satsense Documentation

Netherlands eScience Center

Sep 26, 2019

Contents:

Satsense is a Python library for land use/cover classification using satellite imagery.

CHAPTER 1

Installation

1.1 Installing the dependencies

Satsense has a few dependencies that cannot be installed from PyPI:

- the dependencies of the GDAL Python package
- the dependencies of the netCDF4 Python package

Ubuntu Linux 18.04 and later

To install the above mentioned dependencies, run

```
sudo apt install libgdal-dev libnetcdf-dev
```

this probably also works for other Debian-based Linux distributions.

RPM-based Linux distributions

To install the above mentioned dependencies, run

```
sudo yum install gdal-devel netcdf-devel
```

Conda

Assuming you have conda installed and have downloaded the satsense environment.yml file to the current working directory, you can install all dependencies by running:

```
conda env create --file environment.yml --name satsense
```

or you can install just the minimal dependencies by running

```
conda create --name satsense libgdal libnetcdf nb_conda
```

Make sure to activate the environment after installation:

```
conda activate satsense
```

1.2 Installing Satsense from PyPI

If you did not use conda to install the dependencies, you may still want to create and activate a virtual environment for satsense, e.g. using venv

```
python3 -m venv ~/venv/satsense
source ~/venv/satsense/bin/activate
```

Next, install satsense by running

```
pip install satsense
```

If you are planning on using the *Demonstration Jupyter notebooks*, you can install the required extra dependencies with

```
pip install satsense[notebooks]
```

1.3 Installing Satsense from source for development

Clone the satsense repository, install the dependencies as described above, go to the directory where you have checked out satsense and run

```
pip install -e .[dev]
```

or

```
pip install -e .[dev,notebooks]
```

if you would also like to use the *Demonstration Jupyter notebooks*.

Please read our contribution guidelines before starting development.

1.4 Known installation issues

If you are experiencing 'NetCDF: HDF errors' after installation with pip, this may be resolved by using the following command to install

```
pip install --no-binary netcdf4 satsense
```

see this rasterio issue for more information.

CHAPTER 2

Demonstration Jupyter notebooks

There are a number of demonstration Jupyter Notebooks available to help you get started with satsense. They can be found in the notebooks folder of our github repository.

2.1 Feature Extraction Example

In this example we will extract the Histogram of Gradients (HoG), Normalized Difference Vegetation Index (NDVI) and the Pantex features from a test satelite image.

- The HoG feature captures the distribution of structure orientations.
- The NDVI feature captures the level of vegetation.
- The Pantex feature captures the level of built-up structures.

The image will be split into blocks, in this example 20 by 20 pixels, and each feature is calculated for this block using a certain amount of context information called a window. A feature can be calculated on multiple windows to allow for context at different scales.

2.1.1 In this example

- First we will define the Features we would like to extract and with which window shapes.
- We will then load the image using the `Image` class.
- Then we will split the image into blocks using the `FullGenerator` Class.
- Then we will extract the features using the `extract_features` function.

Live iPython Notebook

If you are reading this example on readthedocs.io a notebook of this example is available in the repository

```
# General imports
import numpy as np

import matplotlib.pyplot as plt
import matplotlib.gridspec as gridspec
%matplotlib inline

# Satsense imports
from satsense import Image
from satsense.generators import FullGenerator
from satsense.extract import extract_features
from satsense.features import NirNDVI, HistogramOfGradients, Pantex
```

2.1.2 Define the features to calculate

First we define a list of windows for each of the features to use.

Hog and Pantex will be calculated on 2 windows of 25x25 pixels and 23x27 pixels. NDVI will be calculated on one window with 37x37 pixels.

These window shapes are chose arbitrarily to show the capabilities of satsense, for your own feature extraction you should think and experiment with these window shapes to give you the best results.

N.B. The NDVI feature here is called NirNDVI because that implementation uses the near-infrared band of the image, there are several other implementations of NDVI available in satsense, see the documentation

```
# Multiple windows
two_windows = [(25, 25), (23, 37)]
# Single window
one_window = [(37, 37),]
features = [
    HistogramOfGradients(two_windows),
    NirNDVI(one_window),
    Pantex(two_windows),
]
```

2.1.3 Load the image

Here we load the image and normalize it to values between 0 and 1. Normalization by default is performed per band using the 2nd and 98th percentiles.

The image class can provide the individual bands, or a number of useful derivatives such as the RGB image or Grayscale, we call these base images. More advanced base images are also available, for instance Canny Edge

```
image = Image('../../test/data/source/section_2_sentinel.tif',
              'quickbird')
image.precompute_normalization()

fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(24, 8), sharey=True)

ax1.axis('off')
ax1.imshow(image['rgb'])
ax1.set_title('RGB image')

ax2.axis('off')
```

(continues on next page)

(continued from previous page)

```
ax2.imshow(image['grayscale'], cmap="gray")
ax2.set_title('Grayscale image')

ax3.axis('off')
ax3.imshow(image['canny_edge'], cmap="gray")
ax3.set_title('Canny Edge Image')

plt.show()
```

