Digital Legacies on Paper: Reading Punchcards with Computer Vision

Gregory Jansen
College of Information Studies
University of Maryland
College Park, USA
jansen@umd.edu

***Abstract*—We describe the development of a computer vision-based workflow for normalizing images of the legacy punchcard data format (IBM 029 - 80 column punchcard standard) and then reading the encoded data. We show the role of a newly developed Punchcard Extractor Tool within the Brown Dog service API. We also point to our showcase of these same computer vision techniques in a Jupyter notebook system.**

Keywords—Computational Thinking, Digital Curation, Computational Archival Science (CAS), PII, Privacy

I. BACKGROUND

At the Digital Curation Innovation Center in the University of Maryland's College of Information Studies we have developed a method for reading stored data from the legacy punchcard computer storage medium. This work supports an archival use case of the NCSA "Brown Dog" service[1], which provides a web-scale API for extracting information from heterogeneous legacy data formats. "Brown Dog" was funded through a National Science Foundation DIBBs grant (Data Infrastructure Building Blocks) and was a partnership between the National Center for Supercomputing Applications at the University of Illinois and the College of Information Studies at the University of Maryland at College Park. This paper further develops a computer vision method, demonstrated in Jupyter notebooks, that we use to showcase the work of computational archival science to UMD students and educators. It is part of a broader effort to showcase and demonstrate computational methods and treatments for archival materials, which we have collected into the cases.umd.edu website[2]. Step-by-step documentation and complete code for this treatment[3] and many other computational archives cases may be obtained through the Computation Archival Science Education System website, or CASES.

II. THE PUNCHCARD ERA

Until the mid-1970s most computer data and programs were stored on punched paper cards. These cards were stacked up in hoppers and then fed into machines which read the data mechanically into limited computer memory. A common card format was the IBM 029 Punched Card, which held 80 columns of text. Other kinds of punched cards have been used as ballots in elections, including the infamous 2000 presidential election in Florida, which prompted a recount due to "hanging chad" errors.

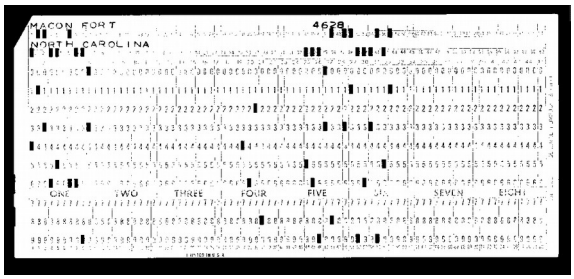

Fig. 1. Front of punchcard from our NARA samples.

Despite being a stable, paper format, punchcards have a very low data density of only 80 characters or bytes per card. In practice, most operations only used the first 73 columns to represent text and reserved the last few columns to record an order number. This was so that if a stack of cards was spilled on the floor, a machine could be used to re-sort the cards into the proper order. A big box of 2000 cards, about the size of a two large pizza boxes, can only hold 160 kilobytes of data. As a result of this extra weight and volume, most punchcards were disposed of years ago or reused for the next decade as convenient scratch paper.

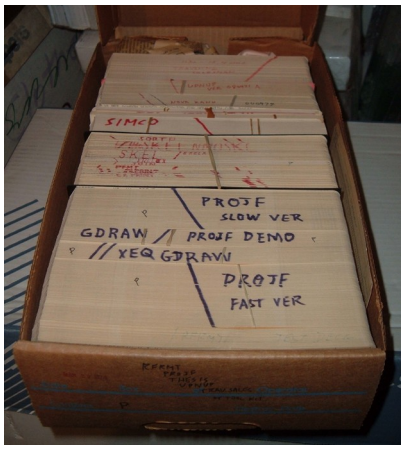

Fig. 2. Box of punchcards containing several computer programs, credit: Arnold Reinhold.

