Hasso Plattner Institute
Information Systems Group

Modeling the Structure of Tabular Files for Data Preparation

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
Gerardo Vitagliano

Dissertation, Universität Potsdam, 2023

Reviewers

Prof. Dr. Felix Naumann
Hasso-Plattner-Institut, Universität Potsdam

Prof. Dr. Sebastian Schelter
University of Amsterdam, Netherlands

Prof. Dr. Paolo Papotti
EURECOM, Campus SophiaTech, France

Published online on the
Publication Server of the University of Potsdam:
https://doi.org/10.25932/publishup-62435
https://nbn-resolving.org/urn:nbn:de:kobv:517-opus4-624351

Modeling the Structure of Tabular Files for Data Preparation

Gerardo Vitagliano

Hasso Plattner Institute
Information Systems Group

University of Potsdam, Germany

To manage tabular data files and leverage their content in a given downstream task, practitioners often design and execute complex transformation pipelines to prepare them. The complexity of such pipelines stems from different factors, including the nature of the preparation tasks, often exploratory or ad-hoc to specific datasets; the large repertory of tools, algorithms, and frameworks that practitioners need to master; and the volume, variety, and velocity of the files to be prepared. Metadata plays a fundamental role in reducing this complexity: characterizing a file assists end users in the design of data preprocessing pipelines, and furthermore paves the way for suggestion, automation, and optimization of data preparation tasks.

Previous research in the areas of data profiling, data integration, and data cleaning, has focused on extracting and characterizing metadata regarding the *content* of tabular data files, i.e., about the records and attributes of tables. Content metadata are useful for the latter stages of a preprocessing pipeline, e.g., error correction, duplicate detection, or value normalization, but they require a properly formed tabular input. Therefore, these metadata are not relevant for the early stages of a preparation pipeline, i.e., to correctly parse tables out of files. In this dissertation, we turn our focus to what we call the *structure* of a tabular data file, i.e., the set of characters within a file that do not represent data values but are required to parse and understand the content of the file. We provide three different approaches to represent file structure, an *explicit* representation based on context-free grammars; an *implicit* representation based on file-wise similarity; and a *learned* representation based on machine learning.

In our first contribution, we use the grammar-based representation to characterize a set of over 3000 real-world CSV files and identify multiple structural issues that let files deviate from the CSV standard, e.g., by having inconsistent delimiters or containing multiple tables. We leverage our learnings about real-world files and propose Pollock, a benchmark to test how well systems parse CSV files that have a non-standard structure, without any previous preparation. We report on our experiments on using Pollock to evaluate the performance of 16 real-world data management systems.

Following, we characterize the structure of files *implicitly*, by defining a measure of structural similarity for file pairs. We design a novel algorithm to compute this measure, which is based on a graph representation of the files' content. We leverage this algorithm and propose Mondrian, a graphical system to assist users in identifying *layout templates* in a dataset, classes of files that have the same structure, and therefore can be prepared by applying the same preparation pipeline.

Finally, we introduce MaGRiTTE, a novel architecture that uses self-supervised learning to automatically learn structural representations of files in the form of vectorial

embeddings at three different levels: cell level, row level, and file level. We experiment with the application of structural embeddings for several tasks, namely dialect detection, row classification, and data preparation efforts estimation.

Our experimental results show that structural metadata, either identified explicitly on parsing grammars, derived implicitly as file-wise similarity, or learned with the help of machine learning architectures, is fundamental to automate several tasks, to scale up preparation to large quantities of files, and to provide repeatable preparation pipelines.

Modellierung der Struktur von Tabellarische Dateien für die Datenaufbereitung

Gerardo Vitagliano

Hasso Plattner Institut

Universität Potsdam

Anwender müssen häufig komplexe Pipelines zur Aufbereitung von tabellarischen Dateien entwerfen, um diese verwalten und ihre Inhalte für nachgelagerte Aufgaben nutzen zu können. Die Komplexität solcher Pipelines ergibt sich aus verschiedenen Faktoren, u.a. (i) aus der Art der Aufbereitungsaufgaben, die oft explorativ oder ad hoc für bestimmte Datensätze durchgeführt werden, (ii) aus dem großen Repertoire an Werkzeugen, Algorithmen und Frameworks, die von den Anwendern beherrscht werden müssen, sowie (iii) aus der Menge, der Größe und der Verschiedenartigkeit der aufzubereitenden Dateien. Metadaten spielen eine grundlegende Rolle bei der Verringerung dieser Komplexität: Die Charakterisierung einer Datei hilft den Nutzern bei der Gestaltung von Datenaufbereitungs-Pipelines und ebnet darüber hinaus den Weg für Vorschläge, Automatisierung und Optimierung von Datenaufbereitungsaufgaben.

Bisherige Forschungsarbeiten in den Bereichen Data Profiling, Datenintegration und Datenbereinigung konzentrierten sich auf die Extraktion und Charakterisierung von Metadaten über die Inhalte der tabellarischen Dateien, d.h. über die Datensätze und Attribute von Tabellen. Inhalts-basierte Metadaten sind für die letzten Phasen einer Aufbereitungspipeline nützlich, z.B. für die Fehlerkorrektur, die Erkennung von Duplikaten oder die Normalisierung von Werten, aber sie erfordern eine korrekt geformte tabellarische Eingabe. Daher sind diese Metadaten für die frühen Phasen einer Aufbereitungspipeline, d.h. für das korrekte Parsen von Tabellen aus Dateien, nicht relevant. In dieser Dissertation konzentrieren wir uns die Struktur einer tabellarischen Datei nennen, d.h. die Menge der Zeichen in einer Datei, die keine Datenwerte darstellen, aber erforderlich sind, um den Inhalt der Datei zu analysieren und zu verstehen. Wir stellen drei verschiedene Ansätze zur Darstellung der Dateistruktur vor: eine explizite Darstellung auf der Grundlage kontextfreier Grammatiken, eine implizite Darstellung auf der Grundlage von Dateiähnlichkeiten und eine erlernte Darstellung auf der Grundlage von maschinellem Lernen.

In unserem ersten Ansatz verwenden wir die grammatikbasierte Darstellung, um eine Menge von über 3000 realen CSV-Dateien zu charakterisieren und mehrere strukturelle Probleme zu identifizieren, die dazu führen, dass Dateien vom CSV-Standard abweichen, z.B. durch inkonsistente Begrenzungszeichen oder dem Enthalten mehrere Tabellen in einer einzelnen Datei. Wir nutzen unsere Erkenntnisse aus realen Dateien und schlagen Pollock vor, einen Benchmark, der testet, wie gut Systeme unaufbereitete CSV-Dateien parsen. Wir berichten über unsere Experimente zur Verwendung von Pollock, in denen wir die Leistung von 16 realen Datenverwaltungssystemen bewerten.

Anschließend charakterisieren wir die Struktur von Dateien *implizit*, indem wir ein Maß für die strukturelle Ähnlichkeit von Dateipaaren definieren. Wir entwickeln einen neuartigen Algorithmus zur Berechnung dieses Maßes, der auf einer Graphen-basierten Darstellung des Dateiinhalts basiert. Wir nutzen diesen Algorithmus und

schlagen Mondrian vor, ein grafisches System zur Unterstützung der Benutzer bei der Identifizierung von *Layout Vorlagen* in einem Datensatz, d.h. von Dateiklassen, die die gleiche Struktur aufweisen und daher mit der gleichen Pipeline aufbereitet werden können.

Schließlich stellen wir MaGRiTTE vor, eine neuartige Architektur, die selbst-überwachtes Lernen verwendet, um automatisch strukturelle Darstellungen von Dateien in Form von vektoriellen Einbettungen auf drei verschiedenen Ebenen zu lernen: auf Zellebene, auf Zeilenebene und auf Dateiebene. Wir experimentieren mit der Anwendung von strukturellen Einbettungen für verschiedene Aufgaben, nämlich Dialekterkennung, Zeilenklassifizierung und der Schätzung des Aufwands für die Datenaufbereitung.

Unsere experimentellen Ergebnisse zeigen, dass strukturelle Metadaten, die entweder explizit mit Hilfe von Parsing-Grammatiken identifiziert, implizit als Dateiähnlichkeit abgeleitet oder mit Machine-Learning Architekturen erlernt werden, von grundlegender Bedeutung für die Automatisierung verschiedener Aufgaben, die Skalierung der Aufbereitung auf große Mengen von Dateien und die Bereitstellung wiederholbarer Aufbereitungspipelines sind.

Abstract

To manage tabular data files and leverage their content in a given downstream task, practitioners often design and execute complex transformation pipelines to prepare them. The complexity of such pipelines stems from different factors, including the nature of the preparation tasks, often exploratory or ad-hoc to specific datasets; the large repertory of tools, algorithms, and frameworks that practitioners need to master; and the volume, variety, and velocity of the files to be prepared. Metadata plays a fundamental role in reducing this complexity: characterizing a file assists end users in the design of data preprocessing pipelines, and furthermore paves the way for suggestion, automation, and optimization of data preparation tasks.

Previous research in the areas of data profiling, data integration, and data cleaning, has focused on extracting and characterizing metadata regarding the *content* of tabular data files, i.e., about the records and attributes of tables. Content metadata are useful for the latter stages of a preprocessing pipeline, e.g., error correction, duplicate detection, or value normalization, but they require a properly formed tabular input. Therefore, these metadata are not relevant for the early stages of a preparation pipeline, i.e., to correctly parse tables out of files. In this dissertation, we turn our focus to what we call the *structure* of a tabular data file, i.e., the set of characters within a file that do not represent data values but are required to parse and understand the content of the file. We provide three different approaches to represent file structure, an *explicit* representation based on context-free grammars; an *implicit* representation based on file-wise similarity; and a *learned* representation based on machine learning.

In our first contribution, we use the grammar-based representation to characterize a set of over 3000 real-world CSV files and identify multiple structural issues that let files deviate from the CSV standard, e.g., by having inconsistent delimiters or containing multiple tables. We leverage our learnings about real-world files and propose Pollock, a benchmark to test how well systems parse CSV files that have a non-standard structure, without any previous preparation. We report on our experiments on using Pollock to evaluate the performance of 16 real-world data management systems.

Following, we characterize the structure of files *implicitly*, by defining a measure of structural similarity for file pairs. We design a novel algorithm to compute this measure, which is based on a graph representation of the files' content. We leverage this algorithm and propose Mondrian, a graphical system to assist users in identifying *layout templates* in a dataset, classes of

files that have the same structure, and therefore can be prepared by applying the same preparation pipeline.

Finally, we introduce MaGRiTTE, a novel architecture that uses self-supervised learning to automatically learn structural representations of files in the form of vectorial embeddings at three different levels: cell level, row level, and file level. We experiment with the application of structural embeddings for several tasks, namely dialect detection, row classification, and data preparation efforts estimation.

Our experimental results show that structural metadata, either identified explicitly on parsing grammars, derived implicitly as file-wise similarity, or learned with the help of machine learning architectures, is fundamental to automate several tasks, to scale up preparation to large quantities of files, and to provide repeatable preparation pipelines.

Zusammenfassung

Anwender müssen häufig komplexe Pipelines zur Aufbereitung von tabellarischen Dateien entwerfen, um diese verwalten und ihre Inhalte für nachgelagerte Aufgaben nutzen zu können. Die Komplexität solcher Pipelines ergibt sich aus verschiedenen Faktoren, u.a. (i) aus der Art der Aufbereitungsaufgaben, die oft explorativ oder ad hoc für bestimmte Datensätze durchgeführt werden, (ii) aus dem großen Repertoire an Werkzeugen, Algorithmen und Frameworks, die von den Anwendern beherrscht werden müssen, sowie (iii) aus der Menge, der Größe und der Verschiedenartigkeit der aufzubereitenden Dateien. Metadaten spielen eine grundlegende Rolle bei der Verringerung dieser Komplexität: Die Charakterisierung einer Datei hilft den Nutzern bei der Gestaltung von Datenaufbereitungs-Pipelines und ebnet darüber hinaus den Weg für Vorschläge, Automatisierung und Optimierung von Datenaufbereitungsaufgaben.

Bisherige Forschungsarbeiten in den Bereichen Data Profiling, Datenintegration und Datenbereinigung konzentrierten sich auf die Extraktion und Charakterisierung von Metadaten über die Inhalte der tabellarischen Dateien, d.h. über die Datensätze und Attribute von Tabellen. Inhalts-basierte Metadaten sind für die letzten Phasen einer Aufbereitungspipeline nützlich, z.B. für die Fehlerkorrektur, die Erkennung von Duplikaten oder die Normalisierung von Werten, aber sie erfordern eine korrekt geformte tabellarische Eingabe. Daher sind diese Metadaten für die frühen Phasen einer Aufbereitungspipeline, d.h. für das korrekte Parsen von Tabellen aus Dateien, nicht relevant. In dieser Dissertation konzentrieren wir uns die Struktur einer tabellarischen Datei nennen, d.h. die Menge der Zeichen in einer Datei, die keine Datenwerte darstellen, aber erforderlich sind, um den Inhalt der Datei zu analysieren und zu verstehen. Wir stellen drei verschiedene Ansätze zur Darstellung der Dateistruktur vor: eine explizite Darstellung auf der Grundlage kontextfreier Grammatiken, eine implizite Darstellung auf der Grundlage von Dateiähnlichkeiten und eine erlernte Darstellung auf der Grundlage von maschinellem Lernen.

In unserem ersten Ansatz verwenden wir die grammatikbasierte Darstellung, um eine Menge von über 3000 realen CSV-Dateien zu charakterisieren und mehrere strukturelle Probleme zu identifizieren, die dazu führen, dass Dateien vom CSV-Standard abweichen, z.B. durch inkonsistente Begrenzungszeichen oder dem Enthalten mehrere Tabellen in einer einzelnen Datei. Wir nutzen unsere Erkenntnisse aus realen Dateien und schlagen Pollock vor, einen Benchmark, der testet, wie gut Systeme unaufbereitete CSV-Dateien parsen.

Wir berichten über unsere Experimente zur Verwendung von Pollock, in denen wir die Leistung von 16 realen Datenverwaltungssystemen bewerten.

Anschließend charakterisieren wir die Struktur von Dateien *implizit*, indem wir ein Maß für die strukturelle Ähnlichkeit von Dateipaaren definieren. Wir entwickeln einen neuartigen Algorithmus zur Berechnung dieses Maßes, der auf einer Graphen-basierten Darstellung des Dateiinhalts basiert. Wir nutzen diesen Algorithmus und schlagen Mondrian vor, ein grafisches System zur Unterstützung der Benutzer bei der Identifizierung von *Layout Vorlagen* in einem Datensatz, d.h. von Dateiklassen, die die gleiche Struktur aufweisen und daher mit der gleichen Pipeline aufbereitet werden können.

Schließlich stellen wir MaGRiTTE vor, eine neuartige Architektur, die selbstüberwachtes Lernen verwendet, um automatisch strukturelle Darstellungen von Dateien in Form von vektoriellen Einbettungen auf drei verschiedenen Ebenen zu lernen: auf Zellebene, auf Zeilenebene und auf Dateiebene. Wir experimentieren mit der Anwendung von strukturellen Einbettungen für verschiedene Aufgaben, nämlich Dialekterkennung, Zeilenklassifizierung und der Schätzung des Aufwands für die Datenaufbereitung.

Unsere experimentellen Ergebnisse zeigen, dass strukturelle Metadaten, die entweder explizit mit Hilfe von Parsing-Grammatiken identifiziert, implizit als Dateiähnlichkeit abgeleitet oder mit Machine-Learning Architekturen erlernt werden, von grundlegender Bedeutung für die Automatisierung verschiedener Aufgaben, die Skalierung der Aufbereitung auf große Mengen von Dateien und die Bereitstellung wiederholbarer Aufbereitungspipelines sind.

Acknowledgments

I would like to express my utmost gratitude to my advisor, Felix Naumann, for all his support and guidance. I learned first-hand from him what it means to be a passionate researcher and a caring advisor, the best I could have ever wished for. At all times, he supported me, recognized and valued my work, shared his knowledge and experience, and guided me with invaluable advice. He trusted me with the freedom to explore my own ideas, inspired me to develop as a researcher and as a person, and pushed me to always strive for greater goals. I hope our paths will keep crossing for many years to come, never drifting apart.

I am also thankful to Tilmann Rabl: his perspective and advice were always helpful and insightful, helping me to navigate through my own journey.

I am grateful to Eugene Wu, who hosted me at Columbia University and brought his unique insights and style to the Pollock project. His influence was refreshing and thought-provoking, a genuine source of inspiration.

I want to thank all the colleagues and friends at Hasso Plattner Institute that I had the pleasure to work with, exchange ideas, and share these past years with. Among them, I want to especially mention Ioannis Koumarelas, Lan Jiang, Mazhar Hameed, Alejandro Sierra-Múnera, Hazar Harmouch, and Michael Loster. These people gave me their support and comfort in the most challenging moments, shared my happiness in the best ones, and made my Ph.D. a remarkable experience. I would also like to acknowledge Lucas Reisener for his support on the Mondrian and Pollock projects.

Last but not least, I cannot forget to thank my loved ones and my friends. I feel grateful for their love, support, and patience that never left me throughout these years. I learned that research is a long and challenging journey, but *iff* one has good company, it is a wonderful one.

"Siempre estoy haciendo cosas que no puedo hacer,
así es como logro hacerlas." – *Pablo Picasso*

"I am always doing things that I cannot do,
that is how I achieve them." – *Pablo Picasso*

Contents

Chapter 1

Tabular Data Preparation

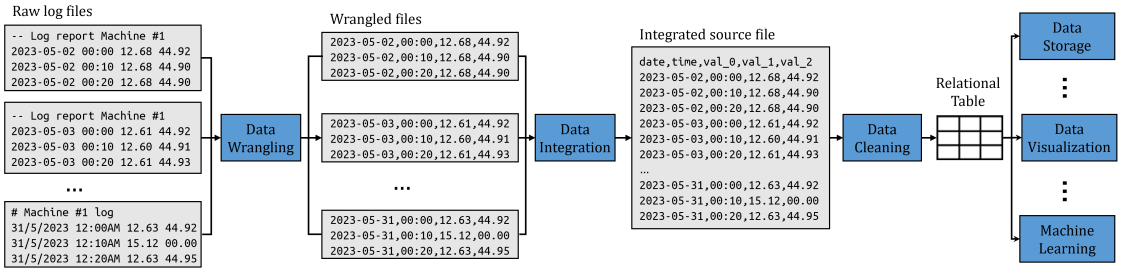

Figure 1: An example pipeline from raw tabular files to downstream tasks.

Data preparation commonly refers to a wide range of necessary operations on raw data to enable a given downstream task in a data-oriented workflow. Data engineers and practitioners often perform such operations by using different systems, tools, or ad-hoc scripts to produce desirable outputs. To illustrate a typical data-driven process, consider the following example: a team of data scientists has to analyze the anomalous behavior of motor appliances in a factory. The core task is a time-series analysis of the data produced in the last productive cycle, e.g., the last month, that is found in a set of tabular files stored in a data lake. An example of the data preprocessing pipeline is illustrated in Figure 1.

First and foremost, they need to collect and integrate data produced by different sensors connected to the machines, which in our example happen to correspond to a set of log files, produced each day, containing multiple tables, one per sensor. First, data engineers are required to split the individual tables based on the custom format of the files. To do so, they may use a custom Python script that detects the boundaries for the tables of each individual sensor, e.g., based on the presence of header or metadata rows.

Then imagine that, due to a software update that happened within the timeframe of the analysis, the newest files use a different format to store the date and time of each record. Before integrating the files, the practitioners first are required to resolve these inconsistencies. This may be done using custom scripts, or by using interactive data wrangling tools, such as Trifacta [110] or OpenRefine [37]. Once all records across different files follow the same format, they can integrate all the different files,

now containing a single table per sensor/day, in a single relational table. If the data integration process is carried out correctly, they can explore the data with custom notebooks, e.g., using Jupyter [60], or data analysis tools such as Excel [74]. To better understand the anomaly, they may have to clean the data, for example by detecting and/or correcting errors and null values; by transforming some of the attributes into more appropriate data types for higher precision; or by normalizing attributes to the range [0,1]. Once this stage is over, they might pivot the table to view average daily values of measurements across sensors and store this table in a relational database, e.g., MySQL [79]. Finally, they might use the refined data to train a machine learning model to detect or predict anomalies in the future and design an interactive Tableau [103] dashboard to visualize the data. As highlighted by the example, several preprocessing steps are required to leverage the information contained in raw data files for downstream tasks.

While some literature labels all preprocessing steps with the generic term data preparation [31, 55, 104, 122], regardless of the assumptions and requirements for each task, we distinguish what is performed *before* data loading, from what is performed *after*.

Specifically, all operations required to parse data from files are, by definition, data-agnostic and *structural*, since they are based on the format of the file and not on its content. We refer to these as *data preparation*. Typically, data preparation operations are tightly coupled to the systems or frameworks used in the pipeline and have the general goal of bringing raw data files into a format that can be loaded. In our example, data preparation steps include extracting multiple tables out of individual files, wrangling their records to resolve the inconsistencies in the date format, and integrating all the information into a single tabular file. These steps are typically carried out with ad-hoc scripts or interactive systems that are time-consuming and require user expertise.

Conversely, we define *data cleaning* as the preprocessing operations that operate on parsed data, and therefore require *semantic* understanding and domain knowledge. Data cleaning operations usually assume a specific data model, are tightly coupled to the tasks, and aim at improving the quality of tabular data for downstream applications. In our example, data cleaning steps are detecting and correcting errors in the sensor data, assigning the correct data types to the attributes of the integrated table, or normalizing its attributes to a given range.

Data management research focused on providing algorithmic solutions for structural tasks, like dialect detection [12], table extraction [63], or row classification [56]; as well as for semantic tasks, like error detection and repair [90], column type detection [127], or entity matching [82].

However, even considering state-of-the-art research solutions, preparing data is mostly an ad-hoc process, reliant on the experience of end users. In the remainder of the introduction, we motivate the need for principled and general frameworks. Then, we present a taxonomy of models proposed to represent tabular data from a semantic perspective, highlighting the value they serve for data management. Finally, employing such a framework we present the main contributions of this thesis and the structure of the dissertation.

1.1 Shortcomings of data preparation

The motivating example of the previous section highlights several critical points of data preparation:

- **System dependency:** Preparation steps are often tightly coupled to the systems used in the pipeline, which have diverse and heterogeneous requirements. Preparation may require user experience with the systems (e.g., with the interactive tools used for wrangling and analysis), or debugging unexpected errors (e.g., the sensors' software changing specification).
- **Task dependency:** Data pipelines are highly tailored for specific tasks: the workflow used to wrangle and integrate the sensor data into a single table for anomaly detection would not be useful for other tasks that require a different analysis.
- **Dataset dependency:** Tabular data may be found in a variety of formats with different accompanying metadata: in the example dataset, sensor files have multiple tables that have to be split according to specific metadata rows. Not all files follow specified standards, e.g., the RFC4180 for CSV [94], and custom formats are not always documented explicitly, forcing users to identify unique formats and design ad-hoc preparation pipelines.

As it stands, data preprocessing pipelines appear to be more similar to a sequence of band-aid fixes to specific problems, solved as they occur, rather than a set of well-engineered steps. We argue that to address the aforementioned challenges it is useful to frame data preprocessing in terms of file metadata, i.e., identifying the characteristics of the input data files, how they differ from those that are required, and what transformation(s) can bridge possible gaps.

Focusing on data preparation, we define as *file structure* the metadata that describes syntactical aspects of data files. Representing file structure can overcome the aforementioned shortcomings:

- **System dependency:** By making system requirements explicit, it is possible to proactively design the necessary preparation steps, not only based on previous end-user experiences or reactively, after discovering loading errors.
- **Task dependency:** By having clear methods to define and describe alternative file structures, it is possible to design standard preparation steps and adapt them to new tasks with minimal effort.
- **Dataset dependency:** By identifying similarities in the structure of files, data preparation pipelines can be designed in a more general way and applied to multiple datasets or files.

As with semantic models for tabular data, we do not believe there is a single, one-size-fits-all model of file structure that can be used in all scenarios. Our research aims at providing several models to represent, leverage, and transform file structure to support data preparation tasks. The next section presents a taxonomy to highlight different flavors and characteristics of existing semantic models. For each category, we briefly present recent related work highlighting their relevancy to data management tasks.

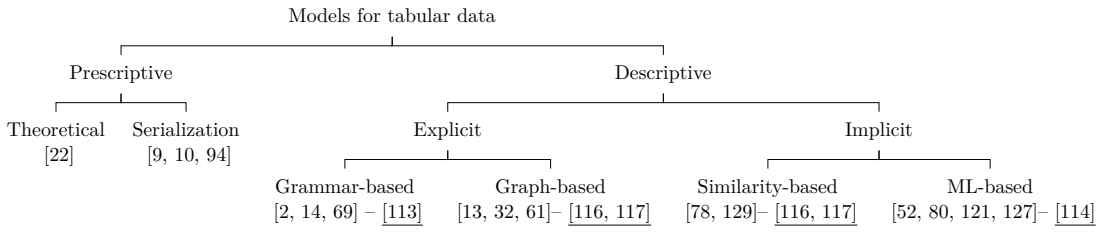

Figure 2: Taxonomy of models for data management, thesis contributions outlined.

1.2 Tabular models for data management

As one of the most prominent formats for structured data, tables have been the center of the attention of data management research for decades [125]. Therefore, numerous models and frameworks have been proposed to represent tabular data, with different focuses and goals. In this section, we use the term *tabular* in the most encompassing way, to include all data composed of a set of records, each with a set of attributes. We organize tabular models according to a novel taxonomy of two main groups (see Figure 2): *prescriptive* and *descriptive*.

Prescriptive models can be distinguished in theoretical models, which are based on logic and formal specifications, and serialization models, which are defined to serialize (or parse) actual files with a grammar and a set of rules. A prime example of the first category is the relational model [22], which influenced much of the data management field. A key motivation for the design of this model was to establish a separation between the logical and physical organization of data, to abstract applications from the underlying storage system. Examples of the second category are the standard specifications grammars of data file formats, such as CSV [94], JSON [9], or XML [10]. Beyond defining the rules to serialize files, these documents also determine the logical model of the data contained within files. For example, the CSV grammar prescribes data to be arranged in tables with a fixed set of attributes (the header), and a set of records with a value for each attribute. On the contrary, the JSON format models individual records as sets of attribute-value pairs, without the requirement for each record to have values for all attributes.

Prescriptive formal models are used a-priori, dictating how information is to be formalized as data. On the contrary, descriptive models are used a-posteriori, i.e., they are defined and applied to existing data, e.g., for tabular files, data lakes, or web pages. We distinguish descriptive models between *explicit* and *implicit*. Models from the former category serve as general frameworks to specify metadata, and they encompass grammar-based methods and graph-based methods. Examples of grammar-based models are the ones found in the works of Arenas et al. [2] and Martens et al. [69]. These works propose languages to describe CSV files that do not (necessarily) comply with the CSV standard, e.g., because they contain multiple header lines or attributes that span multiple columns. Inspired by these works, in [113], we propose a grammar-based model to represent the structure of tabular files described in Chapter 2. A different example of a descriptive grammar is the one proposed by Cetorelli et al. in [14], which aims at modeling tables found within HTML documents, with the goal of data extraction.

Graph-based models leverage graph representations to model relationships between various data elements. For example, in [32] the authors model datasets composed of many relations as hypergraphs, where nodes are attributes, edges connect related attributes, and hyperedges connect related nodes at different granularities. They leverage such representation to highlight relevant connections across datasets to perform data discovery. Similarly, the work of Cappuzzo et al. [13] proposes a model to represent relational datasets as tripartite graphs with nodes that represent all the values, records, and attributes found across the relations of the dataset. Then, these graphs are embedded as vectors and used to perform data integration. A different example is the work of Koci et al. [61], which proposes a graph-based model to represent tables within spreadsheet files, encoding regions of cells as nodes and their spatial relationship as edges. These graphs are then annotated to classify cell types such as header or data and extract relational tables. In Chapter 3, we propose a graph-based model to represent the structure of files that contain multiple tables [116, 117].

The final group of descriptive models are *implicit* models: they do not define explicit rules to represent tabular data, but rather build upon statistical methods that derive a representation from the data itself. We further categorize some implicit models as *similarity-based*, as their goal is to measure the similarity between data elements. For example, the model proposed by Nargesian et al. in [78] defines a similarity function between different relations based on statistical tests that measure the unionability of their attributes. This similarity is used to cluster relations and identify plausible join candidates. Zhang et al. in [129] design a wide set of similarity functions on relations, attributes, records, as well as provenance information and apply them to address four different tasks on tabular data: data augmentation, feature extraction, data cleaning, and data linking. Chapter 3 describes a similarity-based representation for collections of multiple spreadsheet files to identify reoccurring file layouts [116].

Finally, recent advances in machine learning research have led to several architectures that create implicit representations by embedding tabular data into vectorial spaces. We define this category of models as *machine learning-based*. For example, the work of Xu et al. [121] proposes the use of conditional generative adversarial networks (GAN) to learn a latent representation of relational tables. Similarly, the work of Park et al. [80] leverages GANs to represent tables and synthesize new data with a similar distribution, but with anonymized data. Other works, such as [52, 127], leverage transformer-based architectures pre-trained on natural language tasks to learn a representation of tabular records and attributes and propose their use for various tasks, including data cleaning, column type inference, or data augmentation. We propose a machine learning-based model [114] specifically focused on the structure of tabular files to address data preparation challenges, described in Chapter 4.

As the current state of data management research, a wide range of models have been proposed for tabular data, each with different characteristics and applications. However, all of these models focus on semantic aspects of tables, i.e., their data, while no model is explicitly designed to capture structural aspects, i.e., how they are represented. A similar taxonomy of models could be thought for representing the structure and syntax of files, rather than their semantic content. In our research we propose several models, which can be categorized according to the taxonomy in Figure 2, to highlight the features

of the files that are at the center of the data preparation efforts. Our efforts are also towards designing concrete systems to leverage such a representation to support data preparation tasks.

1.3 Dissertation structure and contributions

Our research is motivated by the observation that most data preparation efforts are driven by files with non-standard or inconsistent structures. The work presented in this thesis is focused on the representation of the structure of tabular data files and its applications to data preparation. We propose three descriptive models to represent the structure of tabular data files: an explicit grammar-based model [113], an implicit graph-based model [116], and a machine learning-based model [114, 115]. We apply each of these models to different datasets to address the challenges related to data preparation, including data loading, data discovery, table and metadata extraction, and data preparation effort estimation. Specifically, we make the following contributions:

A grammar-based model to benchmark data loading (Chapter 2)

We introduce a formal, grammar-based model of tabular file structure. Using this model, we present the results of a large-scale survey of over 3500 CSV files publicly available on open data portals, which identified multiple structural inconsistencies. Motivated by our survey, we propose Pollock [113], the first benchmark to evaluate how systems load tabular files with non-standard structures, if they are not previously prepared.

We apply our benchmark on 16 different real-world systems used at all stages of a data pipeline, scoring them and discussing their performances. On the one hand, these results can be used to inform the decision of end-users on which system to employ in their pipelines, possibly with respect to the structural characteristics of their files. On the other hand, the Pollock benchmark can guide system designers in the development of more refined data loading solutions. All data collected for the survey, the benchmark data, and all code to reproduce the analysis are publicly available[1].

This chapter is based on the work presented in [113], which is a joint work with Reisener, Hameed, Jiang, Wu, and Naumann. My own contribution to this work is the collection and experimental survey of the files, the formalization of the grammar-based model, the benchmark design and implementation, and the experimental analysis. Together with Reisener, I also contributed to the implementation of the benchmark on the 16 systems under test. Wu advised on the grammar-based model and contributed to the writing. Hameed and Jiang contributed to the design of the benchmark and of the experiments, and to the analysis of the results. Naumann advised on the design of the survey, the benchmark, the experiments, and contributed to the writing.

A graph- and similarity-based model to detect layout templates (Chapter 3)

In Chapter 3, we introduce a graph-based file structure model and the concept of *file templates*, sets of files with the same structure. We present a computer-vision algorithm to extract structural graphs and identify graph similarity to identify file templates within a dataset. We describe Mondrian, our interactive system that employs this algorithm

[1] `https://github.com/HPI-Information-Systems/Pollock`

to assist end-users in visualizing file structure and identifying templates, i.e., clusters of files with the same structure. Using the detected templates, users can define general preparation pipelines and apply them to several files at once.

Our approach is composed of three phases: first, each file is rendered as an image and inspected for elements that could form regions; then, using a clustering algorithm, the identified elements are grouped to form regions; finally, every file layout is represented as a graph and compared with others to find layout templates. We compare our method to state-of-the-art table recognition algorithms on two corpora of real-world enterprise spreadsheets. We also introduce an interactive web-based interface for Mondrian [117] for end-users to extract, visualize, and discover similarities in the structure of individual spreadsheet files within large-scale datasets.

This chapter is based on the work presented in [117] and [116], where I contributed to the design of the model and the implementation of the algorithm, the collection and annotations of the datasets, the experimental evaluation of its performances, and the design of the user-oriented system. In [117], Reisener contributed to the implementation of the graphical user interface, and Jiang, Hameed, and Naumann advised on system design and contributed to the writing. In [116], Jiang and Naumann advised on the design of the automated system, the experiments, and contributed to the writing.

A machine learning-based model to obtain structural embeddings (Chapter 4)

In Chapter 4, we introduce MaGRiTTE, a machine learning model to learn structural embeddings of tabular files, using an architecture based on transformers and convolutional networks. The model is pre-trained on a dataset of almost 1M real-world tabular files, in a self-supervised fashion, to reconstruct the special characters that make up the structure of the file, and to detect whether pairs of rows belong to the same file. Then, we present four strategies to employ the pre-trained model on the specialized data preparation tasks of dialect detection, row type classification, column type annotation, and data preparation effort estimation. The latter is a novel task, first introduced in our work, which aims at providing a numerical estimate of the effort required to prepare a given file. Our experiments show that these tasks can be solved with good performances, demonstrating the value of structural representations for data preparation. In our experiments, we also highlight the combined value of complementing structural and semantic representations by applying MaGRiTTE with other state-of-the-art models that specialize in semantic features.

This chapter is based on the work presented as an abstract in [114] and described in a full paper, currently under submission, where I contributed to the design and implementation of the machine-learning architecture, investigated and formalized the task of preparation effort estimation, and performed the experimental evaluation for four data preparation tasks. Hameed, Sierra-Múnera, and Naumann advised on the design of machine learning architecture, its application to the data preparation tasks, the experimental analysis, and contributed to the writing.

Finally, in Chapter 5 we conclude the thesis by summarizing our results and discussing future research directions.

Chapter 2

Pollock: a Formal Model to Benchmark Data Loading

The ability to load tables from tabular files is a fundamental requirement for many systems at play in data-driven pipelines. To assess the variety of file formats used to store tabular data, we analyzed 17 repositories of governmental data portals across six continents. Overall, these portals contain 784 062 available datasets, whose file type is reported in Table 1. The full results and the code to reproduce our analysis can be found in the project repository[1].

Excluding visual-oriented documents, such as HTML and PDF, a vast majority of structured data is found in CSV files. As previous research noted [12, 21, 75], real-world CSV files are often not following the official RFC standard [94], and are also lacking appropriate metadata [2, 69], which in turn makes end-user spend time in preparing these files to ensure a correct loading their content.

To automate data loading from non-standard CSV files, several algorithms have been proposed to address specific issues, such as dialect parsing [12, 28, 34], CSV line and cell classification [36, 56], or table extraction [21]. To evaluate to which degree these results have been transferred into real-world systems, and how much manual data preparation

Format	# datasets	% of total
HTML	326 446	41.63%
CSV	**245 594**	**31.32%**
PDF	151 053	19.26%
XML	128 452	16.38%
ZIP	67 024	8.54%
JSON	65 008	8.29%
Total	784 062	100.00

Table 1: Number of datasets by format in 17 governmental portals. One dataset can contain files with multiple formats.

[1] `https://github.com/HPI-Information-Systems/Pollock`

burden is still left to end-users, in this Chapter we propose Pollock[2], a data loading benchmark.

Pollock is based on a formal grammar-based model for the structure of tabular files that can be used (1) to unambiguously describe non-standard structures in real-world files; and (2) to systematically generate large-scale datasets of unprepared files with the corresponding clean ground truth.

To design the benchmark, we analyzed in detail 3 712 real-world CSV files and recorded the problems we encountered. We demonstrate the applicability of our benchmark by testing and scoring 16 different systems: popular CSV parsing frameworks, relational database tools, spreadsheet systems, and a data visualization tool. This chapter is based on our published work in [113]. Specifically, the contributions of this chapter are:

1. An explicit, grammar-based model to describe the format, content, and structure of data files.
2. A survey that analyzes the structural inconsistencies of 3 712 real-world CSV files, manually annotated.
3. A benchmark composed of an input standard CSV file and a set of 2 290 non-standard files, based on the results of the survey. The benchmark comprises several metrics and a set of weights that produce an aggregated score.
4. The experimental results of the application of our benchmark to 16 different systems to evaluate their loading capabilities.

We organize the discussion of the rest of the chapter as follows: In Section 2.1, we discuss the related research efforts that have been proposed to address the challenges of non-standard CSV files; in Section 2.2, we describe our framework to characterize data parsing grammars and introduce the concepts of pollution and structural difference of grammars; Section 2.3 presents our data loading benchmark, whose design is based on a survey of real-world CSV files; Section 2.4 reports the results obtained by 16 different systems on our benchmark and discusses their shortcomings; Section 2.5 concludes the chapter with a summary.

