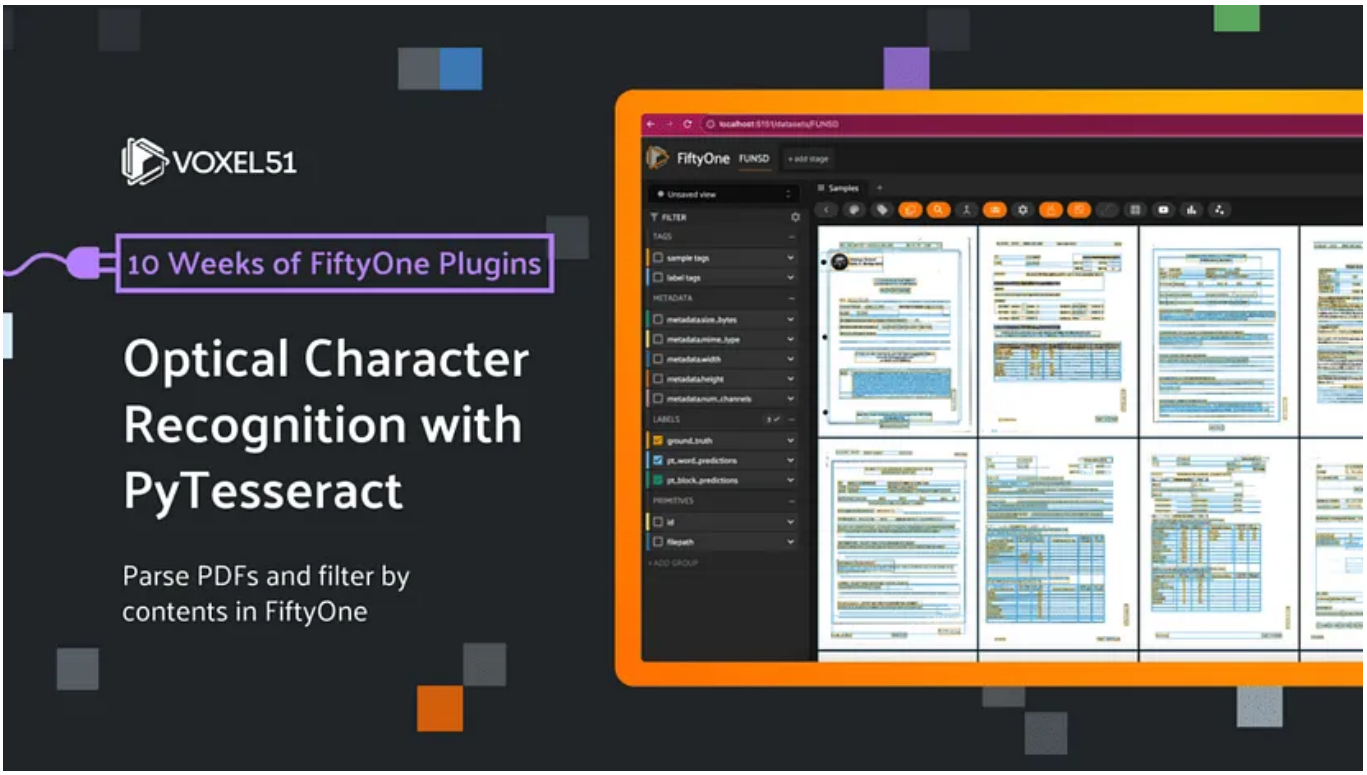
Optical Character Recognition with PyTesseract

Jacob Marks, Ph.D. · Follow

Published in Voxel51 · 10 min read · Sep 21, 2023

Parse PDFs and filter by contents in FiftyOne



Welcome to week five of *Ten Weeks of Plugins*. During these ten weeks, we will be building a FiftyOne Plugin (or multiple!) each week and sharing the lessons learned!

If you're new to them, FiftyOne Plugins provide a flexible mechanism for anyone to extend the functionality of the FiftyOne computer vision App. Similar to how you customize your favorite IDE to make quick work of repetitive or novel tasks, this is what Plugins are all about.

If you are just joining us, you may find the following resources helpful:

- FiftyOne Plugins Repo
- FiftyOne Plugin Docs
- Plugins Channel in the FiftyOne Community Slack

If you've have been following along the last few weeks, let's recap what we've built so far:

- Week 0: 🌩️ Image Quality Issues & 📈 Concept Interpolation
- Week 1: 🎨 AI Art Gallery & Twilio Automation
- Week 2: ❓ Visual Question Answering
- Week 3: 📽️ YouTube Player Panel
- Week 4: 💎 Image Deduplication

Ok, let's dive into this week's FiftyOne Plugins — 👓 Optical Character Recognition (OCR) and 🔑 Keyword Search!

