Economics 2355: OCR and Post-Processing in Practice

Melissa Dell

Harvard University

February 2021

Outline

There Are a Variety of OCR Engines

- Tesseract (open source)
- Easy OCR (open source, geared towards OCR in the wild)
- Google Cloud Vision (proprietary)
- ABBYY Fine Reader (proprietary)
- Amazon Textract (proprietary)...

Tesseract is a good place to start, unless you are working with natural images (in which case try EasyOCR). For a limited number of documents, the cost of a product like GCV is very reasonable if it gives better results.

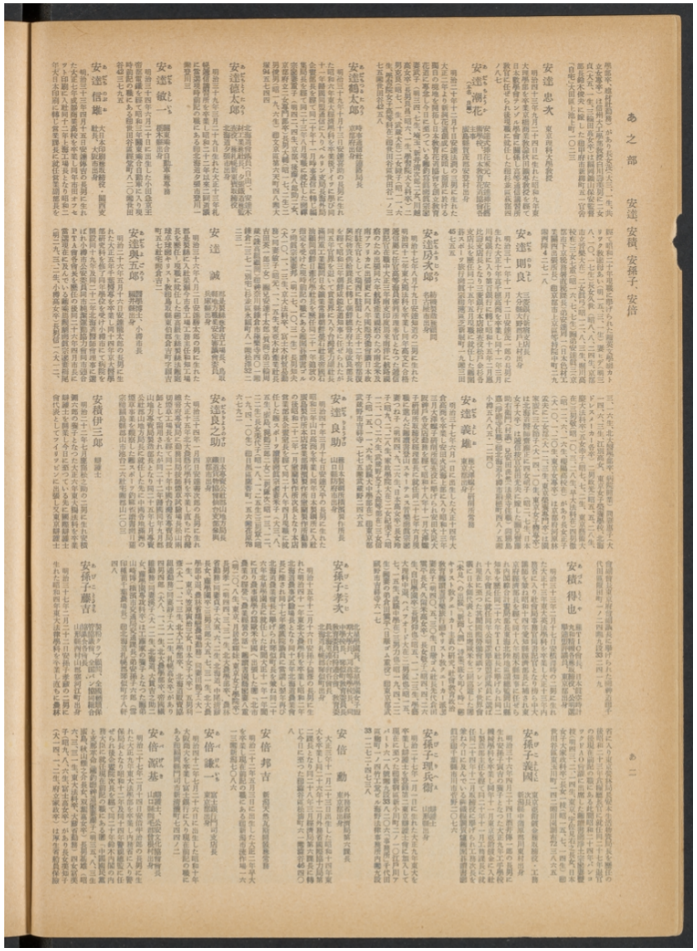
Choosing an OCR Engine

- Tesseract has two major advantages: it is free and you can understand what's going on under the hood (which both result from it being open source). Use the most recent beta v5.
- Easy OCR is likely to be the clear winner if you need to process natural images
- We've found relatively high performance from GCV in accuracy tests, but there's no way to know what it's doing under the hood (difficult to troubleshoot poor results) and it can be pricey
- OCR is likely to change a lot over the coming few years - glaring need for a Transformer-based OCR

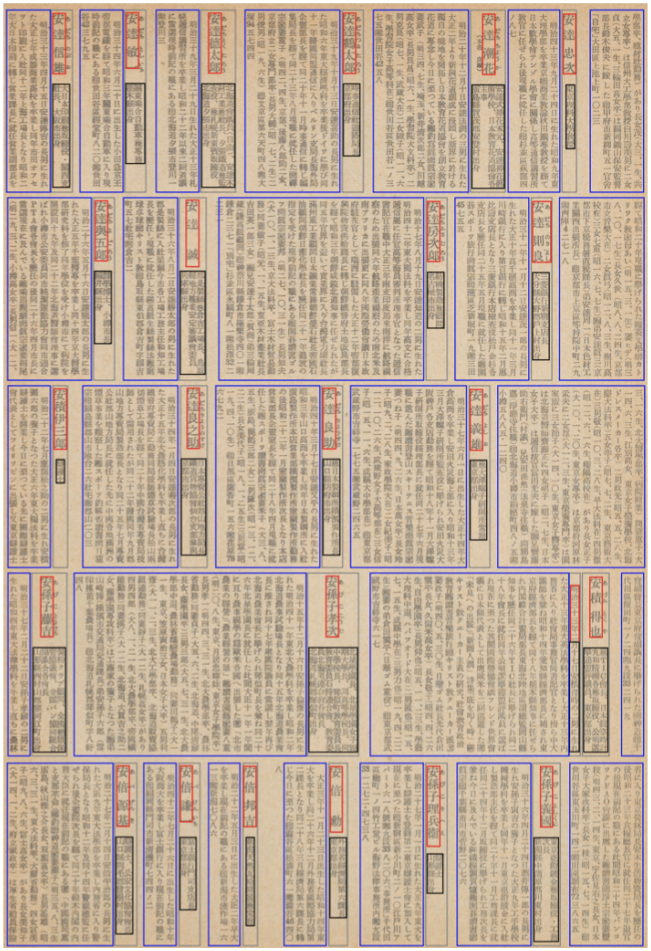
Complex Layouts Confuse OCR

- Characters are more like to mis-OCR when layouts are even mildly complex, and complex layouts make OCR significantly more likely to fail to detect characters at all
- Converting document layouts to standard formats can substantially improve OCR results, for what might otherwise be unusable outputs
- Essentially, the aim is to rearrange documents to mimic the documents that the OCR engine was trained on, i.e. dense, single column texts

An Example



(a) A Input Raw Scan



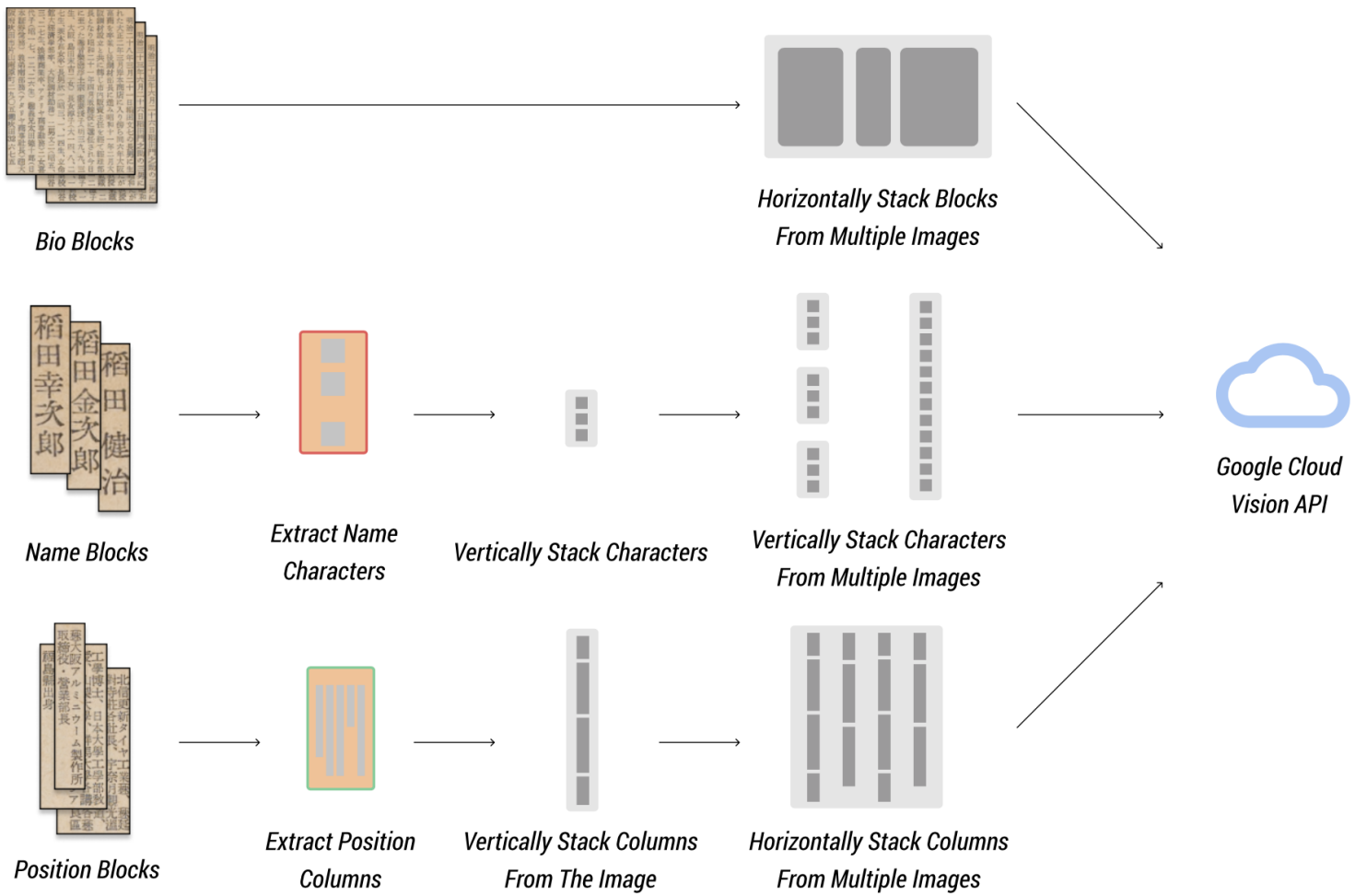
(b) Scan with Extracted Block Boxes and Types