2.1 Challenges of CSV files

The ambiguous CSV file format is a known source of data loading problems. Previous work highlighted the challenges of parsing real-world CSV files. For example, Mitlöhner et al. encountered such problems in their survey of publicly available CSV files [75], where, out of 141 738 parsed CSV files, 36 912 (26.04%) were parsed with errors. They report different errors, such as non-standard dialects, incorrect file extensions, and multiple tables within a file.

Over the years, research has tried to address the unique challenges of CSV files: Döhmen et al. proposed the robust parser Hypoparsr [28]; van den Burg et al. focused on dialect detection for "messy" CSV files with CleverCSV [12]; other projects tried to tackle issues such as table recognition and cell classification in CSV files [21, 56, 116].

[2]Pollock, inspired by the abstract painter, stands for Polluted CSV benchmarK.

Although these issues are known to academia, no explicit formal model has been proposed to characterize syntactical issues and their relationship to the CSV grammar. The first and best-known document defining a standard is the RFC4180 [94] of 2005. This document already mentions the widespread use and lack of formal specification for CSV files and is presented as a consolidation of the most common CSV features encountered in practice, rather than an unambiguous standard. Ten years later, the W3C consortium formalized a "non-normative" document to establish a JSON model for tabular data on the internet [106]. Their model extends RFC4180 with stricter specifications of tabular structures and, perhaps more importantly, includes metadata to describe the structure and dialect of the file content itself. Nevertheless, file distributors do not apply the W3C recommendations and hardly ever distribute CSV file metadata. Most likely, this is due to the impracticality of distributing a curated data package contrasted to the ease of plain CSV files. Simultaneously, academia also started formal work on data serialization grammars: Arenas et al. designed a language focused on the metadata description of CSV files [2], while Martens et al. proposed SCULPT, a formal language to describe web tabular data [69]. These work describe "schema" languages to navigate the content of "CSV-like" files, where the structure of the file is not known in advance. Even though it is generally applicable to files with different dialects of CSV, it cannot be used to describe differences in the structure of their grammars. Rather, given the knowledge of a grammar, it provides a tool to reference and annotate content within a file. The focus of both frameworks is to annotate the content of a tabular file, rather than the grammar used to produce it. In contrast, we are interested in how different grammars may produce files with different syntactical properties (the structure) but the same content. Therefore, we aim at providing a framework that is not tied to any specific grammar or file format, but is applicable to any data serialization/parsing grammar. To leverage such a framework for benchmarking data loading, we are also interested in "generative" rather than "descriptive" models, that can be used to systematically generate files with a desired (non-standard) grammar, in a controlled fashion.

Examples of such models have been traditionally used in security testing, commonly known as grammar *fuzzing* frameworks [126]. Grammar-based fuzzing comprises a set of techniques to generate random inputs that are likely to induce bugs in software, using grammars to ensure the validity of the inputs [48, 98]. Recently, there has been some interest in applying fuzzing to benchmark the performance of data analytics workloads [128].

We propose a particular version of grammar-based fuzzing for data files in order to benchmark data loading. The need for such a benchmark has been recently acknowledged by researchers in discussing the state of real-world data preparation [66, 96]. In their survey, Hameed and Naumann compiled a set of common data preparation tasks and evaluated whether commercial tools for data preparation offered the respective functionalities [39]. Although the focus of that survey is on systems specifically designed for data preparation, they identified how significant data preprocessing was required on non-standard data files to enable loading data into said systems. For some individual tasks occurring in a data-driven pipeline, researchers have proposed specific benchmarks.

Poess et al. designed TPC-DI, a benchmark for data integration [83]. Its core includes files in heterogeneous formats containing information to feed a target decision support

system. The benchmark includes plain-text character delimited files in TXT and CSV format. Because its focus is on system throughput and performance at scale rather than robustness, these files follow the RFC4180 standard, and therefore TPC-DI is not fit to assess a system's data loading capability in the face of pollution.

Shah et al. focus on benchmarking the type-inference task in AutoML platforms [95]. Their work provides a reference labeled dataset of files usable for machine learning tasks, with a variety of commonly used data types. The benchmark evaluates the performance of an AutoML system by running the same machine learning model twice and comparing the results: once loading data with the correct data types (provided as ground-truth), and once loading data with automatic type inference. The task covered by that benchmark, aside from the specific focus on AutoML tools, is fundamentally semantic, while we are more interested in syntactic/structural loading, as correct structural loading is the prerequisite for any further operation.

In the next chapter, we introduce a grammar-based model for file structure to fulfill the characteristics highlighted in this section and bridge the gap towards benchmarking data loading at the structural level.

2.2 A grammar-based model of file structure

As previously mentioned, one of the fundamental obstacles to automated testing and benchmarking data loading is the lack of a formal model to describe the structural issues of CSV files and decouple them from semantic errors. A further challenge is the lack of annotated datasets of unprepared files with a clean ground truth. Creating such a dataset is an expensive task that requires a large amount of data, time, and domain expertise.

To address these challenges, we contribute Pollock, a formal framework to classify file issues with respect to serialization grammars, and its application to systematically generate large-scale datasets of unprepared files with the corresponding clean ground truth. Our framework formally defines the concepts of *content*, *structure*, and *format* of file grammars, and their *dialects*. We introduce the concept of *file pollution*, a systematic transformation of a file that modifies its parsing grammar into a *structurally different* version of it.

In the scope of our work, data files are actual textual files, collections of files, or strings in memory encoded with given grammars. Regardless of their differences, all grammars used for file serialization have similar characteristics: they specify what content is allowed, what rules are used to parse content from the file, and how to format the content in a given representation.

Our framework is based on context-free grammars, which we use to serialize content into files and parse content out of files. We note that our framework is not only applicable to all possible dialects of the CSV grammar, but also to any other data serialization grammars that are context-free, e.g., JSON or XML.

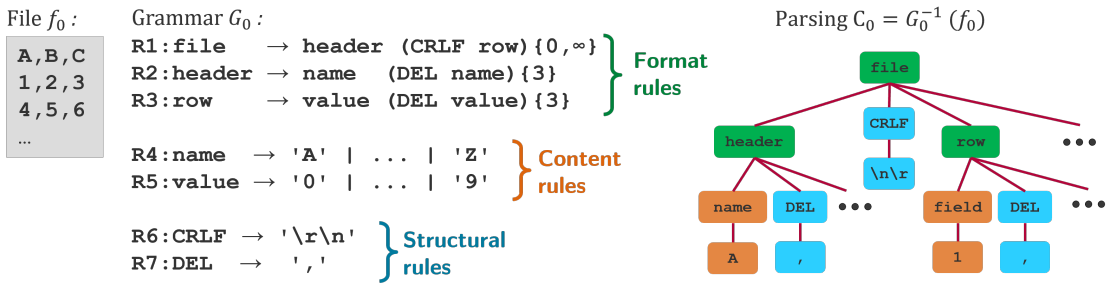

Figure 3: A sample CSV file f_0 and a grammar G_0 to parse it. Each node in the parse tree corresponds to a grammar rule.

2.2.1 Content, Structure, Format

A "data file" is a sequence of characters that expresses content in a given context-free grammar [18].

Definition 1 (Context-Free Grammar)**.** A context-free grammar G is a set of terminal symbols $\mathcal{T}$, a set of non-terminal symbols $\mathcal{V}$, a start symbol $V_s \in \mathcal{V}$, and a set of rules $\mathcal{R} : \mathcal{V} \times (\mathcal{V} \cup \mathcal{T})$.

Since data files may also contain metadata, depending on the application, we use the more general term *file* and refer to the payload of a file as *content*. We refer to *serialization* as the act of producing a file that encodes a content C using the rules of a specific grammar G: $f = G(C)$, and *parsing* as the act of extracting content from a file: $C = G^{-1}(f)$.

Consider the sample CSV file f_0 of Figure 3 and its grammar G_0 to describe it. G_0 is a simplified version of the standard one defined in the RFC4180 document [94]. The content of the file is a set of records containing the values ("`1,2,3`", "`4,5,6`", ...) for the attributes with the headers "`A,B,C`". Other characters found in the file, e.g., commas and newlines, constitute the structure of the file: they are used for parsing but do not belong to file content. Based on this intuition, we classify three types of symbols, and their corresponding rules: *content*, *structural*, and *format*.

Definition 2 (Content symbols and content rule)**.** In a grammar G, a rule $R \in \mathcal{R}$ and terminal symbols $T_i, T_j \in \mathcal{T}$, the set of content symbols is $\mathbb{C} = \{C \in \mathcal{V} \mid \exists\, R : C \to T_i \mid T_j,\ T_i \neq T_j\}$. We call R a content rule.

The above definition states that a rule is a content rule if it may resolve to multiple terminal symbols[3]. Because of this, they describe the objects of serialization, or *what* is allowed in a given file. We define the left-hand side symbol of a content rule as a content symbol. In the example of Figure 3, $R4$ and $R5$ are content rules.

Definition 3 (Structural symbols and structural rules)**.** In a grammar G, given a rule $R \in \mathcal{R}$ and a terminal symbol $T \in \mathcal{T}$, the set of structural symbols is $\mathbb{S} = \{S \in \mathcal{V} \mid \exists!\, R : S \to T\}$. We call R a structural rule.

[3]For notational simplicity, we excluded sequences of symbols. Conceptually, sequences of terminal symbols are equivalent to individual terminal symbols.

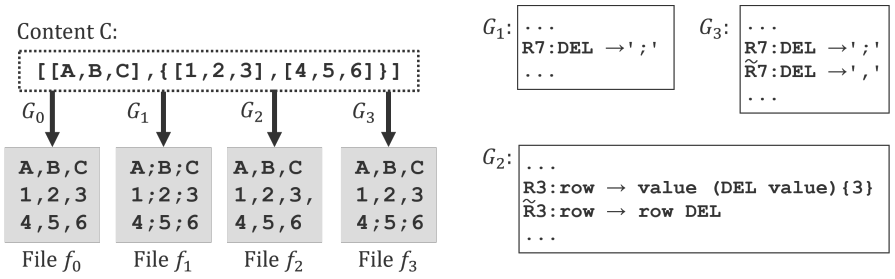

Figure 4: Four different files with equivalent content, serialized with four structurally different grammars. The grammars G_1, G_2, and G_3 all have the same rules of grammar G_0 (described in Figure 3), except for the rules explicitly reported.

The above definition states that a rule is a structural rule if it may only resolve to a unique terminal symbol (or sequence). In simple words, they pose as markers to identify *where* to find content in a given file. The left-hand side symbol of a structural rule is a structural symbol. In the example of Figure 3, $R6$ and $R7$ are structural rules.

Definition 4 (Format symbols and format rules)**.** In a grammar G, given a rule $R \in \mathcal{R}$ and non-terminal symbols $V_0, \ldots, V_n \in \mathcal{V}$, the set of format symbols is $\mathbb{F} = \{F \in \mathcal{V} \mid \exists R :\rightarrow V_0 \ldots V_n\}$. We call R a format rule.

The above definition states that a rule is a format rule if it resolves to a non-terminal symbol (or sequence). Format rules express *how* to combine content with structure in a given format. In the example of Figure 3, $R1$, $R2$, and $R3$ are format rules.

To express format rules with conciseness, we also introduce "symbol cardinality", a notation to specify the repetition of symbols.

Definition 5 (Symbol cardinality)**.** In a grammar G, given a rule $R \in \mathcal{R}$ containing a symbol $V \in \mathcal{V}$, symbol cardinality is the number of times V has to be repeated when applying rule R. Symbol cardinality is expressed by postfixing V with $\{m, n\}$, where $m, n \in \mathbb{N} \cup \{\infty\}$, signifying a repetition of a minimum of m to a maximum of n times. Brackets with a single number define a required cardinality of $m = n$. Lack of notation implies a cardinality of $m = n = 1$.

This notation can be used to express any grammar in Chomsky Normal Form (CNF) [19] (and therefore any CFG grammar) with more conciseness. As proof, suppose a format rule expressed as $R : F \rightarrow V_0 V_1 \{1, m\} V_2$ with a given maximum cardinality $m \neq \infty$. In normal form, non-terminal rules need to be in the form $A \rightarrow BC$: the rule R has to be expanded with $m + 1$ additional rules: $R_0 : F \rightarrow F_m V_2$, $R_1 : F_m \rightarrow F_{m-1} V_1$, $\ldots$, $R_m : F_1 \rightarrow F_0 V_1$, $R_{m+1} : F_0 \rightarrow V_0 V_1$. Formulating rules with symbols having an infinite cardinality in CNF is possible with the addition of an extra rule: for example, $R : F \rightarrow R_0 R_1 \{0, \infty\}$ has to be expressed with the two rules $R_0 : F \rightarrow R_0 F_1$, $R_1 : F_1 \rightarrow R_1 | F_1 R_1$.

Being equivalent to grammars in CNF format, our framework can be applied to any grammar used for serialization and parsing data from files.

2.2.2 Grammar dialects

Data serialization grammars are often regulated by standard specifications [9, 94]. However, real-world files often do not comply with standards. For example, one of four CSV files out of the 3 712 files we sampled in a real-world survey (see Section 2.3.3) uses a field delimiter different from comma (the prescribed RFC4180 standard). Consider the four exemplary files shown in Figure 4: they are all obtained by serializing the same content C, a header row followed by two data rows. The content of the first file can be parsed with the RFC4180-compliant grammar G_0 of Figure 3. All other files require slightly different grammars G_1, G_2, G_3 to be correctly parsed. Referring to our framework, all three grammars have the same content and format of G_0, but:

1. G_1 uses a different separator rule: `R7:DEL→';'`
2. G_2 allows rows with an extra delimiter: $\widetilde{\texttt{R3:}}$ `row → row DEL`
3. G_3 allows rows with different separators: $\widetilde{\texttt{R7:}}$ $\widetilde{\texttt{DEL}}$ → `';'`

Two context-free grammars are equivalent if they can serialize or parse the same sequences of tokens [81]. The grammars to parse the different files in Figure 4 are not strictly equivalent, because they differ in structural tokens or cardinalities. Still, they parse the same content from the four files. Regardless of their grammars, we define two files $f, \widetilde{f}$ as *content equivalent* if the content obtained parsing them with their respective grammars $G, \widetilde{G}$ is the same. Formally:

Definition 6 (Content equivalence)**.** Two files f and $\widetilde{f}$ parsed with the grammars G and $\widetilde{G}$, respectively, are content equivalent if $C = G^{-1}(f) = \widetilde{G}^{-1}(\widetilde{f}) = \widetilde{C}$.

In other words, given the parse trees $C = G^{-1}(f)$ and $\widetilde{C} = \widetilde{G}^{-1}(\widetilde{f})$, f and $\widetilde{f}$ are content equivalent if there exists a homomorphism between format and content symbols, and for format rules, all right-hand side symbols are found in the same order.

Definition 7 (Structurally different grammars)**.** A grammar G is structurally different from a grammar $\widetilde{G}$ if, given two content-equivalent files $f, \widetilde{f}$ with content C, the following hold:

1. $G \neq \widetilde{G}$
2. $f = G(C) = G(\widetilde{G}^{-1}\widetilde{G}(C))$
3. $\widetilde{f} = \widetilde{G}(C) = \widetilde{G}(G^{-1}G(C))$

Two grammars are (only) *structurally different* if they parse the same content from two different, yet content equivalent files.

Definition 8 (Grammar dialects)**.** Given a grammar G, its *dialects* are all grammars $\widetilde{G}$ structurally different from G.

In the example of Figure 4, grammars G_1, G_2, and G_3 are all dialects of G_0 (which is, in turn, a dialect of the RFC4180 CSV grammar).

2.2.3 File pollution

For a given standard grammar G, our goal is to benchmark how real-world systems load files serialized with different dialects of G. However, the set of dialects of a grammar G is infinite; and even for a single grammar G there are infinite possible files with different

contents. Consider the three reasons why a file $\widetilde{f}$ can differ from f: 1. $\widetilde{f}$ is expressed in the same grammar as f, but serializes a different content, i.e., $\widetilde{f} = G(\widetilde{C})$; 2. $\widetilde{f}$ serializes the same content as f but with a different grammar, i.e., $\widetilde{f} = \widetilde{G}(C)$; 3. $\widetilde{f}$ serializes a different content with a different grammar, i.e., $\widetilde{f} = \widetilde{G}(\widetilde{C})$.

To design a data loading benchmark, we are primarily interested in files that belong to (2), i.e., different files serializing the same content with different dialects of a grammar. However, to account for common issues in the wild (see Section 2.3), we also consider restricted cases of (3), where the content C of a file f is a strict subset of the content $\widetilde{C}$ of a file $\widetilde{f}$ (whose grammar may also be structurally different). We call *file pollution* the transformation of a file $f = G(C)$ into a content equivalent file $\widetilde{f} = \widetilde{G}(C)$, where $\widetilde{G}$ is a dialect of G.

We describe a simple procedure to construct file pollutions. Given a file f, rather than modifying its grammar G and then serializing a new file $\widetilde{f} = \widetilde{G}(C)$, we directly modify the parse tree $G^{-}1(f)$ in two ways: by changing structural symbols and by changing the cardinalities of symbols in format rules. These changes guarantee that the content of the file $\widetilde{f}$ has been serialized with a structurally different grammar $\widetilde{G}$, without the need to construct $\widetilde{G}$ explicitly. Given a file f and its parse tree $C = G^{-1}(f)$, we can systematically enumerate all the possible file pollutions. A pollution can: 1. change any of the structural symbols S with a different symbol $\widetilde{S}$, or 2. increase or decrease the cardinality of a symbol V in a format rule.

Our formalization of pollution offers several advantages. First, structural differences that characterize a resulting dialect are well-defined; second, they can be chosen at design time as a parameter; third, as the pollution is a controlled transformation, a ground truth content is available to evaluate the results of loading. Of course, the space of pollutions is still large, so it is unclear how to sample pollutions to concretely instantiate in a data loading benchmark. Given our framework, the problem of designing a data loading benchmark can be formalized with the following problem statement:

Problem Statement: *Given a source file f that serializes content C with a grammar G, find pollutions that generate a set of files $\widetilde{f_0}, \ldots, \widetilde{f_k}$, such that the content C_i of every file f_i is equivalent to C (or a strict superset) and is serialized with a grammar $\widetilde{G_i}$, dialect of the original grammar G.*

We also aim at sampling *relevant* pollutions, i.e., pollutions that replicate non-standard features that are likely to be found in real-world CSV files. We address these challenges by surveying 3 712 real-world CSV files and analyzing the occurrence of structural differences between their grammar and the standard CSV grammar.

In the next section, we describe the results of our survey and the pollutions we sampled from it.

2.3 The Pollock benchmark

In this section, we describe Pollock, a benchmark for CSV data loading that results from the application of grammar-based pollution as introduced in Section 2.2.3. To formally define a benchmark using the pollution framework, we need to specify:

1. A reference grammar G.
2. A source file f that serializes a content C using the grammar G, serving as the basis for the pollution operations.
3. For each of the format rules and structural rules in G, a set of pollutions to obtain different dialects $\widetilde{G}$.
4. Metrics to measure how well a system loads polluted files.

The core idea of Pollock is to systematically replicate real-world dialects. To do so, we synthetically generate different polluted versions of a single input file f, denoted $\widetilde{f}_1, \widetilde{f}_2, \cdots \widetilde{f}_n$, each serialized with a different dialect of the standard grammar G.

We ground the design of pollutions on a survey of 3 712 publicly available real-world files. With the results of the survey, we sample the space of possible pollutions and design a representative input file f to be polluted. We deliberately isolate pollutions to precisely benchmark their effect on data loading. We acknowledge that in practice, real-world files may deviate from the standard CSV grammar with several pollutions interacting at once. Moreover, we also observed loading issues in files serialized with the standard RFC4180 grammar due to system-specific assumptions (e.g., a maximum length for cell values). To gain insights on loading real-world files, in Section 2.4 we also benchmark different systems with a random sample of the survey files, guaranteed to contain all pollutions at least once.

2.3.1 Survey setup

We aim to benchmark CSV data loading. Therefore, the Pollock reference grammar G is the standard RFC4180 grammar for CSV files [94]. Figure 5 presents a formulation of this grammar according to our framework. Cardinalities should be treated as constants for a given file: for example, rules F3 and F4 specify that the header and record rows all have the same number N of cells.

We surveyed a sample of 3 712 real-world files marked as CSV: 2 274 CSV files randomly sampled from the Mendeley Data portal [72] and 1 438 randomly sampled files from the open data portal of the United Kingdom government [24].

The first is a public repository of scientific projects, where researchers can share research artifacts, such as code, data, and experimental results. We crawled all 2 214 projects that, at the time of our survey (July 2021), contained at least one file whose MIME type was "text/*". Out of more than 34 000 files contained in these projects, we retained all 2 274 files with a ".csv" extension.

The files selected from the UK government open data have been crawled from all datasets stored in the portal at the time of the experiments, retrieving a total of 17 851 files marked with the "text/CSV" MIME type, out of which we randomly sampled 1 438 files. For all 3 712 survey files, we manually annotated whether their grammars follow the RFC standard and, if not, which rules differ in their dialect. The collected files with their annotations can be found online[4].

[4] `https://github.com/HPI-Information-Systems/Pollock`

Format rules:

F0: file = table CRLF {0, 1}
F1: table = (header CRLF){0, 1} data
F2: data = record (CRLF record){0, ∞}
F3: header = cell (COMMA cell){N, N} CRLF
F4: record = cell (COMMA cell){N, N}
F5: cell = QUOTE (escaped){0, ∞} QUOTE
F6: cell = text text{0, ∞}
F7: escaped = COMMA | ESCAPE QUOTE | CRLF | text

Content rules:

C0: text = 0x20-21 | 0x23-2B | 0x2D-7E | ε

Structural rules:

S0: CRLF = 0x2C 0x0A
S1: COMMA = 0x2C
S2: QUOTE = 0x22
S3: ESCAPE = 0x22

Figure 5: RFC4180 standard grammar for CSV files.

2.3.2 Input file design

To design the input file for our pollutions, we analyzed the 3 712 survey files regarding their general characteristics. Out of all files, 15 are empty, i.e., they have no content and a dimension of 0 bytes: in the following analysis, we exclude these files. The remaining files contain a total of 46 474 823 rows and 296 602 columns. The minimum number of rows per file is 1, with the maximum being 9 505 531 rows. The distribution of rows per file is highly skewed, with an average of 11 981.14 rows per file but a mode of 2 and a median of 84. Regarding columns, the minimum number of columns is also 1, with the maximum being 34 804. As for columns, the average is 76.46 columns per file, but the mode and median are both at 9 columns per file.

To gain further insights into the data types of the columns, we automatically detected a data type for each one. We use the regular expression-based type detection proposed in the CleverCSV project [12], which classifies cells into one of twelve data types. To classify columns, we detect the type of each of the column cells and record the most frequently occurring type for each column. We further divide the string column type into three types: "short string" if all values in a column are under 100 characters, "long string" if any of the cells is longer than 100 characters, and "fixed length" if all values in a column have the same number of characters, e.g., code identifiers.

Table 2 reports the statistics for each of the column data types. The table reports the number of columns for which CleverCSV was unable to detect a data type, roughly 2% of the total. We note the high number of empty columns in the surveyed files: overall, 1 244 files contain at least one empty column. However, the high number of empty columns is caused by a tiny fraction of files that have an unusually large amount of trailing empty columns. For example, one of the files contains 19 non-empty columns, and 16 383 empty

Data type	# col.	% total	Data type	# col.	% total
Number (digits)	129 531	43.672%	Datetime	165	0.056%
Empty	121 992	41.130%	Percentage	141	0.048%
String (long)	34 285	11.559%	Number (float)	130	0.044%
String (fixed)	1 466	0.494%	Email	103	0.035%
Date	730	0.246%	Time	94	0.032%
String (short)	694	0.234%	Unix path	4	0.001%
URL	261	0.088%	Undetected	6 706	2.261%

Table 2: Column data types in survey files.

```
DATE,TIME,Qty,PRODUCTID,Price,ProductType,"ProductDescription","URL",Comments
28/01/2018,00:00,2,RO-1003,74.69, Men's Waterproof ...,"These water...","http...",
29/01/2018,00:15,0,BX-1011,29.81, Light-Up Running ...,"The next le...","http...",
30/01/2018,00:30,1,BX-1014,80.08, Men's Ventilated ...,"Great grip ...","http...",
31/01/2018,00:45,1,BX-1015,25.55, Switch Fly Rods     ,"This lightw...","http...",
01/02/2018,01:00,9,CC-1021,48.00,"Throw Pillow, Wo...","Add a pop o...","http...",
02/02/2018,01:15,1,CC-1022,34.22, Men's Heavy-Duty... ,"These tough...","http...",
03/02/2018,01:30,2,BX-1031,89.34, Organic Textured... ,"All the sof...","http...",
04/02/2018,01:45,2,MB-1032,19.34,"Cycling Jersey, ...","Designed wi...","http...",
05/02/2018,02:00,0,MB-1034,5.39 ,"Men's Silk Under...","For strong,...","http...",
18/02/2018,05:15,1,RO-1001,90.99,"Kids' Mountain B...","An easy-to-...","http...",
...
```

Figure 6: A subset of the content of the source file of our benchmark. For clarity, columns are visually aligned, and the content of some cells is trimmed.

columns after the last non-empty one. We note that 119 044 columns, 97.58% of all empty columns, are trailing empty columns in a file. These trailing empty columns affect 954 files (25.54% of the total files).

The survey files contain 111 340 columns (38.40% of the total) with at least one quoted cell. We analyzed the distribution of quoted cells inside these columns: in 37 833 columns, only fewer than 10% of their cells are quoted, and 66 275 columns have more than 90% of their cells quoted. This distribution is highly bimodal as the combined two cases cover 93.50% of the total quoted columns, and it reflects two different styles of handling quotation: in the former, only cells that require quotation in a column are quoted ("minimal" style); in the latter, all cells of a column are quoted regardless of need ("holistic" style).

Considering the results of our survey, we design the source file as a CSV file named "source.csv", with 9 columns and 84 lines – one header row and 83 data rows, for a total of 756 file cells. A sample of the source file can be seen in Figure 6, while the full file is available in the project repository.

The number of rows and columns is chosen as the median of the survey files. The rows in the file represent products sold from an online shop at a given time. Overall, the nine columns represent the most frequent data types we encountered in our survey:

- **DATE** expressed as DD/MM/YYYY, with the column containing unambiguous values wrt. day and month (e.g., 28/01/2018).
- **TIME** represents a time of the day. The format used is HH:MM and the values increase the time from 00:00 in steps of 15 minutes.
- **PRODUCTID** contains a fixed-length alphanumeric code.
- **Qty** is a non-negative integer number.
- **Price** contains a currency value, expressed with the US dollar sign and a positive floating-point number with a full stop as a decimal delimiter and two significant digits.
- **ProductType** contains a short string (under 100 characters) in natural language. This column contains quoted cells and escaped characters and is quoted "minimal" style.
- **ProductDescription** contains a long string (above 100 characters) with a natural language description of the products. This column also contains quoted cells and escaped characters and is quoted "holistic" style.
- **URL** contains a sample URL and is quoted "holistic" style.
- **Comments** is a trailing empty column, simulating optional information regarding a given product.

Although the results of Table 2 show that numeric columns in the form of digits are more frequent than other data types and that many files contain numerous trailing empty columns, we design our file to contain one numeric column and one trailing empty column - in an effort of sampling a broader spectrum of data types. We also note that, while we run our experiments of Section 2.4 with this input file, the Pollock pollutions can be applied to any input file that follows the standard CSV format.

2.3.3 Pollution design

Not all files of our survey can be parsed correctly using the standard CSV grammar. Here we report, for each format and structural rule of the RFC grammar (see Figure 3), all different variations of the rules required to parse the real-world files of the survey. For the scope of our benchmark, we include a single pollution type to cover each of these variations individually, even if the dialect of a single survey file might have several. A single pollution may have different possible parameters wrt. a file, e.g., a single-row pollution may apply to any row of a file. To generate polluted files, we first identify and isolate all pollution types affecting real-world files, and then we generate a benchmark file for each possible parameter, e.g., each row, column, or cell. Covering all combinations proved necessary from our experiments: in fact, as discussed in Section 2.4.3, we found certain systems to be sensitive to some pollutions only when they happened in specific cells.

In sum, our benchmark includes 2 290 polluted files. For every pollution type found in our survey, we report how many real-world files were affected and how (many) benchmark files (*Pollock files*) represent this pollution. The list of pollution types and the number of benchmark files are summarized in Table 3.

Currently, our framework does only generate files isolating one pollution at a time. While we acknowledge that combining several pollutions to generate more challenging files would lead to more realistic testing scenarios, we refrain for several reasons. First,

Table 3: Overview of pollutions with respect to the RFC4180 standard grammar.

Grammar rule	# Generated polluted files
F0: file= table CRLF {0,1}	3
F1: table = (header CRLF){0,1} data	7
F2: data = record (CRLF record){0,∞}	2
F3: header = cell (COMMA cell){N,N} CRLF	17
F4: record = cell (COMMA cell){N,N}	1 411
F5: cell = QUOTE (escaped){0,∞} QUOTE	756
S0: CRLF = 0x2C 0x0A	2
S1: COMMA = 0x2C	88
S2: QUOTE = 0x22	1
S3: ESCAPE = 0x22	2

applying several pollutions on the same file requires a notion of dependency to avoid interactions where different pollutions cancel (or alter) each other's effect. Moreover, we aim at providing end-users and system developers with a clear understanding of the data loading behavior of a SUT, while combining several pollutions would make it harder to disentangle the effects of each pollution and analyze the benchmark results. Finally, from our practical experimental experience, it would be time-consuming to run the benchmark as the number of files would increase exponentially (see Table 7). Studying the extension of our framework to create more complex pollutions is an interesting research problem discussed in Section 2.5.

F0: file format

The rule F0 of the grammar specifies that a file is composed of a table with an optional newline sequence CRLF. In our survey, we encountered:

- 15 empty files, with no table;
- 184 files with no trailing newline;
- 3 508 files with one trailing newline;
- 5 files with more than one trailing newline. All these files end with two newlines.

In the following analysis, we exclude the 15 empty files but retain one empty file among the benchmark files.

Pollock files:

- 1 empty file;
- 1 file without a trailing CRLF;
- 1 file with two trailing CRLF.

F1: table format

A table of a standard CSV file is composed of a single optional header line and data. Our survey found:

- 2 751 files with one header line;
- 470 files with no header;
- 476 files with multiple header lines.

Of the files with multiple header lines, 94 contain multi-row table headers spanning two or three lines. The other 282 files contain multiple *preamble* lines: rows with comments or metadata separated from the true table header with at least one empty line, i.e., a line with only separators and no content. Finally, 188 files contain multiple tables. In these files, there are two (or more) sections with header and data that mark different sections of the file content, at times with preamble lines or multiple header lines.

Pollock files:

- 1 file without a header;
- 2 files with multiple header lines (2 and 3 lines);
- 1 file with a preamble line;
- 3 files with two tables: one where both have the same number of columns, one where the second has more columns than the first, and one where the second has fewer columns than the first.

F2: data format

According to rule F2, CSV files contain data arranged in rows, each row containing a record. In our survey, we encountered:

- 3 files with no records but only a header row;
- 4 files with only a single record;
- 3 690 files with multiple records.

Pollock files:

- 1 file with only the header row;
- 1 file with a header and a single data row.

F3,F4: header and record format

The RFC4180 grammar requires that header and record rows have the same number of cells (see Rules F3 and F4 in Figure 3). We did not encounter files where the header does not terminate with a newline sequence. Regarding the number of cells, in our datasets we encountered

- 2 657 files with a consistent number of cells;
- 1 040 files with an inconsistent number of cells.

The number of cells can be inconsistent for different reasons: 221 files have preamble header lines with a different number of separators, some files have multiple tables in them (see above), with different column counts, others have data records with schema drift, where missing or extra cells are present in a subset of the records.

Pollock files:

- 17 files with an inconsistent header: one with a missing column separator for each of the 8 header separators, 9 with an extra separator before each column;

- 1 411 files with an inconsistent row: 664 with a missing column separator for each of the 8 column separators in the 83 data rows, 747 with an extra column separator before each of the 9 columns in all data rows.

F5: Cell format

A cell inside a row can contain any sequence of characters: however, if this sequence contains any of the tokens of the rules S0,S1,S2 of Figure 5, the cell has to be enclosed in quotation characters (see Rule F5). The "reserved" tokens correspond to the structural characters required to separate rows (CRLF), delimit columns inside rows (COMMA), and the quotation character itself (DQUOTE). The quotation character must be escaped with an extra quotation character to disambiguate it from the end of the quoted cell.

We encountered seven files with an incorrectly quoted cell where a quotation mark was not escaped. We note that other pollutions related to quoting and escaping are harder to identify under this scope without explicit domain knowledge, but they are possible to identify with respect to other format rules. For example, a cell containing a newline or extra separator character without a quote would lead to a record having a different number of columns, a problem we identified in the previous analysis of files under rule F4.

Pollock files:

- 756 files with incorrectly quoted cells, adding one unescaped quotation mark in each of the cells.

S0: newline sequence

The RFC4180 defines the newline sequence to be the combination of the carriage return (CR) and line feed (LF) characters. In our survey, we encountered:

- 1 999 files with the sequence of both CR and LF;
- 1 691 files with the only LF character;
- 7 files with only the CR character.

Pollock files:

- 2 files with a non-standard newline sequences in every row, one using CR-only and one using LF-only.

S1: cell delimiter

The standard character to delimit cells of a record is COMMA (see S1 in Figure 3). Nonetheless, it is common for CSV files to have a different delimiter, e.g., due to different locale specifications for floating-point numbers. In the survey dataset:

- 2 754 files use a comma delimiter;
- 834 files use a semicolon delimiter;
- 101 files use a comma plus whitespace or tab character as the delimiter;
- 8 files use a tab or a sequence of white spaces as delimiters.

Among the files using comma as a delimiter, 12 files have some of their rows delimited with sequences of white spaces. These inconsistent rows typically contain metadata regarding the table contained in the file, such as preamble or footnote lines.

Pollock files:

- 4 files with a non-standard delimiter in every row, one using semicolon, one using tab, one using whitespace, one using comma+whitespace;
- 84 files with an inconsistent delimiter in a single row, one using whitespace for each file row.

S2: quotation character

The second structural token specified by the RFC4180 grammar is the quotation character DQUOTE, defined to be the double quotation character (see S2 in Figure 3). In our survey, we identified only two different characters used for this marker:

- 1 596 files do not have any quoted cell;
- 2 090 files use the double quote character;
- 11 files use the apostrophe character.

We note that in the survey files using a different quotation character, no quote is found inside a cell and requires escaping. However, a different quotation character should also be accompanied by a different escaping sequence—following the RFC rules, a doubling of the quotation character.

Pollock files:

- 1 file with a non-standard quotation character in every row (using apostrophe, also escaping any apostrophe in the cells with an extra apostrophe).

S3: escape character

The last structural token in the CSV grammar is ESCAPE, the character used to escape a quotation character when contained in the value of a quoted cell. The RFC standard defines it to be the same as the DQUOTE character (see S3 in Figure 3). This sequence rarely occurs and often leads to errors or inconsistencies if files or parsers do not adhere to the standard. In our experiments, only 2 systems out of 16 can correctly load the whole content of files with a polluted escape quote, with the others either dropping the content of the cells following the escape character or of the whole row altogether.

Of the 2101 files with quoted cells,

- 1 849 files do not contain cells with escaped values;
- 250 files contain cells with values escaped according to the RFC standard;
- 2 files contain cells without any escape sequence.

Even if not observed in our survey, we also note that a common non-standard escaping strategy is to preclude double quotes inside a cell with a backslash symbol (Ux5C).

Pollock files:

- 1 file with a non-standard escape character in every cell (the backslash symbol);

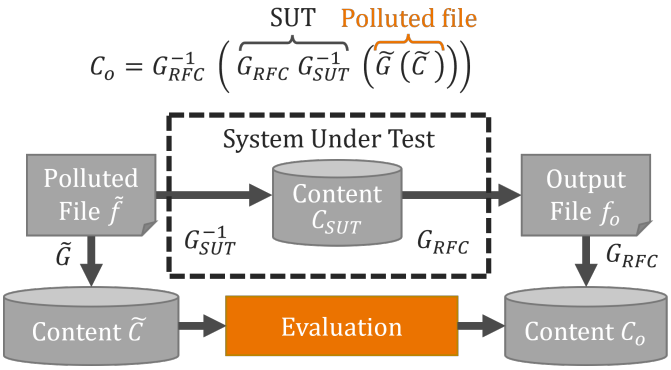

Figure 7: Summary of the benchmarking process.

- 1 file where quotations are not escaped.

2.3.4 Metrics design

Formally, given an input file f that encodes a content C with a standard grammar G, a single pollution obtains a different file $\widetilde{f} = \widetilde{G}(C)$, encoding the same content with a formally equivalent grammar. Once we obtain a polluted file, we aim at benchmarking how a given system under test (SUT) parses and loads it in memory.

Any SUT parses the content of its input files with a given grammar G_{SUT}, its parsing algorithm, which is generally not accessible to end-users. Also, every system has different in-memory representations of a content $C_{SUT} = G_{SUT}^{-1}(f)$. However, all tested systems that can load CSV input files are also capable of exporting content in an output file f_o encoded with the standard RFC4180 grammar: $f_o = G_{RFC}(G_{SUT}^{-1}(f_i))$.

