Layout Inference and Table Detection in Spreadsheet Documents

Dissertation

submitted April 20, 2020

by **M.Sc. Elvis Koci**
born May 09, 1987 in Sarande, Albania

at Technische Universität Dresden
and Universitat Politècnica de Catalunya

Supervisors:
Prof. Dr.-Ing. Wolfgang Lehner
Assoc. Prof. Dr. Oscar Romero

THESIS DETAILS

Thesis Title:	Layout Inference and Table Detection in Spreadsheet Documents
Ph.D. Student:	Elvis Koci
Supervisors:	Prof. Dr.-Ing. Wolfgang Lehner, Technische Universität Dresden
	Assoc. Prof. Dr. Oscar Romero, Universitat Politècnica de Catalunya

The main body of this thesis consists of the following peer-reviewed publications:

1. Elvis Koci, Maik Thiele, Oscar Romero, and Wolfgang Lehner. A machine learning approach for layout inference in spreadsheets. In *IC3K 2016: The 8th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management: volume 1: KDIR*, pages 77–88. SciTePress, 2016

2. Elvis Koci, Maik Thiele, Oscar Romero, and Wolfgang Lehner. Cell classification for layout recognition in spreadsheets. In Ana Fred, Jan Dietz, David Aveiro, Kecheng Liu, Jorge Bernardino, and Joaquim Filipe, editors, *Knowledge Discovery, Knowledge Engineering and Knowledge Management (IC3K '16: Revised Selected Papers)*, volume 914 of *Communications in Computer and Information Science*, pages 78–100. Springer, Cham, 2019

3. Elvis Koci, Maik Thiele, Oscar Romero, and Wolfgang Lehner. Table identification and reconstruction in spreadsheets. In *the International Conference on Advanced Information Systems Engineering (CAiSE)*, pages 527–541. Springer, 2017

4. Elvis Koci, Maik Thiele, Wolfgang Lehner, and Oscar Romero. Table recognition in spreadsheets via a graph representation. In *the 13th IAPR International Workshop on Document Analysis Systems (DAS)*, pages 139–144. IEEE, 2018

5. Elvis Koci, Maik Thiele, Oscar Romero, and Wolfgang Lehner. A genetic-based search for adaptive table recognition in spreadsheets. In *2019 International Conference on Document Analysis and Recognition, ICDAR 2019, Sydney, Australia, September 20-25, 2019*, pages 1274–1279. IEEE, 2019

6. Elvis Koci, Maik Thiele, Josephine Rehak, Oscar Romero, and Wolfgang Lehner. DECO: A dataset of annotated spreadsheets for layout and table recognition. In *2019 International Conference on Document Analysis and Recognition, ICDAR 2019, Sydney, Australia, September 20-25, 2019*, pages 1280–1285. IEEE, 2019

7. Elvis Koci, Dana Kuban, Nico Luettig, Dominik Olwig, Maik Thiele, Julius Gonsior, Wolfgang Lehner, and Oscar Romero. Xlindy: Interactive recognition and information extraction in spreadsheets. In Sonja Schimmler and Uwe M. Borghoff, editors, *Proceedings of the ACM Symposium on Document Engineering 2019, Berlin, Germany, September 23-26, 2019*, pages 25:1–25:4. ACM, 2019

This thesis is jointly submitted to the Faculty of Computer Science at Technische Universität Dresden (TUD) and the Department of Service and Information System Engineering (ESSI) at Universitat Politècnica de Catalunya (UPC), in partial fulfillment of the requirements within the scope of the IT4BI-DC program for the joint Ph.D. degree in computer science (TUD: Dr.-Ing., UPC: Ph.D. in Computer Science). The thesis is not submitted to any other organization at the same time. The author has obtained the rights to include parts of the already published articles in the thesis.

ABSTRACT

Spreadsheet applications have evolved to be a tool of great importance for businesses, open data, and scientific communities. Using these applications, users can perform various transformations, generate new content, analyze and format data such that they are visually comprehensive. The same data can be presented in different ways, depending on the preferences and the intentions of the user.

These functionalities make spreadsheets user-friendly, but not as much machine-friendly. When it comes to integrating with other sources, the free-for-all nature of spreadsheets is disadvantageous. It is rather difficult to algorithmically infer the structure of the data when they are intermingled with formatting, formulas, layout artifacts, and textual metadata. Therefore, user involvement is often required, which results in cumbersome and time-consuming tasks. Overall, the lack of automatic processing methods limits our ability to explore and reuse a great amount of rich data stored into partially-structured documents such as spreadsheets.

In this thesis, we tackle this open challenge, which so far has been scarcely investigated in literature. Specifically, we are interested in extracting tabular data from spreadsheets, since they hold concise, factual, and to a large extend structured information. It is easier to process such information, in order to make it available to other applications. For instance, spreadsheet (tabular) data can be loaded into databases. Thus, these data would become instantly available to existing or new business processes. Furthermore, we can eliminate the risk of losing valuable company knowledge, by moving data or integrating spreadsheets with other more sophisticated information management systems.

To achieve the aforementioned objectives and advancements, in this thesis, we develop a spreadsheet processing pipeline. The requirements for this pipeline were derived from a large scale empirical analysis of real-world spreadsheets, from business and Web settings. Specifically, we propose a series of specialized steps that build on top of each other with the goal of discovering the structure of data in spreadsheet documents. Our approach is bottom-up, as it starts from the smallest unit (i.e., the cell) to ultimately arrive at the individual tables of the sheet.

Additionally, this thesis makes use of sophisticated machine learning and optimization techniques. In particular, we apply these techniques for layout analysis and table detection in spreadsheets. We target highly diverse sheet layouts, with one or multiple tables and arbitrary arrangement of contents. Moreover, we foresee the presence of textual metadata and other non-tabular data in the sheet. Furthermore, we work even with problematic tables (e.g., containing empty rows/columns and missing values). Finally, we bring flexibility to our approach. This not only allows us to tackle the above-mentioned challenges but also to reuse our solution for different (spreadsheet) datasets.

CONTENTS

1 Introduction **13**

1.1 Motivation 14

1.2 Contributions 15

1.3 Outline 16

2 Foundations and Related Work **19**

2.1 The Evolution of Spreadsheet Documents 20
2.1.1 Spreadsheet User Interface and Functionalities 21
2.1.2 Spreadsheet File Formats 22
2.1.3 Spreadsheets Are Partially-Structured 23

2.2 Analysis and Recognition in Electronic Documents 23
2.2.1 A General Overview of DAR 23
2.2.2 DAR in Spreadsheets 26

2.3 Spreadsheet Research Areas 26
2.3.1 Layout Inference and Table Recognition 27
2.3.2 Unifying Databases and Spreadsheets 29
2.3.3 Spreadsheet Software Engineering 30
2.3.4 Data Wrangling Approaches 31

3 An empirical study of spreadsheet documents **33**

3.1 Available Corpora 34

3.2 Creating a Gold Standard Dataset 36
3.2.1 Initial Selection 36
3.2.2 Annotation Methodology 37

3.3 Dataset Analysis 42
3.3.1 Takeaways from Business Spreadsheets 42
3.3.2 Comparison Between Domains 47

3.4 Summary and Discussion 50
3.4.1 Datasets for Experimental Evaluation 52
3.4.2 A Processing Pipeline 52

4 Layout Analysis **55**

4.1 A Method for Layout Analysis in Spreadsheets 56

4.2 Feature Extraction . . . 58
4.2.1 Content Features . . . 58
4.2.2 Style Features . . . 59
4.2.3 Font Features . . . 60
4.2.4 Formula and Reference Features . . . 60
4.2.5 Spatial Features . . . 61
4.2.6 Geometrical Features . . . 63
4.3 Cell Classification . . . 63
4.3.1 Classification Datasets . . . 64
4.3.2 Classifiers and Assessment Methods . . . 65
4.3.3 Optimum Under-Sampling . . . 66
4.3.4 Feature Selection . . . 68
4.3.5 Parameter Tuning . . . 71
4.3.6 Classification Evaluation . . . 72
4.4 Layout Regions . . . 79
4.5 Summary and Discussions . . . 82

5 CLASSIFICATION POST-PROCESSING 83

5.1 Dataset for Post-Processing . . . 84
5.2 Pattern-Based Revisions . . . 85
5.2.1 Misclassification Patterns . . . 86
5.2.2 Relabeling Cells . . . 87
5.2.3 Evaluating the Patterns . . . 87
5.3 Region-Based Revisions . . . 88
5.3.1 Standardization Procedure . . . 88
5.3.2 Extracting Features from Regions . . . 91
5.3.3 Identifying Misclassified Regions . . . 94
5.3.4 Relabeling Misclassified Regions . . . 96
5.4 Summary and Discussion . . . 97

6 TABLE DETECTION 99

6.1 A Method for Table Detection in Spreadsheets . . . 100
6.2 Preliminaries . . . 102
6.2.1 Introducing a Graph Model . . . 102
6.2.2 Graph Partitioning for Table Detection . . . 105
6.2.3 Pre-Processing for Table Detection . . . 105
6.3 Rule-Based Detection . . . 108
6.3.1 Remove and Conquer . . . 109
6.4 Genetic-Based Detection . . . 114
6.4.1 Undirected Graph . . . 114
6.4.2 Header Cluster . . . 114

6.4.3 Quality Metrics . . . 115
6.4.4 Objective Function . . . 117
6.4.5 Weight Tuning . . . 118
6.4.6 Genetic Search . . . 119
6.5 Experimental Evaluation . . . 120
6.5.1 Testing Datasets . . . 120
6.5.2 Training Datasets . . . 120
6.5.3 Tuning Rounds . . . 122
6.5.4 Search and Assessment . . . 122
6.5.5 Evaluation Results . . . 123
6.6 Summary and Discussions . . . 125

7 XLINDY: A RESEARCH PROTOTYPE 127

7.1 Interface and Functionalities . . . 128
7.1.1 Front-end Walkthrough . . . 128
7.2 Implementation Details . . . 129
7.2.1 Interoperability . . . 130
7.2.2 Efficient Reads . . . 130
7.3 Information Extraction . . . 131
7.4 Summary and Discussions . . . 132

8 CONCLUSION 133

8.1 Summary of Contributions . . . 134
8.2 Directions of Future Work . . . 135

BIBLIOGRAPHY 139

LIST OF FIGURES 149

LIST OF TABLES 153

A ANALYSIS OF REDUCED SAMPLES 155

B TABLE DETECTION WITH TIRS 157

B.1 Tables in TIRS . . . 157
B.2 Pairing Fences with Data Regions . . . 158
B.3 Heuristics Framework . . . 158

ACKNOWLEDGMENTS

This dissertation would not have been possible without the help and support of many colleagues, friends, and family members. First and foremost, I would like to thank Prof. Wolfgang Lehner for giving me the opportunity to pursue my PhD studies as a part of his research group. I am especially grateful for his welcoming and supportive nature. In crucial moments, he was there to provide his guidance and help. I am very thankful to my second supervisor, Assoc. Prof. Oscar Romero, from Universitat Politècnica de Catalunya (UPC). Despite the distance and unconventional nature of this research topic, he trusted, believed, and supported me, during my PhD studies. Moreover, I would like to acknowledge the hospitality that he and the other members of the team showed during my several visits to UPC. Special thanks go to Prof. Jordi Vitrià, from the University of Barcelona. His suggestions and advice were a catalyst for many of the ideas that later became a core part of this thesis. This PhD would have not been possible, without the help and encouragement of Maik Thiele. Not only was he my closest collaborator, but also a true friend. He supported me and believed in me, even when I did not. It is especially because of him and Ulrike Schöbel that I felt part of the group. Nevertheless, I would like to thank all my colleagues from Database Systems Group, Technische Universität Dresden (TUD). Throughout the last five years, we have shared some wonderful movements. Most importantly, I thank them for the countless times they were there to answer my questions, sometimes stupid ones. One can only imagine the confusion a foreign student has when moving to a new unknown environment. An especially warm thank you to my parents, Sotiris and Glikeria, as well as to my brother, Anastasis. Throughout the last three decades, I could always count on them. They have always supported my dreams, and they have always been by my side in good and rough times. Last, I want to thank my wonderful wife, Elena, and my newborn son, Alexandros. You are the joys of my life. I am looking forward to all the beautiful moments and adventures that lay ahead of us.

Elvis Koci
Dresden, 20 April 2020

INTRODUCTION

1.1 Motivation

1.2 Contributions

1.3 Outline

1.1 MOTIVATION

Spreadsheets have found wide use in many different domains and settings. They provide a broad range of both basic and advanced functionalities, which enable data collection, transformation, analysis, and reporting. Nevertheless, at the same time spreadsheets maintain a friendly and intuitive interface. In addition, they entail a very low cost. Well-known spreadsheet applications, such as OpenOffice [58], LibreOffice [60], Google Sheets [76], and Gnumeric [103], are free to use. Moreover, Microsoft Excel [35] is widely available, with unofficial estimations putting the number of users to 1.2 billion[1]. Thus, spreadsheets are not only powerful tools, but also easily accessible. For these reasons, among others, they have become very popular with novices and professionals alike.

As a result, a large volume of valuable data resides in spreadsheet documents. In industry, internal business knowledge is stored and managed in this format. Eckerson and Sherman estimate that 41% of Spreadmarts (i.e. reporting or analysis systems running on desktop software) are built on top of Microsoft Excel [47]. Moreover, governmental agencies, nonprofit organizations, and other institutions collect and make available data with spreadsheets (e.g., in open data platforms [29]). In science, spreadsheets act as lab books, or even as sophisticated calculators and simulators [107].

Seeing the wide use and the concentration of valuable data in spreadsheets, industry and research have recognized the need for automatic processing of these documents. This need is more evident, at a time when data is considered "the new oil" [4]. Nowadays, the demand for comprehensive and accurate analysis (of data) has increased. New concepts have emerged, such as *big data* and *data lakes* [96, 100]. It has become more and more apparent that being able to integrate and reuse data from different formats and sources can be very beneficial.

From spreadsheets, of particular interest are data coming in tabular form, since they provide concise, factual, and to a large extend structured information. In this regard, databases seem to be one of the natural progressions for spreadsheet data. After all, tables are a fundamental concept for both spreadsheets and databases. However, as noted in the following paragraphs, spreadsheet tables often carry more (implicit) information than database tables. Thus, transferring data from one format to the other is not as straightforward. In fact, there is a need for sophisticated transformations. Nevertheless, by bringing these two worlds closer we can open the door to many applications. This would allow spreadsheets to become a direct source of data for existing or new business processes. It would be easier to digest them into data warehouses, and in general integrate them with other sources. Most importantly, it will prevent information silos, i.e., data and knowledge being isolated and scattered in multiple spreadsheet files.

Besides databases, there are other means to work with spreadsheet data. New paradigms, like NoDB [12], advocate querying directly from raw documents. Going one step further, spreadsheets together with other raw documents can be stored in a sophisticated centralized repository, i.e., a data lake [100]. Yet, this still leaves an open question: how to automatically understand the spreadsheet contents?

In fact, there are considerable challenges to such automatic understanding. After all, spreadsheets are designed primarily for human consumption, and as such, they favor customization and visual comprehension. Data are often intermingled with formatting, formulas, layout artifacts, and textual metadata, which carry domain-specific or even

[1] https://www.windowscentral.com/there-are-now-12-billion-office-users-60-million-office-365-commercial-customers)

user-specific information (i.e., personal preferences). Multiple tables, with different layout and structure, can be found on the same sheet. Most importantly, the structure of the tables is not known, i.e., not explicitly given by the spreadsheet document. Altogether, spreadsheets are better described as partially structured, with a significant degree of implicit information.

In literature, the automatic understanding of spreadsheet data has only been scarcely investigated, often assuming just the same uniform table layout across all spreadsheets. However, due to the manifold possibilities to structure tabular data within a spreadsheet, the assumption of a uniform layout either excludes a substantial number of tables from the extraction process or leads to inaccurate results.

Therefore, in this thesis, we address two fundamental tasks that can lead to accurate information extraction from spreadsheets. Namely, we propose intuitive and effective approaches for layout analysis and table detection in spreadsheets. One of our main goals is to eliminate most of the assumptions from related work. Instead, we target highly diverse sheet layouts, with one or multiple tables. Nevertheless, we also foresee the presence of textual metadata and other non-tabular data in the sheet. Furthermore, we make use of sophisticated machine learning and optimization techniques. This brings flexibility to our approach, allowing it to work even with complex or problematic tables (e.g., containing empty cells and missing values). Moreover, the intended flexibility makes our approaches transferable to new spreadsheet datasets. Thus, we are not bounded to specific domains or settings.

1.2 CONTRIBUTIONS

This thesis aims at automatic processing methods for spreadsheets, based on the insight that data stored in these documents could be transformed in other more structured forms. Therefore, we propose a processing pipeline for spreadsheet documents. The input sheet goes through a series of steps that gradually infer its structure, and then expose it for further processing. Below, we provide a detailed list of our contributions:

1. We study the history of spreadsheet documents and review a broad body of literature from research on these and other similar documents. Nevertheless, the main focus is on existing approaches in layout analysis and table detection. In particular, we consider works from the Document Analysis and Recognition (DAR) field. We bring well-established concepts and approaches from this field to spreadsheets. (Chapter 2)

2. We perform a large scale analysis of real-world spreadsheets. We put into test claims from related work and discover challenges that were so far overlooked. For this analysis, we consider spreadsheets from both the Web and business domain. The results are visualized and thoroughly discussed in the form of takeaways. Besides the common characteristics, we highlight the differences between the two domains, Web and business. (Chapter 3)

3. Due to the lack of publicly available benchmarks, we develop an annotation tool, which is used to create two datasets. The selection of the files, pre-processing, and annotation procedure are described in a comprehensive manner. The tool and the datasets are made publicly available. (Chapter 3)

4. We propose a machine learning approach for layout analysis in spreadsheet documents. To the best of our knowledge, we are the first to attempt this at the cell level. This approach allows us to capture much more diverse layouts than related work. A large portion of the features implemented for classification are original to this work. We prove that these features are among the most relevant during classification. (Chapter 4). Last, we discuss two methods that correct misclassifications, by studying the immediate and distant neighborhood of the cell (Chapter 5)

5. We propose a formal model to represent the layout of a sheet, after cell classification. This includes a well-defined and motivated procedure for the creation of layout (uniform) regions. (Chapter 4) Moreover, we introduce a graph model that encodes precisely the characteristics of the regions, and their spatial arrangement in the sheet. (Chapter 6)

6. Our work includes two novel and effective table detection approaches. We formulate the detection task as a graph partitioning problem. Besides rules and heuristics, we incorporate genetic algorithms and optimization techniques. For this purpose, we define an objective function that quantifies the merit of a candidate table, with the help of ten specialized metrics. Moreover, these functions can be tuned to match the characteristics of new (unseen) datasets. To the best of our knowledge, there is no other work in literature incorporating such methods for table detection, both in spreadsheets and other similar documents. (Chapter 6)

7. We develop a research prototype (Excel add-in) that allows us to test and improve the aforementioned method. Currently, this prototype provides some support for information extraction. (Chapter 7)

1.3 OUTLINE

The structure of this thesis is visualized in Figure 1.1. In Chapter 2, we discuss fundamental aspects of spreadsheet documents: interface, functionalities, and file format (i.e.,

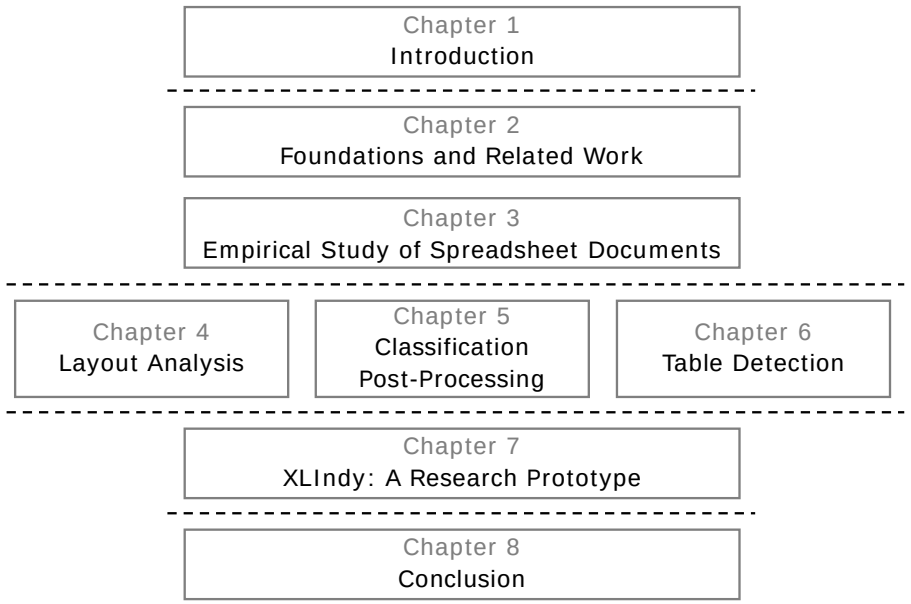


Figure 1.1: Organization of the chapters in this thesis.

how data is encoded). Additionally, we review related works from spreadsheets and the broad area of Document Analysis and Recognition. In Chapter 3, we outline the methods, tools, and results from our empirical analysis of real-world spreadsheet documents. Based on this analysis we derive open challenges (requirements) and define the objectives and scope of this thesis (Section 3.4.2). The next three chapters discuss specific parts of our proposed processing pipeline for layout analysis and table detection in spreadsheets. Specifically, Chapter 4 outlines how we infer the layout of the sheet via cell classification. Chapter 5 discusses an optional step of the pipeline, which attempts to eliminate cell misclassifications prior to table detection. Chapter 6 summarizes our actual contributions with regard to detecting tables in spreadsheets. In fact, we propose multiple approaches for this task. Next, in Chapter 7, we present XLIndy, an Excel add-in that implements the proposed processing pipeline. This tool not only allows us to run the proposed approaches, but also visualize the results, review them, and test different settings. We conclude this thesis in Chapter 8, where we summarize our contributions and lay the ground for future work.

The chapters of this thesis map to our published papers in the following way: Chapter 3 encompasses one of our recent publications from ICDAR'19, which concerns the annotated dataset of spreadsheets [84]. Chapter 4 is based on our publication from KDIR'16 [85]. Chapter 5 incorporates the work originally discussed in the CCIS book chapter [87] and part of the work from the KDIR'16 paper [85]. Next, Chapter 6 is based on three of our publications: CAiSE'17 [83], DAS'18 [86], ICDAR'19 [88]. Finally, our DocEng'19 publication [82] is discussed in Chapter 7.

FOUNDATIONS AND RELATED WORK

2.1 The Evolution of Spreadsheet Documents

2.2 Analysis and Recognition in Electronic Documents

2.3 Spreadsheet Research Areas

In this chapter, we discuss the fundamental concepts and works that are relevant to this thesis. We begin with Section 2.1, where we outline the evolution of spreadsheet documents and highlight their unique technical characteristics. Subsequently, in Section 2.2, we review literature for layout inference and table recognition approaches, especially within the field of Document Analysis and Recognition. Finally, in Section 2.3, we cover actual research in spreadsheets. Besides, layout inference and table recognition, we discuss topics such as formula debugging, database-spreadsheet unification, spreadsheet modeling, etc. Although some of these works do not share the same scope with this thesis, they share similar challenges. Therefore, their findings and proposed approaches are relevant.

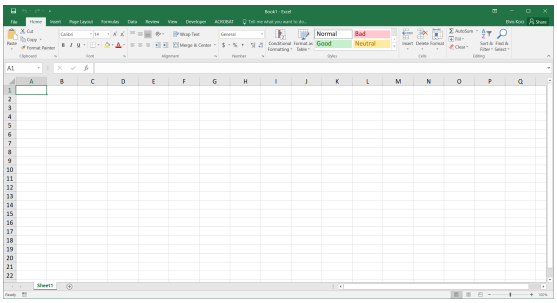
2.1 THE EVOLUTION OF SPREADSHEET DOCUMENTS

Spreadsheets can be simply described as electronic counterparts of paper-based accounting worksheets. It is believed that the latter originate from the 15th century, initially proposed by Italian mathematician Luca Pacioli, often referred to as the father of bookkeeping [61, 112]. However, the idea of organizing data into rows and columns has been around for several Millennia (see Plimpton 322, a Babylonian tablet from 1800BC [28, 62]).

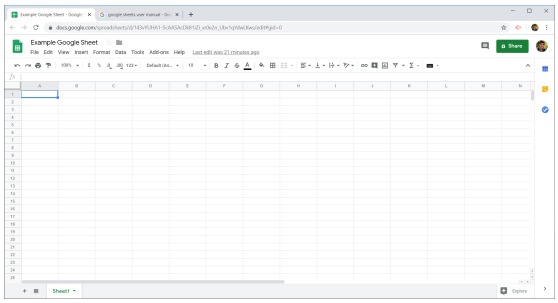
Modern electronic spreadsheets brought this ancient but natural way of organizing data into new heights. The two-dimensional grid of rows and columns was enhanced with an abundance of functionalities [69, 115, 137] and the inherent flexibility of the electronic format. In this easy-to-use and highly expressive environment, users have become informal designers and programmers, with the ability to shape data according to their needs. Consequently, modern spreadsheets have become an essential tool for many companies, supporting a broad range of internal tasks.

In 1979, the first commercially successful spreadsheet software, VisiCalc, was developed by Dan Bricklin and Bob Frankston [25, 26, 120]. Initially, it was released for Apple 2 computer. A version for MS-DOS on the IBM PC followed soon after, in 1981. The success of VisiCalc inspired the development of other similar software, notably Lotus 1-2-3 [111], SuperCalc [128], and Multiplan [127]. In 1983, soon after its release, Lotus 1-2-3 overtook the market. To this contributed its ability to handle larger spreadsheets, while simultaneously being faster than its competitors [69]. However, in the early 90s, Microsoft Excel [35] became the market leader, a position that it has maintained ever since. Excel offered additional functionalities (especially with respect to formatting), improved usability, and faster recalculations [69]. Nowadays, besides Microsoft Excel, we find open-source spreadsheet software, such as Gnumeric [103] and LibreOffice [60]. The market has also introduced web-based collaborative spreadsheet programs. Google Sheets [76] is the most successful example of such an application. In the last years, Microsoft is developing a similar functionality within its Office 365 suite [34, 36].

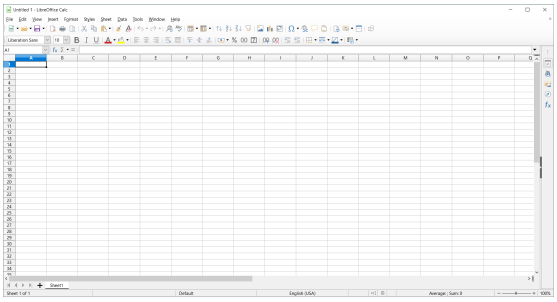
For an extended view on the history of spreadsheets, refer to the book "Spreadsheet Implementation Technology" [115], Felliene Hermans' dissertation [69], the related article [137] in ACM Interactions magazine, and survey papers [5, 22].



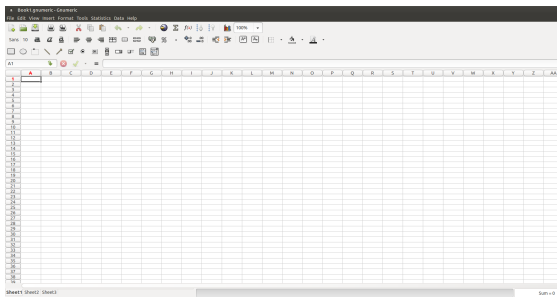
(a) Microsoft Excel 2016



(b) Google Sheets



(c) LibreOffice v6.3



(d) Gnumeric v1.12

Figure 2.1: User Interfaces for Different Spreadsheet Vendors

2.1.1 Spreadsheet User Interface and Functionalities

Modern spreadsheet applications share some essential characteristics, although they target slightly different market segments. Here, we discuss the user experience (interface and functionalities) and define some basic spreadsheet concepts. More information can be found online in the respective user manuals and help pages: Microsoft Excel [35], LibreOffice [59], Google Sheet [76], and Gnumeric [104].

In the main window, the user interacts with a menu bar, which provides access to the functionalities of the spreadsheet application (see Figure 2.1). Below this bar, the user finds a two-dimensional grid, referred to as *sheet* or *worksheet*. A spreadsheet file (also known as *workbook*) can contain one or many related sheets. The basic unit of every such sheet is the *cell*, i.e., the intersection between a *column* and a *row*. Cells can be empty or contain various types of data: *string*, *numeric*, *boolean*, *date*, and *formula*.

Notably, spreadsheets provide an ample number of build-in formulas for arithmetic calculations, statistical analysis, operations with string and dates, and various other utilities. With such formulas, users can reduce large problems into a series of simple computational steps. Furthermore, in an interactive fashion, users can alter the input values (i.e, cell contents) and spreadsheets will *recalculate* on the fly the new output [115]. This enables *"what-if"* analysis, which is regarded as one of the most useful features of spreadsheets.

Clearly, for the aforementioned operations, formulas need to *reference* the contents of other cells or even *ranges* of cells (i.e., a rectangular area of the sheet). In spreadsheets, the most common referencing system is the *A1-style*, which can be seen in Figure 2.1. Respectively from left to right and top to bottom, the columns are labeled with letters (*A*, *B*, *C*, ..., *Z*, *AA*, *AB*, ...) and the rows with numbers (*1*, *2*, *3*,...). Some example references are *B3*, *Z18*, and *C10:F12*. Note, the last one is a range of cells.

Besides formulas and interactive recalculations, spreadsheets provide other useful functionalities. Users can *filter*, *sort*, *find/replace*, and *rearrange* data easily. Built-in *charts* and *diagrams* enable data visualization and analysis. Many and various formatting options allow users to personalize the way data is displayed. For instance, users can change the font color and font size of a (cell) value, the borders and alignment of the cell itself, and up to the column widths and row heights. Another related feature is *conditional formatting*, which allows applying formats on multiple cells in one action, based on predefined or user-defined conditions. In this way, one can quickly format and most importantly analyze visually large amounts of data.

Overall, spreadsheets are intuitive, expressive, flexible, and powerful. Naturally, these characteristics made spreadsheets popular with novices and professionals alike. Therefore, nowadays they are used in many different domains and settings.

2.1.2 Spreadsheet File Formats

In spreadsheets, user-generated content such as values, formatting, and settings, are encoded by the application in a specific format. In fact, vendors have developed their own file format [89], which allows them to efficiently write/read spreadsheet contents. Below, we outline the history of file formats for office applications (including spreadsheets), based on [89].

During the 1990s, the vast majority of spreadsheet vendors used proprietary binary file formats. This made it difficult for third-parties to develop their own custom applications on top of spreadsheets. It also hindered interoperability between spreadsheet applications.

Starting from the late 1990s, there have been attempts to create an XML-based open standard for spreadsheets, and office documents in general. In 2006, the *Open Document Format for Office Applications (ODF)* [99], was accepted as an ISO and IEC standard. Nowadays, ODF is native to LibreOffice and OpenOffice, while being supported by all the other major vendors. Microsoft independently developed an alternative format, called *Office Open XML (OOXML)* [77], which was approved as an ISO/IEC standard in 2008. Commercially, it was introduced with the release of Microsoft Office 2007. All Excel documents created with this or newer versions have a *.xlsx* file extension. Prior to this, Excel documents had the extension *.xls* (i.e., a binary-based format). Nevertheless, both .xls and .xlsx formats are currently supported by other spreadsheet and enterprise applications.

ODF and OOXML have many characteristics in common. Most importantly, the file formats are zip archives that typically contain multiple XML files. Each such file is specialized to store specific aspects of the user-generated content (e.g., values, styling, and settings). Nevertheless, an XML file can also reference other files within the zip archive. The archive itself is organized in a hierarchical way, grouping files into folders and subfolders. Besides the XML files, the achieve will contain other media, such as images, if the user had previously inserted them in the spreadsheet document. In order to recover (decode) the original document, a spreadsheet application needs to parse the XML files (considering the dependencies between them) and subsequently load any linked media.

Overall, due to these open standards, machine processing of spreadsheets has become easier and computationally efficient. In fact, most of the popular programming languages already have specialized libraries to work with ODF and OOXML formats [57, 64, 121]. Equipped with these tools and the detailed technical documentation of the standards, one can easily create custom extensions on top of spreadsheets. This is beneficial not only for businesses but also for the research community who aims at experimenting with innovative techniques on these documents.

2.1.3 Spreadsheets Are Partially-Structured

The introduction of XML-based file formats brought the needed standardization and interoperability. Yet, vendors have not accounted for a formal and systematic way to capture the layout (physical and logical) of the whole sheet and that of the individual tables. Therefore, spreadsheets are still in the realm of partially-structured documents.

There are ways for the users to label indirectly parts of the sheet. However, this makes any sophisticated analysis of spreadsheets highly dependent on user input. For example, Microsoft Excel provides the option to create *pivot tables* [35]. In such cases, the *.xlsx* document (i.e., a zip archive) will contain designated XML files that record the physical structure of these tables: column index, row index, and value areas. Similarly, ODF treats pivot tables in a specific way within its XML-based format. Another instance of indirect labeling is the use of build-in styles [35, 59]. These are intended for special cells (e.g., notes, headings, calculations, etc.) or even entire tables. Again, any usage of such styles will be encoded into the saved spreadsheet file. Thus, users can definitely provide hints that would be valuable to any algorithm attempting automatic spreadsheet analysis. However, in reality, users do not commonly employ the aforementioned options (as is shown in Section 3.3 and in [78]). On the contrary, they often apply custom formatting, as it feels more personalized and/or suitable for the occasion. At times, they might not apply formatting at all.

In fact, not only formatting but also the arrangement of contents is under the full control of the user. This often results in complex sheets, which do not necessarily follow a predefined model or template. Such sheets might be difficult to understand even by humans (see Section 3.2.2), let alone machines. Thus, automatic spreadsheet processing is not a trivial task. Spreadsheets face similar challenges to other well-studied documents in literature (refer to Section 2.2). As a result, specialized and advanced algorithmic steps are required to automatically handle arbitrary spreadsheets.

2.2 ANALYSIS AND RECOGNITION IN ELECTRONIC DOCUMENTS

Document Analysis and Recognition (DAR) is an established field, with well-defined tasks. Below we summarize this field's contributions, based on three surveys [94, 95, 97]. In particular, we discuss DAR tasks that are relevant to this thesis. Namely, we highlight methods for layout analysis and table detection in electronic documents. Later, we adopt these methods for spreadsheet documents.

2.2.1 A General Overview of DAR

The DAR field concerns itself with the automatic extraction of information, from document formats that are primarily designed for human comprehension, into formats that are standardized and machine-readable [95]. Typically, research in this field addresses scanned documents (also known as document images [94, 97]). However, in recent years more diverse document types are considered, such as PDFs, HTML, and digital images [95].

DAR techniques are applied to a variety of tasks, most commonly found in business settings. A well-known application is the automatic sorting of mail, where machines recognize and process the address section on the envelopes [95, 97]. Another application is the

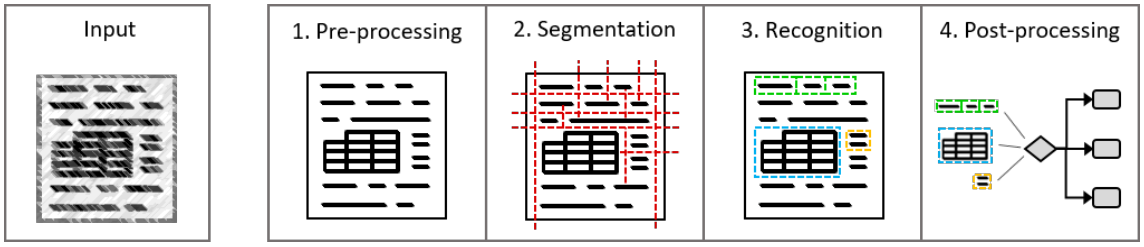


Figure 2.2: The Document Analysis and Recognition Process

automatic processing of business documents, such as invoices, checks, and forms [95]. DAR techniques are also used for research purposes, like the analysis of old manuscripts and ancient artifacts, found in digital libraries [95].

Marinai [95] identifies four principal steps in a DAR system: pre-processing, object segmentation, object recognition, and post-processing. We illustrate them in Figure 2.2. *Pre-processing* concerns techniques that bring input documents into formats that are more suitable to work with. For instance, with respect to images, one could apply binarization and noise reduction techniques, in order to improve their quality. *Object segmentation* aims at dividing the document into smaller homogeneous regions. As stated by Marinai, this task can be performed at different granularities. Some systems are concerned with segmentation at the character or word level. Others seek larger regions, such as text blocks and figures. The third step, *object recognition*, attempts to categorize the resulting regions. Specifically, it assigns logical or functional labels, such as title, caption, footnote, etc. The last step, *post-processing*, decides the next recommended actions based on the recognition results. The available options depend on the design and purpose of the specific DAR system. Overall, these four steps provide a high-level view of a DAR system. However, in reality, such systems are often composed of multiple processes, each one engaging to some extent with the aforementioned steps.

Layout Analysis

Layout analysis[1] is one of the essential processes in DAR [94, 95, 97]. It aims at discovering the overall structure of the document or discovering specific regions of interest. In particular, layout analysis is associated with the detection and recognition of larger document components [95], such as text blocks, figures, and tables. Processes operating at a lower level (e.g., Optical Character Recognition) often precede and provide input to layout analysis. Literature differentiates between *physical* and *logical* analysis of the layout. The former addresses geometry and spatial arrangement, while the latter is concerned with function and meaning. In other terms, physical analysis falls under the object segmentation step, and logical analysis under the object recognition step. Typically, the output of layout analysis is a tree structure, which describes the hierarchical organization of the layout regions [94, 97]. However, for some applications, attribute graphs are more suitable as they are better at encapsulating the properties of the individual regions and the relationships between them [97].

There are three approaches to layout analysis: bottom-up, top-down, and hybrid [94, 97]. In the *bottom-up* approach, smaller components are iteratively merged to form larger elaborate structures. Contrary, the *top-down* approach will start from the whole document and iteratively segment it into smaller components. The *hybrid* approach combines the first two, in an attempt to get faster and better results. According to Marinai [95], the bottom-up approach is more effective when little is known about the document structure. If there is pre-existing knowledge, the top-down approach should be used instead.

[1]This is a widely accepted term within the DAR community. However, in this thesis, we make often use of the term "layout inference", following precedent from previous works in spreadsheet documents.

Table Detection and Recognition

The *detection* and *recognition* of tables are often seen as sub-tasks of layout analysis [94, 95]. The detection task identifies regions that correspond to distinct tables. While the recognition task goes deeper into the analysis of the table structure. Specifically, it recognizes the individual components that make up a detected table, i.e., performs a logical analysis. Some works add an extra step to recognition, which aims at finding the relationship between the table components. This completes the automatic *understanding* of the table and subsequently facilitates accurate information extraction.

Intuitively, one needs to answer "what is a table?", before even starting any detection and recognition process. In literature, there have been multiple attempts to formally define tables, discussed by Embley et al. [49]. The definition varies depending on the domain and application. Therefore, instead of a clear cut definition, here we discuss the Wang model [126], which is widely accepted within the DAR community [37, 75, 134].

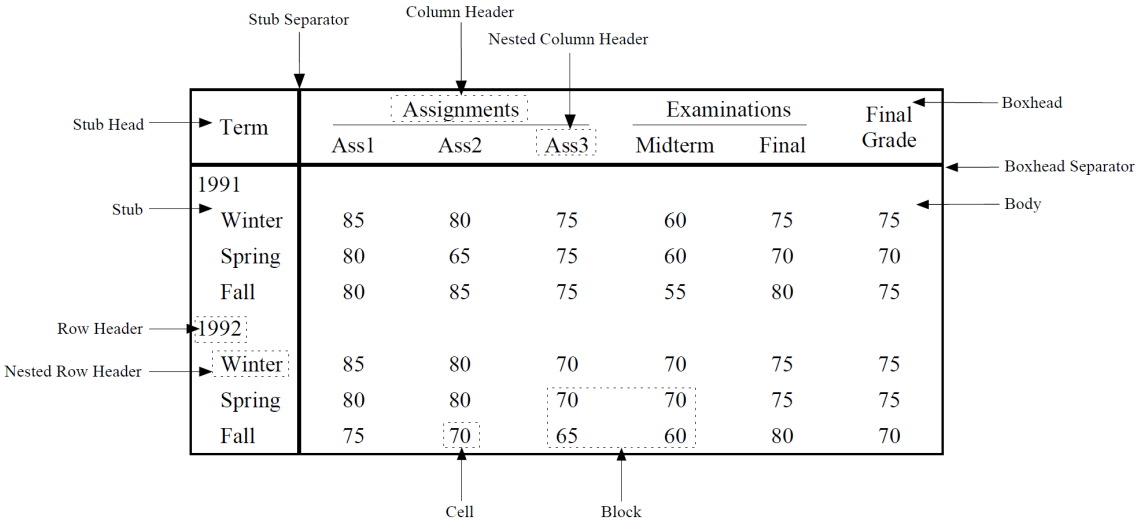


Figure 2.3: The Wang Model [134]

Figure 2.3 displays the model proposed by Wang, together with a few additions from Zanibbi et al. [134]. This model provides terminology to describe the general table anatomy. It identifies physical components: *Cell*, *Row*, *Column*, *Block*, and *Separator*. Moreover, it names logical components: *Stub Head*, *Boxhead* (Column Headers), *Stub* (Row Headers), and *Body*. However, as stated by Zanibbi et al., the model omits titles, footnotes, comments, and other text regions that often surround tables.

Nevertheless, Wang describes a non-trivial table structure. It is characterized by nested headers (i.e., hierarchies), on the top (Boxhead) and the left (Stub). In fact, these can be seen as composite indices that point to values in the Body. In addition, the table in Figure 2.3, has an evident use of spaces (i.e., visual artifacts). These spaces are there to make the top/left hierarchies even more apparent. As well as, they divide the table into three logical sections (i.e., Assignments, Examinations, and Final Grade). Clearly, tables like this one are designed for human consumption. Therefore, they come in a compact form and carry visual clues.

This differs substantially from tables found in relational databases [93]. There, the recipients are not only users but also other applications. Moreover, relational databases are concerned with issues such as efficient execution, concurrent access, and data integrity.

Thus, these databases require tables to be in a canonical form, i.e., following principles outlined by a formal mathematical model (i.e., the relational model).

However, bringing arbitrary tables (like the one in Figure 2.3) into a canonical form requires substantial reasoning capabilities. According to [75], with regards to table recognition, there are disagreements even between human "experts". Without proper knowledge and context, one can misinterpret or overinterpret the structure of the table and its contents. Therefore, table recognition is often regarded as one of the most difficult tasks in (automatic) layout analysis [95, 49].

2.2.2 DAR in Spreadsheets

Despite the differences between electronic documents, when it comes to analysis and recognition, the fundamental principles remain the same. Thus, research on other documents is still very relevant to this thesis. Nevertheless, one has to specialize the four DAR steps (refer to Section 2.2.1), when working with spreadsheet documents. For example, unlike scanned documents, there is not much need for noise removal in spreadsheets. Instead, most of the pre-processing effort goes towards the collection of information (styling features, textual features, etc.) relevant for layout analysis [11, 29, 85].

Like other documents, spreadsheets can be described by means of both physical and logical layout. In this case, the type of certain constructs is known, since it is already encoded into the document format (see Section 2.1.2). Most notably, spreadsheet applications treat figures, charts, and shapes separately from cell contents. Yet, the seemingly simple two-dimensional grid of cells can yield a variety of elaborate structures (as pointed out in Section 2.1.1). In order to extract information from spreadsheets, we still need to identify regions (i.e., cell ranges) of interest, such as titles, footnotes, comments, calculations, and tables. Moreover, any automatic analysis must also discover the relationships between these regions, to enable accurate interpretation of the contents.

In particular, as is experimentally shown in Section 3.3, spreadsheet tables are often complex. Intuitively, top/left hierarchies, like the one in Figure 2.3, can be easily constructed in spreadsheets. In addition, users can apply many different styling options (refer to Section 2.1.1) and make use of spaces (empty cells, rows, and columns). Therefore, any automatic approach needs to handle a big variety of user-generated content, which makes analysis and recognition in spreadsheets rather challenging.

At the end of Chapter 3, we outline the approach proposed by this thesis, based on a thorough analysis of real-world spreadsheets. Specifically, we propose a processing pipeline that follows the bottom-up paradigm. It starts with the smallest unit of a sheet, i.e., the cell, and gradually builds up to tables and other coherent layout regions.

2.3 SPREADSHEET RESEARCH AREAS

Research in spreadsheet documents has focused on different topics. We first visit works that are the most relevant for this thesis, discussing layout inference, table detection, table recognition, and information extraction in spreadsheets (refer to Section 2.3.1). We examine the technical characteristic of these works and make a preliminary assessment. Subsequently, in Sections 2.3.2 - 2.3.4, we outline a broad spectrum of research works that are to some extent relevant. They cover topics such as database-spreadsheet unification, software engineering practices in spreadsheets, and efficient data wrangling methods. Nonetheless, we draw parallels with these works and study how they have approached similar challenges in spreadsheet documents.

2.3.1 Layout Inference and Table Recognition

In the last decade, the number of works tackling layout inference, table detection, table recognition, and information extraction in spreadsheets has significantly increased. These works utilize different methods: machine learning, domain-specific languages, rules, and heuristics. Below we visit specific works for each one of these methods. Note, we cover these works again in the subsequent chapters, where we discuss specific aspects in more detail while comparing with the proposed approach.

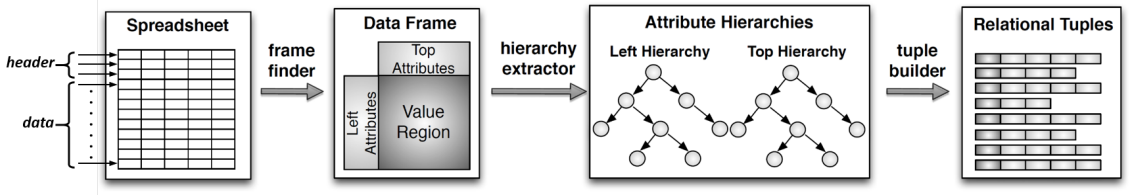


Figure 2.4: Chen et al.: Automatic Web Spreadsheet Data Extraction [29]

Chen et al. [29, 30, 31, 32] worked on the automatic extraction of relational data from spreadsheets. They focus on a specific construct they call *data frame*[2]. These are regions in a sheet that have attributes on the top rows and/or left columns (i.e., roughly corresponding to row/column headers in the Wang model [134]). The remaining cells of the region hold numeric values (see Figure 2.4). In particular, the authors are interested in data frames containing hierarchies (i.e., nested attributes on the left or top). Overall, they use machine learning techniques and few heuristics (i) to recognize the layout of the sheet, (ii) find the data frames, (iii) extract hierarchies from attributes, and (iv) build relational tuples. The overall process is illustrated in Figure 2.4. For the first step, the authors go from the top row to the last row and assign to each one a label: *Title*, *Header*, *Data*, or *Footnote* [29]. They make use of Conditional Random Field (CRF) classifiers, which are suitable for sequential classification. CRFs take into account additional features from previous elements (rows) in the sequence, to predict the class of the current element (row). The second step, data frame detection, is performed based on rules [29]. The presence of a *Header* indicates the start of a new data frame, unless the previous row was as well a Header. Subsequently, columns with strings, located on the left side of the data frame, are marked as left attributes. While, Header rows, on the top of the data frame, constitute the top attributes. Chen et al. have proposed multiple approaches to extract the attribute hierarchies, i.e., the third step. In [29] the authors generate parent-child candidates and then collect features to predict the true pairs via classification (SVM and EN-SVM). In a follow-up work [30], Chen et al. used probabilistic graphical models to encode the potential of individual parent-child candidates and the correlations between them. Moreover, the authors have developed a tool for interactive user repair. This enables continuous learning, as user feedback is incorporated back into the probabilistic graphical model. Furthermore, for the last step (i.e., relational tuple builder) the authors propose two approaches. The first one relies entirely on the inferred hierarchy. For each (numeric) value, the system maps the attributes from the top hierarchy and then from the left hierarchy to build a relational tuple [29]. Nonetheless, in their latest publication [32], Chen et al. attempt to address more complex scenarios. Their method couples machine learning with active learning to identify properties of data frames, such as aggregation columns, aggregation rows, split tables (i.e., multiple tables under the same header), and others. The authors hope to perform more accurate data extraction, once additional properties are known about the data frame. Finally, in [31], Chen et al. present Senbazuru, a prototype system showcasing most of the above-mentioned steps and methods. Additionally, Senbazuru supports *select* and *join* queries on a collection of Web-crawled spreadsheets.

[2]The authors are reluctant to name these constructs as tables.

Adelfio and Samet have worked on schema extraction from Web tabular data, considering both spreadsheets and HTML tables [11]. Similar to Chen et al., the authors start by assigning a label (function) to rows, using supervised machine learning. They experimented with various algorithms, and Conditional Random Field proved to be the most accurate. However, Adelfio and Samet go one step further, by coupling CRF with a novel way to encode the individual cell features into row features. This encoding, called *logarithmic binning*, ensures that rows having roughly the same cell features and length will end up in the same bin (i.e., grouped closely together). Ultimately, logarithmic binning enables the creation of more accurate classification models, as it becomes easier to discriminate among the training instances (i.e., annotated rows). Besides encoding, the authors differ from Chen et al. with regards to the processing steps. Adelfio and Samet capture hierarchies and aggregations during layout inference, instead of introducing designated steps later in the process (e.g., hierarchy extraction [29, 30] and property detection [32]). Therefore, they have defined specialized row labels, bringing them to seven in total. Nonetheless, after layout inference, the table detection step is performed in a similar fashion to Chen et al., i.e., using Header rows as delimiters. This concludes the overall process, proposed by Adelfio and Samet. The structure of the detected tables is described by the enclosed rows, which were previously classified. According to the authors, this can already facilitate schema and information extraction from the detected tables.

