Text and Structure Extraction from Medical Record Document Images

Mikayla Thompson

April 30, 2015

Abstract

PatientBank is a platform for individuals to aggregate their own health information by requesting, gathering, and maintaining a comprehensive and unified record on the individuals behalf. PatientBank receives patients’ records in paper form and the records are scanned to an image format, however the platform should eventually extract and interpret the information from the records to give the patient a new dimension of access to and understanding of their information. One of the key steps to accomplish this objective is turning the image file of the scanned document into a text document that can be parsed and analyzed. This project took shape as a Ruby gem that could be given a document image and return a valid HTML document that encodes the basic structure of the original, defined as tables (with rows and cells) and paragraphs (a catch-all for all other elements of the document). The extraction process can be divided into five individual steps: visually separating discrete items, identifying tables, splitting tables into cells, extracting text, and reformatting into HTML. On simple documents, the tool works quite well to describe the structure. The largest hurdle is in the inaccurate text extraction provided by the Tesseract OCR engine (and worsened by Tesseract’s preference for large blocks of text, rather than individual words or lines). The solution is also not robust enough to recognize more than very basic styling of tables. These issues can be addressed technically to some extent, but a perfectly robust technical solution seems unlikely. The inclusion of manual effort or verification in some steps of the process should be carefully considered.

1 Background

PatientBank is a platform for individuals to aggregate their own health information. Especially for high utilizers of health care, comprehensive health records are incredibly valuable. While electronic health records have become more prevalent, patients’ access to their own information remains limited. Additionally, while doctors and hospitals are used to collecting their patients’ records before treatment, their workflows are often very inefficient, involving long phone calls with patients to collect previous providers’ contact information and fax processes that are difficult to scale efficiently. PatientBank solves this problem for both the patient and the provider by requesting, gathering, and maintaining a comprehensive and unified record on his or her behalf.
2 Introduction

PatientBank receives patients’ records in paper form—primarily by mail and fax—so the records are scanned and uploaded to the platform for patient access. Eventually, the platform should extract and interpret the information from the records to give the patient a new dimension of access to and understanding of their information. For instance, PatientBank should be able to scrape all blood pressure measurements and give the patient a graph of their blood pressure over time.

One of the key steps to accomplish this objective is turning the image file of the scanned document into a text document that can be parsed and analyzed. The simplest component of this is optical character recognition (OCR), the process of recognizing visual pieces as textual characters. This process is not solved, but substantial progress has been made, usually through machine learning methods, and prepackaged solutions are available. This project uses the Tesseract OCR engine[1], an open source engine maintained by Google.

Beyond a direct translation from image to text, however, is the issue of maintaining the structure of the original document. Medical records are complex documents that embed much of their meaning in their structure. Section headings should be separated from the previous section. Each field of a form is an individual entity. Tables, in particular, encode much of essential information in a concise way. Reducing a table to a one dimensional list of characters irrecoverably destroys the meaning. It’s clear that the text extraction must take the documents structure into account, which was the goal of this project.

3 Methods & Results

HTML is already specifically designed to describe a document’s structure and contents in a well-defined and widely manipulatable way. It would be a major step towards full integration to be able to translate document images into an HTML approximation. As PatientBank is written as a Ruby on Rails app, this project took shape as a Ruby gem that could be given a document image and return an HTML document that encodes the basic structure of the original. Eventually, it would be great to produce a fully accurate and descriptive HTML version, but for the sake of this semester, the two elements to be identified are paragraphs and tables. Paragraphs (the \(<p>\) tag) will include traditional paragraphs, but are also a catch-all category, so all text other than tables will fall into this element (for instance: titles, headers, etc.).

The program is object oriented, with the main components being documents, paragraphs, tables, rows, and columns. Figure 1 demonstrates the main dependencies and relationships.

The extraction process can be divided into five modular steps, each of which will be described below.

3.1 Find and separate items

The first step is to separate the image into visually continuous regions, the basis of the paragraph delineations. The general idea is to find boxes in the image that contain text but whose edges don’t overlap any text. Applied in
this form, the algorithm would actually grab only characters or words, rather than paragraphs. Instead, the image is manipulated first by “bleeding” each of the pixels. Dark pixels are expanded outwards from each pixel which allows letters, for instance, to bleed together and be perceived as a continuous block as in Figure 2(a,b). On this image, the described method of finding rectangles can be applied, with the result for this image shown in 2(c).