At first, it would seem straightforward to compare the content of the input file f with the content of the output file f_o. However, in some pollutions it would not lead to fair measurement. In fact, a polluted file $\widetilde{f}$, which is the input to the SUT, may contain slightly different content than the source file f. This behavior is possible with pollutions that edit format rules by deleting content, e.g., to simulate a file with no header row; or by adding content in the input file, e.g., to simulate multiple tables in files. Therefore, it cannot be expected that a system loads content not present in the input file due to deletions from pollutions. Similarly, a correct loading should also include any extra input data not in the original source file but which was introduced by pollutions.

To address this, in measuring systems' performances we cannot use the content of the original source file. However, the polluted content $\widetilde{C}$ can be parsed from the polluted file $\widetilde{f}$ using the polluted grammar $\widetilde{G}$, which is known by design at benchmarking time. Therefore, we compare if the content parsed with the RFC grammar from the output file of a SUT, $C_o = G^-1_{RFC}(f_o)$, is equivalent to the polluted content parsed from the input polluted file $\widetilde{C} = \widetilde{G}^{-1}(\widetilde{f})$.

Figure 7 summarizes our approach to benchmark a system's loading of a polluted file in a general, SUT-independent fashion. We load a polluted file $\widetilde{f}$ in a SUT, which parses it with an unknown grammar G_{SUT}. We then export it back using the standard RFC grammar and compare the contents parsed from the input files using

the polluted grammar $\widetilde{C} = \widetilde{G}^{-1}(\widetilde{f})$, and the content parsed from the system output file $C_o = G_{RFC}^{-1}(G_{RFC}(G_{SUT}^{-1}(\widetilde{G}(C))))$. To compare contents independently of a specific internal system's representation, we normalize the output of individual cells to compare their values. The normalization parses dates and numbers and transforms all string characters to lowercase. For example, two cells containing the same date in two different formats are considered equivalent. To measure the equivalence of two parsed contents, we refer to the hierarchy induced by the format rules of the grammars (see Section 2.2.3). Following the RFC standard grammar (see Figure 5), we identify four content groups: a file is composed of (1) a table; a table is composed of (2) a header and (3) a set of records; records are composed of (4) cells. For the first level, we use a binary measure: *success (S)*. If a file is loaded correctly without any application error, we assign a value of 1 to this score, otherwise a 0. In case of a data loading with a success of 0, meaning the system aborted loading due to some error, we assign a value of 0 to all other scores.

However, even if a system successfully loads a polluted file, the resulting content may still differ from the expected content. A system may either miss some content while loading a file, e.g., excluding a polluted row, or include "spurious" content, e.g., by padding a row with unwanted cells. Therefore, we also use precision to evaluate the loading *completeness*, and recall to evaluate the loading *conciseness*, combining them both into the F1 score.

Given an input set of elements I and an output set of elements O, precision (P), recall (R), and F1 are defined as usual:

$$P = \frac{|(I \cap O)|}{|I|} \qquad R = \frac{|(I \cap O)|}{|O|} \qquad F_1 = 2 \cdot \frac{P \cdot R}{P + R}$$

To obtain a well-rounded score, we compute these metrics at the header, record, and cell level:

1. *Header precision (H_P), recall (H_R), F1 (H_{F_1})*: These metrics are computed on header cells and are necessary because systems often have separate assumptions regarding header and data rows. They measure the effect of pollutions on file headers, e.g., if an extra header column is added because a single data row contains an extra cell.
2. *Record precision (R_P), recall (R_R), F1 (R_{F_1})*: These metrics are computed for each data record, defined as the string hash of its cell values. They capture whether individual records are loaded coherently, or their content is split, merged, or rearranged within different records, e.g., if due to a missing escape character two data rows get merged into one.
3. *Cell precision (C_P), recall (C_R), F1 (C_{F_1})*: These metrics are computed on individual data cells and are the most fine-grained. They identify data errors regardless of their position in the output file, e.g., if a value gets lost due to a missing delimiter.

The range of all scores is $[0, 1]$, with 1 representing a perfect data loading.

The nature of our benchmark is to isolate, whenever possible, different pollutions and test systems on loading files with each of them separately. As such, every system can be benchmarked with respect to a single pollution, and a single dimension. However, we also aim at providing a unified *Pollock score* for every SUT that measures its data loading performance across all different pollutions. To do so, we average all scores obtained

across different polluted files, plus the scores obtained on the source file, and then sum them to obtain a single number per score.

To provide an additional and more realistic score, we weigh the average by the occurrence of the pollution in the real world, as identified by the survey of Section 2.3.1. The weights are normalized, to sum up to 1. In the case of pollutions that replicate a single pollution systematically (e.g., for every row, cell, or column), we scale the weights by the number of repetitions. For example, considering that in our survey 12 files had inconsistent row delimiters, and we repeat the pollution for each of the 84 rows of the source file, the metrics of each polluted file will weight 12/84 in the final average. Considering that every score has a range of $[0, 1]$ and that there are a total of 10 different scores for each polluted file, the maximum Pollock score obtainable by a system under test is 10.

2.4 Experimental results of real-world systems

To demonstrate the usage and usefulness of our benchmark, we experimented by applying it to a set of diverse real-world systems. By evaluating their data loading capabilities, we highlight their shortcomings and simultaneously assess the usefulness of Pollock. As listed in Table 4, we selected 16 systems of four tool-categories to highlight our benchmark's versatile nature and to analyze possible differences in the handling of file pollutions at different stages of a data preparation pipeline. We experimented with:

- Eight CSV parsing frameworks for programming languages
- Four relational database management systems
- Three systems designed for spreadsheet data analysis
- One business intelligence/data visualization tool

We chose programming frameworks for three popular programming languages: Python, R, and Java. For all Python modules, we used Python version 3.10.5. We benchmark the native "csv" module [111], referred to as PyCsv, which is used to read and write csv files; the Pandas module [89], designed for data analysis and manipulation; and CleverCSV [12], a module developed specifically to address loading "messy" CSV files.

For all R modules, we used R version 4.2.1. We benchmark the native "read csv" function [85], referred to as RCsv; and the specialized Hypoparsr algorithm [28], introduced in a research paper to perform "advanced" csv parsing.

For Java modules, we used OpenJDK 11. We benchmark the parsing libraries "Apache CSV Commons", referred to as CSVCommons [1], OpenCSV [97], and Univocity [4]. For those that allowed it, we resorted to automated parameter detection. In other cases, we manually specified suitable parsing parameters, if it was possible to do so. For database systems, we benchmarked four open-source RDBMS: MySQL [79], MariaDB [68], PostgreSQL [38], and SQLite [47]. Due to the nature of relational database systems, loading a file requires creating a table with the correct schema first. To do so, we specify all data types of such a table to be of TEXT or VARCHAR type, as our benchmark is concerned with file structure and not with type detection of files. In Section 2.1, we note how benchmarking type detection relates to data loading. For spreadsheet systems, we benchmarked LibreOffice Calc [107], an open-source desktop system, referred to as

	Preamble lines	Multirow header	Missing header	Newline sequence	Delimiter	Quotation	Escape
CleverCSV 0.7.4 [12]				Auto	Auto	Auto	Auto
CSVCommons 1.9.0				Manual	Manual	Manual	Manual
Hypoparsr 0.1.0 [28]				Auto	Auto	Auto	Auto
OpenCSV 5.6	Manual				Manual	Manual	Manual
Pandas 1.4.3	Manual	Auto	Auto	Manual	Auto	Manual	Manual
PyCsv 3.10.5					Auto	Auto	Auto
RCsv 4.2.1	Manual		Manual	Auto	Auto	Auto	Auto
Univocity 2.9.1		Auto		Auto	Auto	Auto	Auto
MariaDB 10.9.3	Manual		Manual	Manual	Manual	Manual	Manual
MySQL 8.0.31	Manual		Manual	Manual	Manual	Manual	Manual
PostgreSQL 15.0			Manual		Manual	Manual	Manual
SQLite 3.39.0	Manual			Manual	Manual		
Calc 7.3.7					Manual	Manual	
SpreadDesktop	Manual		Manual		Manual	Manual	
SpreadWeb					Manual		
DataViz					Manual	Manual	

Table 4: Configurations of the benchmarked systems. "Auto" stands for automatic detection, "Manual" for manual specification, a missing entry marks the lack of a configurable option.

Calc, a commercial desktop system referred to as SpreadDesktop[5], and an online tool referred to as SpreadWeb[5]. Lastly, we benchmarked a commercial data visualization tool, referred to as DataViz[5]. We note that we ran "best effort" experiments, using every applicable configuration option offered by each tool. Table 4 synthetically reports the loading configurations used for each tool. In the project repository[6], we share the results obtained by all systems and the scripts used to benchmark all non-commercial systems.

In the remainder of this chapter, we present an overview of the most interesting and surprising findings, based on the results of our benchmark, grouped by the rules of the grammar affected by the different pollutions we presented in Section 2.3. For simplicity, in this dissertation we only report F1 scores. For every subsection, we include takeaways for end-users, highlighting which problems require adequate preparation to correctly load the benchmark files, and for the developers of the systems under test (or other data loading systems), to identify opportunities for improvement.

[5]Anonymized due to licenses that forbid disclosing benchmarking results.
[6]`https://github.com/HPI-Information-Systems/Pollock`

Table 5: Systems with imperfect loading of the source file (RFC4180 compliant): success and F1-scores.

	S	H_{F1}	R_{F1}	C_{F1}	Loading time (ms)
Hypoparsr 0.1.0 [28]	1.00	0.00	0.11	0.63	3 277.11 ± 94.66
OpenCSV 5.6	1.00	1.00	0.98	0.99	12.72 ± 0.48
PyCsv 3.10.5	1.00	1.00	0.92	0.99	14.29 ± 3.08
DataViz	1.00	0.77	0.00	0.77	18 569.75 ± 592.11

2.4.1 Source file

Before analyzing the effect of pollutions on data loading, we assessed how systems handle the original source file. All systems are successful in opening this RFC4180 compliant file, but unfortunately not all of them load header, rows, and cells correctly. The results of these systems can be seen in Table 5. Among parsers, Hypoparsr is the only one unable to detect the header correctly, parsing it as a data row and appending a new header to the file, and also unable to detect the structure of rows containing cells with escaped commas and double quotes. PyCsv and OpenCSV both coincidentally fail in the same row, which contains the special symbol "\" and a delimiter: PyCsv considers the backslash symbol as an escape for the character that follows and ignores it in the resulting cell value; OpenCSV splits the cell into two at the delimiter character, even if the cell itself was properly enclosed in quotation marks.

DataViz loads all records erroneously because all values of the *TIME* column, which represents an absolute time in the HH:MM format, are transformed into the values "30/12/1899 HH:MM:00" (HH:MM standing for the original values in the input file).

User takeaways: When loading otherwise standard files with a programming framework, be aware of special symbols usually reserved in the programming language, such as "\": the framework may require extra escaping on top of what is required for the RFC standard. When loading files in more advanced systems, such as those developed for business intelligence, prepare the file such that its data types are compatible with the system.

Developer takeaways: Parsing content of a CSV file, the RFC4180 standard rules should take precedence over those of the language: the value of cells should be interpreted first as a byte string, and then parsed and/or escaped to a more refined data type. Data type parsing should inform users of its "confidence", perhaps defaulting to the raw cell value when confidence is low.

2.4.2 File and table pollution

Our benchmark contains 12 files that are polluted at the file and table level, i.e., serialized with non-standard file, table, and data rules (see Rules F0, F1, F2 in Table 3). The left columns of Table 6 report the results on these files: some systems fail to load them altogether. Notably, the Python parsers PyCsv and Pandas, along with RCsv, SpreadDesktop, and DataViz abort while loading an empty file, while all other systems

	File and table pollution (12 files)				Inconsistent number of delimiters (1 428 files)				Structural character change (850 files)			
	S	H_{F1}	R_{F1}	C_{F1}	S	H_{F1}	R_{F1}	C_{F1}	S	H_{F1}	R_{F1}	C_{F1}
CleverCSV [12]	1.00	0.75	0.91	0.91	1.00	0.99	1.00	0.99	1.00	0.93	0.57	0.74
CSVCommons	0.75	0.50	0.74	0.74	1.00	0.99	1.00	0.99	0.10	0.10	0.10	0.10
Hypoparsr [28]	1.00	0.35	0.30	0.53	1.00	0.07	0.07	0.44	1.00	0.26	0.16	0.69
OpenCSV	1.00	0.75	0.90	0.91	1.00	0.99	0.98	0.99	0.10	0.10	0.10	0.10
Pandas	0.91	0.67	0.85	0.85	1.00	0.99	0.98	0.99	0.99	0.99	0.97	0.98
PyCsv	0.91	0.66	0.78	0.82	1.00	0.99	0.92	0.99	1.00	0.99	0.92	0.98
RCsv	0.91	0.58	0.44	0.79	1.00	0.99	0.83	0.98	0.95	0.94	0.49	0.61
Univocity	1.00	0.75	0.91	0.91	1.00	0.99	1.00	0.99	0.99	0.99	0.98	0.99
MariaDB	1.00	0.75	0.98	0.90	1.00	1.00	0.98	0.88	1.00	0.99	0.97	0.88
MySQL	1.00	0.75	0.98	0.90	1.00	1.00	0.98	0.88	1.00	0.99	0.97	0.88
PostgreSQL	0.50	0.33	0.49	0.37	0.01	0.01	0.01	0.01	0.01	0.01	0.00	0.00
SQLite	1.00	0.66	0.99	0.91	1.00	1.00	1.00	1.00	1.00	0.99	0.98	0.99
Calc	1.00	0.74	0.91	0.90	1.00	0.99	1.00	0.99	1.00	0.98	0.98	0.98
SpreadDesktop	0.91	0.74	0.83	0.74	1.00	0.99	1.00	0.99	0.99	0.98	0.98	0.98
SpreadWeb	1.00	0.74	0.91	0.86	1.00	0.99	1.00	0.94	0.99	0.97	0.97	0.91
DataViz	1.00	0.46	0.16	0.64	1.00	0.73	0.00	0.73	1.00	0.57	0.00	0.48

Table 6: Results (rounding down) of the 16 SUTs, grouped by pollution type.

correctly load it. Interestingly, when the input file shows two trailing newline sequences, PostgreSQL halts due to the presence of empty values in the "time" column — although this error was not thrown while loading the standard source file.

After loading, no system can correctly recognize multiple header rows or preamble rows, even those that claim to perform automatic header detection. When no header is present, some systems load data rows with missing cells: e.g., SpreadDesktop, Calc, RCsv, and DataViz drop the empty column. When multiple tables are present, all systems that successfully load them either remove the extra column from the second table, if the first contains more; or add an extra column to the first table if the second contains more.

User takeaways: All systems show high sensitivity to proper "tabular" formatting of files. No matter the system of choice, and its promised automation level, perform the following preparations: condense header lines into one; remove preamble lines; split multiple tables into separate files.

Developer takeaways: Many systems still lack the support for non-standard headers, preamble lines, and multiple tables. For manual loading, we advise implementing interfaces to ignore or specify which rows are to be considered the header, and which ones are to be considered the data rows. The same interface can also be used to load a multi-table file without the need to split it into separate files. For automated loading, the most common strategy is to give a higher weight to the first few rows, which then influences how the remainder of the file is parsed. Apart from integrating existing research algorithms to detect row classes, multiple tables, and headers [21, 56, 116], we recommend that developers update the existing algorithms with contextual information.

2.4.3 Structural characters and inconsistent rows

The remaining files of the benchmark are polluted with structural changes and inconsistent rows. These include file-wise pollutions, and row-wise pollutions. The former affects all rows, i.e., by changing one of the structural characters across the entire file (see Rules F0, F1, F2 in Table 3). The latter affects individual rows, making them inconsistent with the rest of the file, either by having a different number of delimiters, or by having a different structural character. We apply these pollutions to every cell in the file. This repetition is necessary for fine-grained evaluation: in fact, while e.g., Hypoparsr, incorrectly loads all cells and rows after a misquoted value, other systems, e.g., OpenCSV or SpreadWeb, are more robust and only err on the affected cell/row.

The system with the worst performance is PostgreSQL: it is successful only in loading files where the header has an inconsistent number of delimiters, but if any of the data rows is inconsistent either with an extra or a missing delimiter, it halts the data loading operation. The other database systems are more robust to rows with inconsistent delimiters, loading the record but shifting all cells and/or trimming extra ones. Surprisingly, CSVcommons aborts the loading, but only for the file where the separator is missing from the last header column. For most systems, the headers are not affected by extra delimiters in a data row, except for DataViz, which always includes an extra header cell even if a single data row has an extra separator — leading to an H_{F1} score of 0.57.

Observing the results of Table 6, this set of files proves to be the least successful across many CSV parsing systems for different reasons: CSVcommons and OpenCSV fail to load any file with an extra quotation mark in one of the rows. As for RCsv, its behavior changes depending on the row affected by the quotation mark: if it is in one of the cells of the header row, it appends all the first data row to the cell but parses the other cells correctly and loads the file. If the extra quotation mark is found in one of the cells of the first four data rows, it halts loading with an error reporting an inconsistent number of delimiters. Otherwise, loading is carried out successfully, but several rows are merged into one, hence the low C_{F1} score. Pandas is unsuccessful with a single file: the one where an extra delimiter is present in the last column of the last row.

Curiously, SpreadWeb's only unsuccessful loading is with the file containing an extra quote in row 35. Univocity and SQLite are unable to load a file whose rows terminate with the only carriage return character — a pollution that does not affect any other system's loading capabilities. However, even if their loading does not abort, not all systems can manage inconsistent delimiters and extra quotation characters — apart from the aforementioned cell parsing issues of RCsv, OpenCSV, and Hypoparsr also have a low C_{F1}. These systems all merge the content of subsequent cells, often from multiple rows, if an inconsistent quote or delimiter is found in a given cell.

We note that the more robust systems appear to be PyCsv, Pandas, SQLite, and the spreadsheet systems Calc, SpreadDesktop, and SpreadWeb. As observable by the high C_{F1} and R_{F1} scores, the majority of parsing errors are limited to the rows affected by the pollution, while the remaining rows are parsed correctly.

	Pollock score (2 289 +1) files		Average file-wise time (milliseconds)
	Simple	Weighted	
CleverCSV 0.7.4 [12]	**9.19**	**9.45**	69.96 ± 0.13
CSVCommons 1.9.0	**6.64**	**9.25**	23.96 ± 7.64
Hypoparsr 0.1.0 [28]	**3.88**	**4.37**	6 040.15 ± 8.22
OpenCSV 5.6	**6.63**	**7.74**	18.50 ± 2.37
Pandas 1.4.3	**9.89**	**9.43**	1.39 ± 0.17
PyCsv 3.10.5	**9.72**	**9.43**	13.15 ± 0.13
RCsv 4.2.1	**7.79**	**6.40**	8.29 ± 0.61
Univocity 2.9.1	**9.93**	**7.93**	3.16 ± 0.19
MariaDB 10.9.3	**9.58**	**7.48**	20.96 ± 0.05
MySQL 8.0.31	**9.58**	**7.48**	63.96 ± 1.15
PostgreSQL 15.0	**0.13**	**6.96**	13.59 ± 0.28
SQLite 3.39.0	**9.95**	**9.37**	353.81 ± 22.54
Calc 7.3.7	**9.92**	**7.83**	2 646.06 ± 14.28
SpreadDesktop	**9.92**	**9.59**	28 776.18 ± 14.28
SpreadWeb	**9.72**	**9.43**	2 949.76 ± 16.29
DataViz	**5.00**	**5.15**	24 411.52 ± 292.67

Table 7: Pollock scores (rounding down) and average runtime of the 16 SUTs.

2.4.4 Overall Pollock score

Table 7 reports the Pollock score of all systems under test. We report two scores: one as a simple average, and one weighted by the occurrence of pollutions in our real-world survey, therefore depicting a more realistic scenario, as explained in Section 2.3.4. Different scoring schemes may serve different purposes: although end-users may be interested in the weighted score, to assess real-world performance, parser developers may want to more easily identify "hard" cases, to correct critical bugs.

As a reminder, the score is obtained by summing up 10 different numbers in the $[0, 1]$ range. These numbers correspond to success and precision, recall, and f1 scores at the header, record, and cell levels. Therefore, the maximum score reachable by any system is 10, in both the unweighted and the weighted case.

The last columns of Table 7 report the average file-wise loading-time of the benchmark files. The measurements were obtained by repeating our benchmark three times on a consumer machine equipped with an Intel i7 CPU with 2.20GHz and 16 GB of RAM. We explicitly warn readers to not compare systems across different categories: for example, the conditions under which an RDBMS loads files into a table are much more restrictive and require more specification parameters from end-users (e.g., defining the expected table format), than, for example, automated frameworks.

User takeaways: Among specialized CSV parsing modules, the ones with the highest scores, and the fastest are Pandas and Univocity. We attribute this result to these systems' development maturity: having been in use for a long time and by a large community, they

include safeguards against many pollutions. Comparing different languages, all Python frameworks have good results, while Java frameworks have an overall worse loading performance.

Among databases, SQLite has the best benchmark score, but also the highest average loading time. Interestingly, PostgreSQL shows the highest difference between the simple and the weighted scoring schemes: for almost all pollutions with an inconsistent row or cell, its loading failed altogether with a success of 0 (as can be noted by the central columns of the table). Although these pollutions constitute many files, they are also infrequent, which causes a higher weighted score.

Spreadsheet systems generally have good performance, probably thanks to their maturity and long-time use in a variety of scenarios. However, they are also among the ones with the highest loading times: their user-interfaces make it more cumbersome to load large datasets composed of several files, due to all the interactions needed to specify a correct loading. Finally, business intelligence tools, such as DataViz, are tailored towards cleaner, closer-to-standard data files. Its low score is influenced by an excess of intelligent pre-processing that fails with data not in line with the tool's expectations, for example, incorrectly parsing the "TIME" column of the source file, casting it to a "DATETIME" type with an arbitrary date (30/12/1899). Therefore, before using such tools we advise users to prepare their files, not only up to the CSV standard but also regarding their data types.

Developer takeaways: The lowest scores and the highest loading times among programming frameworks are obtained by systems whose file loading already proved unreliable even for the standard source file itself, e.g., for Hypoparsr and OpenCSV. Within the RDBMS category, the lowest scores are caused by highly restrictive assumptions regarding file structures. For example, PostgreSQL's low success rate is due to halting loading even if a single record is unrecognized by the system. We advise RDBMS developers to be more flexible when loading csv files, and perhaps offer users the option to skip polluted records rather than the whole file, as other benchmarked systems do, e.g., Pandas.

Regarding user-oriented tools, such as spreadsheets and business intelligence systems, one direction of improvement is the inclusion of more sophisticated automated detection strategies. This would improve usability as well as loading time for files, without the need to manually specify parameters through user interfaces.

2.4.5 Real-world loading

To gain further insight into the loading of real-world files, we tested the different systems with a sample of 100 survey files, which were manually cleaned row by row to provide ground truth for the measurements. We provide the sample and cleaned versions of the files on the Pollock page[7]. The sample was chosen at random, while ensuring that each pollution was represented in at least one of the sampled files. The results of the experiment are reported in Table 8.

[7] `https://github.com/HPI-Information-Systems/Pollock`

	S	H_{F1}	R_{F1}	C_{F1}	**Pollock score**	Loading time (ms)	
CleverCSV 0.7.4 [12]	1.00	0.70	0.96	0.95	**8.89**	840.55 ±	2.23
CSVCommons 1.9.0	0.46	0.26	0.43	0.42	**3.85**	297.81 ±	18.47
Hypoparsr 0.1.0 [28]	1.00	0.51	0.27	0.64	**5.43**	2 288.23 ±	15.67
OpenCSV 5.6	0.98	0.78	0.94	0.93	**9.01**	168.65 ±	5.92
Pandas 1.4.3	0.88	0.49	0.63	0.64	**6.28**	8.70 ±	0.26
PyCsv 3.10.5	0.98	0.67	0.88	0.87	**8.33**	176.82 ±	13.50
RCsv 4.2.1	0.97	0.24	0.52	0.58	**5.05**	25.14 ±	22.56
Univocity 2.9.1	0.95	0.40	0.61	0.63	**5.92**	60.38 ±	1.91
MariaDB 10.9.3	0.70	0.67	0.49	0.61	**6.13**	40.92 ±	9.96
MySQL 8.0.31	0.68	0.64	0.47	0.59	**5.89**	200.62 ±	17.90
PostgreSQL 15.0	0.54	0.51	0.53	0.53	**5.30**	12.00 ±	0.26
SQLite 3.39.0	1.00	0.65	0.73	0.90	**7.96**	342.02 ±	139.91
Calc 7.3.7	1.00	0.44	0.47	0.60	**5.60**	3 358.68 ±	460.75
SpreadDesktop	0.98	0.79	0.53	0.80	**7.41**	28 090.21 ±	51.80
SpreadWeb	0.98	0.68	0.60	0.81	**7.31**	4 846.62 ±	1265.19
DataViz	0.98	0.48	0.11	0.77	**5.15**	28 702.13 ±	294.54

Table 8: Results on a sample of 100 files from our survey.

As can be seen from the generally lower scores obtained by all systems, real-world files are more challenging to load for several reasons. Considering the variety of errors, we refrain from providing an extensive rundown of all failures, but identify some of the key reasons that lead to imperfect loading. Automatic detection of parameters does not work well in files affected by multiple pollutions at once, such as Pandas delimiter detection failing to recognize a semicolon delimiter for files with inconsistent numbers of delimiters in rows. In other cases, systems have restricted assumptions regarding otherwise standard CSV files, such as PostgreSQL failing to load files with duplicate or missing header names. Finally, some systems fail to scale with file dimensions, for example, Calc fails to load more than 1M records in a file that contains more than 1.1M records, or MariaDB and MySQL fail to load files if the header name is above 64 characters.

User takeaways: When loading real-world files, several aspects need to be taken into consideration aside from strict adherence to the CSV standard. Not every system is scalable for loading larger files (over 1M records, or 100 columns): programming frameworks offer the highest loading accuracy, whereas database systems are faster but require "cleaner" inputs. One possible strategy is to "sanitize" a polluted file by cleaning it through a programming framework before feeding it into more complicated systems, e.g., database systems or business intelligence tools.

Developer takeaways: Regarding automatic systems, we observed that the detection of dialect parameters is often unreliable for files with structural inconsistencies (varying numbers of row delimiters, multiple tables, etc.). We recommend that automated approaches take into account structural pollutions, either by addressing them separately

in a preprocessing step, or by filtering for "outlier" rows during their automatic detections. Additionally, we urge developers to relax their assumptions regarding file structure and dimension: as data becomes larger and more ubiquitous, it is not uncommon to expect files with high record counts and column sizes.

2.5 Summary

In this chapter, we defined a formal framework to distinguish the concepts of content, structure, and format, and we introduced a definition of grammar dialects based on context-free grammars. We apply this formal model of file structure to systematically categorize issues in real-world CSV files, reproduce them with *file pollutions*, and design Pollock, a benchmark for data loading. After benchmarking 16 real-world systems, our results showed that unsuccessful data loading is often caused by a lack of flexibility in the systems' configurations. We advocate that with the use of our benchmark, system designers have an objective metric to assess the data loading capabilities of their tools, as well as a means to identify unexpected and surprising behaviors, as we did in our experimental results.

Currently, we focused the Pollock benchmark on single-pollution files. However, as our experiments in Section 2.4.5 show, systems struggle more with multiple pollutions at once. The grammar-based framework we presented can be extended to support multiple pollutions, but the design of such pollutions would require notions of dependency and more complex strategies to sample the overall search space, which are interesting directions for future work. Through the formal model used by Pollock, we hope to stir research efforts in the data preparation area toward a more principled direction.

Chapter 3

Mondrian: Modeling Layout Templates of Multiregion Files

Large amounts of structured data can be found in spreadsheet files, either distributed in CSV or Excel format [33, 62, 75, 108]. Even if these files are meant for distribution and analysis, they too are often affected by data quality issues and human-induced errors that require data preparation [17, 46].

One of the biggest challenges is the fact that they are often used as canvases in which data is spread out in multiple, independent regions with a custom layout and without a well-defined tabular format. In many cases, there are multiple tables, but metadata regions are also common, e.g., spreadsheet titles, comment sections, or notes to data cells. Moreover, the layout of files often follows a common *template*, that encompasses the number, layout, and schema of tables and metadata regions in a file. This may happen in files with related data from different sources, e.g., the same tables with different data points, or in files produced from the same source, e.g., different tables produced with the same export process.

Such multi-region files are found in enterprise data lakes or open data repositories, often without proper metadata or provenance, and cause these to grow into unordered collections of heterogeneous data [77]. Data scientists typically must prepare these files by first recognizing their layout and then splitting the different regions into separate files to obtain machine-readable content. The aim of this chapter is to propose a structural representation for these files, which we call *layout templates*. This representation may be leveraged for multiple use cases:

- Performing data preparation on files with the same template, users can automatically target regions across files irrespective of file-specific boundaries.
- Integrating files from multiple sources, users can use templates as metadata indicating data provenance, and decide to exclude the tables of a template, e.g., if it contains conflicting or poor-quality data.
- Exploring the content of a data catalog, like a data lake or data market, is typically done with simple keyword-based queries. Layout templates can be used as more sophisticated indices for file discovery.

We propose an automatic and interactive system, named Mondrian, to obtain explicit (the file *layout*) as well as implicit (layout *templates*), file structure representations. The graphical rendering of file layouts inspired us to name our approach Mondrian, after the abstract painter Piet Mondrian. This chapter is based on published work in [116, 117]. In detail, its contributions are:

- A novel unsupervised approach that maps spreadsheets in the visual image domain to detect and match different regions in spreadsheet files.
- A framework to analyze and compare multiregion spreadsheets, using a graph representation with an associated similarity algorithm to detect layout templates.
- A web-based interactive tool[1] that allows users to inspect and correct the results of the automatic detection, and assists them in performing data preparation of large spreadsheet datasets.
- A publicly available dataset of structural annotations for 886 spreadsheets[2], classifying the position and purpose of their composing regions, and a set of template annotations for two datasets, summing up to a total of above 1500 files, identifying classes of files with the same layout.
- A comprehensive set of experiments to prove the effectiveness of the Mondrian approach in solving the region detection and template inference problems, comparing it with state-of-the-art automated methods.

The remainder of this chapter is structured as follows: Section 3.1 provides an overview of related work and highlights the motivation for our approach; Section 3.2 defines some background concepts and formally introduces multiregion files; Section 3.3 introduces the architecture of the Mondrian approach, from parsing spreadsheets as images to detecting layout templates; Section 3.4 discusses the experimental results; Section 3.5 presents an exemplary use case of Mondrian, showcasing a data preparation workflow using the web-based interactive interface. Finally, Section 3.6 concludes the discussion and outlines possible directions for future work.

3.1 Related work

Detecting layout templates for sets of multiregion files is a novel research problem. Related research work has proposed methods to discover related tables in data lakes [65, 130, 131]. However, these methods are unsuitable for spreadsheets, because these files may contain more than just a single table, that often appear in a canvas-like layout in different positions throughout different files (e.g., due to different metadata regions such as preambles or footnotes). As our survey in Section 2.3.1 demonstrated, these pollutions are not rare in real-world files.

A required step before template detection is *table extraction* from spreadsheet files. Prominent research mainly leveraged supervised learning to address this task. The two systems WebSmatch [23] and TableSense [41], analyze spreadsheets applying techniques from the computer vision domain, leveraging respectively connected component detection and convolutional neural networks based on Excel-specific features. We compare experimentally our approach with these two systems in Section 3.4. Pytheas, a system by

[1] Available at `https://hpi.de/naumann/sites/mondrian/demo`
[2] Available at `https://github.com/HPI-Information-Systems/Mondrian`

Christodoulakis et al. [21], employs a rule-based algorithm with experimentally learned weights The approach presented by Koci et al. [63] combines supervised machine learning and genetic-based algorithms. We use the latter for experimental comparison with our approach in Section 3.4.

However, a fundamental limitation of all these approaches is the generalizability to unseen datasets, given their reliance on large amounts of training data. Our intuition, corroborated by our experimental results in Section 3.4, is that unsupervised learning is more suitable for datasets with unseen file layouts.

Further limitations of some of these approaches, e.g. [41, 61, 63, 64], is their reliance on Excel-specific features, which may not always be available since in many cases spreadsheets are distributed in a CSV format. For example, in two of the data portals whose data distribution we sampled in Chapter 2 (see Table 1), the CSV files are 3–5 times more common than Excel files.

Different from table extraction, to the best of our knowledge, no previous approach aimed at detecting reoccurring layout templates for multiregion files. Solving this problem may prove beneficial to assist end-users with data preparation, since it can be used to define once and reuse the same preparation pipelines to new files following the same layout template.

Moreover, the layout template information can be leveraged by existing spreadsheet systems. For example, to perform information extraction, the work of Chen et al. [16], given table boundaries, leverages active learning to detect interesting *spreadsheet properties*, such as aggregation rows or hierarchies. Detecting spreadsheet templates can reduce the number of files for which user feedback is required, or empower spreadsheet data management systems, like Senbazuru [15], for indexing or querying purposes.

3.2 Multiregion files, layouts, templates

In this section, we introduce definitions for the concepts of multiregion files, layouts, and layout templates; formulate a hierarchy of equivalence notions to compare regions and layouts; and state the research problems addressed by our approach.

Complex layouts with multiple regions are a byproduct of spreadsheet software rendering data on canvases where users freely lay out different data and metadata alike (see Figure 8).[3] We use the term *multiregion* rather than *multitable* files since our approach can also be applied for files whose layout contains a single table but additional metadata regions.

Typically, multiregion files can be found in comma-separated values format (CSV) or Microsoft Excel format (XLS/XLSX). For our purposes, spreadsheets are defined as value-delimited files that contain data in cells with a grid structure. We assume no specific row- or column-based structure of the content.

We assign each cell a unique identifier (x, y), where $x, y \in \mathbb{N}_0$ correspond to the column and row indices, respectively. These (x, y) coordinates are points in a Euclidean space,

[3]Some formats and tools allow a spreadsheet to have more than one worksheet. Without loss of generality, we consider each worksheet as a separate file.

ECONOMIC CALCULATOR FOR FIRM VERSUS NON-FIRM PURCHASE																
		Current Hour =	14		Next Hour =	15						PV				
		Enter Local Generation Avail:	529		Projected Control Area Load:	840		Local Avail.	529			IID	80			
		Enter Remote Generation:	581		PNM Contingent:	41		Gen.	397			real/time	80			
		Firm Purchases into EPE:	25		TNP Firm:	25		UnLoaded	132			net/pre	-75			
		Non-Firm Purchases into EPE:	0		IID Firm + Contingent:	150						Tep/exc	300			
		SPS Firm:	0		Firm Sales:	0		103	Local	529		iso	0			
		Reserves:	0		Non-Firm Sales:	0	=-load	50	Copper	69			385			
		Total Generation for Load:	1135		Total Load Next Hour:	1056		53		598		(If total goes over 600 EPE has to wheel back)				
		Enter Total Spin Required:	78		PNM Contract:	46	(Contingent upon units 7 & 8 number automatically feeds from the calculation tab)									
		Spin Required:	39		IID Firm Contract:	100				Enter Blue Numbers						
		Non-Spin Required:	39		TNP Contract:	25										
		Spin Required + Regulating Margin:	69		SPS Contract:	100	Lambda =	47.74				Spin from Ramp rates				
							System avg. =	25.7								
		*Amount of Spin:	79	Weighted Avg. Purchase Power Calculator								Output	Highs'	Spin	RR	unloaded
		Amount of Non-Spin:	50			MWH	$/MWH				Unit 1	50	80	30	3	30
		Total Spin:	129		Firm Block 1:	0	0	0			2	70	82	12	4.5	12
		Spin Available/(Deficient):	40		Firm Block 2:	0	0	0			3	0	0	0	5.1	0
		Enter Firm Price:	0		Firm Block 3:	0	0	0			GT1	0	0	0	10	0
		Enter Non-Firm Price:	0		Firm Block 4:	0	0	0			GT2	0	0	0	10	0
					Firm Block 5:	0	0	0			GT1S	0	0	0	3.33	0
		MWH of Firm Avail./(Deficient):	40		Total:	0	NA	0	0		GT2S	0	0	0	3.6	0
		Total Cost of Firm:	0			MWH	$/MWH				NM4	148	214	36	3.6	66
		MWH of Non-Firm Avail./(Deficient):	40		Non-Firm Block 1:	0	0	0			Copper	0	0	0	10	0
		Total Cost of Non-Firm:	0		Non-Firm Block 2:	0	0	0			6	0	0	0	2	0
					Non-Firm Block 3:	0	0	0			7	33	33	0	2.1	0
					Non-Firm Block 4:	0	0	0			8	96	120	10	1	24
					Non-Firm Block 5:	0	0	0			Total	397	529	88		132
					Total:	0	NA	0	0		FC	49	108			
											PV	532	581		Lost Gen.	44
NOTE:	* ACTUAL SPIN SHOW MAY BE LESS SINCE UNIT RAMP RATES ARE NOT CONSIDERED.							0				581	689			

Figure 8: A multiregion file of the ENRON corpus viewed in a spreadsheet software

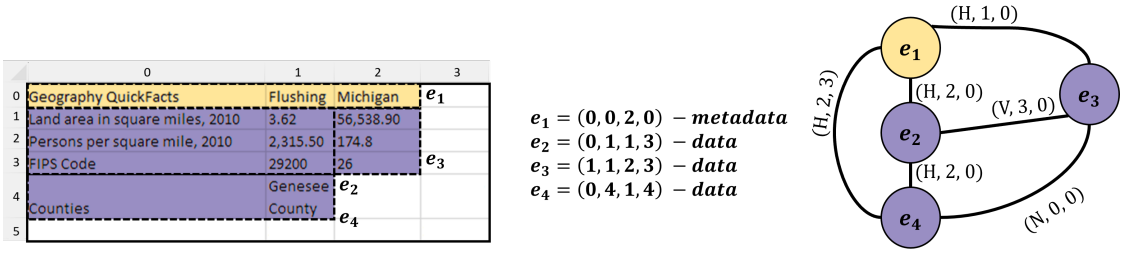

Figure 9: Different elements (yellow: metadata, purple: data) covering a spreadsheet table, their coordinates, and their layout graph annotated with alignment information.

originating in the top-left corner, in analogy to spreadsheet design. Every cell serves some purpose in the spreadsheet. We consider three fundamental types of cells:

Definition 9 (Cell types)**.** A cell c of a spreadsheet S belongs to one of the following mutually disjoint cell types:

1. **Data**, if it carries the data values of a file;
2. **Metadata**, if its information is related to a set of data cells;
3. **Empty**, if it does not contain any data or only whitespace characters, e.g., it is used for visual formatting.

