Semi-structured Document Feature Extraction

Romuald Rousseau

2024-03-02

Abstract

The document discusses the challenges organizations face in dealing with semi-structured documents, particularly spreadsheets, due to their diverse formats and lack of standardization. It highlights the presence of defects within spreadsheets, often unnoticed by end-users, which pose difficulties for automated processes. The document proposes a method to classify spreadsheet elements and create a structured format resembling a JSON file to address these challenges.

1 Introduction

In the current data-driven environment, grappling with the intricacies of semi-structured documents presents a notable hurdle for organizations. These documents, marked by varying formats and a lack of uniformity, frequently demand specific expertise for efficient handling and analysis. Among these documents are spreadsheets, which are ubiquitous across all organizations. Despite being used extensively, they often harbor imperfections that are typically unnoticed by end-users but pose obstacles for automated procedures. Moreover, while containing tabular data, they may also include unstructured text around them.

Below an examples of spreadsheets with a mixed of tabular data, unstructured data and defects (Blank rows or columns put here for aesthetics or by mistakes):

Figure 1: Spreadsheet Example

A document very important 2023-Feb-01

Product 1

Date	Client	Qty	Amount
2023-Feb-01	AAA	1	100
2023-Feb-01	BBB	1	100
2023-Feb-01	BBB	3	300
2023-Feb-01	AAA	1	100
Total			600

Product 2

Date	Client	Qty	Amount
2023-Feb-01	AAA	1	100
2023-Feb-01	BBB	2	200
2023-Feb-01	CCC	4	400
2023-Feb-01	DDD	1	100
Total			800

Product 3

Date	Client	Qty	Amount
2023-Feb-01	AAA	1	100
2023-Feb-01	CCC	1	100
2023-Feb-01	AAA	1	100
2023-Feb-01	DDD	1	100
Total			400

This document describes a method to classify the different elements of a spreadsheet and build a structure similar to a JSON file.

2 Reading Direction

Cognitive research shows that Human reads a document using a certain direction; depending of the culture and language of the individual. The human eye is so much trained that it is almost instinctive to look at the top left for any English reader when a page is displayed on a computer. As such, when a human creates a document, he is influenced by this reading direction because he supposes his future reader to look at the document in the same way he looks

at it. It means the flow of the various elements of the document will follow the reading direction and therefore can be linked to each other along this direction.

2.1 Definition of Reading Direction

A reading direction is defined by a tuple of a directions. Directions are defined as follows:

- Vertical direction when cells are vertically arranged. We can have respectively 2 vertical directions top (T) to bottom (B) or bottom (B) to top (T) that we notes respectively TB or BT.
- Horizontal direction when cells are horizontally arranged. We can have respectively 2 horizontal directions left (L) to right (R) or right (R) to left (L) that we notes respectively LR or RL.
- The first element of the tuple is the primary direction or line direction and the second element is the secondary direction or character direction.

Few examples:

- TB-LR defines the English direction of reading when we read lines from top to bottom and characters from left to right. This direction is sometimes called Gutenberg direction.
- TB-RL defines the Arab direction of reading when we read lines from top to bottom and characters from right to left
- RL-TB defines the traditional Chinese direction of reading when we read lines from right to left and characters top to bottom

3 Feature Extraction

A spreadsheet is composed of unstructured text and of tabular data. Tabular data is a rectangle tightly grouped cells along rows and columns.

We propose the following steps to extract the free text and the tabular data:

3.1 Spreadsheet To Bitmap

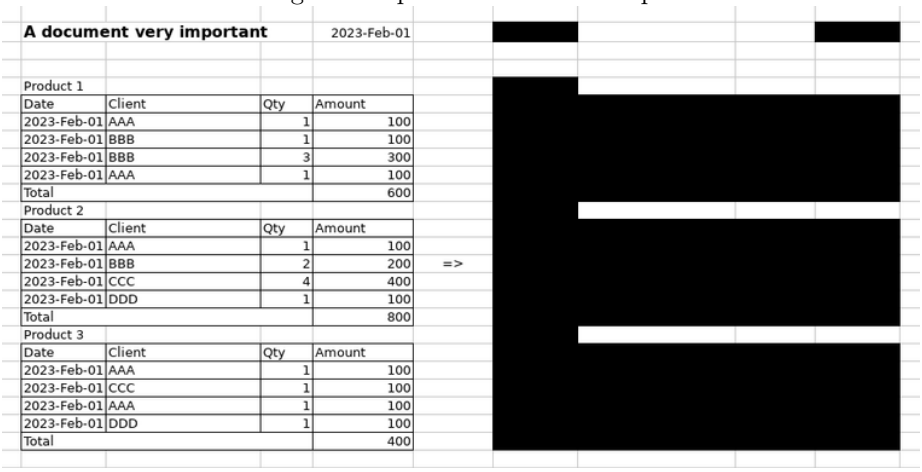
Transform the spreadsheet by an bitmap:

Listing 1: Spreadsheet To Bitmap

```
def spearsheet_to_bitmap(rows):
    for row in rows:
      for cell in row:
        if cell.ispace():
          yield 0
        else:
          yield 1
```

For example:

Figure 2: Spreadsheet To Bitmap



A document very important 2023-Feb-01

Product 1

Date	Client	Qty	Amount
2023-Feb-01	AAA	1	100
2023-Feb-01	BBB	1	100
2023-Feb-01	BBB	3	300
2023-Feb-01	AAA	1	100
Total			600

Product 2

Date	Client	Qty	Amount
2023-Feb-01	AAA	1	100
2023-Feb-01	BBB	2	200
2023-Feb-01	CCC	4	400
2023-Feb-01	DDD	1	100
Total			800

=>

Product 3

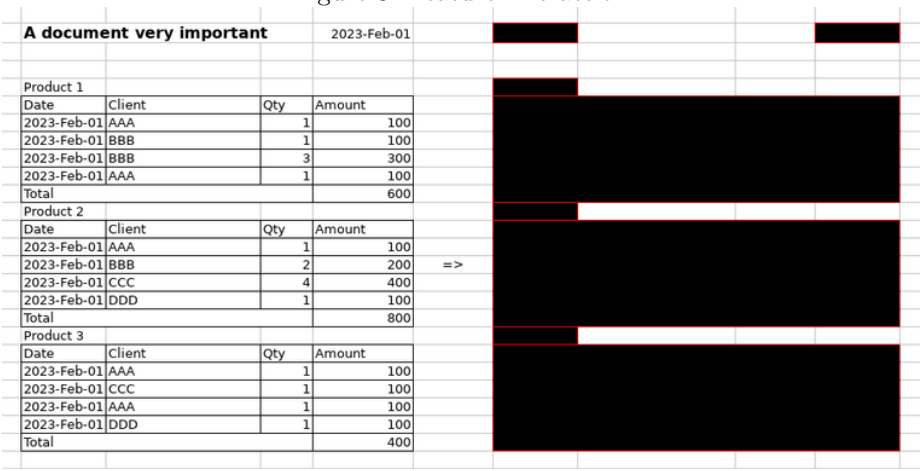
Date	Client	Qty	Amount
2023-Feb-01	AAA	1	100
2023-Feb-01	CCC	1	100
2023-Feb-01	AAA	1	100
2023-Feb-01	DDD	1	100
Total			400

3.2 Feature Extraction From Bitmap

From the bitmap, run a feature extraction such as Hough transformation or Region of Interest or even better a package like OpenCV. In our Java implementation, we use a simple Hough transformation with a convolution (size=3x3, stride=1) over the bitmap and a kernel (3x3) to detect rectangle corners.

For example:

Figure 3: Feature Extraction



A document very important 2023-Feb-01

Product 1

Date	Client	Qty	Amount
2023-Feb-01	AAA	1	100
2023-Feb-01	BBB	1	100
2023-Feb-01	BBB	3	300
2023-Feb-01	AAA	1	100
Total			600

Product 2

Date	Client	Qty	Amount
2023-Feb-01	AAA	1	100
2023-Feb-01	BBB	2	200
2023-Feb-01	CCC	4	400
2023-Feb-01	DDD	1	100
Total			800

=>

Product 3

Date	Client	Qty	Amount
2023-Feb-01	AAA	1	100
2023-Feb-01	CCC	1	100
2023-Feb-01	AAA	1	100
2023-Feb-01	DDD	1	100
Total			400