Reads top-to-bottom and right-to-left (rotate 90 degrees to make like English)

An Example

- The *bio blocks* consist of multiple columns of dense text, and can simply be appended horizontally until reaching the maximum image size that Google Cloud Vision accepts.
- The *name blocks* consist of a single column of characters. We found that accuracy was maximized when stacking them vertically into a single, thin row. To minimize OCR costs, we stacked until reaching the maximum image size that Google Cloud Vision allows.
- Finally, the *position blocks* are typically a few columns long, and the last column is usually partially empty. This introduces a lot of sparsity. Hence, we segment this section into columns. We combine short columns together to create a dense image with homogeneous height.

An Example



Sending individual layout tokens can be expensive and also confuse the OCR engine.

Rearranging Layouts for OCR

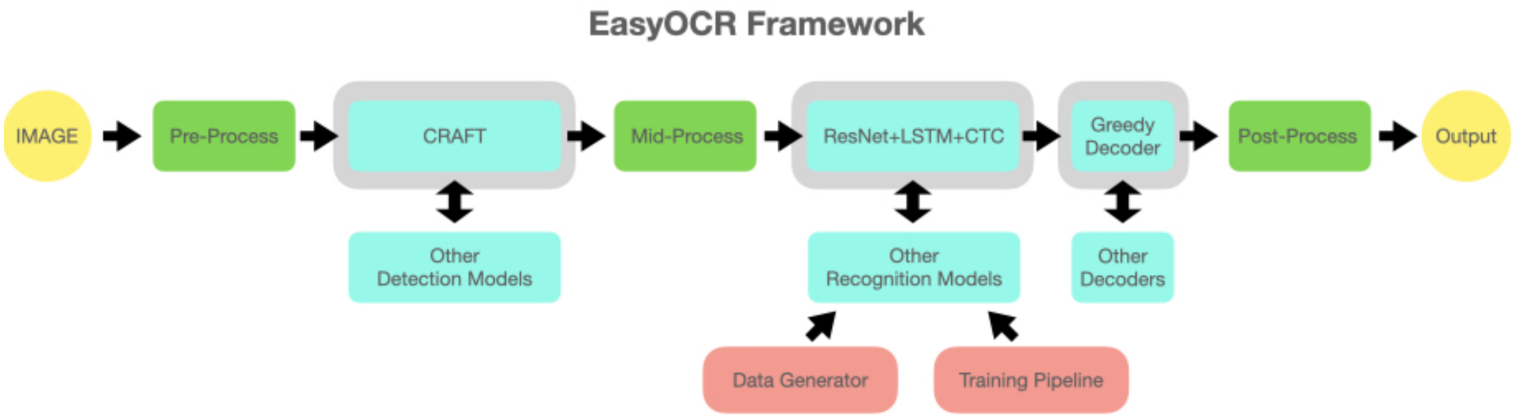
- This can definitely feel a bit hack-y, and can take some experimentation, but when it works it can salvage a project
- We have found in some cases that results can be highly unstable to the details of the restacking, presumably a result of the OCR engine being underexposed to similar documents/fonts in training. In other cases, though, rearranging is all you need for a good OCR

An Open-Source Tool for Rearranging Layouts

- We created an open-source tool that makes it easy to rearrange document layouts for OCR
- It is part of a broader layout analysis package, which I will discuss in a few minutes

Easy OCR

Open-source package primarily for OCR in the wild



Since it's open-source, you should be able to extract their codebase for the ResNet+LSTM+CTC and build your own engine around it. All OCR has various processing - i.e. it gets fed binary images

Tesseract OCR

- Historically, Tesseract used a rule-based architecture, which was a direct extension of the model's development by HP in the 1980s and 1990s. You can read more about this original rule-based architecture here, in a paper authored by the head of Tesseract maintenance at Google: `https://github.com/tesseract-ocr/docs/blob/master/tesseracticdar2007.pdf`
- However, since version 4, Tesseract has switched to using a (mostly) deep architecture based on LSTMs. Specifically, LSTMs are integrated into Tesseract since version 4 as a neural network subsystem that does "line recognition," which is to say that one part of Tesseract handles layout analysis so as to identify single text line regions within a large document, and then another part of Tesseract features a LSTM network used to recognize the content of these regions.