2.1.4 Generator

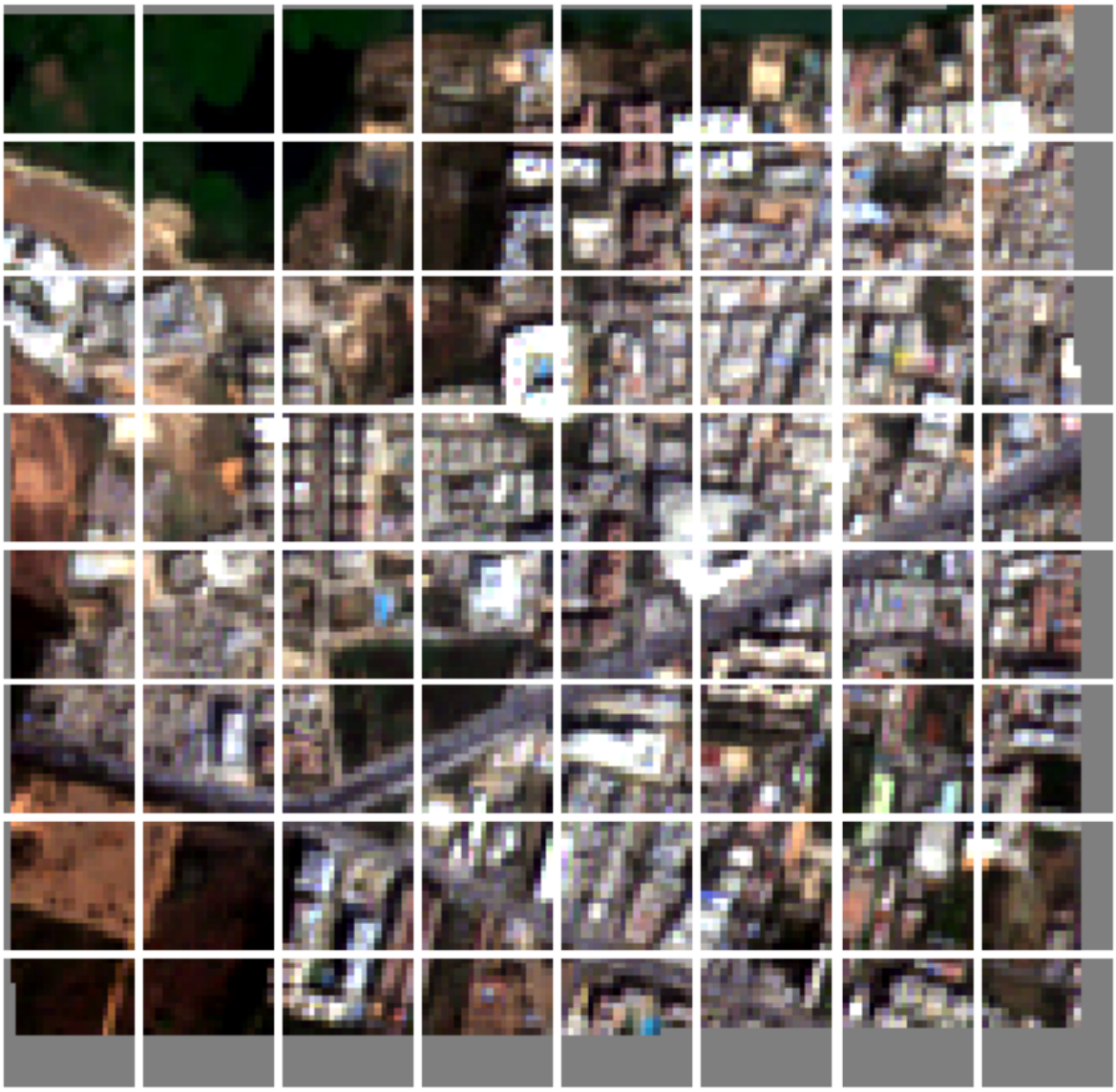
Next we create a FullGenerator which creates patches of the image in steps of 20x20 pixels.

In this cell we also show the images, therefore we load the rgb base image into the generator. This is only needed here so we can show the blocks using matplotlib. In the next section we will be using the `extract_features` function to extract features, which will be loading the correct base images for you based on the features that will be calculated.

The patch sizes are determined by the list of window shapes that you supply the `load_image` function. This is normally also provided by the `extract_features` function.

```
generator = FullGenerator(image, (20, 20))
print("The generator is {} by {}".format(*generator.shape), " blocks")

# Create a gridspec to show the images
gs = gridspec.GridSpec(*generator.shape)
gs.update(wspace=0.05, hspace=0.05)

# Load a baseimage into the generator.
# The window is the same as the block size to show the blocks used
generator.load_image('rgb', ((20, 20),))

fig = plt.figure(figsize=(8, 8))
for i, img in enumerate(generator):
    ax = plt.subplot(gs[i])
    ax.imshow(img.filled(0.5))
    ax.axis('off')
```

```
The generator is 8 by 8  blocks
```



2.1.5 Calculate all the features and append them to a vector

In this cell we use the `extract_features` function from satsense to extract all features.

`extract_features` returns a python generator that we can loop over. Each invocation of this generator returns the feature vector for one feature in the order of the features list. The shape of this vector is (x, y, w, v) where:

- x is the number of blocks of the generator in the x direction
- y is the number of blocks of the generator in the y direction
- w is the number of windows the feature is calculated on
- v is the length of the feature per window

We use a little numpy reshaping to merge these feature vectors into a single feature vector of shape (x, y, n) where n is the total length of all features over all windows. In this example it will be (8, 8, 13) because:

- HoG has 5 numbers per window and 2 windows: 10
- NirNDVI has 1 number per window and 1 window: 1
- Pantex has 1 number per window and2 windows: 2
- Total: 13

```
vector = []
for feature_vector in extract_features(features, generator):
    # The shape returned is (x, y, w, v)
    # Reshape the resulting vector so it is (x, y, w * v)
    # e.g. flattened along the windows and features
    data = feature_vector.vector.reshape(
                *feature_vector.vector.shape[0:2], -1)
    vector.append(data)
# dstack reshapes the vector into and (x, y, n)
# where n is the total length of all features
featureset = np.dstack(vector)

print("Feature set has shape:", featureset.shape)
```

```
Feature set has shape: (8, 8, 13)
```

2.1.6 Showing the resulting features

Below we show the results for the calculated features.

In the result images you can see the edges of the feature vector have been masked as the windows at the edge of the original image contain masked values. Furthermore, please keep in mind that the value for the feature in each block depends on an area around the block.

HoG

Here is the result of the HoG feature, we display the first value for each window.

Histogram of Gradients is a feature that first calculates a histogram of the gradient orientations in the window. Using this histogram 5 values are calculated. This first value is the 1st heaved central shift moment. Heaved central shift moments are a measure of spikiness of a histogram.

