Denis Papathanasiou

All the Pieces Matter

« python setup.py uninstall
Spam-apalooza: A Survey of Modern Blog Comment Spam »

Extracting Text & Images from PDF Files

Update: January 29, 2012

I've corrected this code to work with the current version of pdfminer and it's now available as a github repo: https://github.com/dpapathanasiou/pdfminer-layout-scanner

PDFMiner is a pdf parsing library written in Python by Yusuke Shinyama.

In addition to the pdf2txt.py and dumppdf.py command line tools, there is a way of analyzing the content tree of each page.

Since that's *exactly* the kind of programmatic parsing I wanted to use PDFMiner for, this is a more complete example, which continues where the default documentation stops.

This example is still a work-in-progress, with room for improvement.

In the next few sections, I describe how I built up each function, resolving problems I encountered along the way. The impatient can just get the code here instead.

Basic Framework

Here are the python imports we need for PDFMiner:

```
from pdfminer.pdfparser import PDFParser, PDFDocument, PDFNoOutlines
from pdfminer.pdfinterp import PDFResourceManager, PDFPageInterpreter
from pdfminer.converter import PDFPageAggregator
from pdfminer.layout import LAParams, LTTextBox, LTTextLine, LTFigure, LTImage
```

Since PDFMiner requires a series of initializations for each pdf file, I've started with this wrapper (Lisp macro style) function to take care of the basic preliminary actions (file IO, PDFMminer object creation and connection, etc.).

```
def with_pdf (pdf_doc, pdf_pwd, fn, *args):
    """Open the pdf document, and apply the function, returning the results"""
    result = None
    try:
        # open the pdf file
        fp = open(pdf_doc, 'rb')
        # create a parser object associated with the file object
        parser = PDFParser(fp)
        # create a PDFDocument object that stores the document structure
        doc = PDFDocument()
        # connect the parser and document objects
        parser.set_document(doc)
        doc.set_parser(parser)
        # supply the password for initialization
        doc.initialize(pdf_pwd)

        if doc.is_extractable:
            # apply the function and return the result
            result = fn(doc, *args)

        # close the pdf file
        fp.close()
    except IOError:
        # the file doesn't exist or similar problem
        pass
    return result
```

The first two parameters are the name of the pdf file, and its password. The third parameter, fn, is a higher-order function which takes the instance of the pdfminer.pdfparser.PDFDocument created, and applies whatever action we want (get the table of contents, walk through the pdf page by page, etc.)

The last part of the signature, *args, is an optional list of parameters that can be passed to the high-order function as needed (I could have gone with keyword arguments here instead, but a simple list is enough for these examples).

As a warm-up, here's an example of how to use the with_pdf() function to fetch the table of contents from a pdf file:

```
def _parse_toc (doc):
    """With an open PDFDocument object, get the table of contents (toc) data
    [this is a higher-order function to be passed to with_pdf()]"""
    toc = []
    try:
        outlines = doc.get_outlines()
        for (level,title,dest,a,se) in outlines:
            toc.append( (level, title) )
    except PDFNoOutlines:
        pass
    return toc
```

The _parse_toc() function is the higher-order function which gets passed to with_pdf() as the fn parameter. It expects a single parameter, doc, which is the the instance of the pdfminer.pdfparser.PDFDocument created within with_pdf() itself (note that if with_pdf() couldn't find the file, then _parse_toc() doesn't get called).

With all the PDFMiner overhead and initialization done by with_pdf(), _parse_toc() can just focus on collecting the table of content data and returning them as a list. The get_outlines() can raise a "PDFNoOutlines" error, so I catch it as an exception, and simply return an empty list in that case.

All that's left to do is define the function that invokes _parse_toc() for a specific pdf file; this is also the function that any external users of this module would use to get the table of contents list. Note that the pdf password defaults to an empty string (which is what PDFMiner will use for documents that aren't password-protected), but that can be overriden as needed.

```
def get_toc (pdf_doc, pdf_pwd=''):
    """Return the table of contents (toc), if any, for this pdf file"""
    return with_pdf(pdf_doc, pdf_pwd, _parse_toc)
```

Page Parsing

Next, onto layout analysis. Using the with_pdf() wrapper, we can reproduce the example in the documentation with this higher-order function:

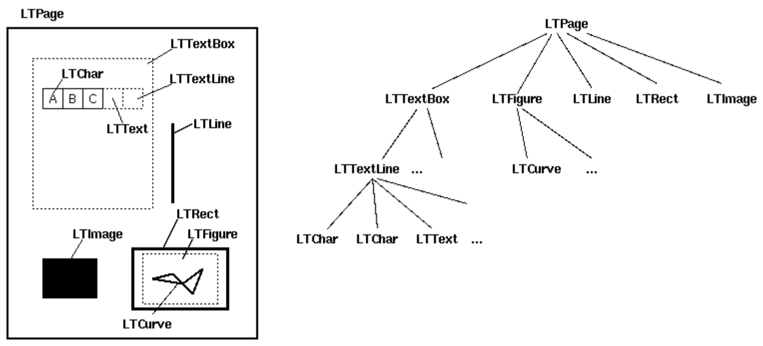
```
def _parse_pages (doc):
    """With an open PDFDocument object, get the pages and parse each one
    [this is a higher-order function to be passed to with_pdf()]"""
    rsrcmgr = PDFResourceManager()
    laparams = LAParams()
    device = PDFPageAggregator(rsrcmgr, laparams=laparams)
    interpreter = PDFPageInterpreter(rsrcmgr, device)

    for page in doc.get_pages():
        interpreter.process_page(page)
        # receive the LTPage object for this page
        layout = device.get_result()
        # layout is an LTPage object which may contain child objects like LTTextBox, LTFigure, LTImage, etc.
```