The bleed function creates a copy of the original image, and parses through the pixels of the original. For each one that is dark colored, the pixels within the specified radius are made to be black pixels in the copy. The radius used in Figure 2 is 4 pixels, which, in informal testing, seems like a good default, but this does depend on the size of the text in pixels. The relevant portion of the code is shown below.

```ruby
bimg = img.copy
img.each_pixel do |pixel, c, r|
  if pixel.intensity < dark_thresh and
    c > radius and r > radius
    p = Pixel.from_color("#000")
  bimg.store_pixels(c-radius, r-radius, radius*2, radius*2, Array.new(4*radius*radius, p))
```

Figure 1: Code structure of gem

Figure 2: Illustration of bleeding and splitting blocks of text.
The split function simply implements the "expanding rectangle" algorithm described. Starting from the top of the bleeding image, a dark colored pixel is found. A 2x2 rectangle is formed, starting with this pixel. Each of the sides is checked for a light colored edge. If the edge is not light, that side is expanded until lightness is achieved. The algorithm rotates through the edges until all are satisfactory. The resulting rectangle should surround a block of text. The next starting point is the next dark pixel below this block.

The dimensions of each block are used to initialize a Paragraph object with the original (non bled) image.

### 3.2 Classify items (as tables or paragraphs)

In this step, Paragraph objects are tested to determine whether they qualify as a table. The key criteria to be a table is the presence of dark vertical lines. It’s very unusual to find more than a single nearly continuous vertical lines in anything other than a table, so this has shown to be an effective criteria.

This depends on a `find_verticals` function that goes column by column, counts the number of dark pixels, and keeps track of a given column if more than a `critical_mass` fraction of pixels are dark. The `is_table?` function of the paragraph simple counts the number of columns returned by `find_verticals`.

```ruby
def find_verticals
  dark_thresh = 30000
  critical_mass = 0.8
  columns = Array.new

  for x in start_col..(end_col)
    count = 0
    for y in start_row..(end_row)
      if @image.pixel_color(x,y).intensity < dark_thresh
        count += 1
      end
    end
    if 1.0*count / height > critical_mass
      columns.push(x)
    end
  end
  return columns
end
```

Paragraphs identified as tables are initialized as Table objects, separated and split into cells.

### 3.3 (For tables) Split into cells

Splitting tables into cells follows roughly the same process as finding verticals for identifying tables. Because of the way tabular HTML is formatted (table → row → cell) it makes sense to split by row first, and then column to form cells.
The row-splitting function operates by the same algorithm as `find_verticals`. The key difference is that adjacent dark rows are dismissed. Most rows delimiters are more than one pixel wide so keeping every single dark row would result in having three or four 1-pixel-tall rows for every real row.

Once rows have been separated and initialized, they’re split into cells, again with the modified `find_verticals`.

### 3.4 Extract text

By this step the entire document has been split into it’s most basic elements: paragraphs and cells, as applicable. Each of these should contain only unstructured text, which is ready to be extracted.

This uses the Tesseract OCR Engine, as described earlier. The engine is only usable as a command line utility, there is not an API available, and it must be given a file to parse. To accommodate this, the image is written to a file, the so the command line utility is called from within Ruby on that file, and then the file is deleted, as shown below. The parameter `psm` is optional and describes a page segmentation method for Tesseract if it is included (otherwise the page is segmented automatically).

```ruby
def extract(psm = false)
  name = "./tmp_img_" + rand(100000).to_s
  @img.write(name + "_.png")
  if psm
    cmd = 'tesseract #{name}.png #{name} -psm #{psm}’
  else
    cmd = 'tesseract #{name}.png #{name}’
  end
  text = File.read("#{name}.txt")
  cmd = ‘rm #{name}.*’
  return text
end
```

### 3.5 Format as HTML

Once all of the text has been extracted, components can be followed up the chain to be recongealed into HTML. Each component is called upon by its parent to form HTML, and (when applicable) recursively calls upon its own children. For paragraphs, of course, the text is simple enclosed in `<p>` tags and passed to the document. The table structure is marginally more complex: cells in `<td>` tags are nested in rows in `<tr>` tags, that are in turn nested in `<table>` tags.

Paragraphs and tables are combined in the order they were originally found, which should reliably follow the vertical alignment of the original document.

The entire output is also wrapped in `<html>` and `<body>` tags, with an empty `<head>` before the body. With these additions, it is a completely well-formed HTML document.
3.6 Combined
An example medical record and the parsed HTML are included in Appendix A. The tool correctly parsed all of the structural elements (paragraph divisions and the table, including all cells). The text, however was far less accurate.

4 Limitations and Future Work
As can be seen in Appendix A, while this technique does capture the overall structure and much of the text, it has significant issues. Additionally, there are many important structures and document styles that it is entirely unable to parse. Several of these limitations and shortcomings are described below, along with potential mitigation strategies. Before use in a production environment or with on critical data, these issues must be resolved.

