BoonAI Python SDK

Introduction

What is Boon AI?

Boon AI is a platform for building custom AI augmented media search solutions for images, video, and documents.

What can Boon AI do for you?

- Provide a sandbox for testing AI based solutions from a variety of vendors.
- Automate content moderation, object detection, document classification, and speech transcription using state of the art AI.
- Execute deep searches of AI generated results using a combination of keywords, faces, and even similar images.
- Help you train your own AI models to classify media with your own keyword taxonomy.

In which sectors can Boon AI be helpful?

- Social media - Content moderation
- Sports - Face recognition
- Media and Entertainment - Deep video timeline search
- E-Commerce - Product search by color, image similarity, keywords, facets

Installation

Installing the Boon AI SDK

The Boon AI SDK allows you to import, analyze, and search visual data using all of the available Analysis Modules. Anyone can install the Boon AI SDK but you will need an API key to make use of it.

```
$ pip install boonsdk
```

Now you can import boonsdk in your python code.

```
import boonsdk
```

Python Module Documentation

BoonSDK Documentation — BoonSDK 1.4.7 documentation https://python.docs.boonai.app/

Installing the Boon AI Lab

You can use the import statement above to use the Boon AI SDK in a Jupyter Notebook environment as is, but an alternate approach is to install the Boon AI Lab. The Boon AI Lab package provides a number of convenience functions to work with asset proxies, plot graphs using analysis results, and other tasks.

```
$ pip install boonlab
```

Generating an API Key

To obtain access to Boon AI API resources, you must create a Boon AI API key with a limited set of permissions.

Creating an API Key

To create an API key, perform these steps:

1. Using your Google account, log on to https://boonai.app.
2. On the **Account Overview** screen, under **QUICK LINKS**, click the **API Keys** icon .
3. On the **Project API Keys** screen, click **CREATE API KEY**.
4. In the **Name** field, enter a name for the API key.
5. Under **Add Scope**, select one or more permissions.
6. Click **Generate Key & Download**.
7. On the next **Project API Keys** screen, under **API Key**, perform one or both of these steps:
    1. Click **Copy Key**.
    2. Click **Download** to store the key in a file on your local machine.
       The name of the API key file is *name.json*, where *name* is the text that you entered in step 4.

Using the API Key

To use the API key, perform one of these procedures:

- Provide a path to the API key file
- Paste the key into the code
- Import a Base64 version of the API key
- Import a Base64 version that is stored in the `BOONAI_APIKEY` environment variable

The following example code imports the API key as a file.

Python

```python
import boonsdk

app = boonsdk.app_from_keyfile("/path/to/my-apikey.json"):
```

The following example code imports the API key that was pasted into the code.

Python

```python
import boonsdk

app = boonsdk.BoonApp({
    "accessKey": "UT3rW1J68QCCkddiPuHOXg",
    "secretKey": "wueiEHo44hVK3H1nPkzh1g"
})
```

The following example code imports the API key as a Base64 version. You can convert your API key into base64 simply by encoding the JSON as Base64. This is useful for embedding the key as an environment variable.

Python

```python
import boonsdk

app = boonsdk.BoonApp(b'eyJhY2Nlc3NLZXkiOiJVVDNyVzFKNjhRQ0NrZGRpUHVI
```

The following example code imports the API key as a Base64 version that is stored in the `BOONAI_APIKEY` environment variable and uses the `boonai.app_from_env()` function to

load the `BoonAiApp` instance.

Python

```python
import os
import boonsdk

os.environ['BOONAI_APIKEY'] = 'eyJhY2Nlc3NLZXkiOiJVVDNyVzFKNjhRQ0NrZ

app = boonsdk.app_from_env()
```

Viewing API Keys

To view all of your API keys, perform one of these steps in the Boon AI web app:

- On the **Project API Keys** screen, click the **View All** tab.
- On the Boon AI Console, in the upper left corner of any screen, click the menu button and then click **API Keys**.

Analysis Modules

Analysis modules are the building blocks for creating a file import pipeline. You can apply analysis modules at any time, either during the file import process or after assets have been imported.

You can get a list of existing analysis modules using the Boon AI Client, as shown in the following example code, or you can refer to the list of analysis modules in this document.

Python

```python
import boonsdk

app = boonsdk.BoonSdkApp('BOONAI-APIKEY')
app.analysis.find_analysis_modules()
```

Analysis Module Results

Once an analysis module is applied, obtain the results of the module by passing its name to the `get_analysis` function on an asset. For example, you can use the following example code to get the results of the `gcp-label-detection` module.

Python

```python
asset.get_analysis("gcp-label-detection")
```

Analysis Modules Data Structure

Analysis modules structure the data written to an asset in three ways:

- Array of predictions
- Content
- Single prediction

Array of Predictions

Any type of convolutional neural network (CNN)-based classifier module, such as label detection, face detection, or object detection, outputs an array of predictions. Each prediction, at a minimum, has a label and a score. Some types of predictions also have supplemental data, such as a bounding box, as shown in the following example code.

Python

```python
print(asset.get_analysis("gcp-object-detection"))

{
    "type" : "labels",
    "predictions": [
        {
            "label": "cat",
            "score": 0.55,
            "bbox": [0.1, 0.1, 0.5, 0.5]
        },
        {
            "label": "dog",
            "score": 0.55,
            "bbox": [0.9, 0.9, 1.0, 1.0]
        }
    ]
}
```

Content

Analysis modules that perform OCR or text detection store their result as a blob of text, as shown in the following example code.

Python

```python
print(asset.get_analysis("boonai-text-detection"))

{
    "type" : "content",
    "words": 9,
    "content": "The quick brown fox jumps over the lazy dog"
}
```

Single Prediction

Any type of K-Nearest Neighbor (KNN)-based classifier module outputs a single prediction, as shown in the following sample code. You can only create this type of analysis module by training a custom model.

Python

```python
print(asset.get_analysis("boonai-custom-model"))

{
    "type": "single-label",
    "label": "cat",
    "score": 0.55
}
```

Supported Analysis Modules

This section lists the analysis modules that are supported by the Boon AI system. The modules are divided into these ML operation types:

- Asset clipifier
- Explicit detection
- Face detection
- Face recognition
- Image description
- Image text detection
- Label detection
- Landmark detection
- Logo detection
- Object detection
- Speech recognition
- Text detection

Naming Convention

The standard naming convention for analysis modules is

<service>-<machine learning operation type>

For example, boonai-face-detection is the analysis module from Zorroa that detects faces in images and documents.

Related APIs

Analysis modules from certain vendors use related APIs.

Vendor Category	API

Amazon Rekognition	Amazon Rekognition
Google Vision and Google Video Intelligence	Google Cloud Vision API
Microsoft Azure	Microsoft Azure Vision APIs

Analysis Modules

Asset Clipifier

Module	Description	Vendor Category
`boonai-extract-layers`	Extracts all layers in multilayer or multipage image formats, such as TIFF and PSD, as separate assets	Zorroa Timeline Extraction
`boonai-extract-pages`	Extracts all pages in Microsoft Office and PDF documents into separate assets	Zorroa Timeline Extraction

Explicit Detection

Module	Description	Vendor Category
`aws-unsafe-detection`	Identifies potentially unsafe or inappropriate content	Amazon Rekognition
`azure-explicit-detection`	Detects adult material in images so that developers can restrict the display of these images in their software	Microsoft Azure Computer Vision

`clarifai-nsfw-detection`	Identifies different levels of nudity in visual content and automatically moderates or filters offensive content	Clarifai Public
`clarifai-unsafe-detection`	Detects whether an image contains concepts such as gore, drugs, explicit nudity, or suggestive nudity	Clarifai Public
`gcp-video-explicit-detection`	Detects explicit content in videos	Google Video Intelligence

Face Detection

Module	Description	Vendor Category
`aws-face-detection`	Detects face bounding boxes within an image or document	Amazon Rekognition
`clarifai-celebrity-detection`	Detects whether images contain the face, or faces, of celebrities	Clarifai Public
`clarifai-face-detection`	Detects whether an image contains human faces and coordinates locations of those faces with a bounding box	Clarifai Public

Face Recognition

Module	Description	Vendor Category
`aws-celebrity-detection`	Recognizes thousands of celebrities in a wide range of categories	Amazon Rekognition

`azure-celebrity-detection`	Identifies well-known people in video and image libraries for cataloging of footage and photographs for marketing, advertising, and media industry use cases	Microsoft Azure Computer Vision
`azure-face-detection`	Detects human faces and attributes in an image	Microsoft Azure Computer Vision
`boonai-face-detection`	Detects faces in images and documents	Zorroa Visual Intelligence

Image Description

Module	Description	Vendor Category
`azure-image-description-detection`	Analyzes an image and generates a human-readable sentence that describes its contents	Microsoft Azure Computer Vision

Image Text Detection

Module	Description	Vendor Category
`aws-text-detection`	Detects text in an image using Amazon Rekognition	Amazon Rekognition
`gcp-image-text-detection`	Performs OCR and detects printed and handwritten text; generates keyword labels for images and documents	Google Vision

Label Detection

Module	Description	Vendor Category
`aws-label-detection`	Identifies objects, locations, activities, animal species, products, and more in a video; generates keyword labels for images and documents	Amazon Rekognition
`azure-category-detection`	In addition to tags and a description, generates the taxonomy-based categories detected in an image	Microsoft Azure Computer Vision
`azure-label-detection`	Identifies and tags visual features in an image from a set of thousands of recognizable objects, living things, scenery, and actions	Microsoft Azure Computer Vision
`clarifai-apparel-detection`	Detects items of clothing or fashion-related items	Clarifai Public
`clarifai-food-detection`	Recognizes more than 1,000 food items and dishes in images down to the ingredient level	Clarifai Public
`clarifai-label-detection`	Recognizes over 11,000 concepts, including objects, themes, moods, and more	Clarifai Public
`clarifai-texture-detection`	Identifies textures and patterns in an image, including glacial, ice, metallic, veined, feathered, and more	Clarifai Public
`clarifai-travel-detection`	Recognizes specific features of residential, hotel, and travel-related properties	Clarifai Public
`clarifai-wedding-detection`	Recognizes over 400 concepts related to weddings, including bride, groom, flowers, and more	Clarifai Public

`gcp-label-detection`	Detects and extracts information about entities in an image across a broad group of categories	Google Vision
`gcp-video-label-detection`	Identifies objects, locations, activities, animal species, products, and more in a video	Google Vision
`boonai-label-detection`	Classifies images or documents using an AI model trained on the ImageNet dataset; provides support for 1,000 categories	Zorroa Visual Intelligence

Landmark Detection

Module	Description	Vendor Category
`azure-landmark-detection`	Detects popular natural and man-made structures in an image	Microsoft Azure Computer Vision
`gcp-landmark-detection`	Detects popular natural and man-made structures in an image	Google Vision

Logo Detection

Module	Description	Vendor Category
`azure-logo-detection`	Identifies commercial brands using brand detection, a specialized mode of object detection that uses a database of thousand of global logos	Microsoft Azure Computer Vision
`clarifai-logo-detection`	Identifies up to 500 company brands and logos	Clarifai Public

`gcp-logo-detection`	Detects popular product logos in an image	Google Vision
`gcp-video-logo-detection`	Tracks and recognizes the presence of over 100,000 brands and logos in video content	Google Video Intelligence

Object Detection

Module	Description	Vendor Category
`azure-object-detection`	Identifies and tags visual features in an image with a bounding box from a set of thousands of recognizable objects and living things	Microsoft Azure Computer Vision
`gcp-object-detection`	Detects and extracts multiple objects in an image	Google Vision
`gcp-video-object-detection`	Detects individual objects and provides a label along with a bounding box that describes the location in the frame for each object	Google Video Intelligence
`boonai-object-detection`	Detects everyday objects in images and documents	Zorroa Visual Intelligence

Speech Recognition

Module	Description	Vendor Category

`aws-transcribe`	Converts speech to text quickly and accurately using Amazon Transcribe, which uses a deep learning process called automatic speech recognition (ASR)	Amazon Transcribe
`gcp-speech-to-text`	Transcribes spoken word audio in a video or video segment into text and returns blocks of text for each portion of the transcribed audio; support for 120 languages	Google Speech-to-Text
`gcp-video-speech-transcription`	Converts audio to text by applying powerful neural network models	Google Video Intelligence

Text Detection

Module	Description	Vendor Category
`azure-text-detection`	Extracts printed or handwritten text from images using OCR capabilities	Microsoft Azure Computer Vision
`gcp-dlp`	Extracts names, dates, and addresses from scanned documents using data loss prevention (DLP)	Google Data Loss Prevention
`gcp-document-text-detection`	Uses OCR technology to detect and transcribe text from PDF and TIFF files located in cloud storage	Google Vision
`gcp-video-text-detection`	Performs OCR to detect visible text from frames in a video, or video segments, and returns the detected text along with information about the frame-level location and timestamp in the video for that text	Google Video Intelligence

`boonai-text-detection`	Extracts text from an image or images in a document using the Tesseract OCR library	Zorroa Visual Intelligence

Importing Assets

Assets are created by importing media files into Boon AI.

A single file can result in the creation of a single asset or many assets, depending on the analysis modules that were applied when the files were imported. Each Asset contains all of the AI-generated metadata for a particular section of a file, such as a page of a PDF document or a scene from a video.

Importing a Remote File Resource

You can use the `FileImport` class to import supported file types into Boon AI from a remote file resource.

You may use one of these publicly available URIs:

- HTTP link *(http://, https://)*
- Google Cloud Storage (GCS) *(gs://)*
- Amazon Web Services (AWS) *(s3://)*
- Microsoft Azure Storage (Azure)

You can use the following example code to import the data from a remote location.

Python

```python
import boonsdk
from boonsdk.asset import FileImport

app = boonsdk.BoonApp(apikey='BOONAI-APIKEY')

files = [
    FileImport('https://i.imgur.com/WkomVeG.jpg')
    FileImport('gs://zorroa-dev-data/image/pluto_2.1.jpg')
]

app.assets.batch_import_files(files)
```

The JSON response provides an asset's ID failure status. The example below shows that no image imports failed and one was created. The value in the array represents that asset's ID.

JavaScript

```
{
  "failed" : [ ],
  "created" : [ "hWsGI5FlI2HRdbT6EzJ3HqF3IA0WBx2_" ],
  "exists" : [  ],
  "jobId" : "30f3196c-050d-128c-a83f-dad16ffe76a1"
}
```

Importing from a CSV File

One of the most convenient ways to import a Assets is by using a CSV file. This is especially true when the CSV file contains data for custom fields or labels that should be imported at the same time.

Basics

The `CsvFileImport` class works by parsing the CSV file and submitting batches of files for import. Let's assume we have a basic CSV file with two fields, one field for the URI of the file and another field which is some data we want to put into a custom field.

Example CSV File

product_id	product_image
abcd1234	https://somewebsite.com/products/abcd1234.jpg
abcd9876	https://somewebsite.com/products/abcd9687.jpg

```
import boonsdk

app = boonsdk.app_from_keyfile("apikey.json")

# Define our CSV file.
csvfile = boonsdk.CsvFileImport(
    "products.csv", # The name of the file
    uri_column=1, # The column of the URI
    field_map={"product_id": 0}, # A dict of custom field name to column i
    header=True) # True if the CSV file has a header

# Inspect how our CSV file was parsed
for item in csvfile:
    print(item.uri)

# Finally import the contents of the CSV.
app.assets.import_csv(csvfile)
```

Using Labels in the CSV File

Often times CSV files contain both the files, training labels, and even a determination of a label is for testing or training (The label scope). The `CsvFileImport` class supports a multitude of ways to detect label scope.

Example CSV File

Scope	Label	URI
test	cat	http://test-example.com/cat01.jpg
train	cat	http://test-example.com/cat02.jpg
test	dog	http://test-example.com/dog01.jpg
train	dog	http://test-example.com/dog02.jpg

Now let's check out the code. In this case our CSV file uses the words "test" and "train" which will work out of the box. (case insensitive)

```
import boonsdk

app = boonsdk.app_from_keyfile("apikey.json")

# Get an existing Dataset, you can also create one.
dataset = app.datasets.get_dataset("pets")

# Define our CSV file.
csvfile = boonsdk.CsvFileImport(
    "pets.csv", # The name of the file
    uri_column=2, # The column of the URI
    dataset=dataset, # The dataset.
    label_column=1, # The column where the label is stored
    label_scope=0, # If label scope is an int, it's assumed to be col name
    header=True) # True if the CSV file has a header

# Inspect how our CSV file was parsed
for item in csvfile:
    print(item.uri)

# Finally import the contents of the CSV.
```

```
app.assets.import_csv(csvfile)
```

Dynamic Label Scope Assignment

Sometimes you might come across a CSV file with labels but no indication on if the label is a Train or Test label. In this case you can use dynamic scope assignment by passing in a floating point value that represents the percentage of labels to make test labels to the `label_scope` argument. For example 0.2 mean 20% of the assets would be labeled Test instead of Train.

```
# Define our CSV file.
csvfile = boonsdk.CsvFileImport(
    "pets.csv", # The name of the file
    uri_column=2, # The column of the URI
    dataset=dataset, # The dataset.
    label_column=1, # The column where the label is stored
    label_scope=0.2, # roughly 20% of the assets will be labeled TEST
    header=True) # True if the CSV file has a header
```

Converting Non-Standard Scope Values

Let's say our CSV file didn't use "train" or "test" to determine label scope, let's pretend that if the column value is 'y' then it's train and 'n' means test. This is easy to handle by setting the `label_scope` argument to a callable that returns the correct `LabelScope` enum.

```

def determine_label_scope(row):
    """Determine LabelScope a value in the row"""
    value = row[10]
    if value == "y":
        return boonsdk.LabelScope.TRAIN
    else:
        return boonsdk.LabelScope.TEST

# Define our CSV file.
csvfile = boonsdk.CsvFileImport(
    "pets.csv", # The name of the file
    uri_column=2, # The column of the URI
```

```
    dataset=dataset, # The dataset.
    label_column=1, # The column where the label is stored
    label_scope=determine_label_scope, # function will set scope
    header=True) # True if the CSV file has a header


```

Uploading Local Files

You can use the `FileUpload` class to upload supported file types directly to Boon AI from a local machine.

You can use the following example code to upload files from a local hard drive.

Python

```python
import boonsdk
from boonsdk import FileUpload

app = boonsdk.BoonApp(apikey='BOONAI-APIKEY')

files = [
    FileUpload("/Documents/1040.tiff"),
    FileUpload("/Documents/1099-MISC.tiff"),
    FileUpload("/Documents/W2.pdf")
]

app.batch_upload_files(files,
    modules=["boonai-label-detection"])
```

The JSON response provides an asset's ID failure status.

JavaScript

```javascript
{
  "failed" : [ ],
  "created" : [ "hWsGI5FlI2HRdbT6EzJ3HqF3IA0WBx2_" ],
  "exists" : [  ],
  "jobId" : "30f3196c-050d-128c-a83f-dad16ffe76a1"
}
```

Creating a Data Source

A data source is a combination of two things: a remote repository for source files and a set of analysis modules to apply to them.

To create a data source, you must have a name and the URI of a remote repository hosted by GCS, AWS, or Azure. Optionally, you may specify one or more analysis modules to apply, a file type filter, and access credentials.

You can use the following example code to create a data source with an optional analysis module (`boonai-label-detection`) and file type filters (`images` , `documents`).

Python

```python
import boonsdk

app = boonsdk.BoonApp('BOONAI-APIKEY')

app.datasource.create_datasource(
    "legal-files",
    "gs://my-legal-department/documents",
    modules=["boonai-label-detection"],
    file_types=["images", "documents"]
)

# Import all files using the defaults specified above.
app.datasource.import_files()
```

Access Credentials

Importing data from a remote repository likely requires access credentials. For example, GCS, AWS, and Azure all require access keys to allow access to their storage accounts.

> Boon AI stores access credentials in an encrypted state, and they cannot be retrieved with the external API.

Reprocessing Assets

Boon AI provides powerful batch reprocessing tools for adding, removing, and updating metadata attributes. Reprocessing enables you to perform these tasks:

- Process new files in a data source
- Add analysis modules to assets
- Remove metadata from analysis modules that are no longer needed
- Retag assets after dataset changes

> During reprocessing, Boon AI can usually accomplish its tasks with locally stored proxy images and video. However, in rare cases, source files may be resynchronized.

Reprocessing by Reimporting a Data Source

To reprocess a data source, perform one of these procedures:

- Reimport the entire data source
- Reimport a portion of the data source

Reimporting an Entire Data Source

You can reprocess an entire data source by calling the `import_files()` function again. New files are downloaded, and existing files receive metadata from new analysis modules that are set on the data source.

> Depending on the number of files in the data source, reimporting all of the files may take a long time.

The following example code imports (reimports) the entire data source named `MarketingData` by calling the `import_files()` function.