And this external function, which defines the specific pdf file to analyze:

```
def get_pages (pdf_doc, pdf_pwd=''):
    """Process each of the pages in this pdf file"""
    with_pdf(pdf_doc, pdf_pwd, _parse_pages)
```

So far, this code doesn't do anything exciting: it just loads each page into a pdfminer.layout.LTPage object, closes the pdf file, and exits.

Within each pdfminer.layout.LTPage instance, though, is an objs attribute, which defines the tree of pdfminer.layout.LT* child objects as in the documentation:



In this example, I'm going to collect all the text from each page in a top-down, left-to-right sequence, merging any multiple columns into a single stream of consecutive text.

The results are not always perfect, but I'm using a fuzzy logic based on physical position and column width, which is very good in most cases.

I'm also going to save any images found to a separate folder, and mark their position in the text with <img /> tags.

Right now, I'm only able to extract jpeg images, whereas xpdf's pdfimages tool is capable of getting to non-jpeg images and saving them as ppm format.

I'm not sure if the problem is within PDFMiner or how I'm using it, but since someone else asked the same question in the PDFMiner mailing list, I suspect it's the former.

This requires a few updates to the _parse_pages() function, as follows:

```
def _parse_pages (doc, images_folder):
    """With an open PDFDocument object, get the pages, parse each one, and return the entire text
    [this is a higher-order function to be passed to with_pdf()]"""
    rsrcmgr = PDFResourceManager()
    laparams = LAParams()
    device = PDFPageAggregator(rsrcmgr, laparams=laparams)
    interpreter = PDFPageInterpreter(rsrcmgr, device)

    text_content = [] # a list of strings, each representing text collected from each page of the doc
    for i, page in enumerate(doc.get_pages()):
        interpreter.process_page(page)
        # receive the LTPage object for this page
        layout = device.get_result()
        # layout is an LTPage object which may contain child objects like LTTextBox, LTFigure, LTImage, etc.
        text_content.append(parse_lt_objs(layout.objs, (i+1), images_folder))

    return text_content
```

and the updated get_pages() function becomes:

```
def get_pages (pdf_doc, pdf_pwd='', images_folder='/tmp'):
    """Process each of the pages in this pdf file and print the entire text to stdout"""
    print '\n\n'.join(with_pdf(pdf_doc, pdf_pwd, _parse_pages, *tuple([images_folder])))
```

New in both functional signatures is images_folder, which is a parameter that refers to the place on the local filesystem where any extracted images will be be saved (this is also an example of why defining with_pdf() with an optional *args list comes in handy).

Aggregating Text

Within the _parse_pages() function, text_content is a new variable of type list, which collects the text of each page, and I've added an enumeration structure around doc.get_pages(), to keep track of which page we're accessing at any given time. This is useful for saving images correctly, since some pdf files use the same image name in multiple places to refer to different images (this creates problems for dumppdf.py's -i switch, for example).

The new critical line in _parse_pages() is this one:

```
text_content.append(parse_lt_objs(layout.objs, (i+1), images_folder))
```

Since the tree of page objects is recursive in nature (e.g., a pdfminer.layout.LTFigure object may have multiple child objects), it's better to handle the actual text parsing and image collection in a separate function. That function, parse_lt_objs(), looks like this:

```
def parse_lt_objs (lt_objs, page_number, images_folder, text=[]):
    """Iterate through the list of LT* objects and capture the text or image data contained in each"""
    text_content = []

    for lt_obj in lt_objs:
        if isinstance(lt_obj, LTTextBox) or isinstance(lt_obj, LTTextLine):
            # text
            text_content.append(lt_obj.get_text())
        elif isinstance(lt_obj, LTImage):
            # an image, so save it to the designated folder, and note it's place in the text
            saved_file = save_image(lt_obj, page_number, images_folder)
            if saved_file:
                # use html style <img /> tag to mark the position of the image within the text
                text_content.append('<img src="'+os.path.join(images_folder, saved_file)+'" />')
            else:
                print >> sys.stderr, "Error saving image on page", page_number, lt_obj.__repr__
        elif isinstance(lt_obj, LTFigure):
            # LTFigure objects are containers for other LT* objects, so recurse through the children
            text_content.append(parse_lt_objs(lt_obj.objs, page_number, images_folder, text_content))

    return '\n'.join(text_content)
```

In this example, I'm concerned with just four objects which may appear within a pdfminer.layout.LTPage object:

1. LTTextBox and LLTextLine (which, because the text extraction is exactly the same, I treat as one case)
2. LTImage (which we'll try to save on to the local filesystem in the designated folder)
3. LTFigure (which we'll treat as a simple container for other objects, hence the recursive call in that case)

For the simple text and image extraction I'm doing here, this is enough. There is room for improvement, though, since I'm ignoring several types of pdfminer.layout.LT* objects which do appear in pdf pages.