However, some are still being preserved in archives around the world, including some at the National Archives and Records Administration of the United States (NARA). It was these punchcards, originally from the US Army, that were brought to the Digital Curation Innovation Center for analysis. In fact, staff at NARA had already photographed the cards and so we received digital images of the original cards. The card images we processed had been an index for Army Morning Reports[4]. The NARA punchcards were our sample data

III. Computer-Vision Method

A. *Legacy Blog Post Code*

At the start of the project we went looking for any pre-existing code that could be used to interpret punchcard images. The best example that we found was a 2012 post[5] on the "Code included" blog by hobbyist Michael Hamilton. Mr. Hamilton built his own machine to image punchcards and then wrote Python 2.x code to make sense of the images. Hamilton's code was just right for our purposes after updating it to work with modern Python and imaging libraries. He had investigated the precise layout of the IBM 029 card format and his software calculates exactly where to look for punched holes in a punchcard image. By measuring the light or darkness of these hole punch areas, he can interpret the card, using a card template to map hole locations to their representative characters.

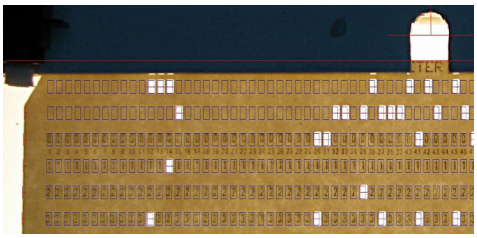

Fig. 3. Example image from Michael Hamilton. Credit: Michael Hamilton 2012

B. *Card Image Data*

We worked with a sample of 38 punchcard images. These were photographed from the back of the card, in order to avoid the stray marks and printing that was common on the front of the cards. What resulted was a clean image of a white card on a black background with black holes where punches were made. The high contrast between light and dark areas makes our computer vision procedure more accurate. Card images were loaded into Python, using the Python image library (Pillow). From there we were able to analyze each card's pixel data mathematically and using standard image processing tools. Whereas Hamilton's punchcard reader code works on backlit images of the fronts of cards, our card images are front-lit images of the backs of the cards. His code cannot be used directly to interpret the NARA punchcards. In addition, in order to better support the "Brown Dog" archival use case, we decided that our tool would be general purpose and support arbitrary punchcard images in any light, flip or orientation, so long as the edges were lined up within the frame of a photo. We also decided that our tool should return a clear yes or no result, indicating whether there is a punchcard in the image at all. This meant that we had to write code to detect punchcards and normalize any punchcard image to a form that Hamilton's code can interpret.

Our first step is to convert the "color mode" of the image to grayscale, since we only care about light and dark and not hue. "L" mode in the Python Image Library indicates a grayscale color mode with 8-bits representing each pixel. That means that each pixel in our image will be in the range of 0 (pitch black) to 255 (white), i.e. the binary values from '00000000' to '11111111') Using grayscale mode makes image math easier, as each pixel is represented by a single brightness value, instead of three separate values for red, green, and blue.

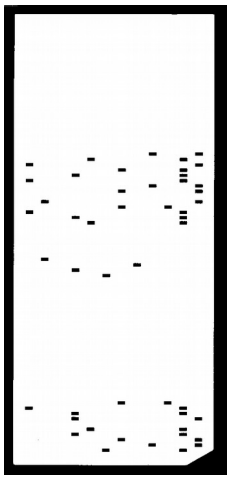

Fig. 4. Back of a card that is front-lit with 90-degree rotation

C. *Backlight*

Some cards may be photographed on backgrounds that are darker than the card itself, making the punches or holes appear darker than the card. Our data reading script is based on backlit cards, with holes showing lighter than the card itself. We must detect this situation and then correct it. We came up with a relatively simple formula for testing whether an image is backlit or not. We simply divide the image up into center and edge regions and then see if the center is darker or lighter on average, than the edge. In the code snippet below, we divided the image into the center region and a region just left of center. The brightness function below calculates the average brightness for an image.

```
w, h = image.size
```

```
# pick a center region
ctr_box = (w/4, h/4, (w/4)*3, (h/4)*3)
ctr_region = image.crop(ctr_box)
# pick a region left of center
left_box = (0, h/4, w/4, (h/4)*3)
left_region = image.crop(left_box)
left_bright = brightness(left_region)
ctr_bright = brightness(ctr_region)
backlit = left_bright > ctr_bright
if not(backlit):
    image = ImageOps.invert(image)
```

Fig. 5. Code for backlight normalization

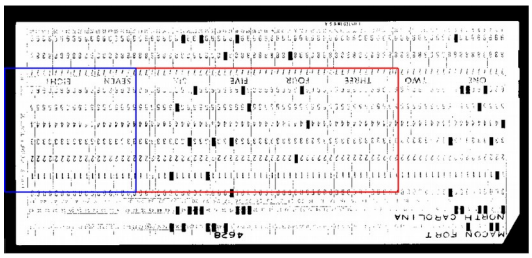

Fig. 6. Card marked with left and center regions.