Python

```python
import boonsdk

app = boonsdk.BoonApp('BOONAI-APIKEY')
app.datasource.get_datasource('MarketingData').import_files()
```

Reimporting a Portion of a Data Source

You can also reprocess a portion of a data source.

The following example code looks at the `/files/march2020` folder within the `MarketingData` data source, imports (reimports) all images and PDF documents, and applies only the `boonai-label-detection` analysis module. If any of the `import_files()` function arguments are left empty, the defaults that were set up when the data source was created are used.

Python

```python
app = boonsdk.BoonApp('BOONAI-APIKEY')

app.datasource.get_datasource('MarketingData').import_files(
    prefixes=["/files/march2020"],
    file_types=["pdf", "IMAGES"],
    modules=["boonai-label-detection"]
)
```

The following example code reprocesses a portion of the `MarketingData` data source (all PNG images containing actors) and applies the new analysis module

`boonai-face-detection` .

Python

```python
import boonsdk

app = boonsdk.BoonApp('BOONAI-APIKEY')

# Add Boon AI face detection to all PNG files in the MarketingData d
app.datasource.get_datasource('MarketingData')
    .import_files("/photos/actors", file_types=['png'], modules(['bo
```

Working with Assets

Assets are the main currency of Boon AI. They typically contain the following data:

- All of the standard metadata associated with uploaded files
- Metadata generated from the various ML providers
- Associated files generated during the importing process, which may include the following:
    - Thumbnail images
    - Low-resolution version, high-resolution video
    - Raw API results from various ML providers
    - Web Video Text Tracks (WebVTT) files for speech transcription services

Getting a Single Asset

You can get assets by asset ID and view their metadata.

Getting an Asset by Unique ID

To get an asset quickly, use its unique ID to call the `get_asset()` function, as shown in the following example code.

Python

```python
import boonsdk

app = boonsdk.BoonApp('BOONAI-APIKEY')
asset = app.asset.get_asset("dd0KZtqyec48n1q1ffogVMV5yzthRRGx")
```

Accessing Asset Attributes

Assets contain a hierarchical structure of metadata attributes. The `asset` class provides a convenient API for accessing metadata attributes using a dot notation syntax. In the following example code, the `source` key contains all of the information about the asset's source file, and `fileSize` is the size of the source file. (~~Standard Fields~~ contains a list of all standard attributes.)

Python

```python
asset.get_attr("source.fileSize")
```

The following example code shows how to get an asset by its unique ID and then inspect some of the asset's properties.

Python

```python
import boonsdk

app = boonsdk.BoonApp(apikey='BOONAI-APIKEY')
asset = app.asset.get_asset("dd0KZtqyec48n1q1ffogVMV5yzthRRGx")

# Get the URI of the asset.
print(asset.uri)
"https://i.imgur.com/Ly80IrC.jpg"

# Get the automatic labels.
print(asset["analysis.boonai-label-detection.predictions"])
[
    { "label": "persian cat", "score": 0.98},
    { "label": "angora cat", "score": 0.97},
    { "label": "cat", "score": 0.98},
    { "label": "house cat", "score": 0.97}
]
```

Getting Metadata from an Asset

Metadata is stored as a deeply nested dictionary. To access the metadata, call the `asset.get_attr()` function, as shown in the following example code.

Python

```python
import boonsdk

app = boonsdk.BoonApp('BOONAI-APIKEY')
asset = app.asset.get_asset('N0E4QURFNkQtMjBFNy00RTAyLUIwMDM')

file_name = asset.get_attr('source.fileName')
print(file_name)
>> cat.jpg
```

Dumping an Entire Asset Document

To view the raw metadata for an asset, call the `asset.get_asset()` function, as shown in the following example code.

Python

```python
import boonsdk
import json

app = boonsdk.BoonApp('BOONAI-APIKEY')
asset = app.asset.get_asset('N0E4QURFNkQtMjBFNy00RTAyLUIwMDM')

print(json.dumps(asset.document, indent=4))
```

Searching for Assets

Boon AI asset search is powered by Elasticsearch modified with Boon AI proprietary plugins. Most features of the Elasticsearch search language are exposed for use on Boon AI. The features include, but are not limited to, the following, which are found in the Elasticsearch Query DSL:

- Full text queries
  - Query string and simple query string (dog and cat)
  - Match phrase query
  - Match query
- Geospatial queries (GPS positioning)
- Span queries (for legal documents and patents)
- Term queries
  - Exact term match
  - Field exists or missing
  - Prefix query
  - Range query
- Aggregations
- Sorting
- Paging

Image Similarity Query

You can run an image similarity query to search for assets by visual appearance using other assets or uploaded files as a search query.

Example Image Similarity Query

Python

```python
import boonsdk

app = boonsdk.BoonApp(apikey='BOONAI-APIKEY')

# Grab an asset by ID.
asset = app.assets.get_asset("hWsGI5FlI2HRdbT6EzJ3HqF3IA0WBx2_")

# Add the asset to a similarity query.
query = SimilarityQuery(asset)

search = app.assets.search(query)
for asset in search:
    print(asset)
```

Example Image Similarity Query with Elasticsearch Boolean Query

You can also work the image similarity query into a standard Elasticsearch compound query, such as an Elasticsearch Boolean query, as shown in the following example code.

Python

```python
import boonai

app = boonai.BoonAiApp(apikey='BOONAI-APIKEY')

# Grab an asset by ID.
asset = app.assets.get_asset("MRefrZCCb2jjJA0qMxKDZvPRkck6a2Ew")

# You can add the query to a must_not clause to find images
# that are not similar.
query = {
    "bool": {
        "must_not": [
            boonai.search.SimilarityQuery(asset)

        ],
        "must": [
            {
                "simple_query_string": {
                    "query": "frog"
                }
            }
        ]
    }
}

search = app.assets.search(query)
for asset in search:
    print(asset.uri)
```

Example Image Similarity Query on a Local File

You can run an image similarity query on a local file that has not been imported into the Boon AI system. First you must upload the file to the server to generate a similarity hash.

> Uploading large images to perform an image similarity search does not increase accuracy. All images are resized to 224 x 224 before the hash is generated.

Python

```python
import boonai

app = boonai.BoonAiApp(apikey='BOONAI-APIKEY')

# Images to do similarity search on.
images = [
    "/path/to/my_image1.jpg",
    "/path/to/my_image2.jpg"
]

# You can add the query to a must_not clause to find images
# that are not similar.
query = {
    "bool": {
        "must": [
            app.assets.get_sim_query(images, min_score=0.5)
        ]
    }
}

search = app.assets.search(query)
for asset in search:
    print(asset.uri)
```

Face Similarity Query

You can run a face similarity query to search for assets by similar faces. To perform this type of search, you must have first run the `boonai-face-detection` module.

Example Face Similarity Query

Python

```python
import boonsdk

app = boonsdk.BoonApp(apikey='BOONAI-APIKEY')

asset = app.assets.get_asset("hWsGI5FlI2HRdbT6EzJ3HqF3IA0WBx2_")

# Get all detected faces from the asset.
preds = asset.get_label_predictions('boonai-face-detection')

# Search for other assets that contain both faces.
search = {
    "query": boonsdk.search.FaceSimilarityQuery(preds)
}

for asset in app.assets.search(search):
    print(asset.uri)
```

Example Face Similarity Query with Elasticsearch Boolean Query

You can use the face similarity search within an Elasticsearch Boolean query to create interesting queries, as shown in the following example code.

Python

```python
import boonai

app = boonai.BoonAiApp(apikey='BOONAI-APIKEY')

asset = app.assets.get_asset("hWsGI5FlI2HRdbT6EzJ3HqF3IA0WBx2_")

# Get all detected faces from the asset.
preds = asset.get_label_predictions('boonai-face-detection')

# Search for other assets that contain the same faces
# as the example asset, but not in a forest.
search = {
    "query": {
        "bool": {
            "must": [
                boonai.search.FaceSimilarityQuery(preds)
            ],
            "must_not": [
                "simple_query_string": {
                    "query": "forest"
                }
            ]
        }
    }
}

for asset in app.assets.search(search):
    print(asset.uri)
```

Label Confidence Query

You can run a label confidence query to filter assets by a set of predicted labels and a range of confidence scores.

Example Label Confidence Query

The following example code searches for any asset where `boonai-label-detection` had a prediction of `lakeside`, with a confidence score between 0.5 and 1.0.

Python

```python
import boonsdk

app = boonsdk.BoonAiApp('BOONAI-APIKEY')

search = {
    "query": boonsdk.search.LabelConfidenceQuery(
        "boonai-label-detection", ["cat"], 0.5, 1.0
    )
}


search = app.assets.search(search)
for asset in search:
    print(asset.uri)
```

Label Confidence Query Arguments

Argument	Description
namespace	Analysis namespace where the predictions are stored
labels	Single label or array of labels to filter by
min_source	Minimum confidence score value to accept

max_score	Maximum confidence score value to accept

Training Set Filters

A key part of the Model training workflow is labeling your training set. In many cases you may want to exclude any labeled Asset from your search result, or get all labeled Assets for a particular model.

Excluding all Training Sets

To exclude any Asset used for model training from the search results you can use the `ExcludeTrainingSetsFilter` class.

Python

```python

import boonsdk
from boonsdk.filters import ExcludeTrainingSetsFilter

app = boonsdk.BoonApp('BOONAI-APIKEY')

search = {
    "query": {
        "match_all": {}
    }
}

for asset in app.assets.search(search=search, filters=ExcludeTrainin
    print(asset)
```

Filtering by Model, LabelScope, and Label name

Using a TrainingSetFilter we can easily filter the Assets that have been labeled for a particular model with a specific scope and purpose.

Python

```python
import boonsdk
from boonsdk.filters import TrainingSetFilter

app = boonsdk.BoonApp('BOONAI-APIKEY')

search = {
    "query": {
        "match_all": {}
    }
}

model = app.models.find_one_model(name="cat breeds")

filt = TrainingSetFilter(model,
                         scopes=[LabelScope.TRAIN],
                         labels=['tabby'])]

for asset in app.assets.search(search=search, filters=[filt]):
    print(asset)
```

Query String Query

You can run a full text search to return assets based on a text query string.

Example Simple Query String Query

Python

```python
import boonsdk

app = boonsdk.BoonApp('BOONAI-APIKEY')

search = {
    "query": {
        "simple_query_string": {
            "query": "persian | angora"
        }
    }
}

for asset in app.assets.search(search=search):
    print("this is a cat: {}".format(asset["source.path"]))
```

Example Search with Elasticsearch Simple Query String Query

The following example code performs the same search as the preceding query, but uses the Elasticsearch simple query string query to form the search request.

Python

```python
import boonsdk

import elasticsearch_dsl as es

app = boonsdk.BoonApp('BOONAI-APIKEY')

search = es.Search().query(
    es.query.SimpleQueryString(query = "persian | angora"))

for asset in app.assets.search(search=search):
    print("this is a cat: {}".format(asset["source.path"]))
```

Searching Specific Fields

By default, `SimpleQueryString` and `QueryString` use all of the fields in the database. You may also specify specific fields to search.

> When you specify fields for any type of inexact keyword search, you should append `.fulltext` to the field name, or the search may not yield the expected results. Fulltext fields are analyzed so that searching or "cat," for example, also matches "cats."

The following example code searches for a specific set of predictions. The field name for the exact value is `source.path`. Therefore, to run a fuzzy (inexact) search, the `source.path.fulltext` field is used.

Python

```python
import boonsdk

import elasticsearch_dsl as es

app = boonsdk.BoonApp('BOONAI-APIKEY')

search = es.Search().query(
    es.query.SimpleQueryString(
        query = "persian | angora",
        fields = [
            "analysis.boonai-text-detection.predictions.label.fullte
            "source.path.fulltext"
        ])
    )

for asset in app.assets.search(search=search):
    print(asset["source.path"])
```

Reference

Elasticsearch query string query

Terms Query

You can run a terms query to return assets that contain the exact term in a provided field.

Example Terms Query

Python

```python
import boonsdk

app = boonsdk.BoonApp('BOONAI-APIKEY')

search = {
    "query": {
        "terms": {
            "analysis.boonai-label-detection.predictions.label": ["t
        }
    }
}

for asset in app.assets.search(search=search):
    print(asset.uri)
```

Example Search with Elasticsearch Terms Query

The following example code performs the same search as the preceding query, but uses the Elasticsearch terms query to form the search request.

Python

```python
import boonsdk

app = boonsdk.BoonApp('BOONAI-APIKEY')

search = es.Search().query("terms", **{
    "analysis.boonai-label-detection.predictions.label" : ["tree_fro
})


for asset in app.assets.search(search=search):
    print(asset.uri)
```

Boolean Query

You can run a Boolean query to compose compound queries comprised of multiple filters.

Example Boolean Query

Python

```python
import boonsdk

app = boonsdk.BoonApp('BOONAI-APIKEY')

search = {
    "bool": {
        "should": [
            {
                "terms": {
                    "analysis.boonai-label-detection.predictions.lab
                },
            {
                "simple_query_string": {
                    "query": "frog"
                }
            }
        ]
    }
}

for asset in app.assets.search(search=search):
    print(asset.uri)
```

Boolean Query Parameters

Parameter	Description
must	Array of queries that must match (AND)

should	Array of queries where at least one has to match (OR)
must_not	Array of queries that should not match
filter	Similar to must but does not affect the sorting; is generally faster than using must

Aggregations

Boon AI supports most of the typical Aggregations you can do with ElasticSearch as well as a few others that are useful on BoonAI Data.

Prediction Label Aggregation

This aggregation returns the 1000 most common labels for a given set of predictions and can be filtered by score. The labels are returned in alphabetical order.

```
import boonsdk
from booksdk.search import PredictionLabelsAggregation

app = boonsdk.BoonApp('BOONAI-APIKEY')

agg = PredictionLabelsAggregation(
    "face_rec",
    "boonai-face-recognition",
    min_score=0.5)

query = {
    "aggs": agg
}

search = app.assets.search(query)
# Pass the PredictionLabelsAggregation instance to
# the search.aggregation() to pull the results for
# the aggregation
for bucket in search.aggregation(agg)['buckets']:
    print(bucket)
```

The results of this aggregation will give you the predicted label along with the number of assets that label shows up in.

```
{'key': 'Clint Eastwood', 'doc_count': 6}
{'key': 'Julia Roberts', 'doc_count': 3}
{'key': 'Unrecognized', 'doc_count': 1}
```

Prediction Metrics Aggregation

This aggregation gives you confidence score metrics about the prediction labels for a specific analysis namespace.

```python
import boonsdk
from booksdk.search import PredictionMetricsAggregation

app = boonsdk.BoonApp('BOONAI-APIKEY')

agg = PredictionMetricsAggregation(
    "face_metrics",
    "boonai-face-recognition",
    agg_type="stats")

query = {
    "aggs": agg
}

search = app.assets.search(query)
# Pass the PredictionLabelsAggregation instance to
# the search.aggregation() to pull the results for
# the aggregation
for bucket in search.aggregation(agg)['buckets']:
    print(bucket)
```

The results of this aggregation will give you the predicted label along with various metrics concerning the prediction score.

```json
[
  {
    "key": "Julia Roberts",
    "doc_count": 3,
    "stats#stats": {
      "count": 3,
      "min": 0.23,
      "max": 0.98,
      "avg": 0.42,
      "sum": 15.9
    }
  },
  {
    "key": "Unrecognized",
```

```
15      "doc_count": 1,
16      "stats#stats": {
17        "count": 1,
18        "min": 0.0,
19        "max": 0.0,
20        "avg": 0.0,
21        "sum": 0.0
22      }
23    }
```

Elasticsearch Query DSL Documentation

Query DSL | Elasticsearch Guide [7.10] | Elastic

https://www.elastic.co/guide/en/elasticsearch/reference/7.10/query-dsl.html

Downloading Files

Each asset has one or more associated files that can be downloaded using the Boon AI Client. The files may include the following types:

- Thumbnail images
- Web optimized video (video files only)
- WebVTT files (from speech-to-text processing)
- Raw API results from ML services

To get a list of the files associated with an asset, use the `get_files()` function on the asset, as shown in the following example code.

Python

```python
files = asset.get_files()
```

You can also use the `get_files()` function to filter the results by certain types or properties, as shown in the following example code.

Python

```python
files = asset.get_files(mimetype='image/jpeg', category='proxy')
```

File Properties

Files have a few properties that...

Property	Description

id	Unique ID of the file
name	Name of the file
size	Size of the file
mimetype	MIME type of the file, for example, image/jpeg
category	Use or purpose of the file, for example, proxy

To download a file, use the `download_file()` function on the `app.assets` instance, as shown in the following example code.

Python

```python
import boonsdk

app = boonsdk.BooApp(apikey='BOONAI-APIKEY')
asset = app.assets.get_asset(...)

thumbnails = asset.get_files(mimetype="image/jpeg")
for thumb in thumbnails:
    app.assets.download_file(thumb, '/tmp/' + thumb.name)
```

Custom Fields

You can create and set values of custom fields. These can be used for data that is not the result of analysis.

To use custom fields, you first need to create the fields desired in the Elastic Search mapping:

```
import boonsdk

app = boonsdk.BoonApp('BOONAI-APIKEY')

try:
  app.fields.create_field('my-data', 'keyword')
except boonsdk.BoonSdkDuplicateException:
  print('Field already created')

```

In the example above, the newly created field will be readable as "custom.my-data", and will be of the `keyword` type. Supported field types are `keyword`, `binary`, `boolean`, and `date`, as well as Numeric types.

Please refer to the Elastic Search documentation for details.

Once the mapping is updated, the fields can be updated with values:

```
from itertools import islice

# Assume that my_data is a dictionary with some keyword for each file
# that we imported, keyed on the file name

update_dict = {}

query = {'query': {'exists': {'field': 'analysis.boonai-image-similarity'}
for asset in app.assets.scroll_search(query):
        file_name = asset.document['source']['filename']
        update_dict[asset.id] = {'my-data': my_data[file_name]}

def chunks(data, SIZE=100):
```

```
    it = iter(data)
    for i in range(0, len(data), SIZE):
       yield {k:data[k] for k in islice(it, SIZE)}

for item in chunks(update_dict):
  try:
    resp = app.assets.batch_update_custom_fields(item)
  except boonsdk.BoonSdkInvalidRequestException as irex:
    pprint.pprint(irex)
```

Note that we are sending `batch_update_custom_fields` only 100 assets at a time, this is to avoid timeouts.

It is also possible to add custom data to assets that are created with `FileImport` or `FileUpload`. In this case, you use the custom option in these classes.

Video Clips

When processing video, a series of video clips is automatically created. These video clips represent predictions along the video's timeline. You can also create video clips manually with the Boon AI Client using your own start and stop points. Thumbnail images are automatically created for manual clips.

Video clips are fully searchable by most properties and support parent-child searches. For example, you can search video clips and also return the associated assets with a sample of video clips.

Video Clip Properties

The properties of video clip objects describe the asset they came from, the in and out points in the video, and the content of the clip.

Property	Description
id	Unique ID of the video clip.
asset_id	Asset ID of the video clip.
content	Content of the video clip, for example, a prediction such as "cat," or a sentence of detected audio dialog.
timeline	Name of the video clip timeline, which is usually the analysis module that created the clip.
track	Name of the track, for example, "cat," "dog," or "parrot," which may be used as track names for an animal detector. Each predicted label becomes a track.
start	Start time of the video clip, in seconds.
stop	Stop time of the video clip, in seconds.
length	Length of the video clip, in seconds.
score	Prediction confidence for the video clip.
simhash	An image similarity hash for the video clip.

Creating Video Clips

Video clips are usually created automatically when processing video files, assuming the selected analysis module supports video files.

Sometimes creating searchable video clips manually is advantageous. You can manually create video clips in the following ways:

- With existing hand-tagged data
- Using production notes from tools like Shotgun
- On the fly or interactively

When a video clip is created, a thumbnail image is automatically generated at the midpoint of the clip time. For example, if the video clip is between 0 and 10 seconds, the thumbnail is created at 5 seconds.

Creating a Single Video Clip

Python

```python
import boonsdk

app = boonsdk.BoonApp(apikey='BOONAI-APIKEY')

asset = app.assets.get_asset('QkU2RjJENDUtRTNFMy00NDRDLTk3Q')

app.clips.create_clip(
    asset,
    'Post Production',
    'VFX Notes',
    4.5, 6.0,
    'Add more lens flare.')
```

Creating Multiple Video Clips Using TimelineBuilder

You can create video clips one at a time, or you can quickly create multiple video clips using TimelineBuilder. TimelineBuilder enables you to describe all of the annotations for an asset on a timeline and to create all of the video clips with a single API call, as shown in the following example code.

