pyveda Documentation

Release 0.0.5

Chris Helm

May 31, 2019

Contents

pyveda is a package for working with machine learning data, models and inference sets stored in Veda.

Warning: pyveda is beta software, and these docs are a work in progress

CHAPTER 1

Veda

Veda is DigitalGlobe's platform for machine learning data. Veda's goals are:

- Generate sample data from satellite imagery
- Share and distribute organized sets of data
- Share and distribute machine learning models
- Use the models and data to perform inference

pyveda is MIT licenced.

Installation:

```
pip install pyveda
```

For general information on the Veda platform and API visit http://github.com/DigitalGlobe/veda.

1.1 Getting started

Veda is a library of object knowledge for machine learning.

Veda stores organized collections of images and their labels. It is integrated with the GBDX library of imagery and can access almost 20 petabytes of satellite images to use for generating image chips. Data sets and models can be published and shared with others.

Veda is spatially aware.

Veda "speaks" geojson and can convert spatial data into machine learning data and back. Use spatially-targeted training sets to develop location-specific models.

Pyveda is the bridge between Veda and machine learning platforms

Pyveda is a Python access library that lets you generate and manage machine learning data. When you're ready to train or test your model with that data, pyveda supplies the data to your algorithms in efficient formats that are ready to analyze. When your model is trained, you can also use pyveda to upload the model to Veda.

Veda does not train or run models - your choice of library and platform is up to you.

1.1.1 Getting access to Veda

Access to Veda requires GBDX credentials. You can sign up for a GBDX account at https://gbdx.geobigdata.io.

Your GBDX credentials are found under your account profile.

PyVeda's authorization tools expect a config file to exist at ~/.gbdx-config with your credentials. (See formatting for this file here: https://github.com/tdg-platform/gbdx-auth#ini-file.)

For questions or troubleshooting email GBDX-Support@digitalglobe.com.

1.1.2 Installing pyveda

```
pip install pyveda
```

1.2 Accessing Data from Veda

Veda stores machine learning data collections. These can be generated from DigitalGlobe's satellite imagery, or can be imported from other sources. For example, Veda hosts copies of the XView dataset.

So how do you get that data out of Veda and into your models?

1.2.1 Veda's Data Storage

Let's start with looking at how Veda stores data.

Note: The language used here is in flux - we use *collection* and *dataset* to mean the same thing. Similarly, *sample* and *datapoint* get interchanged.

Samples

The basic unit of training data is called a sample. It consists of an image, and a label that describes the features in the image. Veda supports three types of labels:

- Classification
- Object Detection
- Segmentation

Classification labels indicate whether a class of features exists in the image. The value is a boolean, where a true value for a class means the class is present.

Object Detection labels describe where features are located, if they are in the image. The value is a bounding box of pixel values, so each label would indicate where in the image the class feature exists.

Segmentation labels are similar to Object Detection labels, but instead of a bounding box, the feature is described by a polygon object that can trace the outline of the feature.

Sample objects also store geospatial information if it's known. Any data generated by Veda will have spatial data, but it's also possible to import ungeoreferenced images.

See the *Label Formats* section for more information about how labels are used in pyveda.

Collections

All of the samples are stored in groups called collections that keep things sorted and organized. A collection would group data around a theme or area. Collections have unique names to identify them. They also enforces consistency in the data. Some of the requirements of a collection are:

- All images must be the same size in pixels
- All images must have the same number of bands
- All images must share the same data type
- All labels must be the same type - classification, object detection, or segmentation.

Collections also track their contents. A collection knows all of the classes represented in the samples' labels. It also tracks the spatial bounds of all the samples.

1.2.2 Finding Collections with pyveda

The first step to working with data stored in Veda is to find a collection to work with. If you're following a tutorial or a notebook the name of the collection should be given. You can use the name to access the collection. This is covered in the section *Accessing Collections*.

If you'd like to find a collection to investigate, pyveda has the *search* method (see `pyveda.main.search()`). Let's start with the simplest example:

```
import pyveda as pv
pv.search()

[VedaCollectionProxy of Las Vegas Buildings (SpaceNet2) (84810fa8-87e2-4f22-9386-
↪4406b4c3add0), VedaCollectionProxy of X-View Data Complete (b030111a-f13f-40d4-90fb-
↪3dcb826069d8)]
```

The *search* method searches Veda for collections and can take key words and filters. It returns a list of VedaCollectionProxy objects, which is a pyveda object that acts as an entry point to working with the collection. We can take a closer look at the collections returned by the search:

```
results = pv.search()
print(results[0])

X-View Data Complete (b030111a-f13f-40d4-90fb-3dcb826069d8)
    - Bounds: [-99.09676905613048, -34.9111829655798, 151.23452243531614, 53.
↪92520692619612]
    - Count: 173075 samples, 87% cached
    - Chips: 256x256, 3 bands int8
    - Labels: object detection type, 113 classes
```

These collection objects have an informative `__str__` method that prints collection details in a readable format. You can use the `repr` function to print the shorter representation of just the name and ID.