Elements are simple structures, grouping cells of the same type:

Definition 10 (Element)**.** Given a spreadsheet file S, an element e is a rectangular set of adjacent cells of S of the same type. The element type of e corresponds to the cell type of its cells.

According to its position in the spreadsheet, an element can be described with the vector $(x_0, y_0, x_1, y_1) \in \{\mathbb{N}_0\}^4$, where the coordinates (x_0, y_0) represent an element's top-left cell and (x_1, y_1) its bottom-right cell. Figure 9 shows an exemplary spreadsheet with four elements and their coordinates.

In a given spreadsheet we can identify several elements and describe their spatial relationships. It is worthwhile noting that, since elements are groups of adjacent cells, the areas of any two given elements in a spreadsheet cannot overlap. Considering the elements' rectangular nature and the grid-like space of spreadsheets, we encode the relationship between two elements with three features: alignment direction, alignment magnitude, and distance.

The *alignment direction* is based on the overlap of the elements' projection on the x-axis and the y-axis:

Definition 11 (Alignment)**.** Two elements $a := (a_{x_0}, a_{y_0}, a_{x_1}, a_{y_1})$, $b := (b_{x_0}, b_{y_0}, b_{x_1}, b_{y_1})$ align:

$$\begin{cases} \textit{Vertically (V)} & \text{if } \max(a_{y_0}, b_{y_0}) \leq \min(a_{y_1}, b_{y_1}) \\ \textit{Horizontally (H)} & \text{if } \max(a_{x_0}, b_{x_0}) \leq \min(a_{x_1}, b_{x_1}) \\ \textit{Not aligned (N)} & \text{otherwise} \end{cases}$$

In Figure 9, elements e_1 and e_2 align horizontally, elements e_2 and e_3 align vertically, and elements e_3 and e_4 are not aligned.

The *alignment magnitude* is the number of shared points across the axis in the case of horizontal or vertical alignment:

Definition 12 (Alignment magnitude)**.** The alignment magnitude between elements a, b is:

$$\begin{cases} \min(a_{y_1}, b_{y_1}) - \max(a_{y_0}, b_{y_0}) + 1 & \text{if } a, b \textit{ align vertically} \\ & \text{if } a, b \textit{ align horizontally} \\ 0 & \text{otherwise} \end{cases}$$

In Figure 9, the alignment magnitude between elements e_1 and e_2 is 2, as they share two cells on the x-axis, the alignment magnitude between elements e_2 and e_3 is 3, as they share three cells on the y-axis, and the alignment magnitude between elements e_3 and e_4 is 0, as they are not aligned.

The distance between the elements is calculated as the distance of their two closest points. In case the two elements are horizontally or vertically aligned, this resolves to the distance between their closest boundaries; otherwise, it is calculated as the Euclidean distance of the two closest corners:

Definition 13 (Distance)**.** The distance $d(a, b)$ between elements is

$$\begin{cases} d_v : |\min(a_{x_1}, b_{x_1}) - \max(a_{x_0}, b_{x_0}) + 1| & \text{if } a, b \textit{ align vertically} \\ d_h : |\min(a_{y_1}, b_{y_1}) - \max(a_{y_0}, b_{y_0}) + 1| & \text{if } a, b \textit{ align horizontally} \\ \sqrt{d_v^2 + d_h^2} & \text{otherwise} \end{cases}$$

In Figure 9, the distance between elements e_1 and e_2 is 0, as they are adjacent, while the distance between elements e_1 and e_4 is 3, as they are three cells apart.

Often, especially in spreadsheets with complex cell layouts, even non-adjacent cells could be logically grouped. For example, a table may have missing values that result in empty rows in-between valid data rows (see The top-left-most table in Figure 8). Elements are therefore not sufficient to completely describe the layout of a spreadsheet, and we need a higher-order abstraction to group semantically related elements, which are not necessarily adjacent to each other. Groups of elements can serve different purposes: examples are tables, preambles, footnotes, or any other domain-specific construct. To abstract their specific purpose, we identify them as regions:

Definition 14 (Region)**.** A region R is a complete graph having as nodes a set of semantically related, non-empty elements $\mathcal{E}$, connected with edges labeled with their pairwise spatial relationships.

The left side of Figure 10 shows two given regions, their composing elements, and their associated graph layouts: region R_1 encompasses the elements of Figure 9; in region R^2 the data element e_3^2 is horizontally aligned to the header element e_1^2 and vertically aligned to the data element e_2^2.

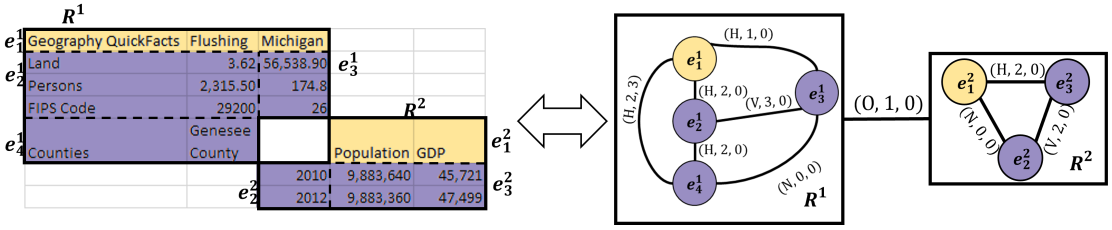

Figure 10: Two overlapping regions and their graph layout ($R1$ from Figure 9).

Considering the definition of regions, a multiregion spreadsheet is trivially defined as a spreadsheet containing multiple regions. Ultimately, our goal is to find structural similarity across different, possibly multiregion files. To do so, it is first important to identify a meaningful set of regions for each file: that is to say, draw the boundaries of different regions such that they are independent and serve distinct purposes. To describe the coordinates of a region boundary in the spreadsheet space, we use the bounding box of its set of elements:

Definition 15 (Region boundary)**.** The boundary of a region R, with its elements $\mathcal{E}$, is defined as a rectangle (x_0, y_0, x_1, y_1), where:

$$x_0 = \min_{e \in \mathcal{E}} e_{x_0}, \quad y_0 = \min_{e \in \mathcal{E}} e_{y_0}, \quad x_1 = \max_{e \in \mathcal{E}} e_{x_1}, \quad y_1 = \max_{e \in \mathcal{E}} e_{y_1}$$

Once regions have been identified, we are concerned with their layout. We extend to pairs of regions the spatial relationship feature vector defined for pairs of elements, using the (x_0, y_0, x_1, y_1) coordinates of region boundaries to compute alignment direction, magnitude, and distance. One caveat is that considering their boundaries, two given regions can, in general, have overlapping bounding boxes, which is not the case for elements. We extend the spatial relationship feature vector for overlapping regions as:

Definition 16 (Overlapping regions)**.** Given two regions, $A := (a_{x_0}, a_{y_0}, a_{x_1}, a_{y_1})$ and $B := (b_{x_0}, b_{y_0}, b_{x_1}, b_{y_1})$, their alignment direction is *overlapping* (O) if $\max(a_{y_0}, b_{y_0}) \leq \min(a_{y_1}, b_{y_1})$ and $\max(a_{x_0}, b_{x_0}) \leq \min(a_{x_1}, b_{x_1})$. Then, the alignment magnitude is $(\min(a_{y_1}, b_{y_1}) - \max(a_{y_0}, b_{y_0}) + 1) \cdot (\min(a_{x_1}, b_{x_1}) - \max(a_{x_0}, b_{x_0}) + 1)$ and the distance is 0.

The magnitude corresponds to the area of the overlap, which ultimately equals the product of the horizontal and vertical alignment magnitudes, considering that two overlapping regions are both horizontally and vertically aligned. For example, the two regions R^1 and R^2 in Figure 10 overlap for one cell, and their spatial relationship vector is (O, 1, 0). Finally, by describing a set of non-empty regions with a complete graph, we can define the layout of a spreadsheet:

Definition 17 (Spreadsheet layout)**.** The layout of a spreadsheet file S is a complete graph having as nodes its set of non-empty regions, connected with edges labeled with their pairwise spatial relationship.

Often, region and file layouts are not one-off models but stem from a systematic creation process. For example, the US Census open data portal contains the same data report for

multiple geographical entities, each downloadable as a separate csv file[4]. Our goal is to provide a framework to define and analyze *templates*, i.e., classes of structural equivalence across multiple files. We compose a hierarchy of equivalence notions, beginning with the finest-grained unit of comparison, the cell, and extend it to elements:

Definition 18 (Cell and element equivalence)**.** Two cells c_1, c_2 are equivalent if their type is equal. Two elements e_1, e_2 are equivalent if their cells are equivalent.

To define region equivalence, we must also be able to include regions with equal structure, but that may have a generally different number of data elements, e.g., two tables with the same schema but different lengths.

Definition 19 (Region equivalence)**.** Two regions R_1, R_2 are equivalent if there is a one-to-one equivalence between their metadata elements and their graphs are isomorphic.

At the spreadsheet level, the definition for layout is similar:

Definition 20 (Layout equivalence)**.** Two layouts L_1, L_2 are equivalent if there is a one-to-one equivalence between their regions and their graphs are isomorphic.

In practice, if many files are collected from different sources, we want to be able to discover entire sets of equivalent spreadsheets:

Definition 21 (Layout template)**.** A layout template $\mathcal{L}$ is a class of equivalent layouts.

Recognizing templates is of great value for data preparation, as it potentially saves users the time to manually inspect and prepare individual files: a pipeline of preparation steps can be defined once and executed repeatedly on different files from the same template. As computing exact graph isomorphism is computationally expensive, Mondrian uses approximate similarity metrics to find templates, which we describe in Sections 3.3.2 and 3.3.3.

Given the definitions stated, the problem of recognizing and matching multiregion spreadsheet layouts is composed of several distinct sub-problems that have an inherently visual nature. The first fundamental problem is to find the correct region boundaries. A human expert would solve this task by understanding the semantics of the data as well as its spatial distribution. Then, to identify recurring layouts, they would be required to manually inspect and compare each separate file looking at its data — a cumbersome, error-prone, and time-consuming task. According to our definitions of equivalence, this task requires semantic concepts and possibly domain knowledge, e.g., to distinguish table schemata. However, to design a general and domain-independent approach, we focus only on structural properties. We present the Mondrian approach to address the following research problem:

Problem Statement: Given a set of spreadsheet files $\mathcal{F}$, each with its layout L_f:

1. Given a file f, how to determine the correct boundaries of its regions R_f that determine its layout L_f?
2. Given two different regions r_x, r_y, how to approximate their equivalence without semantic information?

[4]http://www.census.gov/quickfacts accessed Nov 3, 2021

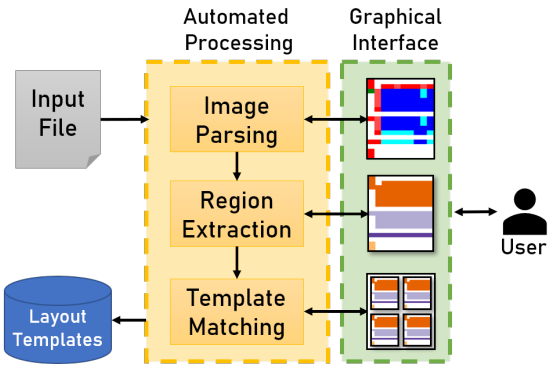

Figure 11: The Mondrian pipeline.

3. Given pairs of files f_x, $f_y \in \mathcal{F}$, how to measure the similarity of their layouts and use these similarities to recognize unique layout templates $\mathcal{L}$ that occur in $\mathcal{F}$?

3.3 The Mondrian approach

To identify the conceptual entities defined in Section 3.2 in practice, without resorting to semantic knowledge, the intuition of Mondrian is to transform the domain of spreadsheets from data content to image. To extract file layouts and detect templates, the Mondrian approach follows three phases: image parsing, region extraction, and template matching.

Figure 11 shows a graphical overview of its pipeline. The *image parsing* phase creates a colored image using the data type of its cells. We convert cells into pixels, encoding their syntactical types into colors. Using the web-based graphical interface, users of Mondrian can visualize the file content and the result of the image parsing.

In the *region extraction* phase, images are segmented with a partitioning step and a clustering algorithm detects independent region boundaries. In the graphical interface, the resulting regions can be visually inspected, and their boundaries can be interactively corrected if needed.

Once regions are identified, the *template matching* phase encodes each file layout as a graph, based on its extracted regions. We analyze pairs of file layout graphs using a similarity measure that is based on the similarity flooding algorithm [71]. Once the similarity of all relevant file pairs is calculated, templates are defined as classes of file layouts with a similarity above a given threshold (configurable by end-users, 0.98 by default). The graphical interface presents the output of the template matching stage as a list of templates and their associated files, allowing users to interactively edit them and save the results. The remaining subsections of this chapter describe the three phases of the Mondrian approach in detail and provide a complexity analysis of the most relevant steps.

3.3.1 Image parsing

To cover the most general cases, our approach takes as input comma-separated value files. Files with different delimiters or formatted with XML markup, such as Microsoft Excel files, can be easily converted into a CSV file.

Ignoring possible markup information is the trade-off for a method applicable to a wide spectrum of spreadsheets, independent of their format specifications. For native csv files, we cannot assume that all rows have the same number of delimiters. Thus, we pad rows with empty cells up to the length of the longest row. Given a csv file with M rows and N columns, we create an image with the dimensions $M \times N$, where each pixel represents a cell in the csv file. Our definitions of entities and their equivalence build upon the concept of "cell type": in practice, we substitute semantic types with *syntactic types* and, correspondingly, relax their equivalences into an approximate *structural similarity*.

We identify four fundamental syntactic types: *number*, *datetime*, *string*, *empty*. Except for *empty*, each of these types can be further refined in subtypes: a *number* can be *integer* or *floating-point*; a *datetime* can be a *time* or a *date*; a string can be either *uppercase*, *lowercase*, *titlecase*, or *generic*. In parsing the spreadsheet as an image, we transform every cell into a pixel with a different color according to its type (see Figures 8 and 12). Table 9 shows the color corresponding to each data type and a sample cell from Figure 8[5] that was parsed according to that type.

Recognizing the syntactic type of cells without semantic knowledge is, in general, a coarse-grained and error-prone operation: consider the uncertain nature of the value "1990", which can be a date or a number. As our experiments in Section 3.4.4 demonstrate, however, coarse-grained parsing is sufficient to approximate region equivalence for the task of template inference, with the reasonable assumption that any parsing mistake would be reflected across all similar files.

To segment the file into elements, we first find connected components, which reflect cell aggregates that could not be so easily recognized in a spreadsheet software view (see Figure 8). The change in width/height proportion happens because each cell occupies one square pixel in the image, while in the spreadsheet software cell columns and rows can have different widths or heights, usually set according to the length of their values. With this *cell normalization*, for example, a human observer is more likely to note the four aligned vertical elements on the left of the image.

[5]Except for the time and date types, which were not present in the original file.

Type	Sub-type	Sample cell	Color
Empty	Empty	" "	White
Number	Integer	"14"	Light Blue
	Floating-point	"47.74"	Dark Blue
Datetime	Time	"17:00"	Light Green
	Date	"17/9/20"	Dark Green
String	Uppercase	"MWH"	Maroon
	Lowercase	"real/time"	Salmon Red
	Titlecase	"Firm Sales"	Tomato Red
	Generic	"System avg. ="	Scarlet Red

Table 9: Data types and their colors.

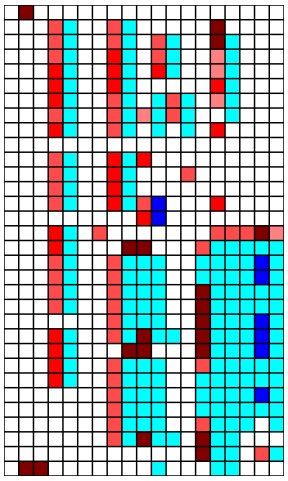

Figure 12: Image rendering for Figure 8

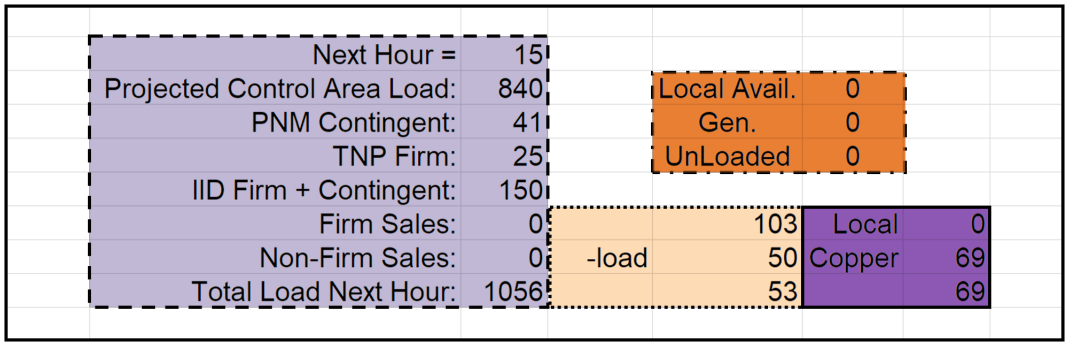

Figure 13: Detail of Figure 8 highlighting adjacent, independent regions.

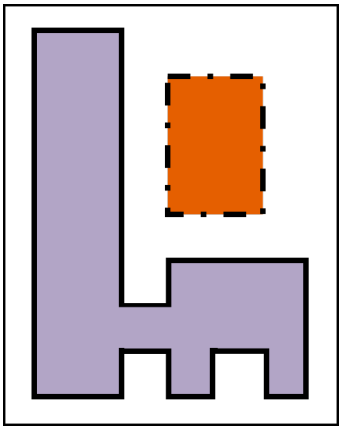

Figure 14: Connected components.

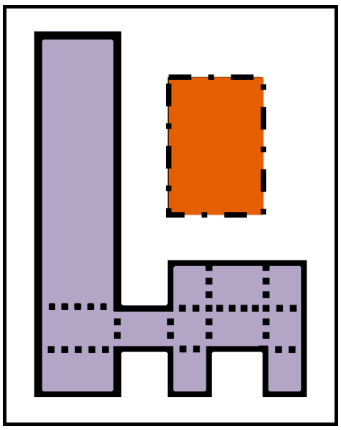

Figure 15: Partitioning step.

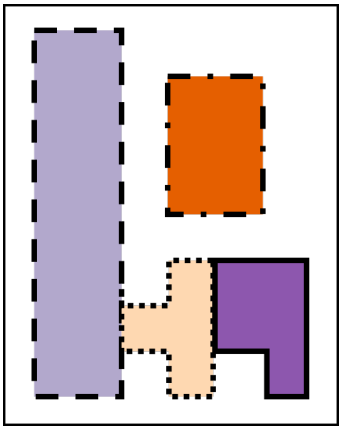

Figure 16: Clustering results.

However, considering connected components as elements could lead to incorrect region boundaries: as highlighted by Figure 13, sometimes regions can be adjacent to each other. In the example, different rectangular regions compose a single connected component with irregular edges (Figure 14).

Therefore, to identify a valid set of elements that leads to correct region boundaries, we need a segmentation strategy for connected components. We *cut* the connected components along their non-concave edges (Figure 15). Formally, we partition the components following a *rectilinear* cut that is obtained by extending the edges incident to concave vertices towards the interior of the polygon, until a polygon boundary is met. Bajuelo et al. show that each given polygon, with v concave vertices, can be split into $O(v^2)$ elements, with $2v + 1$ as a minimum [6].

With this method, even coherent elements could be initially decomposed. This is eventually corrected while searching for regions in the following phase, clustering, where finer-grained elements can be either merged or not, granting the ability to even discover regions that appear directly adjacent in the spreadsheet (Figure 16).

3.3.2 Region extraction

The next phase of Mondrian has the objective of clustering together elements that belong to the same region. For a given spreadsheet, we have no prior knowledge of the number of regions that it contains. Thus, we cannot use *centroid-based* clustering approaches, such as k-means. Instead, we resort to a customized *density-based* approach, modifying DBSCAN [29] to operate with a custom distance metric that highlights the structural properties we seek.

The DBSCAN optimization problem aims at finding points in dense neighborhoods of a given space: if we consider spreadsheet elements as points, a region corresponds to an area with a high density of points. Given a distance function and a minimum number of points m that form a cluster, the algorithm defines as *core points* of a cluster all those that have at least m points closer than a threshold ε, also called the *radius* of the search space. Then, it groups all points that are within ε from a core point, or within ε from non-core points belonging to a cluster. The value of ε is a hyperparameter that can be set globally or for each file, and in our experiments we found that, overall, the best results are obtained when setting a radius of 1.5 (see Section 3.4).

In the original DBSCAN algorithm, every leftover point is labeled as noise. In our scenario, we are interested in labeling all non-empty elements of a spreadsheet. Therefore, we do not consider any element as noise and set the minimum number of elements that can form a region as $m = 1$.

The distance function we use to compare elements is a weighted sum of three terms:

1. **Distance**: The Euclidean distance of their closest cells (Definition 13).
2. **Size difference**: Considering a_0, a_1 as the areas of two elements, with the larger being a_1, the ratio $1 - a_0/a_1$.
3. **Alignment magnitude**: The number of shared points across the horizontal or vertical axis (Definition 12). Defining (x^i_{TL}, y^i_{TL}) as the coordinates of the top-left corner of an element i and conversely (x^i_{BR}, y^i_{BR}) as those of the bottom-right

corner, the alignment is calculated as the sum of horizontal and vertical alignment, using the formula:

$$|y^0_{TL} - y^1_{TL}| + |y^0_{BR} - y^1_{BR}| + |x^0_{TL} - x^1_{TL}| + |x^0_{BR} - x^1_{BR}|$$

In the first term, *distance*, we compute the distance between the closest cell of two elements, to avoid the influence of element size in the calculation. In fact, even if two elements are adjacent, if their width/height extends much farther than the boundary they share, any other distance metric (e.g., the distance of their center points) would be dependent on the width/height of the elements rather than on their visual closeness.

The intuition behind the second term, *size difference*, which is inversely proportional to the difference in the size of the two elements, is that on one hand, two elements that are equally small or large, such as two metadata regions or two tables, are more probable to be two independent regions; on the other hand, larger elements are more likely to be grouped with smaller elements, such as tables and footnote regions.

The third term, *alignment magnitude*, is calculated as the sum of horizontal alignment and vertical alignment. This is meant to compensate for the effect of empty cells within regions: if different elements that are separated by visual space have a high alignment, they are most likely to belong together, e.g., two parts of a table that are separated by an empty column.

The weights for these terms are α, β, γ, respectively, and can be fine-tuned globally or for a given spreadsheet as hyperparameters for optimal boundary detection. Additionally, the value of the radius ε plays an important role in the success of the clustering, as different files can have different properties regarding the size of regions and the mutual distances of their elements. We hypothesize that larger spreadsheets have, on average, a higher number of elements with greater distances, and therefore benefit from larger radii. As Section 3.3.2 points out, the best performances are obtained when setting a custom radius for each file. To reflect a scenario with no specific hyperparameter selection, we also experimented with our approach to find a suitable fixed hyperparameter setting for all files.

Once their boundaries have been identified, we are interested in equivalent regions. Our definition for region equivalence (Definition 19) is based on element boundaries and their types: for example, two footnote regions are equivalent if their entire content is equal, while two tabular regions are equivalent if their header elements are the same, regardless of the actual data content.

In Mondrian, we measure approximate equivalence using a similarity score. Moreover, due to its complexity, we do not compute graph isomorphism for region matching but rather compute region similarity based on syntactic cell types and their color encoding. Note from Table 9 how our color encoding assigns one primary color (red, green, blue, white) to each fundamental data type and then varying shades of the primary color to each subtype belonging to the same fundamental data type. For example, *string* is associated with red, with *lowercase* being "tomato red" (RGB $(255, 75, 75)$) and *titlecase* being "scarlet red" (RGB $(255, 0, 0)$).

In this way, cells with the same fundamental data type, but different subtypes are more similar in the color space than cells from different fundamental types. A given region is

described with the color histograms of its cells, computed with 64 bins for each channel, for a total of 192 bins. The color histogram is a global descriptor of each region that acts as a region "fingerprint": its values are dependent on the amount and distribution of cells of different types. The similarity of any two regions is then computed as the cross-correlation of their color histograms. Furthermore, the color encoding can be easily extended including more, or further refined, data types. If two highly similar regions (that is, whose similarity is over a given threshold) are found in two different files, they are considered equivalent and the file layouts that contain them are candidate instances of the same template.

3.3.3 Template matching

Each spreadsheet file, once its regions have been detected, has an associated file layout, represented as a complete graph with regions as nodes and labeled edges that describe their spatial relationships (Definition 17). As with region equivalence, we do not compute an exact graph isomorphism for layout equivalence but rather approximate it with a similarity measure. Our algorithm is based on the similarity flooding approach proposed by Melnik et al. for graph matching [71]. The core intuition is to first compute an initial pair-wise similarity of nodes across the two file layout graphs using the region similarity metric described in Section 3.3.2. If the graph $\mathcal{G}_a$ has U nodes and the graph $\mathcal{G}_b$ has V nodes, we obtain a matrix σ^0 of $U \times V$ values.

Additionally, we build a $\binom{u+1}{2} \times \binom{v+1}{2}$ matrix Φ of edge similarities, where the value in position $\Phi(i+j,\ k+l)$ with $i, j, k, l \in \mathbb{N}_0$ corresponds to the edge similarity of $edge(u_i,\ u_k)$ and $edge(v_j,\ v_l)$.

The edge similarity is set to 0 if any of the node pairs $(u_i,\ u_k) \in \mathcal{G}_a$, $(v_j,\ v_l) \in \mathcal{G}_b$ has no connecting edge (including the case of both being the same node), or if the two edges have a different *alignment direction*. Otherwise, the edge similarity is computed as the Euclidean distance between the vectors composed of the features *(alignment magnitude, distance)*, normalized by the maximum value to have a similarity score in $[0, 1]$.

The similarity of the nodes in σ^0 is then iteratively "flooded" by multiplying the similarity of each node pair with the similarity of the neighboring node pairs, weighted by the edge similarity in Φ. In formal terms, the similarity of the i-th node of $\mathcal{G}_a$ and the j-th node of $\mathcal{G}_b$ is iteratively updated using the formula from [71]:

$$\sigma^k(i,j) = \sigma^0(i,j) + \sum_{m=0\cdots V,\ n=0\cdots U} \sigma^{k-1}(m,n) \cdot \Phi(i+m, j+n)$$

As we look for a 1:1 node match, we ensure that for every neighboring node pair $(u_i,\ u_j) \in \mathcal{G}_a$, only the node pair $(v_j,\ v_l) \in \mathcal{G}_b$ with the maximum edge similarity is used. To avoid imbalance in similarities for node pairs (u,v) where any of u or v has a high number of neighbors, we normalize the value of Φ dividing $\Phi(u+v, u_i+v_j)$ by 2^{n-m}, where $n,\ m$ are the number of neighbors of u and v, respectively. Finally, at each iteration, we normalize the values of σ^i. The iterative computation is stopped either when the matrix distance $||\sigma^{i+1}, \sigma^i||_2$ falls below a given threshold, or when a maximum number of iterations is reached.

During our experimentation, we empirically observed that in most cases the matrix difference falls quickly (in a handful of iterations) to values in the range $[0.01, 0.1]$ and then stabilizes, reaching values under 0.01 with a much slower convergence speed (in thousands of iterations). Therefore, we recommend setting a threshold of 0.1 and a maximum number of iterations to 10, which we deem sufficient considering the satisfactory results obtained on the template inference task reported in Section 3.4.4.

At the end of the similarity flooding stage, we can consider the matrix σ as the weight matrix of a fully connected bipartite graph $\mathcal{B}$, with the two partitions composed of the nodes of $\mathcal{G}_a$ and $\mathcal{G}_b$, respectively. To compute the final similarity score of ($\mathcal{G}_a$, $\mathcal{G}_b$), we find a maximum weighted matching on $\mathcal{B}$ and average the corresponding weights found, including zero values in the computation for every $| \, |\mathcal{G}_0| - |\mathcal{G}_1| \, |$ node left unmatched. In formal terms, given the weights $w(u, v)$ for nodes $u \in \mathcal{G}_a, v \in \mathcal{G}_b$, the similarity between $\mathcal{G}_a$ and $\mathcal{G}_b$ is computed as:

$$sim(\mathcal{G}_a, \mathcal{G}_b) = \frac{\sum\limits_{u \in \mathcal{G}_a, v \in \mathcal{G}_b} w(u, v)}{\max(|\mathcal{G}_a|, |\mathcal{G}_b|)}$$

As this graph similarity is asymmetrical, because of the matrix normalization included in the calculations, for every pair of files f_a, f_b we compute the final file layout similarity $sim(f_a, f_b)$ averaging between $sim(\mathcal{G}_a, \mathcal{G}_b)$ and $sim(\mathcal{G}_b, \mathcal{G}_a)$. As proven by Melnik et al., in the case of fully connected graphs, the layout similarity computation has a complexity of $O(u^2 \cdot v^2)$ for two files with u and v regions [70]. In our two experimental datasets, files contain an average of 4.43 and 2.09 regions (see Table 10).

As we approximate pairwise layout equivalence with our graph-based similarity measure, we consider two file layouts to be instances of the same template if their pairwise similarity is above a given threshold τ_f (subject to evaluation in Section 3.4.4).

3.3.4 Template clustering

To extend template inference beyond pairs of files, we use an inductive approach: given a set of files, each with its detected regions, we examine the set iteratively. The procedure for template recognition is described in pseudocode in Algorithm 1.

First, every file is parsed as an image (Lines 6–7) and its regions are detected (Line 8). Then, its regions are compared with all the regions r_t from the templates of the files previously seen, in $\mathcal{R}$ (Lines 10–11). If a region r_f is similar to a region r_t in $\mathcal{R}$ more than a threshold τ_r, we add a candidate template pair with the file f and each file f_t whose layout contains r_t (Line 15). During our experimentation, we discovered a region threshold $\tau_r = 0.75$ to be sufficient to obtain valid similar layout candidates.

If no region in $\mathcal{R}$ matches any of the regions in f (Line 17), Mondrian will not add the file as a candidate for layout similarity, and the file is considered an instance of a singleton template. Its regions are added to the global index of regions $\mathcal{R}$, along with the information that these regions are found in the layout of f (Lines 18).

Following, we compute the layout similarity for every candidate pair of files identified (Lines 23–24), i.e., all pairs (f, f_t) for each file f_t with at least one region similar to the

Algorithm 1 Pseudocode for the Mondrian approach

Input: set of input files $\mathcal{F}$, clustering parameters α, β, γ, region similarity threshold τ_r, layout similarity threshold τ_f
Output: templates $\mathcal{T}$, regions $\mathcal{R}$
Global region index $\mathcal{R} \leftarrow \{\}$
Set of templates $\mathcal{T} \leftarrow \{\}$
Candidate file pairs $P \leftarrow \varnothing$
for f in $\mathcal{F}$ **do**
    img $\leftarrow$ parse(f)
    $R_f \leftarrow$ region_detection(img, α, β, γ)
    for r_f in R_f **do**
        for r_t in $\mathcal{R}$ **do**
            $\sigma_r \leftarrow$ region_similarity(r_f, r_t)
            if $\sigma_r \geq \tau_r$ **then**
                $f_t \leftarrow$ file that contains r_t
                * the pair (f, f_t) is a candidate for a template \
                $P \leftarrow P \cup \{(f, f_t)$ for $f_t \in \mathcal{R}\langle r_t \rangle\}$
                $\mathcal{R}\langle r_t \rangle \leftarrow \mathcal{R}\langle r_t \rangle \cup \{f\}$
            else
                $\mathcal{R}\langle r_f \rangle \leftarrow \{f\}$
    if $|\mathcal{R}| == 0$ **then**
        * Initialize the region index with the file regions as a singleton template \
        $\mathcal{R} \leftarrow \{r_f : \{f\} \; \forall \; r_f \in R_f\}$
Similarity graph $G_s \leftarrow \{f$ for $f \in \mathcal{F}\}$
for (f, f_t) in P **do**
    $\sigma_f \leftarrow$ layout_similarity(f, f_t)
    if $\sigma_f \geq \tau_f$ **then**
        Add an edge connecting (f, f_t) in G_s
$\mathcal{T} \leftarrow$ find_connected_components(G_s)
return $\mathcal{T}$, $\mathcal{R}$

ones in f. If the layout similarity of the pair (f, f_t) is greater than τ_f we group f, f_t, and, recursively, all files grouped with both f and f_t (Lines 25–26). We do so by creating a graph with a node for each file, and an edge connecting two nodes if the layout similarity is above the layout threshold τ_f. Templates are found as the connected components of such graph (Line 27), transitively closing the set of templates.

We note that the results for a file set are independent of the order the spreadsheets are processed: at the last iteration, all regions will have been compared against each other, as well as all pairs of files that contain matching regions. If at any given point a file is found matching two distinct templates, these are merged.

We choose this iterative approach for different reasons: first, it suits a continuous development scenario, where the region index and template layouts are persistently stored and can be reused in later stages as new files are pre-processed. Second, it is significantly less computationally expensive to pre-compute region similarities and prune the template search space rather than perform graph similarity for each pair of files, which would anyway include computing the pairwise region similarity for all pairs of regions found across all files.

3.3.5 Approach complexity

Formally, the complexity of Mondrian depends on three main procedures: the region detection, the region similarity calculation and the file layout similarity calculation. The region detection runs for each of the F files in the dataset: for a file containing E rectangular elements, DBSCAN has complexity in the average case of $O(E \cdot logE)$, and, in the worst-case, a complexity of $O(E^2)$ [29]. In the worst-case scenario, where all non-empty cells are non-adjacent (a layout similar to a checkerboard), E is equal to the number of non-empty cells of a file.

The complexity of region and layout similarity depends on the number of files F, the number of overall regions across files in the dataset M, as well as the number of unique regions N. The cost of computing pairwise region similarity is constant, as it is a single operation on two fixed-length vectors, whereas the cost for pairwise layout similarity for two files containing N_1 and N_2 regions is $O(N_1^2 \cdot N_2^2)$, as reported by Melnik et al. [70].

If no regions are similar, i.e., $M = N$, the region similarity stage has a cost of $O(M^2)$, but Mondrian computes no layout similarity. If all files contain the same regions, i.e., $M = N \cdot F$, the region similarity is computed $O(N^2 \cdot F) = O(M \cdot N)$ times.

In this worst-case scenario, no pruning happens, and the complexity for layout similarity is $O(N^4 \cdot F^2)$. Typically, the number of regions in a file is much lower than the number of files: for example, Deco and Fuste, the two real-world datasets used for our experiments described in Section 3.4.1, both contain above 800 files with on average 4.43 and 2.09 regions per file, respectively. Therefore, assuming $N << F$ the upper bound for complexity is given by $O(F^2)$.

Finally, the transitive closure operation to obtain templates has a complexity of $O(F \cdot S)$, where S stands for the number of pairwise similar files: in the worst-case scenario, $S = \frac{F(F-1)}{2}$ and the complexity is $O(F^3)$.

In our empirical observations, the average complexity of template recognition for Mondrian behaves quadratically with the number of files, as reported in Section 3.4.4, with typical absolute runtimes in the order of $[1, 10]$ minutes.

3.4 Evaluation

Multiregion spreadsheets pose interesting data engineering challenges. In Section 3.2 we described three related research problems: region detection, region matching, and template inference. We conducted experiments to evaluate whether it is possible to address these problems using an automated approach that is general with respect to the spreadsheet format, and with respect to domain knowledge. We compare Mondrian to a system that uses connected components to discover tables [23], an approach for genetic algorithm-based table recognition [63], and a CNN-based machine learning model [41].

3.4.1 Evaluation datasets and their properties

To evaluate our approach, we use two datasets of real-world spreadsheets. The first, Deco [62], is a publicly available annotated file sample of enterprise spreadsheets extracted from the ENRON corpus [46]. It is composed of 1,165 MS Excel files used in an energy company and found in email attachments from 2000 to 2001, annotated by Koci et al. [62]. Of those, roughly 27% are classified by the authors as not containing a table (e.g., containing only charts). For the remaining 854 files, in the case of multiple worksheets per file, the authors annotated only one worksheet with regions. We use these regions as candidates for our region detection task. In addition, we manually annotated the dataset at the file level to identify files with the same layout, for the template inference task[6].

The second dataset is sampled from FUSE, a large-scale corpus of spreadsheets crawled from various internet sources [108]. For our evaluation, we annotated the region layout and the templates of all relevant 886 worksheets from 780 unique, randomly sampled spreadsheet files. In the remainder of this section, we call this annotated subset Fuste (FUSE Sample for Template Extraction). The region-level annotations of Fuste have been obtained with the tools proposed in the original Deco paper [62], to stay consistent with those from this dataset.