We can run this code on our sample image to determine if the card in the middle of the image is darker or lighter than the background behind the card. If we find out that our image is already backlit, then there is nothing further to do. If it is not yet backlit, then we will invert the grayscale values of our image. Having normalized the lighting of our punchcard, we can proceed to work on a consistent image of a dark, backlit card.

D. Crop

The Hamilton code already contains some logic for detecting edges and cropping punchcards appropriately. However, in order to correct the rotation and the flip of incoming cards, we need to find the edges ourselves first. We need to know the geometry of the card itself, in order to verify that it has standard punchcard dimensions and in order to detect and place the "dog ear" on the upper-left corner.

One way to put this puzzle is that we need to cut away the portions of the image that surround the card. Now that we have a backlit card, these edge regions are white. Another way of saying this is that pixels on the edge have higher numerical values, close to 255. Our task is to find the left, right, top, and bottom edges, where the white pixels stop, and the black card pixels begin. We use simple math to do this, based on the color values stored for each pixel. Using a two-dimensional array of the image pixel values, we sum up the values vertically, just like you would produce a total row in a spreadsheet. This yields a list of the total brightness from left to right in the image.

```
import numpy
pix = numpy.array(image3)
# 0 here is the vertical axis
totals = numpy.sum(pix, axis=0)
```

Fig. 7. Code to total vertical pixel values.

If we plot our brightness totals for a sample card, then it looks like Fig. 5. Notice that there are higher values on the far left and right edges. These indicate the bright background pixels.

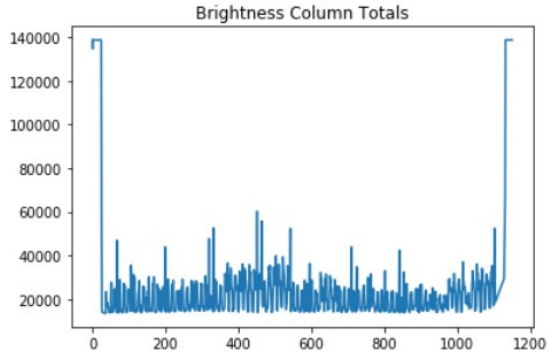

Fig. 8. Plot of brightness totals, left to right

The next snippet shows how we pick a threshold value, as a percentage of the maximum possible value, then loop through the list of brightness totals from each end, in order to find out where the values cross our threshold value. This will give us the left and right edge of our card.

```
max = pix.shape[0]*255
threshold = int(max*.8)
first = 0
for i in range(0, len(totals)-1):
    if(totals[i] < threshold):
        first = i-1
        break
last = len(totals)-1
for i in range(0, len(totals)-1):
    if(totals[len(totals)-1-i] < threshold):
        last = len(totals)-i
        Break
```

Fig. 9. Code to detect card edges.

By summing the image pixel values horizontally, the same algorithm will work to detect the top and bottom edges of the punchcard, giving us the region within the image that represents the punchcard itself. Cropping the image is a simple call to a library function.

```
crop_box = (left, top, right, bottom)
cropped_image = image.crop(crop_box)
```

Fig. 10. Code to crop the card image.

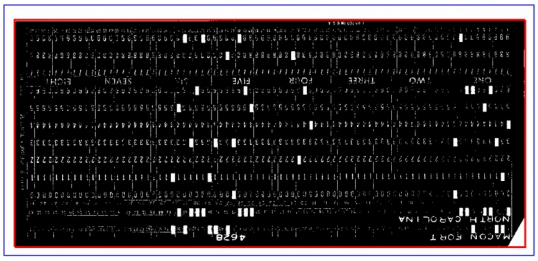

Fig. 11. Card with crop region shown in red.

E. Orientation

Now that we have an image containing the card by itself, we can normalize the orientation of the card to a landscape layout, in which the longer edges are positioned on the top and bottom. This involves a straightforward test to see if the image is taller than it is wide and then a call to the image library.

```
if(image.size[1] > image.size[0]):
    landscape_image = \
      image.transpose(Image.ROTATE_90)
else:
    landscape_image = image
```

Fig. 12. Code to detect card orientation and rotate to landscape.