Python

```python
import boonai

app = boonai.BoonAiApp(apikey='BOONAI-APIKEY')
asset = app.assets.get_asset('QkU2RjJENDUtRTNFMy00NDRDLTk3Q')

tlb = boonai.TimelineBuilder(asset, 'Hand Tagged Data')
tlb.add_clip('Prodct Shots', 45.5, 60.1, 'Sony Erickson')
tlb.add_clip('IMDB Numbers', 26.1, 29.4, ['155241', '225123'])

app.clips.create_clips_from_timeline(tlb)
```

Exporting a WebVTT File

WebVTT is a standard for displaying timed text, such as subtitles and captions, in a web browser. Boon AI allows you to dynamically convert any set of video clips to WebVTT on the fly.

Getting the Full WebVTT

You can download a WebVTT file that includes all of the video clips for every timeline and track, as shown in the following example code.

Python

```python
import boonsdk

app = boonsdk.BoonApp(apikey='BOONAI-APIKEY')

asset = app.assets.get_asset('QkU2RjJENDUtRTNFMy00NDRDLTk3Q')
with open('full.webvtt', 'w') as fp:
    app.clips.get_webvtt(asset, fp)
```

Getting a Filtered WebVTT File

Sometimes you only need a specific timeline or track or you may want to filter the WebVTT by content. In these cases, provide arguments to the `get_webvtt()` function, as shown in the following example code.

Python

```python
import boonai

app = boonai.BoonAiApp(apikey='BOONAI-APIKEY')

timeline = 'boonai-label-detection'
track = ['dog', 'cat']

asset = app.assets.get_asset('QkU2RjJENDUtRTNFMy00NDRDLTk3Q')
with open('full.webvtt', 'w') as fp:
    # All filters can be a single item or an array.
    app.clips.get_webvtt(asset, fp, timeline=timelines, tracks=track
```

Other Filtering Options

To save the WebVTT data as a string rather than as a file, pass a `None` for the `dst_file` argument, as shown in the following example code.

Python

```python
import boonai

app = boonai.BoonAiApp(apikey='BOONAI-APIKEY')

asset = app.assets.get_asset('QkU2RjJENDUtRTNFMy00NDRDLTk3Q')
webvtt_data = app.clips.get_webvtt(asset, None)
```

You can also filter the WebVTT by matching the content of the video clip, as shown in the following example code.

Python

```python
import boonai

app = boonai.BoonAiApp(apikey='BOONAI-APIKEY')

# Find every clip where someone says "This is the way."
content_search = "This is the way."
timeline = 'gcp-speech-to-text'

asset = app.assets.get_asset('QkU2RjJENDUtRTNFMy00NDRDLTk3Q')
webvtt_data = app.clips.get_webvtt(asset, None, content=content)
```

Searching Video Clips

Video clip search allows you to search across all video clips. The search API is the same as searching for assets, with the full power of ElasticSearch at your disposal.

Content Search

The content of a video clip contains the text for the prediction that was made. This may be a single prediction (as shown in the following example code), a list of predictions, or a full sentence from a speech transcription service. Video clip content is stored in an ElasticSearch text field and does not support Elasticsearch aggregations.

Python

```python
import boonsdk

app = boonsdk.BoonApp(apikey='BOONAI-APIKEY')
search = {
    'query': {
        'match': {
            'clip.content': 'snow'
        }
    }
}
clips = app.clips.search(search=search)
for clip in clips:
    print(clip)
```

Searching by Video Clip Properties

You can filter video clips by track, timeline, or asset ID using a basic ElasticSearch `term` filter, as shown in the example code.

Python

```python
import boonai

app = boonai.BoonAiApp(apikey='BOONAI-APIKEY')
search = {
    'query': {
        'bool': {
            'must': [
                {
                    'term': { 'clip.timeline': 'boonai-label-detecti
                },
                {
                    'term': { 'clip.assetId': 'abc123qajhduhnj3232j2
                }
            ]
        }

    }
}
clips = app.clips.search(search=search)
for clip in clips:
    print(clip)
```

Scrolling Large Results

If a query returns more than 10,000 results, you can replace the `search()` function with the `scroll_search()` function, as shown in the example code.

Python

```python
import boonai

app = boonai.BoonAiApp(apikey='BOONAI-APIKEY')
search = {
    'query': {
        'term': {
            'clip.track': 'en-us'
        }
    }
}
clips = app.clips.scroll_search(search=search)
for clip in clips:
    print(clip)
```

Reference

Elasticsearch term query

Custom Models

The Boon AI platform allows you to create, train, and deploy custom models. You can also import pre-trained models.

Once you have trained a model within your project, that model will show up at the bottom of your list of available analysis modules for use in subsequent analyses.

ML Workflow

An Overview of the Process

The machine learning workflow describes a series of phases involved in applying a model to a project. This should be seen as a flexible and iterative process and may look different depending on the requirements of your project. Typically these phases are outlined in 5 steps:

1. **Data collection and pre-processing:** Identify the sources from which you will be importing assets.
2. **Building datasets:** This step involves labeling assets to be used in training your model. During this process, you can apply labels to your assets as designate them for use in training or testing.
3. **Model Training and refinement:** At this point, you can link a dataset to a model for training. You have the flexibility here to test multiple datasets with a single model or to apply a variety of models to a single dataset.
4. **Evaluation:** You can review and evaluate the performance of a model by reviewing predictions for accuracy.
5. **Deployment to production:** This model is deployed as an easy to use REST API for integration into software projects.

> A single project can contain multiple datasets. Each dataset consists of a number of labeled assets.
>
> In addition to datasets, each project can have multiple models associated with it. This allows flexibility in determining which datasets and which models best suit the needs of your project.

Datasets

A dataset is a collection of assets to which a label has been assigned. In order for a model to be trained, it must be attached to a dataset.

Treating datasets as their own entities, separate from models, allows you to train different models using the same dataset repeatedly. Another benefit to organizing labeled assets into datasets is that you can curate different versions of a dataset and then compare the results of different datasets to see which ones allowed your model to make better predictions.

Creating a Dataset

The following example code creates a dataset called `flowers-dataset`.

```
import boonsdk

app = boonsdk.BoonApp(apikey='BOONAI-APIKEY')

dataset = app.datasets.create_dataset('flowers-dataset',
                                      'Classification',
                                      description='This is an example data
```

The dataset name is the first argument to `create_dataset`. The second argument is the dataset type, in this case 'Classification'. Dataset types can be 'Classification', 'Detection' or 'Face Recognition'. The dataset type will determine the format of the labels contained by the dataset, and to what models you can attach this dataset. The last argument is an optional description.

Label Types

The dataset type determines the format of the labels it contains. Classification labels consist of a text label for the whole image. Detection labels consist of a text label and a bounding box, which determines where the labeled object is. Face Recognition labels have a text label a bounding box and a face similarity hash, which is an encoded feature vector that will be used to classify the faces. This vector is computed by the face detection module.

Labeling Assets

After you have created a dataset, it's time to start labeling assets so you can link the dataset to a model that you want to train and test.

> Uploaded models are not trainable, although you can still link datasets to them, in order to do testing.

Labeling a Dataset

The following example code creates a model that identifies tax forms and labels two assets as "IRS Form 1040 Page 1." This example code uses `ModelType.KNN_CLASSIFIER`, which provides reasonable predictions with a small set of labeled assets.

Python

```python
import boonsdk

app = boonsdk.BoonApp(apikey="BOONAI-APIKEY")

dataset = app.datasets.create_dataset("tax-forms-dataset", "Classifi

# Create a label instance we can then apply to some assets.
label = dataset.make_label("IRS Form 1040 Page 1")

asset1 = app.assets.get_asset("hWsGI5FlI2HRdbT6EzJ3HqF3IA0WBx2_")
asset2 = app.assets.get_asset("AfkNjsdaldamnADAW13dasdAsdaSasew")

# Apply the "1040 Page 1" label to asset1 and asset2.
app.assets.batch_update_labels([asset1, asset2], label)
```

Labeling Test Assets

By default, any asset that you label for training is part of a training set. You can also label a group of assets for testing a custom model before it is deployed by providing a label scope as shown in the following example code. Test labels are never used for training, so they can be used to test a model with examples it has never seen before.

Python

```python
import boonai

app = boonai.BoonAiApp(apikey="BOONAI-APIKEY")

dataset = app.datasets.create_dataset("tax-forms-dataset", "Classifi

test_label = dataset.make_label(
      "IRS Form 1040 Page 1",
      scope=boonsdk.LabelScope.TEST))

test_asset = app.assets.get_asset("hWsGI5FlI2HRdbT6EzJ3HqF3IA0WBx2_"

app.assets.batch_update_labels(test_asset, test_label)
```

Labeling with an Asset Search

Because the `app.assets.update_labels()` function takes an array of up to 1,000 assets, you can use an asset search to quickly label a large training set, as shown in the following example code.

Python

```python
import boonai

app = boonai.BoonAiApp(apikey='BOONAI-APIKEY')

dataset = app.datasets.create_dataset("tax-forms-dataset", "Classifi

# Search for all files with the term "form_1040" in the
# file path/URI.
search = {
    'size': 100,
    'query': {
        'match': {
                'source.path.path': {
                    'query': 'form_1040'
                }
            }
        }
    }
}

# Make a label.
label = dataset.make_label("IRS Form 1040 Page 1")

# Iterate through all assets and apply the label in batches of 100.
for assets in app.assets.scroll_search(search).batches_of(100):
    app.assets.update_labels(assets, label)

# Now that some assets have been labeled, you can get a dump of
# the label counts.
for label, count in app.dataset.get_label_counts(dataset).items():
    print("%-25s %d labeled" % (label, count))

>> "IRS Form 1040 Page 1           674 labeled"
```

Bulk Labeling Assets

In order to train certain type of models you often need thousands of examples for each class. To speed up the labeling process, BoonAI allows you to bulk label up to 10,000 Assets using a search. This is most often uses in case where you've used an off the shelf model like `gcp-label-detection` and now want to take some of those predictions and promote them to actual labels with your specific class names.

In this example we're labeling the top 100 Assets where gcp-label-detection detected "rake" or "shovel" with "Landscaping", as long as the confidence score is over 90%.

```
import boonsdk

app = boonsdk.app_from_keyfile("apikey.json")

ds = app.datasets.get_dataset("IAB Advertising")
label = ds.make_label("Landscaping")

search = {
    "query": {
        "bool": {
            "filter": [
              boonsdk.search.LabelConfidenceQuery(
                    "gcp-label-detection", ["rake", "shovel"], 0.9, 1.0
                )
            ]
        }
    }
}

app.assets.label_search(search, label, 100)

```

Models

Once you have trained a model, you can use it to make predictions about the content of other, previously unseen media files . Each prediction is a decimal value between zero and one, representing a confidence score. These scores provide you with a scale to use in assessing the reliability of any given prediction. Confidence scores that are closer to one are more likely to be reliable predictions.

Types of Models

Boon AI supports a wide array of models and configuration options. There are 2 main types of models to choose from, Trainable models that can be fully trained in BoonAI and Uploadable models which can be trained outside BoonAI.

Trainable Models

Trainable models are trained within the BoonAI platform.

KNN Classifier

A KNN classifier can be used to sort images or documents in to groups/classifications with just one labeled example. This type of model generates a single prediction which can be used to quickly organize assets into general groups. If no labels are provided, the model automatically generates numbered groups of similar images. These groups can be renamed and edited in subsequent training passes.

The KNN classifier uses a sklearn.neighbors.KNeighborsClassifier on top of the Similarity Hashes that are calculated for every asset. A Similarity Hash is a quantized feature vector computed from a Resnet-152 CNN trained on Imagenet. This hash is the same used for similarity search.

Tensorflow Transfer Learning Classifier

The CNN classifier utilises Tensorflow 2.x and transfer learning on various base models with Imagenet weights. This type of model is great for identifying real world photography and documents. You must have at least 2 unique classes and 10 examples of each class.

Face Recognition

The Face Recognition model can identify faces with just a few labeled examples. It is implemented as a sklearn.neighbors.KNeighborsClassifier on top of face features computed with the boonai-face-detection module. This module uses MTCNN for detection and an Inception Resnet model to generate the features, as implemented here.

Uploadable Models

Uploadable models can be trained outside BoonAI and then deployed as a service in the BoonAI infrastructure.

Boon Functions

A Boon Function is your own custom code that can be used to not only make predictions but also populate custom fields. Boon Functions are written and Python and deployed into the BoonAI backend system so they are usable as an ingest module or even as an endpoint.

Torch Model Archive Classifier

A Torch Model Archive (MAR file) classifier can be trained outside of BoonAI and deployed into BoonAI. The model must use the `image_classifier` handler.

For examples on how to package a Torch model into a Model Archive see:

serve/examples/image_classifier at master · pytorch/serve

https://github.com/pytorch/serve/tree/master/examples/image_classifier

Torch Model Archives

Beyond training custom models, you can also upload a TensorFlow or PyTorch Image classification models to use in Boon AI. When they are deployed, these models can be used on videos, images, and documents at scale just like a locally trained model.

Torch Model Archives (MAR files) contain trained neural net models along with supporting code and labels that can be deployed to BoonAI.

> Check out a tutorial in model archives for more detailed information.

Tools

In order to use your trained Keras model with BoonAI your model will need an InputShape object with a shape attribute of (length, width, 3). The values of length/width correspond to the sizes of the image you used to train the model.

Before you can upload a Torch model you need to package it as a MAR. To do that you will need to install the `torch_model_archiver` tool.

```
pip install torch-model-archiver
```

Handlers

After training a TensorFlow image classifier locally, you can export the model to a directory using the `save()` method on the model object as shown in the following example code.

BoonAI supports all 4 types of the Torch Serve's inference handlers. An inference handler ensures your model handles a standard input and outputs the results in a standard output.

The BoonAI server will validate that your model is a real model. Any extra files will cause the model to be rejected.

> Your Torch model must either work with the default inference handers or you must provide your pre/post process code.

- `image_classifier = boonsdk.ModelType.TORCH_MAR_CLASSIFIER`
- `object_detector = boonsdk.ModelType.TORCH_MAR_DETECTOR`
- `text_classifier = boonsdk.ModelType.TORCH_MAR_TEXT_CLASSIFIER`
- `image_segmenter = boonsdk.ModelType.TORCH_MAR_IMAGE_SEGMENTER`

Packaging Example

The method for uploading a PyTorch model is basically identical to the method used for uploading a TensorFlow model. The only difference is the model type you provide to Boon AI. In the code below, the model type is declared on line 2.

What do you need to package your model into a MAR file?

- The Torch model itself (.pt / .pth file)
- A model.py file to initialize your model
- An `index_to_name.json` file which translates numeric predictions into readable labels.
- Know your handler type. (see handlers above)

Now that you know or have all these things you can make your model archive.

```
torch-model-archiver --model-name <anything> \
   --model-file model.py --serialized-file my-model.pth \
   --handler image_classifier --extra-files index_to_name.json

```

Now you can use the BoonAI API to upload your MAR file. Note that depending on the size of the model it will take 2-3 minutes to deploy. When the model is ready for use, the model state will change from `Deploying` to `Deployed`.

Python

```python

model = app.models.create_model("my-model",
    boonai.ModelType.TORCH_MAR_CLASSIFIER)

# Replace my-model.mar with the name of your MAR file.
app.models.upload_pretrained_model(
    model,
    'my-model.mar'
)
```

Once Deployed you can test your model using a Dataset with test labels or just run inference manually.

Boon Functions

A Boon Function is your own custom code that is either executed on Assets as they are processed or used as your own custom endpoint. Boon Functions are written and Python and deployed into the BoonAI backend system so they are a fast and scalable solution for handling custom requirements.

There are many reasons as to why you would use a Boon Function.

- To combine, filter, or convert predictions from various ML providers to a single list.
- Use predictions from various providers to determine the value of a custom field.
- Make requests to your own hosted services for additional metadata.

The Basics

Structure

Boon Functions must have at least 1 file named `function.py`. This file must contain a function called `process()` which takes a single argument `asset` and returns a `FunctionResponse`. The asset argument is an instance of a boonsdk Asset class, although it is immutable in this context. Boon Functions must return a value within 60 seconds or the execution is canceled.

The simplest NoOp Boon Function possible would be something like:

function.py

```
from boonsdk import FunctionResponse


def process(asset):
    return FunctionResponse()

```

Dependencies

Boon Function containers are based on Ubuntu Focal and includes Python 3.8. To provide your own dependencies you can provide a **requirements.txt** file in your package file

Packaging

To deploy a Boon Function you must zip up function.py along with any other code resources and upload sing the BoonSDK. The files in the zip should be in the root of the archive, anything in a directory will be ignored.

```
$ zip boon.zip function.py requirements.txt
```

Returning Results

The `FuctionResponse` class is used to store the results that will be applied to your Asset. There are various forms of metadata you can provide, however you can not make arbitrary changes to the Asset's internal structure.

Returning Predicted Labels

To add a set of predicted labels to your Asset, the BoonSDK provides a LabelDetectionAnalysis class.

```
function.py

from boonsdk.func import FunctionResponse, LabelDetectionAnalysis


def process(asset):

    analysis = LabelDetectionAnalysis()
    analysis.add_label_and_score("cat", 0.45)
    analysis.add_label_and_score("dog", 0.99)
```

```
    return FunctionResponse(analysis)


```

The results of this function can later can be found on the Asset by calling get_analysis() with the name of your custom model or by viewing the Asset in the console.

```
import boonsdk

app = boonsdk.app_from_keyfile("key.json")

# Replace the ID here with your Asset ID.
asset = app.assets.get_asset("0CNu7BSI2V7KKflHWzx2yQexmJsvPjKV")

asset.get_analysis("my-model-name")

{
    "type": "labels",
    "predictions": [
        {
            "label": "dog",
            "score": 0.99
        },
        {
            "label": "cat",
            "score": 0.45
        }

    ]
}

```

Returning Content

Another form of Analysis is `ContentDetectionAnalysis`. This class simply stores a blob of text, usually from some type of speech to text / OCR operation.

In this case we're going to combine high confidence Amazon and Google keywords into a single text blob that our users could search on, but only if the confidence score is 0.75 or above.

```python
from boonsdk.func import FunctionResponse, ContentDetectionAnalysis


def process(asset):
    content = ContentDetectionAnalysis()

    aws = asset.get_analysis("aws-label-detection")
    gcp = asset.get_analysis("gcp-label-detection")

    for pred in aws['predictions'] + gcp['predictions']:
        if pred['score'] >= 0.75:
            content.add_content(pred['label'])

    return FunctionResponse(content)


```

Now we'll do a search using our Analysis fields.

```python
import boonsdk

app = boonsdk.app_from_keyfile("key.json")

# Search our new field for any content with both the word
# cat and gorilla.
query = {
    "query": {
        "simple_query_string": {
            "query": "cat gorilla",
            "fields": ["analysis.my-model.content"],
            "default_operator": "and"
        }
    }
}

# Print the results.
results = app.assets.search(query)
for asset in results:
    asset.get_analysis("my-model")
    {
        "type": "content",
        "conent": "gorilla cat mouse cheetah"
    }

```

Setting Custom Fields

Using the `FunctionResponse` class it's also possible to set values on your custom fields.

If you haven't already you must first create a custom field in your project. You only have to do this once. In this case we're going to add a custom field called "blurriness" which will hold a value that describes how blurry the thumbnail for our asset is.

```
import boonsdk

app = boonsdk.app_from_keyfile("key.json")

app.fields.create_field("blurriness", "float")
```

For this we're going to need to provide a requirements.txt file with some additional libraries.

```
requirements.txt

opencv-python-headless
numpy
```

Now the code for function.py

```
function.py

from boonsdk.func import FunctionResponse, get_proxy_image

import cv2
import numpy as np

def process(asset):
    proxy_bytes = get_proxy_image(asset, level=0)
    image = cv2.imdecode(np.frombuffer(proxy_bytes, np.uint8), 1)
    image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    blur = cv2.Laplacian(image, cv2.CV_64F).var()

    result = FunctionResponse()
    result.set_custom_field("blurriness", blur)
```

```
    return result

```

Creating a Model

Creating a Model

The following example code creates a Tensorflow Classifier model called `flowers`. The predictions for this model will be written into the asset in the `analysis.flowers` attribute. At creation, we assign a dataset, which is assumed to already exist.

Python

```python
import boonsdk

app = boonsdk.BoonApp(apikey='BOONAI-APIKEY')

try:
    dataset = app.datasets.find_one_dataset(name='flowers-dataset')
except boonsdk.BoonSdkNotFoundException:
    print('Dataset not found')

model = app.models.create('flowers',
                          boonsdk.ModelType.TF_CLASSIFIER,
                          dataset=dataset)

```

Providing an Analysis Module Name

Optionally, you can provide an Analysis Module name when you create a model,which affects where in the Asset metadata the results are written. In the following example code, the predictions are in the `analysis.Wild Flower Type` namespace. The analysis module name is in the `model.module_name` property.