Recently, the Spreadsheets Intelligence group, part of Microsoft Research Asia, published a paper focused entirely on the task of table detection in spreadsheets [43]. The architecture of the proposed framework, *TableSense*, has three principal modules: a cell featurization, a Convolutional Neural Network (CNN), and a table boundary detection module. The cell featurization module extracts 20 predefined features from each considered cell. Subsequently, the input to the CNN module is a $h \times w \times 20$ tensor. In other terms, the framework operates on a $h \times w$ matrix of cells (i.e. the input sheet), and 20 channels, one per extracted feature. The CNN module then learns how to create a high-level representation of the matrix and additionally to capture spatial correlations. The output of CNN is fed to the boundary detection module, which starts with candidate table regions, progressively refines them, and eventually outputs those for which it has high confidence. The authors postpone any further analysis on the detected tables for future work.

Shigarov et al. [117, 118, 119] have focused on the task of table understanding (recognition and interpretation), under the assumption that the location of the table is given. The authors introduce their own table model, i.e., naming the distinct table components [118]. Nevertheless, their main contribution is a domain-specific language, which is referred to as CRL (Cells Rule Language). As the name indicates, CRL operates directly at the cell level. The defined rules "map explicit features (layout, style, and text of cells) of an arbitrary table into its implicit semantics (entries, labels, and categories)" [118]. Furthermore, the authors have developed TabbyXL [117], a tool that loads the defined rules and subsequently uses them to bring tables into canonical (relational) form. Their evaluation shows that the tool performs well on a domain-specific dataset [118]. However, we note that the authors themselves defined the rules used for this evaluation. There is no report of experiments with other users, who might have different familiarity with the given domain, the relational model, and the rule language itself.

The paper [46] introduces DeExcelerator, a framework which takes as input partially structured documents, including spreadsheets, and automatically transforms them into first normal form relations. The authors go beyond the standard DAR tasks (see Section 2.2.1). Besides table detection and recognition, they address data quality and structural issues: value extrapolation, (data) type recognition, and removal of layout elements (i.e., distortions introduced by user formatting). Their approach works based on a set of rules, which resulted from an empirical study on real-world examples. These rules have a predefined order and can apply to individual cells, rows, or columns. For instance, "the start

of a numeric sequence in one column signals the start of the data segment [and the end of header segment]". In comparison to Shigarov et al., DeExcelerator operates on hard-coded rules. Thus any change requires modification of the existing implementation.

In summary, the above mention works differ not only in the employed methods, but also with regards to the scope, underlined assumptions, and processing steps. Most notably, there is no universal table model within this research community. Furthermore, related works have a different understanding of the overall sheet layout (i.e., logical components). In addition, each work introduces its own evaluation dataset (refer to Section 3.1). These datasets differ substantially in size, composition, and annotation methodology. In Chapter 3 we address this ambiguity, by performing a thorough analysis on a large collection of real-world spreadsheets, originating from both business and Web settings. We test the various assumptions and claims from related works. As well as, we identify challenges that are so far overlooked. Subsequently, based on the results from this analysis, we define the scope of the thesis and outline the proposed solution (in Section 3.4).

2.3.2 Unifying Databases and Spreadsheets

Cunha et al. [40] present their approach for bidirectional data transformation from spreadsheets to relational databases and back. They are able to construct a normalized relational database schema, by discovering the functional dependencies in spreadsheet tabular data. Subsequently, they define a set of data refinement rules that guide the transformation process. The process can be reversed, by reusing the same rules one can bring the database tables back to the original spreadsheet form. Note that the authors assume the location of the table to be known. Moreover, they presume that the top row of the table contains attributes and the remaining ones contain data records. Namely, their approach works on database-like tables, which do not exhibit complex hierarchies or irregular structures.

Bendre et al. [18, 19, 20] aim at a system that holistically unifies spreadsheets with relational databases. Note, the authors are not interested in bringing data into a normalized (canonical) form, but rather enabling spreadsheets to work with massive amounts of data. In other terms, instead of XML-formats (see Section 2.1.2), Bendre et al. propose to store spreadsheet data on a backend database. Nevertheless, at the same time, the system retains the typical spreadsheet user interface and functionalities. As outlined by Bendre et al., there are several challenges to be considered. Notably, the underlying database needs to record not only the values but also their current position (i.e., the cell address) in the sheet. Storing the positions is a necessity, but also brings considerable overhead. For example, the insertion or deletion of a row/column can be very costly, since it might trigger cascading updates in the remaining records in the database. Therefore, the authors propose a positional mapping, which instead of the real row/column numbers it uses proxy keys. Any insertion or deletion will result only in an update of the mappings (proxy keys to records). Additionally, the authors discuss efficient schemes for storing the actual spreadsheet data. Each one of these schemes has advantages and disadvantages, depending on the orientation (mostly columns or mostly rows) and sparsity of data. To find a good representation, Bendre et al. proposes a cost model, which takes into consideration storage size, and execution time for fetch (select) and update operations. These model and other features of the system, together with the complexity of different operations, the overall architecture, and experimental evaluation, are described in detail in [19]. The tool is currently publicly available for download in [17].

There are additional works attempting to unify spreadsheets with relational databases. Witkowski et al. attempt to bring spreadsheet flavor into relational databases [129, 130].

They propose SQL extensions to support calculations over rows and columns in a similar fashion to how users apply formulas in spreadsheets. In [131], the same authors present a tool with an Excel interface and a database engine at the backend. Users fetch data from the database but define the calculations (formulas) via the interface. The system translates these calculations into SQL and subsequently stores them as views in the database. These views can be loaded back to Excel or can be used by other applications sharing access to this database. However, at the moment, their approach supports only a limited subset of Excel formulas, which have correspondents in SQL. Liu and Jagadish propose a similar system, with a spreadsheet-like interface, for a step-by-step manipulation of data stored in a database [92]. Their system targets non-technical users. Thus, they put a strong emphasis on usability. Nevertheless, their main contribution is a new spreadsheet algebra that can support incremental and interactive analysis, while at the same time it keeps strong ties with SQL and RDBMS. The author of [124] has implemented all relational algebra operators using only Excel formulas. The aim of this work is to prove that standard spreadsheets can indeed act as stand-alone relational database engines.

From the aforementioned, the works of Cunha et al. and Bendre et al. are the closest to this thesis. The former can be adopted and extended to work with more generic tables in spreadsheets. Nevertheless, it would still require significant input from preceding steps, such as table detection and recognition. While from the work of Bendre et al. we single out their analysis of table arrangements and data sparsity (density) in spreadsheets [20].

2.3.3 Spreadsheet Software Engineering

In literature we find researchers viewing spreadsheets as a development tool and spreadsheet users as programmers. As such, software engineering practices apply. Topics like spreadsheet usability, modeling, governance, versioning, and formula debugging are covered by this community.

UCheck [8] detects unit errors in spreadsheets. The tool uses several heuristic-based algorithms to performs spatial (layout) analysis [6]. The results from these algorithms are combined, based on a weighting scheme, with the purpose of categorizing the individual cells. They recognize four types of cells: Header, Footer (aggregations), Core (table data), and Filler (blank cells with formatting). Based on this information, UCheck detects the table boundaries and then uses another set of formal rules to infers units and detect errors associated with them. Nevertheless, the authors limit their approach to database-like tables. Moreover, for cell categorization and table detection, the rules are hard-coded (i.e., fixed). It is not known how these rules perform in the general case, as the evaluation was done on a small dataset of 28 spreadsheets. Regardless, the UCheck approach has been adopted by other works targeting related topics in spreadsheets [7, 73, 74].

There are works that attempt to model or even reverse-engineer from spreadsheets. In [50] the authors motivate the need for object-oriented models to guide the design of appropriate and error-free spreadsheet solutions (which involve tables). They propose ClassSheets, a modeling language for spreadsheets that can be described as an extension of UML class diagrams. ClassSheets models can be transformed into ViTSL templates [9]. Essentially, the ViTSL templates are a collection of formal specifications, that are used to validate user-generated tables, ensuring they comply with the defined models. In [7] the authors discuss the automatic inference (reverse-engineering) of such templates from legacy spreadsheets. Cunha et al. adopt and extend the above-mentioned works in MDSheet [39], a framework for model-driven spreadsheet engineering. In particular, the framework is enriched with methods that keep ClassSheets models and their instances synchronized (co-evolution) [41]. As well as, the framework explores the functional

dependencies in tables while performing automatic extraction of models from existing sheets [38]. Hermans et al. propose an alternative approach that extracts UML class diagrams from spreadsheet tables [73]. Their goal is the elicitation of implicit domain knowledge, for the development of not only better spreadsheet solutions but also better enterprise systems. Concretely, using a formal pattern grammar, the authors define five common (table) patterns in spreadsheets. When a region in the sheet matches a pattern, a parse tree is generated. Subsequently, a class diagram is extracted from this tree, based on formalized transformation rules. Roughly, in this approach, titles become class names, headers are used for class attributes, and formulas are associated with class methods.

Multiple papers discuss research on tools and techniques that decrease the risk of logical formula errors in spreadsheets. As cataloged by [51], such errors have costed companies up to millions of dollars. In [74] the authors have defined metrics to automatically identify formula smells, i.e., formulas that might be difficult to read or error-prone. Their implementation generates a spreadsheet risk map that makes the user aware of the smells and their severity. A tool that supports seven refactoring actions for spreadsheet formulas is presented in [15]. While refactoring, this tool will re-write formulas and if needed introduce new cells or an entire column. It also supports the creation of drop-down menus from textual columns, in order to constrain the accepted values. A collaboration between authors of the two aforementioned papers yielded another tool for formula refactoring, called BumbleBee [70]. This tool allows users to define and execute their own refactoring rules, via a transformation language that is based on spreadsheet formula syntax. Finally, in [114] worksheet contents are decomposed into fragments (regions) with the objective to enable faster and focused debugging of formula cells. The authors make use of genetic algorithms, to identify the optimum fragmentation.

In this thesis, we acknowledge that formulas can give insights about spreadsheet contents. For instance, aggregation formulas, such as SUM and AVERAGE, typically refer to cells located inside tables, rather than outside them. We can explore such hints while performing layout analysis. However, before that, we need to ensure that formulas are not erroneous. In this regard, we can make use of the above-mentioned works to improve the quality of spreadsheets, prior to our analysis. The other way around, the methods proposed by this thesis can serve the aforementioned works. At the moment, most of them make simplified assumptions. However, this thesis and other similar works can provide the actual table structure and the overall sheet layout.

2.3.4 Data Wrangling Approaches

Here, we discuss two tools that enable efficient wrangling (transformations, cleaning, mapping, etc.) on data coming from different sources, including spreadsheet documents. We visit these tools since they adopt an alternative approach to information extraction from partially-structured documents. OpenRefine [125] enables various transformations and addresses data quality issues on the imported dataset/s. Some of these operations are offered via Graphical User Interface (GUI), others have to be defined using OpenRefine's programming language. The tool maintains a history of the performed operations, which can be exported and reused for other similar projects. Moreover, OpenRefine can fetch data from web services and incorporate Freebase for entity resolution. Wrangler [79] offers similar functionalities, but with a strong emphasis on usability. It incorporates techniques from Human-Computer Interaction, such as programming by demonstration, natural language description of operations, visual previews and interactive history viewer. Based on user selection and history, Wrangler will suggest the most relevant transformations. It employs a declarative language that among others includes operators for lookups, joins, complex table reshaping, and semantic role assignment. Experimental evaluation with multiple users shows that Wrangler has the potential to considerably

speed up data transformations. However, the authors admit that users need to have some level of familiarity before starting to use Wrangler efficiently. In contrast to both Wrangler and OpenRefine, we aim at a predefined processing pipeline that can be applied offline to a large corpus of spreadsheets. The users can still provide feedback, but their general involvement should be kept at a minimum.

AN EMPIRICAL STUDY OF SPREADSHEET DOCUMENTS

3.1 Available Corpora

3.2 Creating a Gold Standard Dataset

3.3 Dataset Analysis

3.4 Summary and Discussion

We begin this chapter with a discussion of the available spreadsheet corpora in literature (see Section 3.1). Some of them contain raw spreadsheet files, which are not annotated for a specific task. Other contain spreadsheets annotated for layout analysis and table recognition. However, these annotations are not made publicly available. Therefore, in Section 3.2, we describe our efforts to build a gold standard dataset of spreadsheets, which is used in the subsequent chapters of this thesis for the experimental evaluation. Unlike related work, we provide a detailed summary of our annotation methodology and make our annotations available to the research community [84]. Section 3.3 reports our findings, following a thorough analysis of the resulting annotations. We show that some of the assumptions held by previous works do not hold. In fact, we observe that many challenges are rather overlooked. Based on these findings, we outline the solution proposed by this thesis, in Section 3.4.

3.1 AVAILABLE CORPORA

There are multiple spreadsheet corpora in literature. These have almost entirely focused on Microsoft Excel files, as it is the most popular spreadsheet application. Furthermore, such files are typically crawled from the Web, where a considerable amount of spreadsheets is publicly available; for instance, in open data platforms.

We begin our discussion with three well-known spreadsheet corpora: Euses [52], Enron [72], and Fuse [16]. Euses was created with the help of search engines, issuing queries containing keywords such as “financial" and “inventory", together with file type “.xls”. Overall, it comprises of $4,498$ unique spreadsheets, organized into categories based on the used keywords. The more recent Enron corpus contains $15,770$ spreadsheets, extracted from the Enron email archive [98]. This corpus is unique, for its exclusive view on the use of spreadsheets in business settings. All the files were used internally by the Enron company, from August 2000 to December 2001. Overall, these files relate to one or more of the 130 distinct employees, from the email records. Another recent corpus is Fuse [16], which comprises of $249,376$ unique spreadsheets, extracted from Common Crawl [55]. Each spreadsheet is accompanied by a JSON file, which includes NLP tokens and metrics describing the use of formulas.

Researchers viewing spreadsheets from a software engineering perspective (refer to Section 2.3.3) have already made use of these three corpora in their published works [7, 40, 44, 71, 73]. However, this is not the case for works tackling layout analysis and table recognition. So far, such works have used other evaluation datasets, created independently. These datasets differ substantially in size and composition. Among them, we find annotated datasets, where the layout and tables are made explicit. However, the annotations are not publicly available. All in all, there is a lack of benchmark/s in this research community, which has made a comparison between the proposed approaches rather difficult. Regardless, below we re-visit all relevant works in layout analysis and table recognition (summarized in Section 2.3.1) and discuss the datasets used by them.

Chen et al. [29] extracted $410,554$ Microsoft Excel files from ClueWeb09, a large corpus of crawled Web pages [105]. Most of the extracted files come from open data platforms, hosted by U.S., Japanese, UK, or Canadian governments. The authors sampled randomly 200 files, to perform a survey on Web spreadsheet. Based on the reported results, dataframes (refer to Section 2.3.1) occur in half of the surveyed files. In $1/3$ of the files the authors find hierarchies (i.e., nested headers or nested values). To train and evaluate their (CRF and SVM) classifiers, Chen et al. manually annotated another smaller sample, of 100 files. Rows were labeled as *Title, Header, Data*, or *Footnote*. Furthermore, the authors marked the location of dataframes, and annotated the parent-child pairs in the

detected hierarchies. In subsequent works [30, 32], Chen et al. expanded their original evaluation dataset with more files. In particular, their latest work [32], was evaluated on a dataset of 400 Web spreadsheets, annotated with additional properties such as aggregation rows/columns and split tables (i.e., multiple tables under the same header).

Adelfio and Samet [11] simultaneously deal with tables in spreadsheet files and HTML pages. With regard to spreadsheets, the authors created a dataset of $1,117$ Microsoft Excel files, crawled from the Web. They used search engines to find relevant websites, containing ".xls" files. In their dataset, the *.gov*, *.us*, and *.uk* are the top contributing domains. For training and evaluation, the authors annotated at the row level, similar to Chen et al. However, as illustrated in Figure 3.1, they used seven layout labels: *Header*, *Data*, *Title*, *Group Header*, *Aggregate*, *Non-relational (Notes)*, and *Blank*. Furthermore, the authors differentiate between tables. When a table contains at least one Header and one Data row, they mark it as *relational*. Otherwise, it is annotated as *non-relational*.

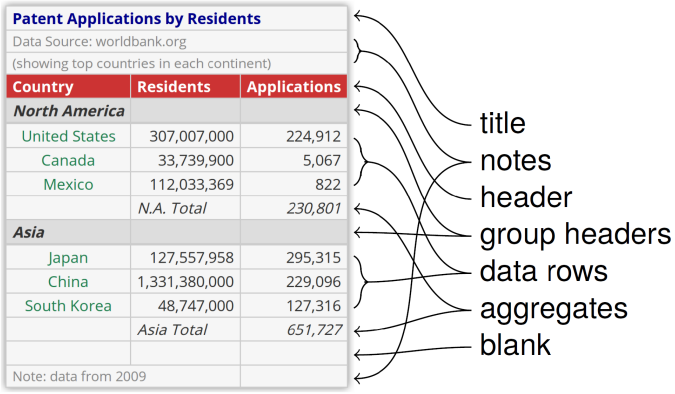


Figure 3.1: Row Labels as Defined by Adelfio and Samet [10]

Like the aforementioned works, Dong et al. (part of Microsoft Research Asia) test their table detection approach on web-crawled spreadsheets [43]. In fact, the authors created two datasets: *WebSheet10K* and *WebSheet400*. Both datasets were hand-labeled by human annotators (judges). Specifically, the judges mark the table regions with the corresponding bounding box. However, it is not known, if the judges annotated the layout of the tables, as well. This is not stated in the paper. Regardless, the *WebSheet10K* dataset, comprising of $10,220$ spreadsheets, was entirely used for training the TableSense framework. While the second dataset, *WebSheet400*, was used for testing the proposed approach. It comprises of 400 distinct sheets, not overlapping with *WebSheet10K*.

Shigarov et al. use a collection of 200 spreadsheet files, which contain statistical data that originate from governmental websites [118]. The authors have not annotated the files per se. Instead, they test the performance of their rule-based approach, by measuring how the output (i.e., extracted data) compares to that of a human expert.

The authors of the DeExcelerator framework [46], evaluate the performance manually per file. Specifically, the assessment was done by a group of 10 database students, on a sample of 50 spreadsheets extracted from *data.gov*. The students score the performance of the framework per file, on various tasks (phases), using a scale from 1 to 5.

3.2 CREATING A GOLD STANDARD DATASET

As discussed in the previous section, in related works the authors have created their own datasets, primarily by crawling spreadsheets from the Web. Nevertheless, we observe that these datasets differ substantially in size. Most importantly, none of the large datasets [11, 32, 43], containing ≥ 400 files, is publicly available. Thus, we can not use them to perform a thorough and wide-reaching study of spreadsheet layouts and tables. Moreover, we can not confirm the claims made by related work, or directly compare the performance of our approach with theirs.

To tackle these challenges, we have created two datasets, annotated for layout analysis and table recognition. The first one is a random sample of $1,160$ files from the Enron corpus [72]. In this way, unlike any of the related works, we address business spreadsheets. In fact, to the best of our knowledge, we are the first to annotate such a large collection of business spreadsheets and perform a thorough analysis on them. Secondly, we annotated a smaller random sample of 406 files from the Fuse corpus (containing web-crawled spreadsheets). In Fuse the *.org* (29.5%), and *.gov* (27.7%) domains are the most common [16]. Thus, the composition of our annotated sample is to some extent similar to the datasets from related works. This means we can indirectly compare the work from this thesis with that from related publications.

In the following sections, we outline the creation of these two datasets. Note, unlike related works, we provide a detailed description of the annotation process and methodology. We discuss the initial selection of files, annotation tools, and annotation phases. The annotated files themselves are made publicly available[1], for future research works.

3.2.1 Initial Selection

Files of the original Enron corpus [72] underwent an initial filtering, after which a considerable number was omitted. The maximum size of the file was limited to 5MB. Additionally, files with macros were filtered out. Moreover, we omitted those having broken external links to other files. Furthermore, we inspected the encoding, keeping only those having character set ANSI (Windows-1252)[2]. This makes it more probable that the selected files have English string values. In addition, we filtered out files that have a similar name (Levenshtein distance ≤ 4) with one of those already selected. This step eliminated the biggest chunk of files but also decreased the chance of having duplicates or near-duplicates. Lastly, some files were eliminated due to exceptions occurring while processing them with Apache POI v3.17 [57]. Overall, the reduced corpus consisted of 5,483 files, from 128 distinct employees. There is a minimum of one file per employee. Yet, we find 15 employees with more than 100 files (with a maximum of 246).

From Fuse we drew a random sample of roughly 1000 files, making sure to balance the occurrence of ".xls" and ".xlsx" extensions. We applied some of the preprocessing filters, used in Enron. With the help of the Apache POI library, we automatically eliminated files that were large, containing macros, external links, and character set other than ANSI. However, the names of the original files in Fuse are hashed. Therefore, we could not apply the Levenshtein distance. In fact, part of the filtering happened during the annotation of the files. As reported in Section 3.3.2, the judges discovered many near-duplicates and non-English files.

[1] https://wwwdb.inf.tu-dresden.de/research-projects/deexcelarator/

[2] The default encoding, for US-based systems

3.2.2 Annotation Methodology

Here, we present the methods and tools that were used to create the gold standard datasets. We start with the definition of two types of annotation labels, one used at the cell level and the other at sheet level. Subsequently, we describe the tool that was developed specifically to assist the judges (annotators). Based on this, we outline the task of annotating a single spreadsheet file. Afterward, we report on the overall annotation process. In particular, we describe the measures we took to ensure consistency (i.e., common understanding of the annotation labels) among the judges.

Cell Labels

We use seven labels for non-empty cells (i.e., filled with a value). We follow closely the labels proposed by Adelfio and Samet [11] since we believe that they are comprehensive and can be mapped to the Wang model [126, 134]. Nevertheless, unlike Adelfio and Samet, we do not annotate at the row level. Our empirical study (refer to Section 3.3.1) shows that cells of the same row can have a different function (label). Therefore, we annotate individual cells or ranges of cells. In this way, we can capture arbitrary sheet layouts. For the same reason, we introduce the label *Other*, which can be used for cells that do not match any of the typical cell functions, proposed by related work. Below, we define each one of the proposed labels. Additionally, we illustrate them in Figure 3.2, with examples.

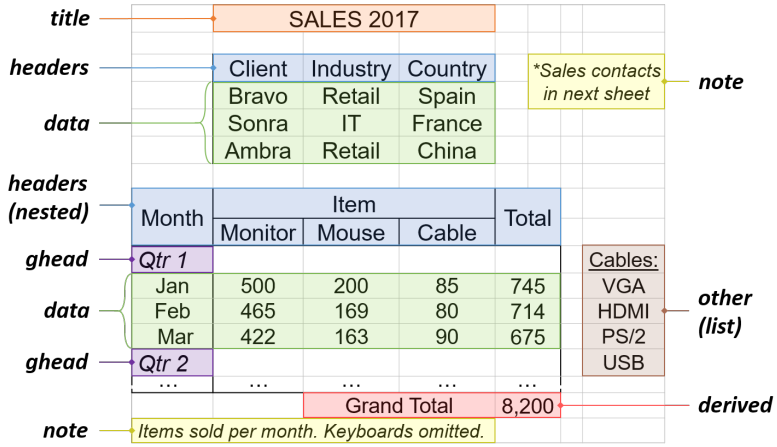


Figure 3.2: Cell Annotation Labels

The basic ingredients for tables are *Headers* and *Data*. In relational terms, Headers correspond to the attributes, while Data represent the collection of tuples (entries). However, in spreadsheets, Headers can be nested occupying several consecutive rows, as shown in Figure 3.2. Nevertheless, even in spreadsheets, Data follow the structure defined by Headers. Likewise, Data cells are the main payload of a spreadsheet table.

Titles and *Notes* provide additional information, effectively improving the understanding of sheet contents. Titles give a name to specific sections (such as a table), or to the sheet as a whole. Notes provide comments and clarifications, which again can apply globally or locally. Typically, Notes take the form of complete or almost complete sentences. On the other hand, Titles can consist of just a single word.

GroupHeaders (also referred to as *GHead*) are reserved for hierarchical structures on the left of a table. In such cases, values in the left column/s are nested, implying parent-child relationships. When such hierarchies are detected, we annotate the parents as GroupHeaders, while the children as Data.

Derived cells are aggregation isles of Data values. Here, we specifically focus on aggregations "interrupting" the Data rows of a table. They act as sums, products, and averages, for the rows above. On the right of these aggregations, we usually find a cell with a string value, which gives a name to the aggregation. Typically, this string contains the word "Total". All in all, we annotate as Derived not only the aggregations but also the accompanying name cell/s. Note, it is important to distinguish Derived cells since they clearly affect (break) the structure of the table. On the contrary, row-wise aggregations, such as sums of multiple Data cells in the same row, have a minimal impact. They tend to follow the structure specified by the Headers, the same as Data cells. Therefore, we annotate row-wise aggregations simply as Data (see Figure 3.2).

The label *Other* is a placeholder for everything else, not fitting to the aforementioned cell labels. Additionally, we use the label Other to annotate ranges of cells (regions) that do not comply with our definition of a table. For instance, occasionally we find regions of "Data" values that are not preceded by a Header row/column. Another example is regions containing key-value pairs. Such pairs can introduce parameters, which are later used for calculations in the sheet. Clearly, the purpose of these key-value regions does not match that of tables.

Finally, a *Table* is annotated with the minimum bounding rectangle (MBR) enclosing all non-empty cells that compose it. In this work, we require that tables contain at least one Header and one Data row, otherwise they are not valid[3]. In the case that the table is transposed, then there must be at least one column each for Header and Data.

All in all, some labels can only be found inside table annotations. GroupHeader is one of them, besides Header and Data. With regards to Derived, they are primarily found in tables. However, when Derived are used to aggregate Data from multiple tables of the sheet, we leave them outside. Moreover, Titles and Notes can refer to the whole sheet. Thus, they are not necessarily an integral part of a single table. Lastly, being a versatile label, Other can be found both in and outside of table borders.

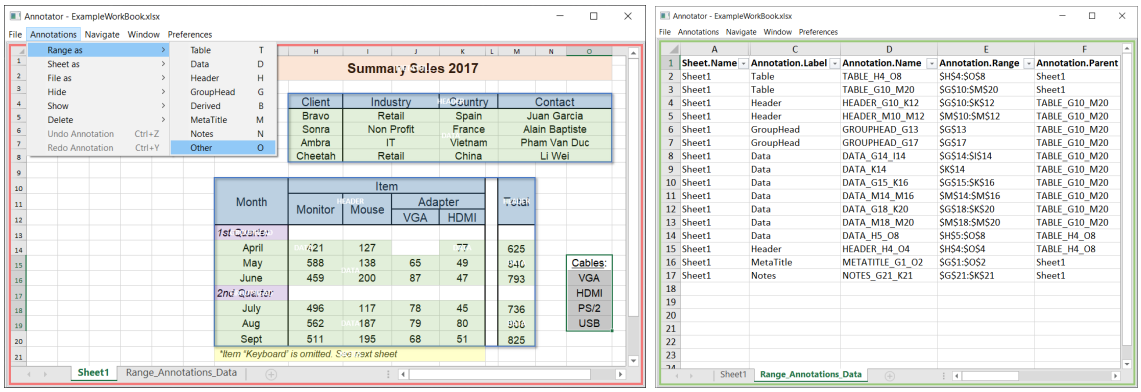
Sheet Labels

Besides cells, we annotate non-empty sheets of a spreadsheet file. Here, we focus on those that do not contain tables, which we refer to as *Not-Applicable (N/A)*. These kinds of sheets are flagged with one of the following labels: Form-Template, Report-Balance, Chart-Diagram, List, NoHeader, and Other. We define *Form-Templates* as sheets intended to be re-used again for similar tasks (e.g., collecting data, performing specialized calculations). Therefore, they are usually accompanied by instructions on how to use them. They might be filled with example values or not. *Balance-Reports* are typically used to summarize financial performance. They might report on the company's assets, liabilities, and shareholders' equity. Often, data in these sheets are not organized in a strictly tabular fashion. Next, the label *List* is used for sheets that have all values placed in a single column. *NoHeader* applies when we do not find Header cells in the sheet, even though there are values in multiple rows and columns. *Chart-Diagrams* are sheets that contain plots/diagrams and the source values (if any). Note, the source can not be a proper table, otherwise, the sheet is applicable and has to be annotated as such (see Section 3.2.2). Finally, we introduce the label *Other*, for sheets that do not match any of the aforementioned *N/A* labels.

[3]Unlike Adelfio and Samet, we differentiate between valid and non-valid tables, rather than between relational and non-relational.

Annotation Tool

For this work, we created a specialized annotation tool [85], shown in Figure 3.3. It was developed in Java programming language, using the Eclipse SWT library [56]. Specifically, the tool loads an Excel file inside an SWT OLEFrame. Within this controlled environment, Excel functionalities that are needed for the annotation are enabled, while the rest are blocked. Essentially, the aim is to prevent any alteration of the original contents and formatting. At the same time, we attempt to simplify the annotation steps, by supporting the user throughout the process.



(a) User Interface and Menus (b) Saved Annotation Data

Figure 3.3: Screenshots from the Annotation Tool

The tool supports the labels defined in the previous sections. Cells are annotated by first selecting a rectangular range from the main window and then a label from the designated sub-menu (see Figure 3.3.a). Likewise, the active sheet itself can be labeled using options from the menu (*Annotations->Sheet As->...*). Nevertheless, users can perform the same actions with shortcuts, which can speed up the annotation process.

The tool is designed for interactive annotation. We use build-in Excel shapes, to display annotations made by the current user. Table annotations are visualized as rectangles, having blue borders but no fill color. While, for other cell annotations, we use semi-transparent rectangles. Their fill color depends on the assigned label (e.g., green for Data, blue for Header, etc.). These shapes overlay the existing content. Thus, they do not interfere with the original data and formatting. On the contrary, they allow the user to keep track of existing annotations, and simultaneously reason on the remaining parts of the sheet. Options from the menu, like delete and undo, allow the user to change annotations if that is necessary.

Additionally, the tool runs background checks, ensuring annotation integrity and enforcing our annotation logic (refer to Section 3.2.2). For example, we prevent the user from annotating regions that overlap with existing annotations. Moreover, we make sure that certain dependencies are satisfied. For instance, labels such as Header and Data are only allowed inside a region previously annotated as Table (see Section 3.2.2). Some background checks occur when the user saves or changes the status of a sheet. Among others, we check if there are non-empty cells without annotation (i.e., unlabeled cells), and raise a warning if so. All in all, background checks like the ones listed above, together with the intuitive interface of the tool, improve the quality of annotations.

When a user saves, the annotations are stored inside the loaded Excel file. For this purpose, the tool creates two hidden and protected sheets. The "*Range_Annotations_Data*"

sheet stores information related to cell annotations. As can be seen in Figure 3.3.b, for an annotated region we store its location (sheet name and address) and the assigned label. We also record the parent of each annotation, which later can be used to build a hierarchical tree representation of the overall layout. The sheet is always the root node in this hierarchy, while tables typically act as parents for Data, Header, GroupHeader, and Derived regions. Furthermore, the status of the overall file and the labels of the individual sheets are recorded in another hidden sheet, named "*Annotation_Status_Data*"[4]. As detailed in Section 3.2.2, the status remains *In-Progress* unless the user indicates from the menu that the file is either *Completed* or *Not-Applicable*.

Files that are saved and not *In-Progress* are organized by the tool into folders. They end up in the "*completed*" or "*not-applicable*" folder, based on the respective status. The *not-applicable* folder is divided further into subfolders, which correspond to the predefined N/A labels (see Section 3.2.2). This with the exception of the *multi-na* subfolder, which holds files with multiple N/A sheets, but flagged with a different label.

Annotation Logic

In this section, we describe the process of annotating a single spreadsheet file. In addition to the constraints mentioned in Section 3.2.2, the user follows predefined annotation steps. These are illustrated with a simplified diagram in Figure 3.4, and as well summarized in the following paragraph.

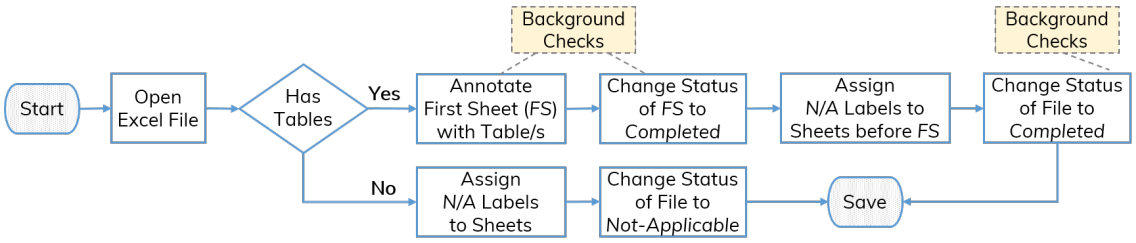


Figure 3.4: Annotation Steps

A judge (annotator) inspects the provided spreadsheet file for table/s. When the file does not contain any table, all the sheets must be flagged with the appropriate *N/A* label (see Section 3.2.2). Subsequently, the status of the file itself must be changed to *Not-Applicable*, before saving it. In the opposite case, the file has one or more sheets with tables. In an effort to reduce duplicate (repetitive) annotations, we ask the judges to annotate only the first sheet (FS), among those having table/s. To determine FS, the judges follow the order of the tabs from left to right. All the tables and non-empty cells in FS are annotated using the labels from Section 3.2.2. Subsequently, the status of FS is changed manually to Completed. Furthermore, all the sheets without a table, coming before FS, have to be annotated[5] with the appropriate *N/A* label. The judge omits the sheets coming after FS. The task concludes when the judge changes the overall status of the file to *Completed* and then saves it.

[4]The "*Annotation_Status_Data*" sheet is not displayed in Figure 3.3
[5]This forces the judge to justify his/her decision for skipping sheets coming before FS

Judges and Annotation Phases

Three judges participated in the creation of the gold standard dataset. All of them are students in STEM fields. They had various degree of familiarity with Excel, prior to this project. To avoid any influence whatsoever, briefing and communication with the judges were handled individually.

The overall annotation process was organized into three phases: training, agreement assessment, and independent annotation. The aim of the first phase was to familiarize the judges with the tool, task, and annotation labels. They were given a written description of the task, annotated examples, and a small sample of files to practice with. In the second phase, we performed an assessment of agreement between the judges, using a common dataset of 128 files (one random file per Enron employee). The aim was to ensure a good and consistent understanding of the annotation labels, before the third phase. Note, this is crucial for a dataset created by multiple independent participants, with different initial knowledge. Therefore, we elaborate more on this in the following section of this chapter. In the third and final phase, the judges were provided with individual datasets and worked under minimum supervision. We excluded files that were used in the earlier phases. Specifically, each judge got a stratified sample from the remaining Enron dataset, covering files from 120 to 122 Enron employees. Additionally, we provided samples from the Fuse dataset. However, these were smaller in size compared to the ones from Enron.

Agreement Assessment

As mentioned previously, the agreement between the judges was assessed on a common dataset of 128 files, extracted from Enron corpus. Subsequently, we instructed them to review files in which we identified substantial differences. These disagreements were described at the cell, sheet, and file level. Note that some disagreements were due to negligence. Thus, another purpose of this phase was to fix trivial mistakes. Regardless, the judges could still choose to keep their annotations, if they considered them to be correct. In other terms, it was up to them to decide if to change their initial annotations.

Following these revisions, the second assessment of the agreement was performed. The results of this assessment are presented in Table 3.1. We use two metrics: Fleiss' Kappa [53] and Agreement Ratio. The former is a statistical measurement for the reliability of agreement between multiple judges. The latter captures the percentage of annotated items for which all judges agree (i.e., they are in unison). For cells, judges voted with one of the seven available annotation labels. For files and sheets, we considered a binary vote: Not-Applicable or Completed. We did not assess agreement individually for the six *N/A* labels (refer to Section 3.2.2)

Table 3.1: Annotation Agreement Assessment

	Files	Sheets	Cells
Fleiss Kappa	0.77	0.72	0.86
Agreement Ratio	0.90	0.97	0.98

	Data	Header	Derived	Title	Other	GHead	Notes
Agreement Ratio	0.98	0.89	0.70	0.53	0.41	0.40	0.20

As shown in Table 3.1, the agreement and its reliability are substantial (i.e., definitely not random) [90], when studied at the file, sheet, and cell level. Moreover, we measured the agreement individually for each cell label[6]. We observe that labels closely associated with tables are more natural to the judges. Specifically, for Data and Header, the agreement ratio is notably high. Additionally, there is a significant agreement for Derived. For the remaining labels, the agreement is much lower. However, we are not the first to encounter such results. With their empirical study, Hu et al. [75] prove that table ground-truthing is hard. They find that there are significant disagreements even between human experts. Besides the human factor, another aspect to consider is the frequency of labels in the annotated sheets (refer to Section 3.3). The impact of disagreements is naturally much more pronounced for labels that have low occurrences, such as Titles, Notes, and GHead.

Nonetheless, before proceeding to the last annotation phase (i.e., independent annotation), we inspected the files once more. However, this time we checked them manually, in order to understand the real reasons behind the remaining disagreements. For each judge, we determined cases where they truly had used the labels incorrectly. We discussed these cases individually with them, clarifying any remaining misunderstandings. Moreover, we instructed them to correct these faulty annotations. Subsequently, we examined again the files, to make sure that the required changes were indeed performed.

3.3 DATASET ANALYSIS

Here, we discuss the analysis of two annotated samples, one from Enron corpus and the other from Fuse corpus. In Section 3.3.1, we focus entirely on the analysis of the Enron sample. We choose to do so since business spreadsheets have been so far overlooked by related work. This is in contradiction to the fact that spreadsheets are extensively used in business settings. Therefore, it is worthwhile focusing on the Enron sample, as it provides a unique and invaluable view on the use of spreadsheets in a real-life company. Nevertheless, in Section 3.3.2, we return our attention to the Fuse sample. We summarize the results of our analysis and make a direct comparison with the Enron sample. We discuss similarities and differences in these two samples, in an attempt to highlight their unique characteristics and challenges.

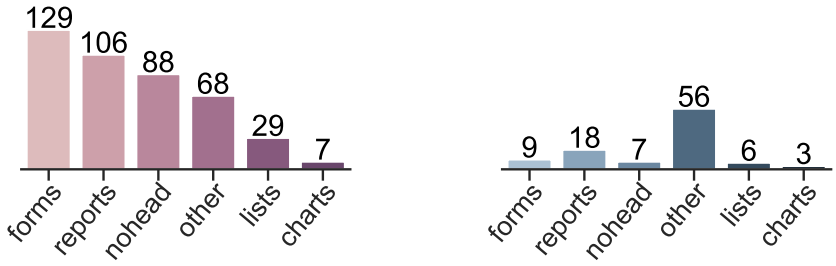
3.3.1 Takeaways from Business Spreadsheets

The judges annotated a total of $1,160$ files from Enron, where 306 were marked as *Not-Applicable* (i.e., do not contain table/s) and 854 were annotated at the cell level. Below we begin our analysis with *N/A* labels.

Not-Applicable Sheets

Here, we discuss sheets annotated with *N/A* labels. As mentioned in Section 3.2.2, these sheets can be found both in *Not-Applicable* and *Completed* files. We report on the occurrences of these sheets, separately in Figures 3.5.a-b. We observe that *Form-Template* and *Report-Balance* sheets are the most frequent. Such use is indeed expected in business settings. Additionally, we notice a high number of *NoHeader* sheets. This suggests that

[6]We omit Fleiss' Kappa for individual cell labels. For most cases, vote distribution is strongly biased towards one label (i.e., the label that is currently being studied). Thus, Fleiss' Kappa is not appropriate [102].



(a) Counts in Not-Applicable Files (b) Counts in Completed Files

Figure 3.5: Number of Sheets per N/A Label

occasionally users might omit headers, and rely on implicit information. We choose to see these Header-less regions, of just "Data" values, as non-valid tables. This complies with existing approaches for table detection and recognition in spreadsheets, such as [6, 11, 29, 46], which largely depend on the context provided by headers.

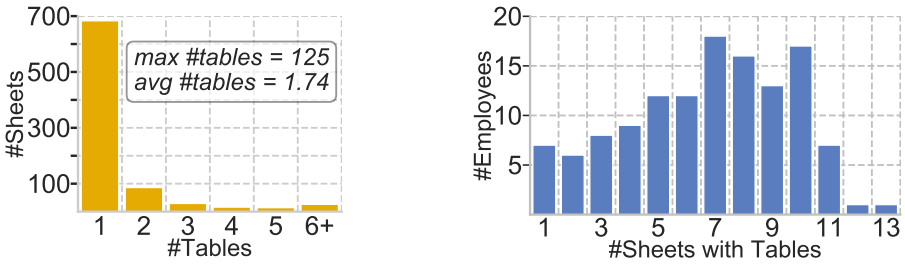
Takeaway 1: Business users rely on implicit information. This might lead to omitted headers.

Annotated Tables

Hereinafter, we discuss sheets containing table annotations. Overall, the judges annotated $1,487$ tables, in 854 sheets. Figure 3.6.a shows the distribution of these tables. The vast majority, 683 sheets, contain only one table. The rest, 171 sheets, have two or more tables. Moreover, we find a few extreme outliers, i.e., sheets with 34, 49, or even 125 tables.

Takeaway 2: In 20% of the sheets we find two or more tables. Thus, the simplistic view of one table per sheet, does not hold for a significant portion of sheets.

In addition, we examine the number of sheets (containing table annotations) per Enron employee. Figure 3.6.b summarizes our analysis. We observe that the vast majority of Enron employees contribute from 5 to 10 sheets in the annotated sample.



(a) Table Distribution (b) Employee Sheet Counts

Figure 3.6: Table Annotations in Numbers

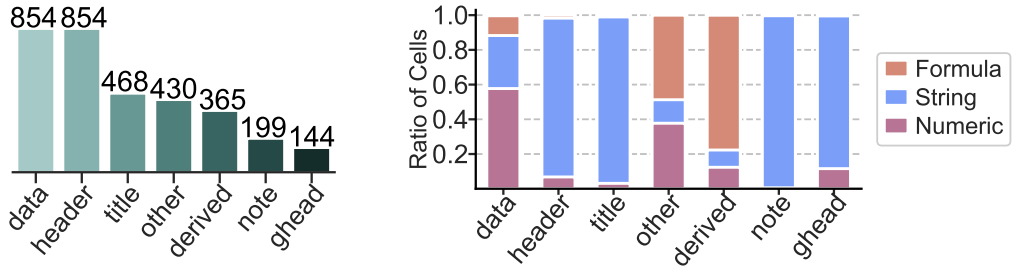
Annotated Cells

Figure 3.7.a summarizes the occurrences of cell labels in applicable sheets (i.e., having table annotations). *Data* and *Header* are present in all these sheets, as it was intended. Moreover, we observe a high occurrence of *Titles*, which seem to be preferred over *Notes*. We find *Derived* in ca. 43% of the sheets. This confirms our expectations since related work reports that 58% of the original Enron files contain formulas [72].

Takeaway 3: We note that >40% of business spreadsheets contain (Derived) aggregations.

We have also identified hierarchies in the applicable sheets. *GroupHeaders* (shortly denoted as *ghead*) represent the parents in the left hierarchies of tables. As shown in Figure 3.7.a, they occur in 144 sheets (17%). Additionally, we analyzed the sheets for nested *Headers* (i.e., top hierarchies). They occur in 32% of the sheets. Overall, 43% of applicable sheets have tables with either top or left hierarchy. These findings call for specialized approaches to handle hierarchical data in spreadsheets, as proposed in [29, 30].

Takeaway 4: A considerable number of business spreadsheets have top and/or left hierarchies.



(a) Sheet Counts per Cell Label

(b) Content Type Ratios per Cell Label

Figure 3.7: Cell Annotations

Furthermore, Figure 3.7.a shows that more than half of the sheets have cells labeled as *Other*. This implies that spreadsheet contents are highly diverse. Thus, even more labels than the ones considered by this work could be used to describe spreadsheet contents.

Takeaway 5: We observe substantial variety of contents in business spreadsheets.

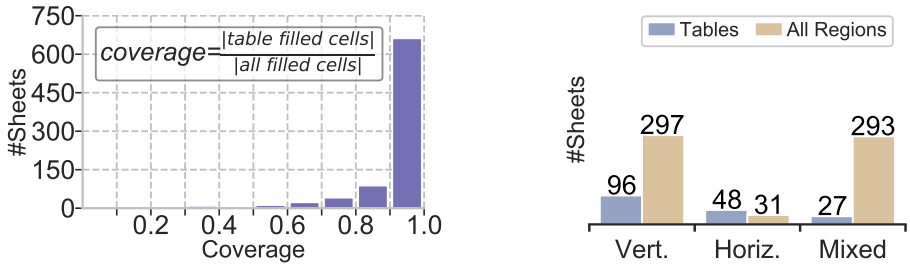
In addition, in this section, we discuss the distribution of content types per cell label. The results are shown in Figure 3.7.b. As anticipated, for cells annotated as *Header*, *Title*, *Note*, and *GroupHeader* we observe mostly string values. With regard to *Data* cells, we find a considerable amount of strings (ca. 30%), as well. In *Derived* cells, we notice a small portion of numeric values, which suggests that occasionally users set the aggregation values manually (i.e, without using formulas). Finally, most of the cells labeled as *Other* are non-strings. This implies that the label *Other* might be closer to *Data* and *Derived*, rather than to the remaining labels.

Sheet Layout

In this section, we discuss the layout of the annotated sheets. We examine the annotated tables, to determine what portion of the sheet they usually occupy. Moreover, we report our findings with regard to the spatial arrangement of layout regions, including tables.

One of the common assumptions is that tables are the main source of information in a sheet. In other words, we expect most of the sheet to be covered by tables. We put this assumption into a test, using the *coverage* metric, proposed by [20]. Intuitively, coverage captures the ratio of filled-in cells (i.e., non-empty cells, with a value) located inside the annotated tables of a sheet. The formula for the calculation of coverage and the results of this analysis are provided in Figure 3.8.a. To show the distributions of this metric, we use a histogram consisting of 10 bins (intervals), each having a width of 0.1. The results indicate that tables occupy $> 80\%$ of the used space for the vast majority (88%) of the sheets. Thus, indeed most of the filled-in cells belong to the annotated tables. In this regard, spreadsheets differ from other document formats, like scanned documents, PDFs, and HTML pages. There, tables use only a small part of the page, the rest is typically text.

Takeaway 6: Tables typically dominate the sheets, by taking up most of the used space.



(a) Table Coverage (b) Arrangements

Figure 3.8: Coverage and Arrangements in Enron Spreadsheets

Figure 3.8.b reports on the arrangement of components (i.e., layout regions) in the annotated sheets. We performed this study twice. First, we analyzed the arrangement of tables. For this, we considered only sheets that contain multiple tables. We find that vertical (top-bottom) arrangements are the most prevalent in multi-table sheets. Nevertheless, we notice a significant number of cases with horizontal (left-right) or mixed (both vertical and horizontal) arrangements. For the second part of this study, we considered sheets having annotated cells outside of tables, such as Titles, Notes, and Other. We examine how such cells are arranged in relation to the tables in the sheet. In this case, our analysis shows that mixed arrangements are almost as common as vertical ones. This means one can potentially find arbitrary content next to the spreadsheet tables, in all four directions.

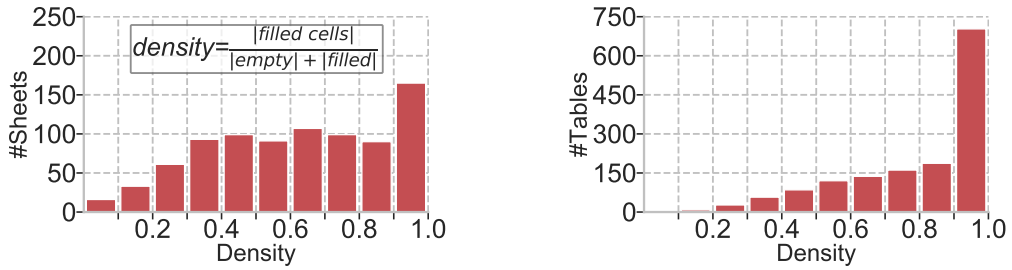
Takeaway 7: We frequently observe mixed arrangements in business spreadsheets. Especially, when we consider other layout regions, besides tables.

These results bring forward the limitations of approaches doing layout analysis at the row level, such as Chen et al. [29], Adelfio and Samet [11]. Clearly, these works would perform poorly when cells of the same row exhibit different layout functions. Our analysis shows that this is often the case, as we find a considerable number of horizontal and mixed arrangements. In Section 3.4, we consider such insights for the proposed solution.

Content Density

In this section, we study the density of spreadsheet contents. For this analysis, we re-use yet another metric proposed by [18]. It captures the concentration of filled-in (i.e., non-empty) cells in a sheet. We can measure this concentration (density) for the whole sheet,

or for specific regions, such as the annotated tables. The formula for the density metric is displayed in Figure 3.9.a. We initially determine the minimum bounding rectangle (MBR) that encloses the region of interest. Subsequently, we find the ratio of filled-in cells inside this MBR.