To search for terms, pass them to the search function:

```
pv.search('building')

[VedaCollectionProxy of Las Vegas Buildings (SpaceNet2) (84810fa8-87e2-4f22-9386-
↪4406b4c3add0)]
```

Filters can also be added. The queries use ElasticSearch syntax:

```
pv.search('building', )
```

1.2.3 Accessing Collections

If you have a run a search, you can work directly on the VedaCollectionProxies returned by the search.

```
results = pv.search('building)
collection = results[0] # get the first search result
# do something with the collection
```

If you have a name or ID of a collection, you can access it with these methods:

```
collection = pv.from_name('X-View Data Complete')

collection = pv.from_id('b030111a-f13f-40d4-90fb-3dcb826069d8')
```

1.2.4 Working with the Samples

Once you have a collection in the form of a VedaCollectionProxy, you can treat it like a list of datapoints. It supports indexing and slicing:

```
sample = collection[0] # the first sample
print(sample)

{'mltype': 'segmentation', 'cached': 'true', 'dataset_id': '84810fa8-87e2-4f22-9386-
↪4406b4c3add0', 'bounds': [-115.2847026, 36.1669076998, -115.2829476, 36.1686626998],
↪ 'dtype': 'uint8', 'tile_coords': [12......
```

If you look representation of the sample object (cut short here), you'll see it's metadata and label data. We can also look at the some of the sample's specific properties:

```
>>> rr.classes
['buildings']

>>> rr.label
{'buildings': [{'type': 'Polygon'.........

>>> rr.image
[image]
```

The sample object also has some basic CRUD methods:

```
# a modified label
label = {buildings: [.......
# update the sample with the new label
sample.update(label)
```

(continues on next page)

(continued from previous page)

```
#delete a sample
sample.delete()
```

1.2.5 Using Samples with Models

In most machine learning examples, you have to download all of the sample image files and labels to a local directory. Then you have to split up your data into training and testing groups, and probably also convert the data into array formats that can be read as inputs.

We saw in the previous section how to look at the samples in the collection. So what's the best way to get that sample data to your model? You could certainly loop through all the samples and save their images and labels to files.

Pyveda does provide a way to download the image and label files - see the *Releases* section. Pyveda makes it easy to get the data in this format - with a single command you get a link to download the compressed file. But pyveda also provides better ways to access the data locally.

To fetch a local cache of the data, use the *store* method. This downloads the sample data to a local file called a VedaBase. Instead of directories of images, Vedabases are single files that hold optimized sample data. All of the sample points have been converted to the correct array formats: the images are numpy arrays, and the labels are one-hot encoded. The *store* method takes a collections name or ID, and a path to store the file to. It then uses a multithreaded downloader to fetch all the data. You will need to wait for this step to complete before proceeding.

```
pv.store('./lasvegas.vdb', dataset_name='Las Vegas Buildings (SpaceNet2)')
```

You can open a saved VedaBase with the `open` function. We recommend using a Vedabase with a context manager:

```
with pv.open(filename='./lasvegas.vdb') as data:
    # do something with the data
```

The `store` method also works as a context manager, but you still will have to wait for the data to download locally.

1.2.6 Collection Groups

VedaBases internally store separate training, validation, and testing groups, so you don't have to divide the data. The ratio of sample is each group is set by the *partition* property of the collection.

Each group is an accessor on the collection. You can access samples in the group just like a list, through iteration or indexing. The value returned is an `(image, label)` tuple. As mentioned above, the image and label have been coverted to the appropriate arrays already.

```
data.train[0] # the first sample in the training group

for image, label in data.test:
    # do something with each image and label in the testing group
```

The collection also lets you access the images and labels separately:

```
for label in data.test.labels:
    # do something with just each label in the testing group
```

This means that a typical training example might look like:

```
from SuperMLDetector import MegaModel

megamodel = MegaModel()

with pv.open(filename='./lasvegas.vdb') as data:
    megamodel.train(data.train)
    megamodel.score(data.test)
```

1.2.7 Batching and Transforming Data

Pyveda includes tools for batching and transforming data. The three groups have a `batch_generator` method that returns a generator that emits batches of data. Optionally the generator can randomly flip the sample, rescale the values of the image, and move the channel data to the end of the numpy array. The basic use of the generator is to pass the size of the batch.

```
# generate batches of 32 samples
gen = vb.train.batch_generator(32)

for x, y in gen:
    # x and y will have a length of 32
```

The batcher takes the following optional arguments:

- shuffle (Boolean): return the samples in the group in random order. Default is True.
- channels_last (Boolean): return image data as Height-Width-Depth. Default is False (RDA returns imagery with the bands last).
- rescale (Boolean): rescale image values between 0 and 1. Default is False.
- flip_horizontal: Boolean. Randomly flip image and labels horizontally. Default is False.
- flip_vertical: Boolean. Randomly flip the image and labels vertically. Default is False.