F. Detect Punchcard

With a cropped card and a known orientation, we have enough data to determine whether or not this is an image of a punchcard or not. A standard IBM 029 punchcard measures 7 3/8 inches long and 3 ¼ inches tall. If we allow for some slight variation, then we can write a test to see if our card's proportions are the same. We'll create two height to width proportions, one slightly higher than standard and one slightly lower. Then we can see if the image proportions lie between those two numbers.

```
WIDTH = 7.0 + 3.0/8.0
HEIGHT = 3.25
TOLERANCE = .15
W_TO_H_RATIO_HIGH = \
  (WIDTH + TOLERANCE) / HEIGHT
W_TO_H_RATIO_LOW = \
  (WIDTH - TOLERANCE) / HEIGHT
card_ratio = \
  float(image.size[0]) / float(image.size[1])
if card_ratio <= W_TO_H_RATIO_HIGH:
  if card_ratio > W_TO_H_RATIO_LOW:
    print("detected punchcard")
```

Fig. 13. Code to detect a punchcard shape.

Now we know if we are looking at an image with punchcard proportions. These are the only images we want to feed to Hamilton's reader code, but before we do that there is one more step.

G. Flip

Hamilton's code expects the "dog ear" or angle cut-out in the card to be positioned in the top-left of the image. That is where the dog ear is normally positioned when you look at the front of a paper punchcard. However, we want our code to work even when the card was photographed from the back, as is the case with our NARA samples. There are two kinds of image flip operations we can do, left-right and top-bottom, but we may need to perform both if the dog ear is in the bottom-right, or neither if the "dog ear" is already in the top-left. Since the dog ear reveals more of the white background, the dog ear corner will have a higher average brightness than the other corners. We can use this fact to choose the right combination of flip operations. In our code we perform each possible combination of flip operations, measuring the brightness of the top-left corner. Then we choose the flip combination for which that corner was brightest. Python has some useful functions that help us iterate through all the combinations of two choices.

```
# top-left corner box
corner_box = (0,0,20,20)
brightest = -1
flipped_image = None

for (lr,tb), value in \
numpy.ndenumerate(numpy.zeros((2,2))):
    test = landscape_image
    if lr:
        test = \
        test.transpose(Image.FLIP_LEFT_RIGHT)
    if tb:
        test = \
        test.transpose(Image.FLIP_TOP_BOTTOM)
    b = brightness(test.crop(corner_box))
    if b > brightest:
        brightest = b
        flipped_image = test
display(flipped_image)
```

Fig. 14. Code to flip image and position the "dog ear".

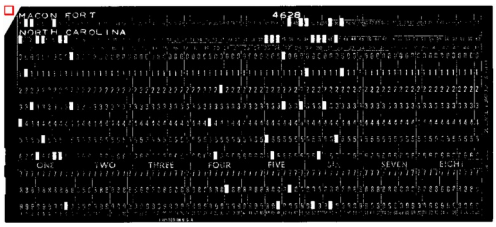

Fig. 15. Flipped image with test box in red.

This code performs every combination of left-right and top-bottom flips, by denumerating the positions in a 2-by-2 array, starting with (0, 0) representing no flip operations, and ending with (1,1) representing both left-right and top-bottom flips. Whichever combination produces the brightest square in the top-left corner is the solution, i.e. the best flip combination.

H. Detect Holes

We have addressed the discrepancies between potential input images and those used by Michael Hamilton. Our image is still cropped to only include the area of the card itself, but Hamilton's code expects a small border region. We can add a border with a call to the image libraries expand function.

```
from PIL import ImageOps
border_image = \
ImageOps.expand(flipped_image, \
    border=10, fill='white')
```

Fig. 16. Code to add a white border.

Having taken these steps, the image in Fig. 17 is ready for reading.

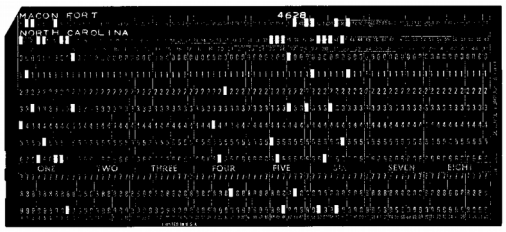

Fig. 17. Punchcard image ready for reading.