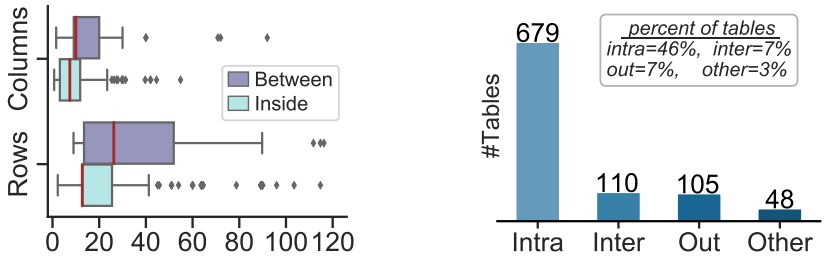
(a) Sheet Densities (b) Table Densities

Figure 3.9: Content Density in Enron Spreadsheets

Figure 3.9 shows the density distribution for annotated sheets and tables. We observe that densities in sheets vary extensively (see Figure 3.9.a). Partially, this is due to cells located outside of tables, such as Titles, Notes, and Other. Typically, users leave some space between these cells and tables. However, as can be seen in Figure 3.9.b, there is a considerable number of sparse tables, as well. Specifically, more than half, (53%) of the tables, have a density of ≤ 0.9. These might be due to missing/implicit values or empty rows and columns. As reported by related work [46], the empty rows/columns are often used for visual padding inside the tables.

Takeaway 8: Content density varies extensively in spreadsheets. Sparse tables are common.

In fact, our analysis shows that empty rows/columns are used both inside and outside tables. This is very relevant for the table detection task. To improve accuracy, a system has to distinguish between these two uses. One naïve approach is to study the size[7] (height or width) of such empty rows/columns (shortly referred to as gaps). Intuitively, gaps inside the tables should be smaller in size than those outside. We have tested this assumption and visualized the results in Figure 3.10.a (outliers >120 are omitted). Note, we treat consecutive empty rows/columns as one gap (i.e., cumulative width/height). Furthermore, in sheets having gaps, we consider only the biggest one inside tables and the smallest one outside tables.



(a) Gaps Between/Inside (b) Formula References

Figure 3.10: Gaps and Dependencies in Enron Spreadsheets

[7]In Excel, height of rows is measured in points, while width of columns is measured in units of 1/256th of a standard font character width.

Our analysis, shown in Figure 3.10.a, revealed that a significant number of tables contain gaps: 546 with empty rows, and 240 with empty columns. Furthermore, for column gaps, we notice considerable overlap in the size (width) distributions, *inside* and *between* tables. Thus, it is not enough to consider just the size of a gap. More context is needed to predict its true purpose: table separator or visual padding.

Takeaway 9: We find empty row/column inside tables. To distinguish them from those found between tables, an analysis that goes beyond their width/height is needed.

Content Dependencies

We conclude with a study of formulas found in the annotated sheets. Here, we focus on the references of these formulas, which ultimately introduce dependencies between the cells (i.e., contents) of a spreadsheet. The results of this analysis are presented in Figure 3.10.b. Intra-table dependencies, i.e., formulas referencing cells within the same table, occur in 679 (46%) annotated tables. We notice external references, i.e., outside of the table, less often. These can be from one table to another (*Inter*), referring to cells found outside the table (*Out*), or references to other sheets of the same file (*Other*).

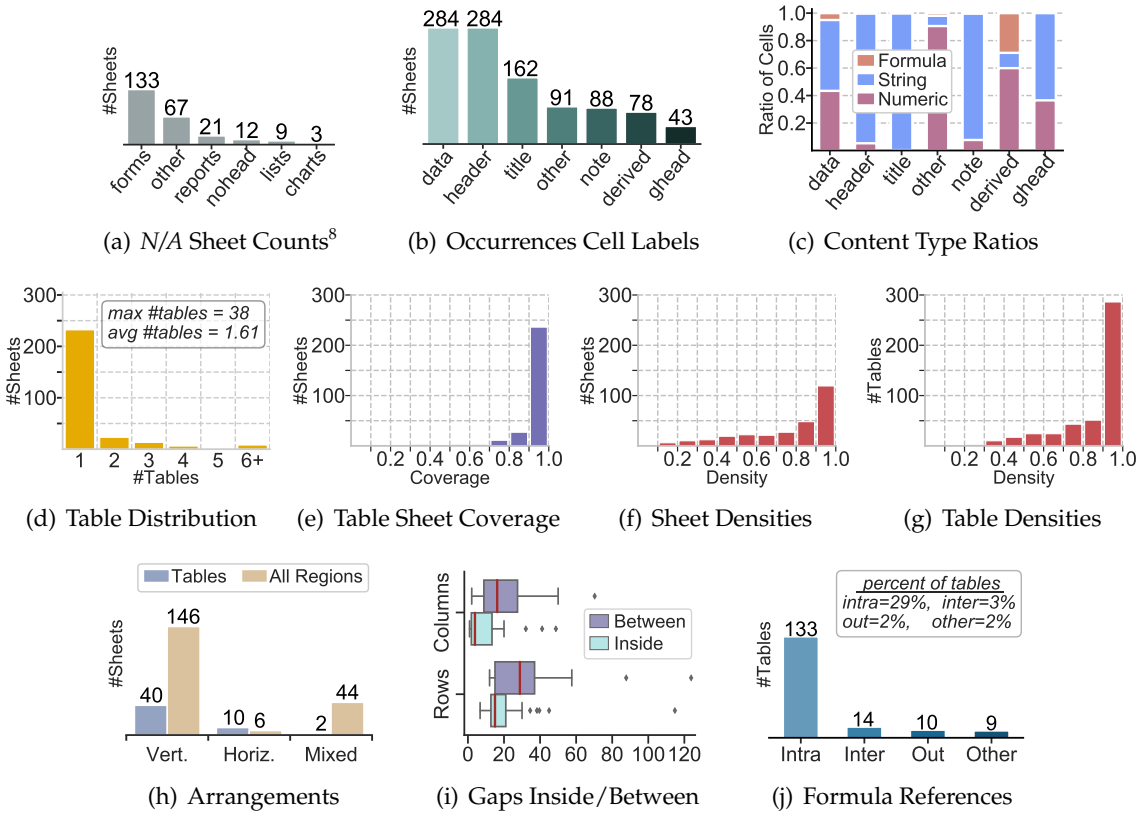
Takeaway 10: Table contents might depend on values found outside its borders; infrequently, these reside in other sheets.

3.3.2 Comparison Between Domains

As discussed in Section 3.1, related works in layout analysis and table recognition have overwhelmingly used web-crawled spreadsheets. However, they have not made their annotations publicly available. Therefore, we annotated a sample of spreadsheets from the Fuse corpus, which originates from the Web. We perform a thorough study on this sample, in an attempt to infer the unique characteristics of Web spreadsheets. Subsequently, we compare with the Enron sample, highlighting similarities and differences.

Many files in the original Fuse sample were filtered during the annotation process, As mentioned in Section 3.2.1, this is because we could not perform some of the preprocessing steps used for Enron. Overall, the judges manually found 278 near-duplicates. Such files most likely originate from the same website, since they have (almost) the same structure and formatting, but differ in values. We choose to eliminate these files, in order to avoid any risk of overfitting on specific examples. Additionally, the judges found 49 non-English files and 71 files that caused run-time exceptions. These files were eliminated from the sample, as well.

The final annotated sample consists of 406 files from Fuse: 122 *Not-Applicable* and 284 *Completed* (i..e, having tables). The latter contain 458 annotated tables, in total. Subsequently, we performed the same analyses, as in Section 3.3.1. The results for the Fuse sample are summarized by charts, shown in Figure 3.11.



(a) *N/A* Sheet Counts[8] (b) Occurrences Cell Labels (c) Content Type Ratios

(d) Table Distribution (e) Table Sheet Coverage (f) Sheet Densities (g) Table Densities

(h) Arrangements (i) Gaps Inside/Between (j) Formula References

Figure 3.11: Characteristics of Annotated Fuse Sample

Similarities

We observe similarities between the two annotated samples. As shown in Figure 3.11.a, likewise in Fuse most of *N/A* sheets are *Form-Templates*. With regard to applicable sheets, we note that many of the takeaways discussed in Section 3.3.1 apply to the Fuse sample, as well. For instance, there are hierarchies in Fuse, i.e., *GroupHeaders* (*ghead*) and nested *Headers*. They occur respectively in 15% and 28% of the sheets, which is comparable to the Enron sample (see Section 3.3.1). Other similarities relate to the annotated tables. In Figure 3.11.d, we note multiple tables in 52 (18%) of the applicable sheets from Fuse (compared to 20% in Enron). Moreover, in the vast majority of applicable sheets, tables cover the largest part of the used area (refer to Figure 3.11.e.). Furthermore, similar to Enron, the content densities (Figures 3.11.f-g) vary extensively for both sheets and tables. In addition, we note the presence of horizontal and mixed arrangements (Figure 3.11.h), together with empty rows/columns in tables (Figure 3.11.i), and external formula references (Figure 3.11.j). However, as mentioned below, such cases are less common in Fuse.

Differences

Here, we highlight the most important differences between the two annotated samples. In Table 3.2, we observe that, with the exception of *Form-Templates* and *Other*, the remaining *N/A* labels occur with much lower frequency in Fuse. Moreover, as shown in Table 3.3, cells labeled as *Other* and *Derived* are not as common as in Enron. With regard

[8]Cumulative count for each *N/A* label, considering both Not-Applicable and Completed files.

Table 3.2: Percentage per *N/A* Label

	Forms	*Other*	*Reports*	*NoHeader*	*Lists*	*Charts*
Enron	26%	24%	24%	18%	7%	2%
Fuse	54%	27%	9%	5%	4%	1%

Table 3.3: Percentage of Applicable Sheets containing each Cell Label

	Data	*Header*	*Title*	*Other*	*Derived*	*Note*	*GHead*
Enron	100%	100%	54%	50%	43%	23%	17%
Fuse	100%	100%	57%	32%	27%	30%	15%

to *Derived*, it has been already reported by related work that the Fuse corpus contains fewer formulas than the Enron corpus [16]. This can be seen in Figure 3.11.c, where the occurrence of formulas is low for all cell labels, including *Derived*[9]. Additionally, in Figure 3.11.j, formula references (i.e., dependencies) are found in a smaller percentage of annotated tables, compared to Enron. Overall, these results indicate that there are fewer dependencies (related to formulas and *Derived* cells) and lower diversity of contents (related to *Other* cells) in the applicable sheets from Fuse.

As mentioned before, there are many cases of sparse sheets and tables in both samples. However, we observe that in Fuse the densities tend to be higher than in Enron. In Table 3.4, we see that 59% of applicable sheets in Fuse have a density of > 0.8, i.e., two times more than Enron. For tables, the difference is much smaller. Nevertheless, it is noticeable that high densities are more common for Fuse tables. Furthermore, we find in Fuse considerably fewer cases of empty columns/rows inside the annotated tables (refer to Table 3.5). Considering these results, we conclude that contents in Fuse sheets are much more compact.

Table 3.4: Occurrences of Densities

	Density > 0.8		*Density* > 0.9	
	Sheets	*Tables*	*Sheets*	*Tables*
Enron	30%	60%	19%	47%
Fuse	59%	74%	42%	62%

Table 3.5: % of Tables with Gaps

	Empty Columns	*Empty Rows*
Enron	16%	37%
Fuse	5%	19%

As can be seen in Table 3.6, there are evident differences when it comes to the arrangement of layout regions. For Fuse, the vertical arrangement clearly dominates over horizontal and mixed arrangements. This is especially true when we consider the arrangement of all layout regions. While, in the same settings, for Enron, the vertical and mixed arrangements are equally likely.

Last, we perform a focused comparison of the two samples. Specifically, we measure the occurrence of *simple sheets*, which we define as follows. They can have multiple layout regions and one or more tables. However, we require that the arrangement in these sheets is vertical. Furthermore, the contained tables must be simple as well. Therefore, we omit

[9] In Fuse spreadsheet users tend to set the aggregation values manually, without the help of formulas.

Table 3.6: Percentage of Applicable Sheets with the Following Arrangements

		Vertical	*Horizontal*	*Mixed*
Multi Tables	Enron	11%	6%	3%
	Fuse	14%	4%	1%
All Regions	Enron	35%	4%	34%
	Fuse	51%	2%	15%

sheets having tables with hierarchies, gaps, and (Derived) aggregations. Additionally, we exclude sheets having *Other* cells, which means arbitrary contents (i.e., not simple). All in all, simple sheets can be described as being closer to the commonly held assumptions about spreadsheet contents. The results, in Table 3.7, show that in Fuse 38% of applicable sheets are simple, compared to 16% in Enron. Arguably, one can attribute Derived cells and nested Headers (i.e., top hierarchies) to the typical spreadsheets. Therefore, we provide the results of our analysis including these cases, as well.

Table 3.7: Percentage of Simple Sheets

	Simple	*+Derived*	*+Top Hierarchies*
Enron	16%	17%	21%
Fuse	38%	40%	46%

Overall, we observe that sheets in the Fuse sample tend to be more regular. To this might contribute the fact that a considerable number of files in the original Fuse corpus come from .org (29.5%), and .gov (27.7%) domains [16]. Thus, it is very likely that these files were crawled from open data platforms or other official sources. This can explain why overall the Fuse sample exhibits more structure, lower diversity in layouts, and higher density of contents.

Contrary, in the Enron sample we find more instances of "irregular" sheets. We take this insight into consideration while comparing with related works. We aim to achieve comparable or even better performance on the Fuse sample, which is closer to the related works. At the same time, we attempt to improve the current state-of-the-art, mainly based on the Enron sample. Ultimately, our goal is to achieve high performance even in complex sheets, exhibiting many of the aforementioned challenges: hierarchical structures, multiple tables, various layout roles, mixed arrangements, sparse contents, in-table gaps, etc.

3.4 SUMMARY AND DISCUSSION

The results of our empirical study show that there are many challenges when it comes to automatic analysis and recognition in spreadsheets. Here, we begin with a brief summary of challenges that have been already identified by related work. Subsequently, we highlight those that are so far overlooked by them. Lastly, we define the scope of this thesis, describe the datasets used for experimental evaluation, and provide a high-level overview of the proposed solution.

As reported by other works, spreadsheets contain tables, but also other structures (forms, reports, etc.). When table/s are present in the sheet, they tend to occupy most of the used area. Nonetheless, next to the tables we can find meta-information, such as Titles and Notes. Moreover, the structure of the tables can vary, even inside the same sheet. In complex cases, we encounter tables with nested Headers (i.e., top hierarchies) and/or nested data in columns (i.e., left hierarchies). Additionally, we find tables containing various aggregations.

Already, the aforementioned challenges, highlight the complexity of analysis and recognition in spreadsheets documents. Nevertheless, we have identified aspects that have not been considered by related work, yet. In particular, we discuss the following additional challenges for spreadsheet documents: diversity of contents, arbitrary arrangements, low density, and in-table gaps (i.e., empty row/columns). Our analysis, in Section 3.3.2, shows that spreadsheets are much more diverse than what is commonly assumed. Specifically, we find that cells annotated as *Other* are fairly frequent. This means, next to the tables, one can find lists, key-value pairs, and other arbitrary contents. Clearly, such instances, need to be distinguished from true tables. Another overlooked challenge is the horizontal and mixed arrangements in spreadsheets (refer to Table 3.6). In such cases, approaches performing layout analysis at the row level, such as [11, 29], will have low accuracy. Thus, in order to cover all arrangements, a candidate solution needs to reason at the level of individual cells or ranges of cells. Another important finding is that spreadsheet tables are often sparse (see Figures 3.9 and 3.11). Again, this can have a negative impact for the approaches proposed at [11, 29]. Specifically, due to empty (missing) cells, the composition and therefore features of rows can vary greatly, even within the same table. As a result, inference (i.e., training and classification) at row level can be difficult, even when we do not account for arbitrary arrangements. Furthermore, gaps in tables introduce considerable overhead for any candidate solution. They can be misinterpreted for separators of contents (tables), when in fact they are just used for visual padding. Finally, the greatest challenge comes from sheets that exhibit simultaneously several "irregularities" (including those mentioned in related work). As shown in Table 3.7, this is the case for more than half of the sheets, for both annotated samples.

Thus, to automatically process arbitrary spreadsheets, one needs to tackle all the abovementioned challenges. Specifically, before extracting information from spreadsheets, one needs to achieve high accuracy for layout analysis, table detection, and table recognition tasks. Therefore, in this thesis, we focus particularly on these three tasks. Our findings, from this chapter, show that these tasks are not fully solved, yet. In addition, we believe that related works have not taken full advantage of the rich spreadsheet ecosystem. There are aspects (features) that could improve the accuracy but have not been explored in the existing approaches. Thus, in this work, we attempt to fill this gap as well.

Nevertheless, there are a few scenarios that will not be covered by this thesis. Specifically, we do not address transposed tables, i.e., the Header cells are on the left columns instead of the top rows. Additionally, we omit cases where tables are horizontally attached, i.e., they are not separated by empty column/s. For a human is easy to see that these are separate tables, due to their formatting and values. However, this is much more challenging for a machine.

Essentially, excluding the aforementioned scenarios, means omitting a small portion of the annotated sheets. Therefore, in the next section, we discuss how this affects the Enron and Fuse sample. We use these slightly reduced samples, to evaluate the proposed approach. In Section 3.4.2 we provide a high-level overview of this approach and outline the remaining structure of this thesis.

3.4.1 Datasets for Experimental Evaluation

We use the two annotated samples, discussed in this chapter, to train and test the proposed solution. However, we need to omit a small portion of annotated sheets that are outside the scope of this thesis. Moreover, we omit few "outlier" sheets, that contain an exceptionally large number (> 17) of annotated tables. We believe that these sheets can bias our approach, by dominating over the other rest. Below, we discuss how these changes have affected the Enron and Fuse sample.

The reduced Enron sample contains 814 annotated sheets, i.e., 40 were omitted. Specifically, we omit 20 sheets that have transposed tables and 18 that have horizontally attached tables. Moreover, we excluded three sheets that contain 34, 49, and 125 tables, each. The latter sheet exhibits horizontally attached tables, as well. Therefore, in total, the number of omitted sheets is 40.

For the Fuse sample, we omit only 10 sheets. Therefore the reduced sample still contains 274 sheets. From the excluded, 5 have transposed tables, and 4 horizontally attached tables. Lastly, a sheet containing 38 tables was excluded, too.

For completeness, in Appendix A, we have recreated the charts provided previously, in Section 3.3, for the reduced samples. With this, we show that indeed the aforementioned omissions have not altered the composition of the samples. The challenges identified and discussed in Section 3.3, are still applicable.

3.4.2 A Processing Pipeline

We propose a spreadsheet processing pipeline, illustrated in Figure 3.12. Our approach is bottom-up, as it starts from the individual cells and ultimately arrives at larger structures (i.e., tables). Note, we have marked the table recognition and information extraction tasks with dotted lines, as our contribution is less significant in them compared to other parts. For these two tasks, we mostly re-use techniques proposed by related work.

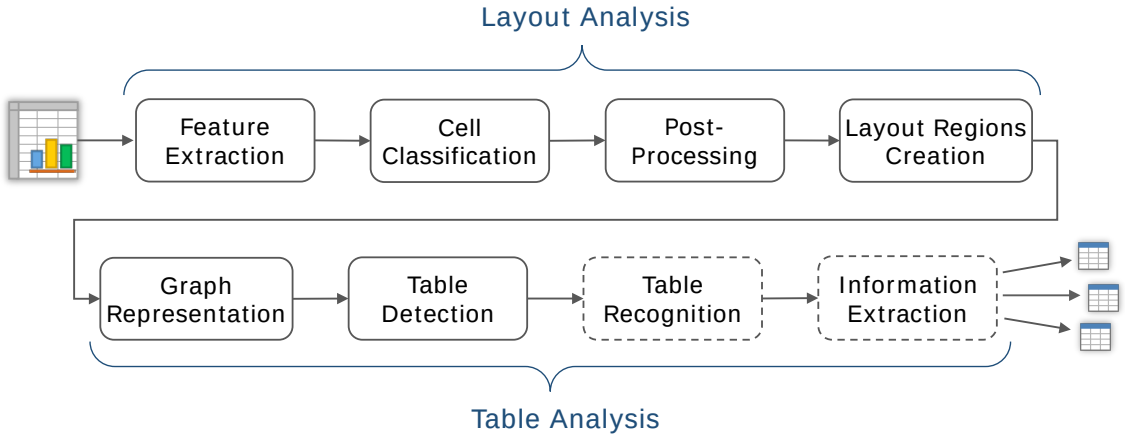


Figure 3.12: A Spreadsheet Processing Pipeline

Overall, the proposed solution follows best practices from the DAR (Document Analysis and Recognition) research field. We chose a bottom-up approach since they are suitable when document layouts are arbitrary (unknown), as suggested in the survey [95].

Furthermore, in this thesis, we make use of machine learning techniques. In the latest years, such techniques have improved the state-of-the-art for many different DAR tasks [94, 97, 95]. Another reason for such techniques is that we envision a solution that works for spreadsheet coming from various domains. Ultimately, our goal is to have a flexible and transferable system, which requires minimum tuning.

In the remaining chapters of this thesis, we describe in detail each one of the steps in our processing pipeline. We begin, with Chapter 4, where we outline how to perform layout analysis, i.e., the first four tasks in our processing pipeline.

4

LAYOUT ANALYSIS

4.1 A Method for Layout Analysis in Spreadsheets

4.2 Feature Extraction

4.3 Cell Classification

4.4 Layout Regions

4.5 Summary and Discussions

Considering that tables in spreadsheets vary extensively, it is rather challenging to directly recognize them as a whole. Thus, we opt to recognize their building blocks, instead. This process is commonly known as layout analysis (refer to Section 2.2.1). In simple terms, the aim of layout analysis is to segment the contents of a document into regions of interest. Typically, the segmentation is done on the basis of both geometrical and logical analysis. In this thesis, we follow the same logic. Specifically, we propose an approach that operates in a bottom-up fashion. We start from the individual cells and infer their logical (layout) function. Then, we group cells into rectangular regions of interest, on the basis of their inferred function and spatial arrangement. The resulting regions are subsequently used for table detection, which is discussed in Chapter 6.

The subsequent parts of this chapter are organized as follows. In Section 4.1 we outline the overall approach for layout analysis. Subsequently, from Section 4.2 to 4.4, we discuss the individual, steps in more detail. Finally, in Section 4.5, we reexamine the approach, this time considering the results from the experimental evaluations.

4.1 A METHOD FOR LAYOUT ANALYSIS IN SPREADSHEETS

In this section, we outline our solution for layout analysis in spreadsheets. We begin with a summary of existing works. Subsequently, we motivate the need for a new approach and describe the individual steps that comprise it.

There are several approaches for layout analysis in literature, which are discussed in more detail in Section 2.3.1 and Section 2.3.3. Here, we provide a brief summary. In [6], the authors infer the headers of tables and subsequently detect unit errors. They make use of several rule-based algorithms to predict the function of individual cells. The DeExcelerator framework [46] implements a wide collection of rules and heuristics for layout analysis. Among other things, the framework addresses header, data, aggregations, and metadata (i.e., notes and titles). Some of the defined rules take into account the characteristics of entire rows/columns. Other rules focus on individual cells. An alternative approach for layout inference is proposed in [118]. The authors define a domain-specific language, refereed to as CRL (Cell Rule Language). Using this language, the authors manually define rules that tag the individual cells as *Category*, *Label*, or *Entry*. Then, they extract information from tables using additional rules, again written in CRL. Other works make use of supervised machine learning techniques. Chen et al. [29] use a CRF (Conditional Random Field) classifier to sequentially predict the layout function of rows in the sheet. They propose four such functions: *Title*, *Header*, *Data*, or *Footnote*. Adelfio and Samet [11] make use of CRF as well to classify rows. However, they define seven layout functions: *Title*, *Header*, *Data*, *Notes*, *Aggregates*, *GroupHeaders*, and *Blank*. These functions are illustrated in Figure 3.1, of Chapter 3. Adelfio and Samet additionally combine CRF with an innovative way to encode the individual cell features at the row level. They refer to this encoding as logarithmic binning.

These works have significantly improved the state of the art, with regard to layout analysis in spreadsheets. Nevertheless, as discussed in Section 3.4, there are open challenges not yet covered by related work. In particular, we emphasize that there is a high diversity of contents in spreadsheets. Here, we refer not only to the diversity of values but also to the arbitrary arrangements and formatting (styling). Moreover, we consider the logical interpretation of contents, i.e., their layout function. In Chapter 3, we show that even seven labels might not be enough to describe spreadsheet layouts in detail. Specifically, we find cells annotated as *Other* in a considerable number of annotated files.

Based on these findings, we aim to address layout analysis using supervised machine learning techniques. We believe that such techniques can generalize better than rules when considering the high diversity in spreadsheets. Specifically, machine learning is not limited to human input (i.e., domain expertise). It can handle large volumes of data (i.e., training examples) and make associations that are hidden and complex for humans. In this way, given a dataset of diverse examples, we can train models that are applicable for spreadsheets coming from different sources (e.g., both business and Web). Ultimately, this enables a flexible and transferable approach.

There are already works that have applied successfully machine learning in spreadsheets. As mentioned previously, Adelfio and Samet [11] and Chen et al. [29] achieve good results using CRF classifiers to predict the layout function of individual rows. Instead, we propose an approach that operates at the smallest unit of spreadsheets, i.e., the cell. Specifically, we predict the layout function for each cell, separately. In this way, we can tackle arbitrary arrangements and table structures. We foresee that cells of the same column and row can have different layout functions. Moreover, we recognize that metadata, such as titles, notes, and lists, can be placed anywhere in relation to the table/s.

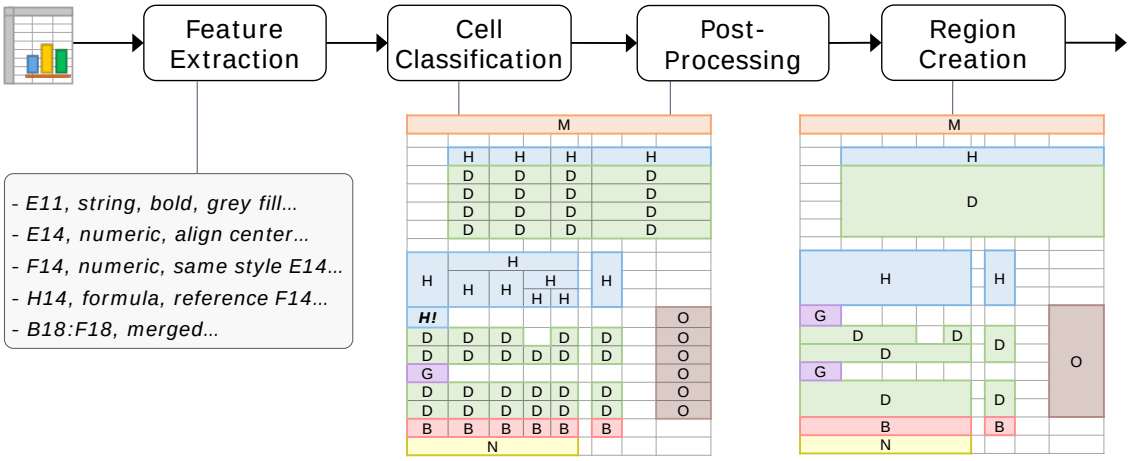


Figure 4.1: The Layout Analysis Process

Figure 4.1 illustrates the four high-level steps that compose our layout analysis approach. We consider the layout functions (i.e., cell labels) defined in Section 3.2.2. In Figure 4.1 we denote each one of these functions with a distinct letter: Header *(H)*, Data *(D)*, Derived *(B)*, Note *(N)*, Title *(M)*, GroupHeader *(G)*, and Other *(O)*.

The proposed approach operates in a bottom-up fashion. Initially, it collects features, which are then used to predict the layout function of individual cells. This is followed by an optional post-processing step, which reviews the classification results. The last step groups the classified cells into larger layout regions. As suggested in literature [95], bottom-up methods are favorable when the layout is not known beforehand. Indeed, as we are going to see in this and the subsequent chapters, our approach shows high flexibility and is capable of capturing a great variety of spreadsheet layouts.

The individual steps of our approach are thoroughly discussed in the following sections of this chapter. Specifically, in Section 4.2, we define the features extracted for each non-empty (i.e., filled with a value) and non-hidden cell in the sheet. Here, we have performed a comprehensive analysis of the spreadsheet ecosystem, considering a large and rich set of features. Many of these features are not covered by related work. Subsequently, we perform feature selection and train models for cell classification (refer to Section 4.3). Furthermore, we propose an optional post-processing step, which is discussed separately, in Chapter 5. This step reviews the classification results and attempts to get rid of obvious errors (denoted with an exclamation mark in Figure 4.1). Finally, in this chapter, Section 4.4, we discuss our methods for grouping cells into layout regions.

4.2 FEATURE EXTRACTION

As mentioned in Chapter 3, in this thesis we consider Microsoft Excel files. Nevertheless, most of the features listed below are applicable for other spreadsheet applications as well. We use the Python library *openpyxl*[1] [64], to process Excel files and extract features from non-empty and non-hidden cells. Some of these features (e.g. styling and font) are directly accessible via the Class attributes of this library. Others require additional custom implementation. This includes all formula, reference, and spatial features. As well, as it includes a large portion of content features.

In this thesis, we incorporate and extend the features proposed by related work [6, 11, 29, 45]. Nevertheless, we have introduced new features for all the considered categories (refer to Sections 4.2.1 to 4.2.6). This is especially true for formula, reference, spatial, and geometrical features.

Lastly, we introduce a naming convention for the features listed in the following section. We mark Boolean features with '?', at the end of their name. Similarly, we use '#' for numeric features. The rest, not explicitly denoted with a special character, are nominal (categorical) features.

4.2.1 Content Features

These features describe the cell value but not its styling. We consider the content type of the cell: *numeric*, *string*, *boolean*, *date*, and *error*. Note, for cells having formulas we use their output (result) to determine the actual content type of the cell. Furthermore, we define features that apply only to certain content types[2]. For instance, with regard to numeric cells, we check if the value is within the year range $[1970, 2020]$. In this way, we distinguish values that could be interpreted as dates. For string cells, we record the length of the value and the number of tokens (separated by white spaces) that compose it. The value of these features is always one for non-string cells. Below, we provide the complete list of content features[3].

- `CONTENT_TYPE`: The cell value can be *numeric*, *string*, *boolean*, *date*, or *error*.
- `IS_FLOAT?`: True, if value is a float. This feature is a specialization for numeric cells.
- `IS_YEAR_RANGE?`: True, if cell value is an integer in the range $[1970, 2020]$.
- `VALUE_LENGTH#`: The number of characters in the value. Set to 1, for non-string cells.
- `VALUE_TOKENS#`: The number of tokens in the value, separated by white spaces. Set to 1, for non-string cells.
- `IS_LOWER?`: True, if string value is in lower case.
- `IS_UPPER?`: True, if string value is in upper case.
- `IS_CAPITALIZED?`: True, if string value is in title case.

[1] The *openpyxl* library specializes in the .xlsx format. Therefore, we manually converted all .xls files in our datasets into .xlsx.

[2] We set the value by default to *False*, for all the other (non-applicable) content types

[3] We omit white spaces when calculating the value for the content features, with the exception of the following: `VALUE_LENGTH`, `VALUE_TOKENS`, and `IS_INDENTED`

- IS_ALPHA?: True, if value contains only alphabetic characters.
- IS_ALPHANUMERIC?: True, if value contains both alphabetic and numeric characters.
- CONTAINS_SPECIAL?: True, if value contains special symbols.
- FIRST_IS_SPECIAL?: True, if the first character is a special symbol.
- IS_NUMBERED_LIST?: True, if first characters are numbers followed by '.' or ')'.
- MULTIPLE_PUNCTUATIONS?: True, if the string contains > 1 punctuation characters. Here, we omit the colon character, which is covered by separate features, below.
- CONTAINS_COLON?: True, if string value contains a colon character.
- LAST_IS_COLON?: True, if the last character of the value is a colon.
- IS_INDENTED?: True, if indentations or tab spaces come before the cell value.
- CONTAINS_TOTAL?: True, if value contains word "total", regardless of case.
- CONTAINS_NOTES?: True, if value contains word "note"/"notes", regardless of case.
- IS_NA?: True, if cell value is equal to the string "n/a" or "na", regardless of case.
- IS_EMAIL?: True, if cell value matches regular expression for emails.

4.2.2 Style Features

In addition to content features, the style of the cell can provide valuable information for the classification process. For instance, certain build-in styles (e.g., *currency* and *percentage*) are often associated with Data and Derived cells. Moreover, for cells that are merged the most common layout functions are Note, Title, and Header. Below, we list the considered style features.

- HRZ_ALIGNMENT: Type of horizontal alignment; typically *left*, *right*, or *center*.
- VRT_ALIGNMENT: Type of vertical alignment; typically *top*, *bottom*, or *center*.
- HAS_FILL_COLOR?: True, if the cell is filled with a (back- or foreground) color.
- FILL_PATTERN: Excel supports several patterns, e.g., dots, and stripes.
- NUMBER_FORMAT: Cell value can be formatted as *general*, *fraction*, *scientific*, etc.
- BUILD_IN_STYLE: We consider four styles: *normal*, *percentage*, *currency*, and *comma*.
- BORDER_{TOP,BOTTOM,LEFT,RIGHT}: We create four separate features to encode the type of the top, bottom, left, and right border of the cell. Excel supports in total 14 border types, e.g. *dashed*, *dotted*, and *double*.
- IS_MERGED?: True, if cell is merged, i.e., multiple cells together form a larger one.

4.2.3 Font Features

The features in this group describe the font of the cell value. We have considered various aspects of the font, such as its size, effects, style, and color. In particular, the features *font_size* and *is_bold* are the most informative. For instance, it is common to find Headers and Titles that are bold. Moreover, the font size might differ depending on the layout function of the cell. For instance, we notice larger font sizes for Titles.

- `HAS_FONT_COLOR?`: True, if the font has color other than the default one, i.e., black.
- `FONT_SIZE#`: The size of the font can vary from 1 to 409.
- `IS_BOLD?`: True, if bold font is used.
- `IS_ITALIC?`: True, if italic font is used.
- `IS_UNDERLINE?`: True, if cell value is underlined.
- `OFFSET_TYPE`: Can be *superscript*, *subscript* or *none*.

4.2.4 Formula and Reference Features

These features explore the Excel formulas and their references in the same worksheet (i.e, intra-sheet references). The observation is that formulas are mostly found in Data or Derived cells. Moreover, cells referenced by formulas are predominantly Data. Nevertheless, below we have grouped formulas and references into several categories. In this way, we attempt to record their use in a much more detailed and accurate manner.

Note in the list below the special case of *shared formulas*. This occurs when multiple cells use the same exact formula (composed of one or more functions), but with different arguments. Excel encodes these cases in the newer OOXML format (see Section 2.1.2). We capture this feature here and use it as well for spatial features.

Last, with regard to referenced cells, we parse the formulas and identify the specific function that made the reference (i.e., used this reference as argument). This means we determine the type of the reference based on the individual function, not the overall formula. Therefore, the type can not be *mixed* (refer to the list below).

- `IS_SHARED_FRML?`: True, if this cell uses the same formula as other cells, but with different input (i.e., arguments)
- `IS_AGGR_FRML?`: True, if all the functions in the formula perform aggregations, e.g., *SUM*, *AVERAGE*, and *PRODUCT*.
- `HAS_AGGR_FUNC?`: True, if at least one of the functions performs aggregation.
- `IS_SIMPLE_NUM_AGGR?`: True, if instead of functions the formula uses simple operators, such as +, -, /, and *. Moreover, all arguments are numbers.
- `IS_SIMPLE_RNG_AGGR?`: True, if the formula makes use of only simple operators, but in this case, the arguments are references to other cells.
- `IS_DATE_FRML?`: True, if all the functions in the formula perform date operations, e.g., *TODAY*, *HOUR*, and *DATE*.

- `IS_LOGIC_FRML?`: True, if all the functions in the formula perform logic operations, e.g., *IF*, *AND*, *NOT*, etc.
- `IS_TEXT_FRML?`: True, if all the functions in the formula perform text operations, e.g., *CONCATENATE*, *LEFT*, and *RIGHT*.
- `IS_LOOKUP_FRML?`: True, if all the functions in the formula perform lookup operations, e.g., *MATCH*, *VLOOKUP*, and *HLOOKUP*.
- `IS_MATH_FRML?`: True, if all the functions in the formula perform math operations, e.g., *ABS*, *SQRT*, and *ROUND*.
- `IS_OTHER_FRML?`: True, if the functions in the formula do not correspond to the above categories.
- `IS_MIXED_FRML?`: True, if the functions in the formula are of a different type.
- `REFENCE_TYPE`: When a cell is referenced, we record the type of the function that made the reference. There are nine options: *aggr*, *simple*, *date*, *logic*, *text*, *lookup*, *math*, *other*, and *none*.

4.2.5 Spatial Features

In this section, we consider features that describe the location and neighborhood of a cell. The former refers to the relative location in the used area (i.e., the minimum bounding box that encloses all non-empty cells) of the sheet. Specifically, we record if the cell is in the first or last rows/columns of the used area. Moreover, we collect features from the neighbors of the cell. Here, we consider the eight immediate neighbors: top *(t)*, top-left *(tl)*, top-right *(tr)*, left *(l)*, right *(r)*, bottom *(b)*, bottom-left *(bl)*, and bottom-right *(br)*. They are illustrated in Figure 4.2.

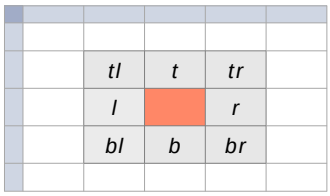


Figure 4.2: The Immediate Neighborhood of a Cell

Having this definition for the neighborhood[4], we introduce features that record how similar (or dissimilar) the cell is to its neighbors. For instance, we compare the style[5], content type, and formulas. Note, when the neighbors are empty or outside of the sheet used area, we introduce special values for the considered features. For Boolean (spatial) features, we set the value to *False*. While for categorical features we introduce two special values: *empty* and *out*.

Another spatial feature is the distance from the nearest non-empty neighbor. Therefore, these neighbors are not necessarily adjacent. Here, we focus only on four directions: top, left, right, and bottom. The horizontal distances, for top and bottom neighbors, are

[4]For merged cells, the *t*, *b*, *l*, and *r* neighbors come from the (smallest) middle column/row.
[5]To compare the styles of different cells we use the attribute *style_id*, as provided by *openpyxl* library [64].

measured in number of rows (i.e., $|cell\_row\_number - neighbor\_row\_number|$)[6]. While, for left and right neighbors, the distance is measured in number of columns. In cases where there does not exist a non-empty neighbor in a direction, the distance is set to *zero*.

Note we omit all hidden[7] rows and columns of a sheet for the calculation of spatial features. In this way, we capture exactly what the user sees. In other words, the classifier learns to interpret only what the Excel user decided to make visible.

Last, for brevity, in the list below we use a naming convention. Values in the curly brackets denote variations of a feature. In other words, we create a separate feature for each value in the curly brackets.

- `IS_FIRST_{ROW,COL}?`: True, if cell is in first row/column of the used area.
- `IS_LAST_{ROW,COL}?`: True, if cell is in last row/column of the used area.
- `IS_SECOND_{ROW,COL}?`: True, if cell is in second row/column of the used area.
- `IS_PENULT_{ROW,COL}?`: True, if cell is in penultimate row/column of the used area.
- `MATCHES_{T,TL,TR,L,R,B,BL,BR}_STYLE?`: True, if the neighbor has the same style as the cell.
- `MATCHES_{T,TL,TR,L,R,B,BL,BR}_TYPE?`: True, if the neighbor has the same content type as the cell.
- `{T,TL,TR,L,R,B,BL,BR}_CONT_TYPE`: The content type of the neighbor. It can take one of the following values: *numeric, string, boolean, date, error, empty,* and *out*.
- `DIST_{T,L,R,B}_NEIGHBOR#`: Distance from the nearest non-empty neighbor.
- `{T,TL,TR,L,R,B,BL,BR}_IS_FRML?`: True, if neighbor contains any formula.
- `{T,TL,TR,L,R,B,BL,BR}_IS_AGGR?`: True, if neighbor is an aggregation formula.
- `{T,TL,TR,L,R,B,BL,BR}_IS_SHARED?`: True, if neighbor uses the same formula as the cell. For more details on shared formulas, refer to Section 4.2.4.
- `{T,TL,TR,L,R,B,BL,BR}_IS_MERGED?`: True, if neighbor is merged cell.
- `{T,TL,TR,L,R,B,BL,BR}_IS_TOTAL?`: True, if neighbor contains the words "total", regardless of case.
- `{T,TL,TR,L,R,B,BL,BR}_IS_NOTES?`: True, if neighbor contains the words "notes" or "note", regardless of case.
- `{T,TL,TR,L,R,B,BL,BR}_IS_SPACES?`: True, if the value of the neighbor is indented.

We cover the whole neighborhood, even though we do not expect it to be informative for all features above. Later, we use automatic feature selection techniques, in Section 4.3.4, to filter out "weak" features. In this way, we avoid human bias in the selection process.

[6] To calculate the distance for merged cells, we use the minimum row for top neighbors and the maximum row for the bottom neighbors.

[7] Excel considers rows/columns hidden when their height/width is *zero*.

4.2.6 Geometrical Features

The final set of features deals with geometrical aspects of cells. In the list below, the first five features are relevant for merged cells. However, the last two apply to all cells. To calculate them, we sum the width/height[8] for the columns/rows of the cell.

- `CELL_AREA#`: The number of individual cells. For merged cells, this number is > 1.
- `CELL_N_ROWS#`: The number of rows that the cell covers. For merged cells, this number is > 1.
- `CELL_N_COLS#`: The number of columns that the cell covers. For merged cells, this number is > 1.
- `CELL_IS_VRT?`: True, if the $n\_rows > n\_cols$.
- `CELL_IS_HRZ?`: True, if the $n\_rows < n\_cols$.
- `CELL_HEIGHT#`: Sum the widths of all columns the cell covers.
- `CELL_WIDTH#`: Sum the heights of all rows the cell covers.

4.3 CELL CLASSIFICATION

We use supervised learning techniques to train classification models that predict the layout function of individual cells in a sheet. Therefore, we extract all the aforementioned features for each annotated cell. Subsequently, given the features and the assigned labels, we consider three different algorithms for classification, discussed in Section 4.3.2. We attempt to find the optimum configuration for each algorithm. Therefore, we perform under-sampling (Section 4.3.3), feature selection (Section 4.3.4), and parameter tuning (Section 4.3.5).

For these experiments, we consider both annotated datasets[9], from Chapter 3 with the adjustments discussed in Section 3.4.1. Initially, we train classification models individually for each one of them. Later, we combine these two datasets, to test whether this contributes to better classification performance. The results from the experimental evaluation of the trained models are presented in Section 4.3.6.

In the next sub-section, we begin by discussing the considered datasets. Specifically, we provide the distribution of the annotation labels, this time in number of cells. Subsequently, we proceed by discussing the tuning and training procedure for the classification models. The overall structure of our discussion is illustrated in Figure 4.3.

[8] In Excel, column widths are measured in units of 1/256th of a standard font character width. While row heights are measured in points.

[9] Hereinafter, we refer to the annotated samples from Enron and Fuse as *datasets*, in order to avoid confusion with the sub-samples discussed in Section 4.3.3

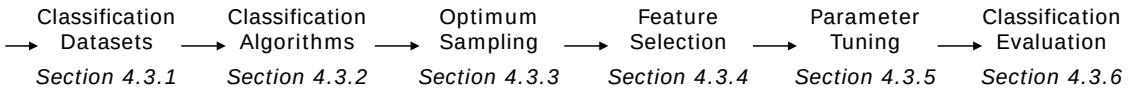


Figure 4.3: Training Cell Classifiers

4.3.1 Classification Datasets

In Chapter 3, we analyze two datasets, one extracted from the Enron corpus and the other from the Fuse corpus. We consider both of them for supervised learning. However, as discussed in Section 3.4.1, we exclude a small number of annotated sheets. Specifically, both datasets contain sheets that are outside of this thesis' scope. In addition to this, we omit four outlier sheets, one from Fuse and three from Enron. They contain an exceptionally high number (≥ 34) of tables. Thus, there is a risk that these sheets bias the overall approach.

Note, as shown in Appendix A, these omissions do not alter the findings from Chapter 3. After all, we exclude only 40 files from the annotated Enron dataset, bringing the number of considered sheets to 814. For the annotated Fuse dataset, we omit only 10 files, leaving 274 for experimental evaluation.

In Tables 4.1 and Tables 4.2 we provide the distribution of annotation labels in number of cells, respectively for Enron and Fuse dataset. It is clear that *Data* cells dominate in both datasets. This creates a strong imbalance, which can affect the supervised learning process [122]. One of the typical ways to tackle imbalance is to randomly under-sample the majority class [122]. In Section 4.3.3 we have experimented with different sample sizes, in order to determine the optimal one for each classification algorithm.

Table 4.1: Cell Counts per Annotation Label in Enron Dataset

Data	Other	Header	Derived	GHead	Title	Note
1.3M	69.2K	15.1K	13.0K	1.7K	0.9K	0.8K
92.60%	5.08%	1.10%	0.96%	0.13%	0.07%	0.06%

Table 4.2: Cell Counts per Annotation Label in Fuse Dataset

Data	Other	Header	Derived	GHead	Note	Title
1.0M	32.5K	5.4K	4.4K	1.4K	0.5K	0.3K
95.75%	3.11%	0.51%	0.42%	0.13%	0.04%	0.03%

Note, in the subsequent sections, we discuss these two datasets separately. Specifically, we tune, train, and evaluate classification models for each one of them. Nevertheless, at the end of this procedure (in Section 4.3.6), we provide the best classification results for the combined dataset, as well.

Furthermore, cells annotated as *Other* require special treatment. First, we notice that there is no label equivalent to *Other* from related work. While, for the remaining six labels, we

can find similar notions. Second, we observe that this label is very versatile. Ultimately, this variety affects the classification accuracy. Thus, we have trained classifiers with and without *Other* cells. This allows a more direct comparison with related work. As well as, it provides the classification accuracy for the remaining labels, independently of *Other*. The experiments with six labels are discussed in all the subsequent sections. However, those with seven labels are only discussed in Section 4.3.6. Regardless, the training procedure remains the same, for both cases.

4.3.2 Classifiers and Assessment Methods

Here, we discuss the classification algorithms used throughout the Sections 4.3.3 - 4.3.6. Additionally, we define the methods and measures used for assessing the performance of the classification models.

Classifiers To train models for layout inference, we consider three well-known classification algorithms. We utilize the implementations from the Python library *scikit-learn* [101]. Concretely, we experiment with *Random Forest*, which is an ensemble learning method [24]. In simple terms, *Random Forest* constructs multiple decision trees and subsequently combines their output to make the final decision (prediction). In addition, we use *Logistic Regression*, which models the probabilities for the classes based on the given instances and features [21]. Finally, we consider support-vector machines (*SVMs*), which outputs maximum-margin hyperplanes that optimally separate the given instances in high-dimensional space [21]. However, traditional *SVMs* can not handle large datasets, i.e., such as the ones considered by this thesis. Instead, we use the *SGDClassifier* from *scikit-learn*, which "implements regularized linear models with stochastic gradient descent (*SGD*)" [101]. We use *hinge* as loss function, which is the same loss function as for *LinearSVM*. In this way, we get an efficient approximation of traditional *SVMs*, on our large datasets of annotated cells.

Cross-Validation To evaluate the performance of the trained models, we use k-fold cross-validation [132]. Each iteration we test on one of the folds, while the rest $(k-1)$ are used for training. We use either $k=5$ or $k=10$ folds, depending on the experiment. This is explicitly stated in the following sections.

Note, cells from an annotated sheet can either be used for training or for testing, but not simultaneously for both. This means, for cross-validation we partition by sheet, i.e., all cells of a sheet go to only one of the folds. As pointed out by [11], this method allows testing on unseen examples. However, we go one step further, by additionally balancing the number of multi- and single-table sheets per fold. In this way, we ensure that folds have a similar composition. These two steps, partitioning by sheet and balancing by tables (*single/multi*), lead to stronger models that are able to generalize better.

Feature Scaling The *Logistic Regression* and *SGDClassifier* are sensitive to the scale differences in the training features. Therefore, it is common practice to bring these features under the same scale [63]. Here, we perform standardization, which results in features having zero mean ($\mu=0$) and unit variance ($\sigma=1$).

Moreover, to avoid information leakage [80], we scale independently for each iteration (fold) of the cross-validation. Concretely, we use *StandardScaler* from *scikit-learn*. We fit a scaler on the $(k-1)$ folds reserved for training. Then we use this scaler (i.e., the discovered parameters) to transform both training and testing folds.

Under-Sampling As noted in Section 4.3.1, one reason to under-sample is the considerable imbalance in annotated datasets. Another reason is to reduce computational overhead. Although the considered algorithms can handle a large number of training instances, it is always more efficient to work with smaller datasets. Ultimately, under-sampling allows us to run more experiments and potentially finding better models.