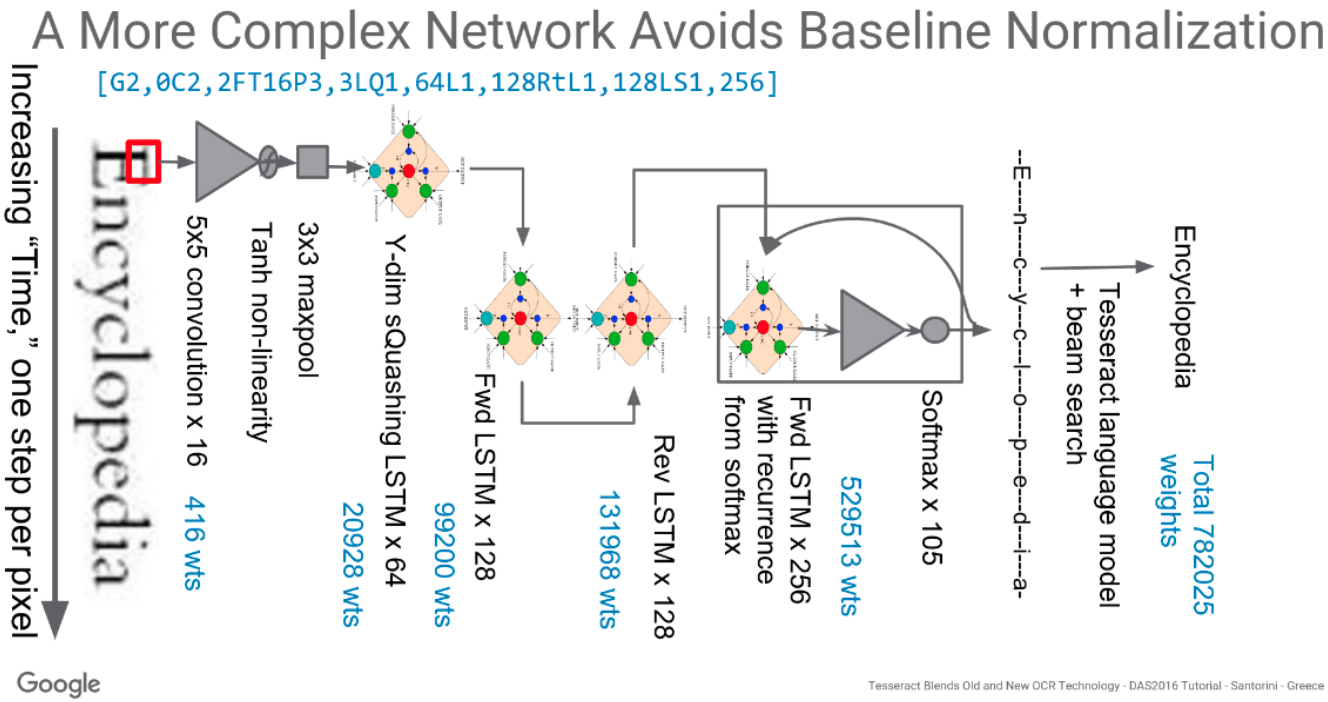
Tesseract OCR

- You can read more about the rule-based layout analysis methods used by Tesseract here: `http://tesseract-ocr.github.io/docs/`, `https://github.com/tesseract-ocr/docs/blob/master/das_tutorial2016/5LayoutAnalysis.pdf`
- The base architecture of the LSTM subnetwork of Tesseract is written in its native Variable-size Graph Specification Language (VGSL) as such, read from bottom to top:

```
O1c105: Output layer produces 1-d (sequence) output, trained with CTC,
  outputting 105 classes.
Lfx256: Forward-only LSTM in x with 256 outputs
Lrx128: Reverse-only LSTM in x with 128 outputs
Lfx128: Forward-only LSTM in x with 128 outputs
Lfys64: Dimension-summarizing LSTM, summarizing the y-dimension with 64 outputs
Mp3,3: 3 x 3 Maxpool
Ct5,5,16: 5 x 5 Convolution with 16 outputs and tanh non-linearity
1,0,0,1: Input is a batch of 1 image of variable size in greyscale
[]: The network is always expressed as a series of layers.
```

Tesseract OCR

- A schematic of the model (circa 2016, the year of introduction) is also seen here:



- For more information about fine-tuning and customizing Tesseract, see: `https://tesseract-ocr.github.io/tessdoc/`

Outline

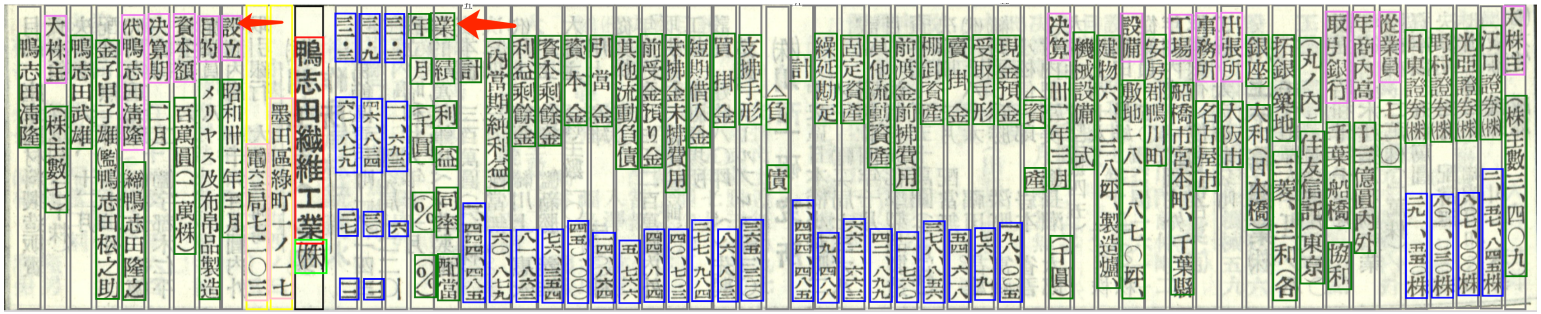
Customized OCR

- You could also customize your own OCR engine
- This is most straightforward in the context of number detection, where ultimately all you need to do is recognize digits and associated characters (comma, dollar signs, percentages, etc)

Customized OCR

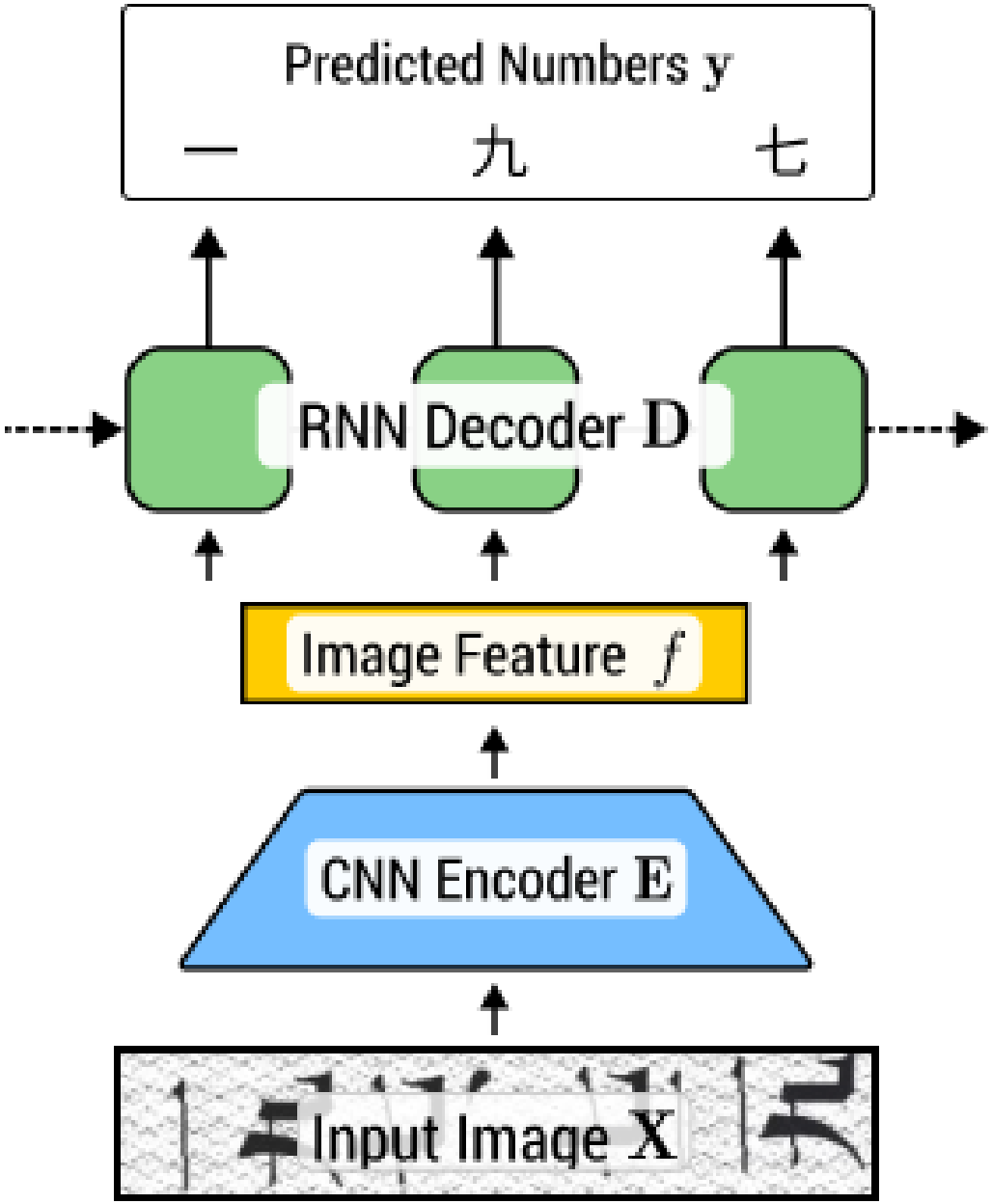
- As we saw last class, the currently best developed architecture involves a CNN+RNN+CTC
- You will need labeled samples to train the model; recall that for this framework labels give the text in the image but do not provide the bounding boxes for each character
- This framework is hence going to struggle with complex layouts. You would want to use i.e. Mask R-CNN to identify layout elements first, then feed those into your detector, rather than feeding a more complex table into the detector

Our Customized Number OCR



Numbers in this document use a flat font. Trivially distinguishable to the human eye, but OCR engines can't even recognize there is text there (tells you something about the sparsity of training data). We first use Mask R-CNN to segment out the numbers, then send them to our own detector.