Python

```python
import boonai

app = boonai.BoonAiApp(apikey='BOONAI-APIKEY')

model = app.models.create('flowers',
    boonai.ModelType.BOONAI_LABEL_DETECTION,
    module_name='Wild Flower Type')

print(model.module_name)
>> "Wild Flower Type"

print(mode.namespace)
>> "analysis.Wild Flower Type"
```

Training a Model

Once your dataset has enough labels to train a model, you can kick off a training job.

Python

```python
import boonsdk

app = boonsdk.BoonApp(apikey="BOONAI-APIKEY")

# Load a model by its name.
try:
    dataset = app.datasets.find_one_modelt(name='flowers')
except boonsdk.BoonSdkNotFoundException:
    print('Model not found')

# Train the mode and apply predictions on all existing data.
job = app.models.train_model(model, post_action=PostTrainAction.APPL

```

The `train_model()` function returns an asynchronous training job that can be monitored in the Console. If you need to wait for the job to complete (either success or failure), you can use the `wait_on_job()` function in the `app.jobs` instance, as shown in the following example code.

Python

```python
# Train the model.
job = app.models.train_model(model)

# Wait on the job.
app.jobs.wait_on_job(job)
```

Training Arguments

Some models allow you to modify parameters that affect training.

Python

```
import boonsdk

app = boonsdk.BoonApp(apikey="BOONAI-APIKEY")

schema = app.models.get_model_type_training_args(
    boonsdk.ModelType.KNN_CLASSIFIER)

boonsdk.util.print_json(schema)

# One argument, n_clusters can be set.
>>{
>>  "args": {
>>    "n_clusters": {
>>      "type": "Integer",
>>      "defaultValue": 15,
>>      "description": "If no labels are setup, this many automatic
>>    }
>>  }
>>}

```

What this means is that you can set the argument "n_clusters" to an integer value, but if the value is not set the default value is 15.

Setting Training Arguments

Once you know the arguments you can pass a dictionary of arguments to the `set_training_args` () method

Python

```python
import boonsdk

app = boonsdk.BoonApp(apikey="BOONAI-APIKEY")

# Load a model by its name.
model = app.models.find_one_model(name="flowers")

app.models.set_training_args(model, {"n_clusters": 4})

```

Checking Your Arguments

Most training arguments have sensible defaults and you don't have to set all available arguments, just the ones you want to change.

If you want to see just the arguments you've set, that data is available on the Model object itself.

Python

```python
import boonsdk

app = boonsdk.BoonApp(apikey="BOONAI-APIKEY")

# Load a model by its name.
model = app.models.find_one_model(name="flowers")
print(model.training_args)
>> { 'n_clusters': 4 }
```

To see all your arguments resolved together with defaults, call the `get_training_args() method.`

Python

```python
import boonsdk

app = boonsdk.BoonApp(apikey="BOONAI-APIKEY")

model = app.models.find_one_model(name="cars")
print(app.models.get_training_args(model))
>> {
>>     'epochs': 12,
>>     'fine_tune_epochs': 10,
>>     'validation_split': 0.3
>>}
```

Training Post Actions

When training a model you can select a single post action to execute once the model is trained, or no post action at all.

- APPLY - The apply post action will apply the train model to your Assets. By default the APPLY action will run through all your assets but this can be modified by setting the models apply search property.
- TEST - The test post action will apply the model to your test labels only. This allows you to generate a confusion matrix to rate the accuracy of your model before applying it to all your data.
- NONE - Take no post action at all, simply train the model.

Applying and Testing

Once a model is uploaded or trained you can make the choice to apply the model to all of your Assets, apply it to only a select set of Assets, or apply it to only Assets with Test labels.

Testing a Model

To run your model on only your test labels you will use the `test_model()` method. This will process enough data to generate a confusion matrix that describes how well your model performed.

Python

```python
app = boonsdk.BoonApp(apikey="BOONAI-APIKEY")

model = app.models.find_one_model(name="cats and mice")
job = app.models.test_model(model)

# Block until the job is complete.
app.jobs.wait_on_job(job)
```

Note that this assumes that the dataset linked to the model you use has test labels.

Applying a Model to All Assets

Python

```python
app = boonsdk.BoonApp(apikey="BOONAI-APIKEY")

try:
    model = app.models.find_one_model(name="cats and mice")
except boonsdk.BoonSdkNotFoundException:
    print('Model not found')

job = app.models.apply_model(model)
```

Applying a Model to Select Assets

The `app.models.apply_model()` method allows you to specify a specific search to apply the model to.

Python

```python
app = boonsdk.BoonApp(apikey="BOONAI-APIKEY")

try:
    model = app.models.find_one_model(name="cats and mice")
except boonsdk.BoonSdkNotFoundException:
    print('Model not found')

search = {
    "query": {
        "term": { "custom.show": "Tom and Jerry" }
    }
}
job = app.models.apply_model(model, search=search)
```

Exporting Models

All of the models available in Boon AI are exportable to your local drive for inspection or use within an external application. All model files come packaged as zip files. The code below demonstrates how you can find a model by name and export it to the location of your choice.

Python

```python
app = boonsdk.BoonApp(apikey="BOONAI-APIKEY")

try:
    model = app.models.find_one_model(name="cats and mice")
except boonsdk.BoonSdkNotFoundException:
    print('Model not found')

app.models.export_trained_model(model, "/tmp/model.zip")
```

Exported Model Contents

The contents of each exported model zip file will be unique based on the type of model you have downloaded. Each model is exported with a `predict.py` and `requirements.txt` script which provide an example of how to run the model on an image using Python.

```
unzip model.zip

# Assuming you don't have dependencies installed
pip3 install -r requirements.txt

pythob3 predict.py <path to image>

# The output will show the label and the score.
Label: ants, Score: 0.6810492873191833
Label: bees, Score: 0.31895074248313904
```

Boonlab

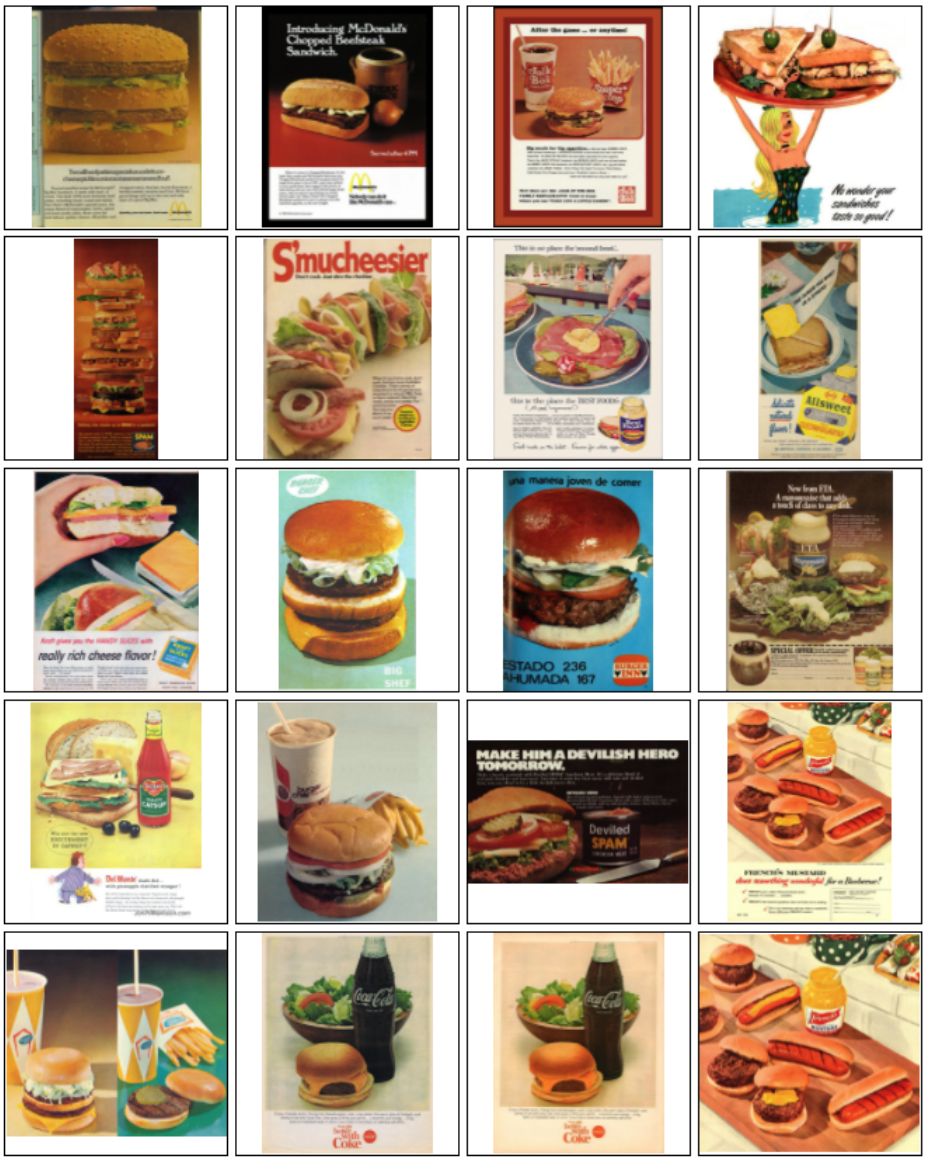
The Boon AI Lab, or boonlab, is a package of convenience functions. It allows you to work with asset proxies, execute searches, and do some basic plotting. Some of the functionality in boonlab is specifically meant to be used from a Jupyter notebook, while other functions are useable in plain Python scripts.

Display

You can display search results and individual asset proxies:

```
import boonsdk
from boonlab.display import show_asset, show_thumbnails

app = boonsdk.BoonApp(apikey='BOONAI-APIKEY')

search = app.assets.search(
                        {
                          "size":20,
                          "query":{
                            "bool":{
                              "must":[
                                {
                                  "simple_query_string":{
                                    "query":"sandwich"
                                  }
                                }
                              ]
                            }
                          }
                        })
images_list = show_thumbnails(search)
```

When run inside a Jupyter notebook, this will produce the following display:



The `show_thumbnails` function also returns a list of filenames, one per asset, containing JPEG images for each proxy.

You can also display one individual asset:

```
# (assuming you run the code to display thuimbnails from above)
asset = search[0]
document = show_asset(asset)
```

The function returns the JSON document corresponding to the asset.

Hashes

The Similarity Hashes that are computed for every asset can be useful for other things other than similarity searches. For example, the KNN classifier model type uses these hashes to classify assets.

It is possible to read all the hashes for assets in a search as a numpy array:

```
import boonsdk
from boonlab.hashes import read_hash_as_vectors

app = boonsdk.BoonApp(apikey='BOONAI-APIKEY')

search = app.assets.search({"size": 20})

hashes =  read_hash_as_vectors(search=search,
                               attr='analysis.boonai-image-similarity.simhash',
                               percentage=100,
                               verbose=True)
```

This will return a numpy array containing all the hashes belonging to the search provided. If no search is given, then an empty one is used, meaning you want hashes representing all the assets in the repository to be returned as an array. The `attr` argument specifies which hash you want.

> *It is possible to have other hashes that are not the similarity one–for example, you could compute a hash based on HSV color values, something based on a Bag of Words analysis of text found via OCR, or any other sort of feature vector.*

The `percentage` argument can be used to get a random subset of the hashes, useful when you have many images and want to work on just a sample set of them.

Pandas

You can get search results in the form of a Pandas dataframe:

```
import boonsdk
from boonlab.hashes import read_hash_as_vectors

app = boonsdk.BoonApp(apikey='BOONAI-APIKEY')

search = app.assets.search({"size": 20})
attrs = ['analysis.zvi-image-similarity', 'analysis.zvi-object-detection']

df_search = search_to_df(search=search, attrs=attrs)
```

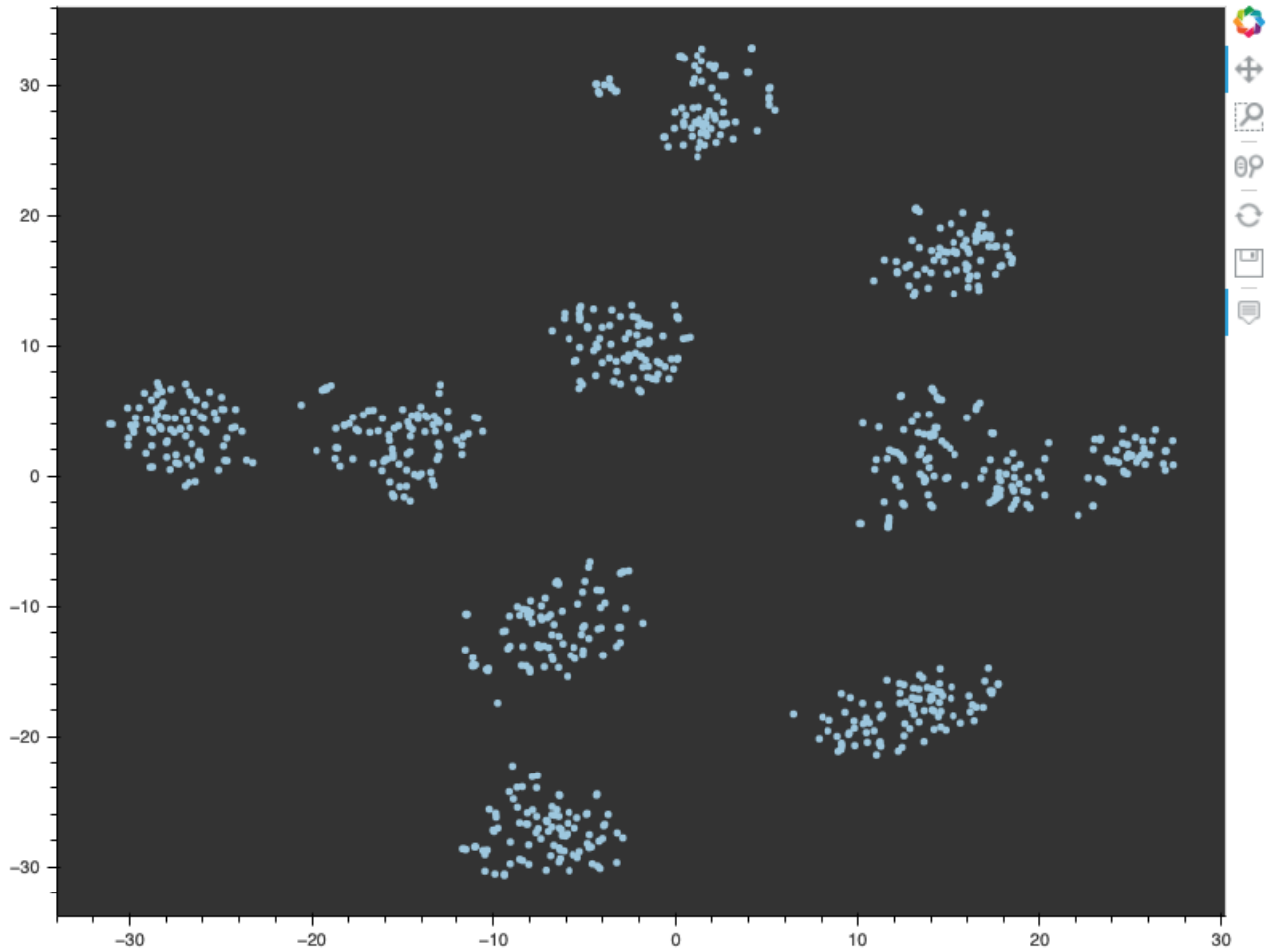
`search_to_df` returns a Pandas dataframe containing one column for each of the attributes enumerated in the `attrs` argument.

Plot

You can use the `plot_tsne` function to do a t-SNE plot of the similarity hashes belonging to the assets returned from a search:

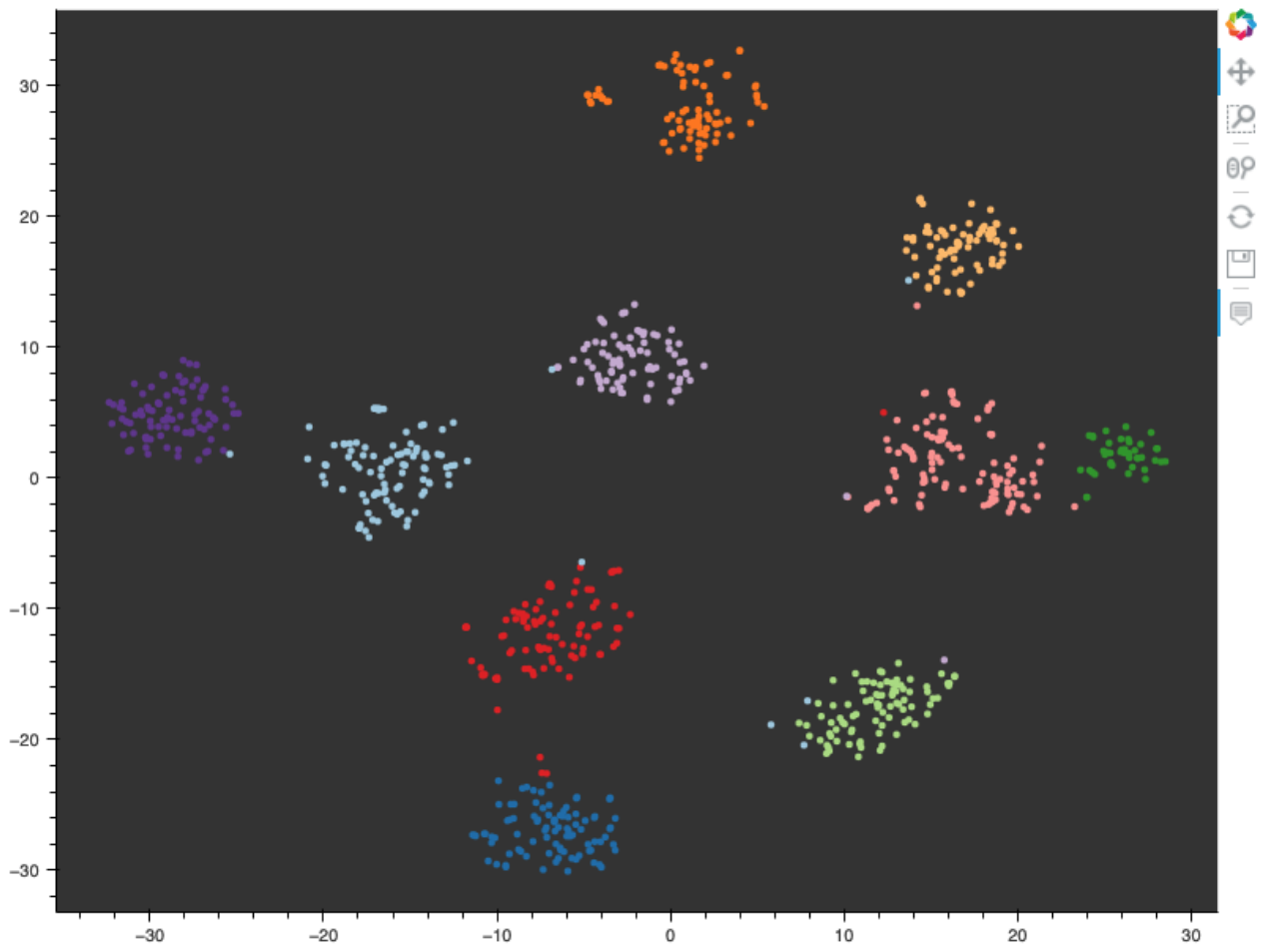
```
import boonsdk
from boonlab.plot import plot_tsne

app = boonsdk.BoonApp(apikey='BOONAI-APIKEY')

x = plot_tsne(thumbs=True)
```

When run from a Jupyter notebook, this will produce a t-SNE plot, and display it using Bokeh:



Using the nClusters argument, this function will also cluster the points using `sklearn.cluster.Kmeans`:

```
import boonsdk
from boonlab.plot import plot_tsne

app = boonsdk.BoonApp(apikey='BOONAI-APIKEY')

x = plot_tsne(thumbs=True, nClusters=10)
```



The `thumbnails` argument creates a tooltip on the graphs's points, such that when you hover over them, the corresponding proxy images are shown.

This function returns an array of assets.

Proxies

You can download proxy images by searching for an asset and then referring to that asset ID as an argument to `download_proxy` as demonstrated below.

```
from boonsdk import app_from_env
from boonlab.proxies import download_proxy

app = boonsdk.BoonApp(apikey='BOONAI-APIKEY')

search = app.assets.search(
                {
  "size":20,
  "query":{
    "bool":{
      "must":[
        {
          "simple_query_string":{
            "query":"sandwich"
          }
        }
      ]
    }
  }
})

proxy = download_proxy(asset, level=-1)
```

The proxy is returned as an opencv image array. The optional `level` argument determines which proxy level to return, and a value of -1 means the last one, which is typically the highest resolution one.

Solutions

This section contains common use cases for BoonAI and their respective solutions. All code examples can be found in the BoonAI examples github project.