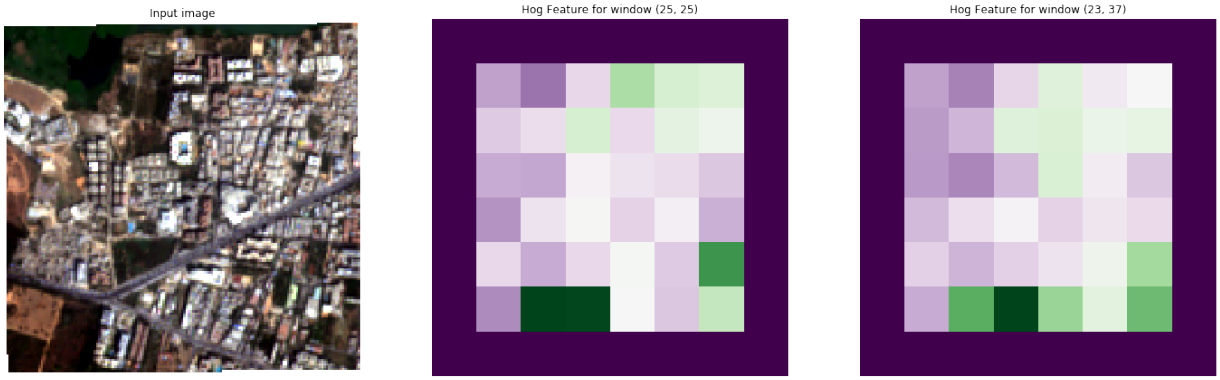
The other values are: the 2nd heaved central shift moment, the orientation of the highest and second highest peaks and the sine of the absolute difference between the highest and second highest peak (this is 1 for right angles).

```
fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(24, 8))

ax1.axis('off')
ax1.imshow(image['rgb'])
ax1.set_title('Input image')

ax2.axis('off')
ax2.imshow(featureset[:, :, 0], cmap="PRGn")
ax2.set_title('Hog Feature for window {}'.format(two_windows[0]))

ax3.axis('off')
ax3.imshow(featureset[:, :, 5], cmap="PRGn")
ax3.set_title('Hog Feature for window {}'.format(two_windows[1]))
```

(continues on next page)

(continued from previous page)

```
plt.show()
```



Normalized Difference Vegetation Index

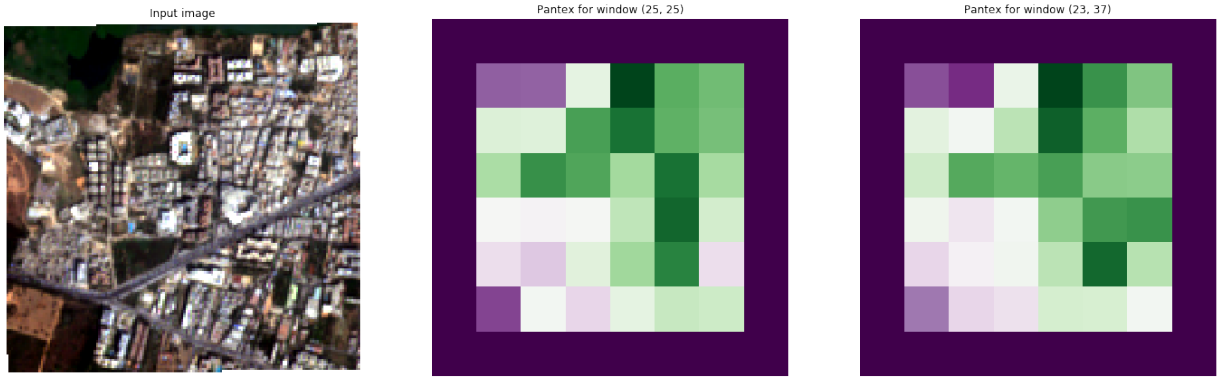
Here we show the result for the NDVI feature The NDVI feature captures the level of vegetation, purple means very little vegetation, green means a lot of vegetation.

```
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(16, 8))

ax1.axis('off')
ax1.imshow(image['rgb'])
ax1.set_title('Input image')

ax2.axis('off')
ax2.imshow(featureset[:, :, 10], cmap="PRGn")
ax2.set_title('NirNDVI for window {}'.format(one_window[0]))

plt.show()
```

Pantex

Here we show the results for the Pantex feature. The Pantex feature captures the level of built-up structures, purple means very little built-up while green means very built-up.

```
fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(24, 8))

ax1.axis('off')
ax1.imshow(image['rgb'])
ax1.set_title('Input image')

ax2.axis('off')
ax2.imshow(featureset[:, :, 11], cmap="PRGn")
ax2.set_title('Pantex for window {}'.format(two_windows[0]))

ax3.axis('off')
ax3.imshow(featureset[:, :, 12], cmap="PRGn")
ax3.set_title('Pantex for window {}'.format(two_windows[1]))

plt.show()
```



CHAPTER 3

Python API Reference

3.1 satsense package

3.1.1 Submodules

satsense.image

Methods for loading images.

class satsense.image.**Image**(*filename*, *satellite*, *band='rgb'*, *normalization_parameters=None*, *block=None*, *cached=None*)

Bases: object

Image class that provides a unified interface to satellite images.

Under the hood rasterio is used, so any format supported by rasterio can be used.

Parameters

- **filename** (*str*) – The name of the image
- **satellite** (*str*) – The name of the satelite (i.e. worldview3, quickbird etc.)
- **band** (*str*) – The band for the grayscale image, or 'rgb'. The default is 'rgb'
- **normalization_parameters** (*dict or boolean, optional*) – if False no normalization is done. if None the default normalization will be applied (cumulative with 2, 98 percentiles)

  f a Dictionary that describes the normalization parameters The following keys can be supplied:

  – **technique: string** The technique to use, can be 'cumulative' (default), 'meanstd' or 'minmax'

  – **percentiles: list[int]** The percentiles to use (exactly 2) if technique is cumulative, default is [2, 98]

– **numstds: float** Number of standard deviations to use if technique is meanstd

- **block** (*tuple or rasterio.windows.Window, optional*) – The part of the image read defined in a rasterio compatible way, e.g. two tuples or a rasterio.windows.Window object
- **cached** (*array-like or boolean, optional*) – If True bands and base images are cached in memory if an array a band or base image is cached if its name is in the array

Examples

Load an image and inspect the shape and bands

from satsense import Image >>> image = Image('test/data/source/section_2_sentinel.tif', 'quickbird') >>> image.shape (152, 155)

```
>>> image.bands
{'blue': 0, 'green': 1, 'red': 2, 'nir-1': 3}
```

```
>>> image.crs
CRS({'init': 'epsg:32643'})
```

See also:

satsense.bands

itypes = {'canny_edge': <function get_canny_edge_image>, 'gray_ubyte': <function

classmethod register(*itype*, *function*)

Register a new image type.