Our Customized Number OCR



Our Customized Number OCR

- As always, a major challenge is how to create labeled training data at reasonable cost
- Started with the closest font we could find, flattened it further, and added random background noise, text skewing, and blurring. Used this library for text generation: `https://github.com/Belval/TextRecognitionDataGenerator`
- In the future, I'd probably start with a generative deep learning model like CycleGAN
- We also needed labeled samples from our actual documents, as the generated data didn't fully mimic the actual data

Outline

Putting it All Together: Layout Parser

- One of my pre-docs, Zejiang Shen, worked on integrating the models and tools we developed into an open-source package called `LayoutParser` (`https://github.com/Layout-Parser/layout-parser`)
- The aim is to streamline the use of DL in document image analysis (DIA) pipelines
- It provides simple and intuitive interfaces for applying and customizing DL models for layout detection, character recognition, and other document processing tasks
- It will also incorporate a community platform (currently under development) for sharing both pre-trained models and full digitization pipelines
- The package is described in a paper that will be released soon, along with a major update

Motivation

- Implementing DIA pipelines from scratch requires substantial expertise
- Currently there is no full-fledged infrastructure for easily curating the target document image datasets and fine-tuning or re-training layout analysis models
- It is difficult for research teams to learn about how full pipelines are implemented and leads them to invest significant resources in reinventing the DIA wheel

Layout Parser Components

1. An off-the-shelf toolkit for applying DL models for layout detection, character recognition, and other DIA tasks
2. A repository of pre-trained neural network models (Model Zoo) that underlies the off-the-shelf usage
3. Comprehensive tools for efficient document image data annotation and model tuning to support different levels of customization
4. A DL model hub and community platform for the easy sharing, distribution, and discussion of DIA models and pipelines for reusability, reproducibility, and extensibility

The library is implemented with simple Python APIs.

The Core Layout Parser Library

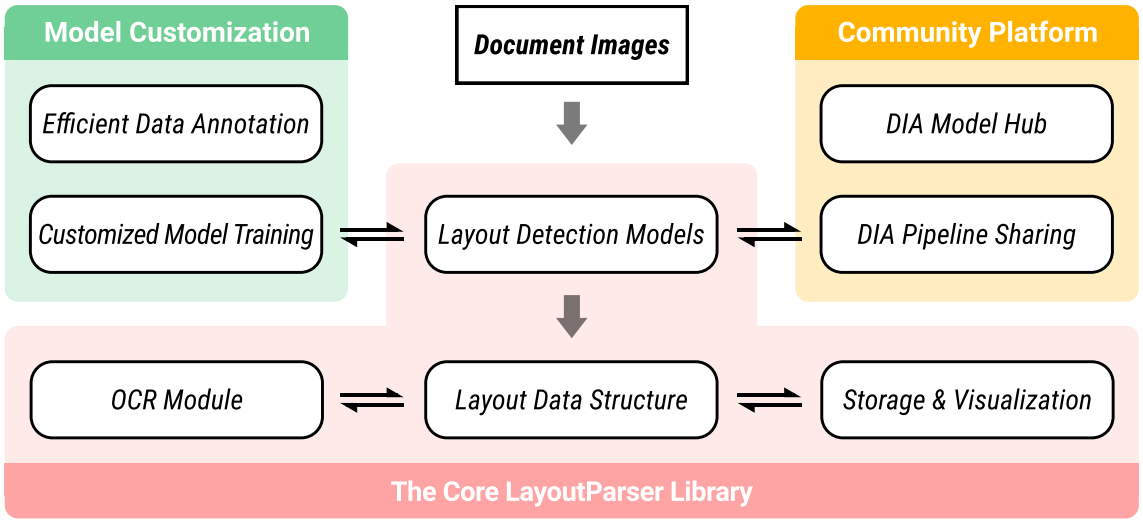


Figure: For an input document image, the core `LayoutParser` library provides a set of off-the-shelf tools for layout detection, OCR, visualization, and storage. `LayoutParser` also supports high level customization via efficient layout annotation and model training functions.

Layout Detection

LayoutParser, built upon Detectron2, provides a minimal API that can perform off-the-shelf layout detection with only four lines of code in Python

```
import layoutparser as lp
image = cv2.imread("image_file") # load images
model = lp.Detectron2LayoutModel(
    "lp://PubLayNet/faster_rcnn_R_50_FPN_3x/config")
layout = model.detect(image)
```

A layout model takes a document image as an input and generates a list of rectangular boxes for the target content regions. It is formulated as an object detection problem and state-of-the-art models like Faster R-CNN and Mask R-CNN are used.

Layout Detection

Dataset	Base Model	Large Model	Notes
PubLayNet	F / M	M	Layouts of modern scientific documents
PRImA	M	-	Layouts of scanned modern magazines and scientific reports
Newspaper Navigator	F	-	Layouts of scanned US newspapers from the 20th century
TableBank	F	F	Table region on modern scientific and business document
HJDataset	F / M	-	Layouts of history Japanese documents

Table: Current layout detection models in the `LayoutParser` model zoo. For each dataset, we train several models of different sizes for different needs (the trade-off between accuracy vs. computational cost). For "base model" and "large model", we refer to using the ResNet 50 or ResNet 101 backbones, respectively. One can train models of different architectures, like Faster R-CNN (F) and Mask R-CNN (M). For example, an F in the Large Model column indicates it has a Faster R-CNN model trained using the ResNet 101 backbone.

Layout Data Structures

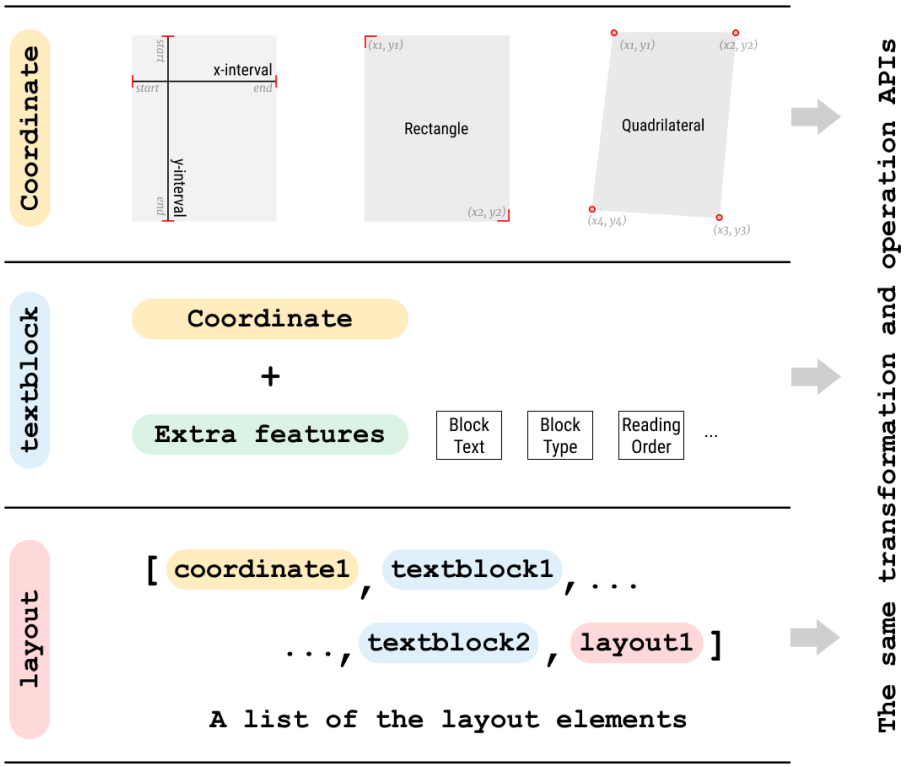


Figure: The relationship between the three types of layout data structures. `Coordinate` supports three kinds of variation; `TextBlock` consists of the coordinate information and extra features like block text, types, and reading orders; a `Layout` object is a list of all possible layout elements, including other `Layout` objects. They all support the same set of transformation and operation APIs for maximum flexibility.