When using either flip option there is a 50% chance an image will be flipped in a given direction.

To create a batcher that will return 16 samples at a time, rescaled to a range of 0 to 1, and flipped randomly in either direction:

```
gen = vb.train.batch_generator(16, rescale=True, flip_horizontal=True, flip_
↪vertical=True)
```

1.2.8 Streaming Remote Data

Warning: This direct access will probably be removed so that data can be accessed on demand while being locally cached. Because the samples aren't cached, they are loaded every time they are accessed. Using this over many epochs will be slower than saving a local file.

For preliminary testing, or algorithms that only need to work with the datapoints once, it's possible to open the collection directly from Veda, and the samples are loaded on demand. This uses the same *open* function as loading a local file, and data access is performed the same.

```
with pv.open(dataset_name='Las Vegas Buildings (SpaceNet2)') as data:
    # do something with the data or

with pv.open(dataset_id='84810fa8-87e2-4f22-9386-4406b4c3add0') as data:
    # do something with the data
```

1.2.9 Creating Releases

A Veda Release is a compressed archive of the images and labels stored as individual files. You can see more about the Release format and structure in the *Releases* section. To generate a release, run:

```
collection.release()
```

1.3 Creating Data

Note: This tutorial is a work in progress. While Veda is set in its functionality, the pyveda

interface is steadily evolving.

The previous section looked at how to access data from Veda. Now we'll look at how to create your own training data.

1.3.1 Creating New Training Data

Veda is a repository for machine learning data. It can also generate data. It can read spatial data and then generate the images from DigitalGlobe's catalog of satellite imagery. It can also use the spatial data's feature properties to generate labels.

To generate data Veda needs to know three things:

- WHERE is the feature
- WHAT is the feature
- WHICH image to use

The WHERE and WHAT of the features are inputted using the GeoJson format. The image can be any gbdxtools Image object from the DigitalGlobe catalog. Veda will handle generating the appropriate tiles to cover the input features. The same basic concept applies to the three machine learning types that Veda supports: classification, object detection, and classification.

All training data sets consist of image/label pairs. The training images are fixed-sized chips extracted from the source image. The labels vary depending on the type of data being stored:

- **Classification**
  - Classification describes whether an object class exists in the image.
  - Multiple classes are supported.
  - Classification data are stored using "one hot encoding", so for a single class the label is either [1] or [0].
  - For multi-class the labels are lists: [0, 0, 0, 1, 0] (zero for every class not present and 1 for every class that is present).

- **Object Detection**
  - Object detection describes where in an image objects occur.
  - Object Detection data are arrays of pixel based bounding boxes for each feature.
  - Labels are stored as lists of bounding boxes: [[minx, miny, maxx, maxy], [minx, miny, maxx, maxy], . . .]
  - The bounding box values must be in pixel coordinates relative the upper left of the image in the same pair
- **Segmentation**
  - Segmentation also describes where in an image objects or classes occur but uses freeform boundaries instead of boxes.
  - Segmentation data are expected to be arrays of geojson features in the same projected units as the image.
  - These features are converted to pixel locations when saving into Veda and are served back to clients are ndarrays of segmentation data.

See the *Label Formats* for more information and examples.

Warning: This is not totally set up to use like a tutorial yet, sorry!

1.3.2 Sample Data

The PyVeda git repository includes a sample geojson file to get started with. The file `sample_data/turbines.geojson` contains bounding boxes of wind turbines in the Alta Wind Energy Center, located outside of the Mojave Desert in California. Each feature has a property called `label` with the value `turbine`.

For a corresponding image to generate the chips from we'll use catalog ID '123456788' from DigitalGlobe's Worldview 2 satellite. To search for other images covering this area see the "Searching for images" section below.

1.3.3 Creating a Collection from Spatial Data

A VedaCollection stores machine learning data in the Veda system. As mentioned above, it needs the inputs of geojson features and an Image object. For the geojson we'll use the sample wind turbine data. We'll also need a gbdxtools Image object. This could be a CatalogImage or a complicated RDA output; gbdxtools gives you almost unlimited processing options for generating images. The images could be a custom mix of bands or have pansharpening applied, for instance.

For most cases we would like imagery that just has the three visible RGB bands, has been pansharpened for maximum detail, atmospheric effects removed, and dynamic range adjusted to an optimum range. These processing options are all included in PyVeda's `MLImage` class. Only a Catalog ID is needed to create an MLImage object.