For the under-sampling step, we follow the same logic as for feature scaling. Every iteration of the cross-validation, we under-sample from the $(k-1)$ folds reserved for training. However, we use all instances from the test fold to evaluate the model. Therefore, even though we train in the under-sampled dataset, we always test on the whole dataset.

Notice that we first perform under-sampling and then we proceed with feature scaling. Intuitively, we need to fit the scaler on the dataset (i.e., the under-sample) used for training. Therefore, under-sampling should precede feature scaling.

Performance Metrics To measure the performance of the classification models, we consider several standard metrics. For the most part, in the subsequent sections, we report the results using the *F1-score* (also known as *F-measure*) [132]. This score is defined as the harmonic mean of *Precision* and *Recall* [132]. We report the *F1-score* for the individual classes and overall. For the latter, we consider two variations: the *Weighted-F1* and the *Macro-F1*. In both cases, the overall *F1-score* is calculated by averaging the scores from the individual classes. However, in the first case, the average is weighted by the number of instances per class. While in the second case the average is not weighted. Finally, in Section 4.3.6, we provide detailed results for the best models. This includes the *F1-score* for the individual classes.

4.3.3 Optimum Under-Sampling

As pointed out in Section 4.3.1, both datasets are severely imbalanced. Therefore, in this section, we attempt to under-sample the datasets, in order to find a more favorable ratio between the annotation labels. Additionally, under-sampling removes some of the computational overhead, since the algorithms process a smaller number of instances. The experiments discussed below apply only to six labels. Cells labeled as *Other*, are omitted from the datasets. Nevertheless, we report results for seven labels in Section 4.3.6.

In this section, we use the *RandomUnderSampler* from *scikit-learn* [101]. This implementation draws random samples with replacement. As the name suggests, these samples are always smaller than the original dataset. Concretely, we perform focused under-sampling for *Data*, i.e., the majority class. Therefore, for the other *five* classes, we keep all the instances. While, for *Data* we experiment with sampling sizes in the range $[15K - 240K]$, gradually increasing at each step by $15K$.

We run the experiments with the classification algorithms, discussed in Section 4.3.2. We keep the default values for the parameters of these algorithms, as specified in *scikit-learn* v0.22.1. However, for Random Forest we additionally run experiments setting the $bootstrap$ parameter to $false$. We consider this alternative configuration since the *RandomUnderSampler* already performs bootstrapping (i.e., random sampling with replacement). Therefore, in the end, we test four classifiers: Random Forest with $bootstrap = true$ (RF_BT), Random Forest with $bootstrap = false$ (RF_BF), Logistic Regression (LOG), and SGDClassifier with $hinge$ as loss function (SGD).

At each step (i.e., sampling size) we perform 5-fold cross-validation, to assess the performance of the classifiers. This process is sketched with pseudocode in Algorithm 4.1, lines

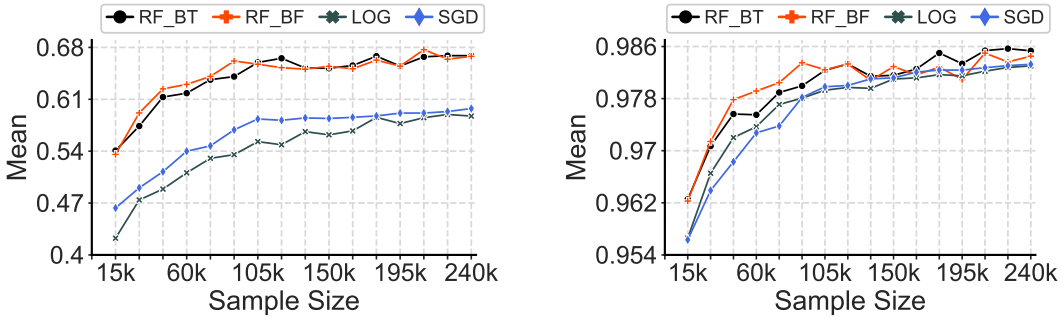
Algorithm 4.1: Finding the Optimum Under-Sample Size

Input: k : number of folds, r : number of sampling rounds, $start$: minimum size for sampling , end : maximum size for sampling, $step$: amount to increase sampling size, $annotations$: feature vectors for annotated cells, $classifier$: RF_BT, RF_BF, LOG, *or* SGD
Output: hof : the hall of fame, i.e., the optimum sampling size with its averaged Macro-F1 and Weighted-F1

```
1  begin
2      k ← 5; r ← 3;
3      start ← 15000; end 240000; step ← 15000;
4      size ← start;
5      hof ← null ;
6      while size ≤ end do
7          macros ← initializeArray(r) ; // array of size r to store Macro-F1 per sampling round
8          weighted ← initializeArray(r) ;                    // same as above for Weighted-F1
9          for i ∈ {1, ..., r} do
10             folds ← createCrossValidationFolds(k, annotations) ;          // partition by sheet
11             results ← ∅ ;                                  // collect cross-validation predictions
12             for j ∈ {1, ..., k} do
13                 testCells ← folds[j];
14                 trainCells ← getTraningCells(folds, j) ; // merge the other k-1 folds for testing
15                 trainSample ← getRandomUnderSample(trainCells, size);
16                 results ← results ∪ trainAndPredict(classifier, trainSample, testCells)
17             macros[i] ← calculateMacroF1(results, annotations);
18             weighted[i] ← calculateWeightedF1(results, annotations);
19         avg_m ← calculateMean(macros);
20         avg_w ← calculateMean(weighted);
21         hof ← updateHallOfFame(size, avg_m, avg_w) ;      // update if current results are better
22         size ← size + step
23     return hof
```

10 – 18. Note, at line 15, sampling occurs for each iteration of the cross-validation. We sample from the training folds while testing on the complete (i.e., not re-sampled) left-out fold. At the end of a cross-validation run, we calculate Macro-F1 and Weighted-F1 to measure the classification performance.

Furthermore, to ensure statistical significance we perform the aforementioned process three times, per sampling size. This can be seen in lines 9 – 22, of Algorithm 4.1. Essentially, we run three distinct 5-fold cross-validations, with unique seed for randomization. We average Macro-F1 and Weighted-F1 from the three runs, as illustrated in lines 19 – 20.

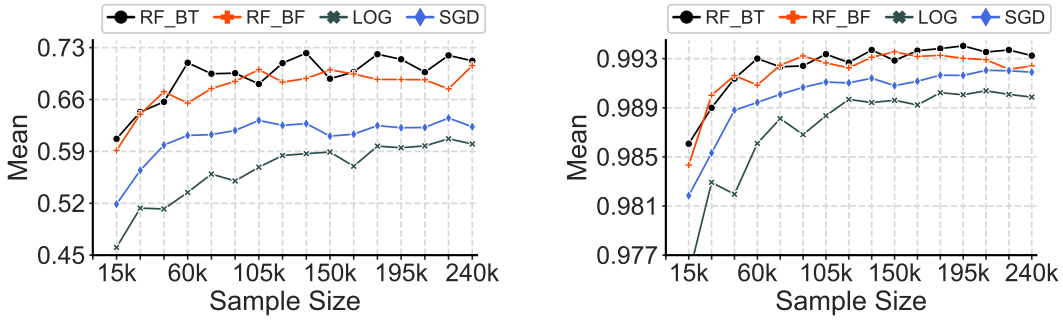


(a) Average Macro-F1 per Sampling Size

(b) Average Weighted-F1 per Sampling Size

Figure 4.4: Analysis of Sampling Sizes for Enron

In Figure 4.4 and Figure 4.5, we respectively show the results for Enron and Fuse dataset. We plot the average Macro-F1 and Weighted-F1 per sampling size (step), for each algorithm. In both datasets, we observe that the general trend is positive, as the sampling size increases. Arguably, for small sizes, there is a loss of information. In other terms, it is useful to have some degree of imbalance in the datasets.



(a) Average Macro-F1 per Sampling Size

(b) Average Weighted-F1 per Sampling Size

Figure 4.5: Analysis of Sampling Sizes for Fuse

We select the optimum sampling size based on both Macro-F1 and Weighted-F1 (see Algorithm 4.1, line 21). For this, we rank the resulting scores, for each metric. Then, we select the sampling size that provides the best combination of ranks. In cases where there is no clear winner, we favor Macro-F1 over Weighted-F1. Essentially, we prioritize sampling sizes that yield good scores across all classes. Table 4.3 and Table 4.4 display the selected sampling sizes per dataset and algorithm. These are accompanied by the respective values for the performance metrics.

Table 4.3: Enron Optimal Sampling

Classifier	Sample Size	Macro-F1	Weighted-F1
RF_BT	$240K$	0.669	0.985
RF_BF	$210K$	0.677	0.985
LOG	$225K$	0.590	0.983
SGD	$240K$	0.598	0.983

Table 4.4: Fuse Optimal Sampling

Classifier	Sample Size	Macro-F1	Weighted-F1
RF_BT	$180K$	0.721	0.994
RF_BF	$240K$	0.706	0.992
LOG	$225K$	0.607	0.990
SGD	$225K$	0.635	0.992

4.3.4 Feature Selection

As outlined in Section 4.3.3, we have experimentally determined the optimum undersampling size per algorithm and dataset. In this section, we proceed with feature selection. The goal of this step is to remove features that are not informative or even noisy [42]. This has the potential to improve the performance of the trained models. Moreover, with a smaller number of features, training and testing become faster.

In total, we have considered 159 distinct features for cell classification, which are thoroughly discussed in Section 4.2. Below, we outline the steps we took to select the most relevant. We begin with binarization, a pre-processing step, and then we apply feature selection methods.

Binarization The binarization process (also known as *one-hot-encoding*) replaces the categorical features with multiple Boolean features. Specifically, it creates a new Boolean feature for every distinct categorical value. It achieves this by identifying all instances in the dataset that have the selected (categorical) value. For these instances, the corresponding Boolean feature is set to *true*. For the remaining instances, which do not have the selected value, the Boolean feature is set to *false*. This process is repeated until all categorical values are encoded into distinct Boolean features.

Binarization is required for Logistic Regression and SGDClassifier. These linear models do not interpret correctly categorical and ordinal features. For this reason, we convert them to Boolean features, which are more appropriate. Additionally, binarization makes feature selection more accurate. It is easier to detect the relevant categorical values, once they are treated separately. We find that this is useful for cell style features (see Section 4.2.2). Excel provides many styling options, but the majority of them are not frequently used. We can easily test this, after binarization.

Concretely, we have binarized all the categorical features describing the cell style (see Section 4.2.2). We apply binarization on the feature `REFERENCE_TYPE`, from Section 4.2.4. Lastly, we binarize categorical features relating to the content type of the cell and its neighbors (refer to Section 4.2.1 and Section 4.2.5).

After binarization, we get 256 and 237 features, respectively for Enron and Fuse dataset. Note, the numbers differ because the values for the styling features differ. We record only those styling options that occur in the annotated sheets. We observe that Enron sheets are more diverse when it comes to style. Therefore, the respective (categorical) style features have a larger number of distinct values.

Eliminating Features by Frequency of Occurrence We initially eliminate Boolean features based on how often they occur, i.e., we count the instances for which they are *true*. We perform this analysis on the sheet and cell granularity, separately for Enron and Fuse dataset. Specifically, we remove Boolean features that occur in ≤ 10 sheets. This is motivated by the fact that in Section 4.3.6 we use 10-fold cross-validation to do the final evaluation of the classification models. Therefore, assuming uniform distribution, we require that at least 1 sheet per fold exhibits a specific characteristic (feature). Additionally, we eliminate Boolean features that occur for $\leq 0.01\%$ of the cells. We set a low threshold considering that some classes have a small number of instances, e.g., *Title* and *Note*. All in all, the above-mentioned operations reduced the number of features to 204 and 179, respectively for Enron and Fuse dataset.

Eliminating with RFECV Subsequently, we select the most relevant features using the *RFECV* option from the *scikit-learn* library [101]. The *RFECV* (Recursive Feature Elimination and Cross-Validation) determines the optimum set of features for a given classification algorithm. Every iteration, *RFECV* eliminates the specified number of features, among those having the lowest importance[10]. Then, it uses cross-validation to train/test

[10]The feature importance is calculated by the classification algorithms, rather than *RFECV*. Each algorithm has a different scoring mechanism

a classifier on the remaining features. Following multiple iterations, *RFECV* outputs the feature set giving the highest classification performance.

We run *RFECV* once per classification algorithm and annotated dataset. We set to 10 the number of features to be eliminated every iteration. Moreover, the same as in Section 4.3.3, we perform 5-fold cross-validation to test the classification performance. However, in this case, we use only *Macro-F1* to measure the performance. *RFECV* does not allow multiple classification metrics.

Note, in Section 4.3.3 we determined the optimum under-sample size for each algorithm and dataset. Here, we make use of these sizes. Specifically, we prepare in advance the under-samples used during the *RFECV* cross-validations. For this, we had to create a custom *iterable*, which we pass to the *cv* parameter of the *RFECV* object.

Table 4.5: Number of Selected Features per Algorithm for Classification with 6 Labels

Dataset	**Nr. Features Before**	**Nr. Selected Features**			
		RF_BT	**RF_BF**	**LOG**	**SGD**
Enron	204	154	64	184	164
Fuse	179	99	129	79	89

Feature Selection Results Table 4.5 provides the results from the *RFECV* runs, for each algorithm. In general, we notice that the number of selected features tends to be lower for the Fuse dataset, compared to the Enron dataset.

In addition, we have determined the most relevant features, overall for the two datasets. These are shown in Table 4.6 and Table 4.7, respectively for Enron and Fuse. Note, the features are not listed in order of importance. Instead, we grouped them thematically. Moreover, many of these features are binarized. For those relating to the content type, we use the following abbreviations: string (S), number (N), out (*), empty (∅).

Table 4.6: Top 20 Features for Enron

Enron Dataset	
CONTENT_TYPE=N? CONTENT_TYPE=S?	IS_FLOAT_VAL?
TL_CONT_TYPE=*? T_CONT_TYPE=*? T_CONT_TYPE=N? TR_CONT_TYPE=*? L_CONT_TYPE=*? R_CONT_TYPE=*? R_CONT_TYPE=N? BL_CONT_TYPE=*? B_CONT_TYPE=*? B_CONT_TYPE=∅?	TR_MATCH_TYPE? R_MATCH_TYPE? B_MATCH_TYPE?
	T_MATCH_STYLE? BL_MATCH_STYLE? B_MATCH_STYLE?

Table 4.7: Top 20 Features for Fuse

Fuse Dataset	
CONTENT_TYPE=N? CONTENT_TYPE=S?	IS_FLOAT_VAL?
TL_CONT_TYPE=*? TL_CONT_TYPE=S? T_CONT_TYPE=S? L_CONT_TYPE=S? L_CONT_TYPE=N? R_CONT_TYPE=S? R_CONT_TYPE=N? BL_CONT_TYPE=*? B_CONT_TYPE=∅?	TL_MATCH_TYPE? T_MATCH_TYPE? R_MATCH_TYPE?
	TR_MATCH_STYLE? BL_MATCH_STYLE? B_MATCH_STYLE?
	HRZ_ALIGN=CENTER? IS_BOLD?

In Table 4.6 and Table 4.7, we observe that most of the features describe the cell neighborhood (see Section 4.2.5). In addition, the majority of the top 20 relate to the content type, of the cell itself or that of the neighbors. There are fewer features describing the style.

The procedure to determine the aforementioned top 20 features goes as follows. We consider the results from *RFECV*. Namely, we take into account the individual set of features selected for RF_BT, LOG, and SGD. We omit RF_BF since the classification model is very similar to RF_BT. Subsequently, for each dataset, we find the features in common, which were selected for all three considered algorithms. Then, we study the feature rankings (importances). The lower is the rank number, the higher is the importance of a feature. Thus, we can sum the ranks from the individual algorithms, to determine the overall importance of a feature. The lower the sum, the better is the overall importance.

4.3.5 Parameter Tuning

In the previous section, we determined the optimum sampling size and number of features, per algorithm and dataset. Here, we proceed to tune the parameters of the individual algorithms. This step has the potential to improve further the classification performance. Notice that in this section, the same as in the previous ones, we discuss experiments for 6 annotation labels.

We make use of *GridSearchCV*, as provided by the Python library *scikit-learn* [101]. This implementation performs parameter tuning, via exhaustive search. First, for each one of the considered algorithms, we select the parameters to tune. Subsequently, we specify the list (range) of values to be tested for the selected parameters. Then, *GridSearchCV* will try all the combinations. Specifically, it runs cross-validation to evaluate the performance of each unique combination of parameter values. Here, we use two scores to measure the performance: Macro_F1 and Weighted_F1. As in Section 4.3.3, we rank the results from *GridSearchCV*, to determine the best combination of parameter values.

We run the aforementioned experiments per algorithm and dataset. Note, we fix the sampling size and feature set, to what was experimentally determined in the previous sections. Additionally, we set the number of folds to $k = 5$, for cross-validation during the *GridSearchCV* run. Then, for each algorithm, we tune the specific parameters discussed below. For the *Random Forest* classifiers, namely RF_BT and RF_BF, we explore various combinations for the parameters *n_estimators, criterion,* and *max_features*. For the other two algorithms, *Logistic Regression* and *SGDClassifier*, we tune only the regularization parameter. Table 4.8 summarizes the results from these experiments.

Table 4.8: Optimal Parameters per Algorithm for Classification with 6 Labels

Classifier	Enron	Fuse
RF_BT	*n_estimators=100, criterion='entropy', max_features='sqrt'*	*n_estimators=200, criterion='entropy', max_features='sqrt'*
RF_BF	*n_estimators=400, criterion='entropy', max_features='log2'*	*n_estimators=50, criterion='gini', max_features='sqrt'*
LOG	*C=0.01*	*C=0.01*
SGD	*alpha=0.0001*	*alpha=0.0005*

4.3.6 Classification Evaluation

In this section, we report the results from the experimental evaluation of the optimized classifiers. As described in previous sections, we already determined the optimum under-sampling size (Section 4.3.3), set of features (Section 4.3.4), and parameter values (Section 4.3.5). Here, we train/test using these optimizations. Specifically, for the final assessment, we run three distinct 10-fold cross-validations per algorithm and dataset. These multiple cross-validations, with unique seed, ensure statistical significance. In the following sections, for each algorithm, we report the averaged results from the three runs.

The subsequent parts are organized as follows. We begin our discussion with the classification results from the experimental evaluation with 6 labels. Then, we discuss the results for 7 labels. We show that when including the label *Other* the classification performance decreases overall and individually for the remaining 6 labels. Subsequently, we combine the two annotated datasets, from Enron and Fuse. Indeed, we observe that training and testing in the combined dataset has a positive effect on classification performance. We conclude by comparing the best classification results from our experimental evaluation with those from related work.

Results for Six Labels

Using the pre-determined sampling sizes (Tables 4.3 and 4.4), feature sets (Table 4.5), and parameter values (Table 4.8) we evaluate the classification algorithms on 6 labels. As mentioned previously, for this final assessment we execute three distinct 10-fold cross-validation runs. The results discussed below are the averages of the three runs.

Moreover, we perform a few additional experiments exclusively with *Random Forest (RF)* classifiers: RF_BT ($bootstrap = true$) and RF_BF ($bootstrap = false$). The parameter values and selected features remain the same, i.e., the ones that were determined in the previous sections. The difference is in the way we run the cross-validations for the final assessment. Concretely, for the additional experiments, we use the complete training folds (i.e., *all* training instances), instead of the under-samples.

The results are shown in Table 4.9 and 4.10, separately for Enron and Fuse dataset. We measure the performance of the classifiers using the F1-score, per class and overall (i.e., *Macro-F1*). The best scores from each column are highlighted with bold font.

Overall, we get better classification performance in the Fuse dataset compared to the Enron dataset. This can be seen when we compare the *Marco_F1* values from Tables 4.9 and 4.10. As discussed in Section 3.3.2, in the Fuse dataset we find more regular sheet layout compared to the Enron dataset. This seems to play a role when it comes to cell classification. In other terms, it is easier for the classifiers to identify frequent patterns.

We also compare the performance for the individual classes. For the Fuse dataset, the trained models achieve higher F1-score for *Data, Derived, Title,* and *Note*. In particular, for the latter class, the difference is very evident. Nevertheless, for *Header* and *GHead* the F1-Scores are higher in the Enron dataset.

Moreover, we observe that the *Random Forest* classifiers, RF_BT and RF_BF, outperform the rest by a significant margin. They achieve higher F1-Score for all classes, in both datasets. The only exception is in Table 4.10. There, the *SGDClassifier* has the highest score for *Derived* class, when down-sampling *(Sample)* is used for training.

Table 4.9: F1-Score per Class for Enron with 6 Labels

Classifier		Data	Header	Derived	Title	Note	GHead	Macro_F1
Sample	*RF_BT*	**0.993**	0.862	0.544	0.689	0.497	**0.484**	0.678
	RF_BF	**0.993**	0.848	0.581	0.683	**0.504**	0.482	**0.682**
	LOG	0.992	0.746	0.559	0.606	0.413	0.336	0.606
	SGD	0.992	0.751	0.549	0.550	0.310	0.332	0.581
All	*RF_BT*	0.993	**0.895**	0.494	**0.698**	0.476	0.430	0.664
	*RF_BF**	**0.995**	0.892	**0.588**	0.690	0.484	0.443	**0.682**
Diff BT		+0.000	+0.033	-0.050	+0.009	-0.021	-0.054	-0.014
Diff BF		+0.002	+0.044	+0.007	+0.007	-0.020	-0.039	+0.000

Table 4.10: F1-Score per Class for Fuse with 6 Labels

Classifier		Data	Header	Derived	Title	Note	GHead	Macro_F1
Sample	*RF_BT*	**0.997**	0.812	0.610	**0.785**	0.753	0.282	0.707
	RF_BF	**0.997**	0.820	0.636	0.767	**0.760**	0.273	0.709
	LOG	0.996	0.720	0.562	0.718	0.522	0.158	0.613
	SGD	**0.997**	0.710	0.645	0.529	0.591	0.155	0.605
All	*RF_BT**	**0.997**	**0.840**	**0.675**	0.760	0.756	**0.322**	**0.725**
	RF_BF	**0.997**	0.823	0.613	0.764	0.749	0.182	0.688
Diff BT		+0.000	+0.028	+0.065	-0.025	-0.003	+0.040	+0.018
Diff BF		+0.000	+0.003	-0.023	-0.003	-0.011	-0.091	-0.021

Furthermore, we capture changes in F1-scores, during the final assessment, when different strategies are employed for training. Specifically, we compare training on *All* instances versus training on *Samples* (short for down-sampling). As shown in Table 4.9 and 4.10, we calculate the difference ($All - Sample$) in F1-scores for the respective Random Forest classifiers (namely for the RF_BTs and RF_BFs). In Table 4.9, for the Enron dataset, we notice that the best scores for *Notes* and *GHeads (GroupHeaders)* occur when down-sampling is used. Nevertheless, the other classes get better scores when training is performed on the complete cross-validation folds. In particular, we notice significant improvement for the *Header* class. In Table 4.10, for the Fuse dataset, we observe that the performance of RF_BF decreases when *All* instances are used for training. However, the opposite is true for the RF_BT classifier. In fact, with this classifier, we see substantial improvements for the *Header*, *Derived*, and *GHead* class. At the same time, the losses for the other classes are small or insignificant.

Based on the above discussion, we find that is more beneficial to perform training on *All* instances. Note that here we refer only to training during the final assessment. We still determine the optimum configuration (i.e., feature selection and parameter tuning) using under-samples. This is still useful since it reduces the computational overhead.

We emphasize that F1-scores for *Header* and *Data* are better when using *all* instances for training. This is favorable since these two classes are crucial for tasks that follow layout analysis, such as table identification (Chapter 6) and information extraction (Chapter 7). Therefore, for Enron, we choose $All\ \ RF\_BF$ as the best model since it additionally achieves a good Macro_F1 score. While, for Fuse, we select $All\ \ RF\_BT$. In Table 4.9 and 4.10, we have marked these classifiers with the star character '*'.

Finally, we observe that the selected classifiers yield F1-scores that emulate the Agreement Ratios from Table 3.1. In fact, the classifiers are able to perform similar or better than human annotators. Moreover, their classification performance is comparable to related work, as discussed in Section 4.3.6

Results for Seven Labels

In this section, we discuss the experimental evaluation with 7 labels, i.e., we additionally include the label $Other$. We consider only *Random Forest (RF)* classifiers, since they outperformed the rest by a significant margin, as was shown in the previous section.

To determine the best configuration for the considered classifiers, RF_BT ($bootstrap = true$) and RF_BF ($bootstrap = false$), once more we use the procedure discussed in the previous sections. Initially, we identify an optimum sampling size, then we select the best features, and finally, we tune the parameters. The resulting configurations are provided in Table 4.11. With regard to the optimum sampling size, we search in the range $[150K - 300K]$. We increased the lower bound since experiments in Section 4.3.3 showed that lower sizes are not beneficial. Moreover, we increased the upper bound, to account for the additional cells labeled as *Other*.

Table 4.11: Optimal Configuration for 7 Labels

Classifier		Parameters	Nr. Selected Features	Sampling Size
Enron	*RF_BT*	*n_estimators=400, criterion='entropy', max_features='sqrt'*	104	$285K$
	RF_BF	*n_estimators=50, criterion='gini', max_features='sqrt'*	204	$255K$
Fuse	*RF_BT*	*n_estimators=100, criterion='gini', max_features='log2'*	79	$240K$
	RF_BF	*n_estimators=100, criterion='entropy', max_features='sqrt'*	139	$255K$

For the final assessment, we again use three distinct 10-fold cross-validations and average the results. Moreover, we skip under-sampling and instead use *all* instances for training. As discussed in the previous section, the later is favorable since it yields better scores for most cases. In particular, it improves the scores for *Header* and *Data* classes, which play a crucial role during the table detection task.

In the first two rows of Table 4.12 and 4.13, we list the classification results for 7 labels, respectively for Enron and Fuse dataset. Again, we highlight the highest scores using

bold font. We observe that there is an evident difference in *F1-scores* when it comes to the class *Other*. In the Enron dataset, the classifiers are able to distinguish this class from the rest. In fact, the F1-score for *Other* is higher than that of *Title, Note,* and *GHead*. However, in the Fuse dataset, the classifiers perform poorly for the class *Other*. A closer look at the confusion matrices, from the distinct cross-validation runs, revealed that cells labeled as *Other* are almost always misclassified as *Data*. Furthermore, 96% of the misclassifications occur in 4 files, which contain a large number of *Other* cells. Therefore, the main reasons for these results seem to be errors and inconsistencies during the annotation phase.

Table 4.12: F1-Score per Class for Enron with 7 Labels

Classifier		Data	Header	Deriv	Title	Note	GHead	Other	M_F1
Seven	*RF_BT**	**0.977**	0.848	**0.547**	0.608	**0.339**	**0.385**	**0.624**	**0.618**
	RF_BF	0.975	**0.849**	0.461	**0.610**	0.333	0.371	0.591	0.599
Six	*RF_BT*	0.993	0.895	0.494	0.698	0.476	0.430	–	0.664
	RF_BF	0.995	0.892	0.588	0.690	0.484	0.443	–	0.682
Diff BT		-0.016	-0.047	+0.053	-0.090	-0.137	-0.045	–	-0.046
Diff BF		-0.029	-0.043	-0.127	-0.080	-0.151	-0.072	–	-0.083

Table 4.13: F1-Score per for Fuse with 7 Labels

Classifier		Data	Header	Deriv	Title	Note	GHead	Other	M_F1
Seven	*RF_BT*	**0.982**	0.827	**0.634**	**0.705**	0.671	**0.321**	0.005	**0.592**
	*RF_BF**	**0.982**	**0.843**	0.603	0.699	**0.685**	0.112	**0.006**	0.561
Six	*RF_BT*	0.997	0.840	0.675	0.760	0.756	0.322	–	0.725
	RF_BF	0.997	0.823	0.613	0.764	0.749	0.182	–	0.688
Diff BT		-0.015	-0.013	-0.041	-0.055	-0.085	-0.001	–	-0.133
Diff BF		-0.015	+0.020	-0.010	-0.065	-0.064	-0.070	–	-0.127

Moreover, in this section, we study how including the label *Other* impacts the classification performance for the rest of the labels. Thus, we consider the models from the previous section, where we assessed the performance for 6 labels while training on *all* instances. We calculate the difference in F1-score between the respective classifiers (*Seven - Six*), for the corresponding classes (i.e., skipping *Other*). The results are listed in the bottom rows of Table 4.12 and 4.13.

With few exceptions, the performance of the remaining 6 classes is negatively impacted by the *Other* class. For both datasets, *Note* and *Title* classes are affected the most. However, the negative impact is smaller in the Fuse dataset compared to the Enron dataset.

Finally, the same as in the previous section, we choose the best model for each dataset. Clearly, based on the *Macro_F1* (shortly denoted as *M_F1*), the *RF_BT* classifiers perform the best for 7 labels. However, for the Fuse dataset, *RF_BT* achieves as lower score than *RF_BF*, when it comes to *Header* class. For *Deriv* (short for *Derived*), *Title*, and *Note* the

difference is not significant. The main concern is with *GHead*, but this class is not as crucial as the *Header* class, with regard to table detection (refer to Chapter 6). Therefore, in the end, we favor the *RF_BF* for the Fuse dataset. In Table 4.12 and 4.13, we use the star character '*' to denote the selected classifiers. All in all, we consider their classification results in the subsequent sections.

Results for Combined Dataset

In an attempt to improve the classification performance, we combine the two datasets into one. We consider *Random Forest (RF)* classifiers, which proved to be the most performant in the previous experiments. We examine once more two separate scenarios: 6 and 7 labels. For each scenario, we optimize and train the classifiers. First, we determine the optimum configuration, and then we use three distinct 10-fold cross-validations to perform the final assessment. The averaged scores from these runs are discussed below.

Note, we ensure that the cross-validation folds are balanced. Specifically, the Enron and Fuse sheets are evenly distributed into the folds. This is in addition to balancing the single- and multi-table sheets, which was mentioned in Section 4.3.2.

The following tables provide the results from our experimental evaluation. Although optimization and training are done on the combined dataset, the classification performance is assessed separately for Enron and Fuse dataset. This allows us to compare with the best models from the previous sections. We distinguish these models by marking them with star character '*'. The other models, which are not marked, originate from the combined dataset. Furthermore, we use *Deriv* as a short form for *Derived*. Similarly, we denote *Macro_F1* shortly as *M_F1*.

Table 4.14: Results for 6 Labels when Training on the Combined Dataset

Classifier		Data	Header	Derived	Title	Note	GHead	M_F1
Enron	*RF_BF*	0.995	0.892	0.580	0.700	0.475	0.430	0.679
	*RF_BF**	0.995	0.892	0.588	0.690	0.484	0.443	0.682
Difference		+0.000	+0.000	-0.008	+0.010	-0.009	-0.013	-0.003
Fuse	*RF_BF*	0.998	0.883	0.710	0.815	0.777	0.242	0.737
	*RF_BT**	0.997	0.840	0.675	0.760	0.756	0.322	0.725
Difference		+0.001	+0.043	+0.035	+0.055	+0.021	-0.080	+0.012

In Table 4.14, we compare the classifiers for 6 labels. When we tested on the combined dataset, RF_BF proved to be the most performant. Thus, in Table 4.14, we consider only the results from this classifier to measure the difference to F1-scores from the previous sections. For Enron, we observe that scores decrease for some classes, but not by a significant margin. For Fuse, the results are much more positive. In fact, for the majority of classes, we see gains of up to 5.5%. However, we also lose 8% for the *GHead* class. Nevertheless, the gains for the rest of the classes outweigh the losses for the *GHead* class.

In Table 4.15, we present the results for 7 labels. In this case, the RF_BT classifier was the most performant, for the combined dataset. When comparing with the results from previous sections, we mostly observe gains. In the Enron dataset, the F1-scores increase

Table 4.15: Results for 7 Labels when Training on the Combined Dataset

Classifier		Data	Header	Deriv	Title	Note	GHead	Other	M_F1
Enron	*RF_BT*	0.978	0.852	0.554	0.609	0.349	0.403	0.624	0.624
	*RF_BT**	0.977	0.848	0.547	0.608	0.339	0.385	0.624	0.618
Difference		+0.001	+0.004	+0.007	+0.001	+0.010	+0.018	+0.000	+0.006
Fuse	*RF_BT*	0.982	0.863	0.730	0.718	0.692	0.161	0.019	0.595
	*RF_BF**	0.982	0.843	0.603	0.699	0.685	0.112	0.006	0.561
Difference		+0.000	+0.020	+0.127	+0.019	+0.007	+0.049	+0.013	+0.034

for all the classes, with the exception of *Other* which, remains the same. In the Fuse dataset, the gains are even more substantial. However, once more they come at the cost of a low score for the *GHead* class.

Regardless, both for 6 and 7 labels, the general assessment is positive. Especially for the Fuse dataset, the classifiers trained on the combined dataset, improve significantly the overall performance. Therefore, we use these classification results to compare to related work, in the following section. For completeness, below in Table 4.16, we additionally report the configurations used to train the aforementioned classifiers (i.e., the most performant ones for the combined dataset).

Table 4.16: Optimal Configurations for the Combined Dataset

Classifier		Parameters	Nr. Selected Features	Sampling Size
6 *labels*	*RF_BF*	*n_estimators=100, criterion='entropy', max_features='sqrt'*	189	$435K$
7 *labels*	*RF_BT*	*n_estimators=400, criterion='entropy', max_features='sqrt'*	149	$405K$

Comparison with Related Work

As discussed in Section 2.3.1 and Section 3.1, a one to one comparison with related work is not possible. Specifically, there are substantial differences with regard to the annotation and classification methodology. Nevertheless, for reference, in Table 4.17 we list the results from Adelfio and Samet [11], and Chen et al [29]. In Table 4.18, we additionally provide the best classification results from our experimental evaluation, both for 6 and 7 labels. Note, the results for *Fuse 6*, *Fuse 7*, and *Enron 7* originate from models trained on the combined dataset. However, the results for *Enron 6* come from a model trained on the distinct Enron dataset. As shown in Table 4.14, this model is slightly better than the one trained on the combined dataset.

The work of Adelfio and Samet [11] and Chen et al [29] are the closest to this thesis since they use machine learning for layout analysis. However, we define in total 7 labels, while

the aforementioned works use respectively 6 and 4 labels. In Table 4.17, when possible, we have mapped our labels to those from related work. Nevertheless, the definitions for the mapped labels do not match exactly.

Yet, the most important difference from related work, is that they operate at the row level. Concretely, they both use variations of the CRF (Conditional Random Field) algorithm to predict the layout role of the individual rows in the sheet. Thus, in Table 4.17, the F1-scores from related work, show their performance on the classified rows. In Table 4.18, for our approach, we report the results for classified cells. For this reason, any direct comparison between the results would not be sound.

Bringing our results to the row level comes with its own challenges. Most importantly, the approach proposed in this thesis foresees heterogeneous rows, i.e., different cell labels in a row. While the approaches proposed by related work assume that rows are always homogeneous. Thus, any conversion of the results will not be complete or fair to one side or the other. Instead, we compare with Adelfio and Samet in terms of classification accuracy in tables, as discussed below and shown in Table 4.19.

Table 4.17: Best Classification Results (F1-scores) from Related Work

	Data	Header	Derived	Title	Note	GHead	Other	Macro_F1
***Adelfio** [11]*	0.998	0.930	0.938	0.793	0.525	0.511	–	0.805
***Chen** [29]*	0.994	0.774	–	0.774	0.834	–	–	0.844

Table 4.18: Best Classification Results (F1-scores) from this Thesis

	Data	Header	Derived	Title	Note	GHead	Other	Macro_F1
***Fuse* 6**	0.998	0.883	0.710	0.815	0.777	0.242	–	0.737
***Fuse* 7**	0.982	0.863	0.730	0.718	0.692	0.161	0.019	0.595
***Enron* 6**	0.995	0.892	0.588	0.690	0.484	0.443	–	0.682
***Enron* 7**	0.978	0.852	0.554	0.609	0.349	0.403	0.624	0.624

Among other things, Adelfio and Samet [11] discuss in their paper the accuracy of their approach at the table granularity. Specifically, they report the percentage of tables where all rows were correctly classified ($Table\ err = 0$). Moreover, they discuss the percentage of tables where at least Header and Data rows are predicted correctly ($H\&D\ err = 0$). In Table 4.19, we provide as well the values for these metrics, considering both Fuse and Enron dataset. Nevertheless, we point out that Fuse is closer to the datasets used by [11] since the annotated sheets were extracted from the Web. While Enron contains business spreadsheets, which tend to be more complex than Web spreadsheets (see Section 3.3.2).

At first glance, the approach of Adelfio and Samet seems to outperform the one proposed by this thesis. In other terms, for $err = 0$ our accuracy is always lower. Yet, when we allow one incorrect prediction (i.e., misclassification), the results favor our approach. In fact, Adelfio and Samet do not report results for $err <= 1$. In their case, a misclassified row can alter the meaning of the table. However, in our case, an error means a single misclassified cell. It is unlikely that one cell will prevent the correct detection and interpretation of an entire table. Besides, it is even possible to correct some of these errors with heuristics (refer to Chapter 5). Based on these arguments and the results below, we can say that our approach is viable. We achieve considerably good performance on Fuse, for both 6 and 7 labels. Moreover, the results for Enron 6 are satisfactory.

Table 4.19: Comparing Classification Accuracy in Tables

	Table $err = 0$	Table $err <= 1$	H&D $err = 0$	H&D $err <= 1$
Adelfio [11]	$\mathbf{56.30}\%$	–	$\mathbf{76.00}\%$	–
***Fuse* 6**	52.40%	$\mathbf{65.10}\%$	66.70%	$\mathbf{80.70}\%$
***Fuse* 7**	44.10%	$\mathbf{62.10}\%$	64.00%	$\mathbf{78.00}\%$
***Enron* 6**	39.70%	54.50%	58.10%	70.80%
***Enron* 7**	36.50%	51.80%	53.40%	66.60%

4.4 LAYOUT REGIONS

In this section, we outline another fundamental operation of the proposed approach. We describe how classified cells of the same label are grouped into larger rectangular regions. We refer to these larger structures as *Layout Regions*. As discussed in Chapter 6, these regions help us to streamline the table identification process. It is much more intuitive to work with collections of cells rather than with individual cells. In addition, the computational overhead decreases, since ultimately the approach handles a much smaller number of elements.

In the following paragraphs, we define concepts that are later used to formulate a definition for the Layout Region itself. Moreover, we illustrate the process of creating Layout Regions in Figures 4.6 and 4.7. For this, we partially reuse the example from Figure 4.1.

Below, we start with a definition for the *sheet* (in Microsoft Excel is known as *worksheet*). The cells in a sheet are organized into rows and columns, ultimately forming a grid-like structure. Therefore, the sheet can be encoded using a two-dimensional matrix.

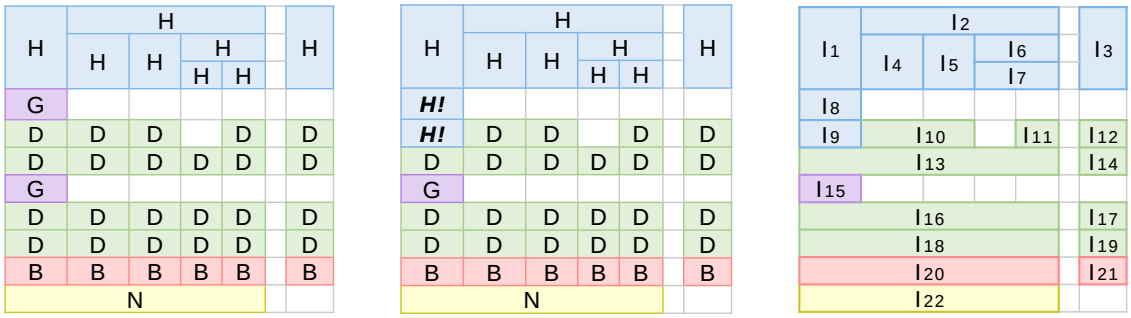
Worksheet can be defined as an m-by-n matrix of cells, denoted as $\mathcal{W}$. We can access a cell in row i and column j as $\mathcal{W}_{i,j}$, such that $1 \leq i \leq m$ and $1 \leq j \leq n$.

Classified Cell is a cell that was assigned a label ℓ by a classifier, where $\ell \in \{Data, Header, Derived, Title, Note, GHead, Other\}$. We use the function $label(\mathcal{W}_{i,j})$, to retrieve the classification result (i.e., predicted label) for a given cell. Note, this function is not defined for empty or hidden cells.

As stated at the beginning of this section, our goal is to group classified cells into larger coherent structures. We start at the row level by bringing together consecutive cells of the same label. In this way, we create coherent row groups, which we refer to as *Label Intervals*. Note, we operate at the row level (from left to right), since we expect the table headers to be on the top rows, i.e., they expand horizontally. As mentioned in Section 3.4, transposed tables with vertical headers are not within the scope of this thesis.

Label Interval is a submatrix $\mathcal{W}[i; j, j']$ of the worksheet $\mathcal{W}$ that holds classified cells of the same label. Let $\mathcal{W}_{i,j''}$ be any cell in the interval, s.t. $j \leq j'' \leq j'$. Then, we always get $label(\mathcal{W}_{i,j''}) = \ell$, i.e., the same label for all cells in this interval. Furthermore, we opt for maximal intervals. Therefore, the following condition must hold: $label(\mathcal{W}_{i,j-1}) \neq \ell$ and $label(\mathcal{W}_{i,j'+1}) \neq \ell$.

In Figure 4.6 we illustrate the creation of Layout Intervals. Figure 4.6.a provides an example annotated sheet. While Figure 4.6.b shows the corresponding classified sheet. We



(a) Annotated Cells (b) Classified Cells (c) Layout Intervals

Figure 4.6: Building Layout Intervals

denote the cell labels with distinct letters: Header *(H)*, Data *(D)*, Derived *(B)*, Note *(N)*, GroupHeader *(G)*. In Figure 4.6.b, the misclassified cells are denoted with an exclamation mark next to the label. Figure 4.6.c displays the resulting intervals. Note, we use I to denote the collection of all intervals in the worksheet $\mathcal{W}$. Therefore, in Figure 4.6.c the intervals are indexed, from left to right and top to bottom.

We observe several emerging scenarios. For instance, in the trivial case, an interval is made out of a single cell, e.g., I_{15}. In addition, we observe the presence of empty cells in between the intervals, e.g., I_{10}, I_{11}, and I_{12}. This complies with the definition discussed above. An interval carries only *classified* cells of the same label. Furthermore, in Figure 4.6.c, we notice that in some cases the *intervals* traverse multiple rows: I_1, I_3, I_4, and I_5. These are cells that were already merged (i.e., using the *Merge* & *Center* option from the Microsoft Excel menu) in the original sheet. At this stage, we do not group these cells with the rest. Nevertheless, we handle them in the subsequent stages.

As stated previously, our goal is to identify large and coherent regions of the sheet, such that they carry cells of the same label. Therefore, we proceed by grouping Label Intervals from consecutive rows. Nevertheless, we impose several constraints. Among others, we take into consideration how the intervals are arranged. Specifically, we require that intervals from consecutive rows share at least one column, i.e., they are vertically stacked. In Figure 4.6.c, the intervals I_9, I_{10}, and I_{11} are stacked on top of I_{13}. While the latter is itself stacked on top of I_{15}.

Stacked Intervals, let I be the set of all intervals in $\mathcal{W}$. Then I_k and $I_{k'}$ are stacked iff there exists at least one pair of cells $(\mathcal{W}_{i,j}, \mathcal{W}_{i+1,j})$ such that $\mathcal{W}_{i,j}$ in I_k and $\mathcal{W}_{i+1,j}$ in $I_{k'}$.

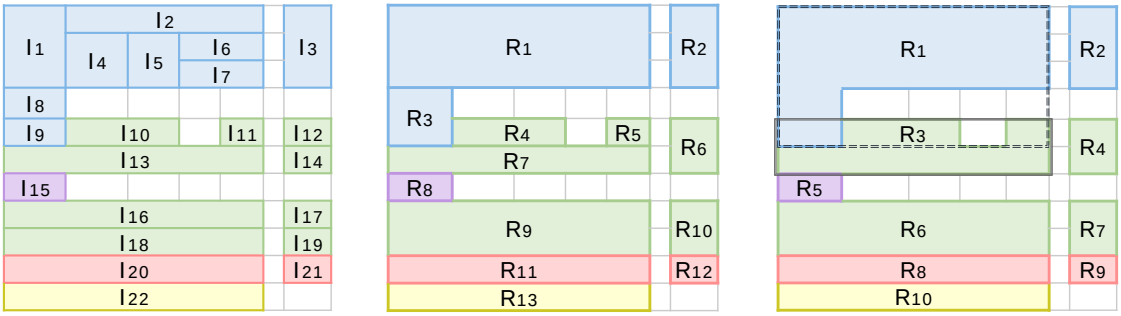
In addition to arrangement, we consider the label of the cells, when attempting to group intervals from consecutive rows. Specifically, the cells from the individual intervals of the group must share the same label. We refer to these coherent groups of stacked intervals as *Label Regions* or *Layout Regions*.

Layout Region is a collection intervals, i.e., a subset of I. In the trivial case, a region contains only one interval. When it contains multiple ones, the following must hold. Let (I_k, $I_{k'}$) be a pair stacked intervals in a given region. Then for any two cells $\mathcal{W}_{i,j}$ in I_k and $\mathcal{W}_{i+1,j'}$ in $I_{k'}$ the classification results is the same, i.e., $label(\mathcal{W}_{i,j}) = label(\mathcal{W}_{i+1,j'}) = \ell$.

Again, we opt for maximal regions, i.e., encompassing as many as possible intervals. Therefore, we satisfy the conditions discussed below. Let R denote the set of all Layout Regions in $\mathcal{W}$. Moreover, let $(I_k, I_{k'})$ be any pair of intervals from two different Layout Regions, s.t. I_k in R_t and $I_{k'}$ in $R_{t'}$. Then, one of the following two statements must hold:

I_k it is not stacked with $I_{k'}$ or the cell labels differ (i.e., $label(\mathcal{W}_{i,j}) \neq label(\mathcal{W}_{i',j'})$, such that $\mathcal{W}_{i,j}$ in I_k and $\mathcal{W}_{i',j'}$ in $I_{k'}$).

Last, we need to address the special case of merged cells. When these cells span multiple columns, but only one row, we simply treat them as Label Intervals. However, when they span multiple rows, we need to treat them separately from the rest. We still attempt to group them with existing intervals, in order to form larger Layout Regions. Nevertheless, in this case, we operate column-wise instead of row-wise. Concretely, we identify intervals that are horizontally stacked with a given merged cell. Clearly, these intervals must also have the same label as the merged cell.



(a) Layout Intervals (b) Strict Layout Regions (c) Non-Strict Layout Regions

Figure 4.7: Building Layout Regions

In Figure 4.7, we present the resulting regions from the example sheet in Figure 4.6. There are two variations for Layout Regions: *strict* (Figure 4.7.b) and *non-strict* (Figure 4.7.c). For the latter variation, the definition is the same as the one discussed above. However, for the former, we add one more constraint. Concretely, for every Layout Region, we require that its intervals collectively cover a strictly rectangular area (range) of the sheet.

This becomes more intuitive once we represent the Layout Regions with the minimum bounding rectangle (*MBR*) that encloses the contained intervals. For strict regions, the resulting *MBRs* are never overlapping. However, overlaps occur when the non-strict definition applies. This is illustrated in Figure 4.7.c. We observe that the region R_1 contains all the intervals I_1 to I_9, with exception of I_3. While the region R_3 encompasses the intervals: I_{10}, I_{11}, and I_{13}. The *MBRs* for these two regions overlap, because the intervals I_9, I_{10}, and I_{11} come from the same row. In general, the *MBRs* from non-strict regions might include empty cells or cells of another label.

We make use of both variations in the subsequent chapters. Each one of them has its own advantages and disadvantages. For instance, in Chapter 5, we use strict Layout Regions to isolate potential misclassifications. Indeed, in Figure 4.7.b, the region R_3 contains two misclassified cells. One of the clues is that R_1 and R_3 do not fit together, even though they are adjacent. On top of that R_3 is much smaller than R_1. Furthermore, in Appendix B, the proposed table detection method uses non-strict Layout Regions. The advantage is that we get a smaller number of regions, compared to when the strict definition is used. Moreover, we show that it is possible to make use of overlaps (i.e. the region *MBRs* overlap). Using this and other available information (e.g., the cell labels), we can determine if two regions belong to the same table or not.

4.5 SUMMARY AND DISCUSSIONS

In conclusion, this chapter introduced three steps of the proposed processing pipeline: *feature extraction*, *cell classification*, and *layout region creation*. These steps are fundamental parts of our approach for layout analysis in spreadsheets. We operate on a bottom-up fashion, starting from the individual cells to later build Layout Regions. This approach allows us to describe a multitude of spreadsheet layouts, with diverse table structures and mixed arrangements.

We make use of machine learning techniques to predict the layout function of individual cells. In Section 4.2, among other things, we propose cell features that were not considered before by related work. A substantial number of them require custom implementation or pre-processing, e.g., the spatial (Section 4.2.5) and formula (Section 4.2.4) features. In Section 4.3.4, we show that the proposed features are the majority of the top 20, with regards to importance.

In Section 4.3, we perform extensive experimentation, to identify the best classification models for each one of the considered datasets (Fuse and Enron). We train models on the individual datasets and the combined one. Furthermore, we study two separate scenarios: 6 and 7 labels. The results from the experimental evaluation show that our approach is viable. We get good classification performance for most of the annotation labels. In particular, we perform well in the Fuse dataset, which contains Web spreadsheets. Moreover, we achieve better classification accuracy inside the tables, when comparing with related work, with regard to Web spreadsheets.