Operations

- A wide collection of transformations like shift, pad, and scale, and operations like intersect, union, and is_in, are supported
- It is common to separate a segment of the image and analyze it individually. `LayoutParser` provides full support for this scenario via image cropping operations crop_image and coordinate transformations like relative_to and condition_on that transform coordinates to and from their relative representations
- A Layout class is built that takes in a list of TextBlocks and supports processing the elements in batch. Layout could also be nested to support hierarchical layout structures

OCR

`LayoutParser` builds a series of wrappers among existing OCR engines, and provides nearly the same syntax for using them.

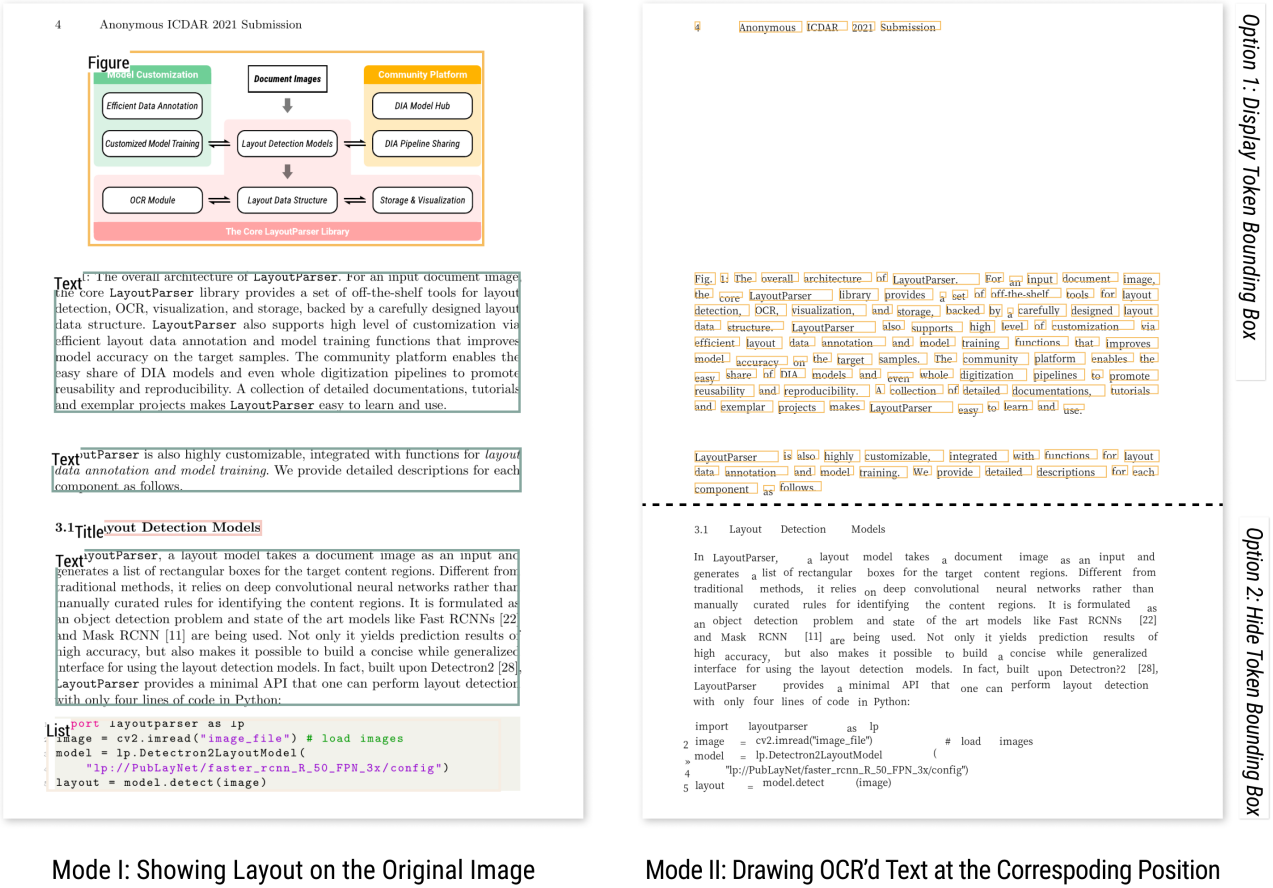
```
ocr_agent = lp.TesseractAgent()
# Can be easily switched to other OCR software
tokens = ocr_agent.detect(image)
```

Currently `LayoutParser` supports the Tesseract and Google Cloud Vision OCR engines. It also comes with a DL-based CNN-RNN OCR model trained with the Connectionist Temporal Classification (CTC) loss. It can be used like the other OCR modules, and can be trained on customized datasets.

Storage and Visualization

- `LayoutParser` supports exporting layout data into different formats like JSON and csv. We also support loading datasets from layout analysis-specific formats like COCO and the Page Format
- `LayoutParser` is built with an integrated API for displaying the layout information along with the original document image

Layout Detection and OCR Results



Mode I: Showing Layout on the Original Image

Mode II: Drawing OCR'd Text at the Corresponding Position

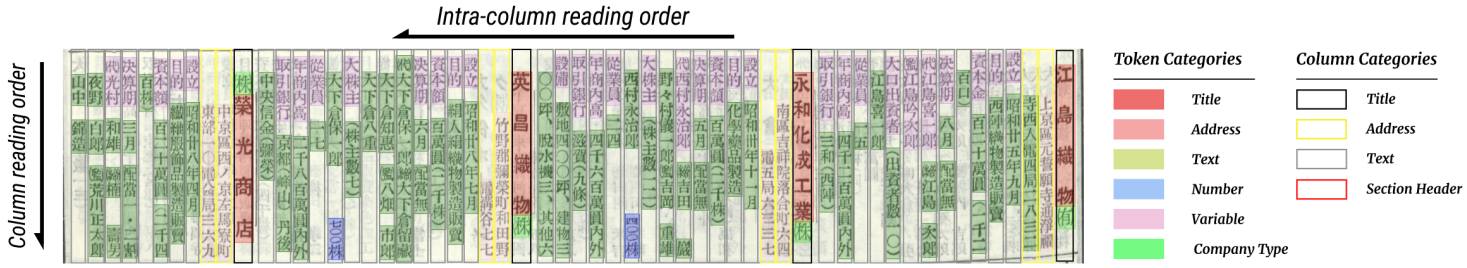
Figure: Mode I directly overlays the layout region bounding boxes and categories over the original image. Mode II recreates the original document via drawing the OCR'd texts at their corresponding positions on the image canvas.

Customized Model Training

- We will be integrating a customized version of Label Studio that incorporates the OLALA score and features (hopefully) in the near future
- `LayoutParser` also provides an integrated API for fine-tuning or pre-training a layout analysis model. Fine-tuning one of the existing models in `LayoutParser` makes it straightforward to apply transfer learning

Use Case: Japanese Firm Reports

To decipher the complicated layout structure, two object detection models have been trained to recognize individual columns and tokens, respectively. A small training set (400 images with approximately 100 annotations each) is curated via the active learning based annotation tool.



Errors can be identified and rectified via checking the consistency of the model predictions.