If you try to run get_pages() now, you might get this error, in the text_content.append(lt_obj.get_text()) line (it will depend on the content of the pdf file you're trying to parse, as well as how your instance of Python is configured, and whether or not you installed PDFMiner with cmap for CJK languages).

```
UnicodeEncodeError: 'ascii' codec can't encode character u'\u2014' in position 61: ordinal not in range(128)
```

As Eliot explains, "This error occurs when you pass a Unicode string containing non-English characters (Unicode characters beyond 128) to something that expects an ASCII bytestring. The default encoding for a Python bytestring is ASCII."

This function, which I wrote after reading this article, solves the problem:

```
def to_bytestring (s, enc='utf-8'):
    """Convert the given unicode string to a bytestring, using the standard encoding,
    unless it's already a bytestring"""
    if s:
        if isinstance(s, str):
            return s
        else:
            return s.encode(enc)
```

So the updated version of parse_lt_objs() becomes:

```
def parse_lt_objs (lt_objs, page_number, images_folder, text=[]):
    """Iterate through the list of LT* objects and capture the text or image data contained in each"""
    text_content = []

    for lt_obj in lt_objs:
        if isinstance(lt_obj, LTTextBox) or isinstance(lt_obj, LTTextLine):
            # text
            text_content.append(lt_obj.get_text())
        elif isinstance(lt_obj, LTImage):
            # an image, so save it to the designated folder, and note it's place in the text
            saved_file = save_image(lt_obj, page_number, images_folder)
            if saved_file:
                # use html style <img /> tag to mark the position of the image within the text
                text_content.append('<img src="'+os.path.join(images_folder, saved_file)+'" />')
            else:
                print >> sys.stderr, "Error saving image on page", page_number, lt_obj.__repr__
        elif isinstance(lt_obj, LTFigure):
            # LTFigure objects are containers for other LT* objects, so recurse through the children
            text_content.append(parse_lt_objs(lt_obj.objs, page_number, images_folder, text_content))

    return '\n'.join(text_content)
```

Running this version gives reasonable results on pdf files where the text is single-column, and without many sidebars, abstracts, summary quotes, or other fancy typesetting layouts.

It really breaks down, though, in the case of multi-column pages: the resulting text_content jumps from one paragraph to the next, in no coherent order.

PDFMiner does provide two grouping functions, group_textbox_lr_tb and group_textbox_tb_rl [lr=left-to-right, tb=top-to-bottom], but they do the grouping literally, without considering the likelihood that the content of one textbox logically belongs after another's.

Fortunately, though, each object also provides a bbox (bounding box) attribute, which is a four-part tuple of the object's page position: (x0, y0, x1, y1).

Using the bbox data, we can group the text according to its position and width, making it more likely the columns we join together this way represent the correct logical flow of the text.

To aggregate the text this way, I added the following Python dictionary variable to the parse_lt_objs() code, just before iterating through the list of lt_objs: page_text={}.

The key for each entry is a tuple of the bbox's (x0, x1) points, and the corresponding value is a list of text strings found within that bbox. The x0 value tells me the left offset for a given piece of text and the x1 value tells me how wide it is.

So by grouping text which starts at the same horizontal plane and has the same width, I can aggregate all paragraphs belonging to the same column, regardless of their vertical position or length.

Conceptually, each entry in the page_text dictionary represents all the text associated with each physical column.

When I tried this the first time, I was surprised (though in retrospect, I shouldn't have been, since nothing about parsing pdfs is neat or clean), that two textboxes which look perfectly aligned visually have slightly different x0 and x1 values (at least according to PDFMiner).

For example, one paragraph may have x0 and x1 values of 28.16 and 153.32 respectively, and the paragraph right underneath it had an x0 value of 29.04 and an x1 value of 152.09.

To get around this, I wrote the following update function, which assigns key tuples based on how close an (x0, x1) pair lies within an existing entry's key. The 20 percent value was arrived at by trial-and-error, and seems to be acceptable for most pdf files I tried.

```
def update_page_text_hash (h, lt_obj, pct=0.2):
    """Use the bbox x0,x1 values within pct% to produce lists of associated text within the hash"""
    x0 = lt_obj.bbox[0]
    x1 = lt_obj.bbox[2]
    key_found = False
    for k, v in h.items():
        hash_x0 = k[0]
        if x0 >= (hash_x0 * (1.0-pct)) and (hash_x0 * (1.0+pct)) >= x0:
            hash_x1 = k[1]
            if x1 >= (hash_x1 * (1.0-pct)) and (hash_x1 * (1.0+pct)) >= x1:
                # the text inside this LT* object was positioned at the same
                # width as a prior series of text, so it belongs together
                key_found = True
                v.append(to_bytestring(lt_obj.get_text()))
                h[k] = v
    if not key_found:
        # the text, based on width, is a new series,
        # so it gets its own series (entry in the hash)
        h[(x0,x1)] = [to_bytestring(lt_obj.get_text())]
    return h
```