Last, we propose a procedure that consolidates the classified cells into Layout Regions. We formally define the concepts used in this procedure. Additionally, we illustrate the creation of regions with figures. In the subsequent chapters, we make use of these regions to repair misclassification and detect tables.

CLASSIFICATION POST-PROCESSING

5.1 Dataset for Post-Processing

5.2 Pattern-Based Revisions

5.3 Region-Based Revisions

5.4 Summary and Discussion

In this chapter, we discuss techniques for handling incorrect predictions (i.e., misclassifications) that occur during the cell classification process (see Chapter 4). Intuitively, the revision of misclassifications can make the subsequent tasks, such as table detection (Chapter 6) and information extraction (Chapter 7), much easier and accurate.

We propose two approaches. In Section 5.2 we discuss the first one, which is based on rules. These rules apply to individual cells and their neighborhood. Note, this approach precedes the creation of layout regions (see Section 4.4). The second approach is based on *strict layout region*, as is detailed in Section 5.3. Therefore, in this case, revision and region creation can be performed together. Moreover, the second approach utilizes machine learning techniques, in addition to heuristics.

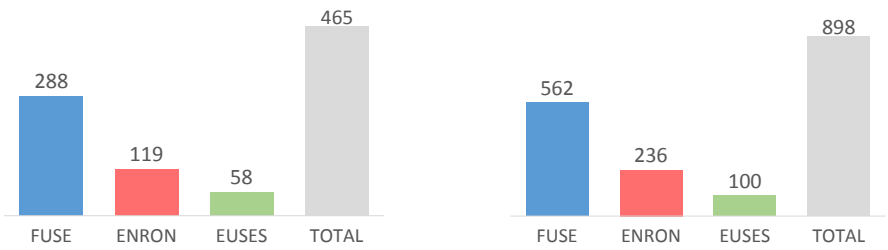
Nevertheless, the experimental evaluation from our original paper [85] and the extended book chapter [87] showed that these approaches have limitations. They extensively depend on the context provided by the immediate and/or extended neighborhood of the cells. Thus, when there are many misclassifications in the sheet, revision might lead to further distortion, rather than improvement. For this reason, we regard these approaches as optional, rather than a core part of our processing pipeline (refer to Figure 3.12). They should be considered when the cell classification accuracy is already high for most of the files in the dataset.

Note, in the following sub-sections we outline the two proposed approaches, based on the original publications [85, 87]. The dataset used for these publications differs from the ones described in Chapter 3 and used in Chapter 4. Therefore, the next section provides a brief summary of the original dataset, as was first introduced in [85].

5.1 DATASET FOR POST-PROCESSING

In our earlier publications [85, 87], we have used a collection of 216 annotated files. Hereinafter, we refer to this dataset as the *KDIR dataset*, based on the name of the conference where the original paper was presented [85]. The files for this dataset were randomly drawn from three different corpora: Fuse [16], Euses [52], and Enron [72]. As detailed in Section 3.1, the first two corpora are crawled from the Web, while the latter one contains business spreadsheets.

Specifically, we selected 133, 56, and 30 files respectively from Fuse, Enron, and Euses. Figure 5.1 highlights the contribution of each corpus, in numbers of annotated sheets and tables. For KDIR we annotated all the sheets having tables, not just the first one (refer to the annotation logic in Section 3.2.2). Therefore, the average number of annotated sheets per file is slightly more than 2.



(a) Sheets per Corpora (b) Tables per Corpora

Figure 5.1: Annotation Statistics for the KDIR Dataset [85]

Furthermore, for the KDIR dataset, we used five annotation labels, instead of seven (see Section 3.2.2). The KDIR labels are illustrated in Figure 5.2. The definition of *Header* and *Data* labels remains the same as in Chapter 3. The label *Metadata* can be seen as a superclass of *Title*, *Note*, and *Other*. The label *Attribute* is closely related to *GroupHeader*, as they both describe hierarchies on the left of the table. However, *Attributes* capture the whole left hierarchy: both parents and children. Instead, *GroupHeaders* capture only the parents in this hierarchy (refer to Figure 3.2). Moreover, for KDIR we annotate all aggregations as *Derived*. Contrary, for the datasets in Chapter 3 we annotate as *Derived* only aggregations interrupting *Data* rows.

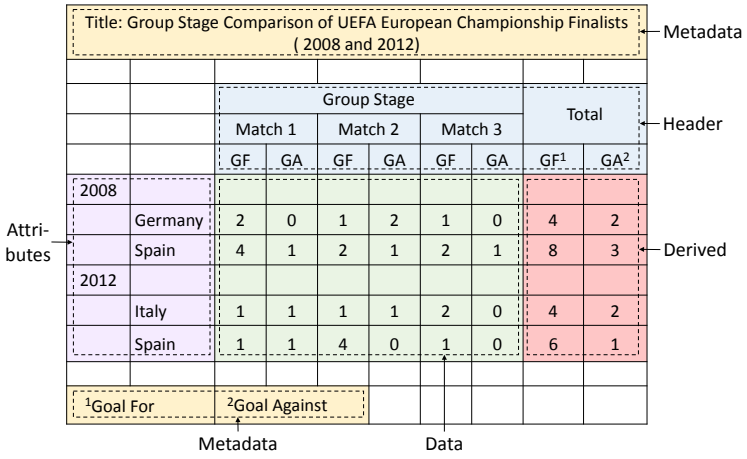


Figure 5.2: The Building Blocks [85]

For the KDIR dataset, we trained classifiers to predict the layout function of non-empty cells. Again, the most accurate model was a Random Forest. Nevertheless, we still got in total $1,237$ misclassified cells, spread among the considered sheets. The approaches discussed in the subsequent sections attempt to repair some of these incorrect predictions.

5.2 PATTERN-BASED REVISIONS

Our initial analysis of the KDIR classification results hinted that neighboring cells can be used to recover some of the misclassifications. Intuitively, the label assigned to a cell should match, in most cases, the label of the neighboring cells in the same row and/or column. Here, we define the neighborhood as a 3-by-3 window around a cell, as was previously illustrated in Figure 4.2. The cell in the center represents a potential misclassification, surrounded by eight neighboring (adjacent) cells.

Not all neighbors have a label. For instance, empty and hidden cells do not get a label, since they are omitted from the classification process. Moreover, cells located at the boundaries of a worksheet (i.e., the minimum and maximum row/column allowed by the spreadsheet application), the neighborhood is smaller than eight. In order to standardize the number of neighbors for any arbitrary cell, we introduce two artificial labels: "Empty" and "Out". The latter is used for (non-existing) neighbors outside the boundaries of the sheet, while the first for all the other cases of un-labeled neighboring cells.

With these two labels and the ones from Figure 5.2, we attempt to infer misclassification patterns. Specifically, we studied the neighborhood of incorrectly classified cells ($1,237$ in total) from the KDIR dataset. We detected 672 unique combinations of labels in these

neighborhoods. Subsequently, based on the 40 most repeated combinations, we inferred manually generic rules (i.e., not bound to specific labels).

These rules are divided into two sets: identification and relabeling. Intuitively, we use the first rule-set to identify incorrect predictions, and then we relabel them using the second rule-set. Below we discuss the considered rules, in detail.

5.2.1 Misclassification Patterns

Figure 5.3 provides a visual representation of the considered rules (patterns). The cell in the center[1] (marked with red dots) is a potential misclassification. Neighbors filled with green diagonal lines (referred to as *Influencers*) share the same label among them, but a different one from the center. Neighbors filled with black dots have the same label as the central cell. Those marked with an 'X' (i.e., diagonal borders) have a label different from the center. However, unlike the *Influencers*, these cells do not necessarily share the same label among them. Furthermore, we treat neighbors labeled as Attributes (see Figure 5.2) in a special way. Therefore, we mark these neighbors explicitly, with the letter 'A'. Finally, for arbitrary neighbors (i.e., can have any label), we leave the cell as blank.

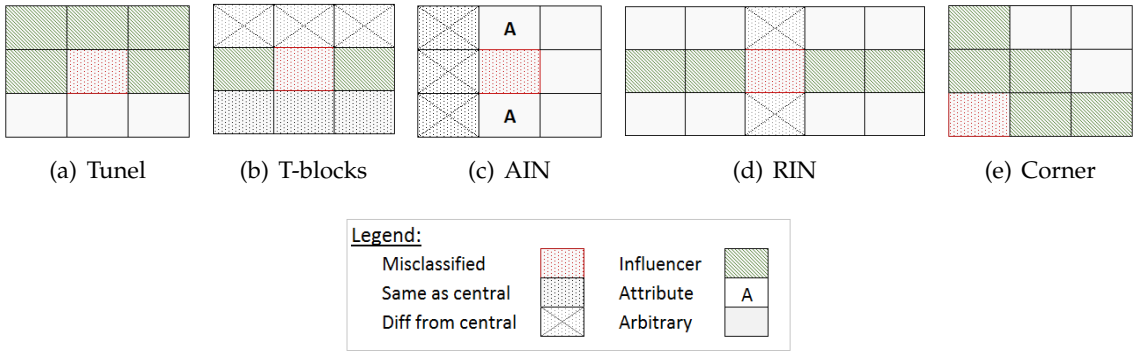


Figure 5.3: Misclassification Patterns

One of the most common patterns is the "Tunel", shown in Figure 5.3.a. Intuitively, it attempts to identify misclassified cells that are completely or partially surrounded by neighbors of a different label. We consider two instances of this pattern. The left and right neighbors are always *Influencers*. However, the remaining three *Influencers* can be either on the top or on the bottom row.

The "T-block" pattern is shown in 5.3.b. It differs from the "Tunel" pattern, mainly due to the bottom row. Here, we require that cells at the bottom have the same label as the central one. In addition, the T-block pattern can occur upside-down, i.e., when the "head" of the T-shape is on the top row.

The "Attribute Interrupter" (AIN) pattern relates to Attribute cells (see Figure 5.3.c). For this reason, AIN has a vertical nature. Intuitively, this pattern specializes in identifying misclassified cells in Attribute columns.

The "Row Interrupter" (RIN) pattern (see Figure 5.3.d) applies when the predicted label of a cell does not match the predicted label of the majority of cells in the row. Note the RIN

[1]This with exception of the Corner pattern, where the potential misclassification is at the bottom-left corner, rather than in the center.

covers a 3-by-5 window. We have expanded the original definition of the neighborhood, in order to make this pattern more accurate.

Finally, the “Corner" pattern typically identifies misclassifications that occur in the corners of the tables. Note, there are four possible variations of this pattern, i.e., the misclassified cell can be in any of the four corners. Figure 5.3.e displays the bottom-left variation.

Table 5.1 provides the number of times each pattern occurs in the classification results. In the majority of times, these patterns match misclassifications (i.e., true positives). However, there are also some false positives, i.e., the label of the cell was predicted correctly by the classifier.

Table 5.1: Pattern Occurrences in KDIR Dataset

Pattern	True Positives	False Positives	Total
Tunel	45	12	57
AIN	41	1	42
RIN	29	4	33
T-blocks	28	1	29
Corner	13	2	15

5.2.2 Relabeling Cells

Here, we describe our strategy for relabeling the cells matched by the misclassification patterns. Essentially, in most cases, we use the *Influencers* to determine the new label (see Figure 5.3). For the Tunel pattern, we update the label of the central cell to match the majority of its neighbors. When the T-block pattern is identified, the label of the right and left neighbors is used. We set the label for the central cell to Attribute, when the AIN pattern occurs. For the RIN pattern, we flip the label to match the rest of the row. Finally, when the Corner pattern occurs, the label of the cell is updated to that of the *Influencers*.

5.2.3 Evaluating the Patterns

Here we evaluate the whole pattern-based procedure, including both steps: identification and relabeling.

Table 5.2: Label Flips

	Attributes	Data	Header	Metadata	Derived
Gained	41	57	26	18	10
Lost	0	1	8	3	2

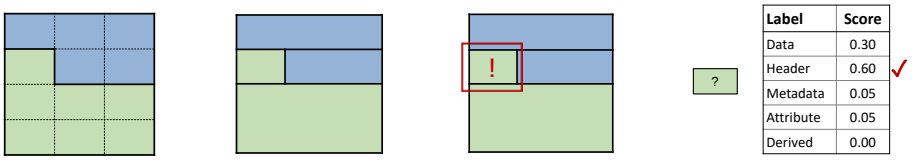
Note, we first established an execution order for the patterns, since they are not mutually exclusive. We experimentally determined that the most accurate results come from a sequential run in the following order: AIN, T-Blocks, Corner, RIN, and Tunel. Essentially, in such a run, updated labels from the previous pattern become an input to the next one. The results show that we managed to repair 152 misclassified cells and lose 14 correctly classified ones. Table 5.2 provides the results per label.

5.3 REGION-BASED REVISIONS

Although the aforementioned pattern-based method is able to recover a number of incorrect classifications, it has limitations. Most notably it assumes that cells in the immediate neighborhood are, in most cases, correctly classified. We analyzed again the neighborhood of $1,237$ misclassified cells from KDIR dataset. In almost half of the cases, we find incorrectly classified neighbors. Specifically, in 42% of the studied neighborhoods, we find 1 or 2 misclassifications. While in 6% of them we find 3 or more misclassifications.

Therefore, in this section, we propose a method that goes beyond the immediate neighborhood. We additionally collect information from distant neighbors, which can provide valuable insights. For instance, in sparse tables (i.e., having many missing values) the immediate neighbors are often empty cells. Thus, one should look in the next row or column for non-empty (classified) neighbors. Moreover, the distance from other cells of the same label can be informative, as well. For example, a Header cell that is far from all the other Headers is most probably a misclassification.

Figure 5.4 illustrates the proposed approach. Initially, we load the classification results. In the second step, we create strictly rectangular regions, using the definition from Section 4.4. These regions enclose adjacent cells of the same label (see Figure 5.4.b). We base this step on the intuition that tables are typically well-formed. Specifically, the distinct sections (i.e., layout regions) that comprise the tables tend to be of rectangular shape. In other words, we do not expect them to come in arbitrary shapes (i.e., rectilinear polygons > 4 edges).



(a) Load (b) Standardize (c) Identify (d) Relabel

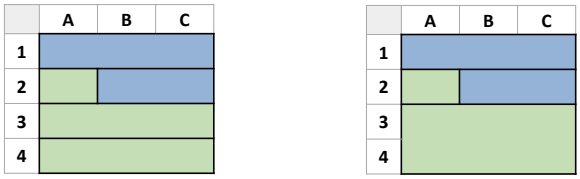
Figure 5.4: Region-Based Approach

Our hypothesis is that irregular regions point towards misclassifications. Essentially, the aim of our approach becomes to isolate the cell/s that break the expected regularity (Figure 5.4.b). Subsequently, we determine if these cells were correctly classified (Figure 5.4.c). If they were not, we attempt to predict the correct label (Figure 5.4.d).

For the last two steps, we use supervised machine learning. Therefore, we collect a good mixture of features, from the immediate and the distant regions. All in all, we get a much more extended view in the neighborhood of potentially misclassified cells. In the following section, we revisit the aforementioned steps, discussing them in more detail.

5.3.1 Standardization Procedure

The standardization procedure is formally described in Section 4.4. Here, we illustrate it again, this time in the context of post-processing revisions. The process starts with the creation of Label Intervals. We define these intervals as a sequence of cells of the same label in a row. Figure 5.5.a displays the intervals from the example shown in Figure 5.4.a.



(a) Row Intervals (b) Regions

Figure 5.5: Original Worksheet

The first, third, and fourth row contain one interval each. While the second row contains two intervals of a different label.

As we emphasized previously, we are interested in strictly rectangular regions. We try to achieve this by grouping intervals of the same label, from consecutive rows. However, in order to preserve the rectangular shape, these intervals must have the same start and end column. Based on this reasoning, for our running example, we group the intervals in the third and fourth row, as illustrated in Figure 5.5.b. This creates the region *A3:C4*. Moreover, we managed to isolate the cell *A2* that prevented the rest of the green cells to form a well-shaped region.

However, we were not able to group the blue intervals from 1st and 2nd row, since they have a different start column. Clearly, there is potential to build a larger blue region, which is *B1:C2*. This would have also isolated the cell *A1*. The latter is desirable, since at this phase our aim is to pinpointing all irregularities, regardless if they are misclassifications or not.

(a) Column Intervals (b) Regions

Figure 5.6: Pivoted Worksheet

One way to tackle this challenge is by creating regions column-wise, in addition to row-wise. We achieve this by transposing the original worksheet, such that the columns become rows, and vice-versa. Now, we can construct column intervals, following the same procedure as described in the above paragraphs. The results are shown in Figure 5.6.a. Once we group the label intervals, we get the output shown in Figure 5.6.b. As intended we create a large blue region *B1:C2*, and simultaneously isolate the blue cell *A1* that does not fit well with the rest.

Intuitively, our standardization procedure produces two alternative partitionings for the classified cells. Typically, the optimum partitioning results either from row intervals or from column intervals. However, in some cases, we need both directions to ensure that all misclassified cells are isolated. Therefore, in the end, we keep both outputs. Nevertheless, we attempt to reduce the number of considered regions by filtering out duplicates (i.e., when the alternative partitionings produce the same region). Figure 5.7 summarizes the overall standardization procedure, which includes a filtering step for duplicates.

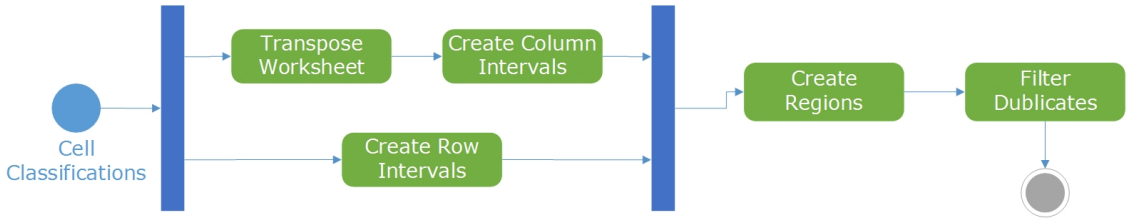
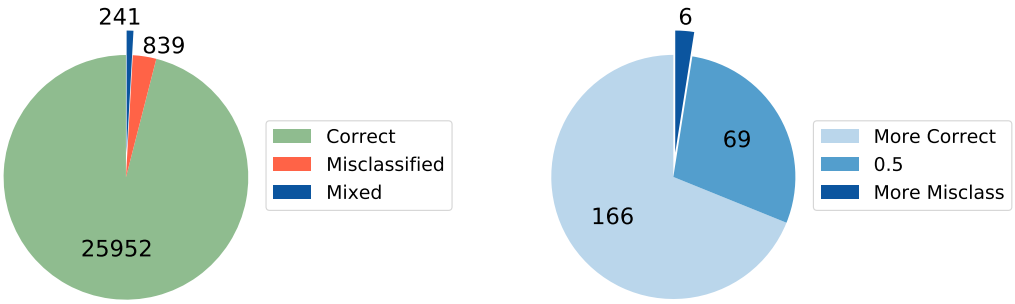


Figure 5.7: Standardization Procedure

Standardization Assessment

We assess the validity of the standardization procedure by testing it on the classification results from the KDIR dataset. We group the resulting regions into three categories, based on the ratio of misclassified cells that they contain. We call "Misclassified" regions that contain only misclassified cells. For those that contain only correct predictions, we use the term "Correct". The remaining cases, regions that contain both correct and incorrect classifications, we refer to as "Mixed". Figure 5.8.a provides the number of regions per category.



(a) Overall Assessment (b) Mixed Regions

Figure 5.8: Region Analysis

We notice that 839 and 241 regions are respectively *Misclassified* and *Mixed*. The latter might raise concerns at first glance. However, *Mixed* regions are a natural by-product of the procedure. Consider again Figure 5.6.b. Region *A2:A4* contains a misclassified cell, i.e., *A2*. This can be seen clearly in Figure 5.4. Thus, for the running example, building regions column-wise introduces a *Mixed* region. In general, both partitioning strategies (row-wise and column-wise) can produce such regions.

In Figure 5.8.b, we provide a more detailed view of *Mixed* regions. We notice that in the majority of *Mixed* regions the number of correctly classified cells is greater than that of misclassified cells. Moreover, there are 69 cases where the numbers are equal, and an insignificant number of cases with more misclassified cells.

To simplify our subsequent operations, we decided to maintain only two categories. Therefore, we redistribute the *Mixed* regions. Those that contain mostly incorrect classifications (> 0.5) are marked as *Misclassified*, the rest are marked as *Correct*. This brings the number of regions per category to 845 and $26,187$ respectively.

Filtering by Size

As mentioned before, one of the drawbacks of our standardization procedure is the considerable number of outputted regions. Ideally, we would like to keep only those that have the most potential of being *Misclassified*. Therefore, we analyzed the *Misclassified* regions, in order to identify their typical characteristics. Our analysis revealed that these regions exhibit small sizes (i.e., number of cells in the region) As shown in Figure 5.9, the larger is the size of a Mislcassified region the least are its occurrences.

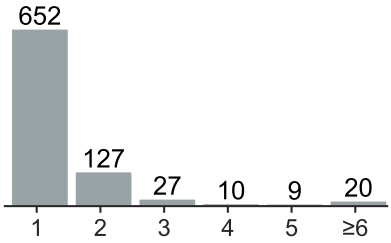


Figure 5.9: Size Occurrences in Misclassified Regions

Based on these results, we decided to keep regions containing up to 3 cells. We omit the larger regions since they occur infrequently. Intuitively, with this action, we also reduce the chances of having false positives (i.e., *Correct* regions flagged as *Misclassified*). After filtering by size, we get in total $12,724$ regions. Out of these, 806 are *Misclassified*, and $11,918$ are Correct regions. We use this reduced dataset for the steps described in the following sections.

5.3.2 Extracting Features from Regions

We use supervised machine learning techniques for the identification and relabeling of *Misclassified* regions. Therefore, we have created a set of features, which are formally defined in the following paragraphs. Most of these features are used both for identification and relabeling.

Table 5.3 summarizes the features that are extracted for each rectangular region. Note, features ending with '#' are numeric, while those ending with '?' are Boolean. Moreover, the table contains a nominal feature, which is the *predicted_label*. We group these 12 features into two categories: "Simple" and "Compound". Features in the latter group derive from multiple simple ones. Some of them are not explicitly listed in Table 5.3.

To extract the proposed features, we represent each region with its minimum bounding rectangle. The worksheet itself can be seen as a Cartesian Coordinate system, where the point $(1, 1)$ is at the top left corner. The values of the x-axis increase column-wise, while for the y-axis they increase row-wise. Having such a coordinate system, it is relatively easy to convert the regions into abstract rectangles. The top-left coordinates of the rectangle correspond to the column and row number of the top-left cell in the region. The width and height can be calculated by counting respectively the number columns and rows in the region. For example, the region *A3:C4* (shown in Figure 5.5.b) will be represented by a rectangle having top-left coordinates $(1, 3)$, $width = 3$, and $height = 2$.

The *simple features*, in Table 5.3, characterize various aspects of the rectangle (region). A region is horizontal when $width > height$, is vertical when $width < height$, and is

Table 5.3: Region Features

Nr.	Simple	Nr.	Compound
1	IS_HORIZONTAL?	9	SIMILARITY_{TOP,BOTTOM,LEFT,RIGHT}#
2	IS_VERTICAL?	10	DISSIMILARITY_{TOP,BOTTOM,LEFT,RIGHT}#
3	IS_SQUARE?	11	EMPTINESS_{TOP,BOTTOM,LEFT,RIGHT}#
4	COUNT_CELLS#	12	INFLUENCE_{TOP,BOTTOM,LEFT,RIGHT}#
5	COUNT_ITS_KIND#		
6	DISTANCE_FROM_ITS_KIND#		
7	DISTANCE_FROM_ANY_KIND#		
8	PREDICTED_LABEL		

square when $width = height$. The feature *count_cell* describes how many cells are in the region (i.e., the area of the rectangle). Additionally, we count the number of regions in the worksheet having the same label (i.e., its own kind) as the current region. The *distance_from_its_kind* captures the smallest Euclidean distance of this region to a region of the same label. While *distance_from_any_kind* captures the smallest Euclidean distance of this region from regions of any label. Finally, the *predicted_label* stores the label assigned by the classifier (see Section 5.1).

With the *compound features*, we analyze the neighborhood of a region. However, unlike in the pattern-based approach (see Section 5.2), the neighborhood comprises of other regions, instead of cells. Note, this time we study the neighborhood only in four directions: top, bottom, left, and right. Additionally, we consider distant regions (neighbors), besides the nearest ones. We examine the label of these regions, to determine if they are similar or dissimilar to the current region. Furthermore, we are interested in regions having a specific label. We measure the influence of these regions on the current one. Finally, we use emptiness for cases where there is no region (neighbor) in a direction.

Below we formally define the individual compound features. For these definitions we utilize the following concepts:

- **Current Region**: The region whose neighborhood we are studying.
- **Directions**: Can be *Top*, *Bottom*, *Left*, or *Right*.
- **Neighbors**: Any region other than the current one.
- **Nearest neighbors (NNs)**: The neighboring regions with the smallest Euclidean distance from the current region in the specified direction.
- **Similar neighbors (SNs)**: Neighbors that have the same label as the current region.
- **Dissimilar neighbors (DNs)**: Neighbors that have a different label from the current region.

Quantifying the Neighborhood

It is important to emphasize that the number of neighboring regions can vary extensively. Moreover, the neighboring regions might come in different sizes (considering both their

width and height). Therefore, we need a method to weight the importance of each neighbor. For this, we utilize two measures: *overlap-ratio* and *distance*. The former quantifies how much of the specified direction is dominated by a neighboring region. The latter quantifies how far or close a neighbor is.

Equation 5.1, illustrates how the overlap ratio is calculated. In this equation, r stands for the current region, n_i for the selected neighbor, and d for the current direction. We use the horizontal edge for top/bottom neighbors. We project this edge to the x-axis. We do the same for the horizontal edge of the current region. Subsequently, we measure the length of the overlap for these projections. For left and right neighbors, we follow the same logic, using the vertical edges. However, we project them to the y-axis, instead. We transform the measurement into a ratio by dividing with the width or height (i.e., respectively, the length of the vertical or horizontal edge) of the current region.

$$\begin{aligned} &OverlapRatio(r, n_i, d) = \frac{Overlap(r, n_i, d)}{EdgeLength(r, d)} \\ &\text{where } \; n_i \in Neighbors(r, d) \; \text{ and } \; d \in Directions \end{aligned} \tag{5.1}$$

Once we have the overlap ratio and the distance to the neighbor, we can calculate its weight as shown in Equation 5.2. In the denominator, we add one to the distance to account for cases where the latter is zero (regions are adjacent, in consecutive rows/columns). Clearly, this equation captures the intuition that the weight (i.e., importance) for a neighbor should increase for smaller distances and bigger overlap ratios.

$$weight_i = OverlapRatio(r, n_i, d) \cdot \frac{1}{1 + Distance(r, n_i)} \tag{5.2}$$

We can now define the *similarity* for a region and a neighbor, as shown in Equation 5.3. Similarity takes a value greater than zero when the neighbor is one of the SNs and simultaneously an NN. When these two conditions are satisfied, the value of the *similarity* equals the weight of the neighbor.

$$similarity_i = \begin{cases} 0 & Label(r) \neq Label(n_i) \;\; \vee \;\; n_i \notin Nearest(r, d) \\ weight_i & otherwise \end{cases} \tag{5.3}$$

$$dissimilarity_i = \begin{cases} 0 & Label(r) = Label(n_i) \;\; \vee \;\; n_i \notin Nearest(r, d) \\ weight_i & otherwise \end{cases} \tag{5.4}$$

Likewise, we calculate the *dissimilarity* for a neighbor, as shown in Equation 5.4. The only difference from the definition of *similarity* is that here the neighbor must be one of the DNs, in addition to being a NN.

Influence goes beyond the immediate neighborhood (i.e., the nearest neighbors). It additionally quantifies how much distant neighbors of a certain label influence the current region. Influence can prevent false positives. Consider the scenario where some of the nearest neighbors of the current region are dissimilar, due to misclassifications. By looking at more distant neighbors, we make more informed decisions. For example, distant neighbors could show that the extended neighborhood is, in fact, more similar (or fitting) to the current region, compared to the immediate neighborhood. Additionally, influence can increase the accuracy of relabeling. For instance, strong influence from multiple bottom neighbors of a Data label can reinforce the belief that Header is the most plausible label for the current region.

Clearly, Influence is tightly coupled with the selected label, as shown in Equation 5.5. Here, we have updated the function *Nearest* by adding the optional parameter *label*, denoted as l. When this parameter is set, the function returns only the closest neighbors of a specific label in the given direction. Influence gets a value greater than zero only when there exists at least one neighbor with the requested label. When there are multiple such neighbors in a direction, we prefer the influence from the nearest ones.

$$influence_i = \begin{cases} 0 & Label(n_i) \neq l \ \vee \ n_i \notin Nearest(r, d, l) \\ weight_i & otherwise \end{cases} \tag{5.5}$$

Aggregating by Direction

All the previous equations hint that there can be more than one nearest neighbor for a given direction. In order to get the total value of a feature, we need to aggregate the values from the individual NNs. Equation 5.6 and 5.7 respectively show how to perform this for *similarity* and *influence*. We can calculate the total *dissimilarity* for a direction in the same way.

$$total\_similarity_d = \sum_{i=1}^{|Nearest(r,d,Label(r))|} Similarity(r, n_i, d) \tag{5.6}$$

$$total\_influence_{d,l} = \sum_{i=1}^{|Nearest(r,d,l)|} Influence(r, n_i, d, l) \tag{5.7}$$

Emptiness, the last compound feature, captures the (partial or complete) non-existence of nearest neighbors in a direction. Emptiness takes the maximum value when there are no neighbors in a direction. When neighbors partially overlap with the current region, *emptiness* takes a value between zero and one. Equation 5.8 illustrates how to calculate the value of this feature in a specific direction. Note, in this equation that we do not set the optional *label* parameter for the *Nearest* function. Thus, it returns all NNs.

$$total\_emptiness_d = 1 - \sum_{i=1}^{|Nearest(r,d)|} OverlapRatio(r, n_i, d) \tag{5.8}$$

We can add additional flavors to the compound features by aggregating them to the level of row (left and right), column (top and bottom), and that of the overall neighborhood (i.e., all four directions). Equation 5.9 illustrates how to calculate the value for the overall neighborhood, using the *similarity* feature as an example. As shown below, we normalize the value from a direction using the ratio between the edge length (in that direction) and the perimeter of the current region.

$$overall\_similarity = \sum_{j=1}^{|Directions|} \left(\frac{EdgeLength(r, d_j)}{Perimeter(r)} \cdot similarity_{d_j} \right) \tag{5.9}$$

5.3.3 Identifying Misclassified Regions

We define misclassification identification as a machine learning task, whose goal is to distinguish real *Misclassified* regions (i.e. true positives) from the Correct regions (i.e.,

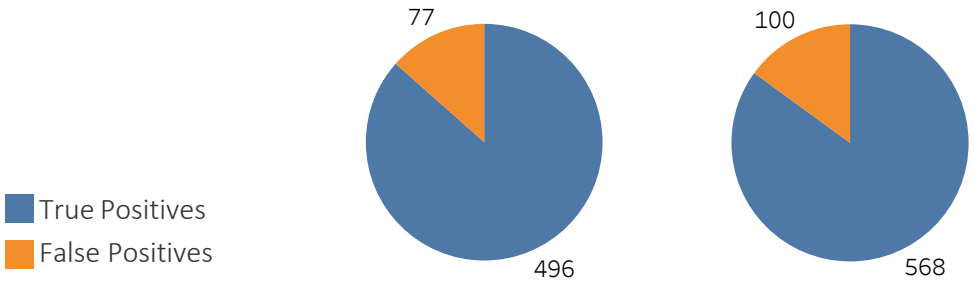
true negatives). For this binary classification problem, we have considered all the simple features mentioned in the previous section (see Table 5.3). Additionally, we use the compound features in all four directions, together with their three flavors (i.e., aggregating by row, column, and the overall neighborhood). For this task, we only consider the influence from neighbors having the same[2] label as the current region. In total, the number of features considered for misclassification identification is 36 (i.e., 8 simple + 28 compound).

Table 5.4: Comparing Classifiers for Misclassification Identification

	Random Forest *-I 100*	**SMO RBF** *-C 19.0, -G 0.1*	**Logistic Regression** *-R 1.0E-14*	**JRIP** *-N 10.0*
F1 Measure	0.97	0.96	0.95	0.96
True Positive Rate	0.64	0.58	0.47	0.60
False Positive Rate	0.03	0.03	0.04	0.04

For our evaluation, we experimented with several classification algorithms (shown in Table 5.4). The Random Forest and Logistic Regression classifiers were already introduced in Section 4.3.2. The SMO RBF classifier is an implementation of SVM classifiers with RBF kernel, optimized for fast training [135]. JRIP is a propositional rule learner [33].

We have used the implementations provided by the Weka tool [132], a workbench for machine learning. We first tuned the parameters of the individual classifiers. Subsequently, we used Weka Experimenter to run 10-fold cross-validation 10 times. The results displayed in Table 5.4, are the averages of all runs. Random Forest achieves the highest F1-measure and simultaneously has the highest true positive rate. With regard to the false positive rate, there is no substantial difference between the classifiers. Considering these results, we selected the Random Forest classifier for our subsequent analysis.



(a) In Regions (b) In Cells

Figure 5.10: Misclassification Identification Results

In Figure 5.10 we display the results from one of the cross-validation runs ($seed = 1$), with the Random Forest classifier. We provide the numbers in terms of regions and in terms of individual cells. We get more false positive cells (i.e., wrongly flagged as *Misclassified*) in comparison to the pattern-based approach (see Table 5.2). Nevertheless, the number of true positive cells (i.e., correctly predicted as *Misclassified*) is several times higher for this approach.

[2]In the next step, i.e., relabeling, we collect the influences from all labels (refer to Section 5.3.4)

5.3.4 Relabeling Misclassified Regions

We define the relabeling task as that of predicting the most plausible label for a region that was previously flagged as *Misclassified*. For this task, we use all the simple features, with the exception of *predicted_label*. From the compound features, we use only *influence*. We capture this feature for each label and direction. In addition, for this feature, we consider the three aggregations (flavors): row, column, and overall. With this, the total number of features used for relabeling becomes 42 (i.e., 7 simple + 35 influences).

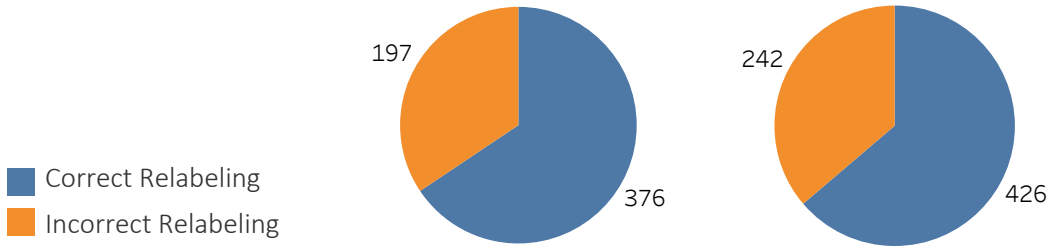
Note, for this task, we train our model (relabeler) on the original annotated sheets (i.e., the ground truth), instead of the predicted labels. In this way, we avoid tight coupling between the original cell classifier and the relabeler. In other words, the relabeler learns from real examples, not the noisy ones that are just a product of cell classification.

All in all, we have constructed rectangular regions from the annotated sheets. In the end, we keep only those regions of size three or smaller for training, the same as in Section 5.3.1. This results in 11,934 regions.

Table 5.5: Relabeling: Trained on Annotated Regions

	Random Forest *-I 350*	**SMO RBF** *-C 16.0 -G 1.0*	**Logistic Regression** *-R 1.0E-8*	**JRIP** *-N 2.0*
F1 Measure	0.64	0.59	0.67	0.49
True Negative Rate	0.65	0.59	0.67	0.49
False Negative Rate	0.11	0.13	0.10	0.16

For our evaluation, we used the same classification algorithms as for misclassification identification. In a similar fashion, we first tuned the parameters of the classifiers on the training datasets. Subsequently, we evaluated their performance on the 573 regions identified as *Misclassified* (refer to Figure 5.10). The results are provided in Table 5.5.



(a) In Regions (b) In Cells

Figure 5.11: Relabeling Results

Figure 5.11 displays the relabeling results for Logistic Regression (LR) classifier. We pick this classifier since as shown in Table 5.5 it achieves the best results. Again, we provide the numbers in regions and cells. Although we managed to find the true label for most of the regions flagged *Misclassified*, there is a considerable number of re-labeling errors.

One possible solution to decrease the number of incorrect predictions is to use class probabilities, instead of fixed membership. By default, the LR classifier assigns to an instance

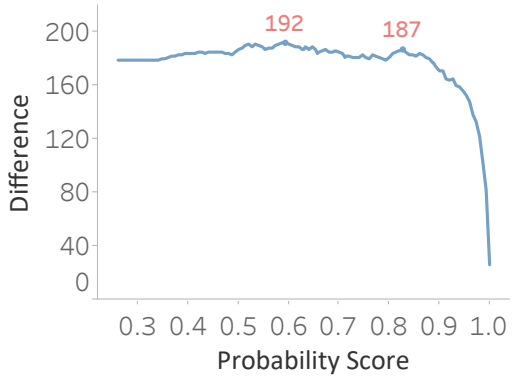


Figure 5.12: Confidence Score Analysis

the class (label) with the highest probability. We can intervene in this process, forcing the LR classifier to relabel only those regions for which the predicted probability is high. Effectively, this means setting a threshold for the class probabilities.

We assessed the validity of this approach, as shown in Figure 5.12. We have analyzed the probabilities assigned by the LR classifier during the relabeling task. For each region, we have recorded the highest class probability. We use the values in this list as thresholds. For each value, we identify the regions that got a probability score greater than or equal to this value. Among them, we count those that were correctly relabeled and those that were not. Subsequently, we record the difference. The largest difference is 192 and is achieved for threshold=0.59. For this threshold, we get 363 correct versus 171 incorrect predictions. However, we decided to be more conservative and set the threshold=0.83. We get a better trade-off since there are 113 wrong predictions and a considerable number of 300 correct predictions.

5.4 SUMMARY AND DISCUSSION

In the above sections, we outlined our strategies for repairing some of the incorrect classifications. The first one, discussed in Section 5.2, is based on immediate neighborhood patterns (rules). These intuitive and easy to implement patterns managed to correct 12% of the misclassified cells while introducing 1% new error. The second approach, from Section 5.3, is a refined three-step process. Initially, we standardize our cells into rectangular regions. Subsequently, we use a classifier to identify regions that contain misclassified cells. Afterward, we attempt to predict their true label, using another specialized classifier. Our evaluation shows that we achieve good results for the misclassification identification task. Specifically, on average we get a 64% true positive rate. Nevertheless, the results are not as encouraging for the subsequent task, i.e., relabeling. Therefore, we proposed to use a threshold for the probabilities predicted by the Logistic Regression classifier (the highest scoring model for relabeling). In this way, we showed that it is possible to reduce the number of re-labeling errors.

All in all, we can say that the proposed approaches are partially successful. They could be used to get rid of random noise, i.e., few misclassifications sparsely distributed within the sheet. However, these approaches are not appropriate for sheets having many misclassified cells. We encounter such sheets, in the dataset considered by this thesis. Therefore, we do not make use of these post-processing approaches in the remaining chapters.

TABLE DETECTION

6.1 A Method for Table Detection in Spreadsheets

6.2 Preliminaries

6.3 Rule-Based Detection

6.4 Genetic-Based Detection

6.5 Experimental Evaluation

6.6 Summary and Discussions

In the previous chapters, we discussed the layout analysis and post-processing steps. Here, we address table detection in spreadsheets. As mentioned before, our overall approach is bottom-up. For table detection, we build on top of the layout regions (see Section 4.4), which were created in the previous steps.

This chapter is based on three of our publications: In [86] we introduced our first table detection approach, which is based on rules. We search for specific arrangements of layout regions, typically found in spreadsheet tables. We have discovered these arrangements via a thorough empirical analysis. Subsequently, in [83], we proposed another approach, this time making use of a graph model. With this model, we encode the spatial arrangement of layout regions in the sheet. Then, we detect tables by means of rule-based graph partitioning. For this, we adapt, simplify, and extend the original rules from [86]. In our last publication [88], we make use of the same graph model (from [83]) but instead apply genetic algorithms and optimization techniques. Essentially, we propose a flexible approach that is not bound to fixed rules. Given a sample of annotated sheets, this approach can learn and adjust to the characteristics of the current dataset.

In the subsequent parts of this chapter, we summarize our published work. We begin with Section 6.1, where we first discuss approaches from related work. Then, we provide a visual overview of our table detection approach. Next, in Section 6.2 we formally introduce the proposed graph model. In Section 6.3, we discuss the approaches from our first two publications, [86] and [83]. We make sure to emphasize how the later approach incorporates elements from the former. In Section 6.4, we introduced our genetic-based approach. While the experimental evaluation for the above-mentioned approaches is discussed in Section 6.5 We conclude this chapter with Section 6.6, where we highlight our specific contributions on the task of table detection in spreadsheets.

6.1 A METHOD FOR TABLE DETECTION IN SPREADSHEETS

Most of the related works consider table detection as a trivial task [11, 29, 46]. This is because they foresee only vertical arrangements (top-down) of tables. Therefore, in such settings, one can simply rely on the identified *header* rows to spot the beginning of a new table and the end of the previous one from above. Moreover, one can choose to include or omit from these tables rows that contain *notes*, *titles*, *aggregations*, etc.

However, in this thesis, we address arbitrary arrangements for tables and layout regions in general. Therefore, the strategy used by related work is not applicable, as it would not be able to cope with horizontal and mixed arrangements. With such arrangements, one has to detect the first and last column, in addition to the first and last row of the tables. A potential solution is to use left-side hierarchies (i.e., *Attributes*[1] or *GroupHeaders*) in a similar fashion to how related work has used *header* rows. However, left-side hierarchies are present only in $15-17\%$ of the annotated sheets (refer to Table 3.3). Alternatively, one could consider empty columns as table separators. Yet, as emphasized by Takeaway 7, in Section 3.3, we often find empty columns inside the annotated tables, not just in-between them. Thus, a potential solution needs to go beyond the aforementioned heuristics and assumptions.

A recent publication from Microsoft Research Asia, Dong et al. [43], addresses table detection in spreadsheets, in the presence of arbitrary arrangements. As discussed in Section 2.3.1, this publication proposes a framework, called *TableSense*, that among other

[1]The label *Attributes* are defined and illustrated in Section 5.1

makes use of Convolutional Neural Networks (CNNs). The *TableSense* framework operates on a $h \times w$ matrix of cells (i.e. the input sheet), and 20 channels, one for each considered (cell) feature. From this input, the CNNs create a high level (abstract) representation of the sheet. This representation is subsequently fed to other modules of the framework, which are specialized for table detection. In particular, we highlight the use of a novel module, referred to as Precise Bounding box Regression (PBR). According to the authors, this module is CNN-based and capable of detecting the bounds of tables with high precision. The final output of *TableSense* is a list of predicted table regions, i.e., their bounding boxes. The overall approach is illustrated in Figure 6.1.

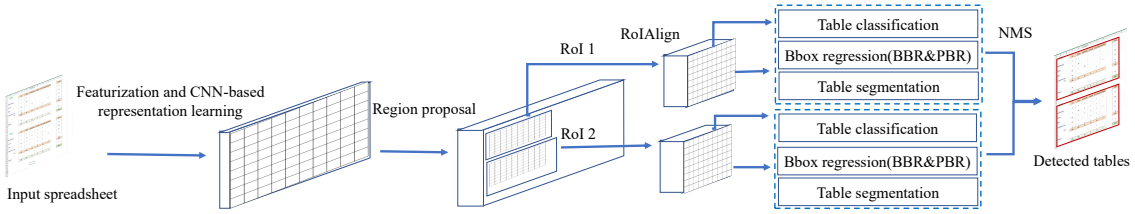


Figure 6.1: Framework of TableSense for Spreadsheet Table Detection [43]

The *TableSense* framework differs substantially from the approach proposed by this thesis. Instead of creating an abstract representation for the sheet, we use cell features to predict their layout function (one of the seven labels, as defined in Section 3.2.2). Then, we group adjacent cells of the same label to form strictly rectangular layout regions (defined in Section 4.4). We encode the spatial arrangement and relations between the regions using a graph model. Then we detect tables via graph partitioning. Specifically, we attempt to identify subgraphs (i.e., connected components of the input graph) that correspond to tables in the original sheet. The proposed approach is illustrated in Figure 6.2.

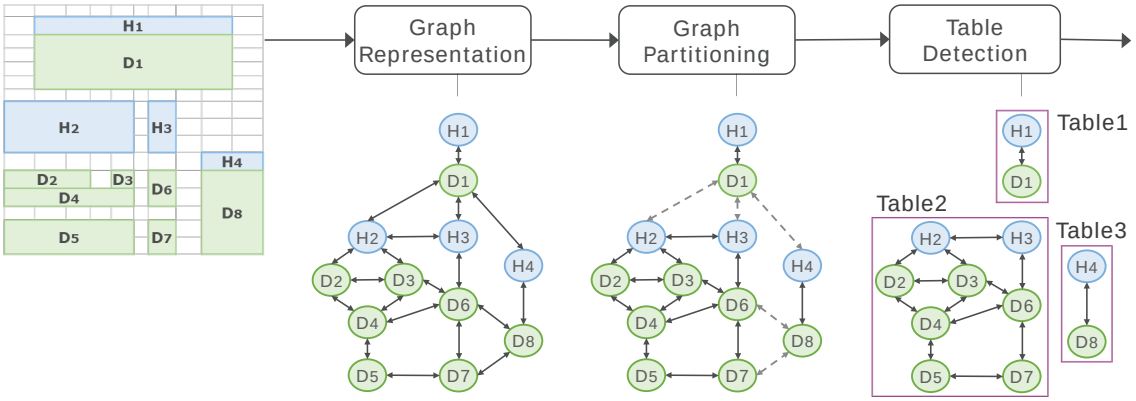


Figure 6.2: Overview of the Proposed Approach for Table Detection

On the top left corner of Figure 6.2, we illustrate a sheet with three tables. This sheet exhibits both horizontal and vertical arrangements. The layout regions are marked with the label of the cells that they enclose. Moreover, for better comprehension, we have numbered (indexed) the regions of the same label. Here, we consider only two layout functions *Data (D)* and *Header (H)*. As detailed and motivated in Section 6.2.3, we reduce the number of labels and layout regions, prior to table detection.

Subsequently, we construct a graph representation (model) for the given sheet. In Section 6.2.1, we formally define this model. Nevertheless, here we provide an intuition. As can be seen in Figure 6.2, the vertices of the graph correspond to the layout regions in the sheet. Edges are introduced between vertices when the corresponding regions are neighboring and aligned (i.e., they share row/s or column/s).

Having this model, the task becomes to identify subgraphs that correspond to real tables in the sheet. In Section 6.2.2, we formulate this task as a graph partitioning problem. In simple terms, the goal is to maintain edges that connect vertices (regions) of the same table, while omitting the rest (denoted with dashed lines, in Figure 6.2). The resulting connected components, having at least one Header and one Data vertex, constitute the predicted tables.

As mentioned previously, we proposed two approaches for partitioning the graphs, a rule-based (in Section 6.3) and a genetic-based (in Section 6.4). To the best of our knowledge, these approaches are novel, even within the field of Document Analysis and Recognition (DAR). Indeed, as stated in [97], graphs have been employed by the DAR community for various tasks, including table detection [108] and analysis [13]. Nevertheless, these works make use of graph re-writing techniques [110], instead of graph partitioning ones. In the most recent years, there is an emergence of graph neural networks [133, 136]. They have also been used for table detection or other similar tasks [106, 109]. There are parallels between these approaches and the ones proposed by this thesis. Yet, we clearly employ very different techniques. In particular, we highlight the use of genetic algorithms, in one of our proposed approaches.

In Section 6.5 we evaluate table detection via graph partitioning. As well as, we compare with the aforementioned related works. We observe that the approaches proposed by this thesis achieve substantial performance, even in the presence of misclassifications.

6.2 PRELIMINARIES

Here, we discuss concepts and processes that are relevant for the table detection approaches, outlined in Section 6.3 and 6.4. In Section 6.2.1, We begin with a formal definition of the proposed graph model. Subsequently, in Section 6.2.2, we formulate the detection of tables as a graph partitioning problem. Finally, in Section 6.2.3, we discuss two pre-processing actions that help us streamline the detection task.

6.2.1 Introducing a Graph Model

As discussed in the survey [97], graphs have been used to represent the structure and layout of documents. With such models, one can easily encode the spatial and logical relations between regions of the document. Specifically, the regions correspond to the vertices of the graph. Additionally, these vertices can be decorated with attributes that describe the properties of the corresponding regions, such as their size and location. Edges connect pairs of vertices (i.e., regions), which are physically or logically related. Again, one or more attributes can be attached to the edges, in order to describe these relationships.

In this thesis, we make use of such *attributed graphs*. In Figure 6.2, we illustrated this use with a simple example. Below, we formalize the proposed graph model. This model describes in a systematic manner the sheet layout. Moreover, as shown in the following sections, the proposed model allows the adoption of well-known algorithms and methods related to graphs.