GitHub - TheBoonAI/examples https://github.com/TheBoonAI/examples

Quick Integrations with Streamlit

A great option for integrations when a user interface is needed is Streamlit. Streamlit is an open source Python library that provides ways to create UI widgets and display images and data.

Getting Started

Install Streamlit:

```
> pip install streamlit
```

For details, check out the Streamlit documentation.

The code examples below can be found here.

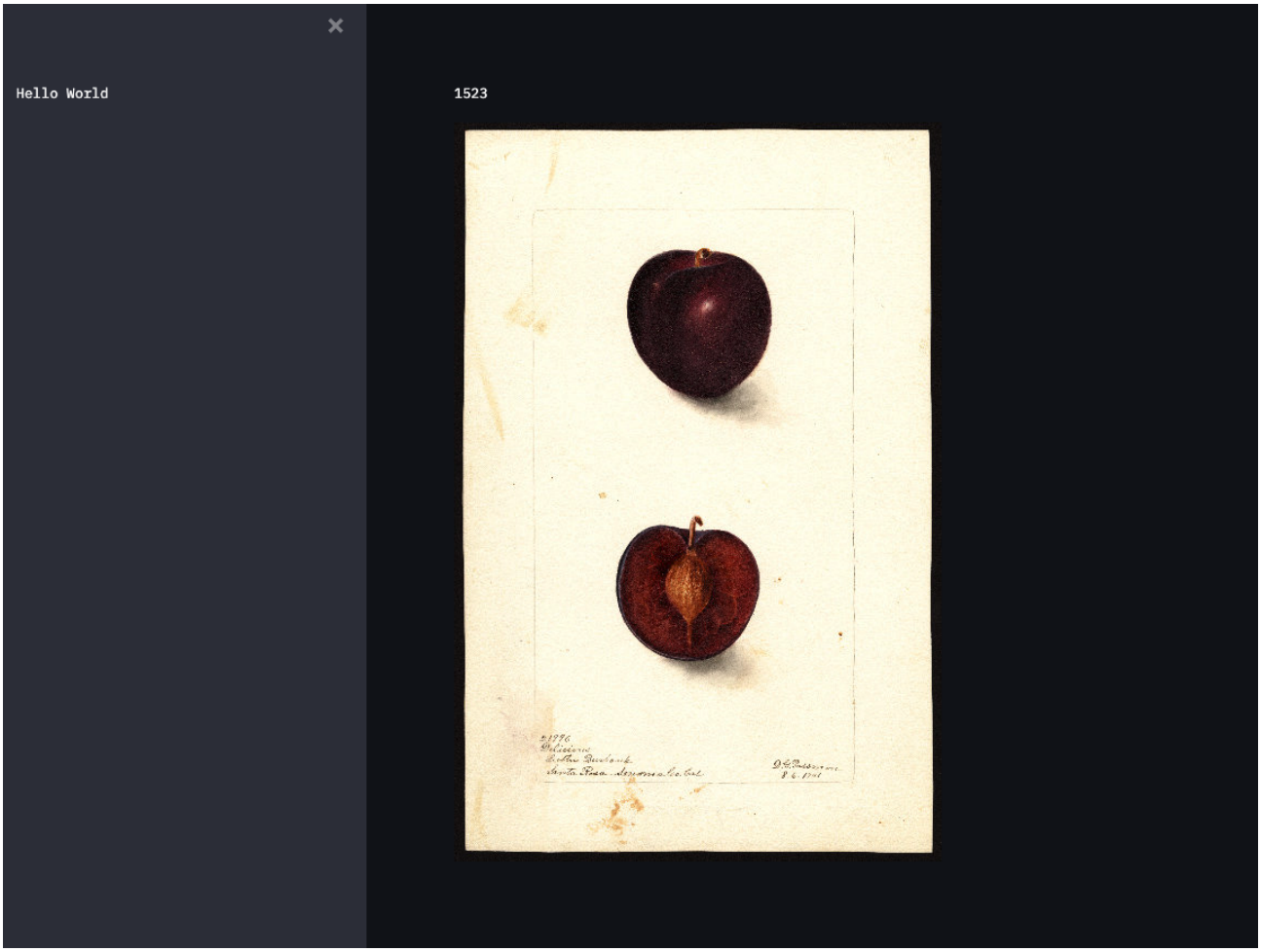
Hello World

The following is a simple script that displays one image at random from your Boon AI repo:

```
'''
This Streamlit script displays one random image from the server you log in
'''

import random
import streamlit as st
from boonsdk import app_from_env
```

```python

import boonsdk

app = boonsdk.app_from_keyfile("apikey.json")

query = {"size": 10000}
search = app.assets.search(query)
count = len(search.assets)
st.text(count)
asset = search.assets[random.randint(0, count)]

thumbnail = asset.get_files(category="web-proxy")[0]

st.image(app.assets.download_file(thumbnail))
st.sidebar.text("Hello World")
```

Run it with `strealmlit run hello_world.py`, and check out the output at `http://localhost:8501/`



Display a Search

The following will display the results of a search query. You can copy the JSON describing the query with the copy button on the Boon Console Visualizer. Once pasted into the query box in this app, the thumbnails for the search will be downloaded and displayed.

```
'''
This Streamlit script displays the results of one search
'''

import json
import numpy as np
from PIL import Image
import cv2
import streamlit as st
from boonsdk import app_from_env

import boonsdk

app = boonsdk.app_from_keyfile("apikey.json")

query = st.text_area("Copy Query from Visualizer", '{"size": 20}')
query = json.loads(query)

search = app.assets.search(query)
count = len(search.assets)

images = []
labels = []
for asset in search:
    thumbnail = asset.get_files()[-1]
    img = np.array(Image.open(app.assets.download_file(thumbnail)))
    img = cv2.resize(img, None, fx=.5, fy=.5)
    images.append(img)
    labels.append(asset.id)

st.image(images, caption=labels)

```

Run it with `strealmlit run hello_world.py`, and check out the output at `http://localhost:8501/`

Here is the output with a similarity search running:

Copy Query from Visualizer

{"query":{"bool":{"must":[{"script_score":{"query":{"match_all":{}},"script":{"source":"similarity","lang":"boonai-similarity","params":{"minScore":0.75,"field":"analysis.boonai-image-similarity.simhash","hashes":
["PPPBPEAMCPAPPBAPALIFDKPCPDEPAPPPPKAPHCPMPPPPKMLCANPPFAPPPCAPGIBDPBPPCPPGPDAPNPCPACPPFENPPPPFPCILEPKNPNABKHPEDPBFPKPGEPJMNPJPPPPPPPECCDFAPCCJPPPKDCPMKADIPPMPEFNNPKFOPHAPPKPJPPAPHPPIBMPKPPECPMPPJPKAPABGBC
JPPPHPPPCFPPPJPBPPHBCPJIPPMIPPAPOJPPPPMPJPPPEPPIPAPBIPDPPPCPJLJGEFPAKDKPAPPCAPBEPFEPPAPEDHGOPPPPFHLKCPADGCPAAJPPPOBPKNECPPHLHAPPEJPPAPPNDPPPJPAJPFAOPPMPPHBPAPAPJEPAMPPPPKGKCNPPAPGMGHPAOAPHEGOEPPPGPKPBMGCPP

iv0AzB-xcb6gpnVbImqg48bjiLVnNRuLI

Xov976qkpd1gGprVlVBYxPIMQcELHJGL

8cf6rnjAPzbU_NOHgTFGyhPN6vlfMNfy

5stb01fEywJazAzmVtI2pzzZsVq1TWTd

83jnCAQk-J_CFY424k51IrWNx6D92pzY

PlKffAe_vOwHSzkc0oqImzgBXnAR-L1z

AT7hRQcieF21TlVG3Fdk794CR-9KqtVlR

HurVkcwN4385GN9JCH4p_cEyCCSnu_6

rIlHwAHH8MQuHEe_dABwQQ23N4RADfkJ

GPu6k9xOERfmdVz5up1KOksGBJorB2Y3

Labeling Video Frames with Streamlit

Introduction

Boon AI allows you to analyze video using models that operate on images. Boon deals with splitting the analyzed video into relevant clips, and then collating the results together.

Now imagine you want to make a training set in order to train the models, but your images are actually frames a set of from videos. One solution would be to manually extract frames, import them into Boon AI, and then label them. But this would potentially create a lot of unused frames, since you don't know a priori which ones you will need to label.

We are going to create a Streamlit app that will look inside local videos and allow you to label a frame. When you label one frame, this image is imported into Boon AI, and the corresponding label is added to your dataset. There are two advantages to this: 1) There's no need to import the videos we will take frames from; and 2) we will only be importing the images we need for our dataset, and no more.

Code

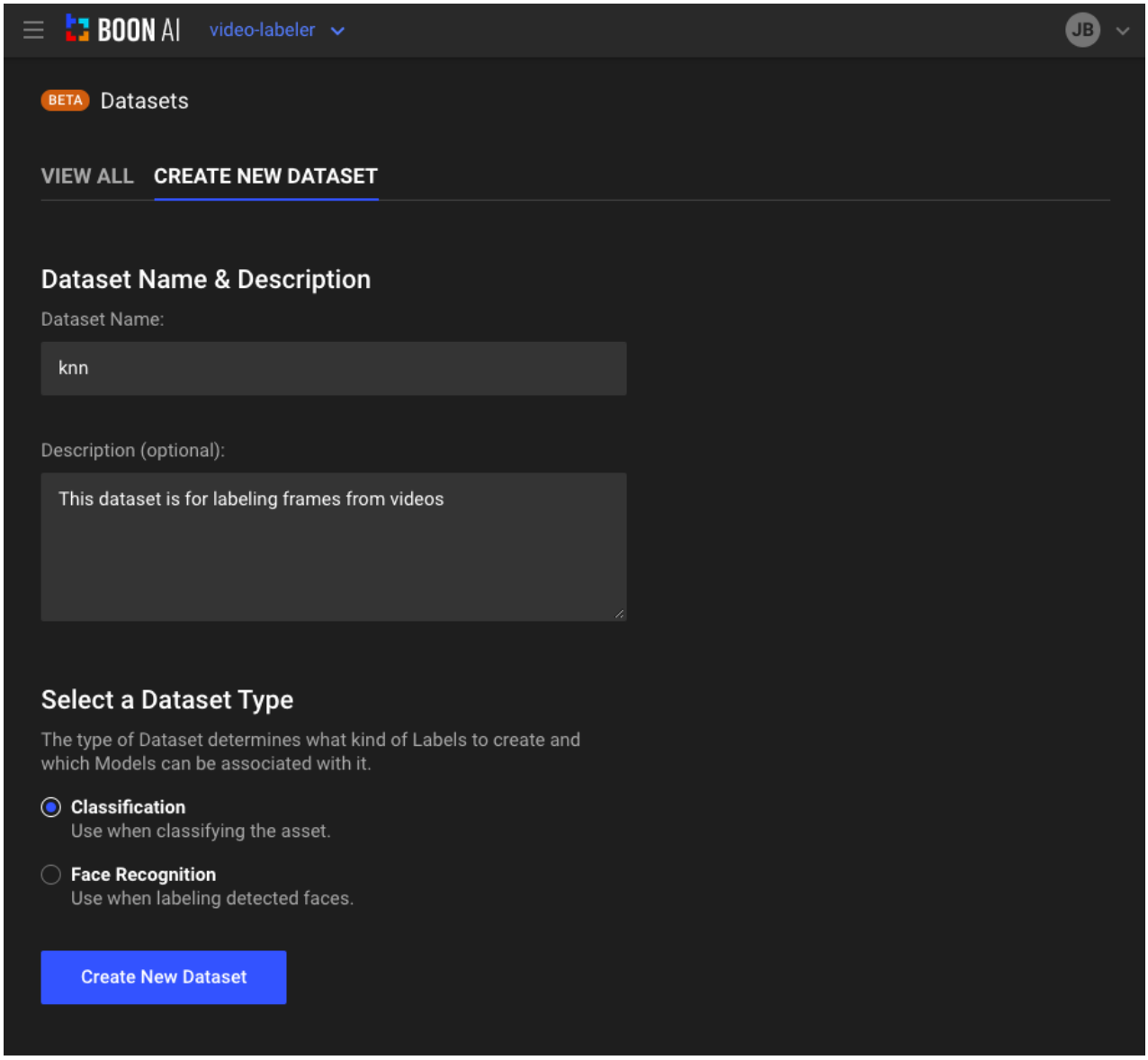
The full code for this app can be found here.

Data Preparation

We just need to create a couple of folders, and make sure there's at least a video ready for our app to work with. Then we will download a couple of videos from archive.org:

```
> mkdir videos
> mkdir frames
> cd videos
> wget https://archive.org/download/FordMustang1966CarCommercial1960sTvAd/
> wget https://archive.org/download/1977ChevroletConcours/1977%20Chevrolet
> cd ..
```

Finally, let's use the Boon AI console to create a dataset, which we will call "knn."



Script

Following is the whole Streamlit script that makes up our app:

```
import streamlit as st

import boonsdk

import os
import cv2
import base64
import glob


@st.cache
```

```
12  def get_frames(video_name, n):
13      cap = cv2.VideoCapture(video_name)
14      frame_count = int(cap.get(cv2.CAP_PROP_FRAME_COUNT))
15      step = int(frame_count / n)
16
17      frames = []
18      vals = []
19      pos = 0
20      for frame in range(0, n):
21          cap.set(cv2.CAP_PROP_POS_FRAMES, pos)
22          ret, frame = cap.read()
23          frames.append(cv2.cvtColor(frame, cv2.COLOR_RGB2BGR))
24          vals.append(pos)
25          pos += step
26
27      return frames, vals
28
29
30  app = boonsdk.app_from_keyfile("apikey.json")
31
32  videos = glob.glob('videos/*.mp4')
33  video_name = st.sidebar.selectbox('Video', videos)
34  cap = cv2.VideoCapture(video_name)
35  frame_count = int(cap.get(cv2.CAP_PROP_FRAME_COUNT))
36
37  dataset_name = st.sidebar.text_input('Dataset', value='knn')
38  dataset = app.datasets.find_one_dataset(name=dataset_name)
39
40  n_frames = st.sidebar.slider('n Frames', min_value=2, max_value=100, value
41
42  label_value = st.sidebar.text_input('Label')
43
44  scope = boonsdk.entity.dataset.LabelScope.TRAIN
45  if st.sidebar.radio('Label Scope', ['Train', 'Test']) == 'Test':
46      scope = boonsdk.entity.dataset.LabelScope.TEST
47
48  frames, vals = get_frames(video_name, n_frames)
49
50  ncols = 6
51  pos = 0
52  buttonid = 0
53  while(frames):
54      cols = st.columns(ncols)
55      for i, col in enumerate(cols):
56          if frames:
57              col.image(frames[0])
58              if col.button('Go to this frame', key=str(buttonid)):
59                  pos = vals[0]
60              buttonid += 1
61              frames = frames[1:]
62              vals = vals[1:]
```

```
pos = st.slider('frame', min_value=0, max_value=frame_count, value=pos)

cap.set(cv2.CAP_PROP_POS_FRAMES, pos)
ret, frame = cap.read()
st.image(cv2.cvtColor(frame, cv2.COLOR_RGB2BGR), use_column_width='always'

if st.sidebar.button('Label this frame'):
    im_name = video_name + '.' + str(pos) + '.jpg'
    cv2.imwrite(im_name, frame)
    label = dataset.make_label(label_value, scope=scope)
    app.assets.batch_upload_files([boonsdk.FileUpload(im_name, label=label

```

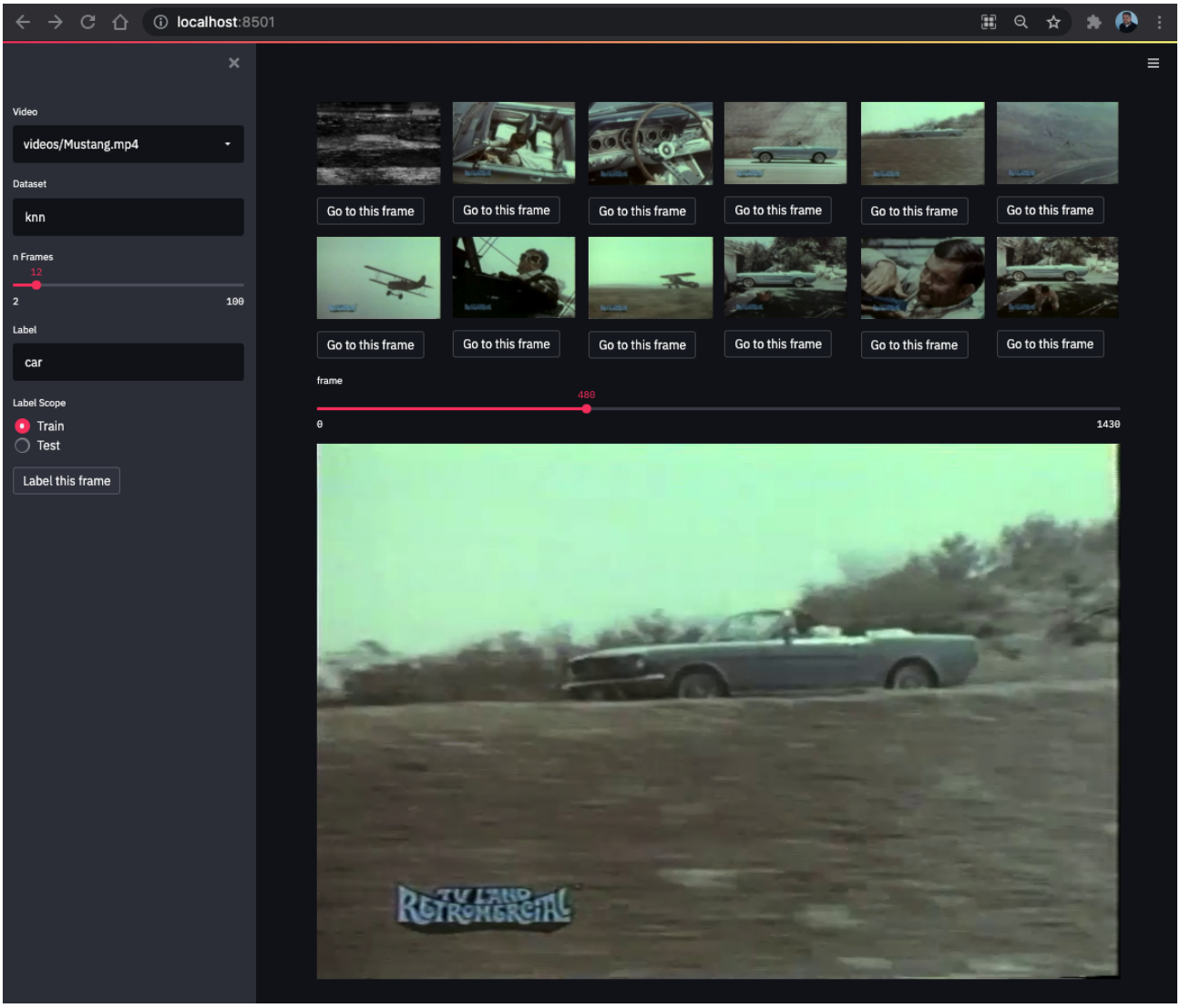
The application provides a drop down menu to select all the videos found in the `videos/` folder. Then it displays a number of frames from the video, with buttons that can be used to jump to that frame in the video displayed below the thumbnails.

On the sidebar, you can input a value for a label, along with the scope desired, "train" or "test". When you click on "Label this Frame", the main displayed frame is sent to the Boon AI server and imported with the label assigned to it.