OCR++ 👁️ 🔤 👊

Optical Character Recognition (OCR) is a fundamental task in computer vision which entails recognizing the characters in a document, when said document is treated as an image. OCR can be employed to recognize typed characters, handwritten text, or even curved word art, and it has applications across multiple industries, from banking and law to healthcare. An OCR "engine" is the pipeline — either rules-based or powered by a machine learning model — which turns a document into a set of localized text strings.

This week, I set out to streamline OCR and natural language document understanding workflows in FiftyOne! To do so, I built two connected plugins. The first plugin PyTesseract OCR, leverages the popular Tesseract OCR engine to perform optical character recognition, and converts the engine's outputs into `Detection` labels. The second plugin is a Keyword Search plugin which allows you to search within the labels generated by the first plugin. When combined, these two plugins effectively allow you to query documents like pages of old books, handwritten notes, or resumes by the text that they contain!

PyTesseract OCR Plugin Overview

The PyTesseract OCR plugin is essentially a wrapper around the Tesseract OCR engine. The plugin has just one operator, `run_ocr_engine`, which performs OCR on each sample in the dataset and stores the results on the samples. Because this is a Python plugin, it interacts with Tesseract through the engine's Python bindings, exposed by the `pytesseract` library.

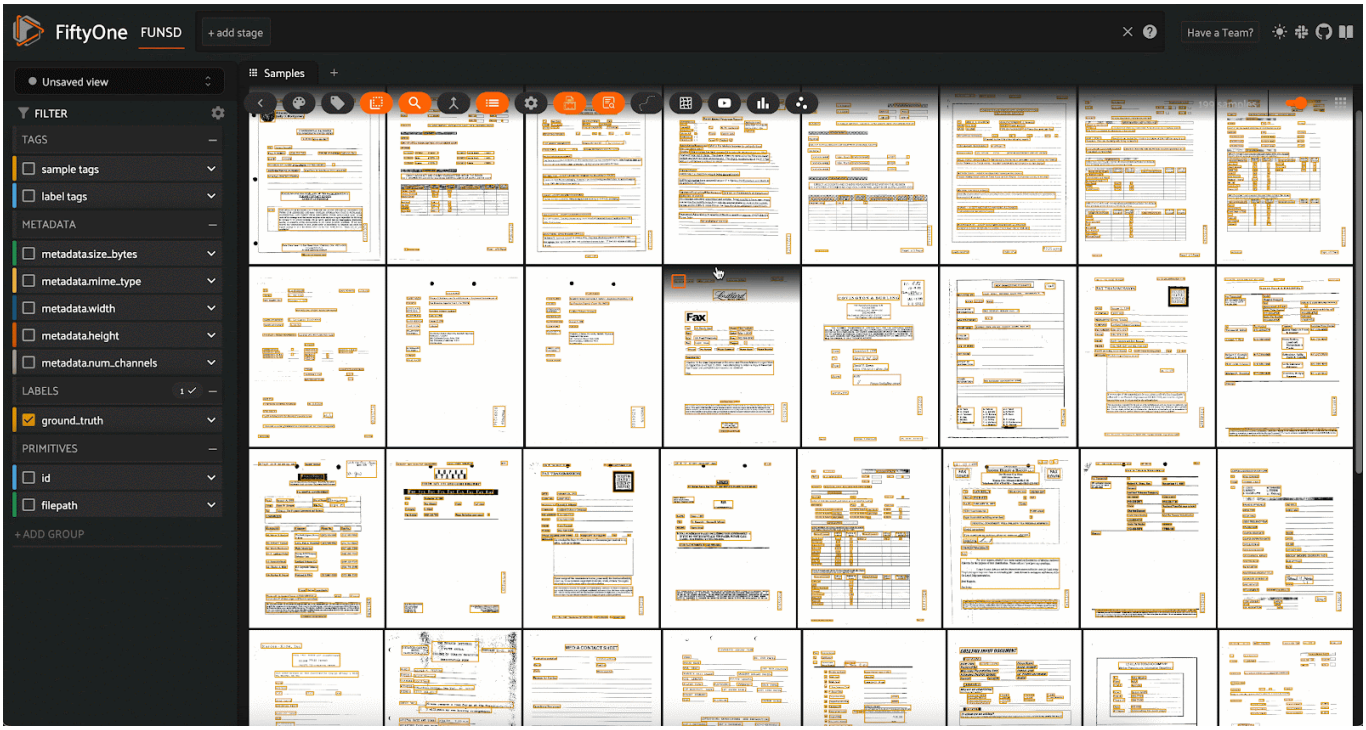
For each sample, this starts by extracting the filepath, and passing this into PyTesseract's `image_to_data` function. This dictionary of data is then parsed

and turned into two sets of detections:

1. Word Detections: a `Detections` field on the sample that contains a separate `Detection`, with a confidence score, for each detected word.
2. Block Detections: a `Detections` field on the sample that aggregates words in each contiguous *block* in the document.