3.3 Classify Features

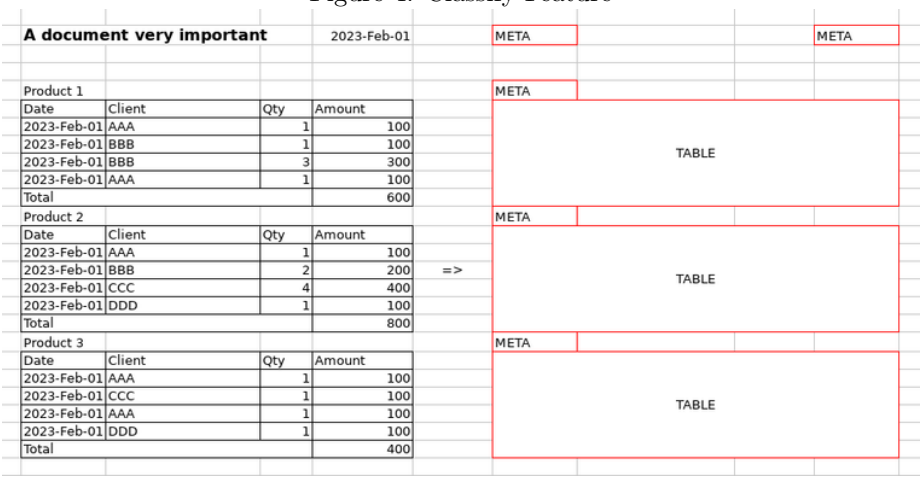
Calculate the area of each rectangle and determine if the rectangle enclose unstructured text (here after called META) or tabular data (here after called TABLE). One easy method is given a threshold, if the area is smaller than this threshold, the area contains a META else a TABLE:

Listing 2: Spreadsheet To Bitmap

```
def area(r):
  return r.rows * r.cols

def classify(RECTANGLES, THRESHOLD):
  for r in RECTANGLES:
    if area(r) < THRESHOLD:
      return "META"
    else:
      return "TABLE"
```

For example with a threshold of 1:

Figure 4: Classify Feature



3.4 Build Tree Structure

Depending of the reading direction, build a tree linking the META and TABLES found above:

Listing 3: Tree Structure

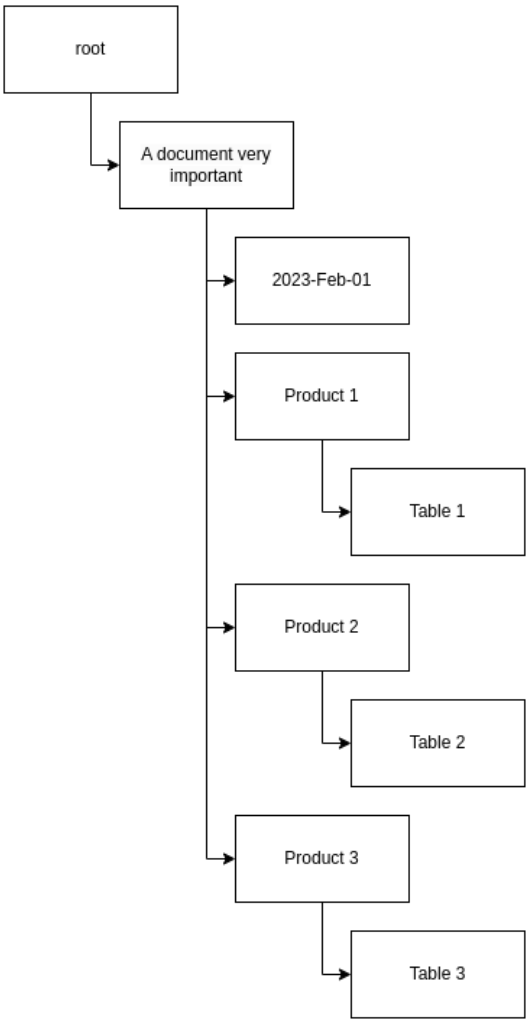
```
# Define reading direction for english
# document start on the top, left
reading_direction_start_cell = { "row": 0, "col": 0 }
```

```
# Child meta are strictly on the right and below a
# current element
reading_direction_next_meta =
        lambda x, y: x.row >= y.row and x.col > y.col

# Child table are on the right and below a current
# element
reading_direction_next_table =
        lambda x, y: x.row >= y.row and x.col >= y.col

def reading_direction_findclosestfromstart():
    closest = None
    min = 0
    for meta in METAS:
        dist = (meta.row -
                reading_direction_start_cell.row)**2
                + (meta.col -
                reading_direction_start_cell.col)**2
        if closest is None or min < dist:
            closest = meta
            min = dist
        return closest


def reading_direction_ischildof(root, elem, func):
    closest = None
    min = 0
    for node in root:
        if node != elem and func(elem, node):
            dist = (node.row - 0)**2 + (node.row - 0)**2
            if closest is None or min < dist:
            closest = elem
            min = dist
        return closest


root = reading_direction_findclosestfromstart()

# METAS are processed first and serve as anchor
# for the TABLES
for meta in METAS:
    parent = reading_direction_ischildof(
            root,
            meta,
            reading_direction_next_meta
```

```
        )
    if parent is None:
    root.append(meta)
    else:
    parent.append(meta)

# TABLES are processed in second and possibliy
# attached to METAS
for table in TABLES:
    parent = reading_direction_ischildof(
            root,
            table,
            reading_direction_next_table
        )
    if table is None:
    root.append(table)
    else:
    parent.append(table)
```

