nrtk

Release 0.7.0

Kitware, Inc.

Jun 13, 2024
Welcome to the documentation for the Natural Robustness Toolkit (NRTK), a platform created for developers seeking to rigorously evaluate and enhance the robustness of computer vision models. This toolkit simulates a wide range of real-world perturbations, focusing on sensor-specific variables such as changes in camera focal length and aperture diameter. It provides a detailed analysis of how these factors affect algorithm performance and expand existing datasets. Whether you’re dealing with subtle shifts in optical settings or more pronounced environmental changes, this toolkit gives you the insights and capabilities necessary to ensure your innovative computer vision solutions are resilient and reliable under diverse conditions.

This documentation is structured to provide you with straightforward and practical instructions and examples, so that you can effectively leverage the toolkit to enhance the robustness and reliability of your computer vision applications in facing real-world challenges.

1.1 Background

Computer vision models are sensitive to data distribution shifts, either due to synthetic image perturbations [1] or those naturally occurring in real data [2]. Existing image augmentation libraries (e.g. imgaug and albumentations) do not cover physics-based, sensor-specific perturbations that are relevant to operational data.

The gold standard for evaluating AI model robustness is to score against comprehensive real-test data spanning all dimensions of expected deployment-condition variability. While there is no substitute for collecting diverse test data, complete coverage is prohibitively expensive or impossible in many applications. You can use NRTK to force-multiply your finite test dataset in a principled way that covers the wider range of natural variations and corruptions expected during real-world deployment.

1.2 Use Cases

You can use NRTK to assess the robustness of computer vision models trained on satellite images to changes in different sensor parameters (e.g. focal length, aperture, pixel pitch, etc.). For example, one use case is the task of designing a new satellite to support AI-based detection and classification of particular objects. With NRTK, you can start with high-resolution aerial imagery with known ground-sample-distance, then emulate imagery that would have been collected from a hypothetical telescope with prescribed properties to explore the trade-off of telescope cost versus performance. You can also develop validated sets of sensor perturbation parameters and expanded datasets for comprehensive model test and evaluation (T&E).

Refer to the Getting Started section for an in-depth example of how to use NRTK.
1.3 Toolkit Overview

The nrtk package is an open source toolkit for evaluating the natural robustness of computer vision algorithms to various perturbations, including sensor-specific changes to camera focal length, aperture diameter, etc. Functionality is provided through Strategy and Adapter patterns to allow for modular integration into systems and applications.

The toolkit includes several types of general image perturbations, as well as sensor-based perturbations based on the open source library pyBSM [3]. pyBSM rigorously models radiative transfer and imaging-sensor physics, allowing a user to provide source images, ideally with minimal corruptions to start. These source images are then rendered to emulate pre-sensor, in-sensor, and post-sensor corruptions that would have been incurred by another sensor with precise specification (altitude, atmospheric turbulence, focal length, aperture size, focus blur, pixel pitch, quantum efficiency, shot/readout noise, and compression, among many others).

The nrtk package provides image perturbation followed by score generation and can work with any computer vision model in a black-box manner. The perturbations themselves are independent or agnostic of a downstream task, but nrtk’s interfaces allow for evaluation of classification and detection models.

The nrtk algorithms can also be organized according to their respective tasks:

- **Image perturbation:**
  - Image Perturbation
  - Perturbation Factory

- **Score generation:**
  - Scoring
  - End-to-End Generation and Scoring
1.4 References


There are two ways to obtain the nrtk package. The simplest is to install via the pip command. Alternatively, you can use Poetry (installation and usage) to acquire the source tree and develop locally.

2.1 From pip

$ pip install nrtk

This method will install all of the same functionality as when installing from source. If you have an existing installation and would like to upgrade your version, provide the -U/--upgrade option.

2.2 From Source

The following assumes Poetry is already installed.

2.2.1 Quick Start

$ cd /where/things/should/go/
$ git clone https://gitlab.jatic.net/jatic/kitware/nrtk ./
$ poetry install
$ poetry run pytest
$ cd docs
$ poetry run make html

2.2.2 Installing Python Dependencies

This project uses Poetry for dependency management, environment consistency, package building, version management, and publishing to PyPI. Dependencies are abstractly defined in the pyproject.toml file, as well as specifically pinned versions in the poetry.lock file, both of which can be found in the root of the source tree.

The following installs both installation and development dependencies as specified in the pyproject.toml file, with versions specified (including for transitive dependencies) in the poetry.lock file:

$ poetry install
2.2.3 Building the Documentation

The documentation for NRTK is maintained as a collection of reStructuredText documents in the docs/ folder of the project. The Sphinx documentation tool can process this documentation into a variety of formats, the most common of which is HTML.

Within the docs/ directory is a Unix Makefile (for Windows systems, a make.bat file with similar capabilities exists). This Makefile takes care of the work required to run Sphinx to convert the raw documentation to an attractive output format. For example, as shown in the Quick Start section (above), calling make html will generate HTML format documentation rooted at docs/_build/html/index.html.

Calling the command make help here will show the other documentation formats that may be available (although be aware that some of them require additional dependencies such as TeX or LaTeX).

Live Preview

While writing documentation in a markup format such as reStructuredText, it is very helpful to preview the formatted version of the text. While it is possible to simply run the make html command periodically, a more seamless workflow of this is available. Within the docs/ directory is a small Python script called sphinx_server.py that can simply be called with:

$ poetry run python sphinx_server.py

This will run a small process that watches the docs/ folder contents, as well as the source files in nrtk/, for changes. make html is re-run automatically when changes are detected. This will serve the resulting HTML files at http://localhost:5500. Having this URL open in a browser will provide you with an up-to-date preview of the rendered documentation.
Note: If you need to install NRTK, see Installation.

NRTK consists of three main parts:

1. **Image Perturbation**
2. **Perturbation Factories**
3. **Model Evaluation**

The following sections will guide you through setting up and using an example perturber.

### 3.1 Image Perturbation

The core of NRTK is based on image perturbation. NRTK offers a wide variety of ways to perturb images. `scikit-image`, `Pillow`, `openCV`, and `pyBSM` are used for various types of perturbation. The perturbation classes take an image and perform a transformation based on input parameters. The examples shown below focus on a `pyBSM` based perturber.

To see examples of other perturbations, the perturbers notebook shows initialization and use of `scikit-image`, `Pillow`, and `openCV` perturbers.

For this example, we are going to use the `PybsmPerturber` from `pyBSM`. This perturber is useful for creating new images based on existing parameters. The `PybsmSensor` and `PybsmScenario` classes contain the parameters for an existing sensor and environment, respectively.

```python
import numpy as np
import pybsm
from nrtk.impls.perturb_image.pybsm.scenario import PybsmScenario
from nrtk.impls.perturb_image.pybsm.sensor import PybsmSensor
from nrtk.impls.perturb_image.pybsm.perturber import PybsmPerturber

optTransWavelengths = np.array([0.58-.08,0.58+.08])*1.0e-6
f = 4  # telescope focal length (m)
p = .008e-3  # detector pitch (m)
sensor = PybsmSensor(
    # required
    name = 'L32511x',
    D = 275e-3,  # Telescope diameter (m)
    f = f,
    px = p,
    optTransWavelengths = optTransWavelengths,  #Optical system transmission, red  band._
```