💡 `run_ocr_engine` is a *delegated* operator. Instead of waiting while the OCR engine runs, executing the operator queues a job, which you can start from the command line!

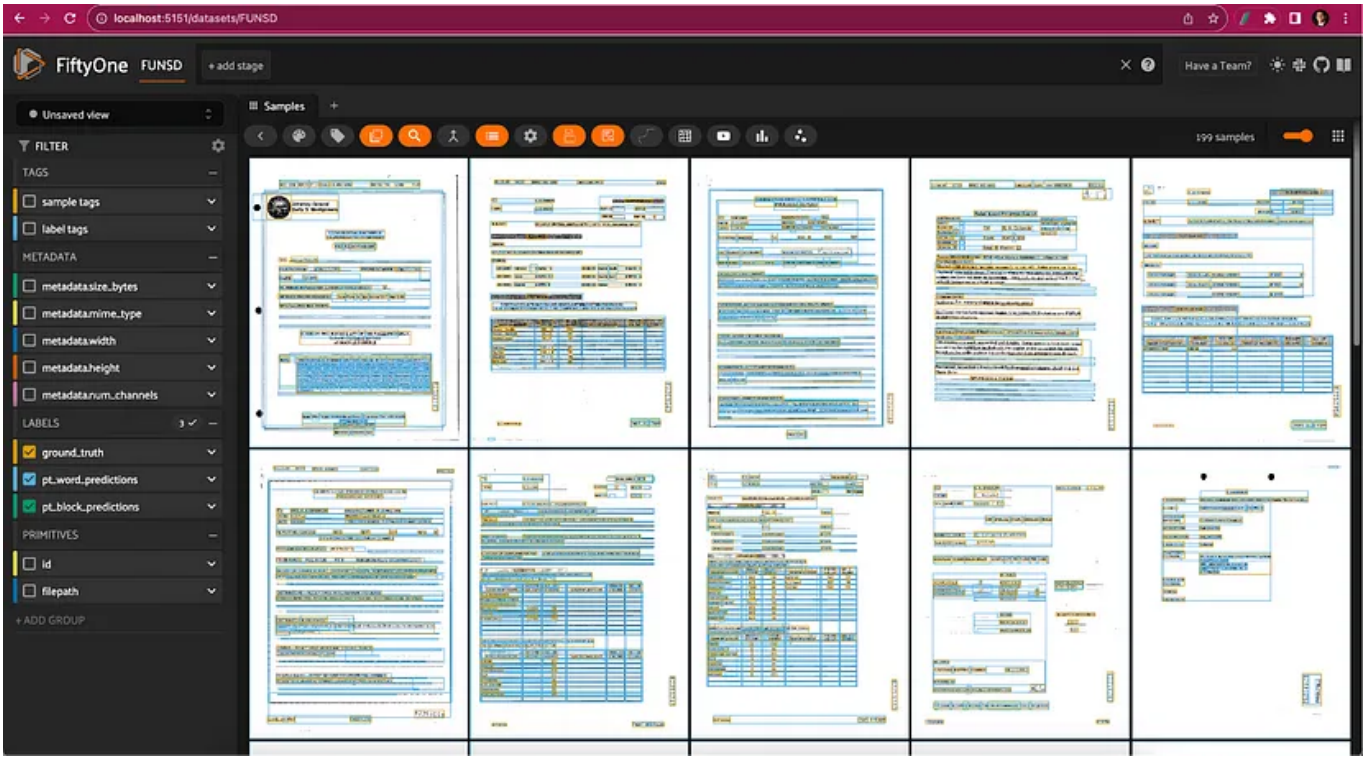
Let's see this in action on the Form Understanding in Noisy Scanned Documents (FUNSD) dataset. First, we queue the job from the FiftyOne App:



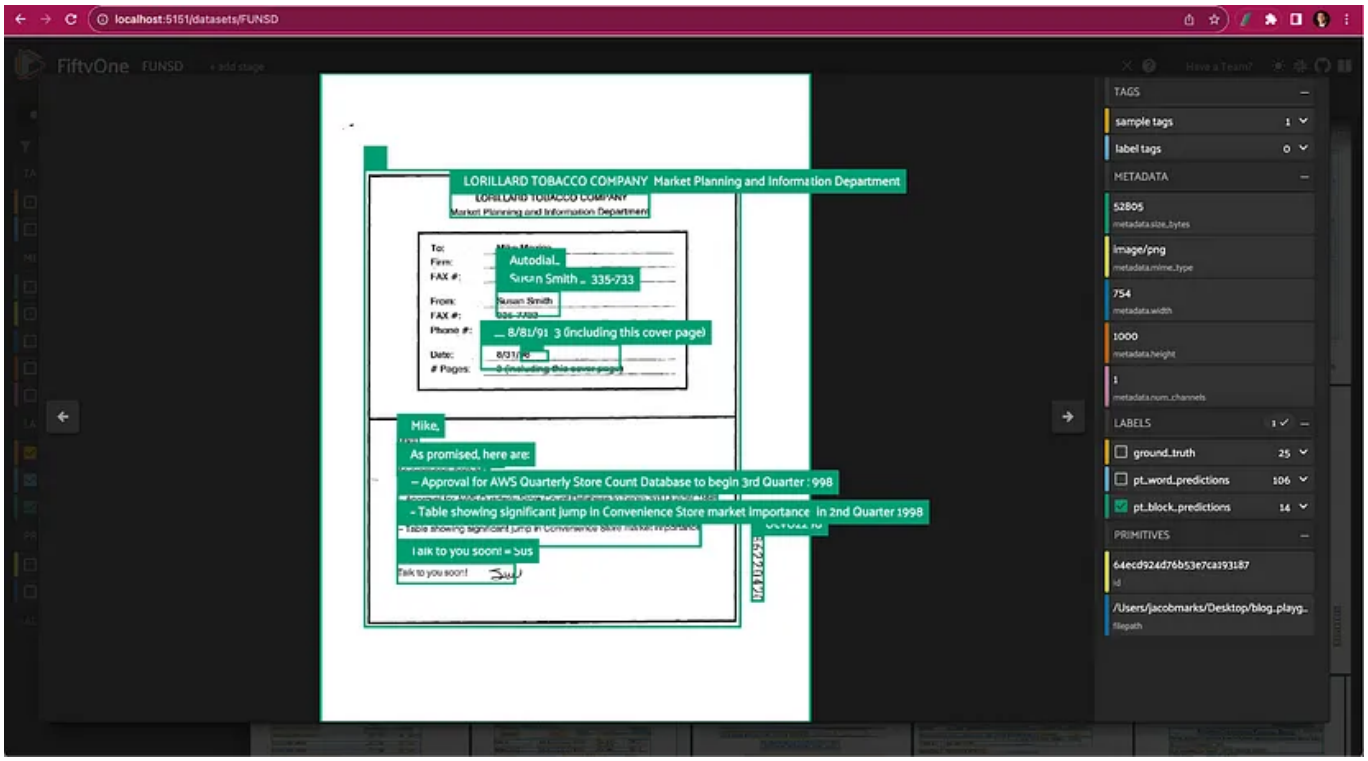
We can then list our queued jobs from the command line with `fiftyone delegated list`:

```
id                        operator                                      dataset
------------------------  --------------------------------------------  ---------
650252f0e262b5c491d50d94  @jacobmarks/pytesseract_ocr/run_ocr_engine  FUNSD
```

To launch the job, we can run `fiftyone delegated launch`. When the job completes, we can refresh the app, and we will see new `pt_word_predictions` and `pt_block_predictions` fields populated on our samples:



This is what the block predictions look like for a single sample:



Keyword Search Plugin Overview

When I read long PDFs for research papers, textbooks, or contracts, I typically find myself relying heavily on keyword search. With a simple control-F, I can find every occurrence of a specific string in the document, across all pages. After running OCR on your documents, being able to search through the generated text seems like a natural thing to do!

To achieve this functionality, I built a Keyword Search plugin. The plugin leverages FiftyOne's `contains_str()` view expression, which tests whether a string field on a sample contains a substring. For example, the following code filters the Quickstart dataset for predictions whose labels contain the string "be":

```
import fiftyone as fo
import fiftyone.zoo as foz
from fiftyone import ViewField as F

dataset = foz.load_zoo_dataset("quickstart")
```

```
# Only contains predictions whose `label` contains "be"
view = dataset.filter_labels(
    "predictions", F("label").contains_str("be")
)
### view will only contain ['bear', 'bed', 'bench', 'frisbee', 'teddy bear']
```

However, the complete syntax for querying the dataset with `contains_str()` depends on the field to which it is applied. For string fields embedded in label fields, such as the `predictions.detections.label` field above, the view stage which achieves a keyword search-like effect would be `match_labels()`. For top-level string fields, on the other hand, the right view stage is just `match()`.

There's an additional level of complexity when considering lists of strings. Take a sample's `tags` field for instance. When we search for a specific keyword, we need to check if any of the strings in the list contain the substring.

The Keyword Search plugin works by finding all string fields and list fields with string elements in the dataset, and letting the user select which of these fields they want to search within. Depending on the type of the selected field, the appropriate querying syntax is used to perform the search. All of this is wrapped in a `search_by_keyword` operator.