Definition 1: Let $G(V, E)$ be a *directed* graph that captures the spatial interrelations of layout regions ($\mathcal{R}$) from a worksheet $\mathcal{W}$. There is a one-to-one correspondence between the set of vertices, V, and the set of layout regions, $\mathcal{R}$.

Furthermore, we carry the attributes of the layout regions into the graph. As mentioned in Section 4.4, the layout regions are homogeneous, when it comes to the label (i.e., layout function) of the enclosed cells. Therefore, we can assign this label also to the vertex that represents the region. Moreover, we encode the location of the region on the sheet. For this, we define the following functions.

Definition 2: The function $lbl : V \mapsto Labels$ maps the vertices of the graph to layout functions. Moreover, $rmin : V \mapsto \mathbb{N}_{>0}$ and $rmax : V \mapsto \mathbb{N}_{>0}$ return respectively the minimum and maximum row number (of the corresponding layout region). Equivalently, $cmin$ and $cmax$ do the same for the column numbers.

The next step is to create edges between the vertices of the graph. Here, our aim is to identify spatial relations such as *top of*, *bottom of*, *left of*, and *right of*. In other terms, we capture the relative location of other regions with respect to the current region in the following four directions: Top, Bottom, Left, and Right. Therefore, we define the following function.

Definition 3: The function $dir : E \mapsto \{Top, Bottom, Left, Right\}$ maps edges to directions. For an edge $(v, u) \in E$ the result of this function communicates the direction that u is the neighbor of v.

Nevertheless, we are not interested in all spatial relations. Instead, we focus only on the nearest neighboring regions for each direction. To better illustrate the creation of edges, below we outline the steps for the identification of the nearest neighbors on the Top direction for a vertex $v \in V$.

$$T_v = \{u \in V |\, rmin(v) > rmax(u) \texttt{ and not}(cmin(v) > cmax(u) \texttt{ or } cmax(v) < cmin(u))\}$$

As shown in the equation above, we identify all vertices whose maximum row is less than the minimum row of v. On the same time, we enforce that the selected vertices span, at least partially, the same columns as v. To get the nearest vertices we use the distance functions discussed below.

Definition 4: For each direction we define a distance function. Let $tdist := (rmin(v) - rmax(u)) - 1$ and $bdist := (rmin(w) - rmax(v)) - 1$ calculate respectively the distance from Top ($u \in T_v$) and Bottom ($w \in B_v$) neighbors of v. Likewise, we define the function $ldist$ for Left and $rdist$ for Right direction.

As can be seen above, depending on the direction, the distance is measured either in number of rows or in number of columns. We subtract 1 in order to record the distance as 0 for adjacent vertices (i.e, regions from consecutive rows/columns). Moreover, note that hidden rows and columns are not considered in the calculation of the distance.

For the rule-based table detection method, discussed in Section 6.3, we measure distance using the above functions. Instead, in Section 6.4, for the genetic-based method, we consider the height/width of rows/columns to calculate the distance. Specifically, in Excel, the row height is measured in points, while column width is measured in units of 1/256th of the standard font character width.

Definition 5: Let the function $rheight := \mathbb{N}_{>0} \mapsto \mathbb{R}_{\geq 0}$ return the height for a given row number. Then we can calculate the distance of vertex v from its Top neighbor, vertex

u, as $\sum_{i=rmax(u)+1}^{rmin(v)-1} rheight(i)$. In similar fashion, we can calculate the distance to any Bottom neighbor of vertex v. Instead, for Left and Right neighbors, we use the function $cwidth := \mathbb{N}_{>0} \mapsto \mathbb{R}_{\geq 0}$, which returns the width for a given column number.

Regardless of the distance functions, the aim is to find the nearest neighbors for the current vertex (region). Previously, we defined T_v as the set of Top neighbors for a vertex v. Below, we continue by defining the set of its *nearest* Top neighbors:

$$T_v^{'} = \{n \in T_v | \; tdist(v, n) = \min_{u \in T_v} tdist(v, u)\}$$

We can now create a directed edge (v, n), for every $n \in T_v^{'}$. The same can be performed for the nearest neighbors of v, in the remaining directions. Furthermore, following the above definitions, we can analyze all pairs of vertices and populate the set of edges E.

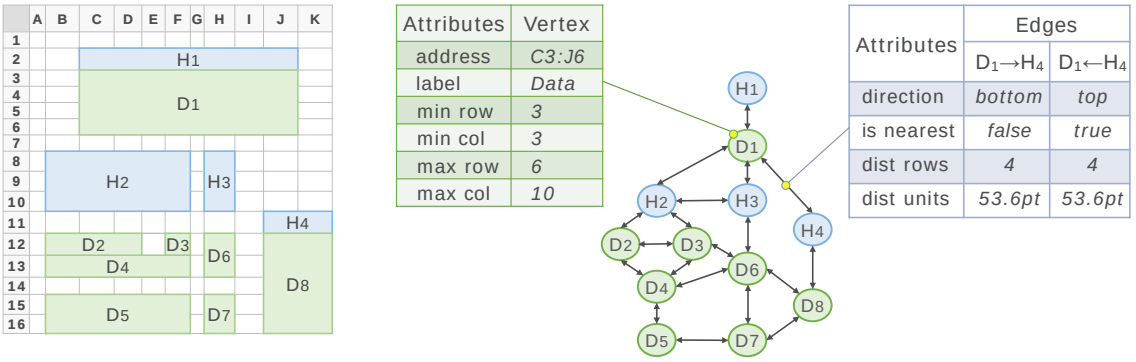


Figure 6.3: The Proposed Graph Representation

Figure 6.3 shows the graph representation for an example sheet. Note that the edges are depicted using bidirectional arrows. We use such arrows since for an edge $(v, u) \in E$ there always exists an equivalent edge $(u, v) \in E$.

Typically, the property of being the nearest neighbor of a vertex holds the other way around, from the viewpoint of the neighbor. However, there are a few exceptions. In Figure 6.3, the nearest Top neighbor for H_4 is D_1, but the nearest Bottom neighbors for D_1 are H_2 and H_3. For such cases, we enforce symmetry. This means we keep the edge $D_1 \mapsto H_4$, even though H_4 is not the nearest neighbor for D_1.

Having acknowledged the aforementioned cases, we can now define the set of edges E in a more concise manner.

Definition 6: For a pair of edges $(v, u) \in E$ and $(u, v) \in E$, the following conditions must hold: Let $\mathcal{N}_v$ be the set of nearest neighbors of v from all directions, i.e., $T_v^{'} \cup B_v^{'} \cup L_v^{'} \cup R_v^{'}$. Similarly, we define $\mathcal{N}_u$ for the vertex u. Then at least one of the following is true: $v \in \mathcal{N}_u$ or $u \in \mathcal{N}_v$. In other terms, it is never the case that both $v \notin \mathcal{N}_u$ and $u \notin \mathcal{N}_v$.

With this definition, we conclude the formalization of the proposed graph model. In the subsequent sections, we discuss how to partition the constructed graph representations, with respect to the table detection task in spreadsheets.

6.2.2 Graph Partitioning for Table Detection

As outlined by the surveys [27], [81], and [113], graph partitioning and clustering are well-studied problems with many applications. Using the representation described in Section 6.2.1, we as well formulate the task of detecting tables in spreadsheets as a graph partitioning problem (GPP).

The aforementioned surveys, [27] and [113], outline a broad range of methods for GPP. While the survey [81] focuses entirely on the use of genetic algorithms for graph partitioning. Based on these works, in this thesis, we propose two approaches. In Section 6.3, we discuss a solution that makes use of rules, for partitioning the input graph. While, in Section 6.4, we employ genetic algorithms and optimization techniques.

Concretely, the input for the two proposed approaches is a graph $G = (V, E)$, that encodes the layout of a given sheet. Here, V is the set of vertices (layout regions), and E is the set of edges (spatial relations). We partition V into disjointed subsets, where $V_1 \cup ... \cup V_k = V$ s.t. $V_i \cap V_j = \emptyset$ for all $i \neq j$. Typically, the number of partitions k for GPP is fixed in advance [27]. This is not feasible in our case, since we are not aware of the number of tables in the sheet, beforehand. Thus, in this work, the number of partitions can be $1 \leq k < |V|$.

In basic terms, we partition by deactivating (omitting) edges from the set E. We illustrate this method in Figure 6.2. When omitting edges we get partitions that are connected components. This is favorable since it eliminates unlikely solutions. Concretely, we expect vertices (regions) of the same table to be close to each other and most importantly connected via path/s. Thus, it is important to enforce a connectivity constraint. This constraint is inherent for the selected partitioning method.

Having the above-mentioned formulation, the goal becomes to find the optimal partitioning of the graph. Specifically, the resulting partitions (i.e., connected components) must correspond to tables in the sheet. However, often due to misclassifications, the input graph might contain vertices that do not necessarily belong to a table. For instance, cells annotated as Note, Title, and Other can be misclassified as Data or Header. Such cases are discussed in more detail in Section 6.2.3. Thus, in the end, the task becomes that of correctly identifying not only true tables but also non-tables, in the given graph representation. In other terms, we attempt to detect tables even in the presence of misclassifications.

The next section discusses some additional actions that we take prior to table detection. Specifically, we attempt to simplify the problem by reducing the considered labels and layout regions.

6.2.3 Pre-Processing for Table Detection

Before constructing the graph representation, we take several actions that help us to simplify the table detection problem. These actions occur after classifying the cells, in the input sheet. However, we do not consider any of the optional post-processing approaches from Chapter 5. Thus, there is no alteration for the methods and the results discussed in Section 4.3. Moreover, these actions do not affect the tasks coming after table detection, which are discussed in Chapter 7. In other words, the proposed adjustments are relevant only for the approaches discussed in this chapter.

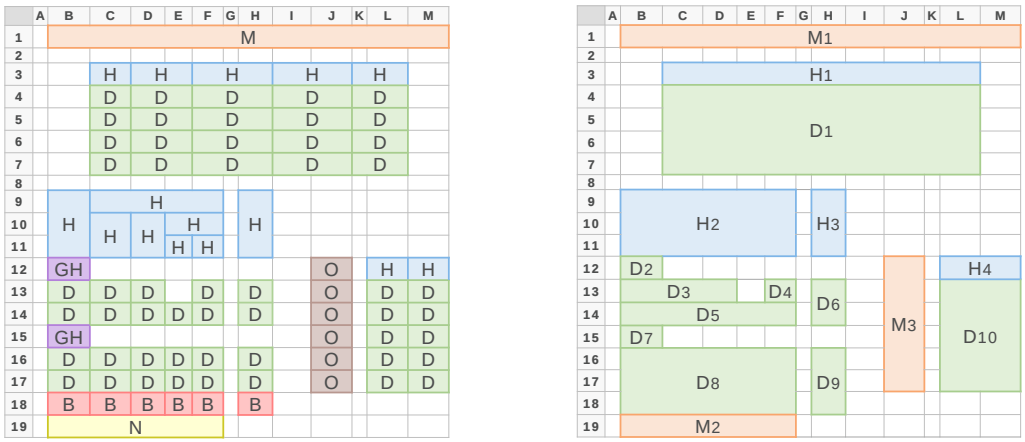
Label Reduction

The first action takes place after cell classification. Concretely, given the classification output, we reduce the number of labels from 7 to 3. We achieve this by introducing three meta-classes: DATA, HEADER, and METADATA. Cells that were classified as *Data (D)*, *GroupHeader (GH)* and *Derived (B)* are assigned to the class DATA. In addition, cells classified as *Title (M)*, *Note (N)*, and *Other (O)* are collectively treated as METADATA. Finally, the class HEADER carries only cells that were classified as *Header (H)*.

After performing these reductions, we proceed to create the layout regions. The procedure remains the same, as it was outlined in Section 4.4. Basically, we treat the meta-classes the same as we treated the classification labels. This means we group adjacent cells having the same meta-class, to form coherent and strictly rectangular layout regions (refer to Section 4.4).

In Figure 6.4 we illustrate the aforementioned procedure. Figure 6.4.a shows the classification results, where each non-empty cell is assigned the predicted label. As discussed above, we reduce these labels and subsequently construct the layout regions. These regions are shown in Figure 6.4.b. For brevity, we use a single letter to denote the meta-classes: DATA (D), HEADER (H), and METADATA (M). Moreover, for better comprehension, we have enumerated the individual regions of the same class.

Here, we highlight the effect that label reduction had on the given example. Notably, cells classified as *Derived (B)* and *Data (D)*, can now be grouped together, since they belong to the same meta-class. Therefore, in Figure 6.4.b we get the DATA regions D_8 and D_9, which span three rows each. In addition, we get two DATA regions, D_2 and D_7, from the cells classified as *GroupHeader (GH)*. While, the cells classified as *Title (M), Note (N)*, and *Other (O)* result in three METADATA regions.



(a) Original Classified Cells (b) Layout Regions after Reducing Labels

Figure 6.4: Reducing Labels Prior to Table Detection

Overall, with the reduction, we get fewer labels, and also a smaller number of regions. Most importantly, as can be seen from Figure 6.4, the reduction does not alter the interpretation of the sheet, with regard to the table detection task. Specifically, the sheet contains three tables, which are clearly distinguishable both in Figure 6.4.a and in Figure 6.4.b.

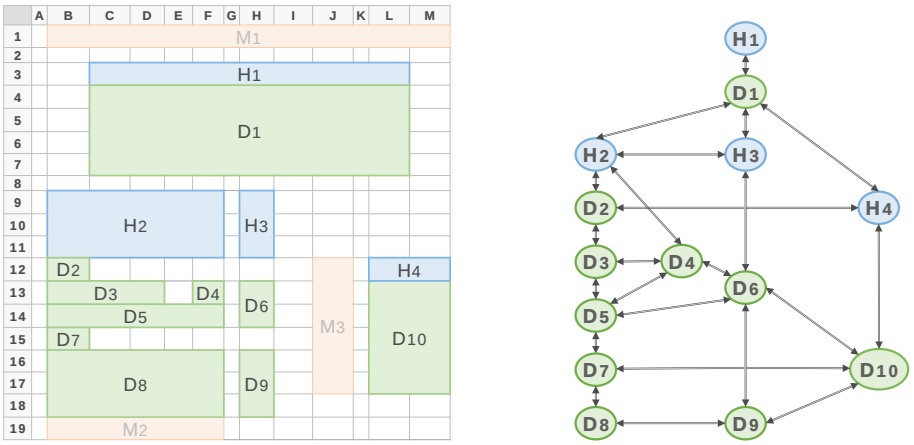
Intuitively, with this first pre-processing action, we have reduced the table detection problem into three main components. The HEADER and DATA regions are the building blocks for tables. While the METADATA regions represent extra (non-essential) parts

of the sheet. Based on this formulation, we can say the main goal of the table detection task is to identify HEADER and DATA regions that belong together, i.e., to the same table. Subsequently, one can attempt to discover the relationship between the METADATA regions and the detected tables in the sheet. However, as discussed in the next section, we do not address the latter challenge in this thesis. Note, this is an open question also for related work [11, 29, 43]. Thus, it still remains a task for future research projects.

Omitting METADATA

Here, we discuss the second pre-processing action, which involves METADATA regions. Clearly, the cells (classified as *Title*, *Note*, and *Other*) in METADATA regions hold information that can potentially help us interpret the tables more accurately. Nevertheless, this information might refer to the whole sheet or multiple parts of the sheet. Thus, it is not necessarily associated with a single table. For this reason, we do not regard the METADATA regions as a core part of the tables. Instead, we detect tables using only the HEADER and DATA regions.

In fact, we completely omit the METADATA regions when constructing the graph representation. This means we treat these regions as if they were not present in the sheet. We illustrate this fact in Figure 6.5. As can be seen, there are no corresponding vertices for the METADATA regions. Furthermore, the regions H_4 and D_{10} can now be connected to other regions on the left, since region M_3 is omitted.



(a) Omitting METADATA Regions (b) The Graph Representation

Figure 6.5: Building a Graph after Pre-Processing

The omission of METADATA regions is favorable when it comes to the proposed table detection approach. First, it means a smaller number of vertices. This, typically translates into a smaller number of edges, too. Thus, the computational overhead decreases, since we ultimately process fewer edges when partitioning the graph (refer to Section 6.2.2). Secondly, the omission of METADATA can introduce more space in-between the tables. This can be seen in Figure 6.5, for the two tables in the bottom rows, which are arranged horizontally. After omitting M_3, the tables are separated by three columns. This fact will be reflected in the edges of the graph, which encode the distance between the regions. Essentially, the more distant the tables (i.e., their regions) are from each other, the better are the chances of accurately detecting them as separate units.

However, this pre-processing action does not come without challenges. As noted before, we operate on the classification results. Thus, due to wrong predictions, the cells can be assigned to the wrong meta-class. Which means, the constructed regions will not reflect the true layout of the sheet. In other words, the output of the second pre-processing action is not guaranteed to be optimal. After omission, the graph might contain false DATA or HEADER vertices. At times, this involves cells from METADATA regions. For instance, in Figure 6.5, if any of the cells in regions M_{1-3} was misclassified as *Header, Data, Derived,* or *GroupHeader*, we would get extra (false) vertices in the graph. The other way around, we might also get "incomplete" graphs, when some of the cells are mistakenly omitted (i.e., they were falsely allocated to METADATA regions).

Nevertheless, the classification results from Section 4.3.6 show that these cases are not frequent. This is especially true, for the second case, i.e., incomplete graphs. We get high classification accuracy for $Data$ and $Header$ label. Moreover, when $Derived$ and $GroupHeader$ cells are misclassified, they usually get mistaken for $Data$ cells. Thus, anyhow they will be assigned to the same meta-class, i.e., DATA. For the remaining three labels, the meta-class errors are more common. Indeed, we observe a considerable number of cases where $Other$ and $Note$ cells are misclassified as $Data$ cells. Moreover, $Title$ cells are occasionally mistaken for $Header$ cells. Therefore, sometimes graphs contain extra or false vertices, i.e., DATA and HEADER regions that in reality are completely or partially METADATA.

For the proposed table detection approaches, in Section 6.3 and 6.4, we take into consideration the above-mentioned cases. In fact, each approach has a slightly different strategy to address misclassifications.

Summary

In the previous section, we proposed two pre-processing actions that simplify the table detection process. The first action results in fewer labels (i.e., meta-classes), making it easier to conceptually formulate the detection problem and the candidate solutions. Moreover, both actions, but especially the second one, can decrease the number of vertices and edges in the graph representation. This can speed up the search for tables.

Nevertheless, even after pre-processing, there are still challenges for the table detection task. In the previous section, we highlighted the presence of misclassifications. In the subsequent sections, we attempt to tackle this and other challenges.

Note, for simplicity, hereinafter we treat the meta-classes the same as *labels*. Therefore, we do not refer to them anymore using their all upper case form. Instead, we use the following form: $Data(D)$, $Header(H)$, and $Metadata(M)$.

6.3 RULE-BASED DETECTION

In [86], we outlined our first attempt to detect tables in spreadsheets. The TIRS (Table Identification and Reconstruction in Spreadsheets) approach studies the spatial arrangement of layout regions in the sheet. Specifically, we have empirically discovered typical table layouts, shown in Figure 6.6. Then, using a series of rules and heuristics, TIRS detects these layouts in the classified sheets.

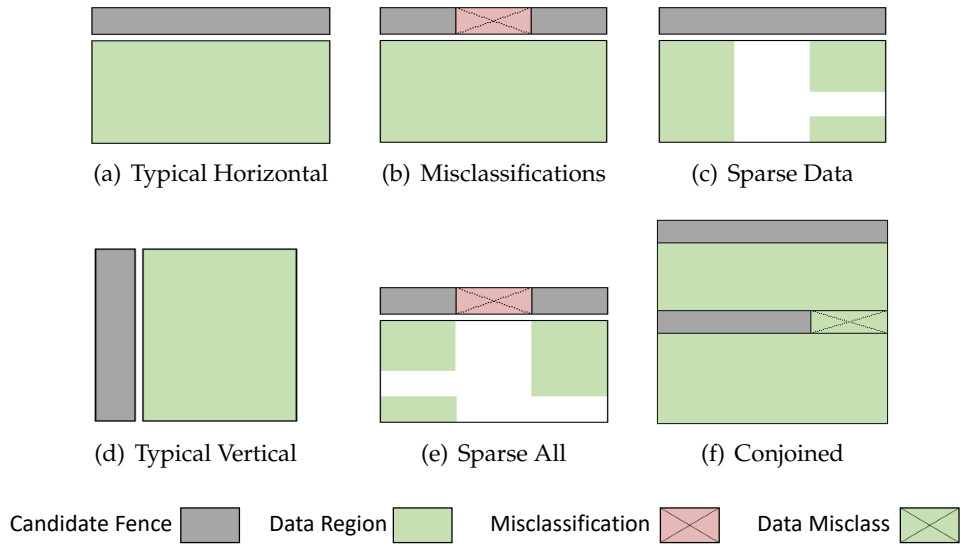


Figure 6.6: Table layouts, cases b, c, e, and f also occur for tables with vertical fences

An important concept in TIRS is the *fence*, a term borrowed from [6]. In Figure 6.6, they are shown as dark grey rectangles. Fences are either Headers or left-hierarchies. Basically, fences help us detect the start of a new table, horizontally or vertically.

The proposed rules attempt to detect tables even in the presence of irregularities. As shown in Figure 6.6, TIRS takes into consideration the occurrence of sparse tables (i.e., multiple fences and Data regions in the same table). Furthermore, TIRS addresses misclassifications that occur in fences and elsewhere.

Nevertheless, TIRS performs worst than the other table detection approaches, discussed in this chapter. One of the main reasons is that TIRS does not make use of a structured representation, such as the graph model from Section 6.2.1. This results in complex rules, which are difficult to debug and extend.

Therefore, in [83] we introduced an approach, referred to as RAC (Remove and Conquer). This approach relies on the proposed graph model. Another novelty of RAC is that the rules go beyond the immediate neighborhood of the region. In other terms, it absorbs significantly more context than the TIRS approach. However, RAC does not make use of left-hierarchies (i.e, vertical fences). It only uses Header regions (i.e., horizontal fences) and Data regions. Nonetheless, RAC incorporates and simplifies many of the rules originally proposed for TIRS [83].

In the subsequent sections, we proceed by discussing in detail the RAC approach. However, in Appendix B, one can refer to the original TIRS approach.

6.3.1 Remove and Conquer

For each worksheet in our dataset, we classify the cells, and then we perform the preprocessing actions, described in Section 6.2.3. Subsequently, we construct a directed graph, as described in Section 6.2.1. Our rule-based algorithm, RAC, processes these graphs individually and outputs for each one of them a set of proposed tables $\mathcal{P}$. In addition to this, in a separate set U, RAC returns vertices that could not form tables.

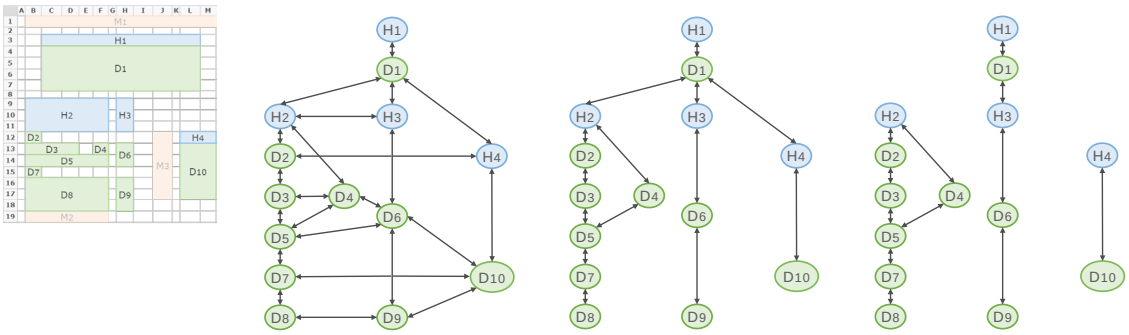
The pseudocode for the proposed approach can be found in Algorithm 6.1. We reuse in this algorithm the functions defined in Section 6.2.1, for the graph model. We also illustrate the steps of RAC, in Figure 6.7 and 6.9. Here, we revisit the example originally introduced in Figure 6.5 and discussed in Section 6.2.3. Lastly, in the following section, we use the term vertex and region interchangeably. They both refer to the layout regions of the sheet (see Section 4.4).

Horizontal Groups

The RAC approach addresses sheets with one or many tables, diverse layouts, and arbitrary arrangements. However, transposed tables (with vertical Headers, i.e., arranged column-wise) are outside of the scope of this work. As stated in Section 3.3, these cases are rare. Therefore, RAC always assumes that Headers cells are on the top rows of the tables, and the Data cells in the lower rows.

This fundamental assumption defines the RAC approach. We remove edges in a way that preserves this top-down arrangement within tables. In other terms, we prioritize Top and Bottom edges, over Left and Right edges.

As seen in Figure 6.7, for sheets exhibiting horizontal arrangements of tables, we get Left and Right edges in the graph representation. These edges typically connect regions from different tables, but can also be found within the same table. Yet, for the latter case, we expect Top and Bottom edges as well. Thus, removing the Left and Right edges (lines 2-4, Algorithm 6.1) should mostly impact the inter-table connections. Nevertheless, we take measures to protect some of the intra-table connections, by not removing edges when the distance is $zero$ (i.e., adjacent regions with no other column in-between them).



(a) Input Graph (b) Horizontal Groups (c) Vertical Groups

Figure 6.7: The RAC Approach

The above-mentioned action, i.e., the omission of Left and Right edges, constitutes the first step of RAC. The resulting connected components are referred to as *horizontal groups*. Note, it is not always the case that we get multiple such groups. For instance, in Figure 6.7.b there is still only one connected component after the edge omissions.

Regardless, in the next step, each one of the horizontal groups is subdivided vertically. For this, we utilize the enclosed Header regions as separators. Ultimately, we get subgroups, which are simply referred to as *vertical groups*. These should already resemble valid tables. Nevertheless, in the final steps, RAC attempts some further corrections, which can improve the quality of the detected tables.

Vertical Groups

Before outlining how RAC subdivides vertically, we need to address the implications arising from misclassified cells. Consider the examples, in Figure 6.8. On the left (Figure 6.8.a), regions are formed using the true layout function of the cells. While, in Figure 6.8.b, we use the predicted functions. Here, the cells in the regions D_1 and H_4 were incorrectly classified.

(a) True Layout Roles (b) Misclassified Cells

Figure 6.8: The impact of misclassifications

RAC needs to infer that all regions in Figure 6.8.b belong to one table, instead of many. The alignment between Header and Data regions provides some hints. The same is true also for the size of these regions. In particular, we prioritize larger Header regions.

We use these insights in lines 5-21 of Algorithm 6.1, which outline the second step of RAC. For each connected component (G^S), i.e., a horizontal group, we seek to pair Header regions with Data regions. As shown in line 7 of Algorithm 6.1, we perform our search for tables from bottom to top. We sort vertices in descending order of their maximum row, followed by the ascending order of their minimum row. Note, for Figure 6.8.b, this means H_2 will be ordered before D_1, H_3, and H_1.

Algorithm 6.1: The RAC (Remove and Conquer) Approach

Input: G: graph representation of a worksheet
Output: $\mathcal{P}$: proposed tables, U: other undetermined
1: $\mathcal{P} \leftarrow \emptyset; U \leftarrow \emptyset$;
2: $E_l \leftarrow \{e \in E | dir(e) = Left$ **and** $ldist(e) > 1\}$
3: $E_r \leftarrow \{e \in E | dir(e) = Right$ **and** $rdist(e) > 1\}$
4: $E \leftarrow E \setminus (E_l \cup E_r)$
5: **for all** $G^S \in getComponents(G)$ **do** // G^S = (S,E_S)
6: $LQ \leftarrow NIL$ // holds Q of last valid Header
7: $S' \leftarrow sortVertices(S)$ // descending order max row, ascending order min row
8: $S'_H \leftarrow \{v \in S' | lbl(v) = Header\}$
9: **if** $|S'_H| > 0$ **then**
10: **for all** $h \in S'_H$ **do**
11: **if** $h \in LQ$ **then continue**
12: $TQ \leftarrow \{s \in S' | \, rmin(s) \geq rmin(h)\}$ // the TQ denotes the temp Q set
13: $Q \leftarrow \{s \in TQ | \, hasRestrictedPath(s, h, E_S, TQ)\}$
14: **if** $isValid(h, Q, 0.5)$ **then**
15: $\mathcal{P} \leftarrow \mathcal{P} \cup \{LQ\}$
16: $S' \leftarrow S' \setminus Q$
17: $LQ \leftarrow Q$
18: **else if** $LQ \neq NIL$ **then**
19: **if** $|Q| = 1$ **and** $isAligned(h, LQ)$ **then**
20: $LQ \leftarrow LQ \cup \{h\}$
21: $\mathcal{P} \leftarrow \mathcal{P} \cup \{LQ\}$ // when the *for all* loop ends
22: $U \leftarrow U \cup S'$ // remaining unpaired
23: $\mathcal{P}, U \leftarrow handleOverlapping(\mathcal{P}, U)$
24: **for all** $u \in U$ **do** // find nearest table left or right
25: $N, dist \leftarrow getNearestVertices(u, (E_l \cup E_r))$
26: $\mathcal{P}' \leftarrow \{P \in \mathcal{P} | \, 0 < |N \cap P|\}$
27: **if** $|\mathcal{P}'| = 1$ **and** $dist \leq 2$ **then**
28: $P \leftarrow P \cup \{u\}$, where $P \in \mathcal{P}'$
29: **return** $\mathcal{P}, U$

We process each Header vertex h individually, as shown in lines 10-20. If h is not already paired (line 11), we proceed to identify vertices having a minimum row that is greater than or equal to that of h (line 12). We denote this collection of vertices (including h) as TQ. All together, these vertices have the potential to form a valid table. However, we also need to handle scenarios like in Figure 6.7.b, where D_{10} satisfies the above condition for H_2 or H_3, even though they are not part of the same table. Thus, we additionally ensure that there is a direct path from h to the other vertices in TQ. Note, this path must involve only vertices from the TQ set. With this additional constraint, as shown in line 13, we get the set Q. This set becomes the base for a candidate table.

Line 14 checks the validity of a Header (discussed in more detail later). Vertices paired with a valid Header are subtracted from S', the list of sorted vertices. However, we do not append Q to $\mathcal{P}$, yet. Consider, the scenario in Figure 6.8.a. Pairing H_3 with D_1 could form a seemingly complete table, but it leaves H_1 and H_2 out. Thus, in lines 18-20 we identify Headers having no other vertex to pair with (i.e., only h satisfies the conditions in lines 12-13). We typically append such Headers to the Q of the last valid Header, denoted as LQ. However, before that, we make sure that these Headers have column/s in common with one or more vertices in LQ. In other terms, we examine their alignment.

Valid Headers

In Figure 6.7.c, vertex H_3 remains connected with D_1. This is because H_3 is not a valid Header, and it could not act as a separator. We handle this case in the subsequent steps of RAC. Here, with Algorithm 6.2, we outline the validity check for Headers.

First, we check that there are vertices below the specified Header h (line 1, Algorithm 6.2). Then, in line 2 we identify Headers in Q that span one or more rows in the range $[cmin(h), cmax(h)]$. The set $Q_{\mathcal{H}}$ acts as a composite candidate Header for the potential table. Note, this set includes h itself. We calculate the alignment ratio of $Q_{\mathcal{H}}$ with the rest of the vertices (lines 3-8). Intuitively, with this metric, we avoid false candidates, i.e., cases where $Q_{\mathcal{H}}$ shares very few columns with $Q \setminus Q_{\mathcal{H}}$. However, we have to also account for the occurrence of misclassifications. They can reduce the original alignment of the Header and Data regions. Thus, in the end, we set the alignment threshold as ≥ 0.5.

Finally, in line 8, we additionally check whether $Q_{\mathcal{H}}$ and the other regions span at least two columns. This is because we consider true tables only those having at least two rows and two columns. Clearly, this condition is in addition to having at least a Header and a Data region. Coming back to Figure 6.7.c, regions H_3, D_6, and D_9 collectively span only one column. Therefore, H_3 is not valid.

Algorithm 6.2: Check if Header is Valid

Input: h: a Header vertex, Q: vertices associated with h, and th: threshold for alignment ratio
Output: $True$ if h is valid, $False$ otherwise

1: **if** $|\{q \in Q | rmin(q) > rmax(h)\}| > 0$ **then**
2: $\quad Q_{\mathcal{H}} \leftarrow \{q \in Q |\ lbl(q) = Header$ **and** $rmin(q) \leq rmax(h)$ **and** $rmin(q) \geq rmin(h)\}$
3: $\quad X \leftarrow \emptyset$; $X' \leftarrow \emptyset$
4: $\quad$ **for all** $u \in Q_{\mathcal{H}}$ **do**
5: $\quad\quad X \leftarrow X \cup \{x \in \mathbb{N} | cmin(u) \leq x \leq cmax(u)\}$
6: $\quad$ **for all** $v \in Q \setminus Q_{\mathcal{H}}$ **do**
7: $\quad\quad X' \leftarrow X' \cup \{x \in \mathbb{N} | cmin(v) \leq x \leq cmax(v)\}$
8: $\quad$ **return** $\dfrac{|X \cap X'|}{|X'|} \geq th$ **and** $|X| > 1$ **and** $|X'| > 1$
9: **else**
10: $\quad$ **return** $False$

Unpaired Vertices

Starting from line 9 of Algorithm 6.1, we identify horizontal groups that do not contain a Header vertex and store them in U (see line 22). Nevertheless, we can get unpaired vertices, even from horizontal groups that have Headers. For instance, there are cases where none of the enclosed Headers is valid. Therefore, we can not create any candidate table. Which means all vertices from this group are moved to U.

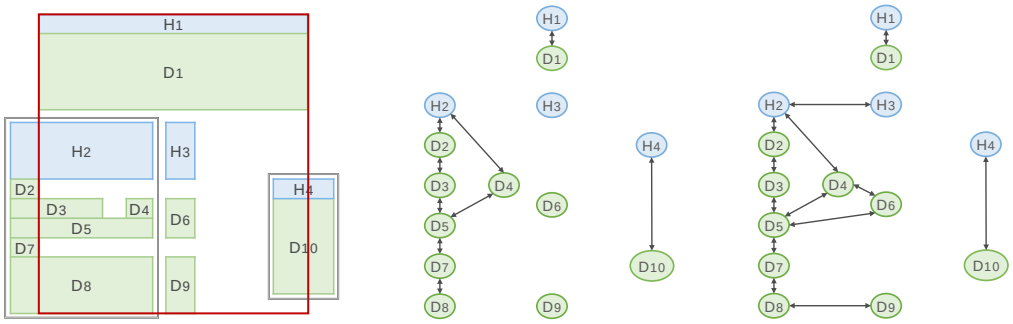
Post-Corrections

In line 23, of Algorithm 6.1, we deal with a special scenario. Occasionally, there are overlaps between the resulting vertical groups. In fact, to detect these overlaps, we use the minimum bounding rectangles (MBRs) of these groups. For example, overlaps occurs for the vertical groups in Figure 6.7, since the vertex H_3 remains connected to D_1. In Figure 6.9.a, we illustrate the situation using the original regions.

The function in line 23, handles such cases. If the overlap is between two tables arranged horizontally, it will attempt to merge them. Otherwise, it identifies the vertices causing the overlap. These vertices are moved from the proposed tables to U, for further processing. This is illustrated in Figure 6.9.b, where the vertices H_3, D_6, and D_9 are unpaired.

The last step of RAC, from lines 24-28 of Algorithm 6.1, attempts to pair vertices in U with the nearest table on their Left or Right. Nevertheless, in line 26 we set a threshold. We pair vertices with tables only when the distance is ≤ 2 columns. Note, in rare cases, there might be multiple nearest tables. We do not handle such scenarios, i.e., the vertex remains in U.

For the running example, in Figure 6.9.c, we show the results of the last step. The proposed table (i.e., vertical group) on the left is the nearest for H_3, D_6, and D_9. Thus we re-introduce the LEFT and RIGHT edges for these vertices. The three connected components from Figure 6.9.c constitute the proposed tables. In this case, they correctly match the true tables in the sheet.



(a) Vertical Group MBRs (b) Handle Overlaps (c) Pair Left and Right

Figure 6.9: The RAC Approach

6.4 GENETIC-BASED DETECTION

In this section, we go beyond the fixed rules, from Section 6.3. We opt for a transferable and flexible approach, which can be adjusted to meet the specific characteristics of a new (unseen) spreadsheet datasets. Clearly, such adjustments should require minimum effort. Therefore, we propose a mostly automatic approach, which can be tuned based on examples (i.e., a sample of annotated sheets).

Again, we use a graph model to encode the layout of the input sheet. However, we search for the optimal partitioning of the graph, based on an objective function. With this function, we quantify the merit of candidate partitionings. Then, using a genetic algorithm, we efficiently traverse the search space to identify the partitioning giving the best score (i.e., the global optimum). Intuitively, this partitioning should have the partitions (i.e., connected components) that correspond precisely to the tables in the sheet.

The objective function, defined in Section 6.4.4, brings together several metrics, which measure different aspects. We study a candidate partitioning as a whole, as wells as the individual partitions that compose it. Nevertheless, some aspects are more important than others. Therefore, the proposed metrics are weighted, based on their relevance. In fact, we determine the weights automatically. For this, we use a training sample of annotated sheets. In this way, we can adjust the objective function (i.e., the weights) to match the current dataset.

In Section 6.4.3, we define the metrics used in the objective function. However, before that, in Section 6.4.1, we briefly discuss the graph model adapted for this approach. We highlight a few differences with the model used in the rule-based approach (Section 6.3). Moreover, in Section 6.4.2 we introduce the concept of *Header clusters*. This concept is relevant since it used in many of the considered metrics.

6.4.1 Undirected Graph

As with the rule-based approach, we perform the pre-processing actions from Section 6.2.3. Then we represent the layout of the sheet with the graph model from Section 6.2.1. However, for the genetic-based approach, we additionally convert the graph model into undirected. In Figure 6.10, we illustrate the updated model. Essentially, a single undirected edge describes the relationship between two vertices. This relationship is, in fact, the orientation of alignment for the corresponding layout regions. Therefore, we replace the *left of* and *right of* edges with an undirected *horizontal* edge. While the *top of* and *bottom of* edges we replace with an undirected *vertical* edge. In addition, these undirected edges encode the distance (denoted as *dist*) between the regions. For this, we sum the width/height of columns/rows that separate the given regions.

6.4.2 Header Cluster

Let $P = \{V_1, ..., V_k\}$ be a candidate partitioning of the input graph $G = (V, E)$. As mentioned before, the objective function studies, among others, the individual partitions that compose P. Therefore, in Section 6.4.3, we define metrics that capture various aspects for each partition.

Many of the proposed metrics use the concept of *Header cluster*. Concretely, with this concept, we address partitions that contain multiple Header regions. This might occurs

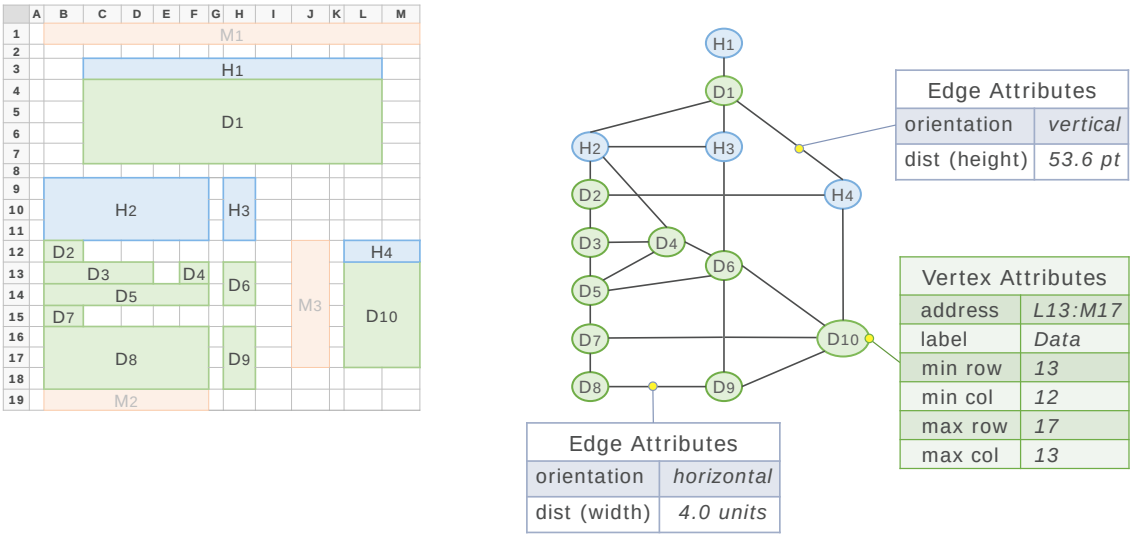


Figure 6.10: Graph Model for the Genetic-Based Approach

when a partition is of bad quality. That means, it brings together Header regions that otherwise belong to different tables. Misclassifications are another factor, to be considered. The given partition might be correct (i.e., corresponding to a table), but some of the enclosed regions were falsely labeled as Header. Finally, there are also cases with no misclassifications, but we still find multiple Header regions in a correct partition. For example, in Figure 6.10, the regions H_2 and H_3 belong to the same table.

We handle the above-mentioned cases, with the help of Header clusters. We create these clusters using the information encoded in the vertices of the graph model, shown in Figure 6.10. Specifically, for each partition, we examine the rows covered by the enclosed regions. We denote as R^d the set of row indices, where we find Data regions. Similarly, R^h records the row indices for Header regions. Then the set difference $R^{d*} = R^d - R^h$ gives us the rows covered only by Data regions. We use these rows, to cluster the Header regions, in the partition. Specifically, two Header regions are part of the same cluster, unless a row $r \in R^{d*}$ stands in between them.

Below, we illustrate Header clusters, with a concrete example. Suppose that the graph from Figure 6.10, is divided into two arbitrary partitions: $V_1 = \{H_1, D_1, H_2, H_3, H_4\}$ and $V_2 = \{D_2, D_3, \dots, D_{10}\}$. The second partition does not contain any Header cluster. However, the first one contains two: $\{H_1\}$ and $\{H_2, H_3, H_4\}$. These two clusters are separated by the region D_1 that stands in between them. While H_4 can be clustered with H_2 and H_3 since row 12 has both Data and Header regions.

In the following sections, we denote a Header cluster as $\mathcal{H}$. Typically, we are interested in the cluster having the smallest row index, among all the clusters in a given partition. We denote this special cluster as $\mathcal{H}^{top}$. Returning to the example, from the previous paragraph, we note that $\mathcal{H}^{top} = \{H_1\}$. Intuitively, the top cluster is seen as the "true" Header of the partition. The remaining clusters are either misclassifications or Header regions from other tables.

6.4.3 Quality Metrics

The quality of a candidate partitioning P is directly related to the partitions that compose it. Therefore, we define metrics that capture how close these partitions are at being tables.

However, we formulate these metrics such that they measure negative properties since we later use minimization to identify the optimal solution. This means, the lower are the metrics' values, the more table-like are the partitions.

We introduce the following functions, that are used in the definitions of the proposed metrics. To get the Data and Header vertices from a partition $V_i \in P$, we use respectively the function $data$ and $heads$. While the function $hgps$ returns the Header clusters in V_i. Moreover, we introduce functions that apply to the vertices of the graph. We use $area$ to get the number of cells for a vertex (region). The function $rows$ and $cols$ return respectively the set of row indices and column indices that the region covers. Finally, we use $cwidth$ to get the width of a column given its index, and $rheight$ to get the height of a row given its index.

In total, we define ten metrics. The first nine apply to the individual partitions that compose P. While the tenth metric considers the whole candidate partitioning.

M1 *Negative Header Alignment Ratio (nhar):* For a partition V_i, we identify the top Header cluster ($\mathcal{H}^{top}$), i.e., the one having the smallest row index among all clusters $\mathcal{H}$ in V_i. Subsequently, we calculate the ratio of columns that $\mathcal{H}^{top}$ shares with the Data regions in V_i. We invert this measurement, to capture the negative cases. Thus, the closer the value of this metric is to 1, the lower is the alignment ratio.

$$C^{ht} = \bigcup_{v \in \mathcal{H}^{top}} \texttt{cols}(v), \quad C^{d} = \bigcup_{u \in \texttt{data}(V_i)} \texttt{cols}(u),$$

$$nhar = \begin{cases} 1 - \frac{|C^{ht} \cap C^{d}|}{|C^{ht}|}, & if\ |C^{d}| \geq 1\ and\ |C^{ht}| \geq 1 \\ 0, & otherwise \end{cases}$$

M2. *Negative Data Alignment Ratio (ndar):* This metric measures the negative alignment from the perspective of the Data regions. Thus, in the fraction below, we divide by C^d.

$$ndar = \begin{cases} 1 - \frac{|C^{ht} \cap C^{d}|}{|C^{d}|}, & if\ |C^{d}| \geq 1\ and\ |C^{ht}| \geq 1 \\ 0, & otherwise \end{cases}$$

M3-4 *Is Data/Header Partition (dp/hp):* Notice that, for the previous two metrics, we omit the cases where the partitions contain either Data or Header regions, but not both. For these cases, we introduce two separate Boolean metrics. Here, we illustrate the calculation of *dp*. We handle *hp*, similarly.

$$dp = \begin{cases} 1 & if\ \texttt{heads}(V_i) = \emptyset\ and\ |\texttt{data}(V_i)| \geq 1 \\ 0, & otherwise \end{cases}$$

M5. *Is All In One Column (ioc):* This is another Boolean metric. If the $\mathcal{H}^{top}$ and the Data regions cover altogether only one column, ioc returns 1, otherwise 0. Intuitively, this metric pushes towards tables that span at least two columns.

$$ioc = \begin{cases} 1, & if\ |C^{d}| = 1\ and\ |C^{ht}| = 1\ and\ |C^{d} \cap C^{ht}| = 1 \\ 0, & otherwise \end{cases}$$

M6. *Count Other Valid Headers (#ovh):* Besides $\mathcal{H}^{top}$, there might be other *valid* Header clusters, in a partition. We consider valid those clusters that (cumulatively) span more than one column. The presence of other valid $\mathcal{H}$ suggests multiple tables in the same partition.

$$\#ovh = |\{\mathcal{H} \in \texttt{hgps}(\texttt{V}_\texttt{i}) \setminus \mathcal{H}^{top} : \; |\bigcup_{v \in \mathcal{H}} \texttt{cols}(v)| \geq 2\}|$$

M7. *Data Above Header Ratio (dahr)*: Measures the portion of Data cells found above the $\mathcal{H}^{top}$, in a given partition. In other terms, we identify Data cells with a row index less than $\min \bigcup_{v \in \mathcal{H}^{top}} \texttt{rows}(v)$. Intuitively, for typical tables, it is expected that all Data cells are below the top Header cluster. However, for arbitrary partitions, especially in multi-table sheets, Data cells could be found above it. This can also occur due to misclassifications.

$$dahr = \frac{\#dcells\_above}{\sum_{u \in \texttt{data}(V_i)} \texttt{area}(u)}$$

M8. *Average Width for Adjacent Empty Columns (avgw_aec):* For each partition, we group adjacent empty columns[2], and measure the commutative width[3] per group. Subsequently, we calculate the average of these widths. In the equation below, we denote as $\mathcal{C}^{emt}$ the list containing these *aec* groups. Intuitively, this metric identifies empty columns that might act as separators of content. This would imply that some vertices of the partition do not belong with others.

$$avgw\_aec = \frac{\sum_{i=1}^{|\mathcal{C}^{emt}|} \sum_{j=1}^{|\mathcal{C}_i^{emt}|} \texttt{cwidth}(\mathcal{C}_{ij}^{emt})}{|\mathcal{C}^{emt}|}$$

M9. *Average Height for Adjacent Empty Rows (avgw_aer):* The presence of empty rows, similar to empty columns, could imply separation of contents.

$$avgw\_aer = \frac{\sum_{i=1}^{|\mathcal{R}^{emt}|} \sum_{j=1}^{|\mathcal{R}_i^{emt}|} \texttt{rheight}(\mathcal{R}_{ij}^{emt})}{|\mathcal{R}^{emt}|}$$

M10. *Overlap Ratio (ovr):* Unlike the other metrics, this one is calculated at partitioning level, P. We identify overlaps between the individual partitions composing P and measure their area. We divide the sum of overlaps with the used area of the sheet (i.e. the minimum bounding box enclosing all regions). Below, C_i and C_j represent the sets of column indices for partition V_i and V_j, respectively. Equivalently, for row indices, we use R_i and R_j.

$$ovr = \frac{\sum_{i=1}^{|P|-1} \sum_{j=i+1}^{|P|} |C_i \cap C_j| * |R_i \cap R_j|}{|\bigcup_{v \in V} cols(v)| * |\bigcup_{v \in V} rows(v)|}$$

6.4.4 Objective Function

In Equation 6.1, we define the function measuring the fitness of the individual partitions that comprise a candidate partitioning $P = \{V_1, ..., V_k\}$.

$$\texttt{fit}(V_i, M, \mathbf{w}) := \sum_{j=1}^{9} M_j(V_i) * \mathbf{w}_j \tag{6.1}$$

[2]Columns can be empty within the partition (i.e., an area of the sheet), not necessarily for the whole sheet.

[3]In Excel, the column width is measured in units of 1/256th of a standard font character width, while the row height is measured in points.