Table 10 reports the main characteristics of the two datasets concerning their files' layouts. The first consideration is the wide presence, in both sources, of multiregion files: roughly 72% and 45% of files from Deco and Fuste, respectively, have more than one region. Fuste has overall a greater number of single region files and on average much fewer regions per file than Deco (2.09 and 4.43, respectively), with Deco having more files with a huge number of regions — the maximum being 321. For the rest of the experiments, we regard as outliers, and therefore exclude, those files with more regions than the 99.9% of the remaining files in the same dataset. These files, two for Deco and one for Fuste, were characterized by an unusually large number of regions sparsely distributed across the spreadsheet. The two datasets also show opposite natures regarding layout templates. Deco has a low level of layout recurrence, with 750 different

[6] `https://github.com/HPI-Information-Systems/Mondrian` accessed Nov 3, 2021

	Deco	Fuste
Total number of files	854	886
Regions across all files	3,785	1,857
Files with one region	233	495
Files with multiple regions	621	391
Average num. of regions per file	4.43±12.19	2.09±1.70
Maximum num. of regions in a file	321	20
Overall layout templates	750	136
Templates with one file	679	105
Templates with more than one file	71	31
Average num. of files per template	1.13±0.59	5.33±32.35
Maximum num. of files per template	12	381

Table 10: A synthetic overview of the evaluation datasets.

layout templates for 854 files, 679 of which are "singletons", i.e., covering only one file. Fuste, on the other hand, contains 136 templates for 886 files, with one encompassing as many as 381 different files and only 105 singleton templates.

Various considerations arise from these fundamental differences. First, both manually annotated datasets represent a relatively small sample of the entire collection of files from the original corpora: ENRON, the source of Deco, is composed of 15,770 unique spreadsheet files with 79,983 sheets [46]; FUSE, the source of Fuste, has 249,376 unique spreadsheets. Possibly, the templates we discovered may cover many more spreadsheets than in our sample. Additionally, the origin and thus usage of the two datasets are different: Deco is a set of enterprise spreadsheets, in which files have possibly a "single-use" scope, be it for reporting or analysis purposes within the company; Fuste is a set of documents crawled from various public internet sources, most likely designed for sharing, with a high homogeneity of files originating from the same source. Finally, as surfaced during our annotation of Deco templates, this dataset could have shown a higher percentage of file similarity with a different choice of the worksheets: the choice of annotating only one worksheet per file excluded various worksheets from the original files that showed the same layout.

3.4.2 Experimental setup

The experiments conducted to evaluate the performance of our region detection approach include, for comparison, the results obtained on the same task using the connected component detection algorithm outlined in the work of Coletta et al. [23], the genetic-based table recognition approach proposed by Koci et al. [63], and the CNN-based TableSense [41]. Furthermore, simply selecting the connected component approach from Coletta et al. can be considered a baseline for our approach: it is the first step from which we build upon element partitioning and clustering.

Genetic-based approach: The genetic-based approach is a more sophisticated process, involving two steps that rely on supervised machine learning methods. In the

first step, a random forest classifier is trained on cell features to label each spreadsheet cell according to its role (e.g., data, header, aggregate) [64]. Afterward, neighboring cells with the same label are grouped and a graph is formed, with cell groups as vertices and their spatial relationship as edges [61]. Different tables are recognized as sets of vertices obtained by partitioning the graph [63] using a supervised genetic-based algorithm. This overall approach relies on rich features extracted from Excel files and aims at solving the more complex task of table recognition. Recall that the region detection task we solve is slightly different in goal and assumptions: we are interested in detecting region boundaries in general multiregion spreadsheets, without assuming special formatting features or any tabular structure.

The comparison was conducted with the help of the original authors, reusing the source code for the feature extraction, cell classification, and the genetic approach[7]. For a fair comparison, we experimented with two versions of the genetic-based approach: one using the full set of Excel-specific features available, and one restricting the input information to only cell content and position, excluding style features, thus simulating a *.csv* file input. The model, following the setup described by the authors in [63], is trained and tested on each dataset using 10-fold cross-validation.

TableSense: TableSense, proposed by Dong et al. [41], is based on Mask R-CNN [43], a convolutional neural network developed for instance segmentation in images. TableSense extends this architecture for the task of table detection in spreadsheets with two specialized modules: a feature extraction stage to map spreadsheets into feature maps that fed as input to the network, and a Precise Bounding Box Regression layer to refine the coordinates of Mask R-CNN detected regions' bounding boxes. The intuition of TableSense, like Mondrian, is to map the region detection task to the visual domain: using a convolutional architecture, it leverages the 2D distribution of cells on a spreadsheet to identify "Regions of Interest", candidate areas of the input file, which are then classified as tables and whose boundaries are refined by the PBR module.

The authors report experimental results of TableSense training the model on the WebSheet10K dataset and testing it on the WebSheet400 dataset. As neither the trained models nor the original source code is publicly available, to compare it with Mondrian in a similar setup we obtained the results training the model on one dataset and testing on the other, i.e., the results for Deco are obtained training TableSense on Fuste and vice-versa. Due to the non-deterministic nature of the approaches that involve machine learning approaches (Genetic-based and TableSense), we repeated the experiments involving the full pipeline three times, and report average scores, with confidence intervals obtained from the standard deviation of the experiment results.

Mondrian: For the region detection stage of Mondrian, we use two setups regarding the choice of the clustering radius: one using an optimal, "dynamic" choice of the clustering radius for each file, and one with a "static" radius used across all dataset files. In the dynamic radius setting, we ran our clustering method on each file, varying the size of the radius between [0.1,2] in steps of 0.1, between [2,10] in steps of 1; and between [10,100] in steps of 10. Additionally, we experimented with different configurations of

[7] `https://github.com/ddenron/gen_table_rec` accessed Feb 25, 2020

the distance features' weights: we kept $\alpha = 1$ as a fixed reference value and varied $\beta, \gamma \in \{0, 0.5, 1, 5, 10\}$. The hyperparameter configuration that showed the best results was $\alpha = 1, \beta = 0.5, \gamma = 1$ for DECO, and $\alpha = 1, \beta = 1, \gamma = 1$ for FUSTE. We use these values for experimenting in the "static" radius setting, in which we tried to find the single radius that showed the best performances across all files. The search space for the radii was the same as the one used in the dynamic setting. We report the result obtained using the radius with the best performance for each dataset, namely 1.5 for DECO and 1.4 for FUSTE.

3.4.3 Region detection accuracy

To evaluate the level of accuracy in region detection, we use the Intersection-over-Union score (IoU) and the Error-of-Boundary score (EoB), defined in [41]. The first value is the graphical equivalent of the Jaccard index for sets. If we define P as the set of non-empty cells of a predicted region, and T as the set of non-empty cells of a target region, the IoU is calculated as:

$$IoU(P,T) = \frac{|P \cap T|}{|P| + |T| - |P \cap T|} = \frac{|P \cap T|}{|P \cup T|}$$

The EoB score is the maximum distance, expressed in number of cells, between any of the top, bottom, left, or right boundaries of a predicted region and those of the target region. To calculate the EoB, if we define a region's top-left coordinates as (x_0, y_0) and bottom-right coordinates as (x_1, y_1), for two regions P and T, we use the following formula from [41]:

$$EoB(P,T) = \max(|P_{x_0} - T_{x_0}|, |P_{y_0}, T_{y_0}|, |P_{x_1}, T_{x_1}|, |P_{y_1}, T_{y_1}|)$$

An IoU score of 1 corresponds to perfectly detected regions and a score of 0 to missed regions. A perfectly detected region has an EoB of 0, with no upper limit for incorrect detections. EoB is undefined in the case of no detected region: whenever such a case arises, we set the EoB as the maximum of the height and width of the file, simulating a completely out-of-boundary detection. The standard in literature is to consider correctly detected all true regions for which the score of at least one predicted region exceeds a given threshold [35, 41, 63]. To provide more accurate results, we measure actual scores rather than their binarization. In general, any of the true regions R_T of a file can be split into multiple R_P predicted regions, or vice-versa, one of the predicted regions can span multiple true regions. Therefore, for M predicted regions and N true regions IoU determines $M \cdot N$ scores: to achieve only one value for a given true region, we assign it to the predicted region with the highest overlap:

$$IoU(T) = \max_{P \in R_P} IoU(P,T)$$

$$EoB(T) = \min_{P \in R_P} EoB(P,T)$$

Figure 17 shows the performance of the different approaches over varying thresholds: the y-axis represents the percentage of tables or regions correctly detected in the two datasets,

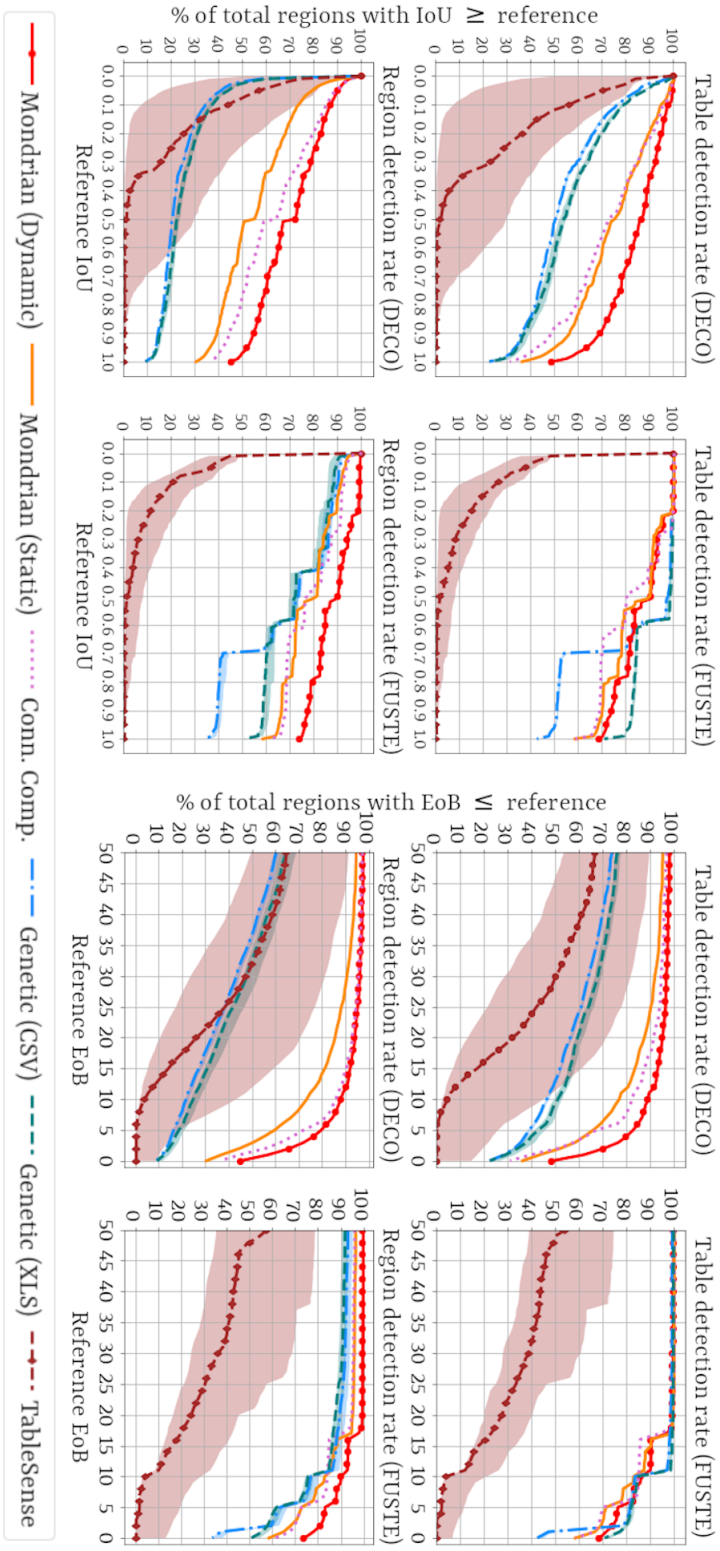

Figure 17: Table and region detection performance.

assuming as "correct" a score better than the given reference on the x-axis. We report the performance for tabular regions only ("table detection"), and the performance across all types of regions ("region detection"), which include tables but also notes, spreadsheet titles, etc.

Mondrian's performance: The best results for all regions are obtained, for both datasets, with our clustering approach assuming a dynamic, optimal choice of the radius for each file. It is interesting to note the difference in the behavior of Mondrian on the two different datasets. DECO, which contains more multiregion files and on average more regions per file, proves to be the harder of the two with approximately 45% of regions perfectly detected (100% IoU). On FUSE, instead, with fewer complex multiregion files, around 75% of the regions are correctly detected. The usage of a static radius yields lower performance: in the case of tables, the accuracy is comparable to detecting connected components, while on other region types, it yields slightly worse results. In our experiments, a smaller radius (≤ 1) made the clustering degenerate into connected component detection, grouping only adjacent partitioned elements. A larger radius, such as the one selected for our static approach (namely 1.5), improves table detection, since a high number of tables is composed of separated connected components, but also brings together different non-tabular regions, which are usually independent. Because of this, the static radius variant of our clustering approach shows slightly worse performance in detecting general regions than tables.

Comparison with the genetic-based approach: It is not surprising that the genetic-based approach shows better results for tables than for generic regions, as it was specifically designed for table recognition. When cell classification and table detection are combined end-to-end, the second step proved to be sensitive to even small errors in the cell classification, with the results visible for the DECO dataset in Figure 17. On FUSTE, where the classification errors were minimal, the genetic-based approach showed much better results. We explain this phenomenon by considering the reliance of the genetic-based search on correctly labeled region boundaries. The incorrect classification of some cells causes the split of one single region into different vertices, some of them necessarily erroneous. Moreover, it appears that non-data cells, such as header or aggregation cells, are crucial for recognizing tabular structure. Such classification errors propagate into unreliable weight learning for the quality measures of the fitness function and finally cascade into poor table boundaries. It is worth noting how, for FUSTE, the contribution of Excel-specific features is much more significant than for DECO: the gap between the two versions of the genetic approach is much wider.

Comparison with TableSense: The results of TableSense show low performance with a high variance. This behavior can be explained by noting the considerable number of regions that are completely missed: on average, 48.81% for DECO and 32.92% for FUSTE. Contrary to Mondrian, which by design does not ignore any non-empty input cell, the CNN architecture of TableSense may completely ignore entire areas of the input if they are not considered "Regions of Interest" or classified as containing an object. This behavior is inherited from the original domain of Mask R-CNN, designed for instance segmentation of images, which may or may not contain relevant objects. Overall, the poor accuracy of TableSense is most likely due to the high complexity of the model, which

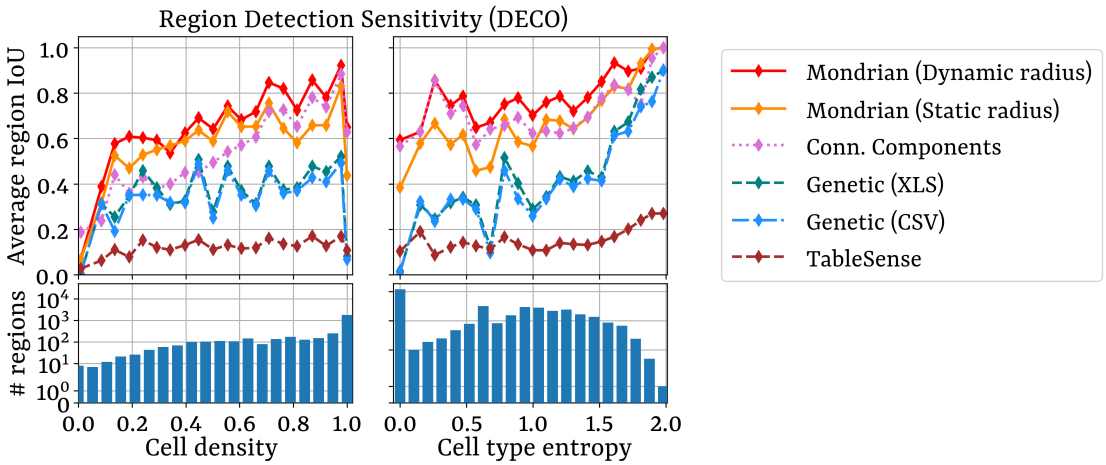

Figure 18: Performances per region composition.

is composed of more than 85 million trainable parameters, and the limited number of training files available for our use case.

Comparing the plots across the two datasets, we note an interesting difference: the DECO plots are much smoother than those of FUSTE, which show abrupt drops in the percentage of tables and regions recognized above a certain threshold. This phenomenon reflects the different dataset natures, as analyzed in Section 3.4.1. Considering that DECO has roughly twice the quantity of regions compared to FUSTE, it is natural to expect a more continuous plot. What is more, FUSTE contains a greater number of files that share the same templates: on average, 5.33 files share the same layout compared to the 1.13 in DECO. In particular, one can observe how the percentage of tables (and regions) detected correctly in FUSTE drops from 80% (60%) to 50% (40%) for the Genetic-CSV approach as soon as the threshold for the IoU is increased from 69% to 70%. Looking for the causes of this behavior, we found that 323 different regions, coming from just as many files with the same layout, were detected in the same way. The absence of the same drop from the Genetic-XLS approach suggests that including style features helped in recognizing these regions correctly.

Sensitivity to region composition: Considering the graphical nature of the clustering performed by Mondrian, its performance on region detection is sensitive to the visual composition of regions. To provide insights into the behavior of the different region detection strategies, we analyzed the effect of two variables: the density of a region, i.e., the ratio of non-empty cells to empty cells contained in a region, and the cell type entropy, i.e., the entropy of a region, which we calculate as $-\sum_{i=1}^{k} P(c_i) \cdot \log P(c_i)$, with $P(c_i)$ being the ratio of cells of (syntactic) type i over the total cells of a region. Figure 18 reports the average IoU scores of the regions of the DECO dataset sorted by their density and entropy. Both plots show that Mondrian is most successful with visually heterogeneous regions: its performance increases with increasing cell type entropy and has a sharp drop for regions with either very low densities, signaling a high number of empty cells, or a low cell type entropy, where it is unable to perform its partitioning. We note that the low score for regions with a density of 1, i.e., with no empty cells, is

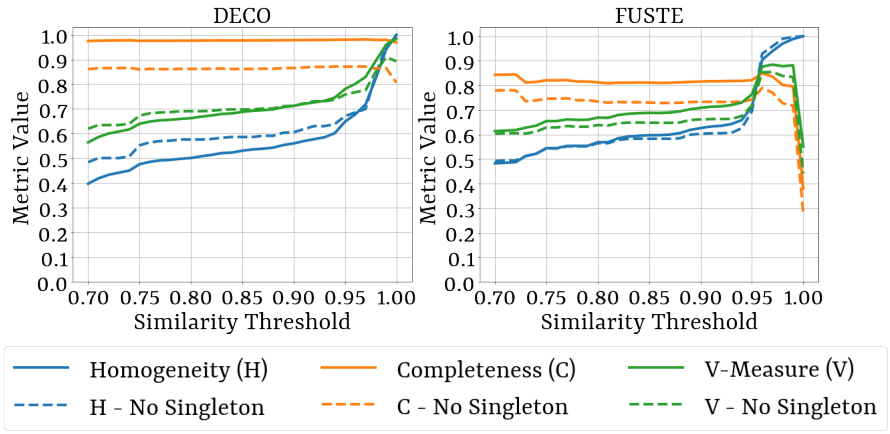

Figure 19: Performance of Mondrian on template inference.

highly correlated to the score for an entropy of 0, as 1 192 out of the total 3 462 regions have both a density of 1 and an entropy of 0. This behavior reflects the inefficiency of visual partitioning for regions with few "visual irregularities". In fact, these regions are those where the connected component baseline outperforms Mondrian.

3.4.4 Template inference accuracy

In evaluating the template inference task, we rely on three external measures for clustering: *homogeneity*, *completeness*, and *v-measure* [92]. The value range of all three scores is [0,1], with 1 being a perfect result. Using the gold standard, homogeneity quantifies how many data points in each predicted cluster belong to the same template. For our problem, in a perfectly homogeneous solution, all files that are grouped indeed share the same layout. Completeness, conversely, quantifies the percentage of elements from the same template that are grouped. V-measure is the harmonic mean of homogeneity and completeness. As described in Section 3.3.3, we group files transitively based on their layout similarity being above a given threshold. We experimented with thresholds in the range [0.7,1] and a spacing of 0.01. To save computational time while repeating the experiments for different thresholds, we did not calculate the layout similarities of pairs for which we can guarantee a threshold lower than 0.7. This pruning was possible given the nature of our approach, where the similarity of two graphs is bound by the absolute difference in their number of nodes, normalized by the maximum number of nodes across the two graphs.

Effect of layout similarity threshold: Figure 19 shows the influence of the threshold value on the results of template recognition using the regions automatically detected by Mondrian in the static radius scenario, for the Deco and Fuste datasets. Considering how, especially for Deco, there is a significant number of singleton templates, i.e., templates that occur in only one file, we report the results of our template recognition approach for the full dataset as well as for the sub-set of files that constitute non-singleton templates (175 files for Deco and 781 for Fuste, See Table 10).

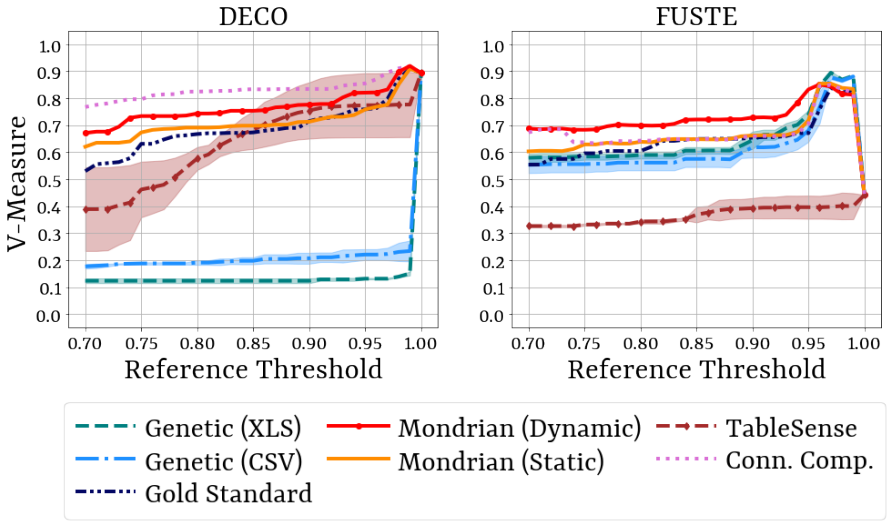

Figure 20: Effect of region detection on template inference.

Increasing the threshold leads to a more selective behavior: for the maximum threshold of 1, homogeneity reaches a perfect value, as the resulting templates are always comprised of one file and therefore trivially homogeneous. This is compensated by the drop in completeness for high thresholds, especially noticeable in the FUSTE dataset. This effect is mitigated on the full DECO dataset thanks to the high number of singleton templates. Overall, the performances of our template inference approach benefit from choosing high thresholds: across the two datasets, the best v-measures are obtained with thresholds between 0.95 and 1.00.

Sensitivity to number of regions To assess how the region composition of file layouts affects the template recognition performance, we partitioned the evaluation datasets into three groups: single region files, files with few regions (2 to 5), and files with many regions (more than 5). In Table 11 we report the scores obtained by Mondrian on the three partitions using a threshold τ_f of 0.99. Across both datasets, the best performances are reached on files with many regions. Conversely, the lowest homogeneity is obtained on single region files, where the layout graphs contain no edges (or presumably a few, due to errors in region detection). This causes layout similarity to be mostly influenced by the approximate region similarity, which causes a slight increase in false positives.

Sensitivity to region detection strategy: The performance of our template inference algorithm is also dependent on the results of the prior region detection phase. To analyze the sensitivity of the graph matching to region boundaries, we experimented with all the region detection strategies considered in Section 3.4.3 plus a configuration using the manually annotated regions from the gold standard. Figure 20 reports the v-measure for the different region detection strategies and baselines across datasets (excluding singleton templates). First, we highlight how approaches with poor region detection performances lead to low template recognition accuracies, most likely due to building graphs for files with a high percentage of misclassified regions. As mentioned

Deco ($\tau_f = 0.99$)

# regions	#files	H	C	V
1	232	0.92	0.97	0.94
$[2, 5]$	470	0.97	0.98	0.98
≥ 6	150	0.99	0.98	0.99

Fuste ($\tau_f = 0.99$)

# regions	# files	H	C	V
1	495	0.98	0.68	0.80
$[2, 5]$	372	0.98	0.76	0.86
≥ 6	18	1.00	0.95	0.97

Table 11: Template inference at varying number of regions.

previously, the high v-measures reached by all strategies at a threshold of 1 are distorted due to perfect homogeneity (all files are clustered individually). Surprisingly, for lower thresholds, using gold standard regions does not lead to better results. We attribute this effect to the increased complexity of the graphs produced with suboptimal regions: as there may be potentially more automatically detected regions than needed, the resulting graphs contain more (noisy) information and therefore show a greater absolute difference in the case of different templates.

3.4.5 Scalability of template inference

Different region detection strategies not only influence the effectiveness of Mondrian's template inference step but also affect its complexity, measurable on the runtime. We report the execution times for the template recognition task in Table 12, obtained as the average run-times of our Python 3.8 scripts across three separate runs on a machine equipped with an AMD Epyc 9 7702P Xeon 3,35 GHz CPU and 512 GB of RAM. The results highlight the tradeoff between template inference accuracy and complexity: the region detection strategies that proved to be better for template inference in Figure 20 are also the ones that need significantly more time to execute while using the region detection results of the genetic-based and TableSense approaches leads to lower running times and more imprecise results. When incorrectly detecting regions, Mondrian has a higher number of graph regions due to its partitioning steps: larger graphs need greater time for computation but lead to more precise similarity estimates. The slowest runtimes on Fuste are obtained by Mondrian in the dynamic radius scenario, because of a few files containing many nodes (above 200) that lead to expensive graph similarity computations. For this dataset, both the static radius and connected component strategies are faster because having a fixed radius and no region partitioning leads to fewer detected regions. Comparing Mondrian across datasets, the runtimes on Deco are lower: as this dataset is characterized by fewer templates, more file pairs with no similar region are pruned. The same pruning strategy is less effective for connected components on Deco because without a clustering stage there are more spuriously similar regions across files.

Figure 21 shows the influence of the number of files and percentage of empty cells in files on the computational time of template detection using perfectly recognized regions. For

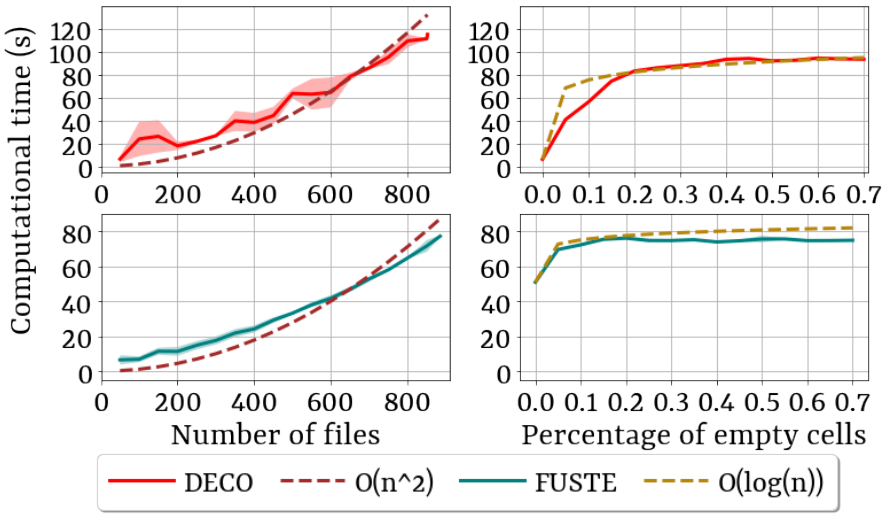

Figure 21: Effect of the number of files and empty cells.

Strategy	**Template inference time** (s)	
	DECO	FUSTE
Gold standard	93.39±0.26	78.87±0.77
Dynamic radius	1 563.51±2.91	8 515.46±194.55
Static radius	343.13±3.81	2 749.20±13.04
Conn. Comp.	15 887.50±127.12	3 529.21±76.67
Genetic (XLS)	102.32±0.51	75.12±0.96
Genetic (CSV)	114.76±1.58	75.13±0.34
TableSense	361.46±47.47	51.54±9.37

Table 12: Time performance of template inference.

the former, we experimented by selecting random file sub-samples, with a step size of 50. For the latter, the sub-samples corresponded to all files with a number of empty cells up to a given percentage of the total file area, with a step size of 0.05%. In both cases, the file sets were sampled without repetition until full coverage of the dataset. The plot shows that the performances with respect to the number of input files follow a quadratic behavior, as Mondrian performs layout comparison for each pair of files in the input set. In turn, increasing the percentage of empty cells leads to a logarithmic behavior.

Therefore, we conclude that the most impactful factor affecting the complexity of Mondrian is the number of input files, as well as the correctness of the region detection stage: detecting regions and multiregion file templates automatically with Mondrian provides a convenient tradeoff between complexity and correctness.

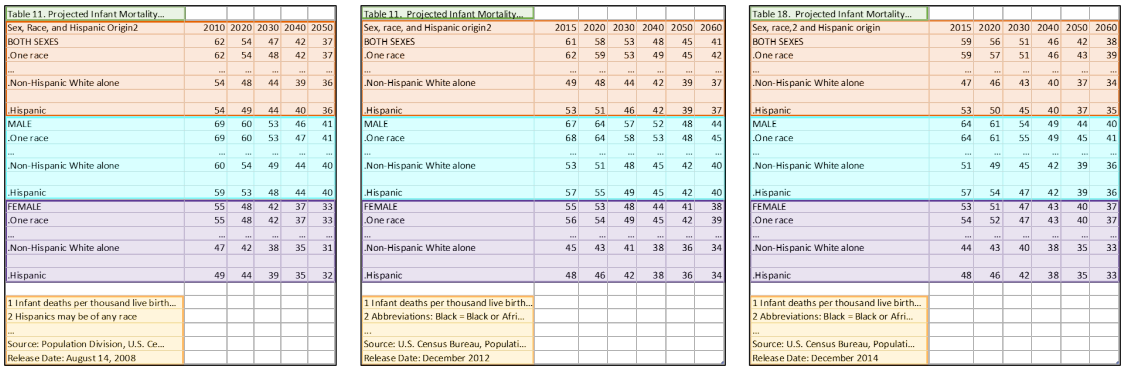

Table 11. Projected Infant Mortality...					
Sex, Race, and Hispanic Origin2	2010	2020	2030	2040	2050
BOTH SEXES	62	54	47	42	37
.One race	62	54	48	42	37
...	...	...	...	...	...
.Non-Hispanic White alone	54	48	44	39	36
.Hispanic	54	49	44	40	36
MALE	69	60	53	46	41
.One race	69	60	53	47	41
...	...	...	...	...	...
.Non-Hispanic White alone	60	54	49	44	40
.Hispanic	59	53	48	44	40
FEMALE	55	48	42	37	33
.One race	55	48	42	37	33
...	...	...	...	...	...
.Non-Hispanic White alone	47	42	38	35	31
.Hispanic	49	44	39	35	32
1 Infant deaths per thousand live birth...					
2 Hispanics may be of any race					
...					
Source: Population Division, U.S. Ce...					
Release Date: August 14, 2008					

Table 11. Projected Infant Mortality...						
Sex, race, and Hispanic origin2	2015	2020	2030	2040	2050	2060
BOTH SEXES	61	58	53	48	45	41
.One race	62	59	53	49	45	42
...	...	...	...	...	...	...
.Non-Hispanic White alone	49	48	44	42	39	37
.Hispanic	53	51	46	42	39	37
MALE	67	64	57	52	48	44
.One race	68	64	58	53	48	45
...	...	...	...	...	...	...
.Non-Hispanic White alone	53	51	48	45	42	40
.Hispanic	57	55	49	45	42	40
FEMALE	55	53	48	44	41	38
.One race	56	54	49	45	42	39
...	...	...	...	...	...	...
.Non-Hispanic White alone	45	43	41	38	36	34
.Hispanic	48	46	42	38	36	34
1 Infant deaths per thousand live birth...						
2 Abbreviations: Black = Black or Afri...						
...						
Source: U.S. Census Bureau, Populati...						
Release Date: December 2012						

Table 18. Projected Infant Mortality...						
Sex, race,2 and Hispanic origin	2015	2020	2030	2040	2050	2060
BOTH SEXES	59	56	51	46	42	38
.One race	59	57	51	46	43	39
...	...	...	...	...	...	...
.Non-Hispanic White alone	47	46	43	40	37	34
.Hispanic	53	50	45	40	37	35
MALE	64	61	54	49	44	40
.One race	64	61	55	49	45	41
...	...	...	...	...	...	...
.Non-Hispanic White alone	51	49	45	42	39	36
.Hispanic	57	54	47	42	39	36
FEMALE	53	51	47	43	40	37
.One race	54	52	47	43	40	37
...	...	...	...	...	...	...
.Non-Hispanic White alone	44	43	40	38	35	33
.Hispanic	48	46	42	38	35	33
1 Infant deaths per thousand live birth...						
2 Abbreviations: Black = Black or Afri...						
...						
Source: U.S. Census Bureau, Populati...						
Release Date: December 2014						

Figure 22: Detailed view of three real-world files sharing the same multiregion layout.

3.5 Data preparation with Mondrian

In this section, we demonstrate the interactive components and the graphical interface of Mondrian following a demonstration scenario. The Mondrian system with the dataset we use in this section, plus two additional datasets for exploration, is available online[8].

Consider a data practitioner interested in analyzing the historical United States population data, retrieved from the open data portal of the United States Census Bureau[9]. The summary tables for the censuses of several years are available in spreadsheet format, as a collection of CSV and XLS files. The files that contain the same tables all share the same layout: they have similar title and footnote cells, and all their tables (when more than one) have the same schema (see Figure 22).

The goal of the practitioner is to consolidate the information contained in different tables in a single source of truth, e.g., a relational database, to enable querying, analysis, and visualization. Because there are slight differences in the files across years, automatic extraction is not trivial. To isolate the tables, the practitioner needs to inspect each file individually and split it according to its structure.

Considering the three files in Figure 22, for example:

- in the footnote region, the last cell reflects the year, and sometimes cells have different content while the semantic meaning is the same (E.g., “Source: Population Division, U.S. Census Bureau” and “Source: U.S. Census Bureau, Population division”)
- The tables themselves have a different number of columns across files, and their headers are updated.
- Because of different table lengths, footnotes appear in lines 41–47 in one of the files and in lines 43–49 in another.
- The table title also changes from “Table 11” to “Table 18”.

Nonetheless, it is obvious at a glance that the three files come from the same layout template.

[8] `https://hpi.de/naumann/sites/mondrian/demo`

[9] `https://www2.census.gov/programs-surveys/popproj/tables/`(accessed Jul. 15, 2023)

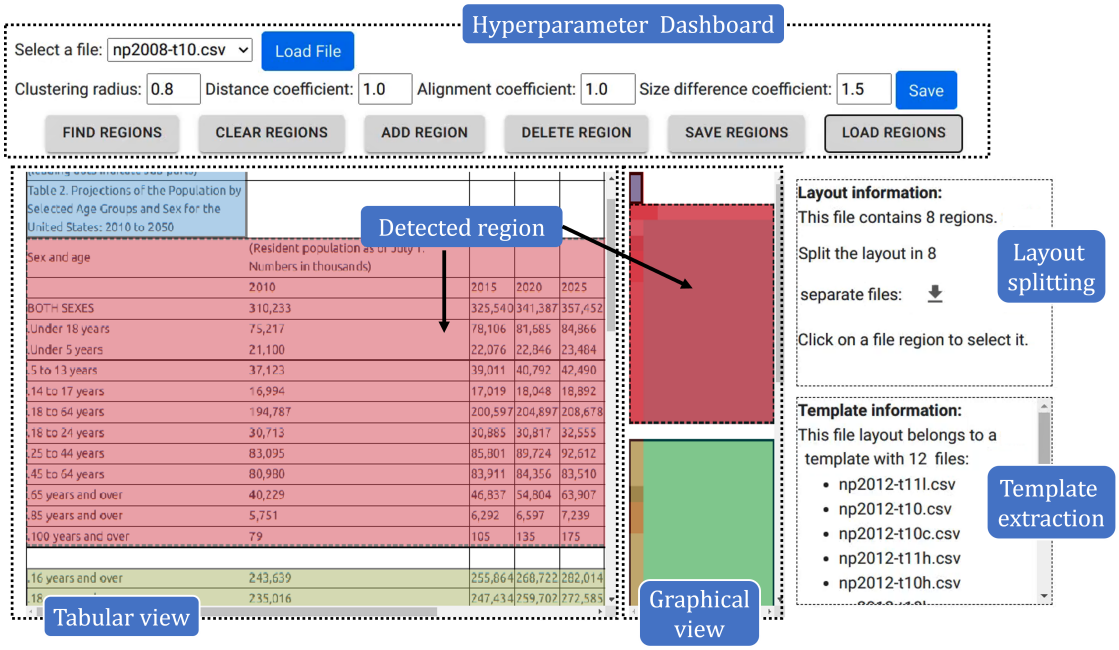

Figure 23: Region detection page with file layout regions.