The four supported options are:

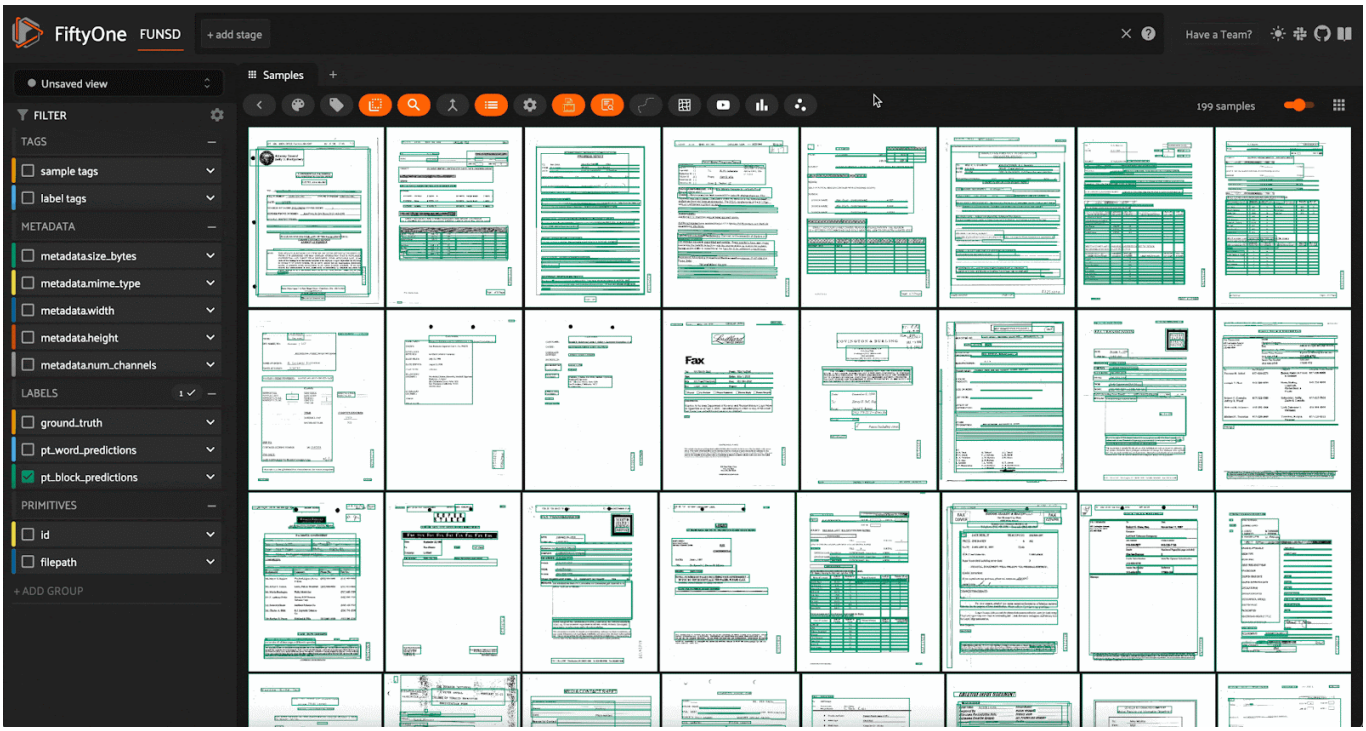
1. Top-level `StringField`
2. `StringField` within a `Label` field

3. Top-level `ListField`, whose elements are strings
4. `ListField` with string elements within a `Label` field

The plugin also exposes the `case_sensitive` argument from `contains_str()` to the user, so they can decide whether they want the search to be performed in a case sensitive or case insensitive fashion.

We can apply this `search_by_keyword` operator to the label field containing our OCR predictions in order to find documents that contain certain keywords!

Here's an example where we filter for documents in the FUNSD datasets with the word `contract` in them:



The final element of the Keyword Search plugin worth noting is that it retains a "memory" of which field you last searched within. This saves you

from selecting the same field from the field selector dropdown each time you want to modify your search! More on this in the lessons learned section below.