Re-Arranging Text to Achieve Better OCR

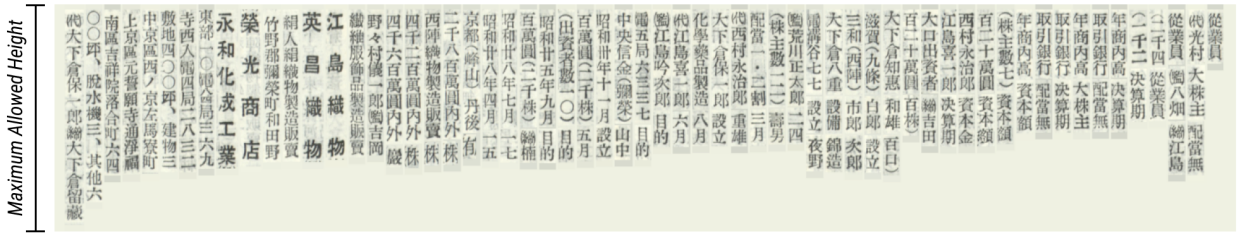


Figure: Illustration of a recreated document image that achieves a much better character recognition recall rate. The reorganization algorithm rearranges the tokens based on the their detected bounding boxes given a text orientation, a fill color (which needs to closely match the background), and the maximum allowed height.

Use Case: Japanese Firm Reports

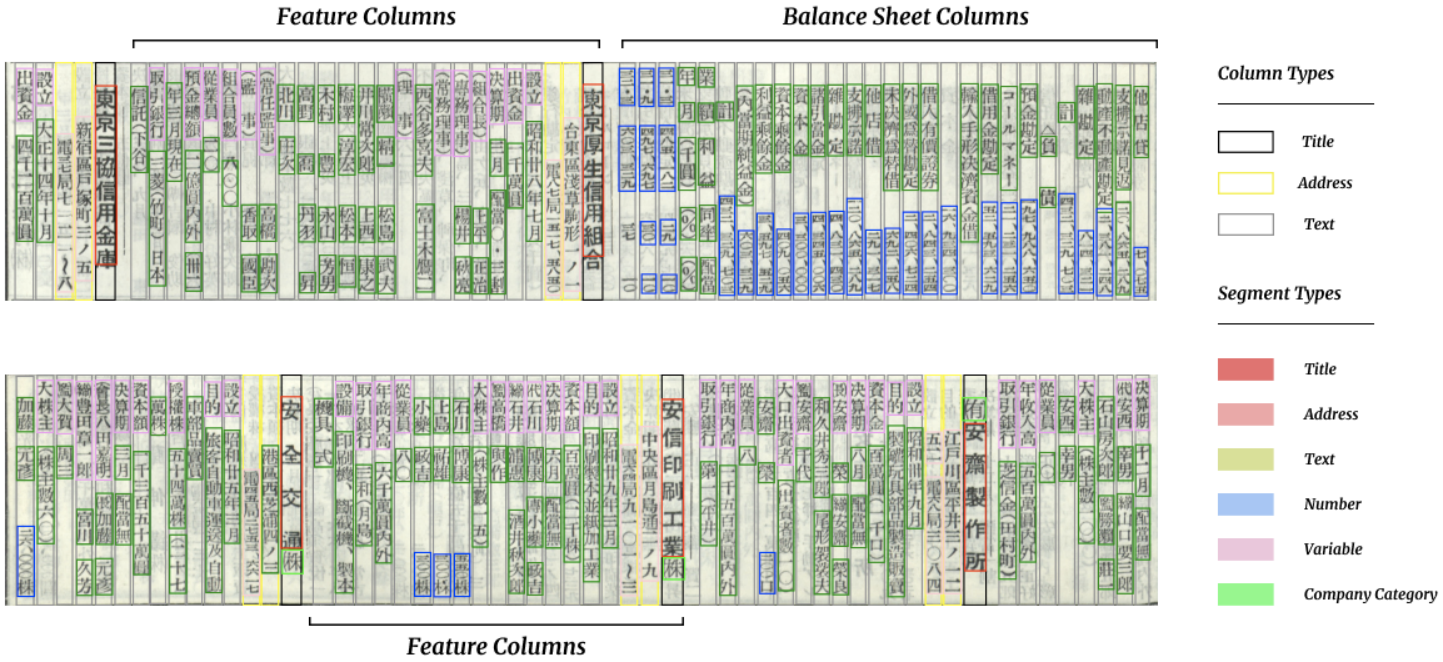
- Though trained on a small dataset, the pipeline achieves a high level of layout detection accuracy: it achieves a 96.97 AP score across 5 categories for the column detection model, and a 89.23 AP across 4 categories for the token detection model.
- We crop images within regions detected as numbers and identify characters within them using a self-trained OCR model based on CNN-RNN.
- The model detects a total of 15 possible categories, and achieves a 0.98 Jaccard score and a 0.17 average Levinstein distances for token prediction on the test set.

Outline

Creating Content Domains

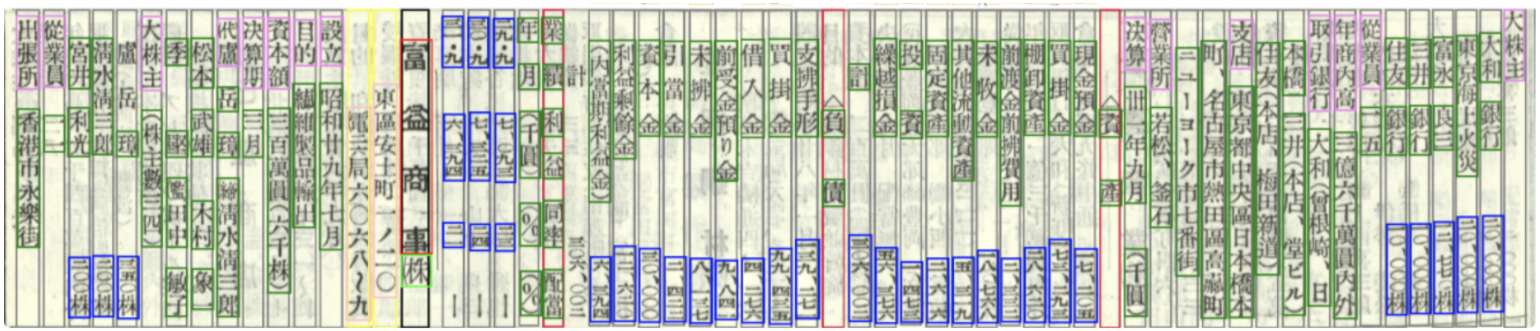
- Once individual content regions are recognized, we need to *associate the different pieces of content with each other.*
- How this is done depends on the nature of the documents.
- For example, if you are digitizing newspapers, you might use a deep-learning based sentence completion model to figure out how different segments of multi-column articles should be appended together.
- It is imperative that you design your pipeline so that you can extract the desired structure with high accuracy.

An Example: Document Structure



The column types tell us what section of the document we are in. A *title column* (black) indicates that information is being presented for a new firm and is followed by the firms' address columns (yellow). Then there are *regular text* columns (grey). These contain *variable names* (pink) and *variable values* (green for text values and blue for numbers).

An Example: Document Structure



Finally, there are *balance sheet columns* (the headers of the balance sheet are denoted in red in the below image; there are three headers for the assets, liabilities, and profits section of the balance sheet). Again, there are *variable names and values* within the balance sheet, but unlike in the main text, variable names are indented. The variable names are aligned at the top and variable values - which are always numbers (blue) are aligned bottom.