To automate the preparation of this spreadsheet dataset, the practitioner can exploit Mondrian's user-friendly web interface, based on the unsupervised layout and template detection approach described in the previous sections. At the time of file upload, Mondrian automatically performs the image parsing and region detection steps. The results of these processes can be viewed on the *Region Detection* page (Figure 23), which displays the content of a single file, both in tabular format and as a *Mondrian* image (Section 3.3).

Colored rectangles represent the boundaries of the automatically detected regions both on the tabular and the graphical file view. Clicking on a rectangle prompts an *edit* mode, where the user can interactively change the detected regions' boundaries by performing drag-and-drop actions. Additionally, entire regions can be added and removed. For ease of use, all interactive actions can be performed either on the tabular view or on the compact graphical view. To leverage the results of the region detection phase and assist users in their further preparation of single multiregion spreadsheets, this page also offers a *split* feature to create a separate file for each of the detected regions. Moreover, if the layout of the file belongs to a template, users can select a given region and extract that region from all files of the same template, using the procedure described in Section 3.3.

To analyze an entire collection of uploaded files and exploit the automated Mondrian layout template matching, users can resort to the *Template Detection* page (Figure 24). The page shows two panels: the panel on the left contains a list populated with the various layout templates found in the dataset. Each entry contains the names of the files sharing the same template. The page allows interactive manipulation of the list: with drag-and-drop actions, users can move files from one template to another, as well

as create new, empty templates. From the list, each template can be downloaded as a ZIP file containing all its files.

The panel on the right contains three tabs with additional visualizations for templates and their files: the *details* tab, the *gallery* tab, and the *compare* tab. The first tab shows the details of a given template selected from the list. Among other information, it contains the pairwise layout similarity of the files belonging to a selected template. By inspecting these scores, users can gain a more in-depth understanding of the reasons that led Mondrian to its clustering decisions. The second tab, selected in Figure 24, displays a gallery view of the renderings for all the files belonging to that template. The gallery shows the Mondrian images for all files that belong to a template. Even from a cursory glance, it is possible to validate the results of the template detection and spot possible *outliers* by visual inspection of the images shown in the gallery.

From the list on the left or from the gallery view, users can select pairs of files for further inspection. A detailed comparison for the selected pair of files then becomes available in the third tab of the right panel (Figure 25). Here, users can find information about the number of detected regions for each of them, as well as their layout similarity. Additionally, this tab presents a side-by-side comparison of the two files' data content and their image rendering.

Finally, to leverage the information about the discovered layout templates, Mondrian can provide users with a downloadable CSV file that contains the template statistics for the dataset: a list of files with their corresponding template, and the average layout similarity to every other file in the same template.

In our US Census Population scenario, out of 99 different spreadsheet files, the system identified 15 different layout templates. A histogram of the number of files per template is shown in Figure 26. The median and mode of the distribution are both 5, as most templates contain 5 files. The dataset contains 4 different singleton templates, and the template with the highest number of files contains 27 files. All non-singleton templates, except for one, contain files from different years. Upon further inspection, two pairs of singleton templates had files with the same structure, but one contained floating-point numbers and the other integer numbers. After interactively readjusting these templates, and knowing which files share the same layout, the user can trivially extract the data tables and consolidate them in a relational database.

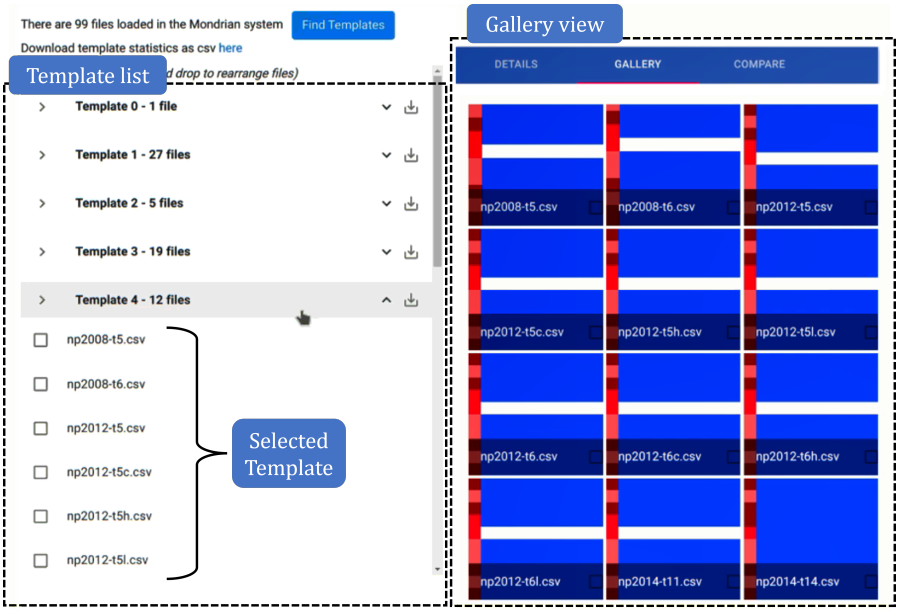

Figure 24: Template detection page: gallery view.

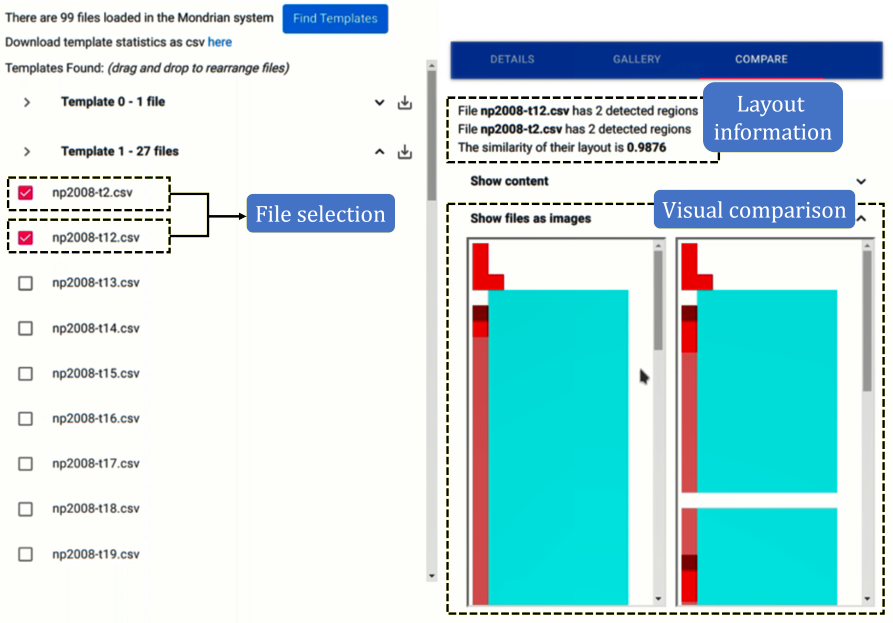

Figure 25: Table detection page: comparison view of two files.

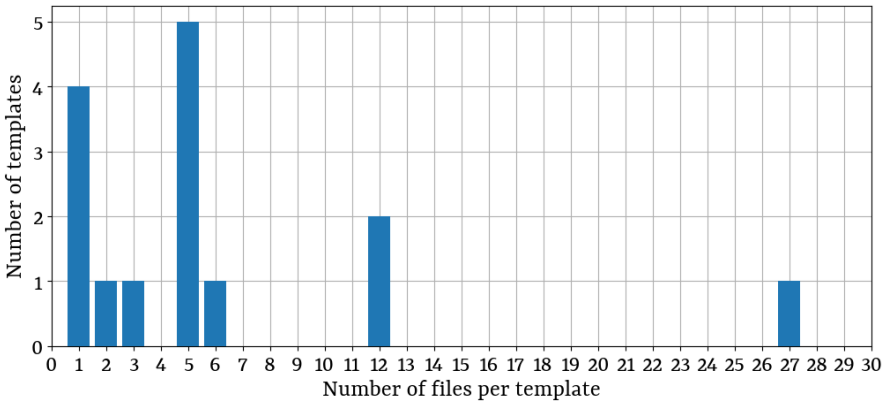

Figure 26: Histogram of the number of files per template.

3.6 Summary

In this chapter, we proposed Mondrian, a system that leverages (1) an explicit graph-based representation of file structure to identify the layout of a multiregion file; and (2), an implicit, similarity-based representation to identify the occurrence of templates within a collection of files. Thanks to these representations, we assist users in the data preparation process of multiregion datasets, by automatically detecting the boundaries of the same regions across different files.

Experiments show that our approach works well in detecting the boundaries of different regions in a multiregion spreadsheet and in identifying layout templates, even though further research can be done to improve the accuracy of the results.

Future work may focus on a finer-grained structure similarity computation, e.g., having more semantic types for cell contents, to better identify structural patterns and correlations within templates. Moreover, an interesting direction is providing explicit representations for layout templates, given their usefulness for a variety of data preparation tasks.

Chapter 4

MaGRiTTE: a Machine-Learning Model for File Structure

In this thesis, we argue how all data preparation steps share the over-arching goal of understanding and transforming the structure of a file so that its payload can be parsed correctly. As motivated in Section 1, data scientists often address data preparation problems individually, applying several tools and solutions.

Some examples of tasks addressed with ad-hoc solutions are:

- inferring the *dialect characters* of a file and parsing its payload [12, 28];
- classifying the *row type* of file rows according to their structural role (header, data, footnote, comment, etc.), using row-level embeddings [40, 56, 84];
- extracting multiple tables that span multiple rows or columns [21, 116].

Contrary to specific frameworks for individual problems, in this Section we present a unique, task-independent model to represent file structure, pre-trained on a large and structurally diverse set of files. With such a general model, file structure can be represented in vectorial embeddings that can be used either by the model itself to address structural preparation, or as external features to enrich other specialized models.

Inspired by the success of representation learning and pre-trained models in fields like natural language processing [87, 91, 112] and computer vision [42, 44, 102], we propose MAGRiTTE, a framework to encode cell-level, row-level, and file-level structure of tabular files as vectorial embeddings. The name stands for MAchine Generated Representation of Tabular files with Transformer Encoders.

The motivation for a specialized architecture stems from the fact that existing approaches for representation learning, including general-purpose large language models [27, 87] and specialized tabular data models [52, 101, 104, 120, 123], focus on downstream tasks operating on the payload of files and therefore assume a correct parsing of file structure. We experimented with such models (as reported in Section 4.1) but found them inadequate to address structural preparation tasks.

MAGRiTTE is a pretrained model, composed of transformer and convolutional layers, specialized to represent file structure for data preparation. We train this large model in

a self-supervised fashion, leveraging almost 1M real-world tabular files from the GitTables corpus [50]. We presented the initial idea as an abstract in [114]. This chapter is based on a full paper currently under review. Specifically, the contributions of this chapter are:

1. The MAGRITTE architecture: a large deep neural network model aimed at representing the structure of CSV data files in high-dimensional vectorial embeddings, with a novel pattern tokenization strategy and two novel pre-training tasks, *structural masking* and *same file prediction*.
2. Fine-tuning strategies to apply the main MAGRITTE model to four preparation tasks: dialect detection, table understanding, column type annotation, and preparation effort estimation, with their experimental analyses. The latter is a novel problem, identifying the effort required to transform the structure of a source file into that of a target file.
3. A dataset of 875 real-world files with their annotated column types, a dataset of manually annotated 100 real-world source-target file pairs with their preparation scripts, all the trained models with their weights, and their code[1].

We organize the discussion of the rest of the chapter as follows: in Section 4.1, we discuss our initial trials on using LLMs for structural preparation, and motivate the need for a specialized framework in light of their shortcomings; in Section 4.2, we discuss the main architecture of MAGRITTE, its components, and training objectives; Section 4.3 proposes three different fine-tuning strategies to apply MAGRITTE for the data preparation tasks of dialect detection, table understanding, and data preparation effort estimation; Section 4.4 presents the results of our experimental evaluation; in Section 4.5, we discuss how MAGRITTE compares with related work; we conclude the chapter with a summary and an outlook of future work in Section 4.6.

4.1 Data preparation with LLMs?

With the advent of Large Language Models (LLMs) like those in the GPT family [87], recent research has experimented with the use of these models for traditional data wrangling/cleaning tasks, for example in [76]. The intuition of this approach is to use a pre-existing LLM and perform zero-shot or few-shot inference to solve data management tasks. We performed an exploratory analysis using the state-of-the-art LLM GPT3.5 (in its version *davinci-003*, like in [76]) to explore its capabilities to solve structural tasks like dialect detection and row classification in a CSV file. To find the best prompt for the model, we spent at most one hour per task and then ran queries to test dialect detection on a subsample of 100 files from the ones we use in Section 4.4.2, and to test row classification on a subsample of 120 files from the ones we use in Section 4.4.3. A selection of the prompts and responses can be seen in Figure 27. Unfortunately, our experience dissuaded us from pursuing this approach for several reasons:

Input considerations: The model proved quite sensitive to the prompt and file content. In the example shown in Figure 27, when asked to detect the "dialect" of a file, it responded with "`CSV (comma separated values)`", which not only does not correspond to the intended response but may also be considered factually incorrect as the input file

[1] `https://github.com/HPI-Information-Systems/Magritte`

values were separated by semicolon. However, when prompted to return the delimiter, quotation, and escape characters, it replied with a correct answer.

Input:

```
Identify the delimiter, quotation and escape characters of the following file.
File:
{file content}
Delimiter, quotation, escape:
```

Output (different input files):

```
Delimiter: (,); Quotation: ""; Escape: \
```

```
Delimiter: "," Quotation: "" Escape: None
```

```
Delimiter: ; Quotation: " Escape: \
```

Input:

```
Identify the dialect of the following file.
File:
{file content}
Dialect:
```

Output (different input files):

```
The file is a CSV file.
```

```
Ruby
```

```
CSV  (Comma Separated Values)
```

Input:

```
Header lines represent the column names of tables; data lines represent records;
group lines organize tables into sub-tables and are the header for a given group;
derived lines contain the result of some operation on data lines; metadata and note
lines contain metadata information respectively before and after tables.
In the following CSV file, identify what lines are header, group header, metadata,
note, and derived.
File:
{file content}
Lists of indices of header, data, group header, metadata, note, derived:
```

Output (different input files):

```
Header: 0, 1
Data: 2-130
Group Header: None
Metadata: None
Note: None
Derived: None
```

```
Header: ""Table dt.04.q: People leaving detention by reason and age, Q2 2011"",
Group Header: Reason,,,,,,,
Data: Total,""6,337"",""6,319"",18,9,2,2,5
Metadata: None
Note: None
Derived: None
```

Figure 27: Sample queries to GPT-3.5 with their output to address structural tasks.

Output considerations: The responses of the model, being in natural language, may be ambiguous and require parsing rules themselves. For example, in Figure 27, in one instance the response stated that the file quotation is a double double-quote mark `"""`. It is unclear if this corresponds to an empty value within quotes for formatting because the delimiter response `","` may suggest so. But the escape response, as well as other responses, suggests that otherwise null characters are indicated with the string `None`.

Generalizability considerations: Given the highly stochastic nature of LLM, and the fact that they have been trained on a massive set of data, two considerations hinder a serious experimental evaluation of their generalizability performances. First, the files used for experimenting have possibly already been seen by this model, being publicly available at the time of the training of these models. Second, the seemingly innocuous content of a file may lead the model to "hallucinate" outside the given prompt [54]. We stumbled on an example, highlighted in Figure 27, where an input file containing the text `Ruby` led the model to respond with `Ruby` as its dialect. Finally, large language models are tools specialized in natural language tasks, and it is unclear how well their embeddings can apply to unseen numeric data, as examined by [88].

Repeatability considerations Most state-of-the-art LLMs, like GPT [11] and PaLM [20], are proprietary and closed-source models, accessed through API calls. This limits any repeatability for experiments since there is no guarantee that, in the future, the internals of a model will not change (as they already did in the past years), or that the models themselves will still be available. While some attempts towards open-sourcing these architectures are underway [109], they still require significant hardware resources to reproduce their performances.

These considerations make LLMs unreliable and hard to integrate into a data management pipeline. Therefore, we resolved to pursue the design of a specialized framework for structural preparation: one that generalizes well with unseen files and is not sensitive to their content; not designed for natural language input/output but rather to be integrated with automated data management components; and that can be run on commonly available hardware. The next section introduces MAGRITTE, our framework that addresses these challenges and represents tabular file structure.

4.2 The MaGRiTTE architecture

The key intuition of MAGRITTE is to design a model that captures the structural information of files by abstracting away the details of their payload. To achieve this scope, we designed a specialized tokenization step for file rows that retains special characters and shrinks sequences of non-special values, creating *row patterns* that resemble regular expressions and abstract away details of the payload. In this way, our model is forcefully led astray from the semantics of data values and focuses on the structure of rows.

Row patterns are fed as input to a transformer-based encoder, modeled after BERT [27], which learns a representation for each of the rows of the file. We also introduce two new training objectives for pre-training: masked structural modeling (MSM) and same-file prediction (SFP). In the first task, we train the model to correctly predict special characters in file rows, which mark their structure; in the latter, we train the model

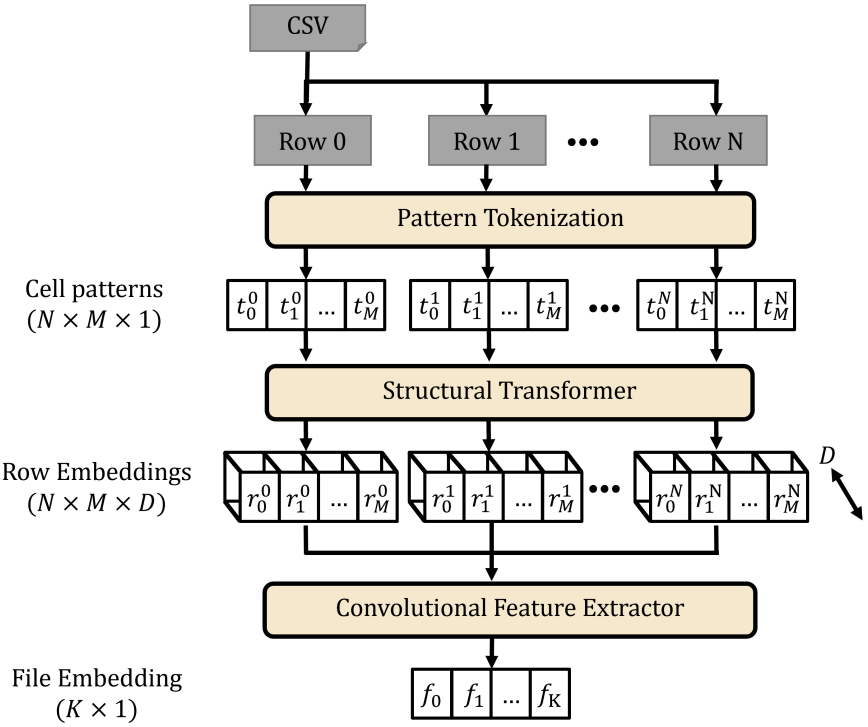

Figure 28: Three-tiered architecture of MaGRiTTE.

to classify whether two rows belong to the same file or not. Once row embeddings are produced by the transformer layers, a convolutional-based encoder based on ResNet [44] reduces row-level embeddings into a single, condensed representation to embed the general file structure. The convolutional encoder is trained together with a twin decoder, as an autoencoder.

The architecture of MaGRiTTE leverages three components, aimed at representing three levels of a tabular data file: *cells* with their sequence of characters, *rows* with their sequence of cells, and *files* with their sequences of rows. An overview of the whole architecture is presented in Figure 28. We briefly introduce the three components in the following paragraphs and then describe them in detail in the remainder of this section.

The training of our model does not require any previous knowledge about a file's dialect to identify tabular cells and rows, but rather uses unsupervised learning to understand its structure. To make this possible, the first step is what we call *pattern tokenization*: this step produces a fixed-length sequence of tokens for every row. In Figure 28, individual tokens are named t_0^0 to t_M^N, where N identifies the number of file rows, and M identifies the length of the token sequences. This step returns, for every row, a sequence of tokens where special symbols are left unchanged and alphanumeric characters are abstracted away. Explicitly assigning tokens to special characters and abstracting cell values promises to highlight the structural elements in the rows. This is syntactically relevant because cell values are typically surrounded by symbols like delimiters or quotations.

The next component is a transformer architecture to encode the tokenized rows. This architecture is composed of six attention layers with twelve attention heads each,

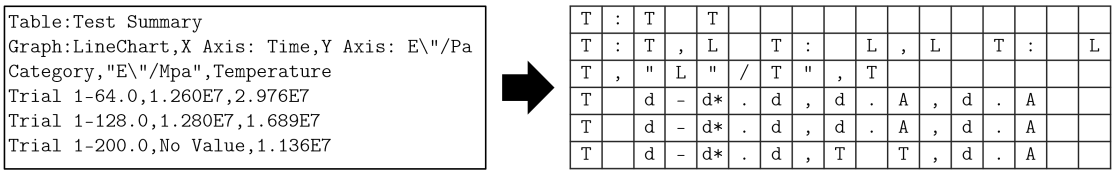

Figure 29: Sample pattern tokenization for a raw file - the second line gets truncated due to token overflow.

encoding the structure of every token, and that of every row in vectors of dimension D (set to 768 in our experiments, following the BERT architecture). To train the transformer layers and focus their attention on structural rather than semantic features, we designed two novel pre-training tasks (described in detail in Section 4.2.2) that operate on file rows. The pre-training is carried out on millions of rows from real-world CSV, from scratch.

The final component of MAGRITTE aims at providing a single embedding, of dimension K (after experimentation, set to 128), to capture file-wise structural features. This embedding is obtained using a convolutional autoencoder architecture, trained to reconstruct the row-wise feature maps produced by the structural transformer. The intuition behind the use of convolutional layers is their capability to capture spatial features and local structures. The remainder of this section explains in further detail each of these components.

4.2.1 Pattern Tokenization

The first component aims at abstracting away the semantic information about the payload of a file and forces the model to focus on its structural properties. This is done by tokenizing the raw character stream of a file into what we call *structural patterns*. Figure 29 presents an example of such tokenization. First, the character stream is split into rows according to newline characters. We note that, generally, such rows may not always correspond to whole records of a file's table, due to possible enclosing quotation marks that signify the presence of a newline within a cell value.

For every row, we tokenize it according to all the special characters that it contains. Then, we abstract everything in between two special characters with a *pattern*. We define a pattern to encode either a single character or a sequence of alphanumeric characters:

- A single lowercase letter is represented as `"l"`, and contiguous lowercase letters are represented as `"l*"`.
- A single uppercase letter is represented as `"L"`, and contiguous uppercase letters are represented as `"L*"`.
- A single digit is represented as `"d"`, and contiguous digits are represented as `"d*"`.
- A single pictogram, ideogram, or other non-syntactic symbols (e.g., emoji or logogram) is represented as `"S"`. Contiguous symbols are represented as `"S*"`.
- Contiguous strings of lowercase, uppercase, and symbolic letters are represented as `"T"` (for text).
- Contiguous strings of numbers and text are encoded as `"A"` (for alphanumeric).

Even though we don't explicitly assume the tokenization of file rows to be consistent within the same file, we deem it a highly probable occurrence. Related data preparation research demonstrated the solidity of this assumption, by leveraging row consistency to detect or repair erroneous rows [40, 84]. For example, tabular columns containing a number with periods separated with a dot as a decimal delimiter would all be represented with a `"d*.d*"` pattern, irrespective of their value. Due to the different lengths of cell values, however, the resulting tokenization of rows may not always align across different records. As described in the next section, MAGRiTTE compensates for this effect with the attention mechanism of the row embedding transformer. Overall, the dataset we use for pre-training [50] contains 409 unique pattern tokens in its vocabulary.

4.2.2 Structural Transformer

Once the content of a file has been tokenized with structural patterns, the goal of MAGRiTTE is to encode structural features into token- and row-level embeddings. The intuition of MAGRiTTE is the use of a transformer neural network architecture, leveraging the attention mechanism to learn which tokens carry more structural information. Transformer models were originally developed for language translation tasks [112], with an encoder-decoder architecture that leverages the concept of *attention*. With attention, a model is trained to recognize the context of a word by looking at all other words in an input sentence [27, 112]. In the natural language domain, the attention mechanism provides the ability to learn long-range dependencies between words in a sentence. Encodings obtained with large language models have proved very successful in a wide variety of NLP tasks [27, 91], have shown to encode numbers with a certain degree of numeracy [119], and have also been used for learning tabular representations and performing data-cleaning tasks [26, 52, 124].

Representing the structure of a file can be thought of as a special language modeling task since the composition of data files also follows grammatical, syntactic, and semantic rules. The second component of MAGRiTTE is composed of a sequence of transformer encoder layers whose (pre-)training stage is specialized for the structural tokenization of file rows. Adapting the tasks used in [27], these structural transformers are pre-trained on *pairs* of file rows. The two input rows are tokenized using our pattern tokenization and concatenated with two special tokens: one prepended at the beginning of the sequence, the classification token `[CLS]`, and one between the first- and the second-row tokens, the separation token `[SEP]`. Once pre-trained, the `[CLS]` token vectors will embed a row-level representation. The separation token is used to separate the two rows of a combined input.

Transformer layers require input sequences of a fixed size M. As reported in Section 4.4.1, we experimentally set this dimension to 128. Since pairs of file rows may have a different number of input tokens, we reserve $(M/2) - 1$ tokens for each row: rows with more tokens are truncated, while shorter ones are padded with the padding token `[PAD]`. The positions of padding tokens are fed as input to the encoding layers of the model to exclude them from attention calculation. Using these input sequences, we pre-train the transformer layers of MAGRiTTE on two novel training tasks: *Structural Masking Modeling* and *Same File Prediction*.

Structural Masking Modeling The first objective is inspired by the *Masked Language Modeling* task, which is defined for natural language sentences. At training time, we randomly substitute 15% (the standard rate in literature) of each file row's special character tokens with a different token and force the model to predict the original token. Following state-of-the-art practice, we mask special tokens with the ad-hoc masking token `[MASK]` with an 80% probability; with a 10% probability, they are replaced with another random token from the training dataset, and with the remaining 10% probability they are not replaced at all. Following [27], this setup leads the model to better generalization rather than simply masking all tokens with the mask token.

Our masked modeling is restricted to special character tokens: we want to push the encoder to encode structural information rather than file cell content. We believe this training is robust and general regarding file structure due to the tokenization strategy. Splitting on special characters regardless of their role within a file means that, in solving the structural masking task, MAGRITTE has to learn the difference between special characters that belong within a cell (e.g., a comma delimiting the digits of a number) and those with a structural role (e.g., a comma as cell delimiter). For this, the attention mechanism of transformer encoder layers plays a vital role: the context of a token may help MAGRITTE to recognize, for example, that a comma may occur within two quotation characters. Moreover, our training set includes a wide variety of files with different dialects, discouraging MAGRITTE from overfitting on a given dialect.

Same File Prediction In this task, the model is trained to classify whether the two rows belong to the same file or not. To do so, the model uses a logistic regression classifier, which takes as input the encoded representation of the `[CLS]` token of the input sequence. Rows from different files have different structural properties like the number of cells, dialect characters, or different data types. Therefore, by learning to solve this task, MAGRITTE is trained to produce row-level encodings that take into account the context and occurrence of special tokens. Although the same file prediction task is technically a supervised learning task, the creation of a suitable dataset of file rows does not require any manual labeling, since labeled pairs can be obtained trivially by sampling from the same or different files.

Following the BERT architectural design [27], the structural transformer of MAGRITTE is composed of 6 transformer encoder layers with 12 attention heads each.

4.2.3 Convolutional Feature Extractor

Although row-level embeddings may capture fine-grained structure, there are structural features that pertain to the file level and not to individual rows: for example, the presence of multiple header rows or multiple tables [116]. Moreover, it is useful to condense multiple row embeddings into a more synthetic representation of the whole file. The third component of MAGRITTE, the convolutional feature extractor, aims at encoding file structure.

To capture these features, locality plays an important role: portions of the file that are close together, such as neighboring cells or subsequent rows, are more likely to possess a similar structure than those that are farther away. To account for locality, MAGRITTE's file embedding component uses a convolutional neural network (CNN) architecture for

feature extraction. The particular characteristic of convolutional layers is the sensitivity to the spatial distribution of values – a feature that made them particularly successful for image recognition tasks, but that has also been applied to structured data to perform tasks such as table recognition on CSV files [41].

The file-embedding component is trained as an autoencoder model [59] inspired by DCGANs [86], composed of a convolutional-based encoder and an inverse convolutional decoder. The input to the encoder layers is the feature map obtained by stacking the row embeddings of the previous stage for all file rows. To fix the input dimensions for the training of the convolutional layers, we set the number of input file rows to 128. For files with fewer rows than required for the input, we include padding rows that contain only the `[CLS]` and `[SEP]` tokens (note that `[SEP]` is used as the end-of-sequence token during the transformer pre-training stage). For files with more rows than required, we use truncation.

The MaGRiTTE file encoder uses a ResNet-18 [44] architecture adapted with an input convolutional layer reading feature maps with the depth of the row embeddings (768) and outputs a single-dimension encoding of size 128. The MaGRiTTE file decoder, used for training the encoding stages, mirrors the encoder: it uses a reverse ResNet-18 architecture to reconstruct the original feature maps. The file encoder stages are trained using the reconstruction loss between the input feature map and the output feature map, calculated as the Mean Squared Error (MSE), but without taking into account the padding values for uneven rows/files.

The embeddings generated by MaGRiTTE can be leveraged to perform individual data preparation tasks. In the next subsection, we report on the adaptation of MaGRiTTE to address data loading, table understanding, and preparation effort estimation.

4.3 Data preparation with MaGRiTTE

To leverage the payload of tabular data files in a data-driven pipeline, several preparation steps are often necessary. The first prerequisite to loading tabular files is knowledge of their dialect. A file's dialect is composed of the characters used to delimit cells, quote their values, and escape quotation characters within cells. Although standards exist for file dialects [94], many real-world files are formatted with different dialects, e.g., because of international localizations, or data values containing dialect characters. When metadata about the dialect is not available, dialect detection is required to parse the file. However, dialect detection is often not sufficient to fully leverage the payload of a tabular file.

In fact, the representation of a table within a file does not always follow a row-oriented structure where the first row represents the table header and each following row represents a data record (see Figure 30). Apart from occasionally empty rows/columns, rows may serve different purposes, e.g., footnotes, preambles, or group headers [56, 63], column names may be missing or non-descriptive [101, 127], or files may have several tables [21, 116]. With automated preparation far from being a reality [66], each of these potential problems requires user efforts to address, which notably make up most of the practitioners' development time.

In the remainder of this section, we describe three fine-tuning strategies that extend the MAGRITTE architecture with classification and regression heads to address several data preparation tasks: dialect detection (in Section 4.3.1), row classification and column type annotation (in Section 4.3.2), and data preparation user effort estimation (in Section 4.3.3).

4.3.1 Dialect Detection

Dialect detection is often required to parse tabular files that do not follow the CSV standard, which is not a rare occurrence [50, 75, 113]. Typically, heuristic or frequency-based algorithms are applied [12, 28, 49]. The task is still challenging and far from solved, as demonstrated by the benchmark results of Chapter 2 [113], because of several reasons:

1. Algorithms fail to detect uncommon dialects because they are not designed to recognize such cases, e.g., have a restricted vocabulary;
2. Files may have inconsistent rows with different dialects, e.g., due to metadata or multiple tables;
3. Files may have broken dialects, e.g., with missing escape characters, that leads to incorrect detections.

To overcome these shortcomings of traditional approaches, we formulate dialect detection as a *syntactic tagging* task: given character spans of file rows, the goal is to classify whether each token corresponds to a cell value (`"C"`), to a cell delimiter (`"D"`), to a quotation mark (`"Q"`), or an escape character (`"E"`). Our problem formulation aims to overcome the aforementioned limitations because classifying individual file characters according to their dialect role does not restrict the output to a fixed vocabulary and is more robust to row-level inconsistencies or broken dialects.

To use MAGRITTE for the dialect detection task, we fine-tune the general architecture using a classification head that takes as input the concatenation of each token-level embedding with the corresponding row embedding (represented by the `[CLS]` token), and with file embedding resulting from the convolutional encoder.

The classification is a logistic regression that outputs the logit probabilities for each of the input file tokens to be a cell, delimiter, quotation, or escape character. As training loss, we use the cross-entropy losses calculated on the whole sequence of file tokens. The final unique dialect characters are chosen as the ones corresponding to the tokens most frequently tagged as delimiter, quotation, and escape character. If, within a file, no token is classified as being a delimiter (or quote, or escape) we consider the file as having an empty delimiter ε (or quote, or escape).

4.3.2 Table Understanding

Much research has been devoted to the general area of *table understanding* [26, 125], to automatically extract the metadata necessary to leverage the data contained in a file. Some examples of tasks in this area are column type annotation [100, 127], entity linking [67], or schema augmentation with related data [30]. These tasks are even more challenging for real-world files, such as the one shown in Figure 30, since all proposed

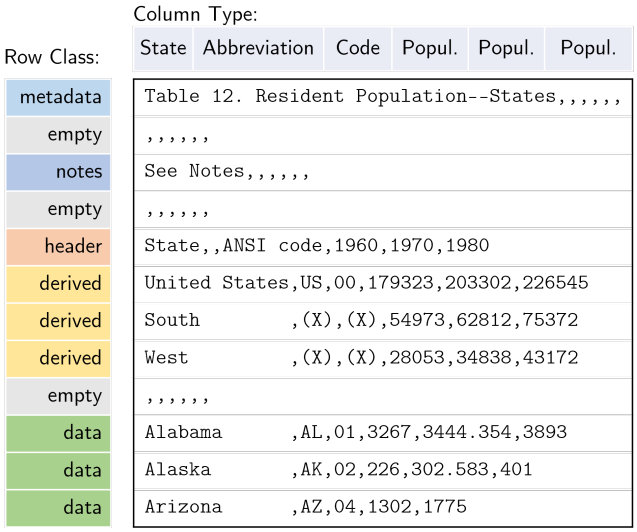

Figure 30: Table understanding results on a real-world file from the SAUS dataset [36] (excerpt, and visually aligned for clarity). A clean version of this file would only include the **header** and **data** rows, with descriptive column names.

solutions operate on relational tables, which first need to be extracted from unprepared files [21, 56, 116].

In this section, we apply MaGRiTTE to perform the necessary preparation for table understanding. We use MaGRiTTE to classify file rows (e.g., as header, metadata, or data), and use these results to extract tables on which we perform column type annotation. Since the representations learned by MaGRiTTE are focused on the structural features of files, while other approaches only take cell content into account, we propose solutions to combine state-of-the-art approaches [56, 101] with MaGRiTTE, in a hybrid solution. Our experiments in Section 4.4.3 show that using the structural embeddings of MaGRiTTE to prepare these files increases the performances of existing approaches.

Row classification The goal of the task of row classification is to identify the header and the data records to obtain a relational table, while recognizing and extracting the useful metadata contained in other rows. This problem has been formulated in slightly different variations, for example as a binary classification of data/non-data rows in [21], or as a graph partitioning problem in [63], depending on the assumptions on the input files. Since the focus of MaGRiTTE is on tabular files, and to perform an adequate comparison, we consider the same conceptual model and problem formulation used in the state-of-the-art approach Strudel [56], a random forest classifier.

In this formulation, row classification is considered a multi-class classification problem where a row of a file can belong to one of the following classes, mutually exclusive: **header**, **data**, **group**, **derived**, **metadata**, and **note**. The **header** rows contain the column names of a table; **data** rows represent the records of a table; **group** rows organize the table into sub-tables (groups) and represent the header for a given group; **derived** rows contain data that is the result of some operation on **data** rows, e.g., a total, average, or aggregation;

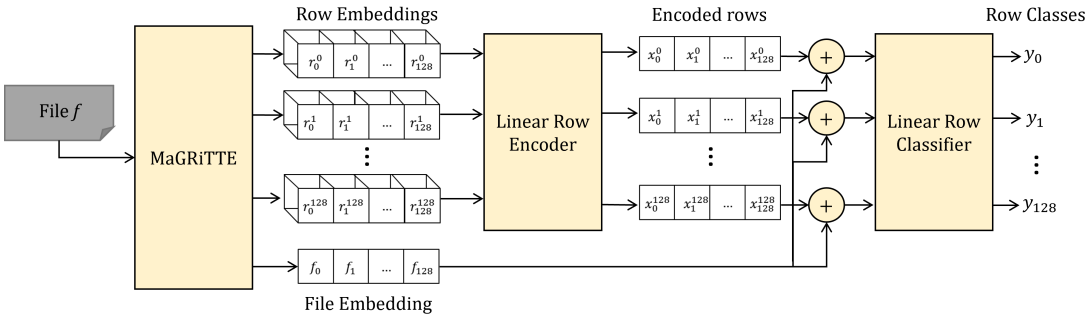

Figure 31: Fine-tuning architecture of MAGRITTE for the row classification task.

metadata and **note** rows contain metadata information respectively before and after a table. For a more detailed description, refer to the original paper [56].

To utilize MAGRITTE for the row classification task, we fine-tune it with a shallow classification head consisting of two logistic layers, illustrated in Figure 31. The first layer takes, for every row, all the 128 token-level embeddings, which all have 768 dimensions, and condenses them into a single dimension. The second layer combines the output of the first layer with the file-wise embedding computed by the convolutional encoder and outputs the class probabilities for each row.

We also experimented with the combination of MAGRITTE and the SOTA model, STRUDEL. In this scenario, we feed the class probabilities calculated by MAGRITTE and consider them as additional features of STRUDEL, both during the training and inference stages. Section 4.4.3 presents and analyzes the experimental results.