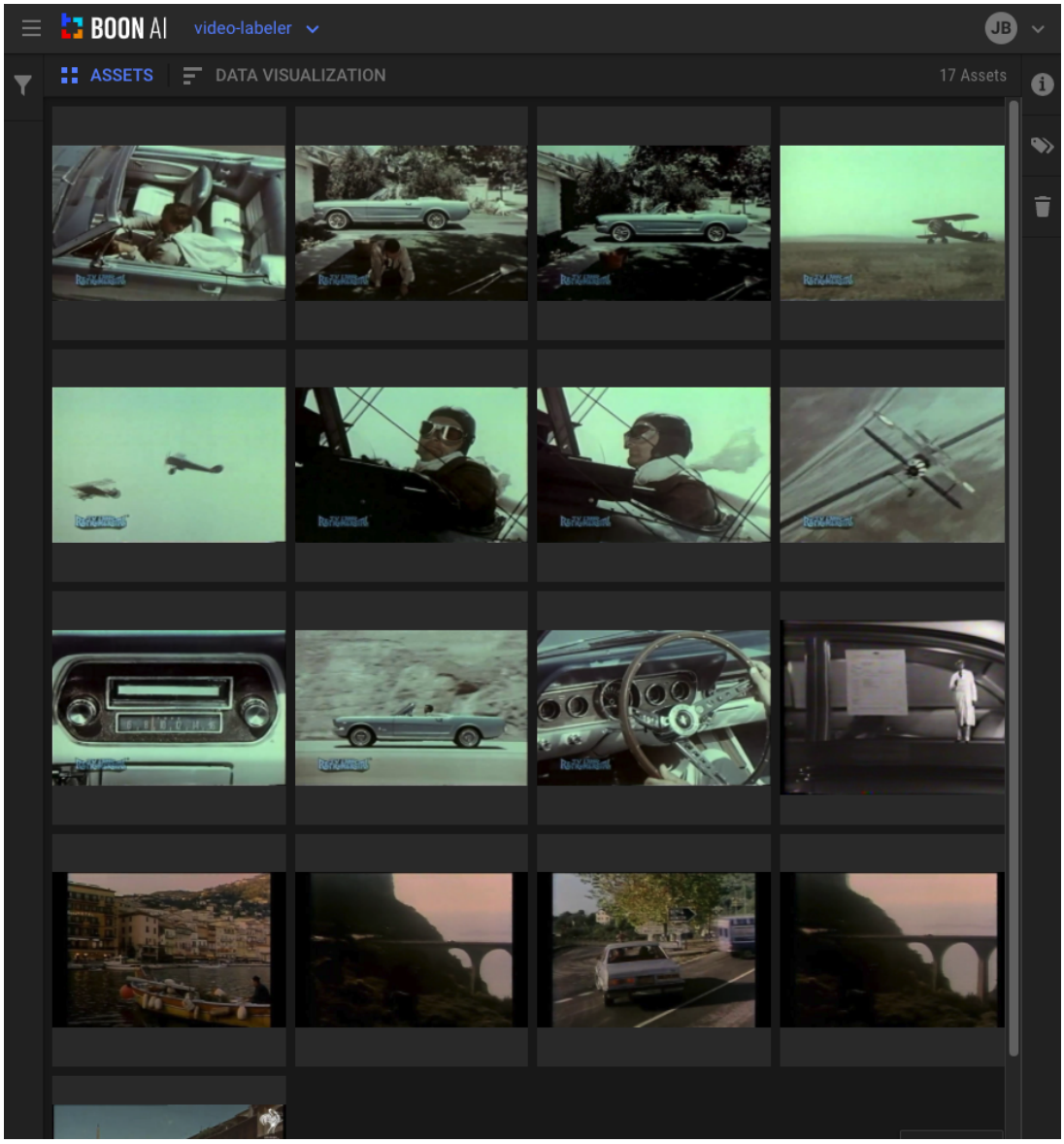
Run the script with

```
> streamlit run video.py
```

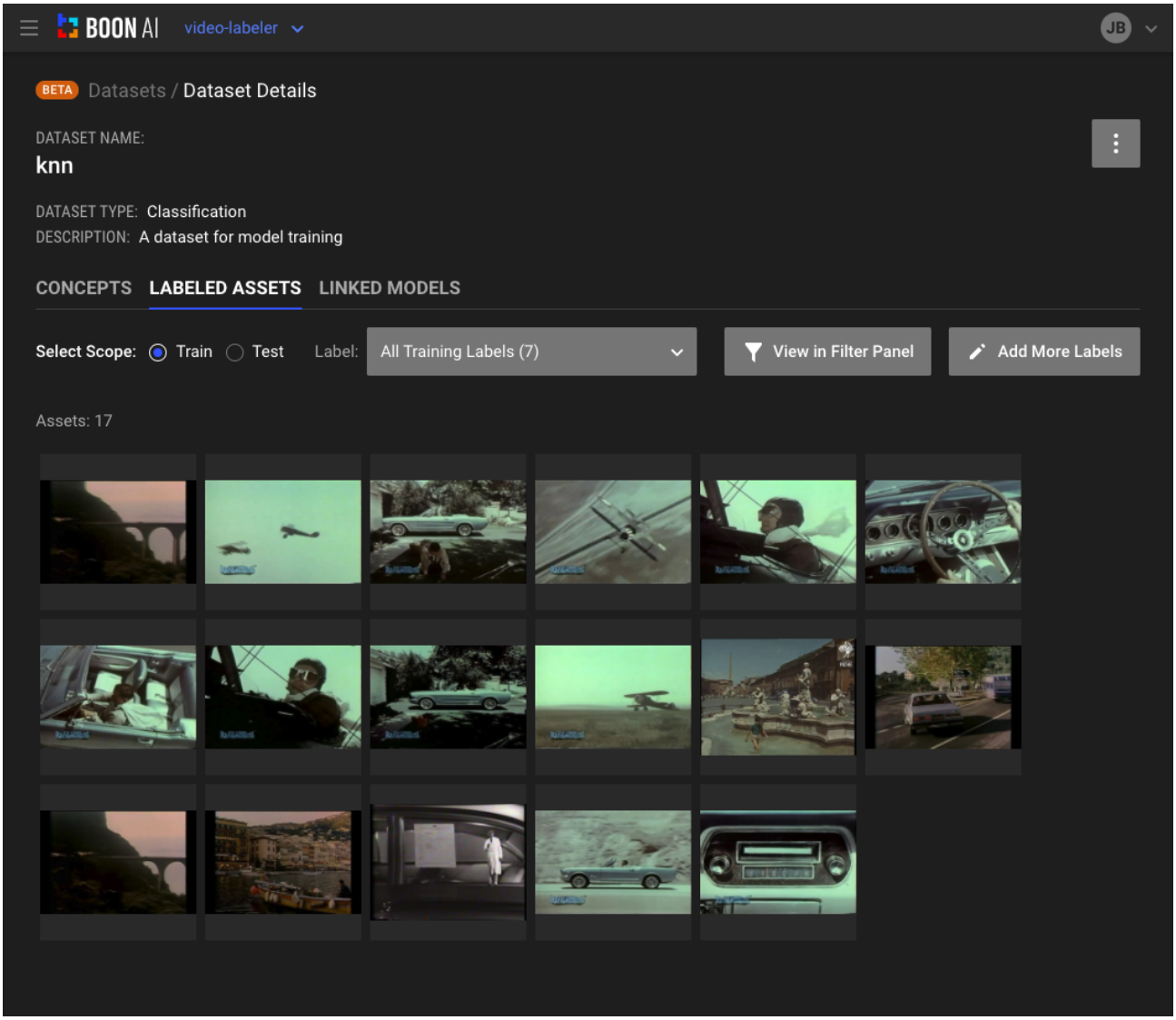
And open a browser to `http://localhost:8501/`



You can control how many thumbnails are shown with the "n Frames" slider on the left. After you've labeled a few frames, the Boon AI Visualizer should show them like so: (processing might take a few minutes, you can always check progress in the Job Queue section of the console.)

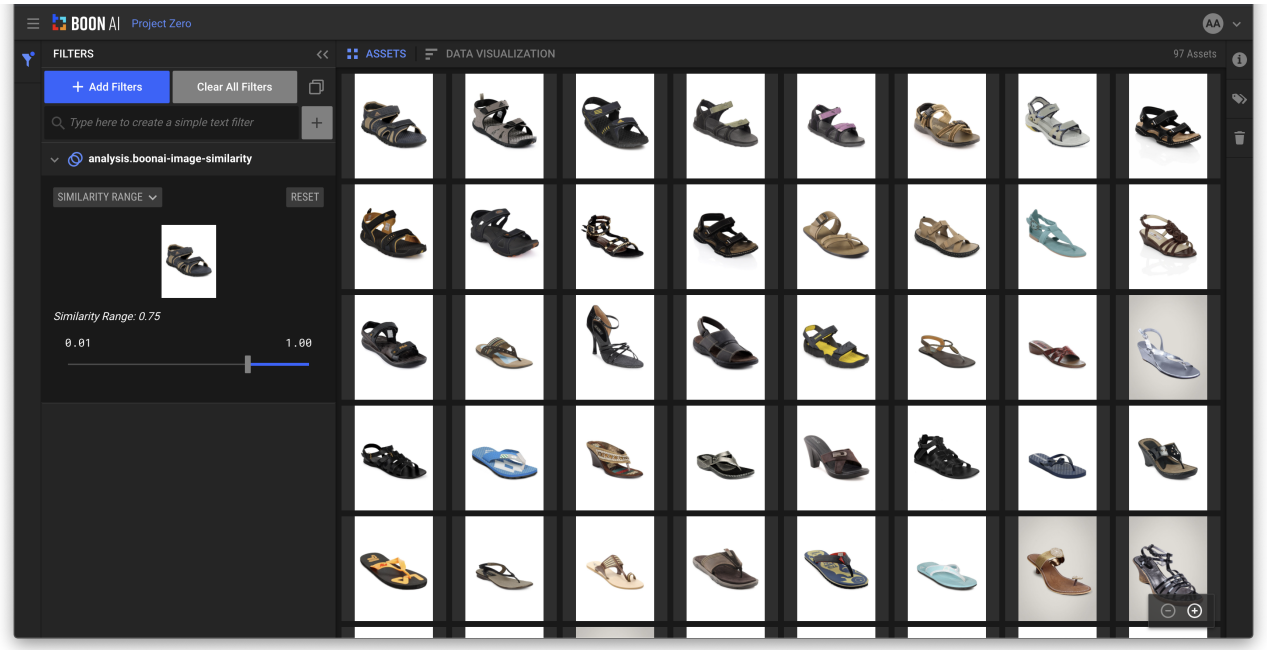


And the Dataset display should show that these have labels assigned:



There you have it, now you can label frames from videos. After you have a decent dataset, you can train a model on it and use it to evaluate both images and videos!

Product Catalog with Similarity Search



Images from the Fashion Product Dataset (Kaggle)

Introduction

One of the simplest solutions to implement with BoonAI is an image similarity search engine. Every Asset imported into BoonAI gets a pre-calculated image similarity 'fingerprint'. This fingerprint represents the semantic meaning of the image and is generated using the convolutional neural network known as Resnet152.

For this solution we're going to assume we have an e-commerce site selling various products and we're going to use BoonAI to provide our customers with a visual search option that can be combined with a traditional facet search using ElasticSearch.

All of the example code for this solution can be found here:

examples/solutions/product_similarity at main · TheBoonAI/examples

https://github.com/TheBoonAI/examples/tree/main/solutions/product_similarity

> You must save and API key file named 'apikey.json' in the same directory as the notebook for the demo code to work seamlessly.

Importing Product Images

BoonAI supports importing URLs embedded in a CSV file, so we're going to use one from Kaggle.

To get the CSV file open to link below, click on "images.csv" in in the data explorer, and then the download button on the far right hand side of screen. This will download the CSV file to your local drive.

Fashion Product Images Dataset https://www.kaggle.com/paramaggarwal/fashion-product-images-dataset

Importing the Data

To import this data we need to prepare a CsvFileImport object with the necessary settings.

```
import boonsdk

app = boonsdk.app_from_keyfile("apikey.json")

# Change this to where you put images.csv
path_to_csv = "~/Downloads/images.csv"

csvfile = boonsdk.CsvFileImport(path_to_csv,
                                uri_column=1,
                                max_assets=1000,
                                header=True)

# Add boonai-label-detection as well for some predictions to test with.
app.asset.import_csv(csvfile, modules=["boonai-label-detection"])

```

What are the CsvFileImport args we used?

```
path_to_csv - The path the CSV file.
uri_col - The column in the CSV where the image URI can be found.
max_assets - The maximum number of assets to import from the file.
header - Set to true if the CSV has a header.
```

Note that we also applied `boonai-label-detection` to these assets which will provide us with some test predictions to do facet filtering on. This module is trained with Imagenet and can be useful on pictures of common everyday items, but it won't be as customizable as any of the premium options.

Monitoring the Job

Now would be a good time to check the BoonAI UI to monitor the import job. In this case we're importing a sample of 1000 assets from the CSV file. Every file has to be analyzed in order to generate the similarity fingerprint necessary to perform similarity search.

Time to Search!

Now that our assets are imported, let's do a similarity search with an external image and display the results in a Jupyter Notebook. Here is our example image:

The search we are going to perform will be a similarity search on a picture of a striped shirt with an additional Term filter for the word "jean".

```

import os
import urllib

import ipyplot
from PIL import Image

img_path = "striped-shirt.jpg"

# Create a search.  Note that we add the LabelConfidenceQuery as a filter,
# this ensures the ML confidence score won't be averaged with the similiar
# which givess better results in this case.
search = {
```

```
    "query": {
        "bool": {
            "must": [
                app.assets.get_sim_query(img_path, min_score=0.75)
            ],
            "filter": [
              boonsdk.search.LabelConfidenceQuery(
                    "boonai-label-detection", ["jean"], 0.1, 1.0
                )
            ]
        }
    },
    "size": 20
}

# Iterate the assets and display them.
images = []
labels = []
for asset in app.assets.search(search):
    thumbnail = asset.get_files(category="web-proxy")[0]
    images.append(Image.open(app.assets.download_file(thumbnail)))
    labels.append(asset.score)

ipyplot.plot_images(images, labels, max_images=20, img_width=150)

```

Success! Our results contain exactly what we were searching for, a lot of guys wearing similar outfits: jeans with a striped shirt.

We got a couple of guys without a striped shirt. Why were they included in our results? Take a close look at their hands and compare it to our search image. The search image subject has his hands in his pocket. Small details like this can affect the results of a similarity search because the fingerprint represents the semantic meaning of the image. Peripheral details like the model's body orientation and even their hair style can play a role in the scoring of results.

Conclusion

Creating a visual search engine with BoonAI that leverages the full power of ElasticSearch and a convolutional neural network is as easy as importing files and writing a query. The next step would be adding your own custom fields to the data such as product ID which would allow you to tie the search results back into your own database.

Real Time Content Moderation with WebHooks

An example of a photograph featuring elicit content you may want to moderate.

Introduction

In this example we're going to use Clarifai's Unsafe Detection module to automatically notify a Slack channel that someone has imported content that may not be suitable for all ages. This could include pictures featuring illegal drugs, nudity, gore, etc. This mechanism could be used to notify a support staff that there is content to be reviewed for potential removal or moderation.

To accomplish this we're going to use a BoonAI Webhook.

> A Webhook is an API mechanism in which an externally hosted service is provided with real time events information. BoonAI allows you to setup Webhooks that receive notifications when Assets are created or updated.

Prerequisites

In order to deploy this solution you will need:

- Docker
- A Google Cloud project
- Google Cloud SDK
- Some knowledge of setting up Slack Apps

Code

All the code and a working Notebook this example can be found here:

examples/solutions/webhook_moderation at main · TheBoonAI/examples
https://github.com/TheBoonAI/examples/tree/main/solutions/webhook_moderation

Webhook Server

Our first step is to make a server that will accept the webhook requests using Docker, Flask, and the BoonSDK.

Let's start with some Python code to implement our web server.

The Endpoint

First we need to implement the actual endpoint. This endpoint will inspect the `clarifai-unsafe-detection` analysis and look for a 'safe' rating under 0.6. If the image is deemed unsafe we call the `send_slack_messages()` function which we'll implement next.

webhook.py

```
@app.route('/', methods=['POST'])
def webhook():

    try:
        # Validate the request is coming from BoonAI.
        validate_webhook_request_headers(flask.request.headers, WEBHOOK_SE

        # Get the BoonAI asset from the request
        asset = boonsdk.Asset(flask.request.get_json()['asset'])

        # Get The "safe" label if one exists and check the score
        safe = asset.get_predicted_label('clarifai-unsafe-detection', 'saf
        if safe and safe['score'] <= 0.6:
            send_slack_message(asset, safe['score'])

        return '', 200
    except Exception as e:
        logger.warning("Faied to process request ", e)
        return '', 400
```

Posting Results to the Slack Channel

Now that we have some tags we need to send it to our Slack channel. This is pretty easy using the Slack API.

webhook.py

```

def send_slack_message(asset, score):
    likely_hood = 100.0 - score
    text = f'{likely_hood}% chance of unsafe content! Please review asset
    try:
        SLACK_CLIENT.chat_postMessage(
          channel=SLACK_CHANNEL,
          text=text
        )
    except slack_sdk.SlackApiError as e:
        logger.warning(f'Error posting to slack {e}', e)


```

Dockerfile

The best way to deploy this server is to package it in a Docker container and deploy it somewhere, mostly likely in Google Cloud, AWS, or Azure. In our example we're using Google Cloud.

Environment variables that will need to be set on the deployed instance:

- WEBHOOK_SECRET - used to validate that requests to our server came from BoonAI.
- SLACK_TOKEN - Your Slack Bot API key.
- SLACK_CHANNEL - The Slack Channel ID where you want the notifications to be posted.

Dockerfile

```
FROM ubuntu:focal

RUN groupadd -r webhook && useradd --no-log-init -r -m -g webhook webhook

RUN apt-get update && apt-get -y upgrade && apt-get --no-install-recommend
    python3 \
    python3-pip \
     --no-install-recommends

RUN pip3 install flask gunicorn boonsdk PyJWT slack_sdk

RUN mkdir /app
WORKDIR /app

COPY ./webhook.py .

USER webhook
CMD exec gunicorn --bind :$PORT --workers 1 --threads 8 --timeout 0 webhoo

```

Building and Deploying the Webhook Server

Building and deploying the webHook server is a simple process as long as you have a Google Cloud project ready to go and you've authenticated your local Google Cloud SDK

Environment vars we need to set:

- GCP_PROJECT - the name of your GCP project
- WEBHOOK_SECRET - what our secret key will be for BoonAI to securely communicate with the server.
- SLACK_TOKEN - A Slack Bot token.
- SLACK_CHANNEL - The Slack channel ID where messages will be posted.

```
# First build container and tag it for Googles container store.
$ docker build . -t gcr.io/$GCP_PROJECT/boonai/webhook-server

# Next push to Google's container store.
$ docker push gcr.io/$GCP_PROJECT/boonai/webhook-server

# Now the fun part, deploy using Google Cloud Run
gcloud run deploy boonai-webbook \
--image gcr.io/$GCP_PROJECT/boonai/webhook-server \
--region us-central1 --platform managed \
--allow-unauthenticated --max-instances 2  \
--update-env-vars "WEBHOOK_SECRET=$WEBHOOK_SECRET,SLACK_TOKEN=$SLACK_TOKEN

```

Creating WebHook in BoonAI

Now that we have our Webhook server up and running, we can configure the Webhook in BoonAI. This is a setup step that only has to happen once. Note that you'll need two things to setup your webhook:

1. The URL of your webhook endpoint.
2. The value of your webhook secret key.

```
# Replace this value with your endpoint URL.
endpoint = "https://boonai-webbook-9999999999-uc.a.run.app"

# Set to the same value as WEBHOOK_SECRET above.
secret = "supersecret123"

# We're going to trigger the webhook when an asset is analyzed
triggers = [boonsdk.WebHookTrigger.AssetAnalyzed]
```

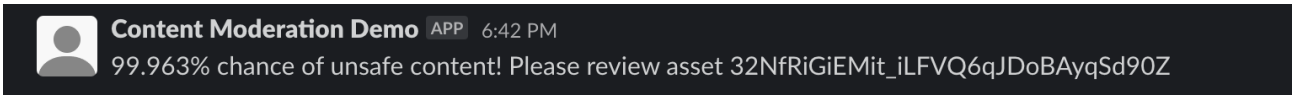
```

app.webhooks.create_webhook(endpoint, secret, triggers)

```

Testing our Solution

Now it's time to test our solution by uploading an image to see if our Slack Channel gets notified.

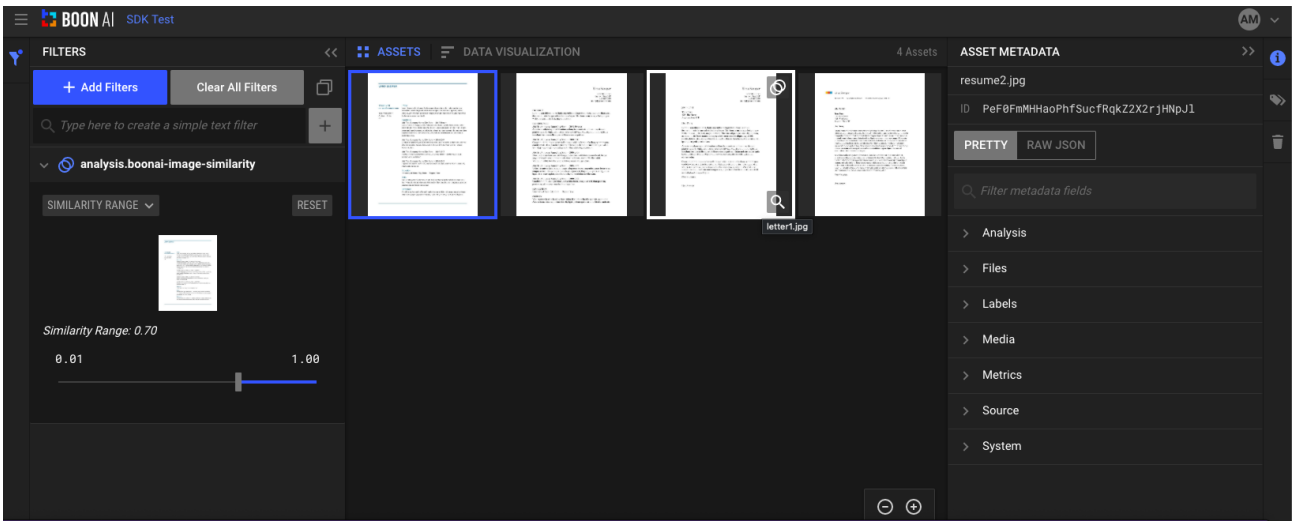
```
app.assets.batch_upload_files(
    boonsdk.FileUpload("drugs.png"),
    modules=["clarifai-unsafe-detection"])
```

Here is an example of the automated moderation post for an image that is very likely to contain unsafe content!