Defining Layout Classes

Layout element classes need to be designed such that:

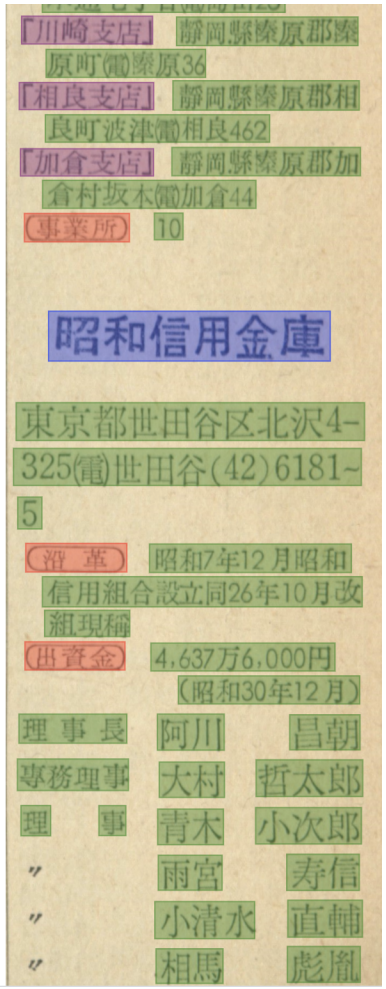
- They can be detected with high accuracy with a reasonable annotation budget. The more subtle the distinction you want to make between layout classes, the more labeled data that will be required
- It is straightforward to structure the digitized data as desired

It doesn't make sense to try to distinguish classes during the layout analysis that will be straightforward to extract with the text subsequently

Defining Layout Classes

- What determines a layout element is typically white space, or other visual features (such as the font) that denote the boundaries between different layout elements.
- Layout elements in different classes should have:
  - Different fonts (sizes/types/boldness)
  - Other distinct visual signatures (i.e. text that is circled, in brackets, in parentheses, etc)
- The more visually distinct, the less labels it will tend to require for accurate classification

Another Example



While there is other layout information that we need to extract from the text before creating structured data, these are the categories that can be distinguished by the visual signatures of the document alone.

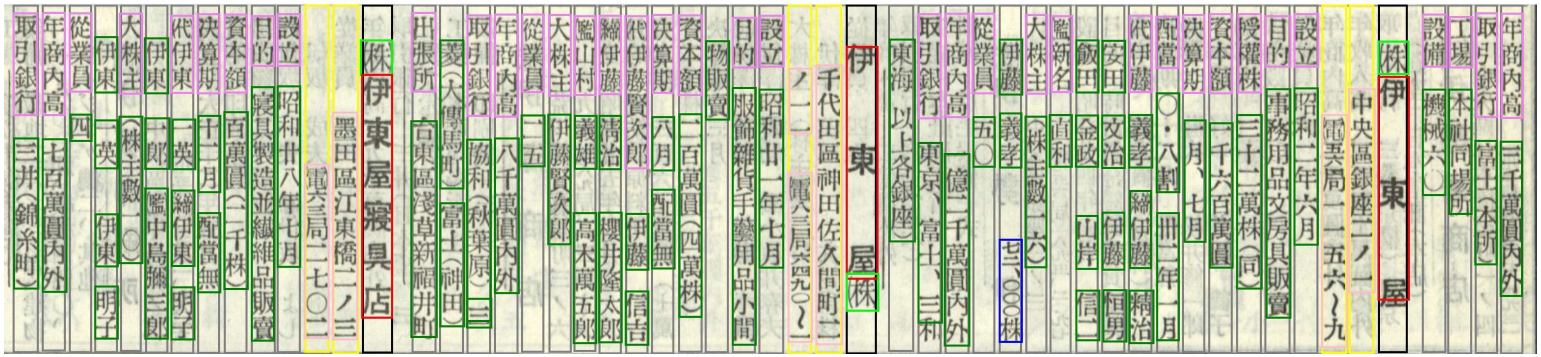
Another Example

- It is important to think about what you *must* distinguish from the visual signature in the image to create a structured database, and what you can more easily extract from the text once it is digitized
- It is obviously very important that we correctly classify the company name region, as otherwise the following firm level information will be associated with the wrong company.
  - The way a human reading the table does this is by the visual signature - the font is bold and large - so that's the way the computer should do it as well.
  - The text won't tell us a ton, as company names can also appear in each firm's text - i.e. in describing their banks, their major shareholders, their firm history, etc.

Another Example

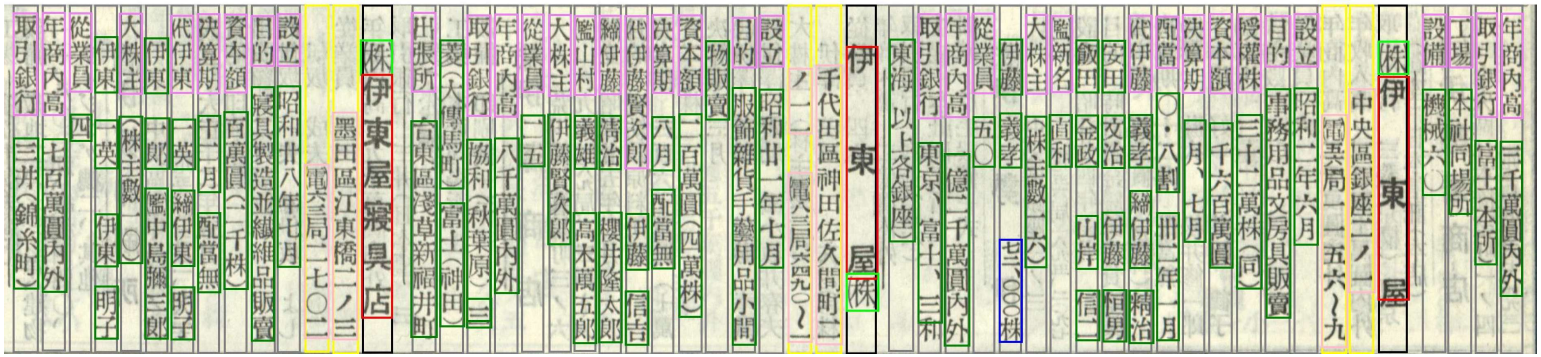
- In contrast, suppose that we want to distinguish individual pieces of information, such as whether a variable name is referring to the shareholders or to the assets of the firm.
- It is easier to do this with the digitized text subsequently - even with the inevitable OCR errors - as the visual signatures of these different variables are extremely similar.

Defining Layout Classes



- There is some room for subtlety in how the layout elements are defined, if enough data are labeled accordingly.
- For example, the company titles - in red boxes - have space between the characters, whereas in the text region (green), spaces denote separate elements.
- Characters composing the company title can be recognized as a single layout element despite the presence of the spacing because the size and boldness of the text distinguishes these regions from the green text class.

Extracting Layout Classes Without Deep Learning



- The font and characters of the variable names (pink) are very similar to those of the variable values (green).
- The way the human eye can easily distinguish what is a variable name (pink) is the indentation.
- Variable names always start at the top of the column. If a variable value spans multiple columns, it is indented on the subsequent lines.

Classifying Layouts Without Deep Learning

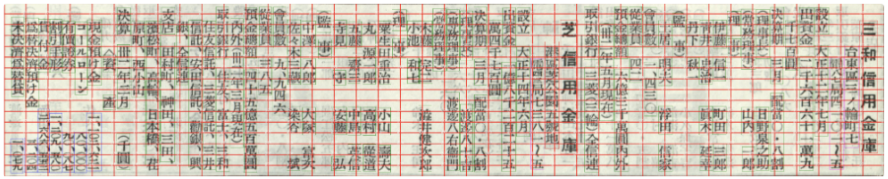
- Mask R-CNN had a much harder time learning to distinguish this indentation versus different fonts.
- Fortunately, the indentation is a very firm rule, and our scans are clean enough that it is never undone by skewness or warping of the scan.
- Hence, rather than having to label a lot of data so that the model can learn this **simple, inflexible** rule, we can just implement this rule in post-processing.