As shown, the fitness for a partition V_i is calculated as a weighted sum of metrics' values. With M we denote the list of implemented metrics' functions. In Equation 6.1, we use only the first nine metrics, since these apply at the partition level (as mentioned in Section 6.4.3). While $\mathbf{w}$ is a vector that holds the corresponding weights for the metrics.

$$\texttt{obj}(P, M, \mathbf{w}) := \mathbf{w}_{10} * M_{10}(P) + \sum_{V_i \in P} \texttt{fit}(V_i, M, \mathbf{w}) \tag{6.2}$$

Equation 6.2 provides the definition for the objective function. We sum up the fitness of the individual partitions. Moreover, we make use of the *Overlap Ratio* metric (M_{10}), which is calculated at the level of the partitioning.

6.4.5 Weight Tuning

One of the challenges of the proposed approach is determining the optimal weights for the objective function (see Equation 6.2). Intuitively, some metrics are more crucial than others. However, it is rather difficult to manually ascertain the exact importance of each metric in relation to the rest. Therefore, optimization algorithms are needed to tune the weights automatically. Here, we make use of Sequential Quadratic Programming (SQP) [23], for constrained minimization.

Tuning Sample. We tune the weights based on a sample drawn from a dataset of annotated sheets. Note, the size this sample are discussed in more detail in Section 6.5.3. We construct the graph representation for each sheet in the sample. Subsequently, the intention is to guide the optimization algorithm in finding weights that favor valid partitions, while penalizing false ones. Therefore, for each one of the graphs, we consider the target partitioning, i.e., the one that corresponds to the true tables in the sample sheet. Note, this partitioning is given since we operate on annotated sheets. Furthermore, we randomly generate multiple alternative partitionings (i.e., false instances).

Tuning Function. We denote the sample used for optimization as $\mathcal{S} = (\mathcal{U}, \mathcal{T})$, where $\mathcal{U}$ holds all the generated alternative partitionings, and $\mathcal{T}$ holds the corresponding target partitionings. As stated previously, for each graph we generate multiple alternative partitionings. Thus in order to ensure a one-to-one mapping, we replicate the target partitionings in $\mathcal{T}$.

$$\arg\min_{\mathbf{w}} \sum_{j=1}^{|\mathcal{U}|} \frac{1 + obj(\mathcal{T}_j, M, \mathbf{w})}{1 + obj(\mathcal{U}_j, M, \mathbf{w})}, \text{such that: } 0 \leq \mathbf{w} \leq 10^3 \tag{6.3}$$

The equation above defines the function used for weight tuning. As can be seen, it is a summation of fractions, designed for minimization. In most of the fractions, the numerators will get smaller values than the denominators (as mentioned in Section 6.4.3, we measure negative properties). Regardless, we still need to increase the gap between targets and alternatives. The challenge is to find weights that penalize alternatives, without affecting many targets.

Note that we add $+1$ to denominators and numerators, in order to avoid exceptions of division by zero, during the automatic search for optimal weights. Furthermore, we constrain the possible values for the weights in the interval $[0, 10^3]$. This is to avoid extremely large or extremely low weights, which might not be realistic, but rather reflecting the peculiarities of the current sample.

6.4.6 Genetic Search

After tuning, we attempt to identify the fittest partitioning for the remaining sheets in the dataset. For small graphs (currently set to ≤ 10 vertices), we perform exhaustive search. However, for larger graphs, a more efficient mechanism is required. For these cases, we use genetic algorithms [14, 66], which can yield near-optimum solutions in a reasonable time. In this work, we make use of the *edge encoding* as described in [81]. We represent the edges of an input graph with a Boolean valued list of size $|E|$. When the corresponding value is set to true, the edge is activated, otherwise not. Dis-activating and re-activating edges lead to various graph partitionings. Intuitively, *edge encoding* ensures that we always get connected components of the input graph. This is favorable since it translates to partitions that enclose neighboring regions, rather than arbitrary ones.

Nevertheless, there are challenges to this formulation. The search space increases exponentially with the number of edges $2^{|E|}$. Thus, identifying the right combination of edges becomes more demanding for larger graphs. We find such graphs (up to $7,484$ edges) in our dataset. They occur due to many implicit/missing values (empty cells) in tables, which inflate the number of vertices (i.e., layout regions), and with that the number of edges.

The pseudocode in Algorithm 6.3 describes the implemented genetic search. We have adopted the $eaMuPlusLambda$ algorithm, as provided by the DEAP library [1, 54]. An initial population, composed of Boolean-valued lists, is generated randomly. The size of this population is not fixed but rather calculated with a function, which takes into account the number of edges in the input graph. In subsequent iterations, new individuals (*Children*) are created from the population of the previous generation (*Parents*). This step is performed using one of the following genetic operations: *random mutation* (inverts Boolean values of a Parent, with an independent probability *indpb* = 0.1), and *uniform crossover* (combines values from two Parents, with *indpb* = 0.5). We pick individuals for the next generation with *tournament selection* of size = 3, considering both Children and Parents. The *hallOfFame* (hof) carries the individual having the smallest score (i.e., the fittest) among all examined candidates.

Algorithm 6.3: Identifying the Optimum Partitioning using Genetic Search

Input: sheet-graph: $G = (V, E)$, metrics: M, weights: $\mathbf{w}$, population size: $npop = \texttt{ceil}(log_{10}(|E|) * 100)$, #generations: $ngen = 200$, crossover probability: $cxpb = 0.5$, mutation probability: $mutpb = 0.5$, #offsprings to generate: $\lambda = npop$, #individuals to select: $\mu = npop$, and $seed$ individual (optional)
Output: The partitioning with the lowest objective function score

```
1 begin
2     Pop ← createInitialPopulation(G, npop, seed);
3     hof ← updateHallOfFame(Pop, G, M, w);
4     for i ∈ {1, ..., ngen} do
5         Children ← createOffsprings(Pop, λ, cxpb, mtpb);
6         hof ← updateHallOfFame(Children, G, M, w);
7         Pop ← selectFittest(Pop ∪ Children, μ, G, M, w)
8     return hof
```

We select genetic operators and parameter values following recommended practices [48]. After extensive experimentation, we favored those that push towards more diverse generations (Children). In this way, we cover a larger search space and decrease the chances of premature convergence.

Furthermore, note that there is an option for a *seed* individual, in Algorithm 6.3. We adopt the rule-based approach, from Section 6.3, to create this seed. It represents a good candidate solution, which is fed to the initial population. Such hybrid approaches are common in practice, as suggested by [68]. They often yield better results than pure genetic ones. We make $\frac{npop}{2} - 1$ copies of the seed, and apply random mutations on them, with *indpb*=0.1. These copies, together with the seed, compose half of the initial population. The rest is randomly generated.

6.5 EXPERIMENTAL EVALUATION

For the rule-based approach, from Section 6.3, the evaluation is straightforward. We run the algorithm once for each classified sheet in the considered dataset. However, the genetic-based approach, from Section 6.4 is not deterministic. The weights for the objective function will differ, depending on the selected training sample. Moreover, the genetic search itself is a meta-heuristic method that does not guarantee the same output, given the same input.

Therefore, for the second approach, we perform 10-fold cross-validation. Note, we ensure that single- and multi-table graphs are balanced among the folds. Each iteration, 9 folds are reserved for training (i.e., weight tuning), and 1 for testing (i.e., table detection). Moreover, the tuning process undergoes several rounds of its own, which are discussed in more detail in Section 6.5.3. Furthermore, after tuning, we execute the genetic search 10 times for test graphs having $|E| > 10$. For the rest, $|E| \leq 10$, we perform exhaustive search. Finally, to ensure statistical significance, we repeat the whole process three times. Each time, before cross-validation, we shuffle the dataset using a different numeric seed.

6.5.1 Testing Datasets

Here, we consider the datasets extracted from Enron and Fuse corpus (refer to Chapter 3). After omitting few files as being out of scope (see Section 3.4.1), the considered datasets contain respectively 814 and 274 sheets. In the Enron dataset, there are 674 single-table sheets and 140 multi-table sheets. For the Fuse dataset, the respective numbers are 222 and 52. Overall, the total number of annotated tables is $1,158$ (Enron) and 458 (Fuse).

Before the experimental evaluation, as described in Section 6.2, we construct the graph representation for each considered sheet. We use as input the classification results from Chapter 4. However, we also consider the original annotations from Chapter 3. Essentially, we perform experiments with the ground-truth graphs and with those carrying misclassifications. In this way, we can compare the evaluation results and infer the impact that misclassifications have on the proposed approaches.

Note, with regard to the cell classifications, we consider from Chapter 4 both scenarios: 6 and 7 labels[4]. In Section 4.3, we mentioned that three distinct cross-validation runs were used to evaluate the classification methods. For the experiments in this chapter, we use the results from one of these runs. We omit the worst and best run, with respect to the *Macro_F1* score. Thus we keep the middle case, the one being closer to the average score.

6.5.2 Training Datasets

The datasets from the previous section are used only for testing. In fact, we generate special datasets for training (i.e., weight tuning). These datasets contain random induced errors. We explain in the following paragraphs how cross-validation works with two separate datasets.

Inducing Random Error: We consider the original cell annotations, i.e., the ground truth. Subsequently, we reduce the cell labels to three meta-classes, as outlined in Section 6.2.3.

[4]In both cases, the pre-processing actions (Section 6.2.3) will reduce the labels to 3 meta-classes.

Thus, cells are now re-organized into Data (D), Header (H), and Metadata (M). Before creating the layout regions, we induce random errors. Basically, we re-assign a small percentage of randomly selected cells to another (false) class.

Specifically, to generate the training datasets, we consider three parameters per class. The first one is the probability of randomly selecting the cells of this class for induced error. Secondly, once a cell is selected, we re-assign it to one of the remaining two classes. Therefore, we need the probability of picking each one of these two classes.

We experimentally determine these 9 parameters (i.e., 3 parameters for each meta-class). Concretely, we generate a training dataset for each one of the considered test datasets. Therefore, in total, there are six training datasets, with induced error as reported in Table 6.1. We report the probabilities as percentages in the following format: % error for this class; (% re-assign to other class1, % re-assign to other class2). The order for the re-assignment probabilities is as follows: D; (M,H), H; (D,M), M; (D,H).

Intuitively the amount of induced error depends on the current testing dataset. In other terms, the corresponding training dataset follows a roughly similar error (misclassification) distribution to that of the testing dataset. We introduce more error when testing on the classification results, compared to the ground-truth dataset. Moreover, our experiments show that the Enron dataset requires more induced-error than the Fuse dataset.

Table 6.1: Induced Noise per Training Dataset

	FUSE			**ENRON**		
	Classification		Ground Truth	Classification		Ground Truth
	6 labels	7 labels		6 labels	7 labels	
D	$0.1; (80, 20)$	$0.5; (80, 20)$	$0.1; (80, 20)$	$0.1; (91, 9)$	$0.5; (91, 9)$	$0.1; (91, 9)$
H	$2.5; (87, 13)$	$2.5; (90, 10)$	$1.0; (90, 10)$	$3.5; (86, 14)$	$3.5; (86, 14)$	$1.0; (86, 14)$
M	$5.0; (98, 2)$	$5.0; (98, 2)$	$2.0; (98, 2)$	$7.0; (96, 4)$	$7.0; (96, 4)$	$2.0; (96, 4)$

Cross-validation with Two Datasets: The training and testing datasets contain the same sheets, but differ when it comes to the meta-classes assigned to the cells. For cross-validation, these two datasets are divided into 10 folds. However, for both of them, we use the same random seed to allocate the sheets into folds. Thus, there is a one to one correspondence. From then on, we follow the standard cross-validation procedure. Each iteration of the cross-validation, the 9 training folds come from the induced-error dataset. The left-out fold comes from the current test dataset.

Our experimental evaluation showed that training with this method improves the detection accuracy. Intuitively, the induced errors expose the tuning mechanism to small irregularities, which help avoid overfitting. This is true not only when testing on the classification results, but also for the original annotations.

6.5.3 Tuning Rounds

Every iteration of the cross-validation, 9 folds (with induced error) are used to tune the weights for the objective function. The target partitioning is determined from the annotations (refer to Chapter 3). In addition to this, we generate multiple alternative (false) partitionings. Then, we use the function from Section 6.4.5 to find optimal weights.

In fact, we limit the number of alternative partitionings to $10 * \#tables$, per graph. This balances the contribution of multi-table and single-table graphs in the tuning sample. Nevertheless, a limit is anyway necessary, since we cannot exhaustively generate all alternative partitioning for each training graph. This would be computationally very expensive since we find large graphs in the dataset, with > 1000 edges and/or vertices.

Instead, we ensure reliable weights by performing 10 rounds of tuning, each time randomly generating new alternative partitionings. The weights, resulting from the different rounds, are averaged. For this, we consider the error rate of each round. This is measured as the portion of alternative partitionings having an obj function value that is lower than that of the targets. Altogether, rounds with higher error rates contribute less to the averaged weights.

6.5.4 Search and Assessment

The rule-based approach and the exhaustive search for tables are deterministic. However, the genetic-search can return different candidate solutions, on multiple runs, for the same input graph. Therefore, we perform the genetic search 10 times and then average the accuracy of the results.

The accuracy itself is assessed by comparing the target partitioning (i.e., ground truth) to the output. Note that we consider from the output only partitions having both Data and Header vertices. The rest we regard as non-valid candidates. Let $T = \{V_1, ..., V_n\}$ be the target partitioning, and $P = \{V_1, ..., V_k\}$ be the predicted partitioning. As shown in Equation 6.4, for a table partition (V_i^t) and a (valid) predicted partition (V_j^p), we calculate the agreement as #cells in common over #cells in union. Basically, we formulate the agreement the same as the Jaccard index [91, 123].

$$\frac{|cells(V_i^t) \ \cap \ cells(V_j^p)|}{|cells(V_i^t) \ \cup \ cells(V_j^p)|} \tag{6.4}$$

Note, for this metric, we consider only non-empty and non-hidden cells. This means, we use the cells from the layout regions, in the table and predicted partition. All in all, a table is marked as detected, when we find an agreement of ≥ 0.9.

In addition to the above metric, we use EoB (Error of Boundary), a metric proposed by Dong et al. [43] for the TableSense framework. EoB is defined below, in Equation 6.5. The bounds for the predicted table are denoted as B. While, B' denotes the ground truth, i.e., the true table bounds. Intuitively, when $EoB == 0$ the predicted and true bounds match exactly. Nevertheless, in [43], the authors consider the table as detected when $EoB \leq 2$. As stated in the paper, the tolerance of 2 is chosen by accounting for the existence of title, footnotes, and side-notes.

$$\begin{aligned} EoB = max(\ & abs(row_{top}^{B} - row_{top}^{B'}), abs(row_{bottom}^{B} - row_{bottom}^{B'}), \\ & abs(col_{top}^{B} - col_{top}^{B'}), abs(col_{bottom}^{B} - col_{bottom}^{B'})) \end{aligned} \tag{6.5}$$

6.5.5 Evaluation Results

Tables 6.3, 6.2, and 6.4 summarize the evaluation results. Specifically, we report the percentage of detected tables, in the testing datasets. The first two tables list results for experiments with classified cells, respectively for 6 labels and 7 labels. While Table 6.4 provides the results for experiments on the ground truth dataset (i.e. the annotated cells).

Clearly, the above-mentioned scenarios, 6 labels, 7 labels, and ground truth, are addressed separately for the Fuse and Enron dataset. Thus, in the end, we tested six datasets. For each scenario, we provide the results cumulatively for all the sheets. In addition, we report results separately for single- and multi-table sheets.

We evaluate the proposed rule-based (RULES) and genetic-based (GE) approach on these six datasets. We run the experiments only once for the first approach. While, for the second one, we perform three cross-validation runs and then average the results. Note for the genetic approach (GE) we use noise (N) for training, as outlined in Section 6.5.2. In addition, in all experiments, we seed (S) the initial population with the output from the rule-based approach. This option was discussed in Section 6.4.6.

We measure the table detection performance using the metrics defined in Section 6.5.4. For brevity, we denote Jaccard index ≥ 0.9 simply as Jc-0.9. For the EoB metric we consider two cases: $EoB \leq 2$ and $EoB == 0$. The first one is denoted as EoB-2 and the second one as EoB-0.

Analysis of Evaluation Results

In this section, we discuss the evaluation results. We make several observations, outlined below. First, we notice that Misclassifications have a notable impact on both proposed approaches. The performance increases as we move from classification with 7 labels (having the most misclassifications), to 6 labels, and finally to the ground truth (having no misclassifications).

Moreover, the results are consistently better for the Fuse dataset, compared to the Enron dataset. We noticed the same pattern also while studying the classification results, in Section 4.3. As emphasized in Section 3.3.2, sheets in the Fuse dataset are far more *regular* than those in the Enron dataset. This plays a significant role when it comes to classification and detection.

Another observation is that overall we get better accuracy for single-table rather than for multi-table sheets. Clearly, the latter cases are more demanding. The algorithm needs to detect two or more tables in the (sheet) graph, which potentially have different layouts, and are arranged in arbitrary ways.

Finally, with the exception of very few cases, the genetic approach outperforms the rule-based one. Nevertheless, we find a notable exception in Table 6.4. For the Fuse ground truth dataset, the rule-based approach performs slightly better. Note that using a rule-based seed for the genetic approach does not guarantee the same or better results. One has to consider also the weights from the objective function, which have more influence on the genetic search. Overall, we can say that the strengths of the genetic approach show better when we deal with irregular (atypical) sheets and misclassifications.

Table 6.2: Percentage of Detected Tables for Classification with 7 Labels

		FUSE			ENRON		
		All	Single	Multi	All	Single	Multi
RULES	*Jc-0.9*	80.2	90.4	66.9	68.7	78.2	55.6
	EoB-2	80.9	86.9	73.1	65.5	72.9	55.2
	EoB-0	65.6	79.5	47.4	50.0	60.5	35.3
GE+N+S	*Jc-0.9*	82.4 ± 0.2	90.1 ± 0.2	72.4 ± 0.3	70.6 ± 0.2	80.3 ± 0.1	57.2 ± 0.6
	EoB-2	84.7 ± 0.0	88.4 ± 0.2	79.8 ± 0.3	68.7 ± 0.2	76.4 ± 0.1	58.0 ± 0.5
	EoB-0	70.8 ± 0.0	82.7 ± 0.2	55.2 ± 0.3	52.1 ± 0.2	62.1 ± 0.2	38.2 ± 0.2

Table 6.3: Percentage of Detected Tables for Classification with 6 Labels

		FUSE			ENRON		
		All	Single	Multi	All	Single	Multi
RULES	*Jc-0.9*	86.9	93.9	77.7	75.5	82.9	65.1
	EoB-2	85.9	90.8	79.4	69.8	79.2	56.6
	EoB-0	75.3	86.5	60.6	56.5	69.9	37.8
GE+N+S	*Jc-0.9*	87.5 ± 0.3	91.4 ± 0.2	82.5 ± 0.5	80.1 ± 0.1	87.4 ± 0.0	69.4 ± 0.3
	EoB-2	87.5 ± 0.3	89.8 ± 0.2	84.4 ± 0.7	73.3 ± 0.2	83.4 ± 0.1	59.2 ± 0.5
	EoB-0	78.2 ± 0.2	87.2 ± 0.2	66.5 ± 0.3	60.7 ± 0.3	74.1 ± 0.1	41.9 ± 0.7

Table 6.4: Percentage of Detected Tables for Ground Truth Cells

		FUSE			ENRON		
		All	Single	Multi	All	Single	Multi
RULES	*Jc-0.9*	96.0	95.6	96.6	84.6	86.7	81.8
	EoB-2	95.8	95.2	96.6	83.2	84.6	81.2
	EoB-0	94.6	94.3	94.9	79.7	80.7	78.3
GE+N+S	*Jc-0.9*	95.3 ± 0.2	94.8 ± 0.4	96.0 ± 0.0	88.2 ± 0.2	90.7 ± 0.1	84.8 ± 0.5
	EoB-2	95.1 ± 0.2	94.2 ± 0.2	96.2 ± 0.3	87.2 ± 0.1	89.0 ± 0.1	84.8 ± 0.2
	EoB-0	93.7 ± 0.2	93.2 ± 0.4	94.3 ± 0.0	82.9 ± 0.3	84.4 ± 0.2	80.9 ± 0.8

Comparison with Related Work

Here, we compare with the TableSense approach, proposed by Dong et al.[43]. The authors report an overall EOB-2 score of 91.3%, and 80.8% for EOB-0. Unlike us, they do

not discuss single- and multi-table sheets separately.

The authors tested TableSense on a dataset of 400 sheets, with a total of 795 tables. While training was done on a separate dataset of $10k$ sheets. To the best of our knowledge, these datasets and the TableSense framework are not publicly available. Thus, we cannot test our approaches on their datasets or vice versa. However, the authors report that they extracted the spreadsheet files from the Web. Therefore, from this thesis, the Fuse dataset is the closest to [43].

Clearly, Dong et al. report better accuracy than the one we achieve on the Fuse dataset, for classification with 6 and 7 labels (see Tables 6.3 and 6.2). However, we achieve better accuracy to [43], when considering the results from the ground truth dataset (see Table 6.4). Thus, there is potential for the approaches proposed in this thesis. The results suggest, in the presence of few or no misclassifications, these approaches can be competitive with or even perform better than the TableSense framework.

Additionally, the processing pipeline discussed in this thesis has some advantages over the TableSense framework. Firstly, we provide the table layouts, in addition to detecting the tables. We can use these layouts to analyze and extract the data from tables (Chapter 7). However, in [43], the table analysis and data extraction tasks are left for future work. Secondly, our approach is more traceable. In [43], the authors make use of Neural Networks, which are known for their "black box" nature. It is hard to trace back their decisions, in order to debug the model. However, this is not the case with our approach. It is much easier to trace back the steps of our processing pipeline, in order to detect where the decision was made or where the potential issue had occurred.

6.6 SUMMARY AND DISCUSSIONS

In conclusion, this chapter introduced steps in our processing pipeline that relate to the table detection task. First, we encode the sheet layout with an intuitive graph model (Section 6.2.1). This step also includes few pre-processing actions (Section 6.2.3), that consolidate the graph representation and additionally simplify the detection task. The latter is handled by the next step. We have formulated the table detection as a graph partitioning problem (Section 6.2.2). Here, we propose two novel approaches: a rule-based (Section 6.3) and a genetic-based (Section 6.4). The output of these approaches is subgraphs (i.e., connected components) of the input graph that correspond to true tables in the sheet.

In Section 6.5, of this chapter, we evaluated the two proposed approaches using a dataset of Web spreadsheets (Fuse) and a dataset of business spreadsheets (Enron). The results show that both approaches achieve fairly good performance, even in the presence of misclassifications. Nevertheless, the accuracy is typically higher for the genetic-based approach. Furthermore, when we experiment with the original annotations (i.e., the ground truth datasets), we observe significantly high performance. In fact, we achieve 96.0% accuracy for Web spreadsheets. While, for business spreadsheets, which exhibit much more irregularities (see Section 3.3.2), we achieve ca. 88% accuracy.

Thus, we can say that these approaches are viable, and can be extended even further by future research projects. Clearly, there is a need to address the cell classification accuracy, since this can lead to better detection results. Moreover, we foresee the addition of more rules and metrics for our table detection approaches. For instance, one can consider the formula references in the sheet. Basically, intra-table references are far more common that inter-table references (see Section 3.3). Thus, when there is a reference from one layout region to another, it most probably means that they are part of the same table.

7

XLINDY: A RESEARCH PROTOTYPE

7.1 Interface and Functionalities

7.2 Implementation Details

7.3 Information Extraction

7.4 Summary and Discussions

This chapter is based on our DocEng'19 publication [82]. We present XLIndy (Excel **Indy**), a Microsoft Excel add-in with a machine learning back-end, written in Python. XLIndy is not an end-user product, but rather a framework for testing and interacting with our novel layout analysis and table detection approaches. For a selected task and method, users can visually inspect the results, change configurations, and compare different runs. Additionally, users can manually revise the predicted layout and tables. Moreover, they can save these revisions as annotations. Finally, data in the detected tables can be extracted for further processing. Currently, XLIndy supports exports in CSV format.

We discuss the above-mentioned features in the following sections of this chapter. We introduce the user interface, in Section 7.1. Then, in Section 7.2, we provide implementation details. In Section 7.3, we outline our procedure for information extraction. We conclude this chapter, in Section 7.4.

7.1 INTERFACE AND FUNCTIONALITIES

Figure 7.1 provides a brief look at the user interface. It captures the state of the tool after the layout analysis, and right before table detection/recognition[1]. Using build-in shape objects provided by Excel, XLIndy displays layout regions as colored-coded rectangles, overlaying non-empty cells of the sheet. In the illustrated example, there is a misclassification, which the user repairs via the context menu.

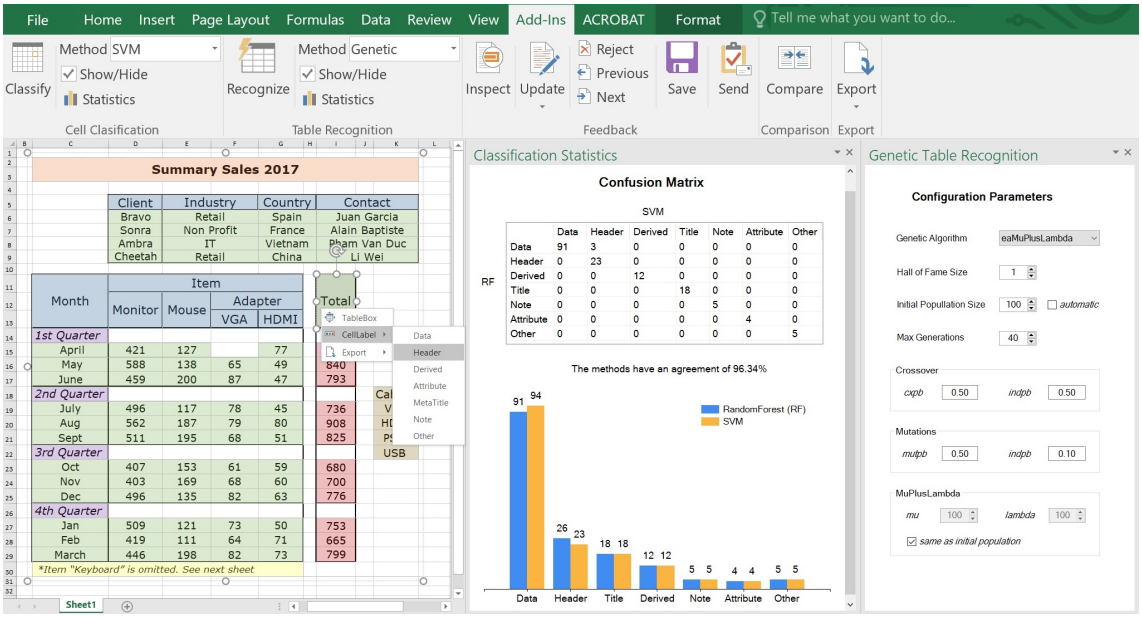


Figure 7.1: UI of the XLIndy tool

7.1.1 Front-end Walkthrough

In the upper part of Figure 7.1, we can see a custom ribbon, which acts as the primary menu for XLIndy. From here, the user carries the majority of the supported functionalities. Starting from the left, "Cell Classification" and "Table Recognition" sections allow

[1]In this chapter, the term recognition is used more often than detection. Nevertheless, the former incorporates the latter. We first detect the table and then analyze its layout.

users to respectively initiate layout analysis and recognition tasks. The user selects one of the several supported methods from the dropdown list, and depending on the desired task he/she clicks "Classify" or "Recognize". Once the results are displayed, the user can click the "Statistics" button, which opens a custom pane providing an overview of the task performance (see Figure 7.1). Next, the "Feedback" section helps the user navigate the results and revise them if needed. Moreover, from the ribbon, the user can compare different runs, via the "Compare" button. This will point to the differences, and emphasize the strengths and weaknesses of each run. Finally, the "Export" button dumps data from the selected table areas, in one of the supported formats (currently CSV).

Note, from the "Feedback" section the user can additionally save as annotations the classification and recognition results, together with his/her revisions. These annotations are stored in a special hidden sheet of the Excel document, allowing future reuse. For instance, the annotations can be used to (re-)train classifiers, which ultimately can lead to improved performance for the proposed approaches. Nevertheless, XLIndy does not replace the original annotation tool (written in Java), from Chapter 3. However, it does incorporate several of this tool's functionalities.

Other panes allow the user to change configurations, before executing a task. In this way, users can fine-tune the selected approach, ultimately achieving the desired accuracy. Figure 7.1 displays the configuration pane for the genetic-based table detection (discussed in Section 6.4). This pane opened automatically when the user selected the "Genetic" method from the drop-down list in the ribbon.

Furthermore, XLIndy supports a series of actions via a custom context menu. This makes certain operations straight forward. For example, in Figure 7.1 the user changes the role of a layout region from *Data* to *Header*. Moreover, from this context menu, the user can manually create a table box (i.e., a rectangle shape with no fill), to indicate a tabular area. In this way, he/she can correct the table detection results. Subsequently, the user can export data from tables, using the available options from this menu.

7.2 IMPLEMENTATION DETAILS

As shown in Figure 7.2, XLIndy has front-end and back-end components. It carries out the whole processing pipeline, described in the previous chapters.

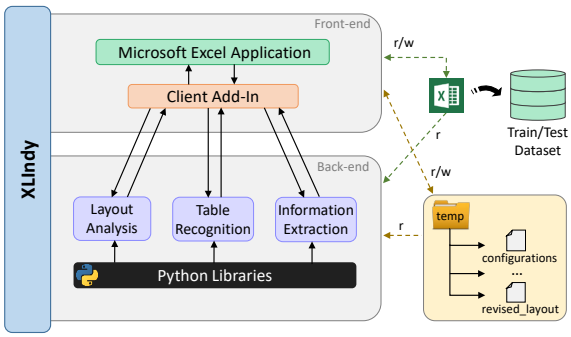


Figure 7.2: Architecture of XLIndy

The users interact with the tool via a familiar interface, i.e., the Excel desktop application. On top of that, we deploy a custom add-in, which was developed in C#. This add-in

triggers the execution of tasks from the pipeline, and subsequently handles the results. Nevertheless, the tasks themselves are performed by Python scripts, which stay at the back-end of the system. In Section 7.2.1, we outline how the add-in communicates with these scripts.

The back-end does not only hold the implementation of the proposed approaches but also some utilities. We use the many available and highly efficient Python libraries. In particular, we take advantage of the rich machine learning ecosystem [3, 101].

Another aspect of the XLIndy system is the physical layer, which holds the spreadsheet, configuration, and temporary run-time files. In Section 7.2.2 we discuss how to efficiently process the spreadsheets files. While in Section 7.2.1 we motivate the need for temporary run-time files.

In the physical layer, resides also the gold standard dataset (see Chapter 3), which is not a direct part of the system, but still tightly related to it. After all, on this dataset, we train/test the proposed layout inference and table recognition methods. Furthermore, with the help of XLIndy, we can expand the gold standard with new annotated sheets. The users can provide feedback and save annotation through the UI (Section 7.1.1).

7.2.1 Interoperability

At the moment, the .NET Framework has some support [2] for Python code. However, it lacks many features and libraries that come with other more popular Python implementations. Therefore, a workaround is to run Python scripts within processes initiated from C# code. Subsequently, the standard output of these processes is parsed and then displayed to the user.

On the other side, the front-end has to send data to the Python scripts, as well. However, operating systems usually impose limitations on the number and memory-size of the arguments passed to processes. To overcome these limitations, for some operations, we use temporary files to store data coming from the front-end. Then, we instruct the Python scripts to read these files.

7.2.2 Efficient Reads

The active Excel document is processed by both ends of the system. The front-end handles various operations which require read and/or write permissions. However, read permissions are sufficient for the back-end, since it only needs to collect features from the active sheet. To ensure consistency, if there are updates, the front-end will save the Excel file before making a call to the local service.

Occasionally, we get large documents. Therefore, the back-end employs an efficient reading mechanism. The .xlsx format is in reality a zipped directory [77] of XML files, which carry information regarding the cell values, formatting, references, and many others (as detailed in the OOXML standard). Therefore, using the *openpyxl* library [64], we read only XML files needed for the current task.

7.3 INFORMATION EXTRACTION

XLIndy supports information extraction from the recognized (detected) tables. Before outlining the extraction procedure, we explain what kind of input this procedure expects. In Chapter 6, we discuss two pre-processing actions that reduce the layout labels prior to the table detection step. Note, these actions do not affect the final layout of the table, i.e., the one used for information extraction. In concrete terms, once the detection is complete, we return back to the original cell classifications. This is illustrated in Figure 7.3.

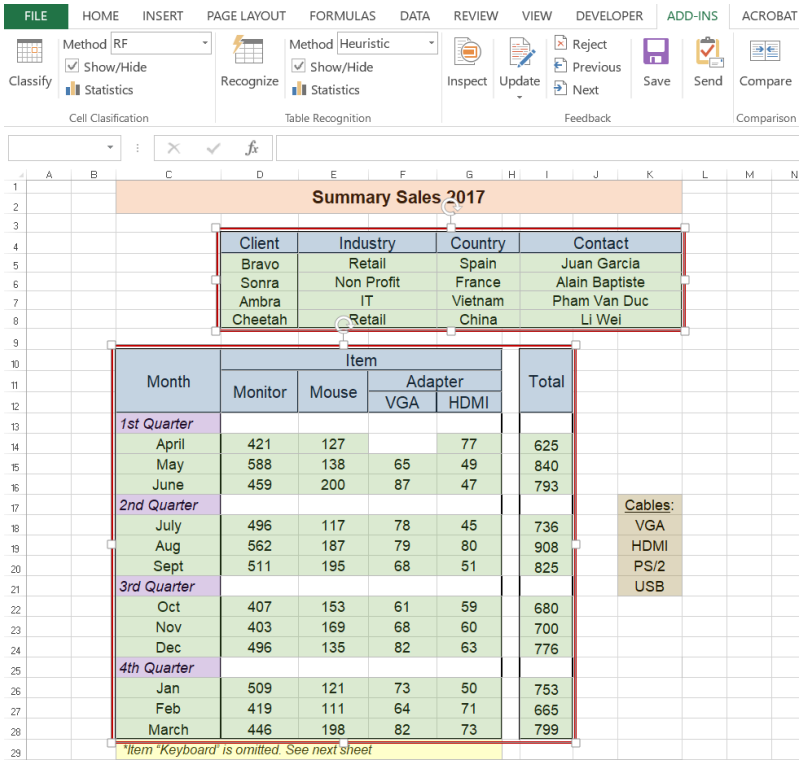


Figure 7.3: Exporting Data from Detected Tables

From now on, the procedure is straightforward. As already suggested by related work [11, 29, 119], we can use the inferred layout functions to derive the structure of the data in the table. In this work, we proceed in a similar fashion. Initially, we check for hierarchies on the top rows and then on the left columns of the table. For instance, in Figure 7.3, the Header cells for the bottom table are stacked, spanning multiple rows. Clearly, the Headers in the lower rows depend on those above. Furthermore, in the left column, months are grouped by quarter (i.e., *GroupHeaders*). Thus, we traverse the top hierarchies, then move to the left hierarchies, and finally to the actual value of the Data cell. An example tuple would be: *['Monitor', '1st Quarter', 'April', 421]*.

However, as noted in the previous chapters, cell classification is not without errors. This means the inferred layout for the detected tables might not be precise. Therefore, to improve the quality of the extracted information, we need a mechanism that disregards obvious misclassifications. Below, we outline one such mechanism, based on simple and intuitive rules. Note, we re-use concepts introduced in previous chapters.

Intuitively, when it comes to the layout labels, we expect the rows inside the detected tables to be homogeneous. However, this might not be the case due to misclassifications. Thus, we proceed to analyze each row and detect the majority label, i.e., the one assigned

to most of the cells in this row. Then, we simply apply this label to the remaining (minority) cells. Effectively, we enforce homogeneity for the table rows.

However, we need to consider two special cases. The *GroupHeaders* (i.e., left side hierarchies) typically occupy only one cell in a row. Although not shown in Figure 7.3, there are cases where we find *Data* cells on the right of GroupHeaders. In such cases, we might lose actual *GroupHeaders*, by naïvely applying the majority rule (discussed in the previous paragraph). Therefore we exclude *GroupHeaders* from this rule. Instead, we do not modify the cells having this label, as long as they are in the first two columns of the table. Last, we need to address cells classified as *Headers*. Especially, those that are misclassified. Here we make use of Header clusters, a concept introduced in Section 6.4.2. We isolate the *top* Header cluster in the table. Cells in this cluster are considered true Headers, and the rest is treated as misclassified. Concretely, this means, when determining the majority label per row, we exclude Headers marked as misclassified. If the row does not carry any other label, besides (misclassified) Headers, we simply mark the enclosed cells as Data.

7.4 SUMMARY AND DISCUSSIONS

In conclusion, in this chapter, we discuss our research prototype. XLIndy implements the proposed processing pipeline for the automatic understanding of spreadsheet data. The front-end of this tool is an Excel add-in. Thus, the user interacts with a familiar interface. From there, the user can execute the steps of our pipeline. These are handled by back-end Python scripts.

One of the main objectives of XLIndy is to visualize the results of the proposed approaches. In this way, the user can inspect the results and provide his/her feedback. Based on this feedback, we can derive challenges and open issues. Ultimately, this means, we can make the proposed approaches even more effective.

CONCLUSION

8.1 Summary of Contributions

8.2 Directions of Future Work

8.1 SUMMARY OF CONTRIBUTIONS

In this thesis, we focus entirely on a ubiquitous document type, i.e., spreadsheets. Because of their user-friendly interface and manifold functionalities, they have been extensively used in business settings and even found on the Web. Altogether, these documents represent a large collection of valuable data. However, spreadsheets are not optimized for automatic machine processing, which limits integration with other sources and systems. Specifically, the structure of the data is not explicitly given by the spreadsheet document or application. The enclosed tables are not trivial, often exhibiting irregularities (e.g., empty rows/columns and missing values) and accompanied by implicit information (e.g., formatting and textual metadata). Therefore, complex algorithmic steps are required to unlock the structure and wealth of spreadsheet data.

In order to better understand this research problem, In Chapter 2 we review the technical characteristics of spreadsheet documents, as well as related work in literature. We begin with the evolution of spreadsheet documents. Besides the user interface and functionalities, we focus on the file format, i.e., the way spreadsheets encode (store) information. In addition, we review existing research work on these documents. Our survey revealed that despite the relevance, the task of automatic understanding in spreadsheets has been scarcely investigated. Therefore, we turned our attention to the historic field of Document Analysis and Recognition (DAR). From there we borrow well-defined concepts and approaches, with the aim to adapt them for layout analysis and table detection in spreadsheets.

We continue the study of spreadsheet documents, in Chapter 3. Despite the existing works, there is no publicly available dataset of annotated spreadsheets. For this reason, in this thesis, we developed our own annotation tool, as described in Section 3.2.2. Three judges were instructed to annotate the layout function of cells, as well as mark the borders of tables. This work resulted in two annotated datasets, which we are the first (among related work) to make publicly available. Differently, from related work, we consider business spreadsheets (extracted from Enron corpus [72]). Nevertheless, we also target Web spreadsheets (extracted from the Fuse corpus [16]), which were to some extent covered by related work. As outlined in Section 3.3, from a thorough analysis of these two datasets, we identify open challenges and derive the requirements for this thesis. Based on these findings, in Section 3.4.2, we propose a spreadsheet processing pipeline for layout analysis and table detection. The proposed solution operates in a bottom-up fashion, following best practices and recommendations from the DAR field.

In Chapter 4, we discuss steps from our processing pipeline that relate to the task of layout analysis. We propose an approach that infers the sheet layout at the cell level. Therefore, unlike related work, at this initial phase, we do not adhere to any predefined orientation or arrangement of contents. Moreover, we consider seven descriptive layout labels (defined in Section 3.2.2), which are assigned to the cells via classification (supervised machine learning). We propose and implement highly relevant cell features that were not considered before by related work. Our thorough experimentation with cell classification shows that our approaches are viable. We achieve higher accuracy inside the tables, compared to related work. The chapter concludes with a formalized procedure that groups cells into uniform regions of interest (i.e., referred to as layout regions). Subsequently, in Chapter 6, we use these regions for table detection.

The cell classification accuracy affects the performance of the subsequent steps in our processing pipeline. Therefore, in Chapter 5, we propose two corrective approaches. The first one is based on rules (i.e., neighborhood patterns), while the second one makes use of refined features and machine learning techniques. Essentially, we study the immediate

and/or distant neighborhood of the classified cells, in order to identify those that are misclassified and afterward predict their true label.

Next, in Chapter 6, we address table detection in spreadsheets. At the beginning of this chapter, we propose a graph model that encodes the sheet layout. Concretely, the graph carries the attributes of layout regions (from Chapter 4) and describes their spatial arrangement in the sheet. Afterward, we partition the graph such that the resulting sub-graphs correspond to true tables in the sheet. To this end, we propose two novel approaches. The first one consists of a series of intuitive rules that partition the graph by omitting edges. The second one attempts to not only achieve high accuracy but also flexibility. Therefore, it incorporates an objective function that can be tuned to match the characteristics of the current dataset. Then, it employs genetic (evolutionary) algorithms to efficiently search for the optimal partitioning. The results of our experimental evaluation show that although both approaches are highly effective, the second one is typically more accurate. With these approaches, we manage to identify tables even in sheets with arbitrary arrangements, irregular tables, and cell misclassifications.

The last contribution of this thesis is a research prototype, discussed in Chapter 7. This tool implements the proposed processing pipeline. The user interface is an Excel add-in, while on the back-end specialized Python scripts execute the steps of the pipeline. Users can visually inspect the results from these scripts. In this way, they can provide their feedback about potential issues or improvements for the proposed approach. Nevertheless, to complete the picture, the tool allows the users to extract information from the detected tables.

All in all, the proposed pipeline is an intuitive and effective approach to deal with the automatic understanding of spreadsheet documents. Moreover, this approach is transferable and flexible. With regard to cell classification, we showed that our models can generalize better by training on spreadsheets from two different datasets (domains). Concerning table detection, we introduced an objective function that can be semi-automatically tuned to match the characteristics of new (unseen) spreadsheet datasets. Nevertheless, even our rule-based (table detection) approach can be adjusted, since it carries very few thresholds (parameters), which can be manually adjusted. Furthermore, the proposed pipeline is traceable. One can easily follow the individual steps to trace where the decision was made or where the potential error has occurred. Therefore debugging is straightforward, and this makes it easier to detect where there is a need for improvements. Last, the proposed pipeline is extendable. When necessary, additional steps can be added to the pipeline, or existing steps can be further specialized.

8.2 DIRECTIONS OF FUTURE WORK

Although this thesis has improved the state of the art (with regards to layout inference at table detection), there are still challenges and open issues that need to be addressed by future work. The user-centric and free-for-all nature of spreadsheets makes automatic understanding particularly hard. The high degree of diversity in contents and formatting leaves no room for naïve solutions. Instead, it calls for sophisticated processes that are composed of multiple specialized tasks. In this regard, we propose future research directions for existing or supplementary steps of our proposed pipeline.

Pre-Selection

In this thesis, we experiment with sheets that contain tables. However, as shown in Chapter 3, we find also sheets that contain forms, reports, lists, and other contents. Thus, to increase the automation of the process, future works can research mechanisms to distinguish sheets that potentially carry tables from those that do not.

Layout Analysis

Understanding the layout is essential for the subsequent tasks of the pipeline. If not for table detection, it is required for table analysis and information extraction. Therefore, future works need to improve even further the state of the art in layout analysis for spreadsheet documents.

One possible direction is to employ more sophisticated machine learning methods and features. In fact, two very recent publications have already proposed enhancements to our classification method [65, 67]. In addition to the features from this thesis, the authors of [65] use pre-trained cell embeddings. In some cases, this already gives a 6% improvement in the Macro-F1 score. Instead, the paper [67] makes use of Active Learning [116] to reduce the training time for cell classification in spreadsheets.

Future works should also revisit the proposed layout functions (labels). Our evaluation in Section 4.3 and the agreement assessment in Section 3.2.2 showed that some labels are more intuitive than others. Future works can eliminate or postpone some labels for later steps of the pipeline. For instance, *GroupHeaders* can be addressed after the table is detected. A similar approach is proposed by Chen et al.[29]. Moreover, there are labels that need further specialization. Most notably, the label *Other* is not as specific as the remaining labels. It is used to describe a very diverse collection of cells.

Table Detection

There are a few cases that were left outside the scope of this thesis and remain open for future work. Specifically, in Section 3.4.1, we excluded sheets with transposed tables and horizontally attached tables (i.e., no empty column in between them). These cases are rare, as per our empirical analysis (see Section 3.4.1). Yet, they occur in both of our annotated datasets.

There are several candidate improvements for our table detection approaches. One possible direction is to review the objective function (Section 6.4.4) and tuning function (Section 6.4.5), used for the genetic-based approach. The current formulations are linear functions, which allow computationally efficient evaluation and optimization. Future works might explore alternative (more descriptive) formulations, which might have higher computational demands but are able to capture more of the underlying characteristics and dependencies. Another direction is to apply entirely new approaches on top of the proposed graph model (see 6.2.1). An obvious candidate is the Graph Neural Networks, which have emerged in the last years [136, 133]. Very recently, GNNs have been considered for table detection in scanned documents [106, 109].

Analysis and Information Extraction

In Section 7.3, we proposed a simple algorithm to analyze the detected tables and export their data. However, there are additional possibilities in literature. For instance, to extract left-side hierarchies from tables, future research can add on top of our work the highly specialized method proposed by Chen et al. [30]. In addition, future research can incorporate the work of Cunha et al. [40], to construct a normalized database schema from the detected tables. Subsequently, using the refinement rules, proposed in the same paper [40], data can be moved from the spreadsheet document to an actual database. Both papers, [30] and [40], were discussed in more detail in Section 2.3.

Last, the analysis of textual metadata in spreadsheets is not addressed by this thesis and other existing works in literature. As mentioned in Section 6.2.3, metadata cells hold information that can potentially help us analyze the tables more accurately. However, one has to discover the relationship between these metadata and the detected tables. For instance, despite being closer to one of the tables, a *Note* cell might hold information that is relevant to multiple tables in the sheet. Therefore, the analysis of metadata has to go beyond spatial arrangement.

BIBLIOGRAPHY

[1] DEAP documentation. (Visited on 29 November, 2019). URL: https://deap.readthedocs.io/en/master/index.html.

[2] IronPython the Python programming language for the .NET Framework. (Visited on 29 November, 2019). URL: https://ironpython.net/.

[3] SciPy, a Python-based ecosystem of open-source software for mathematics, science, and engineering. (Visited on 29 November, 2019). URL: https://www.scipy.org/.

[4] The world's most valuable resource. *The Economist*, page 7, May 6th, 2017. URL: https://www.economist.com/leaders/2017/05/06/the-worlds-most-valuable-resource-is-no-longer-oil-but-data.

[5] Robin Abraham, Margaret Burnett, and Martin Erwig. Spreadsheet programming. *Wiley Encyclopedia of Computer Science and Engineering*, pages 2804–2810, 2007.

[6] Robin Abraham and Martin Erwig. Header and unit inference for spreadsheets through spatial analyses. In *VL/HCC*, pages 165–172. IEEE, 2004.

[7] Robin Abraham and Martin Erwig. Inferring templates from spreadsheets. In *Proceedings of the 28th international conference on Software engineering*, pages 182–191. ACM, 2006.

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

[9] Robin Abraham, Martin Erwig, Steve Kollmansberger, and Ethan Seifert. Visual specifications of correct spreadsheets. In *2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05)*, pages 189–196. IEEE, 2005.

[10] Marco D Adelfio and Hanan Samet. Presentation slides: "Schema extraction for tabular data on the web". 2013.

[11] Marco D Adelfio and Hanan Samet. Schema extraction for tabular data on the web. *Proceedings of the VLDB Endowment*, 6(6):421–432, 2013.

[12] Ioannis Alagiannis, Renata Borovica, Miguel Branco, Stratos Idreos, and Anastasia Ailamaki. Nodb: efficient query execution on raw data files. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 241–252, 2012.

[13] Akira Amano and Naoki Asada. Complex table form analysis using graph grammar. *Lecture notes in computer science*, pages 283–286, 2002.

[14] Christine M Anderson-Cook. Practical genetic algorithms, 2005.

[15] Sandro Badame and Danny Dig. Refactoring meets spreadsheet formulas. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 399–409. IEEE, 2012.

[16] Titus Barik, Kevin Lubick, Justin Smith, John Slankas, and Emerson Murphy-Hill. Fuse: a reproducible, extendable, internet-scale corpus of spreadsheets. In *the 12th Working Conference on Mining Software Repositories*, pages 486–489. IEEE, 2015.

[17] Mangesh Bendre. Dataspread. (Visited on 06 December, 2019). URL: http://dataspread.github.io/.

[18] Mangesh Bendre, Bofan Sun, Ding Zhang, Xinyan Zhou, Kevin Chen-Chuan Chang, and Aditya Parameswaran. Dataspread: Unifying databases and spreadsheets. *Proceedings of the VLDB Endowment*, 8(12):2000–2003, 2015.

[19] Mangesh Bendre, Vipul Venkataraman, Xinyan Zhou, Kevin Chang, and Aditya Parameswaran. Towards a holistic integration of spreadsheets with databases: A scalable storage engine for presentational data management. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 113–124. IEEE, 2018.