With this in place, I could update the parse_lt_objs() to use it.

```
def parse_lt_objs (lt_objs, page_number, images_folder, text=[]):
    """Iterate through the list of LT* objects and capture the text or image data contained in each"""
    text_content = []

    page_text = {} # k=(x0, x1) of the bbox, v=list of text strings within that bbox width (physical column)
    for lt_obj in lt_objs:
        if isinstance(lt_obj, LTTextBox) or isinstance(lt_obj, LTTextLine):
            # text, so arrange is logically based on its column width
            page_text = update_page_text_hash(page_text, lt_obj)
        elif isinstance(lt_obj, LTImage):
            # an image, so save it to the designated folder, and note it's place in the text
            saved_file = save_image(lt_obj, page_number, images_folder)
            if saved_file:
                # use html style <img /> tag to mark the position of the image within the text
                text_content.append('<img src="'+os.path.join(images_folder, saved_file)+'" />')
            else:
                print >> sys.stderr, "error saving image on page", page_number, lt_obj.__repr__
        elif isinstance(lt_obj, LTFigure):
            # LTFigure objects are containers for other LT* objects, so recurse through the children
            text_content.append(parse_lt_objs(lt_obj.objs, page_number, images_folder, text_content))

    for k, v in sorted([(key,value) for (key,value) in page_text.items()]):
        # sort the page_text hash by the keys (x0,x1 values of the bbox),
        # which produces a top-down, left-to-right sequence of related columns
        text_content.append('\n'.join(v))

    return '\n'.join(text_content)
```

The last block before the return statement sorts the page_text (x0, x1) keys so that the resulting text is returned in a top-down, left-to-right sequence, based on where the text appeared visually on the page.

Extracting Images

The last thing to discuss in this example is the extraction of images.

As I mentioned above, this area needs improvement, since it seems that I can only extract jpeg images using PDFMiner (though to be fair to Yusuke, he does describe it as a tool that "*focuses entirely on getting and analyzing text data*", so perhaps doing more than jpeg is out-of-scope for this library).

Within parse_lt_objs(), the following function is called if an LTImage is found; it was based on studying the dumppdf.py source code and how it handled image extraction requests:

```
def save_image (lt_image, page_number, images_folder):
    """Try to save the image data from this LTImage object, and return the file name, if successful"""
    result = None
    if lt_image.stream:
        file_stream = lt_image.stream.get_rawdata()
        file_ext = determine_image_type(file_stream[0:4])
        if file_ext:
            file_name = ''.join([str(page_number), '_', lt_image.name, file_ext])
            if write_file(images_folder, file_name, lt_image.stream.get_rawdata(), flags='wb'):
                result = file_name
    return result
```

The save_image() function needs the following two supporting functions defined:

```
def determine_image_type (stream_first_4_bytes):
    """Find out the image file type based on the magic number comparison of the first 4 (or 2) bytes"""
    file_type = None
    bytes_as_hex = b2a_hex(stream_first_4_bytes)
    if bytes_as_hex.startswith('ffd8'):
        file_type = '.jpeg'
    elif bytes_as_hex == '89504e47':
        file_type = ',png'
    elif bytes_as_hex == '47494638':
        file_type = '.gif'
    elif bytes_as_hex.startswith('424d'):
        file_type = '.bmp'
    return file_type
```

The determine_image_type() function is based on the concept of magic numbers, where it's (sometimes) possible to tell what a binary stream means by exmaing the first two or fours bytes.

In theory, a pdf file can have any of these image types, but in practice, the only one PDFMiner can seem to find as an LTImage object are jpegs.

```
def write_file (folder, filename, filedata, flags='w'):
    """Write the file data to the folder and filename combination
    (flags: 'w' for write text, 'wb' for write binary, use 'a' instead of 'w' for append)"""
    result = False
    if os.path.isdir(folder):
        try:
            file_obj = open(os.path.join(folder, filename), flags)
            file_obj.write(filedata)
            file_obj.close()
            result = True
        except IOError:
            pass
    return result
```

The write_file() function is just basic file IO, but it does some convenient things around checking that the designated folder exists, too.

Finally, to support all three image saving functions, we need the following python imports:

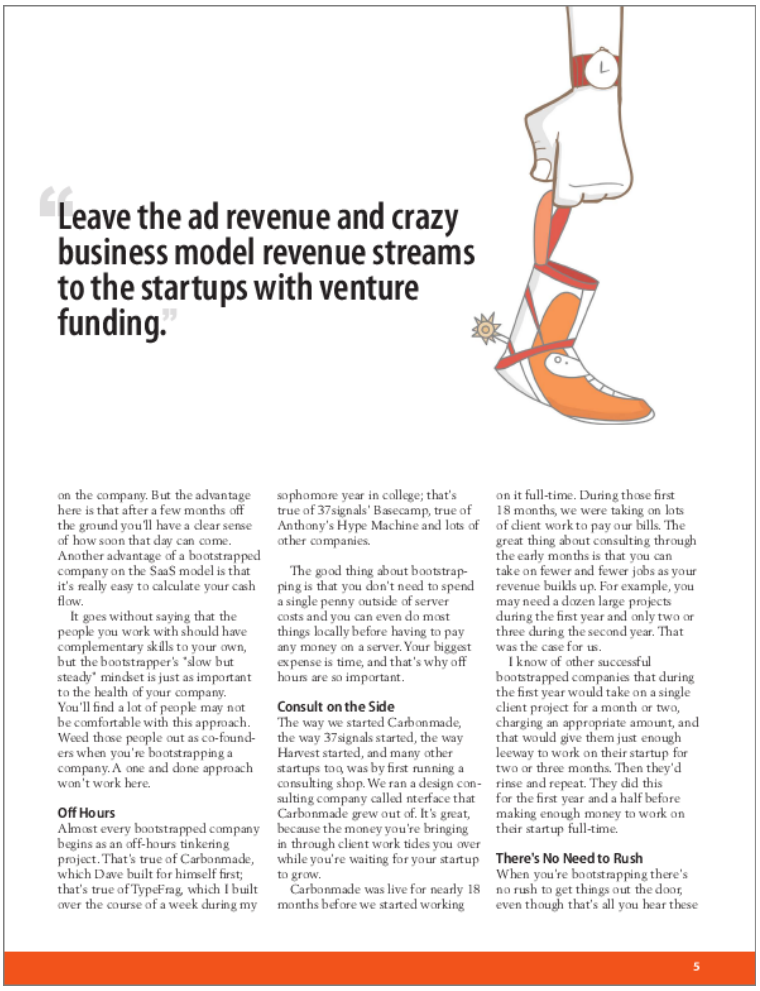
```
import sys
import os
from binascii import b2a_hex
```

Sample Results

So, how well does it work? It's surprisingly good, as it turns out.

Here's an example from using the above code to process the Hacker Monthly Issue 2 pdf file (this was part of the process I used to convert this file to e-book format for inclusion in the Fifobooks Catalog).

Page 5, which looks like this visually:



"Leave the ad revenue and crazy business model revenue streams to the startups with venture funding."

on the company. But the advantage here is that after a few months off the ground you'll have a clear sense of how soon that day can come. Another advantage of a bootstrapped company on the SaaS model is that it's really easy to calculate your cash flow.

It goes without saying that the people you work with should have complementary skills to your own, but the bootstrapper's "slow but steady" mindset is just as important to the health of your company. You'll find a lot of people may not be comfortable with this approach. Weed those people out as co-founders when you're bootstrapping a company. A one and done approach won't work here.

Off Hours

Almost every bootstrapped company begins as an off-hours tinkering project. That's true of Carbonmade, which Dave built for himself first; that's true of TypeFrag, which I built over the course of a week during my sophomore year in college; that's true of 37signals' Basecamp, true of Anthony's Hype Machine and lots of other companies.

The good thing about bootstrapping is that you don't need to spend a single penny outside of server costs and you can even do most things locally before having to pay any money on a server. Your biggest expense is time, and that's why off hours are so important.

Consult on the Side

The way we started Carbonmade, the way 37signals started, the way Harvest started, and many other startups too, was by first running a consulting shop. We ran a design consulting company called nterface that Carbonmade grew out of. It's great, because the money you're bringing in through client work tides you over while you're waiting for your startup to grow.

Carbonmade was live for nearly 18 months before we started working on it full-time. During those first 18 months, we were taking on lots of client work to pay our bills. The great thing about consulting through the early months is that you can take on fewer and fewer jobs as your revenue builds up. For example, you may need a dozen large projects during the first year and only two or three during the second year. That was the case for us.

I know of other successful bootstrapped companies that during the first year would take on a single client project for a month or two, charging an appropriate amount, and that would give them just enough leeway to work on their startup for two or three months. Then they'd rinse and repeat. They did this for the first year and a half before making enough money to work on their startup full-time.

There's No Need to Rush

When you're bootstrapping there's no rush to get things out the door, even though that's all you hear these