Installing the Plugins

If you haven't already done so, install FiftyOne:

```
pip install fiftyone
```

You can now download these plugins from the command line with:

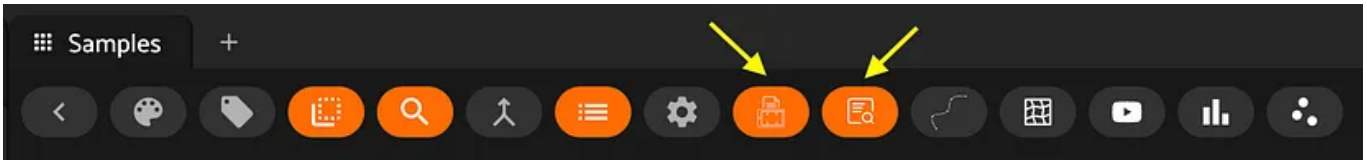
```
fiftyone plugins download https://github.com/jacobmarks/keyword-search-plugin
fiftyone plugins download https://github.com/jacobmarks/pytesseract-ocr-plugin
```

To run the OCR plugin, you will need to have `pllow` and `pytesseract` installed. These are included in the `requirements.txt` file for the OCR plugin, so you can install them by running:

```
fiftyone plugins requirements pytesseract-ocr-plugin --install
```

After downloading the plugins (and installing requirements), refresh the FiftyOne App, and you should see two new buttons in the Sample Actions

Menu:



Buttons for search_by_keyword and run_ocr_engine operators

You will also find these operators in the operators list when you press the "`" key.

Lessons Learned

Both the OCR and Keyword Search plugins are pure Python plugins with the same basic structure:

- `__init__.py` : operators are defined
- `fiftyone.yml` : plugin information is defined and registered
- `README.md` : plugin and operators are described, and installation instructions are documented.
- `assets` folder: operator icons are stored

In addition, the Keyword Search plugin has a cache manager file, `cache_manager.py` , whose purpose will be described shortly, and the OCR plugin has an `ocr_engine.py` file, which handles the running and postprocessing of data from the PyTesseract OCR engine.

Caching in Python Plugins

When building the VoxelGPT plugin (a chat-based coding assistant for the FiftyOne App leveraging LLMs), one of the key considerations was latency. When you chain multiple prompts and large language model queries together, the time elapsed between user question and final response can add up quickly. One of the many techniques we used to minimize latency was aggressively *caching*. For VoxelGPT, we did this to minimize time associated with reading in files — if you aren't changing the contents of the file, but you are going to use the data often, why not store it in a global cache so you only need to read it in once.

For the Keyword Search plugin, I set out to use caching for a completely different purpose — remembering a user's choices. Once the user selects a field which they want to perform the keyword search on, this field becomes our best guess for the field on which the user would want to perform their next keyword search. Instead of making the user select this same field from the dropdown selector each time they open the operator's modal, why not cache the last-used field and set this as the default value?

As I found out while building the Keyword Search plugin, however, there is some nuance required to achieve this effect. First off, you can't do any caching within the `__init__.py` file itself, because this file gets reloaded and its variables reset every time the operator is used. Instead, the solution I came up with was to create a `cache_manager.py` module which exclusively defines a `get_cache()` function, and then import this function from the module in `__init__.py`.

The second subtlety is that caching a user's preference in this manner only works if the application is being served to a single user via a single node. If this approach were used on try.fiftyone.ai, where each interaction is potentially fulfilled by a different instance of the plugin on a different node,

there would be no way of determining which user each preference should be associated with. This is why the import statement for the cache manager in `__init__.py` is wrapped with an if statement that checks if the code is running on a multi-user deployment:

```
def _is_teams_deployment():
    val = os.environ.get("FIFTYONE_INTERNAL_SERVICE", "")
    return val.lower() in ("true", "1")

TEAMS_DEPLOYMENT = _is_teams_deployment()

if not TEAMS_DEPLOYMENT:
    with add_sys_path(os.path.dirname(os.path.abspath(__file__))):
        # pylint: disable=no-name-in-module,import-error
        from cache_manager import get_cache
```

The key takeaway here is that true statefulness within a plugin is only possible with JavaScript plugins!

Menu Buttons in Python Plugins

You don't need to turn your Python plugin into a JavaScript plugin to create sleek buttons for your operators. On the contrary, you can actually specify these in the operator's definition with the `resolve_placement()` method.

This is all the code required to turn the `run_ocr_engine` operator into a nice button:

```
def resolve_placement(self, ctx):
        return types.Placement(
            types.Places.SAMPLES_GRID_ACTIONS,
            types.Button(
                label="Detect text in images",
```

```
                icon="/assets/icon_light.svg",
            ),
        )
```

The first input to types.Placement(), `SAMPLES_GRID_ACTIONS`, is what determines where the button shows up. You can find more info on the other allowed placements here. As in the operator's config, we specify the icon to render with the `icon` argument. There is no requirement that the icon you use for the operator's config (what shows up in the operators list) and the icon that you use for the button are the same, or even related. Hypothetically, you could make them completely different!

Plugging in your Plugins

As I continue to develop more and more plugins, I find myself increasingly thinking about how the plugins will, well, plug into each other. Will the outputs of one plugin be suitable as inputs into another plugin? In other words, will my plugins play nice with each other?