Column type annotation Column type annotation is defined in [26] as the task of annotating a column c of a relational table T with a semantic type $l \in L$ such that all values in c belong to l. Typically, the semantic types associated with columns are highly descriptive, for example, "Professional Athlete" for a column of person names, or "Publication Date" for a column of dates. Successfully addressing the task for clean, standard relational tables is already challenging and the subject of wide research interest [26, 51, 100, 101, 127], given its importance for tasks such as data integration and discovery. In this paper, we focus on solving the task for real-world tabular files that do not have a standard relational tabular structure (see Figure 30).

Considering the combined structural and semantic nature of the task, we propose an end-to-end framework that combines MAGRITTE, STRUDEL, and RECA [101], the state-of-the-art system for column type detection. RECA is a framework that annotates the columns of a table by combining a machine-learning architecture based on the pretrained BERT model [27], with contextual information from different tables with semantically similar content. Therefore, it assumes a dataset of relational tables and cannot directly be applied to raw tabular files. We bridge this gap using the structural embeddings produced by MAGRITTE.

First, for all files in the dataset, we obtain structural embeddings with the specialized MAGRITTE model for row classification described in the previous paragraph. Then, we

combine these embeddings with the features computed by STRUDEL and run our HYBRID row classification model. Using the row classes computed by our HYBRID system, we define a simple set of heuristics that can be easily automated to extract relational tables out of a dataset of raw files:

1. We delete empty rows and those classified as **metadata**, **note**, and **derived**.
2. In the case of multiple detected **header** rows in the file, we merge them together into a single row, each value being separated by a whitespace character.
3. If there are multiple tables (defined as multiple stretches of **header** followed by **data** rows), we extract the first table.
4. If there is no **data** row, we exclude the file from our dataset.

Finally, we train and validate RECA on unprepared, automatically cleaned, and manually cleaned versions of the dataset and compare the results. Our experiments in Section 4.4.3 show the combined value of our framework for the task of column type annotation.

4.3.3 Preparation Effort Estimation

In this section, we introduce a novel data management task, which we call *preparation effort estimation*. We believe that accurately estimating how much *user effort* is required to prepare a given file is useful for several aspects: for time management in data-driven projects, for estimating the cost of using a given data source, e.g., in a data market scenario, or to help choose the most suitable data, trading off cost and benefit.

Given an input file f, we assume that, for a given task, there is a unique target f'. We consider the file f' to be unknown before data preparation, but to be approximated with a template file t. The data scientist's goal is to match the content of f to the structure of t. We define a template file to be a file with the same structure of f' and generally different payload, for example, another prepared file from the same project, or one with dummy data easily generated as a specification file by the end-user.

Given the set of all possible preparations $\mathcal{P}$, a target file f' can be considered $P_t(f)$, the preparation of f to the structure of t. Estimating the preparation effort for a given file f and template t corresponds to finding a distance function δ that measures the structural difference between two files regardless of their content, such that $\delta(f,t) = \delta(f,f')$ but $\delta(t,f') = 0$. Based on these assumptions, we define the model-based preparation estimation problem as:

Given a set of tabular files $\mathcal{F}$, a set of preparations $\mathcal{P}$, and a set of target files $\mathcal{T}$, find a structural distance function δ such that:

$$\delta(f,t) = \delta(f,f') \ \wedge \ \delta(t,f') = 0 \quad \forall f \in \mathcal{F}, \ \forall P_t \in \mathcal{P} \times \mathcal{T}$$

While extensive research has been devoted to the estimation of software development costs (SDCE) [7, 8, 57, 58], fundamental differences make these methods unfit for estimating data preparation efforts. For example, frameworks based on mathematical models like COCOMO II [8] or ESTIMACS [93] define a set of equations that take into account several factors, such as the number of lines of code, the number of functionalities offered by the software (function points), or the size of the teams involved in the development. To estimate preparation effort, such metrics are not relevant: because most

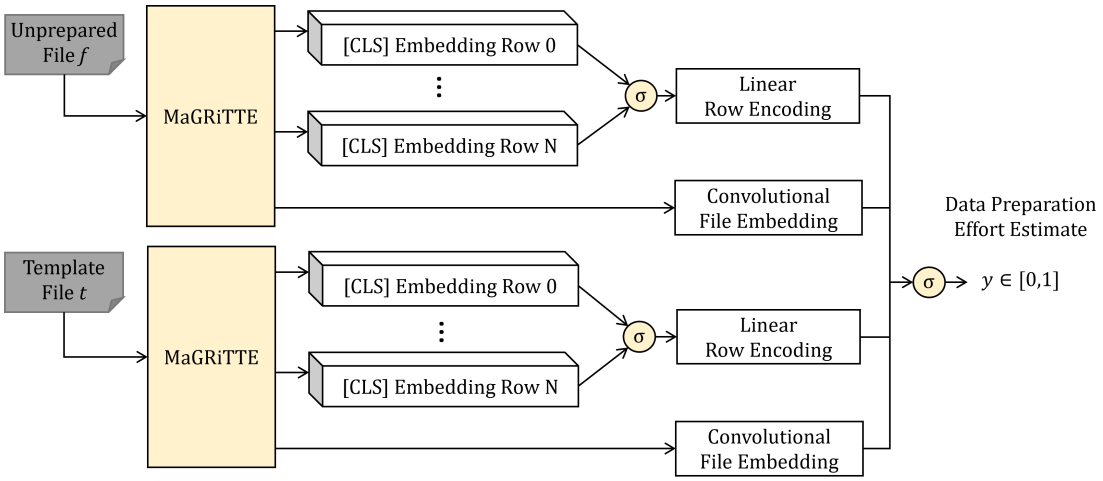

Figure 32: Fine-tuning architecture of MAGRiTTE for the data preparation effort estimation task.

of the user efforts are spent on data exploration, wrangling, or normalization, they are invisible to metrics purely based on the length of programming language code. Moreover, traditional SDCE models make use of these features in formulas with parameters tuned with data from historical projects that were available at the time of the development of these models.

Presently, we are not aware of any model specifically proposed to estimate these efforts quantitatively. We also caution on the application of expert-based techniques, such as the Delphi or Work Breakdown Structure techniques [25, 105], for estimating preparation efforts: first, because new projects/datasets carry a unique set of potentially unseen problems [45]; second, because of the relative infancy of the field, reliable expertise may be hard to come across [66].

With MAGRiTTE, we propose a learning-oriented approach that is grounded on pairs of unprepared/template files and leverages the file embeddings calculated by the convolutional stage of MAGRiTTE and a shallow head composed of two regression layers. We illustrate the regression head visually in Figure 32.

Given the two files, we compute their row-level and file-level embeddings. For each file, we sample the embedding of the `[CLS]` token for each row and use the first fully connected layer to extract a feature vector of dimension 128 (corresponding to a single value for each of the maximum file rows). Then, we concatenate the sampled row embedding and the convolutional file embedding for each of the two input files into a single vector of dimension 512. The second and final layer is a fully connected regression node taking the row encodings computed by the previous nodes along with the convolutional file embeddings of the two files, and computing a single number that estimates the structural difference between the two files. To normalize the estimate to the range $[0, 1]$, we feed it through a sigmoid layer. We train MAGRiTTE with this regression head using the mean squared error between the computed estimate and the ground truth.

To quantify user effort, we use as a proxy the length of the preparation scripts defined to clean dirty files, acknowledging that this is a simple, coarse-grained measurement. Other measures, such as computational or I/O time, or number of performed instructions, would not necessarily reflect the user effort in specifying the operations. For example, filtering all values in a column of a very large file with a specified regular expression may take only a short time to specify, but a rather long processing time. Even if limited in scope, we believe that our contributions in this paper may lay the foundation for an unexplored area of data engineering research.

4.4 Experimental results

In this section, we present the results of our experimental analysis of Magritte. First, we report on the pre-training procedure and the hyperparameter selection. Then, we present experiments to assess the application of Magritte for several real-world data preparation tasks: dialect detection, row classification, column type annotation, and preparation user effort estimation. Each of these tasks is evaluated using real-world, publicly available datasets, and compared with relevant state-of-the-art approaches; details are reported in individual subsections. All experiments were conducted on a machine with an AMD Epyc 7720P@2.00 GHz CPU and an NVIDIA RTX A6000 GPU with 48 GB of dedicated memory.

4.4.1 Pre-training

To pre-train the overall Magritte architecture towards structural embedding, we use the raw files from the GitTables dataset [50], which consists of over 850 000 publicly available tabular files from GitHub. We note that, although these files are marked with the CSV extension, they do not necessarily conform to the RFC standard [94], and may generally have metadata lines along with tables, non-standard dialects, or inconsistencies between rows. We believe these features make the pre-training of Magritte more robust towards different types of tabular files. From each of the GitTables files, we sample 2 500 rows from each file (sampling with repetition for files having fewer rows than that). Then, we create a balanced dataset of 10 million row pairs, half of which are pairs extracted from the same file and the other half from two different randomly chosen files. The same dataset of pairs is used for the Structural Masking Modeling task and the Same File Prediction task since the transformer layers are trained on both tasks at the same time. To simplify pre-training complexity, as suggested in [27], we first pre-train the structural transformer for 15 epochs using a sequence length of 32 and complete the pre-training with 3 epochs using a sequence length of 128. In both cases, we used a batch size of 64, as it was the highest dimension fitting in memory while performing the full model training.

The model hyperparameters for the transformer layers, except for the input length and training batch size, are set according to the original BERT-base model [27], including an encoding dimension of 768. Regarding the maximum length of the input file rows, after exploratory analysis of our training data, we set the number of input tokens for every row to 128, to balance complexity as coverage, as this length covers over 90% of the 10M input rows, with the rest having a significantly higher number of tokens. The

Rank	Delimiter	Quotation	Escape	# Files
1	,	ε	ε	2323
2	,	"	ε	2056
3	;	"	ε	358
4	;	ε	ε	353
5	,	"	"	285
	...	...	...	...
28	,	`	ε	1
29	&	ε	ε	1
30	%	"	ε	1
31	#	ε	ε	1
32	|	`	ε	1

Table 13: The 5 most and least common dialects in the imbalanced dataset. The character ε denotes an empty character.

latent dimension of the convolutional autoencoder was the subject of hyperparameter tuning. We considered dimensions in the range $[32, 64, 128, 256, 512]$, and trained the encoder/decoder layers for 1000 steps using a batch size of 64. To perform this training, we froze the structural transformer layers.

After setting the encoding dimension to 128, which performed the best in our hyperparameter tuning stage, we trained the full model for 10 epochs using the entire set of 871 349 CSV files of GitTables [50], reserving 10% of the files for validation purposes. We used a batch size of 64, consistent with pre-training.

4.4.2 Dialect detection

To perform fine-tuning of MAGRITTE for dialect detection, we train with a set of 18 300 files, obtained with augmentation from an original set of 5 689 CSV files manually labeled with dialect annotations, from [12]. We checked and corrected some annotation errors, and share an updated version of the annotations on the project page of MAGRITTE[2]. These files were never seen by MAGRITTE during pre-training. For development purposes, we used a validation set of 3 660 files (amounting to 20% of the training files), composed of 3 503 augmented files and 157 original files, sampled from the same distribution as the training files. We also report the results obtained on 1 543 files, never seen by MAGRITTE during both the pre-training and the fine-tuning phases. Some of these files show properties that make them non-compliant with the regular CSV standard (apart from the dialect characters), such as having multiple tables, or preamble and comment rows that do not correspond to table headers or records. As systems struggle with non-standard CSV files, we refer to these as the *difficult set*, as opposed to the test files which are compliant with the CSV standard, which we refer to as the *test set*. The test set contains 1 087 files, and the difficult set 456 files.

Dataset augmentation The distribution of dialects in the original training set is heavily imbalanced towards the most common dialects. Table 13 shows the 5 most

[2] `https://github.com/HPI-Information-Systems/Magritte`

common and the 5 least common dialects found in the files of the training dataset, together with their frequency. Overall, in the training set, we identified 11 unique delimiters, 5 unique quotation characters, and 4 unique escape characters. The dataset has a long tail of different dialects, with 32 unique dialects, but most of the files have cells delimited by a comma and no quotation or escape characters. To compensate for this bias, we augment the files to obtain a balanced training set.

Each file of the dataset is augmented by taking its original cell values, and replacing the delimiter, quotation, and escape character with all valid combinations (183 in total) of the distinct dialect characters found in the original dataset. A valid dialect is one where the delimiter is different from the quotation character, and there is no escape character if file cells are not enclosed in quotation marks. We only augment files that do not contain any target dialect character in their content, to avoid generating invalid CSV files. Augmenting a file to have the empty delimiter ε corresponds to generating single-column files. To do so for otherwise multi-column files, we remove all columns except for the first. To augment files towards dialects with a quotation or escape character, we include at least one delimiter or escaped quotation character within random cells of the files (with a probability of 5%), and quote cell values if needed, accordingly.

Experimental setup Considering all possible and valid combinations of dialect characters in the training set, we obtained a total of 183 augmentation dialects. After augmenting all the original files with all valid and applicable augmentation dialects, we sample 100 files per dialect class, randomly chosen from the set of augmented and original files. The final set of augmented files is therefore composed of 18 300 files. We split this set into two folds of 80%/20%, as the training and validation folds, composed respectively of 14 640 and 3 660 files. Overall, the benefit of this augmentation scheme is not only that it counters class imbalance, but it also provides the model with files having the same cell contents but with different dialects. We believe this helps the generalization power of the model and prevents the model from overfitting on file content rather than on structure. We trained MAGRiTTE for 10 epochs using a batch size of 6.

Results Following the experimental approach of [12], we evaluate the results of dialect detection using precision, recall, and F1 score for each of the three dialect classes, as well as *dialect accuracy*, which is computed for each file by assigning a score of 1 if all three classes (delimiter, quotation character, and escape character) are correctly detected. The F1 scores are averaged across the different classes, weighting the average based on the number of samples for each class. All four scores range between 0 and 1, with 1 being a perfect performance.

We compare MAGRiTTE with CLEVERCSV, the state-of-the-art system for dialect detection [12]. CLEVERCSV is a rule-based algorithm to perform dialect detection that uses a *pattern score* and a *type score*. The best dialect is identified as the one that leads to files with rows having the same number of columns (measured by the pattern score) and, within a column, which shows a high homogeneity of data type (measured by the type score). The hyperparameters to tune these scores have been identified by the authors on a set of validation files and then evaluated on a separate test of files.

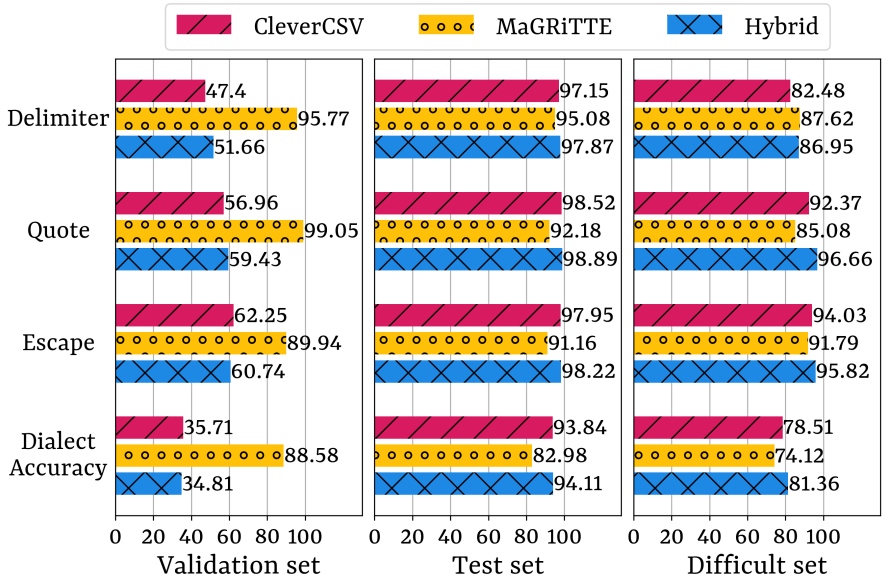

Figure 33: Results of dialect detection on validation and test sets (scaled to $[0, 100]\%$).

Figure 33 reports the results of our experimental evaluation, rescaling the scores to the range $[0, 100]$ for ease of comparison. As can be noted, MAGRITTE, having been trained on files with a balanced set of dialects, outperforms CLEVERCSV on the validation set, which is composed of files with the same balanced distribution. However, its accuracy on the test folds, while still relatively high, does not match the one of CLEVERCSV. While analyzing the errors of MAGRITTE on such files, we noted how in its classification, it tended to weigh the occurrence of unusual characters within the file, such as the pipe symbol `"|"` or the colon symbol `":"`, as being part of the dialect. Therefore, in the test and difficult sets, which contain a higher percentage of files with common dialects, MAGRITTE can be characterized as having a high precision but lower recall, i.e., once it detects a standard dialect it has high confidence, but it may have false negatives due to overestimating the probability of unusual dialects.

In contrast, the performances of CLEVERCSV are specular: as it was developed using an imbalanced set of files with more standard dialects, it tends to overestimate the probability of a file having a standard dialect, therefore, it has a lower precision on standard files than on non-standard files. This can be noted, for example, by observing how its performances differ from the validation set to the test set. To obtain an effective dialect detection system, we propose a HYBRID solution. If MAGRITTE recognizes the dialect of the file as having a comma as a delimiter, a double quote character for quotation, and no escape, i.e., the most common dialect, we consider this prediction as having high confidence. Otherwise, we resort to the results of CLEVERCSV. As seen in Figure 33, this HYBRID approach improves on the shortcomings of both systems. In this approach, MAGRITTE acts as a filtering stage, that only leverages CLEVERCSV for validation in files suspected of having a non-standard dialect.

Finally, we note how all three approaches struggle more for files belonging to the difficult set: these files are often characterized by having multiple tables, many comment rows that do not follow CSV formatting, or rows with an inconsistent number of delimiters (see Figure 34 for an example from this set). These files often confound the dialect detectors, and require more sophisticated approaches that can detect and prune their metadata rows. In the following section, we discuss how MAGRiTTE can be successfully employed to solve this task.

```
1 # LimeSurvey Group Dump
2 # DBVersion 130
3 # This is a dumped group from the LimeSurvey Script
4 # http://www.limesurvey.org/
5 # Do not change this header!
6
7 #
8 # GROUPS TABLE
9 #
10 "gid","sid","group_name","group_order","description","language"
11 "43","10","Student BIO","6","Fragebogen für Biologistudenten<br
     />","de"
12
13 #
14 # QUESTIONS TABLE
15 #
16 "qid","sid","gid","type","title","question","preg","help","other
     ","mandatory","lid","lid1","question_order","language"
17 "211","10","43","L","1","Tragen Sie Biolatschen?<br />","","","N
     ","N","0","0","1","de"
18
19 #
20 # ANSWERS TABLE
21 #
22 "qid","code","answer","default_value","sortorder","language"
23 "211","1","ja","N","1","de"
24 "211","2","nein","N","2","de"
25 "211","3","geht dich nix an!","Y","3","de"
26
27 #
28 # CONDITIONS TABLE
29 #
30
31 #
32 # LABELSETS TABLE
33 #
34
35 #
36 # LABELS TABLE
37 #
38
39 #
40 # QUESTION_ATTRIBUTES TABLE
41 #
```

Figure 34: A file from the difficult set (lines 11 and 16-17 are wrapped for display reasons).

4.4.3 Table understanding

To perform table understanding experiments, we leverage six publicly available datasets, introduced in [56] (see Table 14), that are composed of generally unprepared real-world tabular files, i.e., files do not adhere to the CSV standard because they may contain non-data rows, multiline headers, or multiple tables.

For the column type annotation task, we provide column annotations for all six datasets following the same strategy used in [101, 127]: we annotate columns with a semantic type from DBpedia [3] if their headers (disambiguated with respect to spaces) match with the name of a DBpedia ontology or property. We annotate one column for each file and discard files for which no column can be matched with a DBpedia attribute. Overall, our dataset contains a total of 865 files with 116 unique column type annotations. We distribute the files and annotations on our project page[3].

Because of the architectural limitation of MAGRITTE regarding the dimension of file inputs, and also due to the high imbalance of the datasets towards data rows, we limit our experiments to up to 128 rows per file. To not exclude metadata rows that might occur towards the end of a file (e.g., **note** or **group**), we cannot resort to simple truncation and sample the first 128 rows of each file. Therefore, for files with more than 128 rows, we use the following scheme: we prioritize non-data rows, and, if there are less than 128 non-data rows (which is the case in all but 5 of the files), we fill the rest with data rows. We note that rows are always sampled respecting the original order of the file. We pad files of fewer than 128 rows with rows containing only the `[CLS]` and `[SEP]` tokens, consistently with the padding strategy during the convolutional pre-training stage.

Row classification For the row classification task, we train our models on the four datasets GOVUK, SAUS, CIUS, and DEEX that contain a total of 1 162 files and 221 218 rows. Following the original experiments in [56], we train using 10 cross-validation folds on these four datasets. We also include two separate datasets, MENDELEY and TROY, for out-of-domain testing, containing 262 files and 199 946 rows. When testing on these two datasets, we used all files in the previous four datasets for training. We note that all files from the training or test sets have not been seen by MAGRITTE during pre-training. We train MAGRITTE for 40 epochs with a batch size of 8.

Figure 35 reports the results of our experimental evaluation, rescaling the scores into the range $[0, 100]$ for ease of comparison. The results highlight the competitive performances of MAGRITTE especially when classifying the four row classes **note**, **metadata**, and **group**. Rows from these classes are often recognizable from their structure alone: **group** and **note** typically only have content in the first cells and therefore are characterized by long sequences of delimiter characters. Furthermore, together with **metadata** rows, they are more likely to contain letter characters and fewer digits or numeric symbols. These structural features can be picked up by MAGRITTE thanks to its pattern tokenization strategy. Rows belonging to these four classes are often outliers compared with the rest of the file rows, as typically **data** and **derived** rows have a very regular structure.

From our analysis of the results, the low scores of MAGRITTE for **data** and **derived** rows are due to the model frequently misclassifying one class as the other. This behavior

[3] `https://github.com/HPI-Information-Systems/Magritte`

	GovUk	Saus	Cius	DeEx	Mendeley	Troy
# files (row class)	226	223	269	444	62	200
# files (column type)	110	153	164	260	12	66
# rows	97 212	11 598	34 556	77 852	195 598	4 348
# cols	3482	3955	3656	6390	797	2282
header	519	576	435	1 222	86	280
data	93 584	9 469	31 845	74 245	194 786	2 898
group	850	283	119	302	27	42
derived	665	280	449	664	9	239
metadata	878	472	1 034	713	604	315
note	716	667	674	706	86	575
# column types	32	26	21	66	11	19

Table 14: Table understanding dataset overview, with instances of row classes and column types. The annotated datasets for column type annotations are a subset of those with annotated row classes.

Dataset	Weighted-mean F1	Macro-mean F1
unprepared	77.96%	66.55%
autocleaned with Magritte	80.50 %	67.51 %
standard	82.29 %	71.06%

Table 15: Column type performances of RECA [101] with unprepared, automatically prepared, and standard files.

is to be expected, considering that often the difference between **data** and **derived** rows lies within the content of their cells, a semantic detail that is abstracted away from Magritte in favor of a structural perspective. While on the one hand, Magritte often outperforms Strudel in the detection of classes such as **group** and **metadata**, on the other hand, Strudel always outperforms Magritte on **data**, **header**, and **derived** rows thanks to its more semantic-oriented features.

As for the previous task, we propose a Hybrid approach that can leverage the strengths of both models: we first run Magritte to detect the class probabilities for the rows of a file, and then use these probabilities, one for each class, as extra features in Strudel. This approach outperforms both Strudel and Magritte with very good success for every class in all the cross-validation datasets. However, as can be noted from the performances on the out-of-domain datasets, Mendeley and Troy, the Hybrid approach does not lead to an improvement when any of the two base models have poor performances, or when the datasets have a very skewed distribution of **data** vs. non-**data** rows (see Table 14). This behavior is particularly evident, for example, for the **group** and **note** classes for Mendeley.

Column Type Annotation To assess the impact of Magritte on downstream tasks on the six datasets of real-world files of Table 14, we experimented with the column type annotation task and used the SOTA system RECA [101]. Since this system is reported

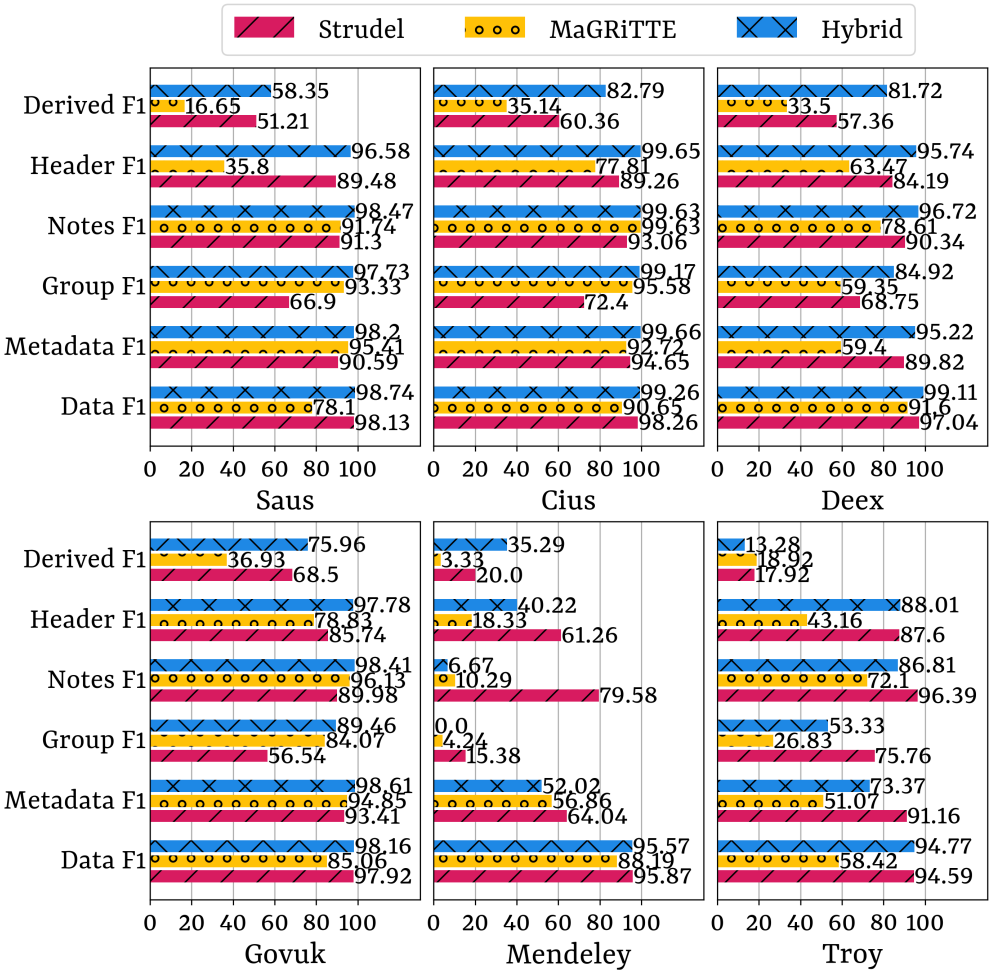

Figure 35: Results of row classification on the experimental datasets (normalized in 0-100%).

to outperform all previously proposed models [51, 100, 127], which are also based on column-level embeddings, we expect our findings to hold for the whole family of models.

We experiment with three versions of the dataset: an **unprepared** version, corresponding to the raw file input; an **autocleaned** version, corresponding to automatically cleaning of the files based on the row classes obtained with MAGRITTE as described in Section 4.3.2; and a **standard** version, corresponding to a CSV-standard version of the files, cleaned with the same procedure but with the ground truth row classes. Following the original implementation and experimental setup publicly available in the code repository of RECA, we split our datasets into a train/validation fold and a test fold, respectively composed of 90% and 10% of the original files. We trained the RECA model on the train/validation folds using 10-fold cross-validation for 20 epochs and tested on the test fold, repeating the test 3 times.

Table 15 reports the weighted- and macro- mean F1 score for all column type classes averaged across the three runs (standard deviations were zero). As it can be noted,

OpenRefine Preparation steps	
Remove a row	Mass transformation of rows
Fill-down values in subsequent rows	Deleting cells in subsequent rows
Split row into separate records	Merge subsequent rows into one
Column rename	Column shift to different position
Column removal	Add a column
Column split on character	Merge two columns with character
Trim whitespace	Add leading and trailing spaces
Detect numeric strings as numbers	Add quotes to numeric columns
Transform strings to upper/lowercase	Make strings titlecase

Table 16: A subset of preparation steps we consider, sampled from those offered by the OpenRefine framework.

using **unprepared** versions of the input files leads to the lowest performance. This is not surprising, considering that to annotate the column type for tables, RECA encodes all the values that belong to a table column. If columns are parsed out of the unprepared file, they may contain (1) unrelated values (see "`See Notes`" appearing in the "State" column of Figure 30), (2) values with heterogeneous types, e.g., if multiple tables or **derived** rows are contained in the file (see "`(X)`" in the "ANSI code" column of Figure 30), or (3) be spuriously filled with empty (see all the empty values in the **metadata**, **empty**, or **note** rows of Figure 30). On the contrary, automatically preprocessing the files with the row types detected with the help of MAGRITTE in the HYBRID scenario, improves the weighted F1 to 80.50%. To put in perspective the contribution of preparing the files with MAGRITTE, performing column type annotation on the **standard** files, prepared using manually annotated ground truth, leads to a further improved F1 of 82.29%.

To summarize, our experiments proved how, thanks to MAGRITTE, it is possible to improve downstream table understanding tasks with automated preparation and minimal user effort. The next and final experiments aim at assessing how well can MAGRITTE provide an estimate for how much user effort is needed to manually prepare files.

4.4.4 Preparation Effort Estimation

One of the greatest challenges in developing a model that estimates preparation effort is the lack of a standard measure to design, compare, and evaluate the structural distance. To address this issue, we propose a reference measure: given a set of possible preparation operations, determine the minimum number of preparation steps necessary to transform a file f into f'. As a concrete implementation of this measure, we use a subset of the preparation operators of OpenRefine, a widely used data wrangling tool [37]. This subset can be seen in Table 16. Assuming that for a pair of files f, f' there is a minimal sequence of preparation steps with a minimum length[4], the problem we want to address is to estimate the length of this sequence, without knowing the exact steps and the resulting f', but with the knowledge of a template file t.

[4]Or multiple equivalent sequences of the same length.

Dataset To train and test our model, we used a random sample of 100 pairs of real-world source-target CSV files with corresponding transformation scripts using OpenRefine [37]. We obtained these files from a sample of 180 public GitHub repositories, crawled as the ones containing at least one preparation script. However, not all repositories contain the files associated with a given script, and even for many that do, it may be undocumented which of the files corresponds to the source, and which to the target. First, we excluded repositories that did not contain at least a CSV file, leaving 83 remaining repositories. For repositories containing only the source file, we ran the scripts and generated the target version. For repositories containing only the target file, we retrieved the source file, if it was documented to be from a publicly available dataset.

We sampled 10 scripts for each length in the range [1,10] to create a balanced dataset, manually checking the script to retrieve the corresponding source and target files. We split the dataset in half, using 50 files for training and validation, during the hyperparameter optimization, and 50 files for testing purposes. To evaluate our approach, we normalize the script lengths to the interval $[0, 1]$, and compare the estimates with the ground truth. To create a large training set, we use the following consideration: if two different unprepared files f_i, f_j go through a preparation script towards their respective targets f'_i, f'_j, we can consider both of the target files as the structural "template" for each other's unprepared version, i.e., $f'_i = t_j$ for f_j. In this way, we obtain pairs of unprepared files and their templates to train the MAGRiTTE regression. As a ground truth to evaluate the source-template distance between f_i and t_j, we use the length of the scripts to transform f_i to f'_i. Considering N unprepared files f_i and their prepared versions f'_i, we obtain a total of $N \cdot (N-1)$ source-template pairs for training (in our case, 2 450 pairs). We use the remaining N pairs, 50 in our dataset, as source-target pairs for validation. We trained MAGRiTTE for 10 epochs using a batch size of 6. For testing purposes, we use a separate set of 50 pairs of source-target files from our OpenRefine dataset. None of these file pairs has been seen by the model during the training or validation stages.

Results The scatter plot of Figure 36 illustrates the results of estimating the length of the script (normalized to the range $[0, 1]$) to transform a source file into a target file. As can be noted, the performances of MAGRiTTE do follow an approximately linear trend, with the estimated efforts being generally higher for source-target pairs whose preparation effort is closer to 1.

However, the effort for files that do not require significant preparation appears to be overestimated. To understand the reason, we analyzed some file pairs for which the preparation effort was overestimated. In three files with a true preparation effort of 0.1, whose single preparation operation was to remove empty rows, MAGRiTTE assigned an estimated preparation effort of 0.44, 0.62, and 0.69, which correspond to the three topmost dots in the plot at the 0.1 mark on the x-axis.

Interestingly, we identified a correlation between the number of empty rows in a file and the preparation effort estimated: the file with the most empty rows, 103, was estimated at 0.69, followed by the one with 65 empty rows estimated at 0.62, and lastly, the file with a single empty row was estimated at 0.42. For two other file pairs, which have the same

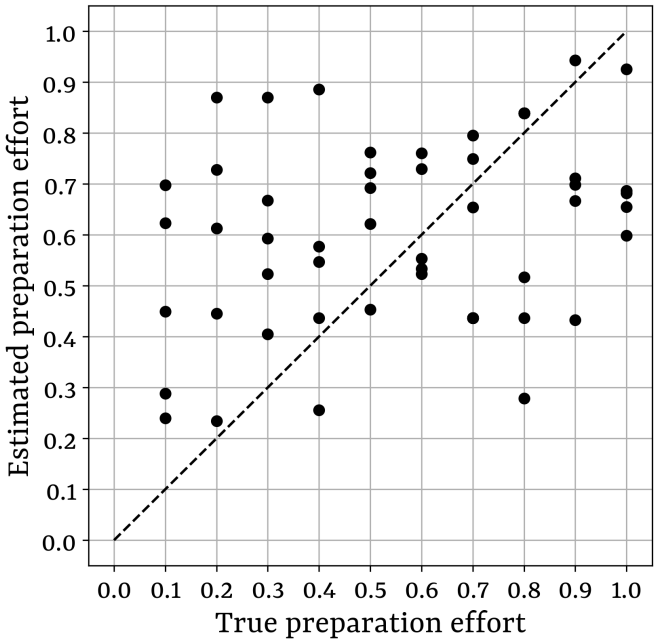

Figure 36: The preparation effort estimated by MAGRiTTE on the test files.

source file but different targets, with a target effort of 0.2 and 0.3, MAGRiTTE estimated the preparation effort to be 0.87 in both cases. In this case, the two scripts change the formatting of strings in every row from lowercase to titlecase and numeric, with the only difference of an extra preparation step that brings some values to uppercase. While the two target files are indeed very similar, justifying a similar or the same score, they differ from the source for the values in every row. Seemingly, even in this case, MAGRiTTE learned to assign high preparation effort for file pairs that differ for many rows.

We acknowledge that the results of our data preparation effort estimation experiments are very much preliminary, and there is room for further experimentation with larger datasets or more sophisticated measures for preparation effort estimation. However, we believe there is promising value to be exploited by applying regression-based methods to the novel task of preparation estimation.

4.5 Related work

The goal of our work is to provide a general representation of the structure of data files. To do so, we employ a neural network architecture that learns such a representation in a self-supervised fashion. Therefore, our work lies at the intersection of data management and representation learning, a research space that has seen a recent surge of interest [5]. To position our research, we briefly discuss related works in the areas of structural preparation and representation learning for data management purposes.

Structural preparation Many of the approaches proposed in the literature to solve file preparation steps automatically or semi-automatically rely on the structural features of the file. For example, a common strategy for the algorithms to detect the dialect of

CSV files, among which are [12, 28, 34, 99], is to generate a set of hypotheses based on the character set of the file, and then rank them based on measures like the consistency of row length, or the presence of empty cells in the parsing results. The main limitation of these approaches, compared to MAGRITTE, is that they generally look for dialect characters that apply to entire files. As found in our experiments (see Figure 30) some files, due to metadata lines or different tables within the same files, do not have consistent dialects across all the lines in their content. Our experimental evaluation highlights how the approach of MAGRITTE, based on local (character-level) and not global classification, proves especially effective in dealing with such difficult cases.

Other approaches for metadata extraction, for example [21, 56, 127], use manually engineered features such as row length, percentage of digit characters, etc., to train machine learning models on tasks such as table extraction, row and cell classification, or column type inference. The main limitation of this feature engineering approach is the lack of generality, as it requires manually defining and possibly tuning the set of features during development with a given corpus of labeled data at hand. The approach we propose in this paper, instead, aims at learning a general representation of the structure of a file, which is task-independent and trained in a self-supervised fashion, therefore leveraging large sets of files. The benefits of this approach can be seen from the experimental results of Section 4.4, where the use of the MAGRITTE structural embeddings proves competitive with various state-of-the-art models.

Transformer models for data management Recently, several works leveraged representation learning using transformer architectures or pre-trained language models on tabular data. Some of the tasks that have been successfully addressed include column type annotation [100, 101, 120], table population [26, 52, 124], error detection and data cleaning [73, 104, 118], entity matching [67, 82, 104], data discovery [30], and table question answering [123]. An extensive review of the unique features of these models is beyond the scope of this work, and we refer readers to a recent survey [5]. One common aspect of all the aforementioned approaches, that differentiates them from MAGRITTE, is that they leverage representations learned from the payload of files, i.e., the cell content of their tables. Therefore, they do not apply to *messy* files where the payload table(s) cannot be directly parsed.