(continues on next page)
\[ first \ (m) \]
\[ \text{# optional} \]
\[ \text{opticsTransmission} = 0.5 * \text{np.ones(optTransWavelengths.shape[0])}, \ # \text{guess at the full} \]
\[ \text{system optical transmission (excluding obscuration)} \]
\[ \text{eta} = 0.4, \ # \text{guess} \]
\[ \text{wx} = p, \ # \text{detector width is assumed to be equal to the pitch} \]
\[ \text{wy} = p, \ # \text{detector width is assumed to be equal to the pitch} \]
\[ \text{intTime} = 30.0e-3, \ # \text{integration time (s) - this is a maximum, the actual} \]
\[ \text{darkCurrent} = \text{pybsm.darkCurrentFromDensity(1e-5, p, p)}, \ # \text{dark current density} \]
\[ \text{readNoise} = 25.0, \ # \text{rms read noise (rms electrons)} \]
\[ \text{maxN} = 96000.0, \ # \text{maximum ADC level (electrons)} \]
\[ \text{bitdepth} = 11.9, \ # \text{bit depth} \]
\[ \text{maxWellFill} = 0.6, \ # \text{maximum allowable well fill (see the paper for the logic)} \]
\[ \text{behind this) \]
\[ \text{sx} = 0.25 * p/f, \ # \text{jitter (radians) - The Olson paper says that it's} \]
\[ \text{"good" so we'll guess} \ 1/4 \ \text{ifov rms} \]
\[ \text{sy} = 0.25 * p/f, \ # \text{jitter (radians) - The Olson paper says that it's} \]
\[ \text{"good" so we'll guess} \ 1/4 \ \text{ifov rms} \]
\[ \text{dax} = 100e-6, \ # \text{drift (radians/s) - again, we'll guess that it's} \]
\[ \text{"really good} \]
\[ \text{day} = 100e-6, \ # \text{drift (radians/s) - again, we'll guess that it's} \]
\[ \text{"really good} \]
\[ \text{qewavelengths} = \text{np.array([.3, .4, .5, .6, .7, .8, .9, 1.0, 1.1])} \times 1.0e-6, \]
\[ \text{qe} = \text{np.array([0.05, 0.6, 0.75, 0.85, .85, .75, .5, .2, 0])} \]
\]
\[ scenario = \text{PybsmScenario(} \]
\[ \text{name='niceday'}, \]
\[ \text{ihaze=1, \ # weather model} \]
\[ \text{altitude=9000.0, \ # sensor altitude} \]
\[ \text{groundRange=0.0, \ # sensor to target} \]
\[ \text{aircraftSpeed=100.0} \]
\[ \) \]
\[ \text{perturber=PybsmPerturber(sensor=sensor, scenario=scenario, groundRange=10000)} \]

In the example above, we have created a pyBSM perturber where the output image will have a groundRange of 10000m instead of 0m. The image below is the original image we will use for future perturbations.

The code block below shows the loading of the image above and the calling of the perturber. It is important to note that the ground sample distance (or img_gsd) is another parameter the user will have to provide. The resulting image is displayed below the code block.

**Listing 2: pyBSM Perturber Execution**

```python
import cv2

INPUT_IMG_FILE = './data/M-41 Walker Bulldog (USA) width 319cm height 272cm.tif'
image = cv2.imread(INPUT_IMG_FILE)
img_gsd = 3.19/165.0 # the width of the tank is 319 cm and it spans ~165 pixels in the
   # image
```
Fig. 1: Original image of a tank
perturbed_image = perturber.perturb(image, additional_params={'img_gsd': img_gsd})

Fig. 2: Image of a tank with ground range of 10000m.

Any of the parameters in either PybsmSensor or PybsmScenario can be modified; however, only one parameter can be modified with one value using the basic perturber. The next section will cover modifying multiple parameters and multiple values.

### 3.2 Perturbation Factories

Building upon image perturbation, perturbation factories are able to take a range of values for parameter(s) and perform multiple perturbations on the same image. This allows for quick and simple generation of multiple perturbations. The scikit-image, Pillow, and openCV perturbers use the StepPerturbImageFactory and the pyBSM perturber uses the CustomPybsmPerturbImageFactory.

Continuing on from the previous example, the snippet below shows the initialization of a CustomPybsmPerturbImageFactory. The theta_keys variable controls which parameter(s) we are modifying and thetas are the actual values of the parameter(s). In this example, we are modifying the focal length \( (f) \) with the values of 1, 2, and 3. The modified images are displayed below the code block.

```
from nrtk.impls.perturb_image_factory.pybsm import CustomPybsmPerturbImageFactory

focal_length_pf = CustomPybsmPerturbImageFactory(
    theta_keys='focal_length',
    thetas=[1, 2, 3],
)
```
sensor=sensor,
scenario=scenario,
theta_keys=['f'],
thetas=[[1, 2, 3]]
)

for idx, perturber in enumerate(focal_length_pf):
    perturbed_img = perturber(image, additional_params={'img_gsd': img_gsd})

Fig. 3: Image of a tank with focal length of 1m.

Fig. 4: Image of a tank with focal length of 2m.

Not only can you modify multiple values on one parameter, but you can also modify multiple parameters at the same time. The code block below shows the focal length and ground range variables being modified. The resulting images are displayed below the code block.
Fig. 5: Image of a tank with focal length of 3m.
Listing 4: CustomPybsmPerturbImageFactory with Multiple Parameters

```python
f_groung_range_pf = CustomPybsmPerturbImageFactory(
    sensor=sensor,
    scenario=scenario,
    theta_keys=['f', 'ground_range'],
    thetas=[[1, 2], [10000, 20000]]
)

for idx, perturber in enumerate(f_groung_range_pf):
    perturbed_img = perturber(image, additional_params={'img_gsd': img_gsd})
```

Fig. 6: Image of a tank with focal length of 1m and ground range of 10000m.

Fig. 7: Image of a tank with focal length of 2m and ground range of 10000m.

Fig. 8: Image of a tank with focal length of 1m and ground range of 20000m.

Fig. 9: Image of a tank with focal length of 2m and ground range of 20000m.

3.2. Perturbation Factories
3.3 Model Evaluation

NRTK provides functionality for evaluating models in the image classification and object detection tasks. The package also provides test orchestration functionality for performing evaluations over a sweep of parameters in order to test model response to varying severity of image degradation.

To see examples of image classification and object detection, the coco_scorer notebook from the examples directory shows different scoring techniques. For examples of model response to image degradations, there are two notebooks to check out. The simple_generic_generator notebook shows model response to image degradation through perturbers based on scikit-image, Pillow, and openCV. The simple_pybsm_generator notebook shows model response to image degradation through pyBSM-based perturbers.
The NRTK API consists of a number of object-oriented functor interfaces for item-response curve (IRC) generation, namely for assessing model response to perturbations on given input data. These interfaces focus on black-box IRC generation. In addition to the driver, or generator, of this task, there are two other main components: reference image perturbation in preparation for black-box testing and black-box scoring for model outputs. The `PerturbImageFactory` interface provides the utility to easily vary specified parameters of a particular perturber. The generator will execute upon a given perturber factory as well as a given model and scorer to generate the IRC for given input data. We define a few similar interfaces for performing the IRC generation, separated by the intermediate algorithmic use cases, one for object detection and one for image classification.

We explicitly do not require an abstraction for the black-box operations to fit inside. This is intended to allow for applications using these interfaces while leveraging existing functionality, which only need to perform data formatting to fit the input defined here. Note, however, some interfaces are defined for certain black-box concepts as part of the SMQTK ecosystem (e.g. in `SMQTK-Classifier`, `SMQTK-Detection`, and other `SMQTK-*` modules).

These interfaces are based on the plugin and configuration features provided by `SMQTK-Core`, to allow convenient hooks into implementation, discoverability, and factory generation from runtime configuration. This allows for both opaque discovery of interface implementations from a class-method on the interface class object, as well as instantiation of a concrete instance via a JSON-like configuration fed in from an outside resource.

Fig. 1: Abstract Interface Inheritance.