Parameters

- **itype** (*str*) – (internal) name of the image type
- **function** – Function definition that should take a single Image parameter and return a numpy.ndarray or numpy.ma.masked_array

See also:

ufunc get_gray_ubyte_image

ufunc get_grayscale_image

ufunc get_rgb_image

copy_block(*block*)

Create a subset of Image.

Parameters **block** (*tuple or rasterio.windows.Window*) – The part of the image to read defined in a rasterio compatible way, e.g. two tuples or a rasterio.windows.Window object

Returns subsetted image

Return type *image.Image*

__getitem__(*itype*)

Get image of a type registered using the *register* method. The following itypes are available to facilitate creating new features: - 'rgb' - 'grayscale' - 'gray_ubyte'

Parameters **itype** (*str*) – The name of the image type to retrieve

Returns **out** – The image of the supplied type

Return type numpy.ndarray or numpy.ma.masked_array

Examples

Get the rgb image >>> image['rgb'].shape (152, 155, 3) >>> image['gray_ubyte'].dtype dtype('uint8')

precompute_normalization(**bands*)
Precompute the normalization of the image

Normalization is done using the normalization_parameters supplied during class instantiation. Normalization parameters are computed automatically for all bands when required, but doing it explicitly can save some time, e.g. if there are more bands in the image than needed.

Parameters ***bands** (*list[str] or None*) – The list of bands to normalize, if None all bands will be normalized

Raises ValueError: – When trying to compute the normalization on a partial image, as created by using the *copy_block* method.

See also:

Image()

func _normalize Get normalization limits for band(s).

shape
Provide shape attribute.

crs
Provide crs attribute.

transform
Provide transform attribute.

scaled_transform(*step_size*)
Perform a scaled transformation.

Returns **out** – An affine transformation scaled by the step size

Return type affine.Affine

satsense.image.**get_rgb_image**(*image: satsense.image.Image*)
Convert the image to rgb format.

Parameters **image** (image.Image) – The image to calculate the rgb image from

Returns The image converted to rgb

Return type numpy.ndarray

satsense.image.**get_grayscale_image**(*image: satsense.image.Image*)
Convert the image to grayscale.

Parameters **image** (image.Image) – The image to calculate the grayscale image from

Returns The image converted to grayscale in 0 - 1 range

Return type numpy.ndarray

See also:

skimage.color.rgb2gray() Used to convert rgb image to grayscale

satsense.image.**get_gray_ubyte_image**(*image: satsense.image.Image*)
Convert image in 0 - 1 scale format to ubyte 0 - 255 format.

Parameters **image** (image.Image) – The image to calculate the grayscale image from

Returns The image converted to grayscale

Return type numpy.ndarray

See also:

skimage.img_as_ubyte() Used to convert the image to ubyte

class satsense.image.**FeatureVector**(*feature*, *vector*, *crs=None*, *transform=None*)
Bases: object

Class to store a feature vector in.

Parameters

- **feature** (*satsense.feature.Feature*) – The feature to store
- **vector** (*array-like*) – The data of the computed feature
- **crs** – The coordinate reference system for the data
- **transform** (*Affine*) – The affine transformation for the data

get_filename(*window*, *prefix=''*, *extension='nc'*)
Construct filename from input parameters.

Parameters

- **window** (*tuple*) – The shape of the window used to calculate the feature
- **prefix** (*str*) – Prefix for the filename
- **extension** (*str*) – Filename extension

Returns

Return type str

save(*filename_prefix=''*, *extension='nc'*)
Save feature vectors to files.

Parameters

- **filename_prefix** (*str*) – Prefix for the filename
- **extension** (*str*) – Filename extension
- **Returns** – 1-D array-like (str)

classmethod from_file(*feature*, *filename_prefix*)
Restore saved features.

Parameters

- **feature** (Feature) – The feature to restore from a file

- **filename_prefix** (*str*) – The directory and other prefixes to find the feature file at

Returns The feature loaded into a FeatureVector object

Return type *satsense.image.FeatureVector*

satsense.bands

Mappings for satelite image bands

0-index based for python, when using gdal add 1

Notes

Known satellites are

- worldview2 - 7 bands
- worldview3 - 7 bands
- quickbird - 4 bands
- pleiades - 5 bands
- rgb - 3 bands
- monochrome - 1 band

Example

if you need direct access to the bands of the image you can find them using this package:

```
>>> from satsense.bands import BANDS
>>> print(BANDS['worldview3'])
{'coastal': 0, 'blue': 1, 'green': 2, 'yellow': 3,
 'red': 4, 'red-edge': 5, 'nir-1': 6, 'nir-2': 7}
```

satsense.features

class satsense.features.**Feature**(*window_shapes*, ***kwargs*)

Bases: abc.ABC

Feature superclass.