Single-Shot Document Classification

Let's assume you have a job hunting website where applicants upload resumes and cover letters. You would like to train a model that correctly identifies each type of document.



The KNN model we will use can be trained with very few examples assigned labels.

The image above shows a dataset comprised of four images to be used in our document classification model. Because this type of classification can be accomplished with very little training information provided, we will use two images labeled as cover letters and two images labeled as resumes in the demonstration.

Example Code

The latest working code for this example can be found in GitHub.

examples/solutions/single_shot at main · TheBoonAI/examples	https://github.com/TheBoonAI/examples/tree/main/solutions/single_shot

Setup

First let's set up our API key and instantiate an BoonApp instance that points to the key we've created for this project.

```
import boonsdk

app = boonsdk.app_from_keyfile("apikey.json")

```

Dataset

For this task we're going to create a Dataset which contains just a couple of examples representing both cover letters and resumes.

```

# Hold onto this dataset because we'll need this later on.
dataset = app.datasets.create_dataset(
    "job-documents",
    boonsdk.DatasetType.Classification
)

```

Now we'll import our example files and pre-label them as part of our dataset. You can also add more labeled assets using either the API or UI at a later time.

```

# If you have downloaded this file, be sure to provide the path to your lo
path_to_csv = "images.csv"

csvfile = boonsdk.CsvFileImport(path_to_csv,
                                uri_column=0,
                                dataset=dataset,
                                label_column=1,
                                max_assets=1000,
                                header=True)

app.assets.import_csv()
```

Model

Once our Dataset is imported, it's time to create and train a model that will predict the type of future documents. For this case we're going to train a KNN Classifier model and attach the Dataset we created earlier.

```

model = app.models.create_model(
    "document-classifier",
    boonsdk.ModelType.KNN_CLASSIFIER,
    dataset=dataset
)

```

Training

Now that we have a Model with a linked Dataset we can train the model. The `train_model()` function will return a job ID that can be polled to check when the training is complete. Training a KNN Classifier is very quick so we shouldn't have to wait too long.

```
app.models.train_model(model)
```

Test the Model via the API

Let's test our model using a sample image.

```
result = app.assets.analyze_file("test-resume1.jpg", ["document-classifier
print(result.get_analysis("document-classifier"))

{
  'count': 1,
  'type': 'labels'
  'predictions': [
    {
      'label': 'resume',
      'score': 0.595
```

```
      }
    ]
  }
```

Improving the Model

There are many different styles of resumes and and you might eventually run into a case where your model fails to recognize the correct type of document. When this happens you can improve your model's performance by labeling more examples and retraining your model.

Calculating Custom Field Values with Boon Functions

Introduction

Often, the results from a pre-trained or out of the box ML model are not exactly what we are looking for. In many cases the results from these models need to be processed further in order to be useful for a your specific workflow.

In the following example we're going to use a Boon Function to calculate the area, in pixels, of the largest face detected in an image. Then we will save that value into a custom field. This allows us to filter our library of assets to show the best faces to use for recognition training.

Code

The code for this solution can be found here:

examples/solutions/face_area_calc at main · TheBoonAI/examples
https://github.com/TheBoonAI/examples/tree/main/solutions/face_area_calc

The Boon Function

A Boon Function is comprised of a python file named **function.py** with a single function called process() which takes an Asset object. You may include other files with your Boon Function but function.py is the only requirement.

> You may include a requirements.txt file for any additional Python dependencies.

```
function.py
```

```python
# The simplest Boon Function!
def process(asset):
    pass
```

Let's augment our Boon Function with the logic we'll need to accomplish our goal.

function.py

```python

from boonsdk.util import denormalize_bbox
from boonsdk.func import FunctionResponse

def process(asset):

    # First grab the height and width of the asset itself, we'll need thes
    # calculate the area later on.
    width = asset.get_attr("media.width")
    height = asset.get_attr("media.height")

    # There could be multiple faces, so we'll stash the area of our bigges
    # face into the big_face variable.
    big_face = 0

    # Not iterate detected faces, calculate the area and keep track of the
    attr = "analysis.boonai-face-detection.predictions"
    faces = asset.get_attr(attr)
    if faces:
        for face in faces:
            bbox = denormalize_bbox(width, height, face['bbox'])
            area = abs(bbox[0]-bbox[2]) * abs(bbox[1]-bbox[3])
            if area > big_face:
                big_face = area

        if big_face:
            # Take our big face and set the value of the face_bbox_area cu
            rsp = FunctionResponse()
            rsp.set_custom_field("face_bbox_area", big_face)
            return rsp

    # It's ok to return None to indicate no modifications to asset.
    return None

```

Deploying The Boon Function

In order to deploy our Boon Script we must create a Model. This may seem strange since our Boon Script is not doing any inference (but it could). We're going to take advantage of the Model deployment system in order to deploy our code.

```
model = app.models.create_model(
    "face-area-calc",
    boonsdk.ModelType.BOON_FUNCTION,
    dependencies=['boonai-face-detection']
)
```

Next we need to package our script.py file into a zip file. For this we'll use the zip utility on the command line.

Bash

```
zip model.zip function.py
```

Now we upload the file "model.zip" to deploy it.

```
app.models.upload_pretrained_model(model, "model.zip")

# Wait on the boon function to be deployed.
app.models.wait_on_deploy(model)
```

Uploading the model will kick off the model deployment process which may take a few minutes. While that is happening, let's create the custom field we're going to use to store the new data.

The Custom Field

In order for BoonAI to store this data, we need to create a custom field called face_bbox_area and set its type to be an integer. This requires just 1 line of code.

```

app.fields.create_field("face_bbox_area", "integer")

```

Testing

Let's test our newly deployed Boon Script by uploading a single image.

```
rsp = app.assets.batch_upload_files(
    boonsdk.FileUpload("face.jpg"), modules=["face-area-calc"])

if app.jobs.wait_on_job(rsp):
    asset_id = rsp['created'][0]
    asset = app.assets.get_asset(asset_id)
    print(asset.get_attr("custom.face_bbox_area"))
else:
    print("Oh no our job failed!")
```

Automatically organizing a set of images into groups

Introduction

The KNN classifier makes it easy to organize a previously unseen set of images into groups that are semantically consistent. In this section we will look at a workflow that does just that.

We are going to use images from the USDA Pomological Watercolors dataset. This is a set of almost 7500 watercolor paintings of fruit, painted between 1886 and 1942.

Code

The full code for organizing these images can be found here.

Data Preparation

First let's set up our API key and instantiate a BoonApp instance that points to the key we've created for this project.

```
import boonsdk

app = boonsdk.app_from_keyfile("apikey.json")
```

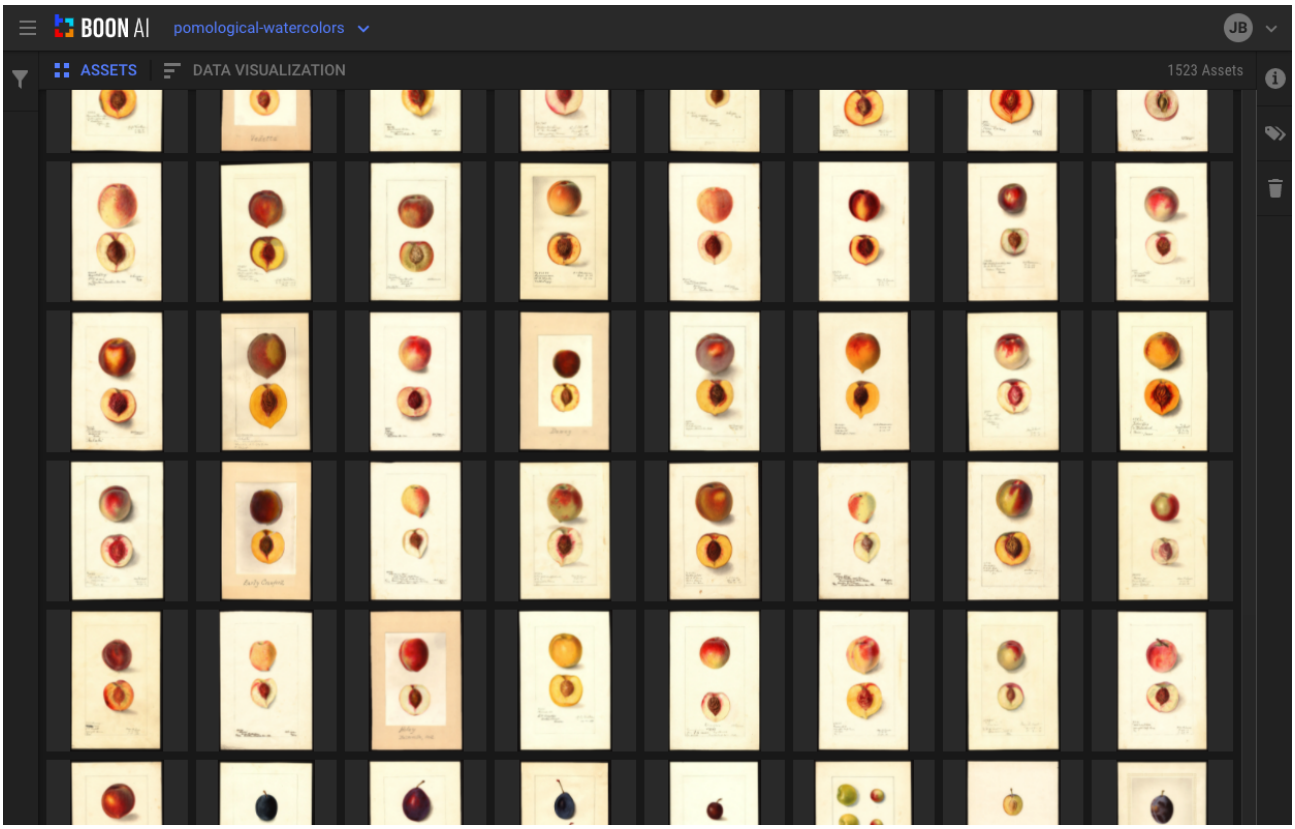
Then we can import some images. A JSON file with data for each watercolor has been provided, so we can use that to loop through the entries and set up an import:

```
import json

with open('fruits.json', 'r') as file:
        fruits = json.load(file)

files = []
```

```
for f in fruits:
    name = f['url'].split('id=')[1]
    imageURL = 'http://naldc-legacy.nal.usda.gov/pom/' + name + '/screen.j
    files.append(FileImport(imageURL))

while files:
    app.assets.batch_import_files(files[:100])
    files = files[100:]
```

This will import about 1500 images, and will take a few minutes. You can monitor the import jobs using the BoonAI console.

Once the import is done, the Visualizer in the BoonAI console should show something like this:

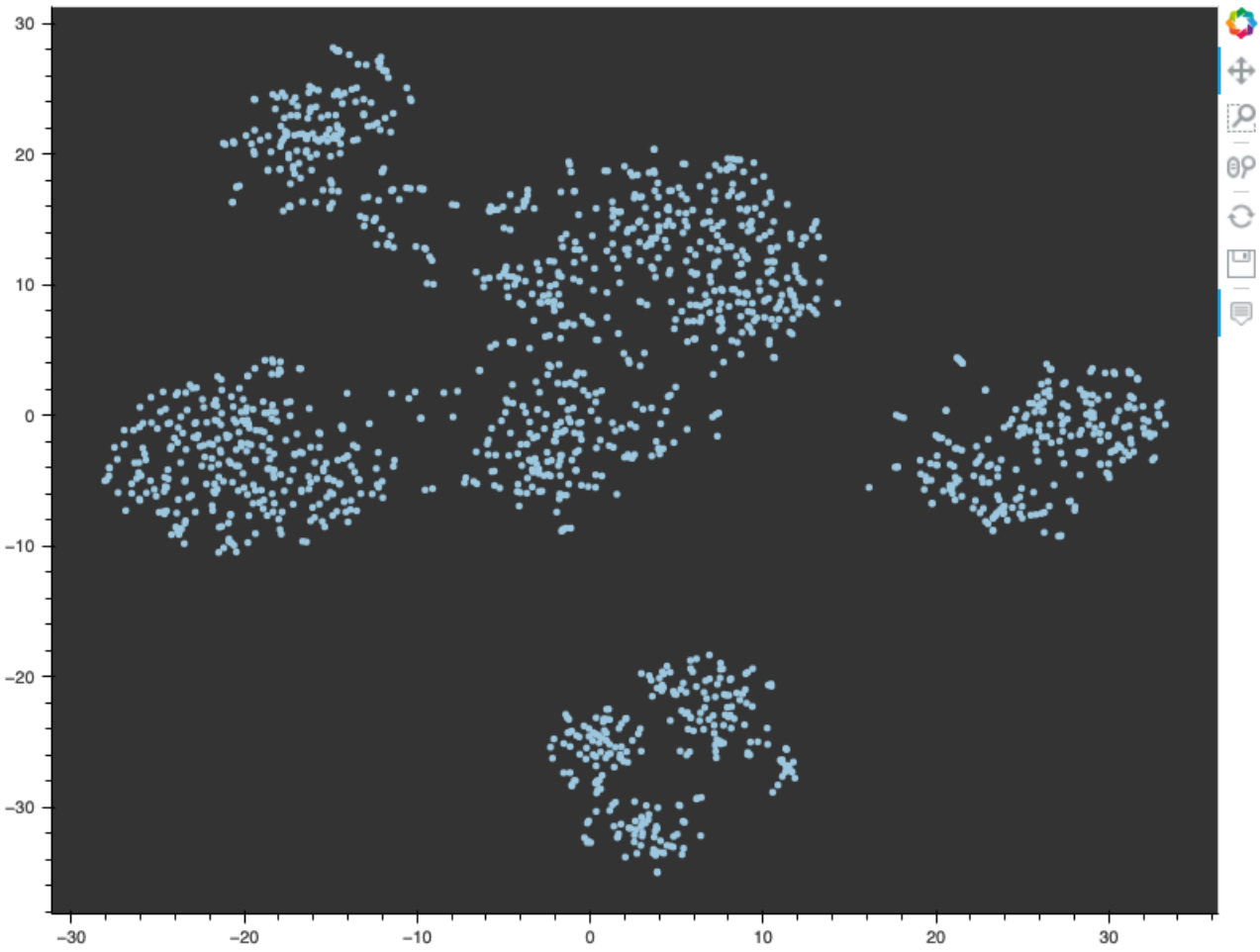


Preliminary Analysis

The boonlab library provides `plot_tsne()`, which is a nice way of looking at a collection of images as a whole. We are going to use it in order to see if there are groups of images that make sense to organize around, and in that case, to decide how many groups we want to use.

This function will plot the assets returned in a search (if no search is provided, all assets are used) as a t-SNE map. By default it uses the semantic similarity hash that is computed for every asset. This hash is a 2048-dimensional vector, and the t-SNE process computes a two-dimensional representational plot of those hashes. This visualization is useful because points that are shown close to one another represent images that are similar to each other. If you see clusters, or groups of points that are distinct, that means that the similarity hash will be useful to group the images meaningfully.

```
from boonlab.plot import plot_tsne
x = plot_tsne(thumbs=False, nClusters=1, verbose=False)
```

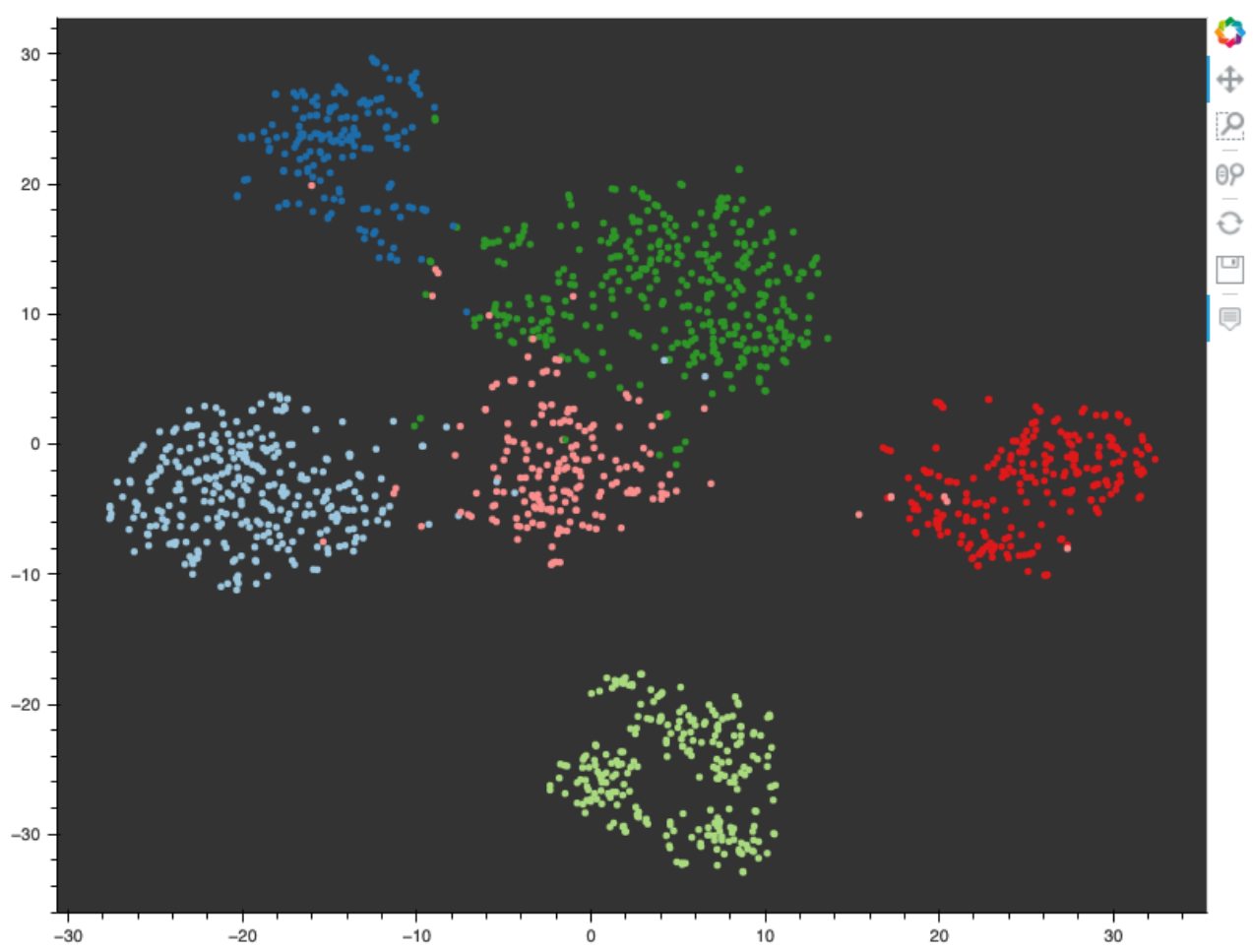


The plot created in this step is interactive, which makes it easy to zoom in when you want examine areas more closely. Another option you have is to set `thumbs=True` which will download thumbnail images and use them to display as a tooltip when you hover the mouse over each point. This can be extremely useful if you want to see exactly what

different points on the plot represent. This feature also makes it easy to identify which assets are outliers. Note that `thumbs=True` will take a long time when generating and downloading the thumbnails for the first time. After that initial download, extra plots will take less time to produce because the use a cache.

Here we can count six more or less distinct clusters. The `plot_tsne` library has a clustering option, so we can make use of that, setting `nClusters` to equal 6, so we can investigate these clusters further:

```
x = plot_tsne(thumbs=True, nClusters=6, verbose=False)
```



KNN Model

Now we can do the actual automatic organization of these assets. We are going to use a K-Nearest Neighbor, or KNN model for this. The KNN model type provided by Boon AI allows

you to train it without any labels assigned. When that happens, the model does a clustering just like the one we did above, finds the centroid points for each cluster, and uses those as labels to train the model. The result is a set of groups that are created automatically.

First we will create our model. We need to create a dataset as well, but we will leave that without any labels assigned. After the model is created, we can launch a training job and finally, apply the model to the assets. When training, we will specify that the model should divide assets among six clusters, which is the number we observed in our previous analysis.