### 4.1 Image Perturbation

#### 4.1.1 Interface: `PerturbImage`

class nrtk.interfaces.perturb_image.PerturbImage

Algorithm that generates a perturbed image for the given input image stimulus as a `numpy.ndarray` type array.

```python
__call__(image: ndarray, additional_params: Dict[str, Any] = {}) → ndarray

Calls perturb() with the given input image.

abstract perturb(image: ndarray, additional_params: Dict[str, Any] = {}) → ndarray

Generate a perturbed image for the given image stimulus. Note perturbers that resize, rotate, or similarly affect the dimensions of an image may impact scoring if bounding boxes are not similarly transformed.

Parameters

- **image** – Input image as a numpy array.

- **additional_params** – A dictionary containing perturber implementation-specific input param-values pairs.
Returns
Perturbed image as numpy array, including matching dtype. Implementations should impart
no side effects upon the input image.

4.2 Perturbation Factory

4.2.1 Interface: PerturbImageFactory

class nrtk.interfaces.perturb_image_factory.PerturbImageFactory(perturber: Type[PerturbImage],
theta_key: str)

Factory class for producing PerturbImage instances of a specified type and configuration.

__getitem__(idx: int) → PerturbImage
Get the perturber for a specific index.

Parameters
idx – Index of desired perturber.

Raises
IndexError – The given index does not exist.

Returns
Perturber corresponding to the given index.

__init__(perturber: Type[PerturbImage], theta_key: str)
Initialize the factory to produce PerturbImage instances of the given type, varying the given theta_key
parameter.

Parameters
• perturber – Python implementation type of the PerturbImage interface to produce.
• theta_key – Perturber parameter to vary between instances.

Raises
TypeError – Given a perturber instance instead of type.

__iter__() → Iterator[PerturbImage]

Returns
Iterator for this factory.

__len__() → int

Returns
Number of perturber instances this factory will generate.

__next__() → PerturbImage

Raises
StopIteration – Iterator exhausted.

Returns
Next perturber instance.
get_config() \rightarrow \text{Dict[str, Any]}

Return a JSON-compliant dictionary that could be passed to this class’s from_config method to produce an instance with identical configuration.

In the most cases, this involves naming the keys of the dictionary based on the initialization argument names as if it were to be passed to the constructor via dictionary expansion. In some cases, where it doesn’t make sense to store some object constructor parameters are expected to be supplied at as configuration values (i.e. must be supplied at runtime), this method’s returned dictionary may leave those parameters out. In such cases, the object’s from_config class-method would also take additional positional arguments to fill in for the parameters that this returned configuration lacks.

Returns
JSON type compliant configuration dictionary.

Return type
dict

property theta_key: str
Get the perturber parameter to vary between instances.

abstract property thetas: Sequence[any]
Get the sequence of theta values this factory will iterate over.

4.3 Scoring

4.3.1 Interface: ScoreDetections
class nrtk.interfaces.score_detections.ScoreDetections

Interface abstracting the behavior of taking the actual and predicted detections and computing the corresponding metric scores.

Implementations should verify the validity of the input data.

Note that current implementations are not required to verify nor correct dimension (in)consistency, which may impact scoring.

__call__ (actual: Sequence[Sequence[Tuple[AxisAlignedBoundingBox, Dict[Hashable, Any]]]], predicted: Sequence[Sequence[Tuple[AxisAlignedBoundingBox, Dict[Hashable, float]]]]) \rightarrow \text{Sequence[float]}

Alias for ScoreDetection.score().

abstract score(actual: Sequence[Sequence[Tuple[AxisAlignedBoundingBox, Dict[Hashable, Any]]]],
predicted: Sequence[Sequence[Tuple[AxisAlignedBoundingBox, Dict[Hashable, float]]]]) \rightarrow \text{Sequence[float]}

Generate a sequence of scores corresponding to a specific metric.

Parameters

- actual – Ground truth bbox and class label pairs.
- predicted – Output detections from a detector with bbox and class-wise confidence scores.

Returns
Metric score values as a float-type sequence with the length matching the number of samples in the ground truth input.
4.4 End-to-End Generation and Scoring

4.4.1 Interface: GenerateObjectDetectorBlackboxResponse

class nrtk.interfaces.gen_object_detector_blackbox_response.
GenerateObjectDetectorBlackboxResponse

This interface describes the generation of item-response curves and scores for object detections with respect to
the given black-box object detector after input images are perturbed via the black-box perturber factory. Scoring
of these detections is computed with the given black-box scorer.

Note that dimension transformations are not currently accounted for and may impact scoring.

__call__ (blackbox_perturber_factories: Sequence[PerturbImageFactory], blackbox_detector:
DetectImageObjects, blackbox_scorer: ScoreDetections, img_batch_size: int, verbose: bool =
False) → Tuple[Sequence[Tuple[Dict[str, Any], float]], Sequence[Sequence[float]]]

Alias for :meth: GenerateObjectDetectorBlackboxResponse.generate.

abstract __getitem__ (idx: int) → Tuple[ndarray, Sequence[Tuple[AxisAlignedBoundingBox,
Dict[Hashable, float]], Dict[str, Any]]

Get the ``idx`` th image and groundtruth pair.

generate (blackbox_perturber_factories: Sequence[PerturbImageFactory], blackbox_detector:
DetectImageObjects, blackbox_scorer: ScoreDetections, img_batch_size: int, verbose: bool =
False) → Tuple[Sequence[Tuple[Dict[str, Any], float]], Sequence[Sequence[float]]]

Generate item-response curves for given parameters.

Parameters

- **blackbox_perturber_factories** – Sequence of factories to perturb stimuli.
- **blackbox_detector** – Detector to generate detections for perturbed stimuli.
- **blackbox_scorer** – Scorer to score detections.
- **img_batch_size** – The number of images to predict and score upon at once.
- **verbose** – Increases the verbosity of progress updates.

Returns

Item-response curve

Scores for each input stimuli
5.1 Image Perturbation

5.1.1 Class: AverageBlurPerturber

class nrtk.impls.perturb_image.generic.cv2.blur.AverageBlurPerturber(ksize: int = 1)
    Applies average blurring to the image stimulus.
    __init__(ksize: int = 1)
        Parameters
        ksize – Blurring kernel size.
    get_config() → Dict[str, Any]
        Return a JSON-compliant dictionary that could be passed to this class’s from_config method to produce an instance with identical configuration.
        In the most cases, this involves naming the keys of the dictionary based on the initialization argument names as if it were to be passed to the constructor via dictionary expansion. In some cases, where it doesn’t make sense to store some object constructor parameters are expected to be supplied at as configuration values (i.e. must be supplied at runtime), this method’s returned dictionary may leave those parameters out. In such cases, the object’s from_config class-method would also take additional positional arguments to fill in for the parameters that this returned configuration lacks.
        Returns
        JSON type compliant configuration dictionary.
        Return type
dict
    perturb(image: ndarray, additional_params: Dict[str, Any] = {}) → ndarray
        Return image stimulus after applying average blurring.
5.1.2 Class: BrightnessPerturber

class nrtk.impls.perturb_image.generic.PIL.enhance.BrightnessPerturber(factor: float = 1.0)
    Adjusts image stimulus brightness.

    perturb(image: ndarray, additional_params: Dict[str, Any] = {}) → ndarray
        Return image stimulus with adjusted brightness.

5.1.3 Class: ColorPerturber