Parameters

- **window_shapes** (*list[tuple]*) – List of tuples of window shapes to calculate the feature on
- ****kwargs** (*dict*) – Keyword arguments for the feature

base_image

base_image = None

static compute(*window*, ***kwargs*)
Compute the feature on the window This function needs to be set by the implementation subclass `compute = staticmethod(my_feature_calculation)` :param window: The shape of the window :type window: tuple[int] :param **kwargs: The keyword arguments for the compustation :type **kwargs: dict

indices
The indices for this feature in a feature set .. seealso:: *FeatureSet*

size = None

windows
Returns the windows this feature uses for calculation :returns: :rtype: tuple[tuple[int]]

class satsense.features.**FeatureSet**
Bases: object

FeatureSet Class

The FeatureSet class can be used to bundle a number of features together. this class then calculates the indices for each feature within a vector of all features stacked into a single 3 dimensional matrix.

items

add(*feature*, *name=None*)

Parameters

- **feature** (Feature) – The feature to add to the set
- **name** (*str*) – The name to give the feature in the set. If none the features class name and length is used
- **Returns** –

  name [str] The name of the added feature

  feature [Feature] The added feature

remove(*name*)
Remove the feature from the set :param name: The name of the feature to remove :type name: str

Returns Wether the feature was succesfully removed

Return type bool

index_size
The size of the index

base_images
list[str] List of base images that was used to calculate these features

class satsense.features.**HistogramOfGradients**(*window_shapes*, ***kwargs*)
Bases: *satsense.features.Feature*

Histogram of Oriented Gradient Feature Calculator

The compute method calculates the feature on a particular window this returns the 1st and 2nd heaved central shift moments, the orientation of the first and second highest peaks and the absolute sine difference between the orientations of the highest peaks

Parameters

- **window_shapes** (*list[tuple]*) – The window shapes to calculate the feature on.
- **bins** (*int*) – The number of bins to use. The default is 50

- **kernel** (*typing.Callable*) – The function to use for smoothing. The default is `scipy.stats.norm().pdf`.
- **bandwidth** (*float*) – The bandwidth for the smoothing. The default is 0.7

size
The size of the feature vector returned by this feature

Type int

base_image
The name of the base image used to calculate the feature

Type str

Example

Calculating the HistogramOfGradients on an image using a generator:

```
from satsense import Image
from satsense.generators import FullGenerator
from satsense.extract import extract_feature
from satsense.features import HistogramOfGradients

windows = ((50, 50), )
hog = HistogramOfGradients(windows)

image = Image('test/data/source/section_2_sentinel.tif',
              'quickbird')
image.precompute_normalization()
generator = FullGenerator(image, (10, 10))

feature_vector = extract_feature(hog, generator)
```

base_image = 'grayscale'

size = 5

static compute(*window*, *bins=50*, *kernel=None*, *bandwidth=0.7*)
Calculate the hog features on the window.

Features are the 1st and 2nd order heaved central shift moments, the angle of the two highest peaks in the histogram, the absolute sine difference between the two highest peaks.

Parameters

- **window** (*numpy.ndarray*) – The window to calculate the features on (grayscale).
- **bands** (*dict*) – A discription of the bands used in the window.
- **bins** (*int*) – The number of bins to use.
- **kernel** (*typing.Callable*) – The function to use for smoothing. The default is `scipy.stats.norm().pdf`.
- **bandwidth** (*float*) – The bandwidth for the smoothing.

Returns The 5 HoG feature values.

Return type numpy.ndarray

class satsense.features.**Pantex**(*window_shapes*, ***kwargs*)
Bases: *satsense.features.Feature*

Pantext Feature Calculator

The compute method calculates the feature on a particular window this returns the minimum of the grey level co-occurence matrix contrast property

Parameters

- **window_shapes** (*list*) – The window shapes to calculate the feature on.
- **maximum** (*int*) – The maximum value in the image.

size
The size of the feature vector returned by this feature

Type int

base_image
The name of the base image used to calculate the feature

Type str

Example

Calculating the Pantex on an image using a generator:

```
from satsense import Image
from satsense.generators import FullGenerator
from satsense.extract import extract_feature
from satsense.features import Pantex

windows = ((50, 50), )
pantex = Pantex(windows)

image = Image('test/data/source/section_2_sentinel.tif',
              'quickbird')
image.precompute_normalization()
generator = FullGenerator(image, (10, 10))

feature_vector = extract_feature(pantex, generator)
```

base_image = 'gray_ubyte'

size = 1

static compute(*window*, *maximum=255*)
Calculate the pantex feature on the given grayscale window.

Parameters

- **window** (*numpy.ndarray*) – A window on an image.
- **maximum** (*int*) – The maximum value in the image.

Returns Pantex feature value.

Return type float

class satsense.features.**NDXI**(*window_shapes*, ***kwargs*)
Bases: *satsense.features.Feature*

The parent class of the family of NDXI features.

Parameters **window_shapes** (*list*) – The window shapes to calculate the feature on.

compute
Used by autodoc_mock_imports.

size = 1

class satsense.features.**NirNDVI**(*window_shapes*, ***kwargs*)
Bases: *satsense.features.NDXI*

The infrared-green normalized difference vegetation index.

For more information see[2].

Parameters **window_shapes** (*list*) – The window shapes to calculate the feature on.

Notes

base_image = 'nir_ndvi'

class satsense.features.**RgNDVI**(*window_shapes*, ***kwargs*)
Bases: *satsense.features.NDXI*

The red-green normalized difference vegetation index.

For more information see[3].

Parameters **window_shapes** (*list*) – The window shapes to calculate the feature on.

Notes

base_image = 'rg_ndvi'

class satsense.features.**RbNDVI**(*window_shapes*, ***kwargs*)
Bases: *satsense.features.NDXI*

The red-blue normalized difference vegetation index.

For more information see[4].

Parameters **window_shapes** (*list*) – The window shapes to calculate the feature on.

Notes

base_image = 'rb_ndvi'

class satsense.features.**NDSI**(*window_shapes*, ***kwargs*)
Bases: *satsense.features.NDXI*

The snow cover index.

Parameters **window_shapes** (*list*) – The window shapes to calculate the feature on.

base_image = 'ndsi'

[2] https://en.wikipedia.org/wiki/Normalized_difference_vegetation_index
[3] Motohka, T., Nasahara, K.N., Oguma, H. and Tsuchida, S., 2010. "Applicability of green-red vegetation index for remote sensing of vegetation phenology". Remote Sensing, 2(10), pp. 2369-2387.
[4] Tanaka, S., Goto, S., Maki, M., Akiyama, T., Muramoto, Y. and Yoshida, K., 2007. "Estimation of leaf chlorophyll concentration in winter wheat [Triticum aestivum] before maturing stage by a newly developed vegetation index-RBNDVI". Journal of the Japanese Agricultural Systems Society (Japan).

class satsense.features.**NDWI**(*window_shapes*, ***kwargs*)
Bases: *satsense.features.NDXI*

The water cover index.