For example:

Figure 5: Tree Structure



4 From Tree Structure to JSON structure

The tree structure above can be easily converted into a JSON structure. Below the final JSON structure of the spreadsheet:

Listing 4: JSON Structure

```
[
    {
                "meta": "A document very important",
                "meta": "2023-Feb-01"
```

```
    },
    {
                "meta": "Product 1",
                "table":
                [
            {
                "Date": "2023-Feb-01",
                "Client": "AAA", "Qty": 1,
                "Amount": 100
            },
        ]
    },
    {
                "meta": "Product 2",
                "table":
                [
                        {
                "Date": "2023-Feb-01",
                "Client": "AAA", "Qty": 1,
                "Amount": 100
            },
        ]
    },
    {
                "meta": "Product 3",
                "table":
                [
                        {
                "Date": "2023-Feb-01",
                "Client": "AAA", "Qty": 1,
                "Amount": 100
            },
        ]
    }
]
```

5 From JSON structure to Tabular Output

As a complementary to this method, we give the pseudo Python code to transform any JSON structure into Tabular Outputs:

Listing 5: JSON structure to Tabular Output

```
import csv
import json
```

```
def is_list_of_list(l):
    return len(l) > 0 and all([
        isinstance(e, list)
        for e in l
    ])


def flatten_list_of_list(l):
    if all([is_list_of_list(e) for e in l]):
        return [
            x for e in l
            for x in flatten_list_of_list(e)
        ]
    else:
        return l


def json_to_tabular_rec(o, column_prefix, rows, headers):
    if isinstance(o, dict):
        for k, v in o.items():
            _, rows = json_to_tabular_rec(
                    v,
                    column_prefix + "." + k,
                    rows,
                    headers
                )
        return headers, rows
    elif isinstance(o, list):
        return headers, flatten_list_of_list(
            [
                json_to_tabular_rec(
                        x,
                        column_prefix,
                        rows,
                        headers
                    )[1]
                for x in o
            ]
        )
    else:
        headers.add(column_prefix)
        if is_list_of_list(rows):
            return headers, [
                    a + [(column_prefix, o)]
                    for a in rows
```

```
                ]
        else:
            return headers, rows + [(column_prefix, o)]


def json_to_tabular(o):
    return json_to_tabular_rec(o, "column", [], set())

def tabular_to_csv(headers, rows, fp):
    writer = csv.writer(
            fp,
            delimiter=';',
            quoting=csv.QUOTE_MINIMAL
        )
    writer.writerow(list(headers))
    for row in rows:
        records = [""] * len(headers)
        for cell in row:
            index = list(headers).index(cell[0])
            records[index] = cell[1]
        writer.writerow(records)

with open("example3.json", "r") as fp1:
    with open("example3.csv", 'w', newline='') as fp2:
        tabular_to_csv(
                *json_to_tabular(json.load(fp1)), fp2)
```

6 References

Han, S., Northoff, G. Reading direction and culture. Nat Rev Neurosci 9, 965 (2008). https://doi.org/10.1038/nrn2456-c2

7 Implementation

Java - github