5

came out like this:

```
<img src="/tmp/5_Im0.jpeg" />
"Leave the ad revenue and crazy
business model revenue streams
to the startups with venture
funding."

on the company. But the advantage
here is that after a few months off
the ground you'll have a clear sense
of how soon that day can come.
Another advantage of a bootstrapped
company on the SaaS model is that
it's really easy to calculate your cash
flow.
It goes without saying that the
people you work with should have
complementary skills to your own,
but the bootstrapper's "slow but
steady" mindset is just as important
to the health of your company.
you'll find a lot of people may not
be comfortable with this approach.
Weed those people out as co-found-
ers when you're bootstrapping a
company. A one and done approach
won't work here.

off Hours
Almost every bootstrapped company
begins as an off-hours tinkering
project. That's true of Carbonmade,
which Dave built for himself first;
that's true of TypeFrag, which I built
over the course of a week during my

sophomore year in college; that's
true of 37signals' Basecamp, true of
Anthony's Hype Machine and lots of
```

```
other companies.

The good thing about bootstrap-
ping is that you don't need to spend
a single penny outside of server
costs and you can even do most
things locally before having to pay
any money on a server. your biggest
expense is time, and that's why off
hours are so important.

Consult on the Side
The way we started Carbonmade,
the way 37signals started, the way
Harvest started, and many other
startups too, was by first running a
consulting shop. We ran a design con-
sulting company called nterface that
Carbonmade grew out of. It's great,
because the money you're bringing
in through client work tides you over
while you're waiting for your startup
to grow.
Carbonmade was live for nearly 18
months before we started working

on it full-time. During those first
18 months, we were taking on lots
of client work to pay our bills. The
great thing about consulting through
the early months is that you can
take on fewer and fewer jobs as your
revenue builds up. For example, you
may need a dozen large projects
during the first year and only two or
three during the second year. That
was the case for us.
I know of other successful
bootstrapped companies that during
the first year would take on a single
client project for a month or two,
charging an appropriate amount, and
that would give them just enough
leeway to work on their startup for
two or three months. Then they'd
rinse and repeat. They did this
for the first year and a half before
making enough money to work on
their startup full-time.

there's no need to Rush
When you're bootstrapping there's
no rush to get things out the door,
even though that's all you hear these

  5
```

While there were some small problems around capitalization and spacing, the conversion did recognize and save the background image, it distinguished the summary quote as being separate from the rest of the text, and the columns were merged correctly, flowing in the same manner the author wrote them.

Room for Improvement

There are several things I'd like to be able to do better; some probably require changes to PDFMiner itself, while others are things in my code which I should improve.

- Column Merging — while the fuzzy heuristic I described works well for the pdf files I've parsed so far, I can imagine more complex documents where it would break-down (perhaps this is where the analysis should be more sophisticated, and not ignore so many types of pdfminer.layout.LT* objects).
- Image Extraction — I'd like to be able to be *at least* as good as pdftoimages, and save every file in ppm or pnm default format, but I'm not sure what I could be doing differently
- Title and Heading Capitalization — this seems to be an issue with PDFMiner, since I get similar results in using the command line tools, but it is annoying to have to go back and fix all the mis-

capitalizations manually, particularly for larger documents.

- Title and Heading Fonts and Spacing — a related issue, though probably something in my own code, is that those same title and paragraph headings aren't distinguished from the rest of the text. In many cases, I have to go back and add vertical spacing and font attributes for those manually.
- Page Number Removal — originally, I thought I could just use a regex for an all-numeric value on a single physical line, but each document does page numbering slightly differently, and it's very difficult to get rid of these without manually proofreading each page.
- Footnotes — handling these where the note and the reference both appear on the same page is hard enough, but doing it when they span different (even consecutive) pages is worse.

Tags: HOW-TO, PDF, PDFMiner, Text and Image Extraction

This entry was posted on Wednesday, August 4th, 2010 at 8:11 pm and is filed under Programming, Technology. You can follow any responses to this entry through the RSS 2.0 feed. You can leave a response, or trackback from your own site.

18 Responses to “Extracting Text & Images from PDF Files”

1. *Tweets that mention Denis Papathanasiou» Blog Archive » Extracting Text & Images from PDF Files -- Topsy.com* Says:
   August 5th, 2010 at 12:44 am

   […] This post was mentioned on Twitter by 김태성, Proggit Articles. Proggit Articles said: Extracting Text & Images from PDF Files: submitted by dpapathanasiou [link] [comment] http://trim.li/nk/35va […]

2. *Bob* Says:
   August 27th, 2010 at 8:50 am

   Hi Denis

   Thank you for a very useful and enlightening blog.

   I personally like pythons “with” and “yield”-statements better than lisp macro style, so I am trying to use something like the following instead:

```
from pdfminer.pdfparser import PDFParser, PDFDocument, PDFNoOutlines
from pdfminer.pdfinterp import PDFResourceManager, PDFPageInterpreter
from pdfminer.converter import PDFPageAggregator
from pdfminer.layout import LAParams, LTTextBox, LTTextLine, LTFigure, LTImage

def parse_pages(doc):
rsrcmgr = PDFResourceManager()
laparams = LAParams()
device = PDFPageAggregator(rsrcmgr, laparams=laparams)
interpreter = PDFPageInterpreter(rsrcmgr, device)

for page in doc.get_pages():
interpreter.process_page(page)
# receive the LTPage object for this page
layout = device.get_result()
# layout is an LTPage object which may contain child objects like LTTextBox, LTFigure, LTImage, etc.
yield layout

class PdfMinerWrapper(object):
"""
Usage:
with PdfMinerWrapper(‘2009t.pdf’) as doc:
for page in doc.get_pages():
```

```
.....
"""
def __init__(self, pdf_doc, pdf_pwd=''):
self.pdf_doc = pdf_doc
self.pdf_pwd = pdf_pwd
def __enter__(self):
#open the pdf file
self.fp = open(self.pdf_doc, 'rb')
# create a parser object associated with the file object
parser = PDFParser(self.fp)
# create a PDFDocument object that stores the document structure
doc = PDFDocument()
# connect the parser and document objects
parser.set_document(doc)
doc.set_parser(parser)
# supply the password for initialization
doc.initialize(self.pdf_pwd)
return doc
def __exit__(self, type, value, traceback):
self.fp.close()

def main():
with PdfMinerWrapper('my.pdf') as doc:
for page in parse_pages(doc):
for obj in page.objs:
print obj

if __name__ =='__main__':
main()
```