First, let's get set up:

```python
import pyveda as pv
from pyveda.rda import MLImage

catID = '123456789'
geojson = 'path/to/data/turbines.geojson'
```

Next, let's create our VedaCollection. It requires some basic parameters to get started:

- The geojson source to use
- The gbdxtools image object to use for the source imagery
- A name for the collection.

There are other parameters to set like the size of the images, but to start we can use the default settings.

```
name = 'Wind Turbines' # give this a more distinctive name
image = MLImage(catID)
vc = pv.create_from_geojson(gojson, image, name)
```

It will take about 5 minutes for Veda to generate all the training data. You can check `vc.status` to track its progress.

The end result will be collection of 256x256 pixel image tiles extracted from catalog ID 123456. Each image will have a corresponding label of the class *turbine*. Because this analysis is classification (the default type), each label will be *1*.

(more stuff about creating VCs)

1.3.4 Creating a Collection using the Bulk Import API

If you would like to import existing image and label data, you can have Veda download and process a compressed archive of the data using the Bulk Import API:

```
pv.create_from_tarball('s3://path/to/tarball', 'Collection Name')
```

The compressed archive needs to follow the pyveda Release format, as described in the *Releases for Importing* section. It also has to be stored in Amazon S3. The path should be given as a *s3://bucket/prefix/file* path.

> Running this command will create a new collection in Veda and it can be accessed with the standard access methods of `pyveda.main.open()` and `pyveda.main.store()`.

1.3.5 Adding Data to Existing Collections

1.3.6 Creating Samples from Scratch

1.4 Features and Labels

Veda features are serialized as geojson features. The machine learning labels are specified in the `Properties` item of the geojson feature. The label format varies based on the machine learning type.

1.4.1 Label Formats

Classification

Input

Geojson features must have a **single** positive class , from either a `Properties` field called `label`, mapped from another field using the `label_field` parameter, or supplied as a default value using the `default_value` parameter. All of these examples add a positive label of the class _house_ to the matching image.

Classification labels are `1` for positive detection and `0` for the absence of the class. It is not needed to have negative classes labelled - if a class is missing it will automatically be labelled as `0`. A shorthand version passing just the positive class name is also acceptable. The class name does not have to be a string as an input, but will be stored as a string.

A full label of a `house` feature, in a VedaCollection that has classes `['house', 'car', 'boat']` is:

```
'Properties': {
    'label': {'house':1, 'car':0, 'boat':0}
}
```

The same feature with a label with the positive value only:

```
'Properties': {
    'label': {'house':1}
}
```

Shorthand version:

```
'Properties': {
    'label': 'house'
}
```

Using the parameter `label_field`:

```
// using label_field='building_type'
'Properties': {
    'building_type': 'house'
}
```

or `default_value`:

```
default_value='house'
```

The parent VedaCollection can determine the classes on the first load of data, or the classes can be set by passing an optional *classes* parameter on initialization, in the case that not all classes are present in the first set of data uploaded. Classes are fixed and can not be modified after the first set of data is loaded.

Classification features can be any geometry type, though it is recommended that multi-geometry types be exploded to individual features.

DataPoint Output

The DataPoint class has a `labels` property that includes all labels in the parent VedaCollection. This is a dictionary keyed with class names, with 0 or 1 labels that indicate positive or negative presence of that class in the image tile. For this and all following examples we'll assume the image tile has houses and cars, but no boats. The DataPoint API JSON response has a labels property in the `Properties` field.

```
datapoint.labels = {
    'house': 1,
    'car': 1,
    'boat': 0
}
```

VedaBase Output

The VedaBase class stores labels as a NumPy array of boolean values using one-hot encoding. The label classes are stored in a *classes* property that maps to the same position in the label array. The label order is always alphabetical.

```
vedabase.classes = ['boat', 'car', 'house']
vedabase.train.labels[0] = [0, 1, 1]
```

Object Detection

Input

Input features for object detection share the same label requirements as for classification. The objects will be stored as the bounding box of the feature's geometry. The geometry must be of type Polygon but does not need to be a rectangle. Point and Polyline geometries will need to be buffered by an appropriate amount first.

```
"Properties": {
    "label": {
        "car": [
            {"type": "Polygon", "coordinates": [[[...]]]},
            {"type": "Polygon", "coordinates": [[[...]]]}
        ],
        "house": [
            {"type": "Polygon", "coordinates": [[[...]]]}
        ]
    }
```

DataPoint Output

The *label* property of the DataPoint is similar in structure to the Classification case, with keys representing classes. The value of each class is a list of features representing the bounding box of the input geometries. The features use GeoJSON structure, but use the NumPy convention of the origin at top left, and uses units of pixels.

```
datapoint.labels[0] = {
    'house': [f1, f2,..fn],
    'car': [f1, f2,..fn],
    'boat': []
}
```