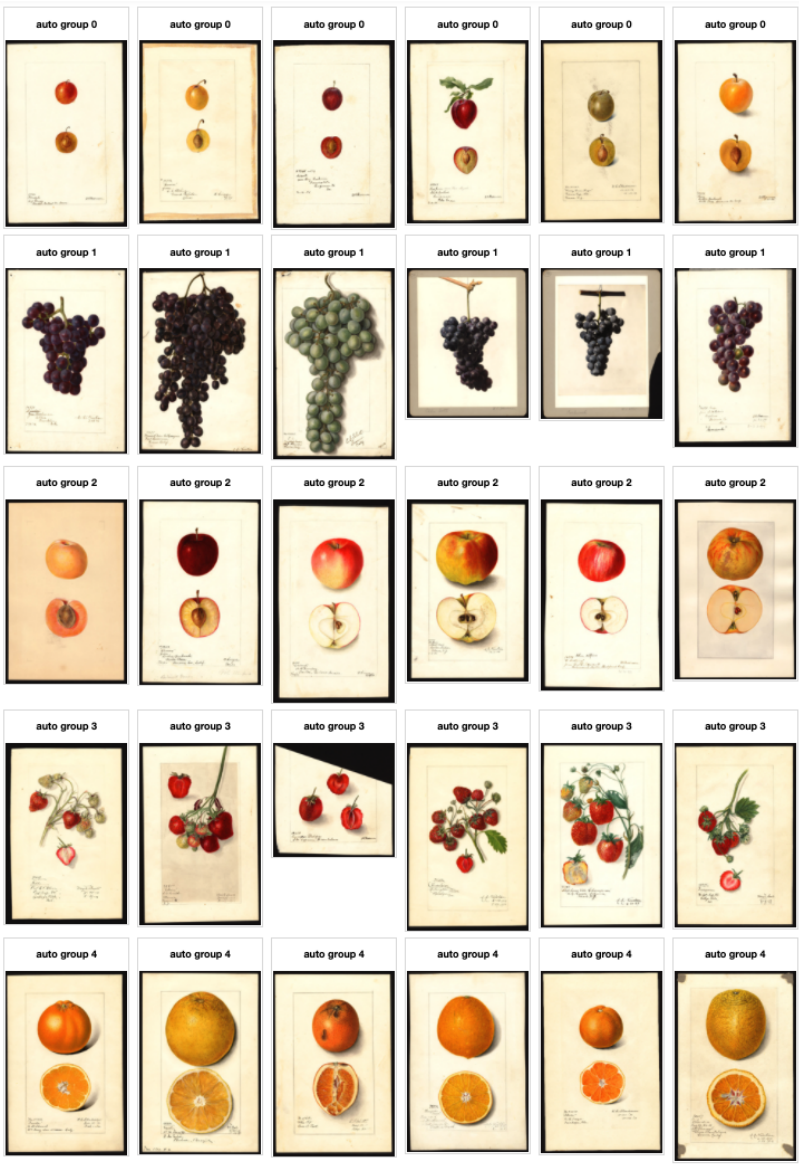
The following code does all that:

```
# Create an empty dataset
dataset = app.datasets.create_dataset(
    "fruits",
    boonsdk.DatasetType.Classification
)

# Create a model and attach our empty dataset
model = app.models.create_model(
    "fruit-groups",
    boonsdk.ModelType.KNN_CLASSIFIER,
    dataset=dataset
)

# Launch the training job, with six clusters
training_job = app.models.train_model(model, train_args={'n_clusters': 6})

# Wait for training to complete
app.jobs.wait_on_job(training_job)

# Apply the model to our assets
app.models.apply_model(model)
```

Once the model is trained and applied, we can visualize the results:

```
import ipyplot
from PIL import Image

images = []
labels = []
```

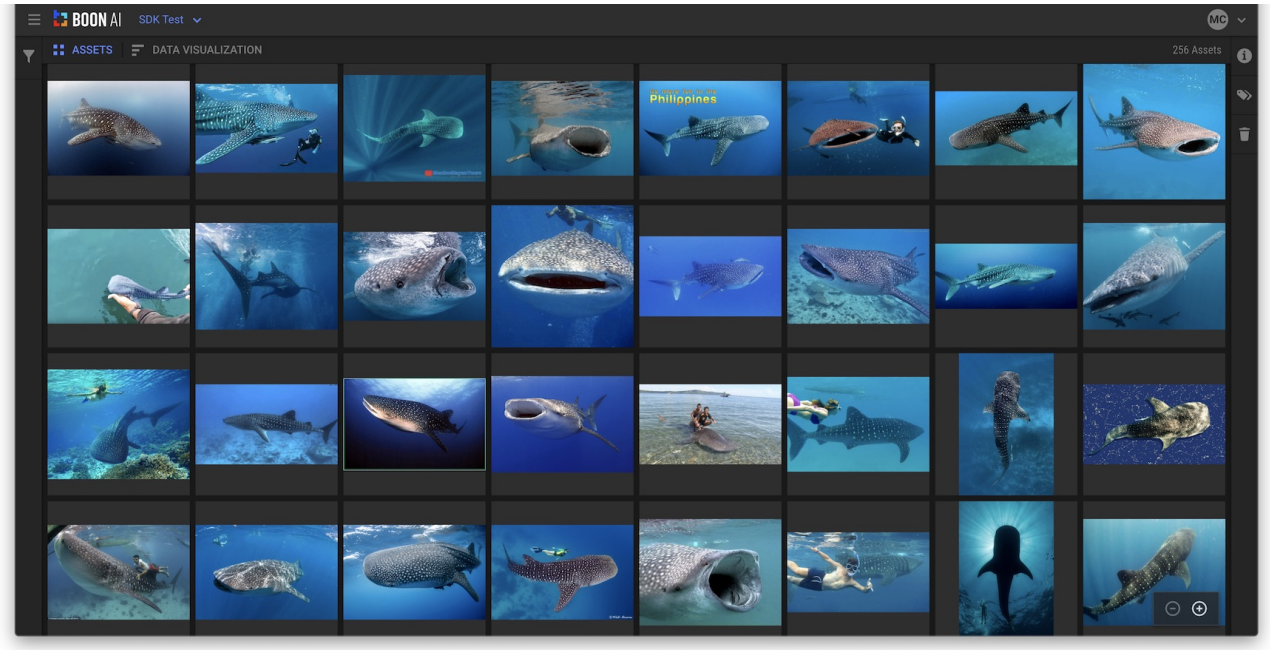
```
6
7  for group in range(0, 6):
8      search = {"size": 6, "query":{"bool":{"must":[{"simple_query_string":{
9
10     for asset in app.assets.search(search):
11         thumbnail = asset.get_files(category="web-proxy")[0]
12         images.append(Image.open(app.assets.download_file(thumbnail)))
13         labels.append(asset.get_attr("analysis.fruit-groups.predictions")[
14
15 ipyplot.plot_images(images, labels, img_width=150)
```



As illustrated in the image above, each one of the six groups corresponds to a fruit type.

Because the training job actually creates one label per cluster, it is possible to use the Visualizer in the Boon AI console to rename those automatically created labels to more sensible names. So you could name the groups above 'Cherries', 'Grapes', 'Apples', Strawberries', and 'Oranges'. You can also add more labels, delete less useful labels, and customize the way you label labeling these assets to whatever degree needed, depending on your project's goal.

Classifying Sharks with EfficientNetB1



Introduction

In this solution walkthrough we're going to use transfer learning to train an EfficientNet B1 model to identify species of sharks. EfficientNet is currently the state of the art model architecture used for predicting Imagenet labels, which is a massive library of photographs used to initially train our EfficientNet B1 model. Using a technique called "transfer learning" we're going to retrain this model to only care about 2 species of sharks, thus taking advantage of the fact the model is already very good at identifying realistic photography.

The dataset we are going to use a Shark Species dataset on Kaggle which you can download here. There is also a link in the Notebook. In the interest of saving time, we're not going to train for all species of sharks in the dataset. Instead, we'll focus on only Bull and Whale sharks. If you would like to expand on this exercise, it is possible to use the techniques demonstrated here to train for all species.

Code

The full code for training a shark species classifier with BoonAI can be found here:

examples/solutions/shark_classifier at main · TheBoonAI/examples

https://github.com/TheBoonAI/examples/tree/main/solutions/shark_classifier

Data Preparation

Assuming you've already downloaded the dataset and you have a local copy of the `archive.zip` file which is about 600MB, the next thing you'll want to do is unzip it and then delete the zip.

```
unzip archive.zip
rm archive.zip
```

This should leave you with a directory named "sharks" which contains the images we will use to training the model.

Setting up the Boon SDK

For this solution we're going to use a command line tool called `boonctl` to quickly import the images for whale sharks and bull sharks into our project. `Boonctl` requires our shell be configured with an environment variable that points the BoonSDK to an api key. In the following line of code, we're assuming the key is in the current directory. Make sure the code you write points to the location of your downloaded API key.

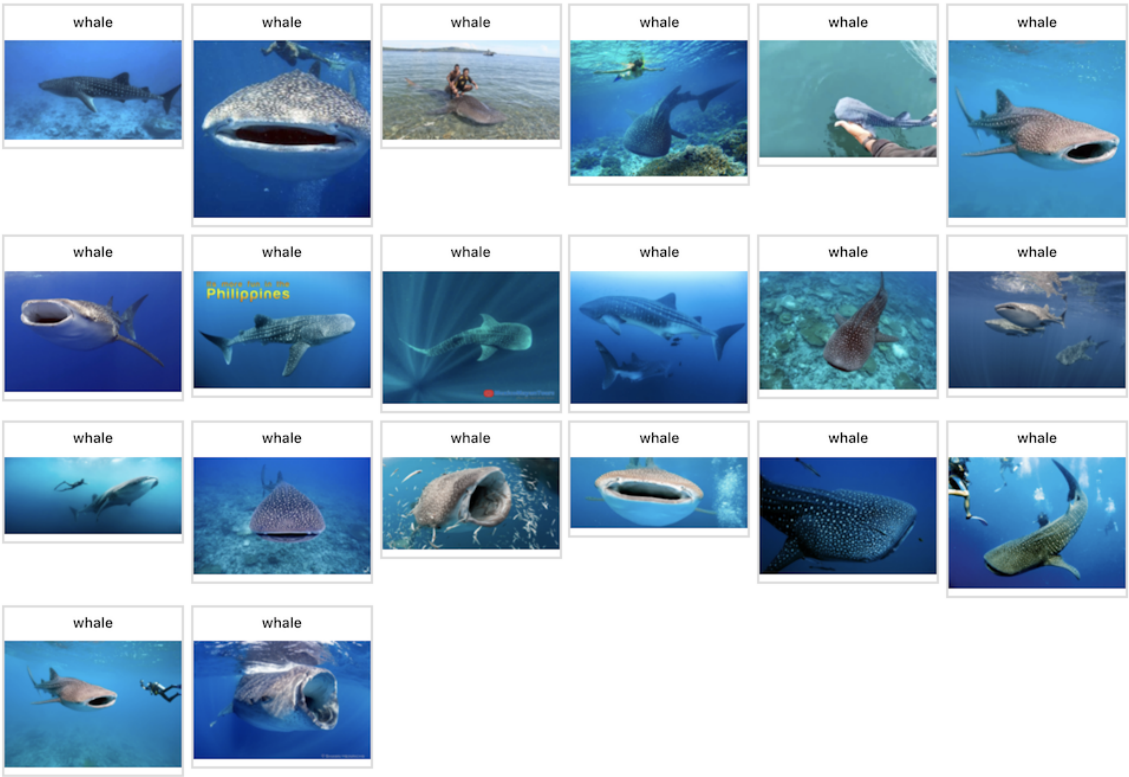
```
export BOONAI_APIKEY_FILE="apikey.json"
```

Now we can issue the `boonctl` command. This command will automatically create a Dataset called 'sharks' and import pre-labeled images for whale sharks and bull sharks. Additionally `boonctl` will label some images for training while setting others aside for testing.

```
$ boonctl datasets import classification sharks -q -l bull -l whale
Detected Name: sharks (exists: False)
Detected Labels: ['hammerhead', 'bull', 'nurse', 'mako', 'blue', 'lemon',
Ok? (ctr-c to abort)
Importing 106 "bull" assets. Train: 95 Test: 11
Importing 133 "whale" assets. Train: 119 Test: 14
```

> If you want to train for all species of sharks in the Dataset, just remove both the -l flags from the `boonctl` command.

If you are using the notebook provided at the top of this page you can visualize the assets once the import job has completed.



Creating and Training the Model

Training this model should take about 6 minutes to complete. Note that the model type is a `TF_CLASSIFIER`. The TF stands for Tensorflow, which is a popular library for training

models. The default base model type for a `TF_CLASSIFIER` is EfficientNetB1 however this is configurable along with various other settings.

```
# Fetch the dataset
ds = app.datasets.get_dataset("sharks")

# Create the model
model = app.models.create_model("shark_classifier",
    boonsdk.ModeleType.TF_CLASSIFIER, dataset=ds)

# Launch the training job
training_job = app.models.train_model(model)

# Wait for training to complete
app.jobs.wait_on_job(training_job)
```

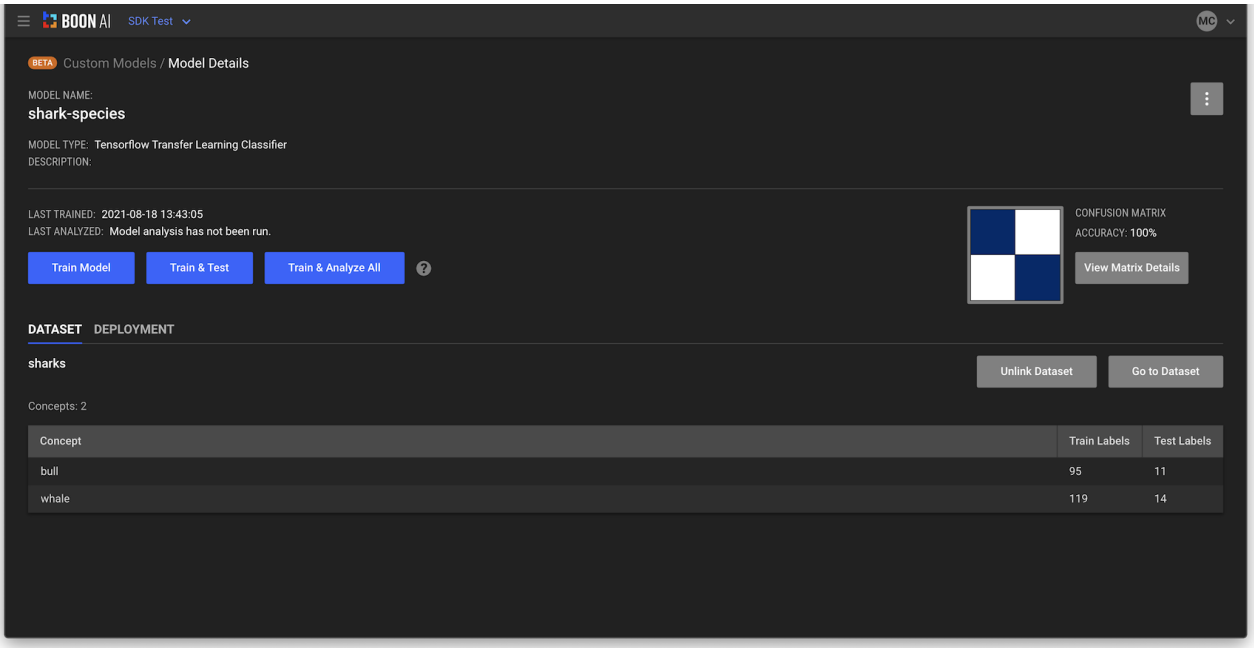
Testing the Model

Once our model is trained we can test it to ensure it's making the most accurate predictions possible. You may remember that when we imported the pictures of sharks, some of them were tagged as test images. These images were not used to train the model. They were held back so we would be able test the model by showing it data that it's never seen before.

Testing the model will produce a confusion matrix which will give us a visual way of understanding how well the model performed.

```
test_job = app.models.test_model(model)
app.jobs.wait_on_job(test_job)
```

Once our test job is complete, you can visit the model's page in the BoonAI console to see the confusion matrix. 100% accuracy!


BOON AI
SDK Test
MC
BETA
Custom Models / Model Details
MODEL NAME:
shark-species
MODEL TYPE: Tensorflow Transfer Learning Classifier
DESCRIPTION:
LAST TRAINED: 2021-08-18 13:43:05
LAST ANALYZED: Model analysis has not been run.
Train Model
Train & Test
Train & Analyze All
CONFUSION MATRIX
ACCURACY: 100%
View Matrix Details
DATASET
DEPLOYMENT
sharks
Unlink Dataset
Go to Dataset
Concepts: 2
Concept
Train Labels
Test Labels
bull
95
11
whale
119
14

Supported File Types

These files types are supported by the Boon AI system.

File extension	Description
Document files	
.doc and .docx	Microsoft Word document file
.pdf	Portable Document Format (PDF) document file
.ppt and .pptx	Microsoft PowerPoint document file
.vdw	Microsoft Visio web drawing file
.vsd	Microsoft Visio diagram file
.vss	Microsoft Visio smartshapes data file
.vst	Microsoft Visio flowcharts file
.xls and .xlsx	Microsoft Excel document file
Image files	
.bmp	Bitmap (BMP) image file
.cin	Kodak Cineon (CIN) bitmap file
.dpx	Digital Moving-Picture Exchange (DPX) bitmap file
.exr	OpenEXR High Dynamic-Range bitmap file
.gif	Graphics interchange format (GIF) image file
.jpeg and .jpg	Joint Photographic Experts Group (JPEG) image file
.png	Portable Network Graphics (PNG) image file
.psd	Adobe Photoshop (PSD) image file

.la	Alias Wavefront Raster bitmap file
.tif and .tiff	Tagged image file format (TIFF) image file
Video files	
.m4v	Apple MP4 video file
.mov	Apple QuickTime video file
.mp4	MPEG-4 video file
.mpeg and .mpg	Moving Picture Experts Group (MPEG) video file
.mxf	Material eXchange Format (MXF) video file
.ogv	Ogg Vorbis video file
.webm	Matroska WebM video file

Typographical Conventions

Typographical Conventions

These are the typographical conventions used throughout this document.

Type style	Description
Bold font	GUI elements
Italics font	File names and file paths
Light blue text (in code examples)	Variables that you should substitute with real values
`Monotype` font highlighted in light blue (in text)	Programming code

Abbreviations

Abbreviation	Definition
AI	artificial intelligence
API	application programming interface
ASR	automatic speech recognition
AWS	Amazon Web Services
CNN	convolutional neural network
COCO	Common Objects in Context
DLP	data loss prevention
DPX	Digital Moving-Picture Exchange
GCS	Google Cloud Storage
GIF	graphics interchange format
GUI	graphical user interface
JPEG	Joint Photographic Experts Group
KNN	K-Nearest Neighbor
MIME	Multipurposed Internet Mail Extensions
ML	machine learning
MXF	Material eXchange Format
OCR	optical character recognition
MPEG	Moving Picture Experts Group
PDF	Portable Document Format
PNG	Portable Networks Graphics

PSD	Adobe Photoshop document
RGB	red, green, blue
TIFF	tagged image file format
URI	universal resource identifier
UTC	universal time coordinated
WebVTT	Web Video Text Tracks

Glossary

analysis module

The building blocks used to create a custom media processing pipeline, which can augment assets with AI-generated metadata.

asset

The result of importing source files from a remote repository into the Boon AI system.

attribute

A property that is stored on an asset. Assets contain a hierarchical structure of metadata attributes.

data source

A combination of the address of a remote repository where source files are stored and the ML operations to be applied to them.

data visualization

A representation of specific information from a dataset that resulted from a data source creation.

field

A storage location for attributes.

[To be relocated]

Fields are defined by strict Elasticsearch mapping used for all Boon AI projects. You can add your own fields with these specific data types:

- text, keyword
- long, integer, short, byte, double, float, half_float, scaled_float
- date
- Boolean
- integer_range, float_range, long_range, double_range, date_range
- binary

> Fields are limited to 150 per Boon AI project.

label scope

Whether a label is a training label or a test label. Boon AI uses test labels to generate confusion matrices for the model you're testing.

model

An algorithm for making predictions.

module

See "analysis module"

prediction label

A predicted value according to a given model that run on the image.

similarity hash

A quantized feature vector, or embedding. The Boon AI platform uses these to do similarity search and to train fast KNN image classifiers.

source files

The files, located in a remote repository, that are imported into the Boon AI system for analysis. Also called *source data*.

test label

A label assigned to an asset within a dataset, which is not used for training. Test labels are there so the model can be tried with images that it has not seen during training.

training label

A label assigned to an asset within a dataset, used for training models.