The representations learned by MAGRITTE are orthogonal to the ones learned by the aforementioned tabular models. As demonstrated by the experiments in Section 4.4.3, structural embeddings can be used *in conjunction with* semantic embeddings, to enable given downstream tasks, such as row classification column type annotation, on large unprepared datasets and improve their performances.

4.6 Conclusions

In this chapter, we presented MAGRITTE, a novel neural network architecture to represent the structure of tabular files with cell-level, row-level, and file-level embeddings. We demonstrated how the embeddings generated by MAGRITTE can be used to solve a variety of preparation tasks, not only as the sole features to solve them automatically, but more importantly in conjunction with other approaches, potentially more geared

towards semantic features. Thanks to their exposure to large amounts of data, and their synthetic coverage of a wide number of features, we believe that structural embeddings may be leveraged in even more steps of a data preparation pipeline.

In a preliminary study, we also show how structural embeddings can be used to estimate the user effort required to prepare a file. We hope that future research picks up on our efforts to develop more refined models to estimate the preparation effort for more and more types of file or preparation operations, possibly extending or leveraging our publicly available, manually labeled dataset.

We envision that foundational models, such as MAGRITTE, can and will be used to assist users at all stages of the preparation pipeline, either to automate cumbersome operations or to empower software engineering and decision-making.

Chapter 5

Summary and Outlook

Plain text tabular formats are amongst the most popular way to create, store, distribute, and consume data files. This thesis introduced the notion of file *structure*, the set of characters within a file that do not carry information per se, but are necessary to parse the *payload* of the file correctly and leverage its data. While existing research uses the term *data preparation* as a broad term for all preprocessing operations [31, 104, 122], we formally define it as the pipeline of heterogeneous steps that are required to wrangle the structure of a file and correctly parse its payload. Presently, data preparation is challenging because it often resorts to an ad-hoc, manual, time-consuming, and error-prone process [31, 53, 66]. We claim that these challenges can be addressed by providing principled models to represent file structure, that can advance data preparation towards a more engineered process.

In Chapter 2, we presented a formal, grammar-based model to represent file structure based on context-free grammars used to serialize and parse tabular files. Our application of this model to survey over 3 000 real-world files showcased how commonly the structure of tabular files differs from the standards supported by existing systems. Drawing on these findings, we leverage our framework to design file *pollutions* and present Pollock, a benchmark to assess a system's robustness in loading data from non-standard CSV files. We experimented with our benchmark on 16 real-world systems, highlighting their shortcomings. We publicly release the code and artifacts to reproduce our benchmark[1], hoping to guide the development of existing and future systems.

In Chapter 3, we proposed a framework to represent the structure of multiregion files, i.e., spreadsheet files that contain multiple regions arranged with custom layouts. Our system, Mondrian, is based on a graph-based model of file *layouts*, and on a similarity-based model to recognize layout *templates*. The intuition behind our approach is that, by identifying recurring layout templates, users can automatically extract and prepare tables that contain information spread across different files. We demonstrate the applicability of our system for end-users thanks to a web-based graphical interface, publicly available online[2].

[1] `https://github.com/HPI-Information-Systems/Pollock`
[2] `https://hpi.de/naumann/sites/mondrian/demo`

In Chapter 4, we introduced MAGRITTE, a machine-learning model to embed a representation of file structure into vectorial embeddings. The architecture of our model is based on transformers and convolutional networks, which learn to represent the structure of a file by leveraging contextual information from the special characters present in a file's character stream. The base MAGRITTE model, pre-trained on almost 1M real-world tabular files, can be used to obtain general, task-independent embeddings, and fine-tuned to address specific data preparation tasks. We experimentally demonstrate the effectiveness of our model on four such tasks: dialect detection, row classification, column type annotation, and data preparation user effort estimation. We publicly release the code, datasets, and weights of the model as a resource for the community[3].

The contributions of this thesis can be considered the first step towards a more principled approach to data preparation. Following the research directions of our work, we outline interesting questions open for future work.

Extensions for the pollution model Currently, we leveraged our grammar-based model to generate files whose dialect is affected by a single pollution. Our experimental results of Section 2.4.5 demonstrate that real-world files may contain multiple pollutions at once and that the performances of SUTs may be impacted more significantly in these cases. In the future, we envision extending our framework to reproduce the presence of multiple pollutions at once, possibly learning from the distribution of pollutions within files of a given dataset and reproducing it synthetically. To do so, a notion of *pollution dependency* must be explicitly modeled, to understand the interaction of the pollution of different rules of a grammar at once. For example, if we consider applying a pollution to change the cell delimiter in the whole file, together with a pollution to include footnote rows to a table, the application of these two is order-sensitive, resulting in different files leading to potentially different system behavior. Moreover, applying multiple pollutions together would also lead to a combinatorial increase of their parameter space. This would require the design of more sophisticated strategies to sample the parameter search space effectively and obtain an applicable benchmark.

Extending Pollock with interacting pollutions would make the benchmark dataset more realistic and open the door to a *dynamic benchmark*, where the datasets can be adapted to fit specific use cases or scenarios. Following the overarching goal of Pollock, we encourage the developers of data loading systems to leverage our benchmark to improve their systems, relieving users of their data preparation burden.

An end-to-end data preparation system In Chapter 3, we showcased how a specific structural model can be integrated within a user-friendly system to assist practitioners during data preparation. One of the significant challenges of data preparation is the wide variety of solutions and systems, each with its own assumptions on the input data, and the lack of a unified system that can serve end-to-end, from raw tabular files to relational tables. The challenges to design such a system are not only of an engineering nature, but also require research efforts on the integration of different tabular models, structural and semantic, as well as regarding the user interaction with files, datasets, and pipelines. Moreover, as highlighted by the practical limitations of Mondrian, several challenges arise

[3] `https://github.com/HPI-Information-Systems/Magritte`

with the complexity of running data preparation operations on large datasets. Addressing these issues requires the design of parallel or distributed algorithms, perhaps trading off accuracy with scalability.

Applications of Structural Embeddings We believe that tabular file embeddings are a promising research direction to advance state-of-the-art data preparation. First, we envision that structural embeddings can prove useful for more tasks than those we experimented with in Chapter 4, for example, table extraction, value normalization, or data transformation from one format to another (e.g., CSV to JSON). Moreover, an interesting direction to explore is the use of file-level embeddings to index datasets of tabular files before relational tables are extracted, e.g., in scenarios like data lakes, or data markets. Finally, we envision the design of more complex models, based on structural as well as semantic embeddings, which can automatically infer and execute whole preparation pipelines given a source file and a target downstream task.

Estimating Data Preparation User Effort In Section 4.3.3, we presented a novel research problem: the estimation of the user effort required to prepare a file. This problem is particularly challenging because it involves not only understanding the preparation required for a given file/task pair but also modeling user behavior and expertise. We believe that this problem is worth investigating further, as it can be useful for several purposes: assessing the time and cost estimation of data-driven projects, guiding the design of more user-friendly systems, and providing a principled framework to evaluate them. Research efforts should be devoted to collect large and representative datasets of unprepared and prepared files and to conduct extensive user studies to obtain ground truth measurements against which to design and evaluate new models.

We end this dissertation certain that data preparation will reach the maturity of other data management tasks and become a well-defined discipline, with its grounding principles, methods, and tools. We envision that the research field of data preparation will continue to grow and attract the attention of several communities for which it is fundamental, e.g., the data management community, the machine learning community, the data visualization and human-computer interaction communities, and potentially extend to data-driven scientific communities outside traditional computer sciences.

Selbstständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Doktorarbeit mit dem Thema:

Modeling the Structure of Tabular Files for Data Preparation

selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Potsdam, den 03.08.2023

References

[1] Apache Software Foundation. *Apache Commons CSV*. July 2022. URL: https://commons.apache.org/proper/commons-csv/ (visited on 05/10/2023).

[2] Marcelo Arenas, Francisco Maturana, Cristian Riveros, Domagoj Vrgoč. "A Framework for Annotating CSV-like Data". *PVLDB* 9.11 (2016), pp. 876–887.

[3] Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, Zachary G. Ives. "DBpedia: A Nucleus for a Web of Open Data". *ISWC/ASWC*. Vol. 4825. Lecture Notes in Computer Science. Springer, 2007, pp. 722–735.

[4] Jeronimo Backes. *Univocity CSV Parser*. 2022. URL: https://github.com/uniVocity/univocity-parsers (visited on 05/10/2023).

[5] Gilbert Badaro, Mohammed Saeed, Paolo Papotti. "Transformers for Tabular Data Representation: A Survey of Models and Applications". *Transactions of the Association for Computational Linguistics (TACL)* 11 (2023), pp. 227–249. ISSN: 2307-387X.

[6] António Leslie Bajuelos, Ana Paula Tomás, Fábio Marques. "Partitioning Orthogonal Polygons by Extension of All Edges Incident to Reflex Vertices: Lower and Upper Bounds on the Number of Pieces". *International Conference on Computational Science and Its Applications (ICCSA)*. 2004, pp. 127–136.

[7] Barry W. Boehm, Chris Abts, Sunita Chulani. "Software development cost estimation approaches - A survey". *Annals of Software Engineering* 10 (2000), pp. 177–205.

[8] Barry W. Boehm, Bradford K. Clark, Ellis Horowitz, J. Christopher Westland, Raymond J. Madachy, Richard W. Selby. "Cost Models for Future Software Life Cycle Processes: COCOMO 2.0". *Annals of Software Engineering* 1 (1995), pp. 57–94.

[9] Tim Bray. *The JavaScript Object Notation (JSON) Data Interchange Format*. RFC 8259. RFC Editor, Dec. 2017. URL: https://www.rfc-editor.org/rfc/rfc8259.txt.

[10] Tim Bray, Jean Paoli, Michael C. Sperberg-McQueen, Eve Maler, François Yergeau. *Extensible Markup Language (XML) 1.0 (Fifth Edition)*. W3C Recommendation REC-xml-20081126. World Wide Web Consortium, Nov. 2008. URL: `https://www.w3.org/TR/2008/REC-xml-20081126/`.

[11] Tom B. Brown et al. "Language Models are Few-Shot Learners". *Proceedings of the International Conference on Conference on Neural Information Processing Systems (NeurIPS)*. 2020.

[12] Gerrit J. J. van den Burg, Alfredo Nazábal, Charles Sutton. "Wrangling Messy CSV Files by Detecting Row and Type Patterns". *Data Mining and Knowledge Discovery* 33.6 (2019), pp. 1799–1820. ISSN: 1573-756X.

[13] Riccardo Cappuzzo, Paolo Papotti, Saravanan Thirumuruganathan. "Creating Embeddings of Heterogeneous Relational Datasets for Data Integration Tasks". *Proceedings of the International Conference on Management of Data (SIGMOD)*. ACM, 2020, pp. 1335–1349.

[14] Valerio Cetorelli, Paolo Atzeni, Valter Crescenzi, Franco Milicchio. "The Smallest Extraction Problem". *PVLDB* 14.11 (2021), pp. 2445–2458.

[15] Zhe Chen, Michael Cafarella, Jun Chen, Daniel Prevo, Junfeng Zhuang. "Senbazuru: A Prototype Spreadsheet Database Management System". *PVLDB* 6.12 (2013), pp. 1202–1205.

[16] Zhe Chen, Sasha Dadiomov, Richard Wesley, Gang Xiao, Daniel Cory, Michael J. Cafarella, Jock D. Mackinlay. "Spreadsheet Property Detection with Rule-Assisted Active Learning". *Proceedings of the International Conference on Information and Knowledge Management (CIKM)*. 2017, pp. 999–1008.

[17] Laura Chiticariu, Yunyao Li, Sriram Raghavan, Frederick R. Reiss. "Enterprise Information Extraction: Recent Developments and Open Challenges". *Proceedings of the International Conference on Management of Data (SIGMOD)*. 2010, pp. 1257–1258.

[18] Noam Chomsky. "Three models for the description of language". *IRE Transactions on Information Theory* 2.3 (1956), pp. 113–124.

[19] Noam Chomsky, Marcel P Schützenberger. "The algebraic theory of context-free languages". *Studies in Logic and the Foundations of Mathematics*. Vol. 26. Elsevier, 1959, pp. 118–161.

[20] Aakanksha Chowdhery et al. "PaLM: Scaling Language Modeling with Pathways". *CoRR* abs/2204.02311 (2022).

[21] Christina Christodoulakis, Eric Munson, Moshe Gabel, Angela Demke Brown, Renée J. Miller. "Pytheas: Pattern-based Table Discovery in CSV Files". *PVLDB* 13.11 (2020), pp. 2075–2089.

[22] Edgar F Codd. "A relational model of data for large shared data banks". *Communications of the ACM* 13.6 (1970), pp. 377–387.

[23] Remi Coletta, Emmanuel Castanier, Patrick Valduriez, Christian Frisch, DuyHoa Ngo, Zohra Bellahsene. "Public Data Integration with WebSmatch". *Proceedings of the International Workshop on Open Data (WOD)*. 2012, pp. 5–12.

[24] Crown, UK. *UK Open Data Portal*. data.gov.uk. 2021.

[25] Norman Dalkey, Olaf Helmer. "An experimental application of the Delphi method to the use of experts". *Management science* 9.3 (1963), pp. 458–467.

[26] Xiang Deng, Huan Sun, Alyssa Lees, You Wu, Cong Yu. "TURL: Table Understanding through Representation Learning". *PVLDB* 14.3 (2020), pp. 307–319.

[27] Jacob Devlin, Ming-Wei Chang, Kenton Lee, Kristina Toutanova. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, (NAACL-HLT)*. Vol. 1. Association for Computational Linguistics, 2019, pp. 4171–4186.

[28] Till Döhmen, Hannes Mühleisen, Peter Boncz. "Multi-Hypothesis CSV Parsing". *Proceedings of the International Conference on Scientific and Statistical Database Management (SSDBM)*. ACM, 2017, pp. 1–12.

[29] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu. "A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise". *Proceedings of the International Conference on Knowledge Discovery and Data Mining (SIGKDD)*. 1996, pp. 226–231.

[30] Grace Fan, Jin Wang, Yuliang Li, Dan Zhang, Renée J. Miller. "Semantics-aware Dataset Discovery from Data Lakes with Contextualized Column-based Representation Learning". *PVLDB* 16.7 (2023), pp. 1726–1739.

[31] Alvaro A. A. Fernandes, Martin Koehler, Nikolaos Konstantinou, Pavel Pankin, Norman W. Paton, Rizos Sakellariou. "Data Preparation: A Technological Perspective and Review". *SN Computer Science* 4.4 (2023), p. 425.

[32] Raul Castro Fernandez, Ziawasch Abedjan, Famien Koko, Gina Yuan, Samuel Madden, Michael Stonebraker. "Aurum: A Data Discovery System". *Proceedings of the International Conference on Data Engineering (ICDE)*. IEEE Computer Society, 2018, pp. 1001–1012.

[33] Marc Fisher, Gregg Rothermel. "The EUSES Spreadsheet Corpus: A Shared Resource for Supporting Experimentation with Spreadsheet Dependability Mechanisms". *ACM SIGSOFT Software Engineering Notes* 30.4 (2005), pp. 1–5. ISSN: 0163-5948.

[34] Chang Ge, Yinan Li, Eric Eilebrecht, Badrish Chandramouli, Donald Kossmann. "Speculative Distributed CSV Data Parsing for Big Data Analytics". *Proceedings of the International Conference on Management of Data (SIGMOD)*. 2019, pp. 883–899.

[35] Azka Gilani, Shah Rukh Qasim, Imran Malik, Faisal Shafait. "Table Detection Using Deep Learning". *Proceedings of the IAPR International Conference on Document Analysis and Recognition (ICDAR)*. 2017, pp. 771–776.

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

[37] Google, Inc. *OpenRefine*. 2021. URL: `www . openrefine . org` (visited on 05/10/2023).

[38] The PostgreSQL Global Development Group. *PostgreSQL*. 2022. URL: `https://www.postgresql.org` (visited on 05/10/2023).

[39] Mazhar Hameed, Felix Naumann. "Data Preparation: A Survey of Commercial Tools". *SIGMOD Record* 49.3 (2020), pp. 18–29. ISSN: 0163-5808.

[40] Mazhar Hameed, Gerardo Vitagliano, Lan Jiang, Felix Naumann. "SURAGH: Syntactic Pattern Matching to Identify Ill-Formed Records". *Proceedings of the International Conference on Extending Database Technology (EDBT)*. OpenProceedings.org, 2022.

[41] Dong Haoyu, Liu Shijie, Han Shi, Fu Zhouyu, Zhang Dongmei. "TableSense: Spreadsheet Table Detection with Convolutional Neural Networks". *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*. 2019, pp. 69–76.

[42] Kaiming He, Georgia Gkioxari, Piotr Dollár, Ross Girshick. "Mask r-cnn". *Proceedings of the International Conference on Computer Vision (ICCV)*. 2017, pp. 2961–2969.

[43] Kaiming He, Georgia Gkioxari, Piotr Dollár, Ross B. Girshick. "Mask R-CNN". *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*. 2017, pp. 2980–2988.

[44] Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun. "Deep Residual Learning for Image Recognition". *Proceedings of the International Conference on Computer*

Vision and Pattern Recognition (CVPR). IEEE Computer Society, 2016, pp. 770–778.

[45] Joseph M. Hellerstein, Jeffrey Heer, Sean Kandel. "Self-Service Data Preparation: Research to Practice." *IEEE Data Engineering Bulletin*. IEEE Computer Society, 2018.

[46] Felienne Hermans, Emerson Murphy-Hill. "Enron's Spreadsheets and Related Emails: A Dataset and Analysis". *Proceedings of the International Conference on Software Engineering (ICSE)*. 2015, pp. 7–16.

[47] Richard D Hipp. *SQLite*. Version 3.39.0. 2022. URL: `https://www.sqlite.org/index.html`.

[48] Renáta Hodován, Ákos Kiss, Tibor Gyimóthy. "Grammarinator: A Grammar-Based Open Source Fuzzer". *A-TEST@ESEC/SIGSOFT FSE*. ACM, 2018, pp. 45–48.

[49] Leonardo Hübscher, Lan Jiang, Felix Naumann. "ExtracTable: Extracting Tables from Raw Data Files". *Proceedings of the Conference Datenbanksysteme in Büro, Technik und Wissenschaft (BTW)*. Vol. P-331. LNI. Gesellschaft für Informatik e.V., 2023, pp. 417–438.

[50] Madelon Hulsebos, Paul Groth, Çagatay Demiralp. "GitTables: A Large-Scale Corpus of Relational Tables". *Proceedings of the International Conference on Management of Data (SIGMOD)*. ACM, 2023, pp. 49–64.

[51] Madelon Hulsebos, Kevin Zeng Hu, Michiel A. Bakker, Emanuel Zgraggen, Arvind Satyanarayan, Tim Kraska, Çagatay Demiralp, César A. Hidalgo. "Sherlock: A Deep Learning Approach to Semantic Data Type Detection". *Proceedings of the International Conference on Knowledge Discovery and Data Mining (SIGKDD)*. ACM, 2019, pp. 1500–1508.

[52] Hiroshi Iida, Dung Thai, Varun Manjunatha, Mohit Iyyer. "TABBIE: Pretrained Representations of Tabular Data". *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Association for Computational Linguistics, 2021, pp. 3446–3456.

[53] Ihab F Ilyas, Theodoros Rekatsinas. "Machine Learning and Data Cleaning: Which Serves the Other?" *Journal on Data and Information Quality* 14.3 (2022), pp. 1–11.

[54] Ziwei Ji, Nayeon Lee, Rita Frieske, Tiezheng Yu, Dan Su, Yan Xu, Etsuko Ishii, Ye Jin Bang, Andrea Madotto, Pascale Fung. "Survey of hallucination in natural language generation". *ACM Computing Surveys* 55.12 (2023), pp. 1–38.

[55] Lan Jiang, Gerardo Vitagliano, Felix Naumann. "A Scoring-based Approach for Data Preparator Suggestion". *Proceedings of the Conference on "Lernen, Wissen, Daten, Analysen" (LWDA)*. Vol. 2454. CEUR Workshop Proceedings. CEUR-WS.org, 2019, pp. 6–9.

[56] Lan Jiang, Gerardo Vitagliano, Felix Naumann. "Structure Detection in Verbose CSV Files". *Proceedings of the International Conference on Extending Database Technology (EDBT)*. OpenProceedings.org, 2021, pp. 193–204.

[57] Magne Jørgensen. "A review of studies on expert estimation of software development effort". *Journal of Systems and Software (JSS)* 70.1-2 (2004), pp. 37–60.

[58] Magne Jørgensen, Martin J. Shepperd. "A Systematic Review of Software Development Cost Estimation Studies". *IEEE Transactions of Software Engineering (TSE)* 33.1 (2007), pp. 33–53.

[59] Diederik P. Kingma, Max Welling. "Auto-Encoding Variational Bayes". *Proceddings of the International Conference on Learning Representations (ICLR)*. 2014.

[60] Thomas Kluyver et al. "Jupyter Notebooks - a publishing format for reproducible computational workflows". *International Conference on Electronic Publishing (ELPUB)*. IOS Press, 2016, pp. 87–90.

[61] Elvis Koci, Maik Thiele, Wolfgang Lehner, Oscar Romero. "Table Recognition in Spreadsheets via a Graph Representation". *Proceedings of the IAPR International Workshop on Document Analysis Systems (DAS)*. 2018, pp. 139–144.

[62] Elvis Koci, Maik Thiele, Josephine Rehak, Oscar Romero, Wolfgang Lehner. "DECO: A Dataset of Annotated Spreadsheets for Layout and Table Recognition". *Proceedings of the IAPR International Conference on Document Analysis and Recognition (ICDAR)*. 2019, pp. 1280–1285.

[63] Elvis Koci, Maik Thiele, Oscar Romero, Wolfgang Lehner. "A Genetic-Based Search for Adaptive Table Recognition in Spreadsheets". *Proceedings of the IAPR International Conference on Document Analysis and Recognition (ICDAR)*. 2019, pp. 1274–1279.

[64] Elvis Koci, Maik Thiele, Oscar Romero, Wolfgang Lehner. "A Machine Learning Approach for Layout Inference in Spreadsheets". *Proceedings of the International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management (IC3K)*. 2016, pp. 77–88.

[65] Christos Koutras, George Siachamis, Andra Ionescu, Kyriakos Psarakis, Jerry Brons, Marios Fragkoulis, Christoph Lofi, Angela Bonifati, Asterios Katsifodimos.

"Valentine: Evaluating Matching Techniques for Dataset Discovery". *Proceedings of the International Conference on Data Engineering (ICDE)*. 2021, pp. 468–479.

[66] Arun Kumar. "Automation of Data Prep, ML, and Data Science: New Cure or Snake Oil?" *Proceedings of the International Conference on Management of Data (SIGMOD)*. ACM, 2021, pp. 2878–2880.

[67] Yuliang Li, Jinfeng Li, Yoshihiko Suhara, AnHai Doan, Wang-Chiew Tan. "Deep Entity Matching with Pre-Trained Language Models". *PVLDB* 14.1 (2020), pp. 50–60.

[68] MariaDB Foundation. *MariaDB*. 2022. URL: `www.mariadb.org` (visited on 05/10/2023).

[69] Wim Martens, Frank Neven, Stijn Vansummeren. "SCULPT: A Schema Language for Tabular Data on the Web". en. *Proceedings of the International World Wide Web Conference (WWW)*. Florence Italy: International World Wide Web Conferences Steering Committee, 2015, pp. 702–720.

[70] Sergey Melnik, Hector Garcia-Molina, Erhard Rahm. *Similarity Flooding: A Versatile Graph Matching Algorithm (Extended Technical Report)*. Technical Report 2001-25. Stanford / Stanford InfoLab, 2001.

[71] Sergey Melnik, Hector Garcia-Molina, Erhard Rahm. "Similarity Flooding: A Versatile Graph Matching Algorithm and Its Application to Schema Matching". *Proceedings of the International Conference on Data Engineering (ICDE)*. 2002, pp. 117–128.

[72] Mendeley Ltd. *Mendeley Data*. data.mendeley.com. 2021.

[73] Zhengjie Miao, Yuliang Li, Xiaolan Wang. "Rotom: A Meta-Learned Data Augmentation Framework for Entity Matching, Data Cleaning, Text Classification, and Beyond". *Proceedings of the International Conference on Management of Data (SIGMOD)*. New York, NY, USA: ACM, 2021, pp. 1303–1316. ISBN: 9781450383431. DOI: `10.1145/3448016.3457258`. URL: `https://doi.org/10.1145/3448016.3457258`.

[74] Microsoft Corporation. *Microsoft Excel*. 2022. URL: `www.microsoft.com/excel` (visited on 05/10/2023).

[75] Johann Mitlöhner, Sebastian Neumaier, Jürgen Umbrich, Axel Polleres. "Characteristics of open data CSV files". *Proceedings of the Image Analysis and Processing Conference (ICIAP)*. 2016, pp. 72–79.

[76] Avanika Narayan, Ines Chami, Laurel J. Orr, Christopher Ré. "Can Foundation Models Wrangle Your Data?" *PVLDB* 16.4 (2022), pp. 738–746.

[77] Fatemeh Nargesian, Erkang Zhu, Renée J. Miller Miller, Ken Q. Pu, Patricia C. Arocena. "Data Lake Management: Challenges and Opportunities". *PVLDB*. Vol. 11. 2019, pp. 813–825.

[78] Fatemeh Nargesian, Erkang Zhu, Ken Q Pu, Renée J Miller. "Table Union Search on Open Data". *PVLDB* 11.7 (2018), pp. 813–825.

[79] Oracle Corporation. *MySQL*. 2022. URL: `www.mysql.com` (visited on 05/10/2023).

[80] Noseong Park, Mahmoud Mohammadi, Kshitij Gorde, Sushil Jajodia, Hongkyu Park, Youngmin Kim. "Data Synthesis based on Generative Adversarial Networks". *PVLDB* 11.10 (2018), pp. 1071–1083.

[81] Marvin C. Paull, Stephen H. Unger. "Structural Equivalence of Context-Free Grammars". *Journal of Computer and System Science (JCSS)* 2.4 (1968), pp. 427–463.

[82] Ralph Peeters, Christian Bizer. "Dual-Objective Fine-Tuning of BERT for Entity Matching". *PVLDB* 14.10 (2021), pp. 1913–1921. ISSN: 2150-8097.

[83] Meikel Poess, Tilmann Rabl, Hans-Arno Jacobsen, Brian Caufield. "TPC-DI: The First Industry Benchmark for Data Integration". *PVLDB* 7.13 (2014), pp. 1367–1378. ISSN: 2150-8097.

[84] Abdulhakim Ali Qahtan, Ahmed K. Elmagarmid, Mourad Ouzzani, Nan Tang. "FAHES: Detecting Disguised Missing Values". *Proceedings of the International Conference on Data Engineering (ICDE)*. IEEE Computer Society, 2018, pp. 1609–1612.

[85] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing. Vienna, Austria, 2022. URL: `https://www.R-project.org/` (visited on 05/10/2023).

[86] Alec Radford, Luke Metz, Soumith Chintala. "Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks". *Proceedings of the International Conference on Learning Representations (ICLR)*. 2016.

[87] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever. *Improving Language Understanding by Generative Pre-Training*. Tech. rep. 2018.

[88] Yasaman Razeghi, Robert L Logan IV, Matt Gardner, Sameer Singh. "Impact of Pretraining Term Frequencies on Few-Shot Numerical Reasoning". *Findings of the Association for Computational Linguistics (EMNLP)*. Abu Dhabi, United Arab Emirates: Association for Computational Linguistics, Dec. 2022, pp. 840–854. URL: `https://aclanthology.org/2022.findings-emnlp.59`.

[89] Jeff Reback et al. *pandas-dev/pandas: Pandas 1.4.3*. Version v1.4.3. June 2022. DOI: 10.5281/zenodo.6702671. URL: https://doi.org/10.5281/zenodo.6702671.

[90] Theodoros Rekatsinas, Xu Chu, Ihab F. Ilyas, Christopher Ré. "HoloClean: holistic data repairs with probabilistic inference". *PVLDB* 10.11 (2017), pp. 1190–1201.

[91] Anna Rogers, Olga Kovaleva, Anna Rumshisky. "A Primer in BERTology: What We Know About How BERT Works". *Transactions of the Association for Computational Linguistics (TACL)* 8 (2020), pp. 842–866.

[92] Andrew Rosenberg, Julia Hirschberg. "V-Measure: A Conditional Entropy-Based External Cluster Evaluation Measure". *Proceedings of the Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*. 2007, pp. 410–420.

[93] Howard A Rubin. "Macro-estimation of software development parameters: The ESTIMACS system". *SOFTFAIR Conference on Software Development Tools, Techniques and Alternatives*. IEEE Press. 1983, pp. 109–118.

[94] Y. Shafranovich. *Common Format and MIME Type for Comma-Separated Values (CSV) Files*. RFC 4180. RFC Editor, Nov. 2005, pp. 1–8. URL: http://www.rfc-editor.org/rfc/rfc4180.txt.

[95] Vraj Shah, Jonathan Lacanlale, Premanand Kumar, Kevin Yang, Arun Kumar. "Towards Benchmarking Feature Type Inference for AutoML Platforms". *Proceedings of the International Conference on Management of Data (SIGMOD)*. ACM, 2021, pp. 1584–1596.

[96] Lisa Singh et al. "NSF BIGDATA PI Meeting - Domain-Specific Research Directions and Data Sets". *SIGMOD Record* 47.3 (2019), pp. 32–35. ISSN: 0163-5808.

[97] Glen Smith, Scott Conway, Andrew Rucker Jones, Sean Sullivan, Kyle Miller, Tom Squires, Kyle Miller, Maciek Opala, J.C. Romanda. *OpenCSV - Project page*. July 2022. URL: http://opencsv.sourceforge.net (visited on 05/10/2023).

[98] Ezekiel O. Soremekun, Esteban Pavese, Nikolas Havrikov, Lars Grunske, Andreas Zeller. "Inputs from Hell". *IEEE Transaction on Software Engineering* 48.4 (2022), pp. 1138–1153.

[99] Elias Stehle, Hans-Arno Jacobsen. "ParPaRaw: Massively Parallel Parsing of Delimiter-Separated Raw Data". *PVLDB* 13.5 (2020), pp. 616–628.

[100] Yoshihiko Suhara, Jinfeng Li, Yuliang Li, Dan Zhang, Çagatay Demiralp, Chen Chen, Wang-Chiew Tan. "Annotating Columns with Pre-trained Language

Models". *Proceedings of the International Conference on Management of Data (SIGMOD)*. ACM, 2022, pp. 1493–1503.

[101] Yushi Sun, Hao Xin, Lei Chen. "RECA: Related Tables Enhanced Column Semantic Type Annotation Framework". *PVLDB* 16.6 (2023), pp. 1319–1331.

[102] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, Zbigniew Wojna. "Rethinking the inception architecture for computer vision". *Proceedings of the International Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016, pp. 2818–2826.

[103] Tableau Software, LLC. *Tableau*. 2022. URL: `www.tableau.com` (visited on 05/10/2023).

[104] Nan Tang, Ju Fan, Fangyi Li, Jianhong Tu, Xiaoyong Du, Guoliang Li, Samuel Madden, Mourad Ouzzani. "RPT: Relational Pre-trained Transformer Is Almost All You Need towards Democratizing Data Preparation". *PVLDB* 14.8 (2021), pp. 1254–1261.

[105] Robert C. Tausworthe. "The work breakdown structure in software project management". *Journal of Systems and Software* 1 (1979), pp. 181–186.

[106] Jeni Tennison, Gregg Kellogg, Ivan Herman. *Model for Tabular Data and Metadata on the Web*. W3C Recommendation. https://www.w3.org/TR/2015/REC-tabular-data-model-20151217/. W3C, Dec. 2015.

[107] The Document Foundation. *LibreOffice Calc 7.3.4*. 2022. URL: `https://www.libreoffice.org/discover/calc/` (visited on 05/10/2023).

[108] Barik Titus, Lubick Kevin, Smith Justin, Slankas John, Murphy-Hill Emerson R. "Fuse: A Reproducible, Extendable, Internet-Scale Corpus of Spreadsheets". *IEEE/ACM Working Conference on Mining Software Repositories, MSR*. 2015, pp. 486–489.

[109] Hugo Touvron et al. "LLaMA: Open and Efficient Foundation Language Models". *CoRR* abs/2302.13971 (2023).

[110] Trifacta Inc. *Trifacta Data Engineering Cloud*. 2021. URL: `www.trifacta.com` (visited on 05/10/2023).

[111] Guido Van Rossum, Fred L. Drake. *Python 3 Reference Manual*. Scotts Valley, CA: CreateSpace, 2009. ISBN: 1441412697.

[112] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, Illia Polosukhin. "Attention is All you Need". *Advances in Neural Information Processing Systems (NIPS)*. 2017, pp. 5998–6008.

[113] Gerardo Vitagliano, Mazhar Hameed, Lan Jiang, Lucas Reisener, Eugene Wu, Felix Naumann. "Pollock: A Data Loading Benchmark". *PVLDB* 16.8 (2023), pp. 1870–1882. DOI: 10.14778/3594512.3594518.

[114] Gerardo Vitagliano, Mazhar Hameed, Felix Naumann. "Structural Embedding of Data Files with MaGRiTTE". *Table Representation Workshop at NeurIPS*. 2022.

[115] Gerardo Vitagliano, Mazhar Hameed, Alejandro Sierra-Múnera, Felix Naumann. "Structural Embeddings for Tabular File Preparation". *Under review* ().

[116] Gerardo Vitagliano, Lan Jiang, Felix Naumann. "Detecting Layout Templates in Complex Multiregion Files". *PVLDB* 15.3 (2022), pp. 646–658.

[117] Gerardo Vitagliano, Lucas Reisener, Lan Jiang, Mazhar Hameed, Felix Naumann. "Mondrian: Spreadsheet Layout Detection". *Proceedings of the International Conference on Management of Data (SIGMOD)*. ACM, 2022, pp. 2361–2364.

[118] David Vos, Till Döhmen, Sebastian Schelter. "Towards Parameter-Efficient Automation of Data Wrangling Tasks with Prefix-Tuning". *Table Representation Workshop (TRL) at NeurIPS*. 2022.

[119] Eric Wallace, Yizhong Wang, Sujian Li, Sameer Singh, Matt Gardner. "Do NLP Models Know Numbers? Probing Numeracy in Embeddings". *Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 2019, pp. 5306–5314.

[120] Daheng Wang, Prashant Shiralkar, Colin Lockard, Binxuan Huang, Xin Luna Dong, Meng Jiang. "TCN: Table Convolutional Network for Web Table Interpretation". *Proceedings of the International World Wide Web Conference (WWW)*. ACM / IW3C2, 2021, pp. 4020–4032.

[121] Lei Xu, Maria Skoularidou, Alfredo Cuesta-Infante, Kalyan Veeramachaneni. "Modeling Tabular data using Conditional GAN". *NeurIPS*. 2019, pp. 7333–7343.

[122] Cong Yan, Yeye He. "Auto-Suggest: Learning-to-Recommend Data Preparation Steps Using Data Science Notebooks". *Proceedings of the International Conference on Management of Data (SIGMOD)*. ACM, 2020, pp. 1539–1554.

[123] Jingfeng Yang, Aditya Gupta, Shyam Upadhyay, Luheng He, Rahul Goel, Shachi Paul. "TableFormer: Robust Transformer Modeling for Table-Text Encoding". *ACL (1)*. Association for Computational Linguistics, 2022, pp. 528–537.

[124] Pengcheng Yin, Graham Neubig, Wen-tau Yih, Sebastian Riedel. "TaBERT: Pretraining for Joint Understanding of Textual and Tabular Data". *ACL*. Association for Computational Linguistics, 2020, pp. 8413–8426.

[125] Richard Zanibbi, Dorothea Blostein, James R Cordy. "A survey of table recognition". *Document Analysis and Recognition* 7.1 (2004), pp. 1–16.

[126] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, Christian Holler. *The Fuzzing Book*. Saarbrücken: CISPA + Saarland University, 2019.

[127] Dan Zhang, Yoshihiko Suhara, Jinfeng Li, Madelon Hulsebos, Çagatay Demiralp, Wang-Chiew Tan. "Sato: Contextual Semantic Type Detection in Tables". *PVLDB* 13.11 (2020), pp. 1835–1848.

[128] Qian Zhang, Jiyuan Wang, Muhammad Ali Gulzar, Rohan Padhye, Miryung Kim. "BigFuzz: Efficient Fuzz Testing for Data Analytics Using Framework Abstraction". *ASE*. IEEE, 2020, pp. 722–733.

[129] Yi Zhang, Zachary G. Ives. "Finding Related Tables in Data Lakes for Interactive Data Science". *Proceedings of the International Conference on Management of Data (SIGMOD)*. ACM, 2020, pp. 1951–1966.

[130] Yi Zhang, Zachary G. Ives. "Finding Related Tables in Data Lakes for Interactive Data Science". *Proceedings of the International Conference on Management of Data (SIGMOD)*. 2020, pp. 1951–1966.

[131] Erkang Zhu, Dong Deng, Fatemeh Nargesian, Renée J. Miller. "JOSIE: Overlap Set Similarity Search for Finding Joinable Tables in Data Lakes". *Proceedings of the International Conference on Management of Data (SIGMOD)*. 2019, pp. 847–864.