VedaBase Output

Object detection classes are stored in the *classes* property in the same manner as Classification data. Because object bounding boxes can overlap, the labels are in the form of a list of lists of bounding boxes. The bounding boxes are lists of *[minx, miny, maxx, maxy]* in pixel coordinates with the origin at top left. The position of the classes and labels lists match.

```
vedabase.classes = ['boat', 'car', 'house']
vedabase.train.labels[0] = [
    [], # boat bboxes
    [[0, 0, 1, 1], [1, 2, 4, 5], [3, 3, 7, 9]], # car bboxes
    [[0, 0, 1, 1], [1, 2, 4, 5], [3, 3, 7, 9]] # house bboxes
]
```

Segmentation

Input

The requirements for input data features for segmentation are the same as for object detection. The object must be a Polygon. The geometry will automatically be clipped to fit inside the image tile.

DataPoint Output

The *label* property of the DataPoint has the same structure as for object detection, except the label features represent the input feature's geometry instead of the feature's bounding box. The coordinate values are again in pixels from the top left of the image.

VedaBase Output

Segmentation classes continue in the same structure as the other machine learning types, but include a `None` value in the first position representing the pixels that have no segmentation data. The background pixel value is set to 0, and the segmented pixels store values that match the array index of the class name in the `classes` list. The `labels` for segmentation are 2D NumPy arrays with the pixels representing the class list indices. The 2D array is the same size as the image tile.

```
vedabase.classes = [None, 'boat', 'car', 'house']
vedabase.train.labels[0] = [
    [0, 0, 0, 0, 0, ...],
    [0, 3, 3, 0, 2, ...],
    [0, 3, 3, 0, 0, ...]
]
```

1.5 Model Training

```
from pyveda.pretrainedmodels.keras.classification.resnet50 import ResNet50

model = ResNet50()
ds = pv.open(filename='buildings.vdb')

batch_size = 16
tester = ds.test.batch_generator(batch_size)
trainer = ds.train.batch_generator(batch_size)

# Train the model...
epochs = 1
model.fit_generator(trainer, steps_per_epoch=len(trainer), epochs=epochs, validation_
↪data=tester)
```

1.6 Models

1.6.1 Saving a Model

```
my_model.save('model.h5')
my_model.save_weights('model_weights.h5')
!tar -cvzf model_test.tar.gz model.h5 model_weights.h5
```

1.6.2 Uploading a Model to Veda

```
from pyveda.models import Model
import pyveda as pv

pv.config.set_dev()

vc = pv.from_id('db3c619b-8774-4051-a330-a21771822586')

model_path = './model.tar.gz'

model = Model('XView Burkina Faso Model',
```

(continues on next page)

(continued from previous page)

```
            model_path,
            library="keras",
            bounds=vc.bounds,
            mltype="object_detection",
            shape=vc.imshape,
            dtype=vc.dtype,
            training_set=vc.id)

model.save()
```

1.6.3 Deploying a Model

```
m = Model.from_id('930638be-a247-423a-bdca-987eb19026689')
m.deploy()
```

1.6.4 Searching for Models

```
from pyveda.models import search as model_search

for m in model_search():
    print(m.name, m.id)
    print('\t location:', m.location)
    print('\t lib:', m.library)
    print('\t type:', m.mltype)
    print('\t deployed', m.deployed)
    print('\t public:', m.public, '\n')
```

1.6.5 Downloading a Model

```
m = Model.from_id('930638be-a247-423a-bdca-987eb19026689')
m.download(path='my_local_model.tar.gz')

!ls my_local_model2.tar.gz
```

1.6.6 Predicting with Veda Models

```
image = MLImage(...)
AOI = [...]
model.predict(image, aoi)


model.status
```

1.7 Releases

Releases are compressed files storing metadata files and two directories, one for images and one for labels. `meta.json` contains information about the training set. `bounds.geojson` is a geojson FeatureSet containing the bounds of all the image chips. Images and labels have matching names based on their ID, so a typical file stucture looks like:

```
ts_104-1.0.1.tar.gz

ts_104-1.0.1
|    meta.json
|    bounds.geojson
|
└--  images
|       1.tif
|       2.tif
|       3.tif
└--  labels
        1.json
        2.json
        3.json
```

Images are saved as geotiff.

The meta.json file includes:

```
{'bbox': [<coordinate list>],
'classes': <class data, varies>,
'mlType': <classification type, str>,
'name': <name, str>,
'nclasses': <number of classes, int>,
'public': <bool>,
'source': <image source, str>,
'version': <release version, str>,
's3_location': <S3 location, str>,
'source_url': <url of parent dataset in Veda, str>}
```

The bounds.geojson file contains a geojson feature describing the bounds of the release data.