class nrtk.impls.perturb_image.generic.PIL.enhance.ColorPerturber(factor: float = 1.0)
    Adjusts image stimulus color balance.

    perturb(image: ndarray, additional_params: Dict[str, Any] = {}) → ndarray
        Return image stimulus with adjusted color balance.

5.1.4 Class: ContrastPerturber

class nrtk.impls.perturb_image.generic.PIL.enhance.ContrastPerturber(factor: float = 1.0)
    Adjusts image stimulus contrast.

    perturb(image: ndarray, additional_params: Dict[str, Any] = {}) → ndarray
        Return image stimulus with adjusted contrast.

5.1.5 Class: GaussianBlurPerturber

class nrtk.impls.perturb_image.generic.cv2.blur.GaussianBlurPerturber(ksize: int = 1)
    Applies Gaussian blurring to the image stimulus.

    __init__(ksize: int = 1)
        Parameters
            ksize – Blurring kernel size.

    get_config() → Dict[str, Any]
        Return a JSON-compliant dictionary that could be passed to this class’s from_config method to produce
        an instance with identical configuration.

        In the most cases, this involves naming the keys of the dictionary based on the initialization argument names
        as if it were to be passed to the constructor via dictionary expansion. In some cases, where it doesn’t make
        sense to store some object constructor parameters are expected to be supplied at as configuration values
        (i.e. must be supplied at runtime), this method’s returned dictionary may leave those parameters out. In
        such cases, the object’s from_config class-method would also take additional positional arguments to fill
        in for the parameters that this returned configuration lacks.

        Returns
            JSON type compliant configuration dictionary.

        Return type
dict

    perturb(image: ndarray, additional_params: Dict[str, Any] = {}) → ndarray
        Return image stimulus after applying Gaussian blurring.
5.1.6 Class: GaussianNoisePerturber

class nrtk.impls.perturb_image.generic.skimage.random_noise.GaussianNoisePerturber(rng: Generator | int | None = None, mean: float = 0.0, var: float = 0.05)

Adds Gaussian-distributed additive noise to image stimulus.

perturb(image: ndarray, additional_params: Dict[str, Any] = {}) → ndarray

Return image stimulus with Gaussian noise.

5.1.7 Class: MedianBlurPerturber

class nrtk.impls.perturb_image.generic.cv2.blur.MedianBlurPerturber(ksize: int = 1)

Applies median blurring to the image stimulus.

__init__(ksize: int = 1)

Parameters
ksize – Blurring kernel size.

get_config() → Dict[str, Any]

Return a JSON-compliant dictionary that could be passed to this class’s from_config method to produce an instance with identical configuration.

In the most cases, this involves naming the keys of the dictionary based on the initialization argument names as if it were to be passed to the constructor via dictionary expansion. In some cases, where it doesn’t make sense to store some object constructor parameters are expected to be supplied at as configuration values (i.e. must be supplied at runtime), this method’s returned dictionary may leave those parameters out. In such cases, the object’s from_config class-method would also take additional positional arguments to fill in for the parameters that this returned configuration lacks.

Returns
JSON type compliant configuration dictionary.

Return type
dict

perturb(image: ndarray, additional_params: Dict[str, Any] = {}) → ndarray

Return image stimulus after applying Gaussian blurring.
5.1.8 Class: nop_perturber

class nrtk.impls.perturb_image.generic.nop_perturber.NOPPerturber

Example implementation of the PerturbImage interface. An instance of this class acts as a functor to generate a perturbed image for the given input image stimulus.

This class, in particular, serves as pass-through “no operation” (NOP) perturber.

get_config() \rightarrow \text{Dict[\text{str}, \text{Any}]}

Return a JSON-compliant dictionary that could be passed to this class’s \text{from_config} method to produce an instance with identical configuration.

In the most cases, this involves naming the keys of the dictionary based on the initialization argument names as if it were to be passed to the constructor via dictionary expansion. In some cases, where it doesn’t make sense to store some object constructor parameters are expected to be supplied at as configuration values (i.e. must be supplied at runtime), this method’s returned dictionary may leave those parameters out. In such cases, the object’s \text{from_config} class-method would also take additional positional arguments to fill in for the parameters that this returned configuration lacks.

\text{Returns} \\
\text{JSON type compliant configuration dictionary.}

\text{Return type} \\
dict

perturb(image: \text{ndarray}, additional_params: \text{Dict[\text{str}, \text{Any}]} = \{\}) \rightarrow \text{ndarray}

Return unperturbed image.

5.1.9 Class: PepperNoisePerturber

class nrtk.impls.perturb_image.generic.skimage.random_noise.PepperNoisePerturber(rng: \text{Generator} | \text{int} | \text{None} = \text{None}, amount: \text{float} = 0.05)

Adds pepper noise to image stimulus.

perturb(image: \text{ndarray}, additional_params: \text{Dict[\text{str}, \text{Any}]} = \{\}) \rightarrow \text{ndarray}

Return image stimulus with pepper noise.

5.1.10 Class: PybsmPerturber

class nrtk.impls.perturb_image.pybsm.perturber.PybsmPerturber(sensor: \text{PybsmSensor}, scenario: \text{PybsmScenario}, reflectance_range: \text{ndarray} = \text{array([0.05, 0.5])}, **kwargs: \text{Any})

\text{__call__}(image: \text{ndarray}, additional_params: \text{Dict[\text{str}, \text{Any}]} = \{\}) \rightarrow \text{ndarray}

Alias for NIIRS.apply().
__init__(sensor: PybsmSensor, scenario: PybsmScenario, reflectance_range: ndarray = array([0.05, 0.5]), **kwargs: Any) → None

Parameters

• sensor – pyBSM sensor object.
• scenario – pyBSM scenario object.
• reflectance_range – Array of reflectances that correspond to pixel values.

Raises

ValueError if reflectance_range length != 2

Raises

ValueError if reflectance_range not strictly ascending

__repr__() → str

Return repr(self).

__str__() → str

Return str(self).

classmethod from_config(config_dict: Dict, merge_default: bool = True) → PybsmPerturber

Instantiate a new instance of this class given the configuration JSON-compliant dictionary encapsulating initialization arguments.

This base method is adequate without modification when a class’s constructor argument types are JSON-compliant. If one or more are not, however, this method then needs to be overridden in order to convert from a JSON-compliant stand-in into the more complex object the constructor requires. It is recommended that when complex types are used they also inherit from the Configurable in order to hopefully make easier the conversion to and from JSON-compliant stand-ins.

When this method does need to be overridden, this usually looks like the following pattern:

D = TypeVar("D", bound="MyClass")

class MyClass (Configurable):

    @classmethod
    def from_config(
        cls: Type[D],
        config_dict: Dict,
        merge_default: bool = True
    ) -> D:
        # Perform a shallow copy of the input `config_dict` which
        # is important to maintain idempotency.
        config_dict = dict(config_dict)

        # Optionally guarantee default values are present in the
        # configuration dictionary. This is useful when the
        # configuration dictionary input is partial and the logic
        # contained here wants to use config parameters that may
        # have defaults defined in the constructor.
        if merge_default:
            config_dict = merge_dict(cls.get_default_config(),
                                      config_dict)

(continues on next page)
Perform any overriding of `config_dict` values here.

Create and return an instance using the super method.

