111] Yossi Rubner, Carlo Tomasi, and Leonidas J. Guibas. A metric for distributions with applications to image databases. In *Procedings of the International Conference on Computer Vision (ICCV)*, pages 59–66, 1998.

[112] Ritesh Sarkhel and Arnab Nandi. Visual segmentation for information extraction from heterogeneous visually rich documents. In *Proceedings of the International Conference on Management of Data (COMAD)*, pages 247–262, 2019.

[113] Philipp Schirmer, Thorsten Papenbrock, Sebastian Kruse, Felix Naumann, Dennis Hempfing, Torben Mayer, and Daniel Neuschäfer-Rube. DynFD: Functional dependency discovery in dynamic datasets. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 253–264, 2019.

[114] Yakov Shafranovich. Common Format and MIME Type for Comma-Separated Values (CSV) Files. https://rfc-editor.org/rfc/rfc4180.txt, October 2005. Accessed: 2021-07-04.

[115] Alexey O Shigarov and Andrey A Mikhailov. Rule-based spreadsheet data transformation from arbitrary to relational tables. *Information Systems (IS)*, 71:123–136, 2017.

[116] Eric Smalley. AI-powered drug discovery captures pharma interest. *Nature biotechnology*, 35(7):604–606, 2017.

[117] Carolin Strobl, Anne-Laure Boulesteix, Achim Zeileis, and Torsten Hothorn. Bias in random forest variable importance measures: Illustrations, sources and a solution. *BMC Bioinformatics*, 8(1):1–21, 2007.

[118] Carolin Strobl, Anne-Laure Boulesteix, Thomas Kneib, Thomas Augustin, and Achim Zeileis. Conditional variable importance for random forests. *BMC Bioinformatics*, 9(1):1–11, 2008.

[119] Sandeep Tata, Navneet Potti, James B Wendt, Lauro Beltrão Costa, Marc Najork, and Beliz Gunel. Glean: Structured extractions from templatic documents. *PVLDB*, 14(6):997–1005, 2021.

[120] John Alan Terrell. *An Experimental Investigation of a New System for Automatically Regulating the Voltage of an Alternating Current Circuit.* Number 9. Rensselaer Polytechnic Institute, 1913.

[121] Saravanan Thirumuruganathan, Nan Tang, Mourad Ouzzani, and AnHai Doan. Data curation with deep learning. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 277–286, 2020.

[122] Fabian Tschirschnitz, Thorsten Papenbrock, and Felix Naumann. Detecting inclusion dependencies on very many tables. *ACM Transactions on Database Systems (TODS)*, 42(3):18:1–18:29, 2017.

[123] Gerrit JJ van den Burg, Alfredo Nazábal, and Charles Sutton. Wrangling messy csv files by detecting row and type patterns. *Data Mining and Knowledge Discovery*, 33(6):1799–1820, 2019.

[124] Jovan Varga, Oscar Romero, Torben Bach Pedersen, and Christian Thomsen. Towards next generation BI systems: The analytical metadata challenge. In *Proceedings of the International Conference on Data Warehousing and Knowledge Discovery (DaWaK)*, pages 89–101, 2014.

[125] Petros Venetis, Alon Y. Halevy, Jayant Madhavan, Marius Pasca, Warren Shen, Fei Wu, Gengxin Miao, and Chung Wu. Recovering semantics of tables on the web. *PVLDB*, 4(9):528–538, 2011.

[126] Gerardo Vitagliano, Lan Jiang, and Felix Naumann. Detecting layout templates in complex multiregion files. *PVLDB*, 2021. (accepted).

[127] Alexander Wachtel, Michael T Franzen, and Walter F Tichy. Context detection in spreadsheets based on automatically inferred table schema. *International Journal of Computer and Information Engineering*, 10(10):1892–1899, 2016.

[128] Yalin Wang and Jianying Hu. A machine learning based approach for table detection on the web. In *Proceedings of the International World Wide Web Conference (WWW)*, pages 242–250, 2002.

[129] Yiheng Xu, Minghao Li, Lei Cui, Shaohan Huang, Furu Wei, and Ming Zhou. Layoutlm: Pre-training of text and layout for document image understanding. In *Proceedings of the International Conference on Knowledge discovery and data mining (SIGKDD)*, pages 1192–1200, 2020.

[130] Cong Yan and Yeye He. Auto-suggest: Learning-to-recommend data preparation steps using data science notebooks. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 1539–1554, 2020.

[131] Junwen Yang, Yeye He, and Surajit Chaudhuri. AutoPipeline: Synthesize data pipelines by-target using reinforcement learning and search. *PVLDB*, 14(11):2563–2575, 2021.

[132] Ekim Yurtsever, Jacob Lambert, Alexander Carballo, and Kazuya Takeda. A survey of autonomous driving: Common practices and emerging technologies. *IEEE access*, 8:58443–58469, 2020.

[133] Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J Franklin, et al. Apache spark: a unified engine for big data processing. *Communications of the ACM*, 59(11):56–65, 2016.

[134] Meihui Zhang and Kaushik Chakrabarti. InfoGather+: semantic matching and annotation of numeric and time-varying attributes in web tables. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 145–156, 2013.

[135] Meihui Zhang, Marios Hadjieleftheriou, Beng Chin Ooi, Cecilia M Procopiuc, and Divesh Srivastava. On multi-column foreign key discovery. *PVLDB*, 3(1-2):805–814, 2010.

## Selbstständigkeitserklärung

Ich erkläre hiermit, dass

- ich die vorliegende Dissertationsschrift selbständig und ohne unerlaubte Hilfe angefertigt sowie nur die angegebene Literatur verwendet habe,
- die Dissertation keiner anderen Hochschule in gleicher oder ähnlicher Form vorgelegt wurde,
- mir die Promotionsordnung der Digital Engineering Fakultät der Universität Potsdam vom 27. November 2019 bekannt ist.

Potsdam, den 14. Dezember 2021

Lan Jiang