3. *Kevin Brubeck Unhammer* Says:
   December 27th, 2010 at 10:18 am

   I think layout objs changed from a public member to a function “get_objs()”, so it should be

   text_content.append(parse_lt_objs(layout.get_objs(), (i+1), images_folder))

   and

   text_content.append(parse_lt_objs(lt_obj.get_objs(), page_number, images_folder, text_content))

4. *Kevin Brubeck Unhammer* Says:
   December 27th, 2010 at 10:18 am

   By the way, is there a way to extract the background color of textboxes?

5. *Kevin Brubeck Unhammer* Says:
   December 27th, 2010 at 12:18 pm

   Doh! The API changed again. Now layout itself is an iterator, so just send that. Also, “get_text()” changed to “text”

6. *Denis* Says:
   December 30th, 2010 at 10:19 am

   Kevin,

   Thank you for noticing the changes to the API in the latest version; I've updated the script accordingly.

As for background color, I don't think this is supported yet. I asked Yusuke a similar question about determining font color, and this is how he replied:

"There's no color attribute for now, though. The color system in PDF is very complex and that scared me out. (They support several different coloring systems including RGB or CMYK.)"

7. *Erik* Says:
[January 18th, 2011 at 11:31 am]

I'm trying to run this on a pdf file of mine, but I'm not exactly sure on the syntax for the command. Can you briefly go over how to do that? I'm not sure which function I'm after to get it going.

Thanks.

8. *Denis* Says:
[January 18th, 2011 at 1:49 pm]

Since I didn't write a main() function, you have to import pdf_miner_layout_scanner inside a Python shell.

From there, call get_toc() for a list of the table of contents, and/or get_pages() for the full text:

```
>>> import pdf_miner_layout_scanner
>>> pdf_miner_layout_scanner.get_toc('/path/to/document.pdf')
[(1, 'Chapter One'), (1, 'Chapter Two'), (1, 'Chapter Three') … ]
>>> pdf_miner_layout_scanner.get_pages('/path/to/document.pdf')
This is the text of document.pdf …
…
```

The output of get_pages() goes to standard output, but if you change the print statement in line 202 to return, then you can capture the full text in a single variable:

```
>>> import pdf_miner_layout_scanner
>>> full_text = pdf_miner_layout_scanner.get_pages('/path/to/document.pdf')
>>> full_text
This is the text of document.pdf …
…
```

Similarly, if you remove the join statement in line 202, then you'll get a list of strings, each of which represents all the the text found on each page:

```
>>> import pdf_miner_layout_scanner
>>> all_pages = pdf_miner_layout_scanner.get_pages('/path/to/document.pdf')
>>> all_pages[2]
This is all the text from page 3 of document.pdf …
…
```

You can go further with this idea: e.g., remove the join statement from line 175, the last line of the parse_lt_objs() function, and instead of getting each page as a single text string, you can see the list of individual text objects which make up the page.

9. *Erik* Says:
[January 18th, 2011 at 2:37 pm]

thanks so much! i'm playing around with it now

10. *Ted* Says:
January 26th, 2011 at 7:43 pm

    Do you have any sense how to extract tables from PDFs? Many medical papers contain clinical data. It would be great to be able to extract these tables into some structured form.

11. *Denis* Says:
February 1st, 2011 at 9:23 am

    The short answer is: use the bound box info from each LT* object to determine its position, then reconstruct the table accordingly.

    I'm actually planning an update of this post, to include some html transforms which I've been experimenting with recently.

12. *Tim* Says:
March 15th, 2011 at 3:03 am

    While inspecting the result and comparing with the original pdf in Acrobat, I noticed the [Save As Text] menu item. It produces a text file with all of the text and the tabulated data is not a long list.

    So the question is, why all the effort to do this when the Acrobat Reader does a better job?

13. *Denis* Says:
March 18th, 2011 at 9:11 am

    It depends what you want to do with the document text.

    PDFMiner can give you the font type, font size and physical position (among other things) of every character on the page.

    If you want to convert the pdf file into a format such as html or xml, the 'save as text' function is not enough.

14. *Jan Palencar* Says:
May 1st, 2011 at 9:35 am

    Great post!
    I was a moment before adding sorting of LT* elements by their coordinates.
    And just bumped to your blog.
    Thank you very much. Very thorough and self-explanatory.

15. *Andreas Thienemann* Says:
October 19th, 2011 at 7:35 am

    Looks like the 20110515 release uses get_text() again.

    http://www.unixuser.org/~euske/python/pdfminer/programming.html

16. *如何自動化測試 PDF 報表的內容 « 在電梯裡遇見雙胞胎* Says:
December 6th, 2011 at 12:48 am

    […] Denis Papathanasiou » Blog Archive » Extracting Text & Images from PDF Files (2010-10-28) […]

17. *Antonin Duroy* Says:
June 7th, 2012 at 4:42 am

First of all, I’d like to thank you for your great job.
Second, when I tried to use your code above, I noticed that your “parse_lt_objs” function didn’t “produce a top-down, left-to-right sequence of related columns” as you mentioned in your comments.