Classifying Layouts Without Deep Learning

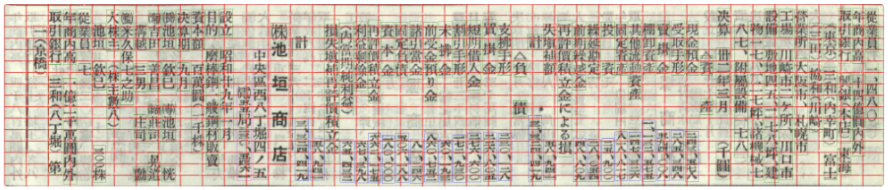
- To detect variable names in post-processing, it is best to use relative coordinates, rather than hard coding values.
- Based on the aspect ratio, we divide each column into a “ 16 x 40 “ grid. In the text columns - even when the column images are quite skewed - variable names always start in the top row whereas variable values do not.
- While we’ve cautioned that specifying rules to process layouts is often messier than you might anticipate, this is an example where a simple rule really does work extremely well in distinguishing layouts.

Classifying Layouts Without Deep Learning

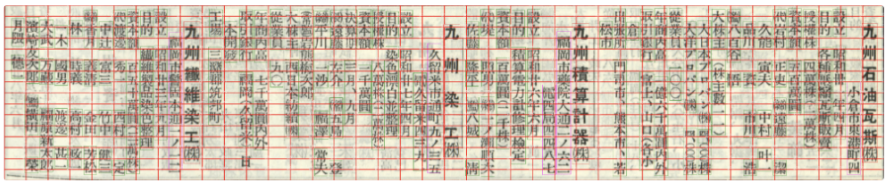
tk1957_0018_5_0-row4



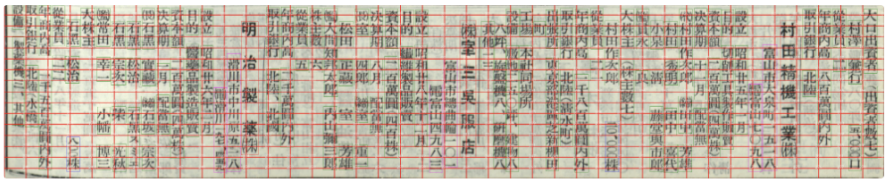
tk1957_0024_3_0-row6



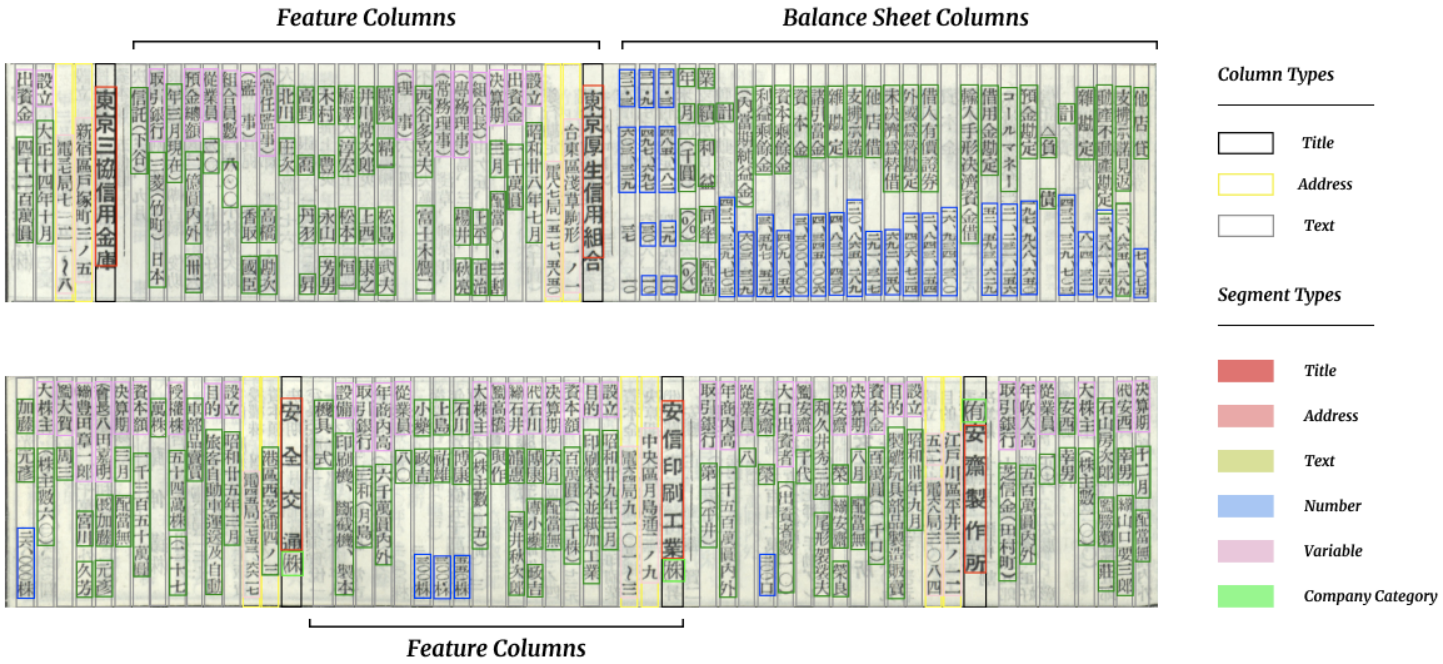
tk1957_0290_5_0-row3



tk1957_0249_2_0-row7



Layout Hierarchies



Such hierarchical structures are particularly common in complicated tables. In some cases, detecting them may be a necessary pre-characteristic for creating structure data. In other cases, it will at a minimum provide an extra check on the accuracy of the classification.

Layout Hierarchies

- **Necessary**: In one of our publications, how text is wrapped within tables varies based on the section of the table it is in (i.e. a personnel section with names wraps text differently than a section with financial variables)
- **Very useful**: Hierarchical layouts can provide an additional check that is useful for ensuring very high accuracy in classification. For example, only company names and company type tokens should appear in the company title row. Any other combination denotes an error. If this happens with any frequency, we know we need to label more data or enhance accuracy in some other way.

Creating Domains

- Since the original documents follow strict rules and we are able to classify layouts with high accuracy - especially once we remove discrepancies by comparing column and segment level outputs - we can use simple rules to create variable domains.
- Note this would not work well if you had lots of errors in the layout class predictions.

Creating Domains

We store the structured output as a json file, from which we can easily build a CSV file later with the information required for a given analysis. There are five first level hierarchy elements:

1. Company title
2. Company type
3. Address
4. Balance sheet
5. Variables (non-balance sheet information)

Creating Domains

- The company title, company type, and address fields simply consist of strings.
- The variables field consist of a list of tuples. Each 3-tuple contains the variable key (OCR'ed name), tag (corrected variable name), and values. The values may themselves be a list. For example, the board of directors key could contain a list of 2, 5, or 20 names, depending on the nature of the firm.
- A table domain starts when a company title column appears, and continues until another company title appears. Variables work similarly.
- Tables span multiple rows, and it is straightforward to stitch domains together across rows.

Outline

Road Map for the NLP Section of the Course

The course website has an updated syllabus for the NLP portion of the course:

Natural Language Processing	
Mar 10	Models of Words (word2vec, GLoVe)
Mar 15	Dependency Parsing; Language Models (N-Grams, RNN/LSTM review, GRU)
Mar 17	Sequence to Sequence Learning
Mar 22	The Transformer
Mar 24	Transformer-Based Models
Mar 29	What's in an Embedding; Sentiment Analysis
Mar 31	NO CLASS (Wellness Day)
Apr 5	Retrieval and Question Answering
Apr 7	Zero-Shot and Few-Shot Learning in NLP
Apr 12	Natural Language Generation; Summarization (time permitting)
Apr 14	NLP on Noisy Data

Course Projects

- The final two weeks of the course will be devoted to presenting the class projects
- Please start getting your hands dirty with coding and implementation now. This will provide time to adjust for any deficiencies in your coding background and adjust the scope of the project if it ends up (as it typically does) being more ambitious than initially anticipated