```python
return super().from_config(config_dict, merge_default=merge_default)
```

Note on type annotations: When defining a sub-class of configurable and override this class method, we will need to defined a new TypeVar that is bound at the new class type. This is because super requires a type to be given that descends from the implementing type. If `C` is used as defined in this interface module, which is upper-bounded on the base `Configurable` class, the type analysis will see that we are attempting to invoke super with a type that may not strictly descend from the implementing type (`MyClass` in the example above), and cause an error during type analysis.

Parameters

- `config_dict` (dict) – JSON compliant dictionary encapsulating a configuration.
- `merge_default` (bool) – Merge the given configuration on top of the default provided by `get_default_config`.

Returns

- Constructed instance from the provided config.

get_config() → Dict[str, Any]

Return a JSON-compliant dictionary that could be passed to this class’s `from_config` method to produce an instance with identical configuration.

In the most cases, this involves naming the keys of the dictionary based on the initialization argument names as if it were to be passed to the constructor via dictionary expansion. In some cases, where it doesn’t make sense to store some object constructor parameters are expected to be supplied at as configuration values (i.e. must be supplied at runtime), this method’s returned dictionary may leave those parameters out. In such cases, the object’s `from_config` class-method would also take additional positional arguments to fill in for the parameters that this returned configuration lacks.

Returns

- JSON type compliant configuration dictionary.

Return type
dict

classmethod get_default_config() → Dict[str, Any]

Generate and return a default configuration dictionary for this class. This will be primarily used for generating what the configuration dictionary would look like for this class without instantiating it.

By default, we observe what this class’s constructor takes as arguments, turning those argument names into configuration dictionary keys. If any of those arguments have defaults, we will add those values into the configuration dictionary appropriately. The dictionary returned should only contain JSON compliant value types.

It is not be guaranteed that the configuration dictionary returned from this method is valid for construction of an instance of this class.

Returns

- Default configuration dictionary for the class.

Return type
dict
>>> # noinspection PyUnresolvedReferences
class SimpleConfig(Configurable):
    def __init__(self, a=1, b='foo'):
        self.a = a
        self.b = b
    def get_config(self):
        return {'a': self.a, 'b': self.b}

self = SimpleConfig()
config = self.get_default_config()
assert config == {'a': 1, 'b': 'foo'}

perturb(image: ndarray, additional_params: Dict[str, Any] = {}) → ndarray

Raises
    ValueError if ‘img_gsd’ not present in additional_params

5.1.11 Class: PybsmScenario

class nrtk.impls.perturb_image.pybsm.scenario.PybsmScenario(name: str, ihaze: int, altitude: float, groundRange: float | None = 0.0, aircraftSpeed: float | None = 0.0, targetReflectance: float | None = 0.15, targetTemperature: float | None = 295.0, backgroundReflectance: float | None = 0.07, backgroundTemperature: float | None = 293.0, haWindspeed: float | None = 21.0, cn2at1m: float | None = 1.7e-14)

Wrapper for pybsm.scenario.

ihaze:
    MODTRAN code for visibility, valid options are ihaze = 1 (Rural extinction with 23 km visibility) or ihaze = 2 (Rural extinction with 5 km visibility)

altitude:
    sensor height above ground level in meters. The database includes the following altitude options: 2 32.55 75 150 225 500 meters, 1000 to 12000 in 1000 meter steps, and 14000 to 20000 in 2000 meter steps, 24500

groundRange:
    distance on the ground between the target and sensor in meters. The following ground ranges are included in the database at each altitude until the ground range exceeds the distance to the spherical earth horizon: 0 100 500 1000 to 20000 in 1000 meter steps, 22000 to 80000 in 2000 m steps, and 85000 to 300000 in 5000 meter steps.

aircraftSpeed:
    ground speed of the aircraft (m/s)

targetReflectance:
    object reflectance (unitless)

targetTemperature:
    object temperature (Kelvin)

backgroundReflectance:
    background reflectance (unitless)
**backgroundTemperature:**
background temperature (Kelvin)

**haWindspeed:**
the high altitude windspeed (m/s). Used to calculate the turbulence profile.

**cn2at1m:**
the refractive index structure parameter “near the ground” (e.g. at h = 1 m). Used to calculate the turbulence profile.

**Raises**
ValueError if ihaze not in acceptable ihaze values

**Raises**
ValueError if altitude not in acceptable altitude values

**Raises**
ValueError if ground range not in acceptable ground range values

**get_config() → Dict[str, Any]**
Return a JSON-compliant dictionary that could be passed to this class’s `from_config` method to produce
an instance with identical configuration.

In the most cases, this involves naming the keys of the dictionary based on the initialization argument names
as if it were to be passed to the constructor via dictionary expansion. In some cases, where it doesn’t make
sense to store some object constructor parameters are expected to be supplied at as configuration values
(i.e. must be supplied at runtime), this method’s returned dictionary may leave those parameters out. In
such cases, the object’s `from_config` class-method would also take additional positional arguments to fill
in for the parameters that this returned configuration lacks.

**Returns**
JSON type compliant configuration dictionary.

**Return type**
dict

### 5.1.12 Class: PybsmSensor

**class** nrtk.impls.perturb_image.pybsm.sensor.PybsmSensor(name: str, D: float, f: float, px: float,
optTransWavelengths: ndarray, opticsTransmission: ndarray | None = None, eta: float | None = 0.0, wx: float |
None = None, wy: float | None = None, intTime: float | None = 1.0, darkCurrent: float | None = 0.0, readNoise: float |
None = 0.0, maxN: float | None = 100000000.0, bitdepth: float | None = 100.0, maxWellFill: float | None = 1.0, sx: float |
None = 0.0, sy: float | None = 0.0, dax: float | None = 0.0, day: float | None = 0.0, qewavelengths: ndarray | None = None, qe:
ndarray | None = None)

Wrapper for pybsm.sensor.
Attributes (the first four are mandatory):

name :
    name of the sensor (string)

D :
    effective aperture diameter (m)

f :
    focal length (m)

px and py :
    detector center-to-center spacings (pitch) in the x and y directions (m)

optTransWavelengths :
    numpy array specifying the spectral bandpass of the camera (m). At minimum, start and end wavelength should be specified.

opticsTransmission :
    full system in-band optical transmission (unitless). Loss due to any telescope obscuration should not be included in with this optical transmission array.

eta :
    relative linear obscuration (unitless)

wx and wy :
    detector width in the x and y directions (m)

intTime :
    maximum integration time (s)

qe :
    quantum efficiency as a function of wavelength (e-/photon)

qewavelengths :
    wavelengths corresponding to the array qe (m)

otherIrradiance :
    spectral irradiance from other sources (W/m^2 m). This is particularly useful for self emission in infrared cameras. It may also represent stray light.

darkCurrent :
    detector dark current (e-/s)

maxN :
    detector electron well capacity (e-)

maxFill :
    desired well fill, i.e. Maximum well size x Desired fill fraction

bitdepth :
    resolution of the detector ADC in bits (unitless)

ntdi :
    number of TDI stages (unitless)

coldshieldTemperature :
    temperature of the cold shield (K). It is a common approximation to assume that the coldshield is at the same temperature as the detector array.

opticsTemperature :
    temperature of the optics (K)
nrtk, Release 0.7.0

**opticsEmissivity**: emissivity of the optics (unitless) except for the cold filter. A common approximation is 1-optics transmissivity.

**coldfilterTransmission**: transmission through the cold filter (unitless)

**coldfilterTemperature**: temperature of the cold filter. It is a common approximation to assume that the filter is at the same temperature as the detector array.

**coldfilterEmissivity**: emissivity through the cold filter (unitless). A common approximation is 1-cold filter transmission

**sx and sy**: Root-mean-squared jitter amplitudes in the x and y directions respectively. (rad)

**dax and day**: line-of-sight angular drift rate during one integration time in the x and y directions respectively. (rad/s)

**pv**: wavefront error phase variance (rad^2) - tip: write as (2*pi*waves of error)^2

**pvwavelength**: wavelength at which pv is obtained (m)

**Lx and Ly**: correlation lengths of the phase autocorrelation function. Apparently, it is common to set the Lx and Ly to the aperture diameter. (m)

**otherNoise**: a catch all for noise terms that are not explicitly included elsewhere (read noise, photon noise, dark current, quantization noise are all already included)

**filterKernel**: 2-D filter kernel (for sharpening or whatever). Note that the kernel is assumed to sum to one.

**framestacks**: the number of frames to be added together for improved SNR.

.raises
  ValueError if optTransWavelengths length < 2

.raises
  ValueError if optTransWavelengths is not ascending

.raises
  ValueError if optTransWavelengths and (if provided) opticsTransmission lengths are different

**classmethod from_config**(config_dict: Dict, merge_default: bool = True) -> *PybsmSensor*

Instantiate a new instance of this class given the configuration JSON-compliant dictionary encapsulating initialization arguments.

This base method is adequate without modification when a class’s constructor argument types are JSON-compliant. If one or more are not, however, this method then needs to be overridden in order to convert from a JSON-compliant stand-in into the more complex object the constructor requires. It is recommended that when complex types *are* used they also inherit from the Configurable in order to hopefully make easier the conversion to and from JSON-compliant stand-ins.

When this method *does* need to be overridden, this usually looks like the following pattern:
```python
D = TypeVar("D", bound="MyClass")