1.7.1 Releases for Importing

The release format can also be used for imported existing data in Veda using the Bulk Loading API. The *meta.json* and *bounds.geojson* files are not needed for importing. The only requirement is that the data are stored in a tar.gz file, and follow the image and tile structure shown above. Each tile and corresponding json file needs to have the same base name and the extensions *.tif* and *.json*.

Images must in tif format. They do not need to be geotiff files. The image's georeferencing information is taken from the matching json file.

The json file is a single geojson feature consisting of a *Polygon 'describing the image bounds. The label for the image is stored in the feature's 'Properties*. The label format is the same as described in the Label Formats section.

A simplified example of the contents of the json file for classified images is:

```
>>> cat labels/tile_0.json

{
"type": "Feature",
"geometry": {
    "type": "Polygon",
    "coordinates": [[...]]
},
"properties": {
```

(continues on next page)

(continued from previous page)

```
    "label": {
        "clouds": 1,
        "building": 0,
        "car": 1
    }
}
}
```

Images labelled for segmentation or object detection will have a json file like:

```
>>> cat labels/tile_0.json

{
"type": "Feature",
"geometry": {
    "type": "Polygon",
    "coordinates": [[...]]
},
"properties": {
    "label": {
        "building": [
            {"type": "Polygon", "coordinates": [[[]]]},
            {"type": "Polygon", "coordinates": [[[]]]}
        ],
        "car": [
            {"type": "Polygon", "coordinates": [[[]]]}
        ]
    }
}
}
```

Images that do not have georeferencing are also supported. If the geometry field is left empty Veda will treat the image as having a pixel-based coordinate system. Label geometries, if present, are described in units of pixels using the top left of the image as the origin. Data points without spatial information can not be found with spatial searches or filters.

```
>>> cat labels/tile_0.json

{
"type": "Feature",
"geometry": {},
"properties": {
    "label": {
        "clouds": 1,
        "building": 0,
        "car": 1
    }
}
}
```

Once the images and labels are collected in the correct formats and directory structure, they need to be compressed as a *tar.gz* file and placed in a publicly available accessible S3 bucket.

The data can be imported using the `pyveda.main.create_from_tarball()` method, see the *Creating a Collection using the Bulk Import API* section.

1.8 API Reference

1.8.1 Main Interface

1.8.2 Batching

Notebook Documentation

1.9 Accessing Data from Veda

Veda stores machine learning data collections. These can be generated from DigitalGlobe's satellite imagery, or can be imported from other sources. For example, Veda hosts copies of the XView dataset.

So how do you get that data out of Veda and into your models?

1.9.1 Veda's Data Storage

Let's start with looking at how Veda stores data.

> The language used here is in flux - we use collection and dataset to mean the same thing. Similarly, sample and datapoint get interchanged.

1.9.2 Samples

The basic unit of training data is called a sample. It consists of an image, and a label that describes the features in the image. Veda supports three types of labels:

- Classification
- Object Detection
- Segmentation

Classification labels indicate whether a class of features exists in the image. The value is a boolean, where a true value for a class means the class is present.

Object Detection labels describe where features are located, if they are in the image. The value is a bounding box of pixel values, so each label would indicate where in the image the class feature exists.

Segmentation labels are similar to Object Detection labels, but instead of a bounding box, the feature is described by a polygon object that can trace the outline of the feature.

Sample objects also store geospatial information if it's known. Any data generated by Veda will have spatial data, but it's also possible to import ungeoreferenced images.

See the Label Formats section for more information about how labels are used in pyveda.

1.9.3 Collections

All of the samples are stored in groups called collections that keep things sorted and organized. A collection would group data around a theme or area. Collections have unique names to identify them. They also enforces consistency in the data. Some of the requirements of a collection are:

- All images must be the same size in pixels
- All images must have the same number of bands
- All images must share the same data type
- All labels must be the same type - classification, object detection, or segmentation.

Collections also track their contents. A collection knows all of the classes represented in the samples' labels. It also tracks the spatial bounds of all the samples.

1.9.4 Finding Collections with pyveda

The first step to working with data stored in Veda is to find a collection to work with. If you're following a tutorial or a notebook the name of the collection should be given. You can use the name to access the collection. This is covered in the section Accessing Collections.

If you'd like to find a collection to investigate, pyveda has the search method (see pyveda.main.search()). Let's start with the simplest example:

```
[2]: import pyveda as pv

     pv.search()[:5]
```

```
[2]: [VedaCollectionProxy of Austin Segmentation (1245073f-dae9-40ac-ac5c-52ca349ae8dd),
      VedaCollectionProxy of Clouds Classification (bulkload) (4cf03238-e5f6-48ca-a28c-
     ↪fc742345f698),
      VedaCollectionProxy of Kaggle Amazon Small (97c03bd9-d83d-43c5-8dde-ceaf14e071bf),
      VedaCollectionProxy of Austin Obj Detection (e8cda011-e92d-4b99-80f5-28e628588bc8),
      VedaCollectionProxy of Kaggle Amazon Small (92e271dd-cc72-49e3-bf3d-26c0ada58531)]
```