I’m actually working on research papers (PDF format) and the result wasn’t the expected one.

```
+———–+ (x1, y1)
| |
| |
| |
| |
+———–+
x0 /\
|| y0
||
```

Thus I modified the code to become:

```
####################################
# I added y0 and y1 to check whether or not an LT* object is positioned at
# the same height (+- 20%)
####################################
def update_page_text_hash (h, lt_obj, pct=0.2):
x0 = np.floor(lt_obj.bbox[0])
y0 = np.floor(lt_obj.bbox[1])
x1 = np.floor(lt_obj.bbox[2])
y1 = np.floor(lt_obj.bbox[3])

…
if x1 >= (hash_x1 * (1.0-pct)) and (hash_x1 * (1.0+pct)) >= x1:
# the text inside this LT* object was positioned at the same
# width as a prior series of text, so it belongs together
hash_y1 = np.floor(k[3])
if y0 >= (hash_y1*(1.0-pct)) and (hash_y1*(1.0+pct)) >= y0:
key_found = True
v.append(to_bytestring(lt_obj.get_text()))
h[k] = v
if not key_found:
h[(x0,x1, y0, y1)] = [to_bytestring(lt_obj.get_text())]

return h

####################################
# I sorted the y1 coordinate first, then the x0 one.
# Denis’s code sorted (x0,x1)
####################################
def parse_lt_objs (lt_objs, page_number, images_folder, text=[]):
…
s = sorted([(key,value) for (key,value) in page_text.items()], key=lambda x: x[0][3], reverse=True)
for k, v in sorted(s, key=lambda x: x[0][0]):
text_content.append(”.join(v))

return ‘\n’.join(text_content)
```

Do you agree with that modification ?
Best regards

18. *Denis* Says:

June 9th, 2012 at 7:55 pm

You're right in that you'll get different results with different texts (for example, the logic I described falls apart in delaing with two column or three column layout pages).

I have been experimenting with LTChar font properties (type and size), using those in combination with bounding box information sometimes produces better results (for example, when a quote is highlighted in a larger, bolder text and inserted visually in the middle of a paragraph, that technique can tell it belongs separate from the rest of the paragraph text).

It's not stable enough to post as an update, but I may add it to the github repo eventually, if I can find time to work on it.

In the meantime, I'd enourage you to fork my repo and apply your contribution, as it would help others who are also dealing with academic papers.

Leave a Reply

Name (required)

Mail (will not be published) (required)

Website

Submit Comment

- Donate

  Please donate if you find the information on this site helpful (you can also contribute by clicking on the ads, below)

- AdChoices

  Asian Women for Dating

  Browse 10k Members No Time Limit. Free Expose Your Profile.Join Now!
  www.asiamatcho…

- **Archives**

- July 2012
- June 2012
- May 2012
- March 2012
- January 2012
- December 2011
- November 2011
- September 2011
- August 2011
- June 2011
- May 2011
- April 2011
- March 2011
- January 2011
- December 2010
- October 2010
- September 2010
- August 2010
- July 2010
- June 2010
- May 2010
- April 2010
- March 2010
- December 2009
- November 2009
- October 2009
- September 2009
- August 2009
- July 2009
- June 2009
- May 2009
- April 2009
- February 2009
- November 2008
- July 2008
- June 2008

Categories

- Aikido
- Career
- Funny
- Japan
- Philosophy
- Politics
- Programming
- Random Nonsense
- Technology

- Live Traffic Feed

See your visitors in RealTime! Get the

Free Live Traffic Feed
Get Feedjit Now!

A visitor from Singapore viewed "Denis Papathanasiou » Blog Archive » Extracting Text & Images from PDF Files" 0 secs ago

A visitor from San Francisco, California viewed "Denis Papathanasiou » Programming" 3 mins ago

A visitor from Noisy-le-grand, Ile-de-France viewed "Denis Papathanasiou » Blog Archive » Illustrated ebooks with epub: using the svg image element" 7 mins ago

A visitor from Madrid viewed "Denis Papathanasiou » Blog Archive » Node.js and MongoDB: A Simple Example" 19 mins ago

A visitor from Szeged viewed "Denis Papathanasiou » Blog Archive » Illustrated ebooks with epub: using the svg image element" 1 hour 12 mins ago

A visitor from Köln, Nordrhein-Westfalen left "Denis Papathanasiou » Blog Archive » Node.js and MongoDB: A Simple Example" via nodebeginner.org 1 hour 27 mins ago

A visitor from Köln, Nordrhein-Westfalen left "Denis Papathanasiou » Blog Archive » Node.js and MongoDB: A Simple Example" via nodebeginner.org 2 hours 9 mins ago

A visitor from Köln, Nordrhein-Westfalen viewed "Denis

-