Parameters **window_shapes** (*list*) – The window shapes to calculate the feature on.

base_image = 'ndwi'

class satsense.features.**WVSI**(*window_shapes*, ***kwargs*)
Bases: *satsense.features.NDXI*

The soil cover index.

Parameters **window_shapes** (*list*) – The window shapes to calculate the feature on.

base_image = 'wvsi'

class satsense.features.**Lacunarity**(*windows=((25, 25),)*, *box_sizes=(10, 20, 30)*)
Bases: *satsense.features.Feature*

Calculate the lacunarity value over an image.

Lacunarity is a measure of 'gappiness' of the image. The calculation is performed following these papers:

Kit, Oleksandr, and Matthias Luedeke. "Automated detection of slum area change in Hyderabad, India using multitemporal satellite imagery." ISPRS journal of photogrammetry and remote sensing 83 (2013): 130-137.

Kit, Oleksandr, Matthias Luedeke, and Diana Reckien. "Texture-based identification of urban slums in Hyderabad, India using remote sensing data." Applied Geography 32.2 (2012): 660-667.

base_image = 'canny_edge'

static compute(*canny_edge_image*, *box_sizes*)
Calculate the lacunarities for all box_sizes.

class satsense.features.**Sift**(*windows*, *kmeans: <sphinx.ext.autodoc.importer._MockObject object at 0x7f0afa86e4a8>*, *normalized=True*)
Bases: *satsense.features.Feature*

Scale-Invariant Feature Transform calculator

First create a codebook of SIFT features from the suplied images using *from_images*. Then we can compute the histogram of codewords for a given window.

See the opencv SIFT intro for more information

Parameters

- **window_shapes** (*list*) – The window shapes to calculate the feature on.
- **kmeans** (*sklearn.cluster.MiniBatchKMeans*) – The trained KMeans clustering from opencv
- **normalized** (*bool*) – If True normalize the feature by the total number of clusters

Example

Calculating the Sift feature on an image using a generator:

```
from satsense import Image
from satsense.generators import FullGenerator
from satsense.extract import extract_feature
from satsense.features import Sift
```

(continues on next page)

(continued from previous page)

```
windows = ((50, 50), )

image = Image('test/data/source/section_2_sentinel.tif', 'quickbird')
image.precompute_normalization()

sift = Sift.from_images(windows, [image])

generator = FullGenerator(image, (10, 10))

feature_vector = extract_feature(sift, generator)
print(feature_vector.shape)
```

base_image = 'gray_ubyte'

static compute(*window_gray_ubyte*, *kmeans: <sphinx.ext.autodoc.importer._MockObject object at 0x7f0afa86e4a8>*, *normalized=True*)

Calculate the Scale-Invariant Feature Transform feature

The opencv SIFT features are first calculated on the window the codewords of these features are then extracted using the previously computed cluster centers. Finally a histogram of these codewords is returned

Parameters

- **window_gray_ubyte** (*ndarray*) – The window to calculate the feature on
- **kmeans** (*sklearn.cluster.MiniBatchKMeans*) – The trained KMeans clustering from opencv, see *from_images*
- **normalized** (*bool*) – If True normalize the feature by the total number of clusters

Returns The histogram of sift feature codewords

Return type ndarray

classmethod from_images(*windows*, *images: Iterator[satsense.image.Image]*, *n_clusters=32*, *max_samples=100000*, *sample_window=(8192, 8192)*, *normalized=True*)

Create a codebook of SIFT features from the suplied images.

Using the images *max_samples* SIFT features are extracted evenly from all images. These features are then clustered into *n_clusters* clusters. This codebook can then be used to calculate a histogram of this codebook.

Parameters

- **windows** (*list[tuple]*) – The window shapes to calculate the feature on.
- **images** (*Iterator[satsense.Image]*) – Iterable for the images to calculate the codebook no
- **n_cluster** (*int*) – The number of clusters to create for the codebook
- **max_samples** (*int*) – The maximum number of samples to use for creating the codebook
- **normalized** (*bool*) – Wether or not to normalize the resulting feature with regards to the number of clusters

class satsense.features.**Texton**(*windows*, *kmeans: <sphinx.ext.autodoc.importer._MockObject object at 0x7f0afa8741d0>*, *normalized=True*)

Bases: *satsense.features.Feature*

Texton Feature Transform calculator

First create a codebook of Texton features from the suplied images using *from_images*. Then we can compute the histogram of codewords for a given window.

For more information see[1].

Parameters

- **window_shapes** (*list*) – The window shapes to calculate the feature on.
- **kmeans** (*sklearn.cluster.MiniBatchKMeans*) – The trained KMeans clustering from opencv
- **normalized** (*bool*) – If True normalize the feature by the total number of clusters

Example

Calculating the Texton feature on an image using a generator:

```
from satsense import Image
from satsense.generators import FullGenerator
from satsense.extract import extract_feature
from satsense.features import Texton

windows = ((50, 50), )

image = Image('test/data/source/section_2_sentinel.tif', 'quickbird')
image.precompute_normalization()

texton = Texton.from_images(windows, [image])

generator = FullGenerator(image, (10, 10))

feature_vector = extract_feature(texton, generator)
print(feature_vector.shape)
```

Notes

base_image = 'texton_descriptors'

static compute(*descriptors*, *kmeans: <sphinx.ext.autodoc.importer._MockObject object at 0x7f0afa8741d0>*, *normalized=True*)

Calculate the texton feature on the given window.

classmethod from_images(*windows*, *images: Iterator[satsense.image.Image]*, *n_clusters=32*, *max_samples=100000*, *sample_window=(8192, 8192)*, *normalized=True*)

Create a codebook of Texton features from the suplied images.

Using the images *max_samples* Texton features are extracted evenly from all images. These features are then clustered into *n_clusters* clusters. This codebook can then be used to calculate a histogram of this codebook.