4.1 Accuracy of Tesseract
Tesseract’s text extraction is far from perfectly accurate. The errors fall on a spectrum from decipherable to gibberish: "Bbse" → "Pulse", "Provid D12 lane" → "Provider: Dr. Jane", "95,Z"""" → "98.7".
In a medical context, even small errors (misreading a digit in a temperature or dosage) could have critical consequences. Explaining some of the range of quality, Tesseract tends to do better on paragraphs of text than a few characters.
Experimenting with Tesseract’s page segmentation modes can occasionally drastically improve the interpretation of a section. However, choosing too specific a mode can also make a section completely illegible. It might be possible to run Tesseract with multiple modes on each section and select the best extraction using heuristics like fewest unusual characters or most dictionary words.

4.2 Implementing a spell check
An obvious proposal for resolving some of the Tesseract issues is implementing a spell check on the extracted text. The objective of last semester’s CS 490 project was to implement a spell check with English and medical words on a text extracted from Tesseract for search indexing. [2] It was very successful in that respect, but would be an inappropriate choice for this use case. In that case, the correction options were appended after the word, but the original was not replaced. This made sense for search indexing where superfluous words were not a problem.
However, in this case, in order to be legible, incorrect words would need to be replaced. Replacing carries with it the risk of losing information, perhaps critically. For instance, the names of patients and doctors would often not be recognized, but replacing them would obviously be very erroneous.
A spell check could possibly be combined with a user interface. A user, perhaps an intake coordinator, would be able to review unrecognized words, along with the the correction options and the image of the original word and decide what the most appropriate correction would be (similar to a multiple choice version of a Captcha). Of course, this means that the system would not be fully automated and therefore slow down the process and make scaling more challenging.
4.3 Table formatting

The algorithms to identify tables and split them into cells are predicated on a very particular set of assumptions:

1. Cells are divided with straight, solid, dark-colored lines;
2. Each table is surrounded with a straight, solid, dark-colored frame;
3. Tables have no joined cells (each row has an equal number of columns and vis-à-vis);
4. There are no straight, solid, dark-colored lines within cells;
5. Non-paragraph elements of the document have no straight, solid, dark-colored vertical lines.

While these assumptions describe many tables and documents, they are not universally true. Many tables are delineated by whitespace instead; some contain graphics or sub-tables inside cells; many have dark lines inside but no frame; many join several columns near the top to form a title area. None of these variations would be correctly parsed by this tool.

It is definitely possible to build more flexibility into the algorithm (i.e. recognizing a solid white line that’s at least several pixels wide as a vertical delimiter). These would help, but there are a nearly infinite variety of potential table designs, and their characteristics will overlap significantly with non-table elements of other documents. It seems unlikely that it’s possible to effectively hardcode a complete solution.

Again, it makes sense to bring up the option of incorporating some human input. A human could scan through the document for tables that the algorithm may not recognize and, for example, create a table overlay on each. This would be extremely effective because recognizing a table is a near instantaneous task for most people. Again, however, incorporating manual effort inherently slows and limits the process.

5 Conclusion

The progress made on this project demonstrates encouraging potential in extracting the structure of medical records. In fact, in limited and simplified tests, the accuracy of the structure extraction tool exceeded the accuracy of pre-rolled text extraction tools.

There are several major hurdles to overcome before this could begin to be considered in production setting. Many of these could be largely surmounted with a small amount of human input. The trade-off between speed and accuracy would need to be carefully considered, but each task would be simple and self-contained enough to make a relatively inexpensive solution like Mechanical Turk a possibility.

References

Visit Summary

John X. Doe

Provider: Dr. Jane Physician

Vital Signs/Measurements

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>BP</td>
<td>119/82</td>
</tr>
<tr>
<td>Pulse</td>
<td>72</td>
</tr>
<tr>
<td>Pulse Ox.</td>
<td>99%</td>
</tr>
<tr>
<td>Temperature</td>
<td>98.7</td>
</tr>
<tr>
<td>Resp. Rate</td>
<td>14</td>
</tr>
</tbody>
</table>

Diagnoses this Visit:
High ankle sprain of lower extremity
Right knee injury  MCL +/- Meniscus

Medications Ordered this Visit:
naproxen (NAPROXYN) 375 MG tablet
Sig: Take 1 tablet (375 mg total) by mouth two times daily
Start: 4/9/15
Quantity: 30 tablets
Diagnoses this Visit:
- High ankle sprain of lower extremity
- Right knee injury MCL +/- Meniscus

Medications Ordered this Visit:
- naproxen (NAPROXYN) 375 MG tablet
  Sig: Take 1 tablet (375 mg total) by mouth two times daily
  Start: 14/9/15
  Quantity: 30 tablets