class MyClass (Configurable):
    @classmethod
    def from_config(
        cls: Type[D],
        config_dict: Dict,
        merge_default: bool = True
    ) -> D:
        # Perform a shallow copy of the input `config_dict` which
        # is important to maintain idempotency.
        config_dict = dict(config_dict)

        # Optionally guarantee default values are present in the
        # configuration dictionary. This is useful when the
        # configuration dictionary input is partial and the logic
        # contained here wants to use config parameters that may
        # have defaults defined in the constructor.
        if merge_default:
            config_dict = merge_dict(cls.get_default_config(),
                                      config_dict)

        # # Perform any overriding of `config_dict` values here.

        # Create and return an instance using the super method.
        return super().from_config(config_dict,
                                     merge_default=merge_default)
```

**Note on type annotations:** When defining a sub-class of configurable and override this class method, we will need to define a new TypeVar that is bound at the new class type. This is because super requires a type to be given that descends from the implementing type. If C is used as defined in this interface module, which is upper-bounded on the base Configurable class, the type analysis will see that we are attempting to invoke super with a type that may not strictly descend from the implementing type (MyClass in the example above), and cause an error during type analysis.

**Parameters**
- **config_dict (dict)** – JSON compliant dictionary encapsulating a configuration.
- **merge_default (bool)** – Merge the given configuration on top of the default provided by `get_default_config`.

**Returns**
- `get_config()` → Dict[str, Any]

Return a JSON-compliant dictionary that could be passed to this class’s `from_config` method to produce an instance with identical configuration.

In the most cases, this involves naming the keys of the dictionary based on the initialization argument names as if it were to be passed to the constructor via dictionary expansion. In some cases, where it doesn’t make sense to store some object constructor parameters are expected to be supplied at as configuration values (i.e. must be supplied at runtime), this method’s returned dictionary may leave those parameters out.

5.1. Image Perturbation
such cases, the object's `from_config` class-method would also take additional positional arguments to fill in for the parameters that this returned configuration lacks.

**Returns**

JSON type compliant configuration dictionary.

**Return type**

dict

### 5.1.13 Class: SaltAndPepperNoisePerturber

class `nrtk.impls.perturb_image.generic.skimage.random_noise.SaltAndPepperNoisePerturber`(
    rng: Generator | int | None = None,
    amount: float = 0.05,
    salt_vs_pepper: float = 0.5)

Adds salt & pepper noise to image stimulus.

**get_config()** → Dict[str, Any]

Return a JSON-compliant dictionary that could be passed to this class’s `from_config` method to produce an instance with identical configuration.

In the most cases, this involves naming the keys of the dictionary based on the initialization argument names as if it were to be passed to the constructor via dictionary expansion. In some cases, where it doesn’t make sense to store some object constructor parameters are expected to be supplied at as configuration values (i.e. must be supplied at runtime), this method’s returned dictionary may leave those parameters out. In such cases, the object’s `from_config` class-method would also take additional positional arguments to fill in for the parameters that this returned configuration lacks.

**Returns**

JSON type compliant configuration dictionary.

**Return type**

dict

**perturb**(*image: ndarray, additional_params: Dict[str, Any] = {}*) → ndarray

Return image stimulus with S&P noise.
5.1.14 Class: SaltNoisePerturber

class nrtk.impls.perturb_image.generic.skimage.random_noise.SaltNoisePerturber(rng: Generator | int | None = None, amount: float = 0.05)

Adds salt noise to image stimulus.

perturb(image: ndarray, additional_params: Dict[str, Any] = {}) → ndarray
    Return image stimulus with salt noise.

5.1.15 Class: SharpnessPerturber

class nrtk.impls.perturb_image.generic.PIL.enhance.SharpnessPerturber(factor: float = 1.0)

Adjusts image stimulus sharpness.

__init__(factor: float = 1.0)

    Parameters
   (rng – Enhancement factor.

perturb(image: ndarray, additional_params: Dict[str, Any] = {}) → ndarray
    Return image stimulus with adjusted sharpness.

5.1.16 Class: SpeckleNoisePerturber

class nrtk.impls.perturb_image.generic.skimage.random_noise.SpeckleNoisePerturber(rng: Generator | int | None = None, mean: float = 0.0, var: float = 0.05)

Adds multiplicative noise to image stimulus. Noise is Gaussian-based.

perturb(image: ndarray, additional_params: Dict[str, Any] = {}) → ndarray
    Return image stimulus with multiplicative noise.
5.2 Perturbation Factory

5.2.1 Class: CustomPybsmPerturbImageFactory

class nrtk.impls.perturb_image_factory.pybsm.CustomPybsmPerturbImageFactory(
    sensor: PybsmSensor,
    scenario: PybsmScenario,
    theta_keys: Sequence[str],
    thetas: Sequence[Any])

__init__(sensor: PybsmSensor, scenario: PybsmScenario, theta_keys: Sequence[str], thetas: Sequence[Any])

Parameters

• sensor – pyBSM sensor object.
• scenario – pyBSM scenario object.
• theta_keys – Perturber parameter(s) to vary between instances.
• thetas – Perturber parameter(s) values to vary between instances.

5.2.2 Class: StepPerturbImageFactory

class nrtk.impls.perturb_image_factory.generic.step.StepPerturbImageFactory(
    perturber: Type[PerturbImage],
    theta_key: str,
    start: int, stop: int, step: int = 1)

Simple PerturbImageFactory implementation to step through the given range of values.

__init__(perturber: Type[PerturbImage], theta_key: str, start: int, stop: int, step: int = 1)

Initialize the factory to produce PerturbImage instances of the given type, varying the given theta_key parameter from start to stop with given step.

Parameters

• perturber – Python implementation type of the PerturbImage interface to produce.
• theta_key – Perturber parameter to vary between instances.
• start – Initial value of desired range (inclusive).
• stop – Final value of desired range (exclusive).
• step – Step value between instances.

Raises

TypeError – Given a perturber instance instead of type.

get_config() → Dict[str, Any]

Return a JSON-compliant dictionary that could be passed to this class’s from_config method to produce an instance with identical configuration.
In the most cases, this involves naming the keys of the dictionary based on the initialization argument names as if it were to be passed to the constructor via dictionary expansion. In some cases, where it doesn’t make sense to store some object constructor parameters are expected to be supplied at as configuration values (i.e. must be supplied at runtime), this method’s returned dictionary may leave those parameters out. In such cases, the object’s from_config class-method would also take additional positional arguments to fill in for the parameters that this returned configuration lacks.

**Returns**
JSON type compliant configuration dictionary.

**Return type**
dict

_property theta_key:_ str
Get the perturber parameter to vary between instances.

_property thetas:_ Sequence[int]
Get the sequence of theta values this factory will iterate over.

### 5.3 Scoring

#### 5.3.1 Class: ClassAgnosticPixelwiseIoUScorer

class nrtk.impls.score_detections.class_agnostic_pixelwise_iou_scorer.

**ClassAgnosticPixelwiseIoUScorer**
An implementation of the ScoreDetection interface that computes the Pixelwise IoU scores in a Class-Agnostic manner. The call to the scorer method returns a sequence of float values containing the Pixelwise IoU scores for the specified ground truth and predictions inputs.

**get_config() → Dict[str, Any]**
Return a JSON-compliant dictionary that could be passed to this class’s from_config method to produce an instance with identical configuration.

In the most cases, this involves naming the keys of the dictionary based on the initialization argument names as if it were to be passed to the constructor via dictionary expansion. In some cases, where it doesn’t make sense to store some object constructor parameters are expected to be supplied at as configuration values (i.e. must be supplied at runtime), this method’s returned dictionary may leave those parameters out. In such cases, the object’s from_config class-method would also take additional positional arguments to fill in for the parameters that this returned configuration lacks.

**Returns**
JSON type compliant configuration dictionary.

**Return type**
dict

**score**(actual: Sequence[Sequence[Tuple[AxisAlignedBoundingBox, Dict[Hashable, Any]]]], predicted: Sequence[Sequence[Tuple[AxisAlignedBoundingBox, Dict[Hashable, float]]]]) → Sequence[float]
Computes pixelwise IoU scores and returns sequence of float values equal to the length of the input data.
5.3.2 Class: COCOScorer