Parameters

- **windows** (*list[tuple]*) – The window shapes to calculate the feature on.

[1] Arbelaez, Pablo, et al., "Contour detection and hierarchical image segmentation," IEEE transactions on pattern analysis and machine intelligence (2011), vol. 33 no. 5, pp. 898-916.

- **images** (*Iterator[satsense.Image]*) – Iterable for the images to calculate the codebook no
- **n_cluster** (*int*) – The number of clusters to create for the codebook
- **max_samples** (*int*) – The maximum number of samples to use for creating the codebook
- **normalized** (*bool*) – Wether or not to normalize the resulting feature with regards to the number of clusters

satsense.generators

Module providing a generator to iterate over the image.

class satsense.generators.**BalancedGenerator**(*image: satsense.image.Image*, *masks*, *p=None*, *samples=None*, *offset=(0, 0)*, *shape=None*)

Bases: object

Balanced window generator.

Parameters

- **image** (Image) – Satellite image
- **masks** (*1-D array-like*) – List of masks, one for each class, to use for generating patches A mask should have a positive value for the array positions that are included in the class
- **p** (*1-D array-like, optional*) – The probabilities associated with each entry in masks. If not given the sample assumes a uniform distribution over all entries in a.
- **samples** (*int, optional*) – The maximum number of samples to generate, otherwise infinite

Examples

Using BalancedGenerator

```
>>> from satsense.generators import BalancedGenerator
>>> BalancedGenerator(image,
                      [class1_mask, class2_mask, class3_mask],
                      [0.33, 0.33, 0.33])
```

class satsense.generators.**FullGenerator**(*image: satsense.image.Image*, *step_size: tuple*, *offset=(0, 0)*, *shape=None*)

Bases: object

Window generator that covers the full image.

Parameters

- **image** (Image) – Satellite image
- **step_size** (*tuple(int, int)*) – Size of the steps to use to iterate over the image (in pixels)

- **offset** (*tuple(int, int)*) – Offset from the (0, 0) point (in number of steps).
- **shape** (*tuple(int, int)*) – Shape of the generator (in number of steps)

load_image (*itype*, *windows*)
Load image with sufficient additional data to cover windows.

Parameters

- **itype** (*str*) – Image type
- **windows** (*list[tuple]*) – The list of tuples of window shapes that will be used with this generator

split (*n_chunks*)
Split processing into chunks.

Parameters **n_chunks** (*int*) – Number of chunks to split the image into

satsense.extract

Module for computing features.

satsense.extract.**extract_features** (*features: Iterator[satsense.features.Feature], generator: satsense.generators.FullGenerator, n_jobs: int = -1*)
Compute features.

Parameters

- **features** – Iterable of features.
- **generator** – Generator providing the required windows on the image.
- **n_jobs** – The maximum number of processes to use. The default is to use the value returned by `os.cpu_count()`.

Yields *satsense.FeatureVector* – The requested feature vectors.

Examples

Extracting features from an image:

```
import numpy as np
from satsense import Image
from satsense.generators import FullGenerator
from satsense.extract import extract_features
from satsense.features import NirNDVI, HistogramOfGradients, Pantex

# Define the features to calculate
features = [
    HistogramOfGradients(((50, 50), (100, 100))),
    NirNDVI(((50, 50),)),
    Pantex(((50, 50), (100, 100))),
]

# Load the image into a generator
# This generator splits the image into chunks of 10x10 pixels
image = Image('test/data/source/section_2_sentinel.tif', 'quickbird')
```

(continues on next page)

(continued from previous page)

```
image.precompute_normalization()
generator = FullGenerator(image, (10, 10))

# Calculate all the features and append them to a list
vector = []
for feature_vector in extract_features(features, generator):
    # The shape returned is (x, y, w, v)
    # where x is the number of chunks in the x direction
    #       y is the number of chunks in the y direction
    #       w is the number of windows the feature uses
    #       v is the length of the feature per window
    # Reshape the resulting vector so it is (x, y, w * v)
    # e.g. flattened along the windows and features
    data = feature_vector.vector.reshape(
                *feature_vector.vector.shape[0:2], -1)
    vector.append(data)
# dstack reshapes the vector into and (x, y, n)
# where n is the total length of all features
featureset = np.dstack(vector)
```

satsense.extract.**extract_feature**(*feature*, *generator*)
Compute a single feature vector.

Parameters

- **feature** (Feature) – The feature to calculate
- **generator** – Generator providing the required windows on the image.

satsense.performance

The initialization module for the performance measures package.

satsense.performance.**jaccard_index_binary_masks**(*truth_mask*, *predicted_mask*)

satsense.performance.**jaccard_index_multipolygons**(*truth_multi*, *predicted_multi*)

satsense.util

satsense.util.**save_mask2file**(*mask*, *filename*, *crs=None*, *transform=None*)
Save a mask to filename.

satsense.util.**load_mask_from_file**(*filename*)
Load a binary mask from filename into a numpy array.

@returns mask The mask image loaded as a numpy array

satsense.util.**show_multipolygon**(*multipolygon*, *axis*, *show_coords*, *extent*, *color*, *alpha*, *title*)
Visualize multipolygon in plot.

satsense.util.**load_shapefile2multipolygon**(*filename*)
Load a shapefile as a MultiPolygon.

satsense.util.**save_multipolygon2shapefile**(*multipolygon*, *shapefilename*)
Save a MultiPolygon to a shapefile.

satsense.util.**multipolygon2mask**(*multipolygon*, *out_shape*, *transform=<sphinx.ext.autodoc.importer._MockObject object>*, *all_touched=False*)

Convert from shapely multipolygon to binary mask.

satsense.util.**mask2multipolygon**(*mask_data*, *mask*, *transform=<sphinx.ext.autodoc.importer._MockObject object>*, *connectivity=4*)

Convert from binary mask to shapely multipolygon.

3.1.2 Module contents

Satsense package.

class satsense.**Image**(*filename*, *satellite*, *band='rgb'*, *normalization_parameters=None*, *block=None*, *cached=None*)

Bases: object

Image class that provides a unified interface to satellite images.