> *A large portion of the value inherent in plugins is the ability to interweave multiple plugins to construct sophisticated, tailored workflows. And as the FiftyOne plugin ecosystem grows, the space of enabled workflows grows exponentially. To make the most of this interconnectivity, plugins need to be modular and flexible.*

The Keyword Search plugin arose from a specific problem: I wanted to be able to search through the contents of the documents on which I had run OCR. In theory, this only required the ability to search through `String` field attributes of `Detection` labels. However, it wasn't a huge stretch for me to imagine other workflows involving keyword search on lists of strings, or lists

of strings within `Detection` label fields. The plugin that I built is flexible enough to plug into any/all of these workflows!

I hope this approach proves helpful as you build out your arsenal of FiftyOne plugins!

Conclusion

By combining OCR with Keyword Search, you can filter PDFs or other text-based documents by their content, in the same way that you would filter for images that have dogs or cats. These two plugins are certainly not exhaustive, but I hope they demonstrate how you can approach natural language based document understanding with the visualization and querying capabilities of FiftyOne!

Stay tuned over the remaining weeks in the *Ten Weeks of FiftyOne Plugins* while we continue to pump out a killer lineup of plugins! You can track our journey in our ten-weeks-of-plugins repo — and I encourage you to fork the repo and join me on this journey!

What's Next?

Join the thousands of engineers and data scientists already using FiftyOne to solve some of the most challenging problems in computer vision today!

- ⭐ Star the FiftyOne GitHub repo (4,000+ stars and counting)
- ⭐ Star the FiftyOne Plugins repo on GitHub
- 📚 Check out the FiftyOne plugin docs
- 🎉 Join the 2,000-strong Slack community of ML engineers and data scientists, and check out the #plugins channel!

- 🎉 Join 5,000+ in the Computer Vision meetup network — and stay tuned for our upcoming workshop on plugins!

Originally published at https://voxel51.com on September 21, 2023.

Computer Vision | AI | Artificial Intelligence | Machine Learning | Data Science

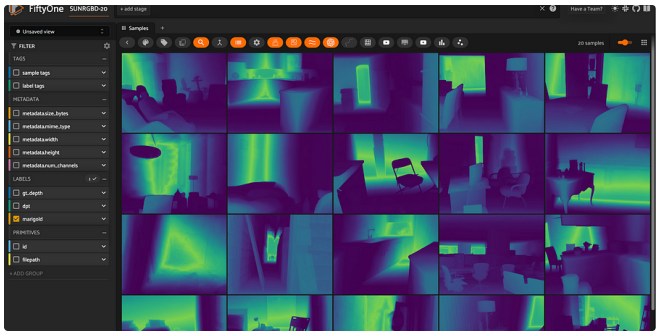
Written by Jacob Marks, Ph.D.

Follow

7.4K Followers · Editor for Voxel51

ML @ Voxel51 | Ex-Google X, Wolfram Research | Stanford Theoretical Physics PhD
https://www.linkedin.com/in/jacob-marks

More from Jacob Marks, Ph.D. and Voxel51



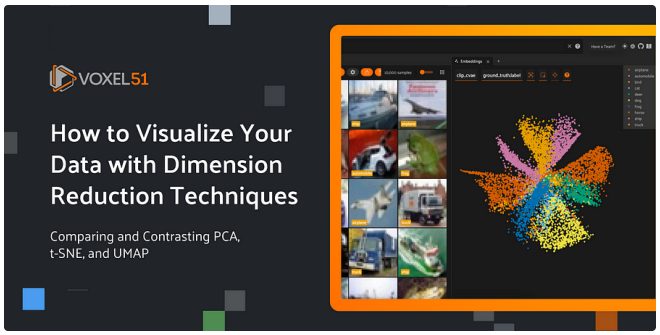
Jacob Marks, Ph.D. in Towards Data Science

How to Estimate Depth from a Single Image

Run and Evaluate Monocular Depth Estimation Models with Hugging Face and...

10 min read · Jan 26, 2024

664 2



Jacob Marks, Ph.D. in Voxel51

How to Visualize Your Data with Dimension Reduction Techniques

Comparing and Contrasting PCA, t-SNE, and UMAP

9 min read · Jan 31, 2024

319 2

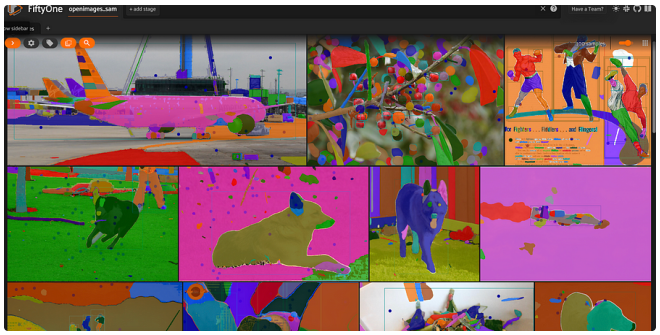
Jacob Marks, Ph.D. in Voxel51

Why 2023 was the most exciting year in computer vision history (s...