class nrtk.impls.score_detections.coco_scorer.COCOScorer(gt_path: str, stat_index: int = 0)

An implementation of the ScoreDetection interface that conforms to the COCO data formatting and metrics. An instance of this class reads in the path to the ground truth data and specifies a particular statistic index. Finally, the call to the scorer method returns a set of float metric values for the specified statistic index.

get_config() → Dict[str, Any]

Return a JSON-compliant dictionary that could be passed to this class’s from_config method to produce an instance with identical configuration.

score(actual: Sequence[Sequence[Tuple[AxisAlignedBoundingBox, Dict[Hashable, Any]]]], predicted: Sequence[Sequence[Tuple[AxisAlignedBoundingBox, Dict[Hashable, float]]]]) → Sequence[float]

Computes scores for a particular statistic index and returns sequences of float values equal to the length of the input data.

5.3.3 Class: NOPScorer

class nrtk.impls.score_detections.nop_scorer.NOPScorer

Example implementation of the ScoreDetection interface. An instance of this class acts as a functor to generate scores for a specific metric based on a given set of ground truth and predicted detections.

This class, in particular, serves as a pass-through “no operation” (NOP) scorer.

get_config() → Dict[str, Any]

Return a JSON-compliant dictionary that could be passed to this class’s from_config method to produce an instance with identical configuration.

score(actual: Sequence[Sequence[Tuple[AxisAlignedBoundingBox, Dict[Hashable, Any]]]], predicted: Sequence[Sequence[Tuple[AxisAlignedBoundingBox, Dict[Hashable, float]]]]) → Sequence[float]

Return sequence of zeros equal to the length of the ground truth input.
5.3.4 Class: RandomScorer

class nrtk.impls.score_detections.random_scorer.RandomScorer(rng: int | None = None)

An implementation of the ScoreDetection interface that serves as a simple test for reproducibility. An instance of this class acts as a functor to generate scores for a specific metric based on a given set of ground truth and predicted detections.

This class, in particular, implements a random scorer that returns random float values.

get_config() → Dict[str, Any]

Return a JSON-compliant dictionary that could be passed to this class’s from_config method to produce an instance with identical configuration.

In the most cases, this involves naming the keys of the dictionary based on the initialization argument names as if it were to be passed to the constructor via dictionary expansion. In some cases, where it doesn’t make sense to store some object constructor parameters are expected to be supplied at as configuration values (i.e. must be supplied at runtime), this method’s returned dictionary may leave those parameters out. In such cases, the object’s from_config class-method would also take additional positional arguments to fill in for the parameters that this returned configuration lacks.

Returns

JSON type compliant configuration dictionary.

Return type
dict

score(actual: Sequence[Sequence[Tuple[AxisAlignedBoundingBox, Dict[Hashable, Any]]]], predicted: Sequence[Sequence[Tuple[AxisAlignedBoundingBox, Dict[Hashable, float]]]]) → Sequence[float]

Return sequence of random float values equal to the length of the ground truth input.

5.4 End-to-End Generation and Scoring

5.4.1 Class: SimpleGenericGenerator

class nrtk.impls.gen_object_detector_blackbox_response.simple_generic_generator.SimpleGenericGenerator(images: Sequence[ndarray], groundtruth: Sequence[Sequence[Tuple[AxisAlignedBoundingBox, Dict[Hashable, float]]]])

Example implementation of the GenerateObjectDetectorBlackboxResponse interface.

__getitem__(idx: int) → Tuple[ndarray, Sequence[Tuple[AxisAlignedBoundingBox, Dict[Hashable, float]]]], Dict[str, Any]]

Get the image and groundtruth pair for a specific index.

Parameters

idx – Index of desired data pair.

Raises

IndexError – The given index does not exist.
Returns
Data pair corresponding to the given index.

__init__(images: Sequence[ndarray], groundtruth: Sequence[Sequence[Tuple[AxisAlignedBoundingBox, Dict[Hashable, float]]]])
Generate response curve for given images and groundtruth.

Parameters
- **images** – Sequence of images to generate responses for.
- **groundtruth** – Sequence of sequences of detections corresponding to each image.

Raises
**ValueError** – Images and groundtruth data have a size mismatch.

__len__() \rightarrow int

Returns
Number of image/groundtruth pairs this generator holds.

get_config() \rightarrow Dict[str, Any]
Return a JSON-compliant dictionary that could be passed to this class’s from_config method to produce an instance with identical configuration.

In the most cases, this involves naming the keys of the dictionary based on the initialization argument names as if it were to be passed to the constructor via dictionary expansion. In some cases, where it doesn’t make sense to store some object constructor parameters are expected to be supplied at as configuration values (i.e. must be supplied at runtime), this method’s returned dictionary may leave those parameters out. In such cases, the object’s from_config class-method would also take additional positional arguments to fill in for the parameters that this returned configuration lacks.

Returns
JSON type compliant configuration dictionary.

Return type
dict

5.4.2 Class: SimplePybsmGenerator

class nrtk.impls.gen_object_detector_blackbox_response.simple_pybsm_generator.SimplePybsmGenerator(images: Sequence[ndarray], img_sds: Sequence[float], groundtruth: Sequence[Sequence[Tuple[AxisAlignedBoundingBox, Dict[Hashable, float]]]])
Example implementation of the GenerateObjectDetectorBlackboxResponse interface.

__getitem__(idx: int) \rightarrow Tuple[ndarray, Sequence[Tuple[AxisAlignedBoundingBox, Dict[Hashable, float]]]], Dict[str, Any]]
Get the image and groundtruth pair for a specific index.
Parameters
idx – Index of desired data pair.

Raises
IndexError – The given index does not exist.

Returns
Data pair corresponding to the given index.

__init__(images: Sequence[ndarray], imggds: Sequence[float], groundtruth: Sequence[Sequence[AxisAlignedBoundingBox, Dict[Hashable, float]]])
Generate response curve for given images and groundtruth.

Parameters
• images – Sequence of images to generate responses for.
• groundtruth – Sequence of sequences of detections corresponding to each image.

Raises
ValueError – Images and groundtruth data have a size mismatch.

__len__() → int
Returns
Number of image/groundtruth pairs this generator holds.

generate(blackbox_perturber_factories: Sequence[PerturbImageFactory], blackbox_detector: DetectImageObjects, blackbox_scorer: ScoreDetections, img_batch_size: int, verbose: bool = False) → Tuple[Sequence[Tuple[Dict[str, Any], float]], Sequence[Sequence[float]]]
Generate item-response curves for given parameters.

Parameters
• blackbox_perturber_factories – Sequence of factories to perturb stimuli.
• blackbox_detector – Detector to generate detections for perturbed stimuli.
• blackbox_scorer – Scorer to score detections.
• img_batch_size – The number of images to predict and score upon at once.
• verbose – Increases the verbosity of progress updates.

Returns
Item-response curve

Scores for each input stimuli

get_config() → Dict[str, Any]
Return a JSON-compliant dictionary that could be passed to this class’s from_config method to produce an instance with identical configuration.

In the most cases, this involves naming the keys of the dictionary based on the initialization argument names as if it were to be passed to the constructor via dictionary expansion. In some cases, where it doesn’t make sense to store some object constructor parameters are expected to be supplied at as configuration values (i.e. must be supplied at runtime), this method’s returned dictionary may leave those parameters out. In such cases, the object’s from_config class-method would also take additional positional arguments to fill in for the parameters that this returned configuration lacks.

Returns
JSON type compliant configuration dictionary.
Return type

dict
The process for reviewing and integrating branches into nrtk is described below.
For guidelines on contributing, see CONTRIBUTING.md.
For guidelines on the release process, see Release Process and Notes.

The review process consists of the following parts:

- **Merge Request (MR)**
  - **Workflow Status**
    - Draft
    - Open
    - Closed
  - **Continuous Integration**
    - Code Style Consistency ([test-py-lint](#))
    - Static Type Analysis ([test-py-typecheck](#))
    - Documentation Build ([test-docs-build](#))
    - Unit Tests ([test-pytest](#))
    - Code Coverage ([test-coverage-percent](#))
    - Release Notes Check ([test-release-notes-check](#))
    - Example Notebooks Execution ([test-notebooks](#))
- **Human Review**
  - Notebooks
- **Resolving a Branch**
  - Merge
  - Close
6.1 Merge Request (MR)

An MR is initiated by a user intending to integrate a branch from their forked repository. Before the branch is integrated into the nrtk master branch, it must first go through a series of checks and a review to ensure that the branch is consistent with the rest of the repository and doesn’t contain any issues.

6.1.1 Workflow Status