The search method searches Veda for collections and can take key words and filters. It returns a list of VedaCollectionProxy objects, which is a pyveda object that acts as an entry point to working with the collection. We can take a closer look at the collections returned by the search:

```
[3]: results = pv.search()
     print(results[0])
```

```
Austin Segmentation (1245073f-dae9-40ac-ac5c-52ca349ae8dd)
        - Bounds: [-97.767934, 30.265703, -97.733246, 30.271354]
        - Count: 250 samples, 100% cached
        - Chips: 256x256, 3 bands uint8
        - Labels: segmentation type, 1 class
```

These collection objects have an informative `__str__` method that prints collection details in a readable format. You can use the `repr` function to print the shorter representation of just the name and ID.

To search for terms, pass them to the search function:

```
[ ]: pv.search('Austin')
```

Filters can also be added. The queries use ElasticSearch syntax:

```
[ ]: # TODO
     pv.search('Austin')
```

1.10 Accessing Collections

If you have a run a search, you can work directly on the VedaCollectionProxies returned by the search.

```
[7]: results = pv.search()
     collection = results[0] # get the first search result
     # do something with the collection
```

If you have a name or ID of a collection, you can access it with these methods:

```
[9]: collection = pv.from_name('Austin Segmentation')

     collection = pv.from_id('1245073f-dae9-40ac-ac5c-52ca349ae8dd')

     print(collection)
```

```
Austin Segmentation (1245073f-dae9-40ac-ac5c-52ca349ae8dd)
        - Bounds: [-97.767934, 30.265703, -97.733246, 30.271354]
        - Count: 250 samples, 100% cached
        - Chips: 256x256, 3 bands uint8
        - Labels: segmentation type, 1 class
```

1.11 Working with the Samples

Once you have a collection in the form of a VedaCollectionProxy, you can treat it like a list of datapoints. It supports indexing and slicing:

```
[10]: sample = collection[0] # the first sample
      print(sample)
```

```
{'mltype': 'segmentation', 'cached': 'true', 'dataset_id':␣
↪'1245073f-dae9-40ac-ac5c-52ca349ae8dd', 'bounds': [-97.74184363143011, 30.
↪270496899310096, -97.74107094008983, 30.271269590650377], 'dtype': 'uint8',␣
↪'tile_coords': [962, 178], 'sha':␣
↪'36e016e0e9db8ce7d8c8b579ec91d53e6dd5d6dc77506b4d5b483f0e37dce685', 'queue':␣
↪'b98i8k5ca8m6', 'rda_template':␣
↪'87856bc985e698e4eb5d0c7784771c8fb5117ad818c4cfe35e5e8a80e7269eb2', 'id':␣
↪'8728c1fd-533a-4f5c-bc60-4dee90d3ad2f', 'label': {'building': [{'type': 'Polygon',␣
↪'coordinates': [[[0, 5], [4, 6], [0, 18], [0, 5]]]}, {'type': 'Polygon',␣
↪'coordinates': [[[68, 188], [223, 231], [213, 256], [43, 256], [68, 188]]]}]},␣
↪'classes': ['building']}
```

If you look representation of the sample object, you'll see it's metadata and label data. We can also look at the some of the sample's specific properties:

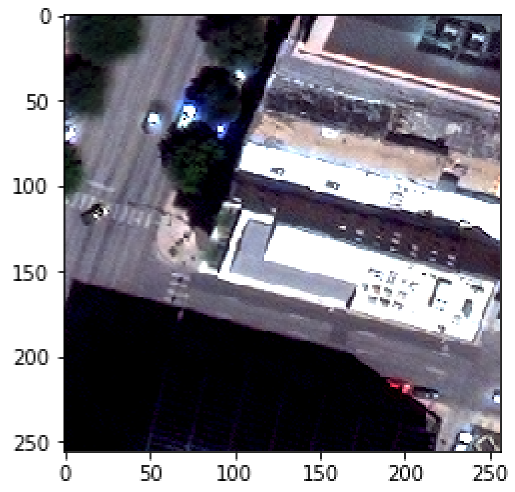
```
[12]: print(sample.classes)
      print(sample.label)
```

```
['building']
{'building': [{'type': 'Polygon', 'coordinates': [[[0, 5], [4, 6], [0, 18], [0, 5]]]},
↪ {'type': 'Polygon', 'coordinates': [[[68, 188], [223, 231], [213, 256], [43, 256],␣
↪[68, 188]]]}]}
```

We can also look at the sample's image data, which is returned as a numpy array:

```
[15]: %matplotlib inline
      import matplotlib.pyplot as plt
      plt.imshow(sample.image)
```

```
[15]: <matplotlib.image.AxesImage at 0x7fb600259748>
```



To browse the samples in a collection, use the `preview()` method:

```
[ ]: # this is an interactive widget
     # so it's not displayed in the docs
     collection.preview()
```

The sample object also has some basic CRUD methods:

```
# a modified label
label = {buildings: [.......
# update the sample with the new label
sample.update(label)

#delete a sample
sample.delete()
```

1.11.1 Using Samples with Models

In most machine learning examples, you have to download all of the sample image files and labels to a local directory. Then you have to split up your data into training and testing groups, and probably also convert the data into array formats that can be read as inputs.