Hamilton's punchcard reading code is available as a Python library, so the code to use it is terse. In Fig. 18 we load the punchcard library, load the image, read the image, and then print the results.

```
from punchcard import PunchCard
from PIL import Image
image = Image.open('prepared_image.png')
# using 127 or neutral gray as threshold for hole
vs. card
card = PunchCard(image, bright=127)
print('Punchcard Text: {}'.format(card.text))
print()
card.dump('my card')
```

Fig. 18. Code that reads a punchcard.

```
Punchcard Text: MACON FORT                    4628      NORTH CAROLINA

 Card Dump of Image file: my card Format Dump threshold= 127
 123456789-123456789-123456789-123456789-123456789-123456789-123456789-123456789-
 ,-------------------------------------------------------------------------------.
/MACON FORT                      4628      NORTH CAROLINA                       |
|.OO...O.....................................0.OO...O.O.....................    |
|O..OO..OO.....................................OOO.....OOO.O.....................|
|.........O.....................................O.....................    |
|.O.....................................................O.....O.....................|
|.....................................O.....................    |
|..O......O.....................................O..O...O.....................|
|O.....................................O.....................    |
|....O.....................................O..........O.....................|
|...O..OO.....................O.........O.......O.....................    |
|                                                                               |
|.....................................O..........O.....................    |
|.........O.....................................O.....O..O.....................|
 `-------------------------------------------------------------------------------'
 123456789-123456789-123456789-123456789-123456789-123456789-123456789-123456789-
```

Fig. 19. Printout of punchcard text and the card rendered as plain text.

IV. Further Resources

A. NCSA Brown Dog Extractor Tool

The code examples used above are also incorporated into an information extraction tool within the Brown Dog service. This punchcard extractor tool is available on demand to users of Brown Dog and will return both the text string found on any punchcard image, as well as whether a punchcard was detected at all. The Brown Dog service is full of different extractor tools that operate on all kinds of formats, from documents to multi-media or dataset files. There are many other computer vision extractors, including optical character recognition and facial recognition. There is even an extractor that looks at old maps and attempts to clean up and extract the paths of rivers. For more information, please see the Brown Dog website[6] and the Brown Dog tools catalog[7].

B. Computational Archival Science Education System (CASES)

CASES is a community-based initiative, hosted by the Digital Curation Innovation Center, that seeks to provide working examples of the computation treatments of archival materials. The CASES website[8] includes a showcase of Jupyter notebooks that have been developed and curated to support the teaching of computation archival science in any archives program. These notebooks, including the Punchcard Reader notebook[9], are cataloged according to relevant archival practices, computational practices, and ethical concerns. Each notebooks is provided as a Git code repository to facilitate reuse and improvements. Users may check out their own copies of any notebook to a hosted service, like MyBinder[10], or to a local Jupyter Lab[11] install. We hope that this can be a resource for educators, practitioners, and students who wish to pursue computational treatments.

I. References

[1] Brown Dog website. Available online: http://browndog.ncsa.illinois.edu/ (accessed on 7 October 2019)

[2] Available online: http://cases.umd.edu (accessed on 7 October 2019)

[3] Jansen, G., "Reading Punchcards Jupyter Notebook", 2019. Available online: TBD

[4] Army Morning Reports, ca. 1912 – 1946, available online: https://catalog.archives.gov/id/85713825

[5] M. Hamilton, punchcard Reader – The Software, Code included Blog, Jul 25, 2012. Available online: http://codeincluded.blogspot.com/2012/07/punchcard-reader-software.html (accessed on 7 October 2019)

[6] Brown Dog website. Available online: http://browndog.ncsa.illinois.edu (accessed on 12 October 2019)

[7] Brown Dog tools catalog. Available online: https://browndog.ncsa.illinois.edu/toolscatalog/ (accessed on 12 October 2019)

[8] The CASES website. Available online: http://cases.umd.edu/ (accessed on 12 October 2019)
[9] Punchcard Reader Notebook. Avaiable online: http://cases.umd.edu/github/cases-umd/Reading-Punchcards/blob/master/index.ipynb (accessed on 12 October 2019)
[10] MyBinder website. Available online: https://mybinder.org/ (accessed on 12 October 2019)
[11] Project Jupyter website. Available online: https://jupyter.org/ (accessed on 12 October 2019)