The 10 developments that reshaped computer vision

11 min read · Dec 20, 2023

426 3



Jacob Marks, Ph.D. in Towards Data Science

See what you SAM

Generate and visualize Segment Anything Model predictions

10 min read · May 3, 2023

654

See all from Jacob Marks, Ph.D. See all from Voxel51

Recommended from Medium

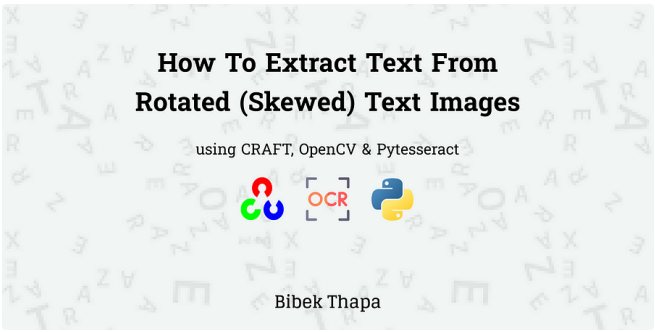
Mehmet Çağrı Çalpur

OCR with Vision Transformers

Vision transformers are disrupting the conventional visual tasks such as...

3 min read · Jan 17, 2024

95



Bibek Tha... in Artificial Intelligence in Plain Engli...

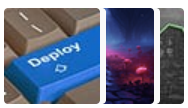
How to Extract Texts from Rotated(Skewed) Text-Images...

I as an AI engineer was tasked with creating a Module that would extract texts from...

14 min read · Oct 17, 2023

105

Lists



Predictive Modeling w/ Python

20 stories · 918 saves

Natural Language Processing

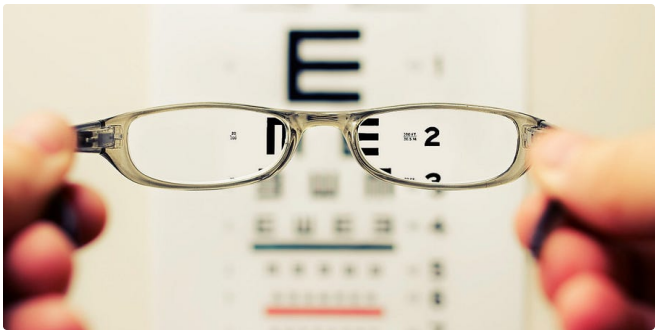
1205 stories · 686 saves

Practical Guides to Machine Learning

10 stories · 1081 saves

The New Chatbots: ChatGPT, Bard, and Beyond

12 stories · 307 saves



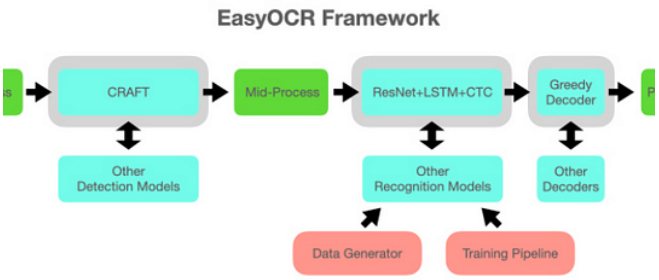
Vinod Baste

Unlocking the Power of PaddleOCR

An Introduction to Text Detection and Recognition

8 min read · Sep 20, 2023

85



Aditya Mahajan

EasyOCR: A Comprehensive Guide

Detailed explanation of EasyOCR with usage examples

11 min read · Oct 28, 2023

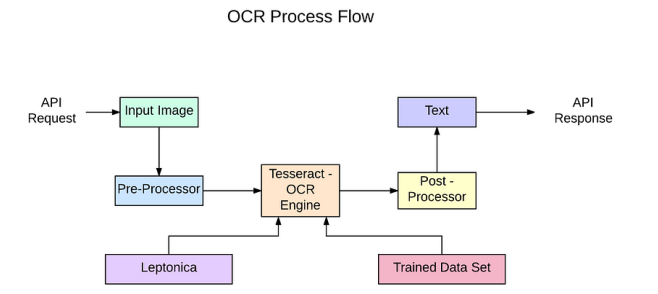
156

Yohannes Abrha

Text Extraction and Parsing from Contemporary Maps by Leveragin...

Maps serve as invaluable tools for conveying geographical data and facilitating decision-...

7 min read · Oct 4, 2023



Riwaj Neupane

OCR with PyTesseract and EasyOCR

Tesseract is an open source text recognition (OCR) Engine, available under the Apache 2....

4 min read · Jan 15, 2024

See more recommendations