We saw in the previous section how to look at the samples in the collection. So what's the best way to get that sample data to your model? You could certainly loop through all the samples and save their images and labels to files.

Pyveda does provide a way to download the image and label files - see the Releases section. Pyveda makes it easy to get the data in this format - with a single command you get a link to download the compressed file. But pyveda also provides better ways to access the data locally.

To fetch a local cache of the data, use the `store` method. This downloads the sample data to a local file called a VedaBase. Instead of directories of images, Vedabases are single files that hold optimized sample data. All of the sample points have been converted to the correct array formats: the images are numpy arrays, and the labels are one-hot encoded. The store method takes a collections name or ID, and a path to store the file to. It then uses a multithreaded downloader to fetch all the data. You will need to wait for this step to complete before proceeding.

```
[ ]: pv.store('./lasvegas.vdb', dataset_name='Las Vegas Buildings (SpaceNet2)')
```

You can open a saved VedaBase with the `open` function. We recommend using a Vedabase with a context manager:

```
[ ]: with pv.open(filename='./lasvegas.vdb') as data:
         # do something with the data
```

When used as a context manager you still will have to wait for the data to download locally before the rest of the code executes.

1.11.2 Collection Groups

VedaBases internally store separate training, validation, and testing groups, so you don't have to divide the data. The ratio of sample is each group is set by the `partition` property of the collection.

Each group is an accessor on the collection. You can access samples in the group just like a list, through iteration or indexing. The value returned is an `(image, label)` tuple. As mentioned above, the image and label have been coverted to the appropriate arrays already.

```
[ ]: data.train[0] # the first sample in the training group

     for image, label in data.test:
         # do something with each image and label in the testing group
```

The collection also lets you access the images and labels separately:

```
[ ]: for label in data.test.labels:
         # do something with just each label in the testing group
```

This means that a typical training example might look like:

```
from SuperMLDetector import MegaModel

megamodel = MegaModel()

with pv.open(filename='./lasvegas.vdb') as data:
    megamodel.train(data.train)
    megamodel.score(data.test)
```

1.12 Batching and Transforming Data

Pyveda includes tools for batching and transforming data. The three groups have a `batch_generator` method that returns a generator that emits batches of data. Optionally the generator can randomly flip the sample, rescale the values of the image, and move the channel data to the end of the numpy array. The basic use of the generator is to pass the size of the batch.

```
[ ]: # generate batches of 32 samples
     gen = vb.train.batch_generator(32)

     for x, y in gen:
         # x and y will have a length of 32
```

The batcher takes the following optional arguments:

- **shuffle** (Boolean): return the samples in the group in random order. Default is True.
- **channels_last** (Boolean): return image data as Height-Width-Depth. Default is False (RDA returns imagery with the bands last).
- **rescale** (Boolean): rescale image values between 0 and 1. Default is False.
- **flip_horizontal** (Boolean): Randomly flip image and labels horizontally. Default is False.
- **flip_vertical** (Boolean): Randomly flip the image and labels vertically. Default is False.

When using either flip option there is a 50% chance an image will be flipped in a given direction.

To create a batcher that will return 16 samples at a time, rescaled to a range of 0 to 1, and flipped randomly in either direction:

```
[ ]: gen = vb.train.batch_generator(16, rescale=True, flip_horizontal=True, flip_
     ↪vertical=True)
```

1.12.1 Streaming Remote Data

This direct access will probably be removed so that data can be accessed on demand while being locally cached. Because the samples aren't cached, they are loaded every time they are accessed. Using this over many epochs will be slower than saving a local file.

For preliminary testing, or algorithms that only need to work with the datapoints once, it's possible to open the collection directly from Veda, and the samples are loaded on demand. This uses the same open function as loading a local file, and data is accessed in the same manner.

```
[ ]: with pv.open(dataset_name='Las Vegas Buildings (SpaceNet2)') as data:
         # do something with the data or

     with pv.open(dataset_id='84810fa8-87e2-4f22-9386-4406b4c3add0') as data:
         # do something with the data
```

1.13 Creating Releases

A Veda Release is a compressed archive of the images and labels stored as individual files. You can see more about the Release format and structure in the Releases section. To generate a release, run:

```
[ ]: collection.release()
```