Under the hood rasterio is used, so any format supported by rasterio can be used.

Parameters

- **filename** (*str*) – The name of the image
- **satellite** (*str*) – The name of the satelite (i.e. worldview3, quickbird etc.)
- **band** (*str*) – The band for the grayscale image, or 'rgb'. The default is 'rgb'
- **normalization_parameters** (*dict or boolean, optional*) – if False no normalization is done. if None the default normalization will be applied (cumulative with 2, 98 percentiles)

  f a Dictionary that describes the normalization parameters The following keys can be supplied:

  - **technique: string** The technique to use, can be 'cumulative' (default), 'meanstd' or 'minmax'
  - **percentiles: list[int]** The percentiles to use (exactly 2) if technique is cumulative, default is [2, 98]
  - **numstds: float** Number of standard deviations to use if technique is meanstd
- **block** (*tuple or rasterio.windows.Window, optional*) – The part of the image read defined in a rasterio compatible way, e.g. two tuples or a rasterio.windows.Window object
- **cached** (*array-like or boolean, optional*) – If True bands and base images are cached in memory if an array a band or base image is cached if its name is in the array

Examples

Load an image and inspect the shape and bands

from satsense import Image >>> image = Image('test/data/source/section_2_sentinel.tif', 'quickbird') >>> image.shape (152, 155)

```
>>> image.bands
{'blue': 0, 'green': 1, 'red': 2, 'nir-1': 3}
```

```
>>> image.crs
CRS({'init': 'epsg:32643'})
```

See also:

satsense.bands

itypes = {'canny_edge': <function get_canny_edge_image>, 'gray_ubyte': <function get_

classmethod register(*itype*, *function*)
Register a new image type.

Parameters

- **itype** (*str*) – (internal) name of the image type
- **function** – Function definition that should take a single Image parameter and return a numpy.ndarray or numpy.ma.masked_array

See also:

ufunc get_gray_ubyte_image

ufunc get_grayscale_image

ufunc get_rgb_image

copy_block(*block*)
Create a subset of Image.

Parameters **block** (*tuple or rasterio.windows.Window*) – The part of the image to read defined in a rasterio compatible way, e.g. two tuples or a rasterio.windows.Window object

Returns subsetted image

Return type *image.Image*

precompute_normalization(**bands*)
Precompute the normalization of the image

Normalization is done using the normalization_parameters supplied during class instantiation. Normalization parameters are computed automatically for all bands when required, but doing it explicitly can save some time, e.g. if there are more bands in the image than needed.

Parameters ***bands** (*list[str] or None*) – The list of bands to normalize, if None all bands will be normalized

Raises ValueError: – When trying to compute the normalization on a partial image, as created by using the *copy_block* method.

See also:

Image()

func _normalize Get normalization limits for band(s).

shape
Provide shape attribute.

crs
Provide crs attribute.

transform
Provide transform attribute.

scaled_transform(*step_size*)
Perform a scaled transformation.

Returns **out** – An affine transformation scaled by the step size

Return type affine.Affine

satsense.**extract_features**(*features: Iterator[satsense.features.Feature], generator: satsense.generators.FullGenerator, n_jobs: int = -1*)
Compute features.

Parameters

- **features** – Iterable of features.
- **generator** – Generator providing the required windows on the image.
- **n_jobs** – The maximum number of processes to use. The default is to use the value returned by `os.cpu_count()`.

Yields *satsense.FeatureVector* – The requested feature vectors.

Examples

Extracting features from an image:

```
import numpy as np
from satsense import Image
from satsense.generators import FullGenerator
from satsense.extract import extract_features
from satsense.features import NirNDVI, HistogramOfGradients, Pantex

# Define the features to calculate
features = [
    HistogramOfGradients(((50, 50), (100, 100))),
    NirNDVI(((50, 50),)),
    Pantex(((50, 50), (100, 100))),
]

# Load the image into a generator
# This generator splits the image into chunks of 10x10 pixels
image = Image('test/data/source/section_2_sentinel.tif', 'quickbird')
image.precompute_normalization()
generator = FullGenerator(image, (10, 10))

# Calculate all the features and append them to a list
vector = []
for feature_vector in extract_features(features, generator):
    # The shape returned is (x, y, w, v)
    # where x is the number of chunks in the x direction
    #       y is the number of chunks in the y direction
    #       w is the number of windows the feature uses
    #       v is the length of the feature per window
    # Reshape the resulting vector so it is (x, y, w * v)
    # e.g. flattened along the windows and features
    data = feature_vector.vector.reshape(
                *feature_vector.vector.shape[0:2], -1)
```

(continues on next page)

(continued from previous page)

```
    vector.append(data)
# dstack reshapes the vector into and (x, y, n)
# where n is the total length of all features
featureset = np.dstack(vector)
```

class satsense.**FeatureVector**(*feature*, *vector*, *crs=None*, *transform=None*)

Bases: object

Class to store a feature vector in.

Parameters

- **feature** (*satsense.feature.Feature*) – The feature to store
- **vector** (*array-like*) – The data of the computed feature
- **crs** – The coordinate reference system for the data
- **transform** (*Affine*) – The affine transformation for the data

get_filename(*window*, *prefix=''*, *extension='nc'*)

Construct filename from input parameters.

Parameters

- **window** (*tuple*) – The shape of the window used to calculate the feature
- **prefix** (*str*) – Prefix for the filename
- **extension** (*str*) – Filename extension

Returns

Return type str

save(*filename_prefix=''*, *extension='nc'*)

Save feature vectors to files.

Parameters

- **filename_prefix** (*str*) – Prefix for the filename
- **extension** (*str*) – Filename extension
- **Returns** – 1-D array-like (str)

classmethod from_file(*feature*, *filename_prefix*)

Restore saved features.

Parameters

- **feature** (Feature) – The feature to restore from a file
- **filename_prefix** (*str*) – The directory and other prefixes to find the feature file at

Returns The feature loaded into a FeatureVector object

Return type *satsense.image.FeatureVector*

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

S

Index

Symbols

A

B

C

E

F

G

H

I

J

L

M

N

P

R

S

T

W