[20] Mangesh Bendre, Vipul Venkataraman, Xinyan Zhou, Kevin Chen-Chuan Chang, and Aditya Parameswaran. Scaling up to billions of cells with dataspread: Supporting large spreadsheets with databases. Technical report.

[21] Christopher M Bishop. *Pattern recognition and machine learning*. springer, 2006.

[22] Alexander Asp Bock. A literature review of spreadsheet technology. Technical report, IT University of Copenhagen, 2016.

[23] Paul T Boggs and Jon W Tolle. Sequential quadratic programming. *Acta numerica*, 4:1–51, 1995.

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

[25] Daniel Bricklin. *Bricklin on technology*. John Wiley & Sons, 2009.

[26] Daniel Bricklin and Bob Frankston. VisiCalc: Information from its creators. (Visited on 06 November, 2019). URL: http://www.bricklin.com/visicalc.htm.

[27] Aydın Buluç, Henning Meyerhenke, Ilya Safro, Peter Sanders, and Christian Schulz. Recent advances in graph partitioning. In *Algorithm Engineering*, pages 117–158. Springer, 2016.

[28] Bill Casselman. The Babylonian tablet Plimpton 322. (Visited on 07 November, 2019). URL: https://www.math.ubc.ca/~cass/courses/m446-03/pl322/pl322.html.

[29] Zhe Chen and Michael Cafarella. Automatic web spreadsheet data extraction. In *International Workshop on Semantic Search over the Web*, page 1. ACM, 2013.

[30] Zhe Chen and Michael Cafarella. Integrating spreadsheet data via accurate and low-effort extraction. In *the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1126–1135. ACM, 2014.

[31] Zhe Chen, Michael Cafarella, Jun Chen, Daniel Prevo, and Junfeng Zhuang. Senbazuru: A prototype spreadsheet database management system. *Proceedings of the VLDB Endowment*, 6(12):1202–1205, 2013.

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

[33] William W. Cohen. Fast effective rule induction. In *Twelfth International Conference on Machine Learning*, pages 115–123. Morgan Kaufmann, 1995.

[34] Microsoft Corporation. Collaborate with office 365. (Visited on 07 November, 2019). URL: https://support.office.com/en-us/article/collaborate-with-office-365-ac05a41e-0b49-4420-9ebc-190ee4e744f4.

[35] Microsoft Corporation. Excel help center. (Visited on 11 November, 2019). URL: https://support.office.com/en-us/excel.

[36] Microsoft Corporation. Office 365. (Visited on 07 November, 2019). URL: https://www.office.com/.

[37] Bertrand Coüasnon and Aurélie Lemaitre. Recognition of tables and forms. *Handbook of Document Image Processing and Recognition*, pages 647–677, 2014.

[38] Jácome Cunha, Martin Erwig, and João Alexandre Saraiva. Automatically inferring classsheet models from spreadsheets. In *Proceedings of the Symposium on Visual Languages and Human-Centric Computing-VL/HCC*, pages 93–100. IEEE, 2010.

[39] Jácome Cunha, João Paulo Fernandes, Jorge Mendes, and João Saraiva. Mdsheet: A framework for model-driven spreadsheet engineering. In *International Conference on Software Engineering*, pages 1395–1398. IEEE Press, 2012.

[40] Jácome Cunha, João Saraiva, and Joost Visser. From spreadsheets to relational databases and back. In *SIGPLAN workshop on Partial evaluation and program manipulation*, pages 179–188. ACM, 2009.

[41] Jácome Cunha, Joost Visser, Tiago Alves, and João Saraiva. Type-safe evolution of spreadsheets. In *International Conference on Fundamental Approaches to Software Engineering*, pages 186–201. Springer, 2011.

[42] Manoranjan Dash and Huan Liu. Feature selection for classification. *Intelligent data analysis*, 1(3):131–156, 1997.

[43] Haoyu Dong, Shijie Liu, Shi Han, Zhouyu Fu, and Dongmei Zhang. Tablesense: Spreadsheet table detection with convolutional neural networks. In *Association for the Advancement of Artificial Intelligence (AAAI)*, 2019.

[44] Wensheng Dou, Liang Xu, Shing-Chi Cheung, Chushu Gao, Jun Wei, and Tao Huang. Venron: a versioned spreadsheet corpus and related evolution analysis. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, pages 162–171. IEEE, 2016.

[45] Julian Eberius, Katrin Braunschweig, Markus Hentsch, Maik Thiele, Ahmad Ahmadov, and Wolfgang Lehner. Building the dresden web table corpus: A classification approach. In *In 2nd IEEE/ACM International Symposium on Big Data Computing (BDC)*. IEEE/ACM, 2015.

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

[47] Wayne W Eckerson and Richard P Sherman. Q&a: Strategies for managing spreadmarts. *Business Intelligence Journal*, 13(1):23, 2008.

[48] Agoston Endre Eiben and Selmar K Smit. Evolutionary algorithm parameters and methods to tune them. In *Autonomous search*, pages 15–36. Springer, 2011.

[49] David W Embley, Matthew Hurst, Daniel Lopresti, and George Nagy. Table-processing paradigms: a research survey. *International Journal of Document Analysis and Recognition (IJDAR)*, 8(2-3):66–86, 2006.

[50] Gregor Engels and Martin Erwig. Classsheets: automatic generation of spreadsheet applications from object-oriented specifications. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 124–133. ACM, 2005.

[51] European Spreadsheet Risks Interest Group (EuSpRIG). Horror stories. (Visited on 06 December, 2019). URL: http://www.eusprig.org/horror-stories.htm.

[52] Marc Fisher and Gregg Rothermel. The euses spreadsheet corpus: a shared resource for supporting experimentation with spreadsheet dependability mechanisms. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 1–5. ACM, 2005.

[53] Joseph L Fleiss. Measuring nominal scale agreement among many raters. *Psychological bulletin*, 76(5):378, 1971.

[54] Félix-Antoine Fortin, François-Michel De Rainville, Marc-André Gardner, Marc Parizeau, and Christian Gagné. DEAP: Evolutionary algorithms made easy. *Journal of Machine Learning Research*, 13:2171–2175, Jul 2012.

[55] Common Crawl Foundation. Common Crawl. (Visited on 27 December, 2019). URL: http://commoncrawl.org/.

[56] Eclipse Foundation. SWT: The Standard Widget Toolkit. (Visited on 10 December, 2019). URL: https://www.eclipse.org/swt/.

[57] The Apache Software Foundation. Apache POI - the Java API for Microsoft Documents. (Visited on 14 November, 2019). URL: https://poi.apache.org/.

[58] The Apache Software Foundation. Open office: The free and open productivity suite. (Visited on 29 November, 2019). URL: https://www.openoffice.org/.

[59] The Document Foundation. LibreOffice 6.3 Help: Welcome to the LibreOffice Calc Help. (Visited on 11 November, 2019). URL: https://help.libreoffice.org/6.3/.

[60] The Document Foundation. Libreoffice calc. (Visited on 06 November, 2019). URL: https://www.libreoffice.org/discover/calc.

[61] Raffaella Franci and Laura Toti Rigatelli. Towards a history of algebra from leonardo of pisa to luca pacioli. *Janus*, 72(1-3):17–82, 1985.

[62] Jöran Friberg. Methods and traditions of babylonian mathematics: Plimpton 322, pythagorean triples, and the babylonian triangle parameter equations. *Historia Mathematica*, 8(3):277–318, 1981.

[63] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. *The elements of statistical learning*, volume 1. Springer series in statistics New York, 2001.

[64] Eric Gazoni and Charlie Clack. openpyxl - A Python library to read/write Excel 2010 xlsx/xlsm files. (Visited on 14 November, 2019). URL: https://openpyxl.readthedocs.io/en/stable/.

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

[66] David E Goldberg. *Genetic algorithms*. Pearson Education India, 2006.

[67] Julius Gonsior, Josephine Rehak, Maik Thiele, Elvis Koci, Michael Günther, and Wolfgang Lehner. Active learning for spreadsheet cell classification. In *1st Workshop on Search, Exploration, and Analysis in Heterogeneous Datastores (SEAdata)*, 2020. (In press).

[68] Crina Grosan and Ajith Abraham. Hybrid evolutionary algorithms: methodologies, architectures, and reviews. In *Hybrid evolutionary algorithms*, pages 1–17. Springer, 2007.

[69] Felienne Hermans. Analyzing and visualizing spreadsheets, January 2013. Doctoral Thesis.

[70] Felienne Hermans and Danny Dig. Bumblebee: a refactoring environment for spreadsheet formulas. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 747–750. ACM, 2014.

[71] Felienne Hermans, Bas Jansen, Sohon Roy, Efthimia Aivaloglou, Alaaeddin Swidan, and David Hoepelman. Spreadsheets are code: An overview of software engineering approaches applied to spreadsheets. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 5, pages 56–65. IEEE, 2016.

[72] Felienne Hermans and Emerson Murphy-Hill. Enron's spreadsheets and related emails: A dataset and analysis. In *the 37th IEEE/ACM International Conference on Software Engineering*, volume 2, pages 7–16. IEEE, 2015.

[73] Felienne Hermans, Martin Pinzger, and Arie Van Deursen. Automatically extracting class diagrams from spreadsheets. In *European Conference on Object-Oriented Programming*, pages 52–75. Springer, 2010.

[74] Felienne Hermans, Martin Pinzger, and Arie van Deursen. Detecting code smells in spreadsheet formulas. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 409–418. IEEE, 2012.

[75] Jianying Hu, Ramanujan Kashi, Daniel Lopresti, George Nagy, and Gordon Wilfong. Why table ground-truthing is hard. In *Proceedings of Sixth International Conference on Document Analysis and Recognition*, pages 129–133. IEEE, 2001.

[76] Google Inc. Google sheets. (Visited on 06 November, 2019). URL: https://www.google.com/sheets/about/.

[77] Ecma International. Standard ECMA-376. (Visited on 11 November, 2019). URL: https://www.ecma-international.org/publications/standards/Ecma-376.htm.

[78] Bas Jansen and Felienne Hermans. The use of charts, pivot tables, and array formulas in two popular spreadsheet corpora. *Proceedings of the 5th International Workshop on Software Engineering Methods in Spreadsheets*, 2018.

[79] Sean Kandel, Andreas Paepcke, Joseph Hellerstein, and Jeffrey Heer. Wrangler: interactive visual specification of data transformation scripts. In *CHI'11.*, pages 3363–3372, New York, NY, USA, 2011. ACM. doi:10.1145/1978942.1979444.

[80] Shachar Kaufman, Saharon Rosset, Claudia Perlich, and Ori Stitelman. Leakage in data mining: Formulation, detection, and avoidance. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 6(4):1–21, 2012.

[81] Jin Kim, Inwook Hwang, Yong-Hyuk Kim, and Byung-Ro Moon. Genetic approaches for graph partitioning: a survey. In *GECCO'11*, pages 473–480.

[82] Elvis Koci, Dana Kuban, Nico Luettig, Dominik Olwig, Maik Thiele, Julius Gonsior, Wolfgang Lehner, and Oscar Romero. Xlindy: Interactive recognition and information extraction in spreadsheets. In Sonja Schimmler and Uwe M. Borghoff, editors, *Proceedings of the ACM Symposium on Document Engineering 2019, Berlin, Germany, September 23-26, 2019*, pages 25:1–25:4. ACM, 2019.

[83] Elvis Koci, Maik Thiele, Wolfgang Lehner, and Oscar Romero. Table recognition in spreadsheets via a graph representation. In *the 13th IAPR International Workshop on Document Analysis Systems (DAS)*, pages 139–144. IEEE, 2018.

[84] Elvis Koci, Maik Thiele, Josephine Rehak, Oscar Romero, and Wolfgang Lehner. DECO: A dataset of annotated spreadsheets for layout and table recognition. In *2019 International Conference on Document Analysis and Recognition, ICDAR 2019, Sydney, Australia, September 20-25, 2019*, pages 1280–1285. IEEE, 2019.

[85] Elvis Koci, Maik Thiele, Oscar Romero, and Wolfgang Lehner. A machine learning approach for layout inference in spreadsheets. In *IC3K 2016: The 8th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management: volume 1: KDIR*, pages 77–88. SciTePress, 2016.

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

[87] Elvis Koci, Maik Thiele, Oscar Romero, and Wolfgang Lehner. Cell classification for layout recognition in spreadsheets. In Ana Fred, Jan Dietz, David Aveiro, Kecheng Liu, Jorge Bernardino, and Joaquim Filipe, editors, *Knowledge Discovery, Knowledge Engineering and Knowledge Management (IC3K '16: Revised Selected Papers)*, volume 914 of *Communications in Computer and Information Science*, pages 78–100. Springer, Cham, 2019.

[88] Elvis Koci, Maik Thiele, Oscar Romero, and Wolfgang Lehner. A genetic-based search for adaptive table recognition in spreadsheets. In *2019 International Conference on Document Analysis and Recognition, ICDAR 2019, Sydney, Australia, September 20-25, 2019*, pages 1274–1279. IEEE, 2019.

[89] Jirka Kosek. From the office document format battlefield. *IT Professional*, 10(3):51–55, 2008.

[90] J Richard Landis and Gary G Koch. The measurement of observer agreement for categorical data. *biometrics*, pages 159–174, 1977.

[91] Michael Levandowsky and David Winter. Distance between sets. *Nature*, 234(5323):34–35, 1971.

[92] Bin Liu and HV Jagadish. A spreadsheet algebra for a direct data manipulation query interface. In *2009 IEEE 25th International Conference on Data Engineering*, pages 417–428. IEEE, 2009.

[93] David Maier. *The theory of relational databases*, volume 11. Computer science press Rockville, 1983.

[94] Song Mao, Azriel Rosenfeld, and Tapas Kanungo. Document structure analysis algorithms: a literature survey. In *Document Recognition and Retrieval X*, volume 5010, pages 197–207. International Society for Optics and Photonics, 2003.

[95] Simone Marinai. Introduction to document analysis and recognition. In *Machine learning in document analysis and recognition*, pages 1–20. Springer, 2008.

[96] Hrushikesha Mohanty, Prachet Bhuyan, and Deepak Chenthati. *Big Data: A Primer*. Springer India, 2015.

[97] Anoop M Namboodiri and Anil K Jain. Document structure and layout analysis. In *Digital Document Processing*, pages 29–48. Springer, 2007.

[98] Nuix and EDRM. The Enron PST Data Set Cleansed of PII by Nuix and EDRM. (Visited on 27 December, 2019). URL: http://info.nuix.com/Enron.html.

[99] OASIS. OASIS Open Document Format for Office Applications (OpenDocument) TC. (Visited on 11 November, 2019). URL: https://www.oasis-open.org/committees/office/charter.php.

[100] Daniel E. O'Leary. Embedding ai and crowdsourcing in the big data lake. *IEEE Intelligent Systems*, 29(5):70–73, 2014.

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

[102] David MW Powers. The problem with kappa. In *Proceedings of the 13th Conference of the European Chapter of the Association for Computational Linguistics*, pages 345–355. Association for Computational Linguistics, 2012.

[103] The GNOME Project. Gnumeric. (Visited on 06 November, 2019). URL: http://www.gnumeric.org/.

[104] The GNOME Project. The gnumeric manual, version 1.12. (Visited on 11 November, 2019). URL: https://help.gnome.org/users/gnumeric/.

[105] The Lemur Project. The ClueWeb09 Dataset. (Visited on 27 December, 2019). URL: http://lemurproject.org/clueweb09.php/.

[106] Shah Rukh Qasim, Hassan Mahmood, and Faisal Shafait. Rethinking table recognition using graph neural networks. In *2019 International Conference on Document Analysis and Recognition (ICDAR)*, pages 142–147. IEEE, 2019.

[107] Cliff T Ragsdale. *Spreadsheet modeling and decision analysis*. Thomson south-western, 2004.

[108] M Armon Rahgozar and Robert Cooperman. A graph-based table recognition system. *Document Recognition*, 111:192–203, 1996.

[109] Pau Riba, Anjan Dutta, Lutz Goldmann, Alicia Fornés, Oriol Ramos, and Josep Lladós. Table detection in invoice documents by graph neural networks. In *2019 International Conference on Document Analysis and Recognition (ICDAR)*. IEEE, 2019.

[110] Grzegorz Rozenberg. *Handbook of graph grammars and computing by graph transformation*, volume 1. World scientific, 1997.

[111] JM Sachs. Recollections: Developing lotus 1-2-3. *IEEE Annals of the History of Computing*, 3(29):41–48, 2007.

[112] Alan Sangster and Giovanna Scataglinibelghitar. Luca pacioli: the father of accounting education. *Accounting Education: an international journal*, 19(4):423–438, 2010.

[113] Satu Elisa Schaeffer. Graph clustering. *Computer science review*, 1(1):27–64, 2007.

[114] Thomas Schmitz, Dietmar Jannach, Birgit Hofer, Patrick W. Koch, Konstantin Schekotihin, and Franz Wotawa. A decomposition-based approach to spreadsheet testing and debugging. In *VL/HCC*, pages 117–121. IEEE Computer Society, 2017.

[115] Peter Sestoft. Spreadsheet implementation technology, 2014.

[116] Burr Settles. Active learning literature survey. Technical report, University of Wisconsin-Madison Department of Computer Sciences, 2009.

[117] Alexey Shigarov, Vasiliy Khristyuk, Andrey Mikhailov, and Viacheslav Paramonov. Tabbyxl: Rule-based spreadsheet data extraction and transformation. In *International Conference on Information and Software Technologies*, pages 59–75. Springer, 2019.

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

[119] Alexey O Shigarov, Viacheslav V Paramonov, Polina V Belykh, and Alexander I Bondarev. Rule-based canonicalization of arbitrary tables in spreadsheets. In *International Conference on Information and Software Technologies*, pages 78–91. Springer, 2016.

[120] Robert Slater. *Portraits in silicon*. Mit Press, 1989.

[121] OpenDoc Society. ODF tools. (Visited on 14 November, 2019). URL: http://www.opendocsociety.org/tools/odf-tools/.

[122] Yanmin Sun, Andrew KC Wong, and Mohamed S Kamel. Classification of imbalanced data: A review. *International journal of pattern recognition and artificial intelligence*, 23(04):687–719, 2009.

[123] Pang-Ning Tan, Michael Steinbach, and Vipin Kumar. *Introduction to data mining*. Pearson Education India, 2016.

[124] Jerzy Tyszkiewicz. Spreadsheet as a relational database engine. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 195–206. ACM, 2010.

[125] Ruben Verborgh and Max De Wilde. *Using OpenRefine*. Packt Publishing Ltd, 2013.

[126] Xinxin Wang. Tabular abstraction, editing, and formatting. Technical report, University of Waretloo, Waterloo, Ontaria, Canada, 1996.

[127] WinWorld. Multiplan 1.x. (Visited on 07 November, 2019). URL: https://winworldpc.com/product/multiplan/106.

[128] WinWorld. SuperCalc 5.1. (Visited on 07 November, 2019). URL: https://winworldpc.com/product/supercalc/51.

[129] Andrew Witkowski, Srikanth Bellamkonda, Tolga Bozkaya, Gregory Dorman, Nathan Folkert, Abhinav Gupta, Lei Shen, and Sankar Subramanian. Spreadsheets in rdbms for olap. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 52–63. ACM, 2003.

[130] Andrew Witkowski, Srikanth Bellamkonda, Tolga Bozkaya, Nathan Folkert, Abhinav Gupta, Lei Sheng, and Sankar Subramanian. Business modeling using sql spreadsheets. In *Proceedings 2003 VLDB Conference*, pages 1117–1120. Elsevier, 2003.

[131] Andrew Witkowski, Srikanth Bellamkonda, Tolga Bozkaya, Aman Naimat, Lei Sheng, Sankar Subramanian, and Allison Waingold. Query by excel. In *Proceedings of the 31st international conference on Very large data bases*, pages 1204–1215. VLDB Endowment, 2005.

[132] Ian H Witten and Eibe Frank. Data mining: practical machine learning tools and techniques with java implementations. *Acm Sigmod Record*, 31(1):76–77, 2002.

[133] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S Yu. A comprehensive survey on graph neural networks. *arXiv preprint arXiv:1901.00596*, 2019.

[134] Richard Zanibbi, Dorothea Blostein, and James R Cordy. A survey of table recognition. *Document Analysis and Recognition*, 7(1):1–16, 2004.

[135] Zhi-Qiang Zeng, Hong-Bin Yu, Hua-Rong Xu, Yan-Qi Xie, and Ji Gao. Fast training support vector machines using parallel sequential minimal optimization. In *2008 3rd international conference on intelligent system and knowledge engineering*, volume 1, pages 997–1001. IEEE, 2008.

[136] Jie Zhou, Ganqu Cui, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. Graph neural networks: A review of methods and applications. *arXiv preprint arXiv:1812.08434*, 2018.

[137] Melissa Rodriguez Zynda. The first killer app: a history of spreadsheets. *ACM Interactions*, 20(5):68–72, 2013.

LIST OF FIGURES

1.1 Organization of the chapters in this thesis. . . . 16

2.1 User Interfaces for Different Spreadsheet Vendors . . . 21
(a) Microsoft Excel 2016 . . . 21
(b) Google Sheets . . . 21
(c) LibreOffice v6.3 . . . 21
(d) Gnumeric v1.12 . . . 21
2.2 The Document Analysis and Recognition Process . . . 24
2.3 The Wang Model [134] . . . 25
2.4 Chen et al.: Automatic Web Spreadsheet Data Extraction [29] . . . 27

3.1 Row Labels as Defined by Adelfio and Samet [10] . . . 35
3.2 Cell Annotation Labels . . . 37
3.3 Screenshots from the Annotation Tool . . . 39
(a) User Interface and Menus . . . 39
(b) Saved Annotation Data . . . 39
3.4 Annotation Steps . . . 40
3.5 Number of Sheets per N/A Label . . . 43
(a) Counts in Not-Applicable Files . . . 43
(b) Counts in Completed Files . . . 43
3.6 Table Annotations in Numbers . . . 43
(a) Table Distribution . . . 43
(b) Employee Sheet Counts . . . 43
3.7 Cell Annotations . . . 44
(a) Sheet Counts per Cell Label . . . 44
(b) Content Type Ratios per Cell Label . . . 44
3.8 Coverage and Arrangements in Enron Spreadsheets . . . 45
(a) Table Coverage . . . 45
(b) Arrangements . . . 45
3.9 Content Density in Enron Spreadsheets . . . 46
(a) Sheet Densities . . . 46
(b) Table Densities . . . 46
3.10 Gaps and Dependencies in Enron Spreadsheets . . . 46
(a) Gaps Between/Inside . . . 46
(b) Formula References . . . 46
3.11 Characteristics of Annotated Fuse Sample . . . 48
(a) *N/A* Sheet Counts[1] . . . 48
(b) Occurrences Cell Labels . . . 48
(c) Content Type Ratios . . . 48
(d) Table Distribution . . . 48
(e) Table Sheet Coverage . . . 48
(f) Sheet Densities . . . 48
(g) Table Densities . . . 48
(h) Arrangements . . . 48
(i) Gaps Inside/Between . . . 48
(j) Formula References . . . 48

3.12 A Spreadsheet Processing Pipeline 52

4.1 The Layout Analysis Process 57
4.2 The Immediate Neighborhood of a Cell 61
4.3 Training Cell Classifiers 64
4.4 Analysis of Sampling Sizes for Enron 67
(a) Average Macro-F1 per Sampling Size 67
(b) Average Weighted-F1 per Sampling Size 67
4.5 Analysis of Sampling Sizes for Fuse 68
(a) Average Macro-F1 per Sampling Size 68
(b) Average Weighted-F1 per Sampling Size 68
4.6 Building Layout Intervals 80
(a) Annotated Cells 80
(b) Classified Cells 80
(c) Layout Intervals 80
4.7 Building Layout Regions 81
(a) Layout Intervals 81
(b) Strict Layout Regions 81
(c) Non-Strict Layout Regions 81

5.1 Annotation Statistics for the KDIR Dataset [85] 84
(a) Sheets per Corpora 84
(b) Tables per Corpora 84
5.2 The Building Blocks [85] 85
5.3 Misclassification Patterns 86
(a) Tunel 86
(b) T-blocks 86
(c) AIN 86
(d) RIN 86
(e) Corner 86
5.4 Region-Based Approach 88
(a) Load 88
(b) Standardize 88
(c) Identify 88
(d) Relabel 88
5.5 Original Worksheet 89
(a) Row Intervals 89
(b) Regions 89
5.6 Pivoted Worksheet 89
(a) Column Intervals 89
(b) Regions 89
5.7 Standardization Procedure 90
5.8 Region Analysis 90
(a) Overall Assessment 90
(b) Mixed Regions 90
5.9 Size Occurrences in Misclassified Regions 91
5.10 Misclassification Identification Results 95
(a) In Regions 95
(b) In Cells 95
5.11 Relabeling Results 96
(a) In Regions 96
(b) In Cells 96
5.12 Confidence Score Analysis 97

6.1 Framework of TableSense for Spreadsheet Table Detection [43] 101
6.2 Overview of the Proposed Approach for Table Detection 101

6.3 The Proposed Graph Representation . . . 104
6.4 Reducing Labels Prior to Table Detection . . . 106
(a) Original Classified Cells . . . 106
(b) Layout Regions after Reducing Labels . . . 106
6.5 Building a Graph after Pre-Processing . . . 107
(a) Omitting METADATA Regions . . . 107
(b) The Graph Representation . . . 107
6.6 Table layouts, cases b, c, e, and f also occur for tables with vertical fences . 109
(a) Typical Horizontal . . . 109
(b) Misclassifications . . . 109
(c) Sparse Data . . . 109
(d) Typical Vertical . . . 109
(e) Sparse All . . . 109
(f) Conjoined . . . 109
6.7 The RAC Approach . . . 110
(a) Input Graph . . . 110
(b) Horizontal Groups . . . 110
(c) Vertical Groups . . . 110
6.8 The impact of misclassifications . . . 111
(a) True Layout Roles . . . 111
(b) Misclassified Cells . . . 111
6.9 The RAC Approach . . . 113
(a) Vertical Group MBRs . . . 113
(b) Handle Overlaps . . . 113
(c) Pair Left and Right . . . 113
6.10 Graph Model for the Genetic-Based Approach . . . 115

7.1 UI of the XLIndy tool . . . 128
7.2 Architecture of XLIndy . . . 129
7.3 Exporting Data from Detected Tables . . . 131

A.1 Characteristics of Reduced Enron Sample . . . 155
(a) Occurrences Cell Labels . . . 155
(b) Content Type Ratios . . . 155
(c) Table Distribution . . . 155
(d) Table Sheet Coverage . . . 155
(e) Sheet Densities . . . 155
(f) Table Densities . . . 155
(g) Arrangements . . . 155
(h) Gaps Inside/Between . . . 155
(i) Formula References . . . 155
A.2 Characteristics of Reduced Fuse Sample . . . 156
(a) Occurrences Cell Labels . . . 156
(b) Content Type Ratios . . . 156
(c) Table Distribution . . . 156
(d) Table Sheet Coverage . . . 156
(e) Sheet Densities . . . 156
(f) Table Densities . . . 156
(g) Arrangements . . . 156
(h) Gaps Inside/Between . . . 156
(i) Formula References . . . 156

B.1 Table types handled by the TIRS framework. The cases b, c, e, and f can also occur for tables with vertical fences. . . . 159
(a) Typical Horizontal . . . 159
(b) Misclassifications . . . 159
(c) Sparse Data . . . 159
(d) Typical Vertical . . . 159
(e) Sparse All . . . 159
(f) Conjoined . . . 159

LIST OF TABLES

3.1 Annotation Agreement Assessment . . . 41
3.2 Percentage per *N/A* Label . . . 49
3.3 Percentage of Applicable Sheets containing each Cell Label . . . 49
3.4 Occurrences of Densities . . . 49
3.5 % of Tables with Gaps . . . 49
3.6 Percentage of Applicable Sheets with the Following Arrangements . . . 50
3.7 Percentage of Simple Sheets . . . 50

4.1 Cell Counts per Annotation Label in Enron Dataset . . . 64
4.2 Cell Counts per Annotation Label in Fuse Dataset . . . 64
4.3 Enron Optimal Sampling . . . 68
4.4 Fuse Optimal Sampling . . . 68
4.5 Number of Selected Features per Algorithm for Classification with 6 Labels 70
4.6 Top 20 Features for Enron . . . 70
4.7 Top 20 Features for Fuse . . . 70
4.8 Optimal Parameters per Algorithm for Classification with 6 Labels . . . 71
4.9 F1-Score per Class for Enron with 6 Labels . . . 73
4.10 F1-Score per Class for Fuse with 6 Labels . . . 73
4.11 Optimal Configuration for 7 Labels . . . 74
4.12 F1-Score per Class for Enron with 7 Labels . . . 75
4.13 F1-Score per for Fuse with 7 Labels . . . 75
4.14 Results for 6 Labels when Training on the Combined Dataset . . . 76
4.15 Results for 7 Labels when Training on the Combined Dataset . . . 77
4.16 Optimal Configurations for the Combined Dataset . . . 77
4.17 Best Classification Results (F1-scores) from Related Work . . . 78
4.18 Best Classification Results (F1-scores) from this Thesis . . . 78
4.19 Comparing Classification Accuracy in Tables . . . 79

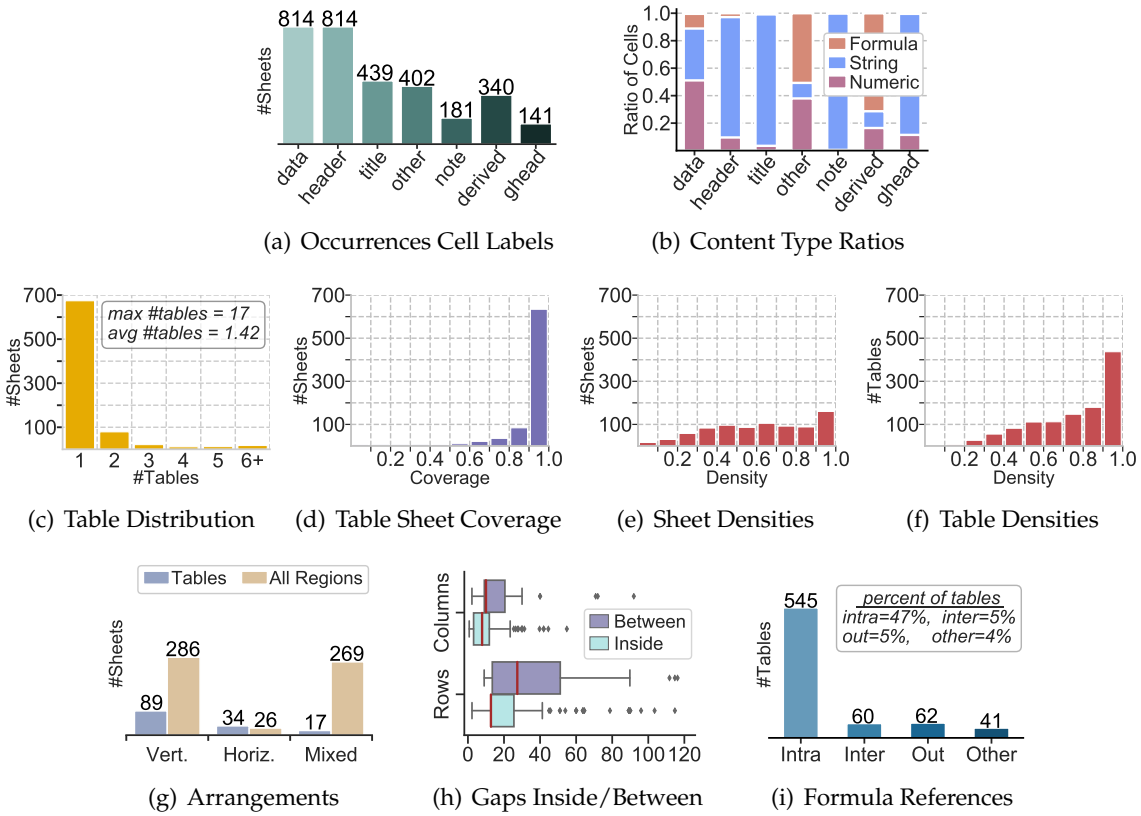
5.1 Pattern Occurrences . . . 87
5.2 Label Flips . . . 87
5.3 Region Features . . . 92
5.4 Comparing Classifiers for Misclassification Identification . . . 95
5.5 Relabeling: Trained on Annotated Regions . . . 96

6.1 Induced Noise per Training Dataset . . . 121
6.2 Percentage of Detected Tables for Classification with 7 Labels . . . 124
6.3 Percentage of Detected Tables for Classification with 6 Labels . . . 124
6.4 Percentage of Detected Tables for Ground Truth Cells . . . 124

A

ANALYSIS OF REDUCED SAMPLES

Here, we have performed once more the analyses from Section 3.3, for the reduced samples. The results are summarized with charts, shown separately in Figure A.2 and in Figure A.2. As mentioned in Section 3.4.1, we omit 40 sheets from the Enron sample and 10 sheets from the Fuse sample. The excluded sheets exhibit characteristics that are outside of the scope of this thesis. Moreover, 3 sheets from the Enron sample and 1 from the Fuse sample were omitted due to an exceptionally high number of tables.

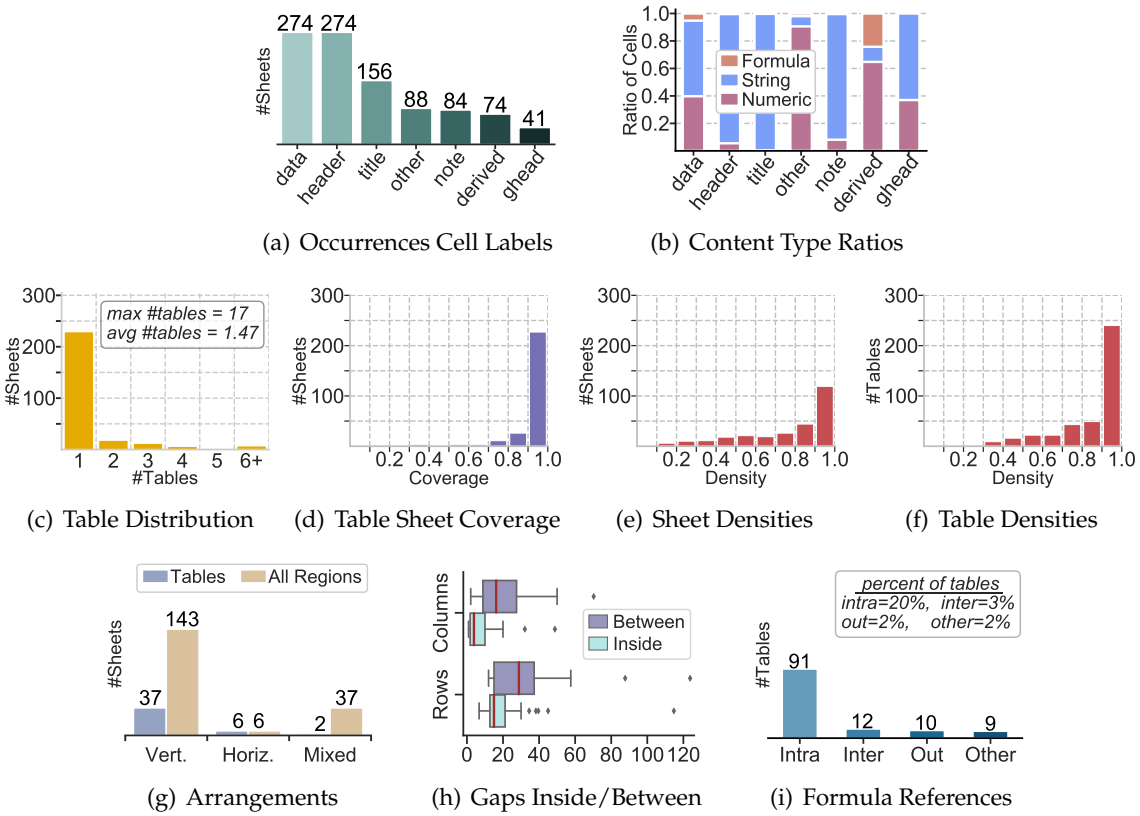


(a) Occurrences Cell Labels (b) Content Type Ratios

(c) Table Distribution (d) Table Sheet Coverage (e) Sheet Densities (f) Table Densities

(g) Arrangements (h) Gaps Inside/Between (i) Formula References

Figure A.1: Characteristics of Reduced Enron Sample



(a) Occurrences Cell Labels (b) Content Type Ratios

(c) Table Distribution (d) Table Sheet Coverage (e) Sheet Densities (f) Table Densities

(g) Arrangements (h) Gaps Inside/Between (i) Formula References

Figure A.2: Characteristics of Reduced Fuse Sample

B

TABLE DETECTION WITH TIRS

TIRS (Table Identification and Reconstruction in Spreadsheets) consists of a series of rules and heuristics that are based on the concepts presented in Chapter 4 and Chapter 6. In addition to covering various table layouts, with TIRS we attempt to minimize the effects of incorrect classifications and empty cells (i.e., missing values). Furthermore, we opted for rules that work on sheets having multiple tables, stacked horizontally and/or vertically.

For the TIRS framework, we used the KDIR dataset, discussed in Section 5.1. This means TIRS utilizes 5 labels, instead of the 7 introduced in Chapter 3. Concretely, the label $Attribute$ is used to describe hierarchies on the left columns of tables. Moreover, the label $Metadata$ is used to collectively describe the instances of $Title$, $Note$, and $Other$ cells. Another difference is that $Derived$ can occur both horizontally (row-wise) and vertically (column-wise), within the table.

Here, similarly to the other two table detection approaches from Chapter 6, we group the classified cells into layout regions ($\mathcal{LR}s$). However, for the TIRS approach, we use the *non-strict* variation (refer to Section 4.4). Below, we discuss how such regions are used to detect tables.

B.1 TABLES IN TIRS

Data, Header, and Attribute regions play the most important role in our analysis. Intuitively, a Data region ($\mathcal{LRD}$) acts like the core that brings everything together. A Header ($\mathcal{LRH}$) and Attribute region ($\mathcal{LRA}$) can help us distinguish the boundaries of tables. Therefore, we refer to them as "fences", a term borrowed from [6]. Fences can be horizontal (i.e., Headers) or vertical (i.e., both Headers and Attributes)

A valid table should have at least a fence ($\mathcal{LRF}$) paired with a $\mathcal{LRD}$. In terms of dimension, tables must be at least a 2×2 ranges of cells. This means that $\mathcal{LRD}$ and $\mathcal{LRF}$ regions are at least 1×2 or 2×1 ranges of cells.

$$table := \{Data, HHeaders, VHeaders, Attributes, Derived, Metadata, Other\}$$

Tables extracted by TIRS can be stored as collections of layout regions ($\mathcal{LR}s$). Specifically, as shown above, a table has seven distinct sets of $\mathcal{LR}s$. In most of the cases, we organize the regions forming the table by their label. We specialize Headers to vertical and horizontal. While the set "Other" contains regions for which the layout function is uncertain or possible misclassification. We provide more details on the latter in the following sections.

Finally, we utilize the MBR concept for tables, in addition to label regions. A table MBR is the minimum bounding rectangle for the $\mathcal{LR}s$ that compose it.

B.2 PAIRING FENCES WITH DATA REGIONS

As mentioned in the previous section, TIRS needs to pair $\mathcal{LRD}s$ with $\mathcal{LRF}s$ to form tables. Valid pairs comply with the following three conditions.

C1. The $\mathcal{LRF}$ is on the top or on the left of the $\mathcal{LRD}$ although not necessarily adjacent to it.

C2. For a $\mathcal{LRF}$, the selected $\mathcal{LRD}$ is the closest. Specifically, for a horizontal fence, we measure the Euclidean distance from the top edge of the Data region. Respectively, we work with the left edge for vertical fences.

C3. The pair of MBRs representing correspondingly the $\mathcal{LRD}$ and the $\mathcal{LRF}$ are projected in one of the axes, depending on the fence orientation. The length of the segment shared by both projections represents the overlap. We transform the overlap into a ratio by dividing it with the largest projection.

$$\frac{Overlap(xProjection(\mathcal{LRD}), xProjection(\mathcal{LRF}))}{Max(xProjection(\mathcal{LRD}), xProjection(\mathcal{LRF}))} > \theta \tag{B.1}$$

Equation B.1 shows how to calculate this for the x-axis (relevant to horizontal fences). The threshold θ was determined empirically and set to 0.5.

B.3 HEURISTICS FRAMEWORK

The TIRS framework is composed of eight heuristic steps. The initial Data-Fence pairs are created in the first five steps. While the subsequent activities aim at completing the table construction by incorporating the remaining unpaired regions. In the following paragraphs, we discuss the individual steps and illustrate them with examples from Figure B.1. Moreover, Algorithm B.1 provides pseudo-code that demonstrates the overall process.

We note that the examples in Figure B.1 hide the complexity of tables in our real-world dataset. For instance, fences might contain hierarchical structures, spanning multiple rows or columns. Furthermore, misclassifications and empty cells can occur in arbitrary locations and implicate various label regions (not only fences).

S1. In the first step, we attempt to create one-to-one pairs of Fence-Data, based on the three conditions listed in Section B.2. Figure B.1.a and Figure B.1.d provide examples of such tables.

S2. Mainly due to misclassifications multiple fence regions can be found that satisfy C1 and C2, but fail to comply with C3. An example is shown in Figure B.1.b. In such cases, we treat the individual regions as one composite fence, omitting the in-between "barriers". Equation 2 and 3 respectively show how to calculate the overlap ratio and projection-length to the x-axis for a composite fence ($\mathcal{CF}$), containing N sub-regions. We handle these calculations similarly for y-axis projections. Having the results from the equations, we proceed to check if C3 is satisfied.

$$cmp\_overlp = \sum_{i=1}^{N} Overlap(xProjection(\mathcal{LRD}), xProjection(\mathcal{CF}_i)) \tag{B.2}$$

$$cmp\_length = \sum_{i=1}^{N} xProjection(\mathcal{CF}_i) \tag{B.3}$$

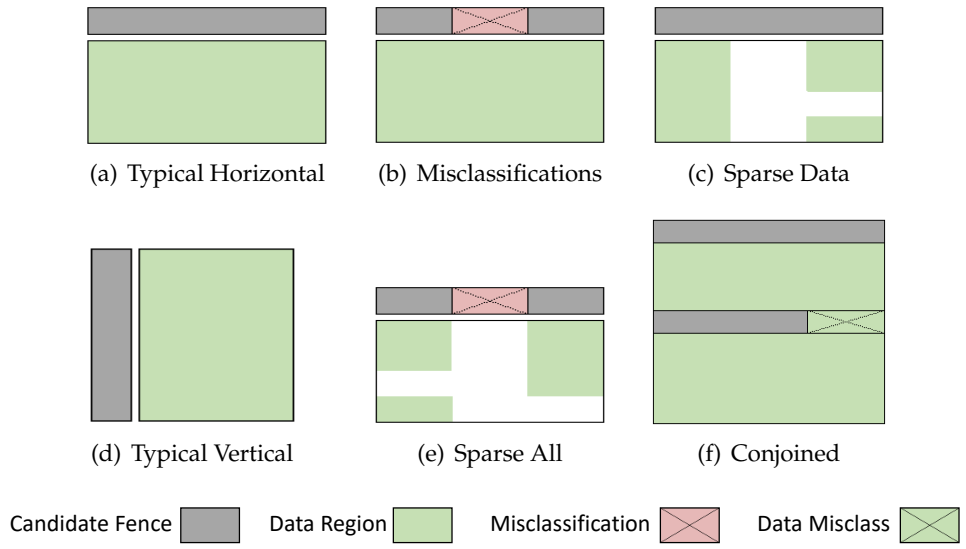


Figure B.1: Table types handled by the TIRS framework. The cases b, c, e, and f can also occur for tables with vertical fences.

S3. There can be a fence (simple or composite) that satisfies C3, but it is located inside the Data region far from the top edge or left edge. This might happen due to incorrect classification in worksheets that contain conjoined tables (i.e., not separated by empty columns or rows). We provide an example in Figure B.1.f. When such a fence is identified, we separate the Data region into two parts. When the fence is horizontal, we pair it with the lower part, otherwise with the right part.

S4. There are cases where "small" Data regions are under or on the right of a "bigger" fence (e.g. the table in Figure B.1.c). For these cases, the fence is treated as a first-class citizen. Data regions that comply to condition C1 and are closer to this fence, than other ones, are grouped together. Again, we use similar formulas to Equation 2 and 3 to calculate the overlap and the projection-length of composite Data regions.

S5. At this step, we take a much more aggressive approach, in order to form tables with the remaining unpaired regions. We start by grouping fences. When working horizontally, we merge fences whose y-axis projections overlap. Likewise, we look for overlaps on the x-axis for vertical fences. Afterward, we proceed in the same way as in step S4. Figure B.1.e illustrates a table that can be the output of this step.

S6. Here, we attempt to incorporate unpaired regions located in-between existing tables (i.e., constructed during S1-S5). In addition to the Data and fences, we also consider Metadata and Derived regions. For a pair of tables stacked horizontally, we assign the unpaired regions to the top table. When working with vertically stacked tables, we favor the left one. Obviously, this and the following step, make sense when there are more than one extracted tables.

S7. We proceed by merging tables whose MBRs overlap. This will correct inconsistencies that might have happened during the previous steps. For example, a Data region is partially under a fence from another table.

S8. Finally, we assign the remaining unpaired regions, regardless of label, to the nearest existing table.

Algorithm B.1: Table creation in TIRS

Input: Set of $\mathcal{LRDS}$ (D), set of $\mathcal{LRHS}$ (H), set of $\mathcal{LRAS}$ (A)
Output: Set of extracted tables from the sheet (T)

```
1  begin
2      T ← ∅;
3      UF ← ∅, UD ← D ;                          // UF: unpaired LRFs,UD: unpaired LRDs
4      O ← {Horizontal, Vertical};
5      foreach o in O do
6          if o == Horizontal then UF ← H else UF ← UF ∪ A;
7          foreach d in UD do
8              f ← GetNext(UF), newtbl ← false;
9              while f ≠ null and newtbl == false do
10                 if IsValidPair(f,d,o) then                          // S1: line 10-12
11                     (T, UF, UF) ← Construct(f,d,UD,UF,T,o);
12                     newtbl = true;
13                 else if IsDataBreaker(d,f) then                     // S3: line 13-16
14                     (d1, d2) ← BreakInTwoParts(d,f);
15                     (T, UF, UF) ← Construct(f,d2,UD,UF,T,o);
16                     d ← d1;
17                 f ← GetNext(UF);
18             if newtbl == false then                                 // S2: line 18-20
19                 CF ← GetCompositeFence(d,UF,o);
20                 if IsValidPair(d,CF,o) then
21                     (T, UF, UF) ← Construct(d,CF,UD,UF,T,o)
22     UH ← UF ∩ H, UA ← UF ∩ A;                 // Extract unpaired Headers & Attributes
23     foreach o in O do
24         if o == Horizontal then UF ← UH else UF ← UF ∪ UA;
25         foreach f in UF do                                          // S4: line 23-25
26             CD ← GetCompositeData(f,o,UD);
27             if IsValidPair(CD,f,o) then
28                 (T, UD, UF) ← Construct(CD,f,UD,UF,T,o);
29         foreach MF in MergeByOrientation(UF,o) do                   // S5: line 26-28
30             CD ← GetCompositeData(MF,o,UD);
31             if IsValidPair(CD,MF,o) then
32                 (T, UF, UF) ← Construct(CD,MF,UD,UF,T,o)
33     return T;
34 Function Construct(SD,SF,UD,UF,T,o):          // SD: Selected LRDs, SF: Selected LRFs
35     table ← CreateTable(SD,SF,o);
36     TT ← T, TUD ← UD, TUF ← UF;                // Temporary variables in this function
37     (TUD, TUF) ← FilterOutPaired(table,TUD,TUF);
38     ConT ← HandleTableBreakers(table,TUF,o);          // Trivial case ConT = {table}
39     foreach t in ConT do
40         foreach u in {TUD ∪ TUF} do
41             if IsInside(table,u) or IsOverlap(table,u) then AddOtherRegion(table,u) ;
42         (TUD, TUF) ← FilterOutPaired(table,TUD,TUF);
43         TT ← TT ∪ {t};
44     return (TT, TUD, TUF);
```

Algorithm B.1 provides a high-level view from the execution of table creation steps (S1-S5). For each individual step S1 to S5, we first process horizontal and then vertical fences. Our empirical analysis showed the former are by far more common, thus we prioritize them. Additionally, we give priority to Headers over Attributes. It is fair to claim that Headers represent more "secure" fences since fewer misclassifications involve this label compared to Attributes [85]. Another detail is that the steps S4 and S5 are executed only after all the types of fences are processed by steps S1-S3.

Furthermore, to avoid any inconsistencies, after the table creation we execute a series of operations. We incorporate regions that partially overlap or fall inside (complete overlap)

the table (lines $34 - 35$). We exclude the paired regions from the next iterations (lines 31 and 36). Also, we call function *HandleTableBreakers*, which basically is a batch execution of step S3.

Finally, at line 35 we use function *AddOtherRegion*. At this point in the algorithm, we can not tell what the role of the fully or partially overlapping region is, since we already have paired the main components of the table. Therefore, we keep such regions at a special set called "Other".

CONFIRMATION

I confirm that I independently prepared the thesis and that I used only the references and auxiliary means indicated in the thesis.

Dresden, April 27, 2020