The submitter must set the status of their MR:

Draft

Indicates that the submitter does not think that the MR is in a reviewable or mergeable state. Once they complete their work and think that the MR is ready to be considered for merger, they may set the status to Open.

Open

Indicates that an MR is ready for review and that the submitter of the MR thinks that the branch is ready to be merged. If a review is received that requests substantial changes to the contributed content, effectively returning the task at hand into a “development” phase, the MR status should be changed to Draft (by the author of the MR) to indicate that such changes are underway.

If the submitter is still working on the MR and simply wants feedback, they must request it and leave their branch marked as a Draft.

Closed

Indicates that the MR is resolved or discarded.

6.2 Continuous Integration

The following checks are included in the automated portion of the review process, and are triggered whenever a merge-request is made or changes, a tag is created, or when the main branch is updated. These are run as part of the CI/CD pipeline driven by GitLab CI pipelines, and defined by the .gitlab-ci.yml file. The success or failure of each may be seen in-line in a submitted MR in the “Checks” section of the MR view.

6.2.1 Code Style Consistency (test-py-lint)

Runs flake8 to quality check the code style. You can run this check manually in your local repository with poetry run flake8.

Passage of this check is strictly required.
6.2.2 Static Type Analysis (test-py-typecheck)

Performs static type analysis. You can run this check manually in your local repository with `poetry run mypy`. Passage of this check is strictly required.

6.2.3 Documentation Build (test-docs-build)

Performs a build of our Sphinx documentation. Passage of this check is strictly required.

6.2.4 Unit Tests (test-pytest)

Runs the unittests created under `tests/` as well as any doctests found in docstrings in the package code proper. You can run this check manually in your local repository with `poetry run pytest`. Passage of these checks is strictly required.

6.2.5 Code Coverage (test-coverage-percent)

This job checks that the lines of code covered by our Unit Tests checks meet or exceed certain thresholds. Passage of this check is not strictly required but highly encouraged.

6.2.6 Release Notes Check (test-release-notes-check)

Checks that the current branch’s release notes has modifications relative to the merge target’s. Passage of this check is not strictly required but highly encouraged.

6.2.7 Example Notebooks Execution (test-notebooks)

This check executes included example notebooks to ensure their proper functionality with the package with respect to a merge request. Not all notebooks may be run, as some may be set up to use too many resources or run for an extended period of time. Passage of these checks is strictly required.

6.3 Human Review

Once the automatic checks are either resolved or addressed, the submitted MR will need to go through a human review. Reviewers should add comments to provide feedback and raise potential issues on logical and semantic details of the contributed content that would otherwise not be caught by the discrete automatic checks above. Should the MR pass their review, the reviewer should then indicate that it has their approval using the GitLab review interface to flag the MR as Approved.

A review can still be requested before the checks are resolved, but the MR must be marked as a Draft. Once the MR is in a mergeable state, it will need to undergo a final review to ensure that there are no outstanding issues.

If an MR is not a draft and has an appropriate amount of approving reviews, it may be merged at any time.
6.3.1 Notebooks

The default preference is that all Jupyter Notebooks be included in execution of the Notebook CI job (listed under the parallel:matrix section). If a notebook is added in the MR, it should be verified that it has been added to the list of notebooks to be run. If it has not been, the addition should be requested or for a rationale as to why it has not been. Rationale for excluding specific notebooks from the CI job should be added to the relevant section in examples/README.md.

6.4 Resolving a Branch

6.4.1 Merge

Once an MR receives an approving review and is no longer marked as a Draft, the repository maintainers can merge it, closing the merge request. It is recommended that the submitter delete their branch after the MR is merged.

6.4.2 Close

If it is decided that the MR will not be integrated into nrtk, then it can be closed through GitLab.
7.1 Steps of the NRTK Release Process

Three types of releases are expected to occur:

- major
- minor
- patch

See the CONTRIBUTING.md file for information on how to contribute features and patches. See the docs/review_process.rst for information about how to, and what is involved in, reviewing contributions.

The following process should apply when any release that changes the version number occurs.

7.1.1 Create and Merge Version Update Branch

This step of the release process depends on whether the release is considered to be major or minor, or if it is a patch release.

Major and Minor Releases

Major and minor releases may add one or more trivial or non-trivial features and functionalities.

1. Create a new branch off of the main named something like update-to-v{NEW_VERSION}, where NEW_VERSION is the new X.Y version.
   a. Use the scripts/update_release_notes.sh script to update the project version number, create docs/release_notes/v{NEW_VERSION}.rst, and add a new pending release notes stub file.

   ```bash
   $ # When creating a major release
   $ ./scripts/update_release_notes.sh major
   $ # OR when creating a minor release
   $ ./scripts/update_release_notes.sh minor
   ```

   b. Add a descriptive paragraph under the title section of docs/release_notes/v{NEW_VERSION}.rst summarizing this release.

2. Push the created branch to the upstream repository, not your fork (this is an exception to the normal forking workflow).
3. Create a pull/merge request for this branch with `release` as the merge target. This is to ensure that everything passes CI testing before making the release. If there is an issue, then topic branches should be made and merged into this branch until the issue is resolved.

4. Get an approving review.

5. Merge the pull/merge request into the `release` branch.

6. Tag the resulting merge commit. See Tag new version below for how to do this.

7. As a repository administrator, merge the `release` branch into `main` locally and push the updated `main` to upstream. (Replace “upstream” in the example below with your applicable remote name.)

   ```
   $ git fetch --all
   $ git checkout upstream/main
   $ git merge --log --no-ff upstream/release
   $ git push upstream main
   ```

8. Draft a new release on GitLab for the new version.

### Patch Release

A patch release should only contain fixes for bugs or issues with an existing release. No new features or functionality should be introduced in a patch release. As such, patch releases should only ever be based on an existing release point (git tag).

This list assumes we are creating a new patch release off of the latest release version, i.e. off of the `release` branch. If a patch release for an older release version is being created, see the Patching an Older Release section.

1. Create a new branch off of the `release` branch named something like `update-to-v{NEW_VERSION}`, where `NEW_VERSION` is the target X.Y.Z, including the bump in the patch (Z) version component.
   
   a. Use the `scripts/update_release_notes.sh` script to update the project version number, create `docs/release_notes/v{NEW_VERSION}.rst`, and add a new pending release notes stub file.

   ```
   $ ./scripts/update_release_notes.sh patch
   ```

   b. Add a descriptive paragraph under the title section of `docs/release_notes/v{NEW_VERSION}.rst` summarizing this release.

2. Push the created branch to the upstream repository, not your fork (this is an exception to the normal forking workflow).

3. Create a pull/merge request for this branch with `release` as the merge target. This is to ensure that everything passes CI testing before making the release. If there is an issue, then topic branches should be made and merged into this branch until the issue is resolved.

4. Get an approving review.

5. Merge the pull/merge request into the `release` branch.

6. Tag the resulting merge commit. See Tag new version below for how to do this.

7. As a repository administrator, merge the `release` branch into `main` locally and push the updated `main` to upstream. (Replace “upstream” in the example below with your applicable remote name.)

   ```
   $ git fetch --all
   $ git checkout upstream/main
   $ git merge --log --no-ff upstream/release
   $ git push upstream main
   ```
8. *Draft a new release on GitLab* for the new version.

**Patching an Older Release**

When patching a major/minor release that is not the latest version, a branch needs to be created based on the release version being patched to integrate the specific patches into. This branch should be prefixed with `release-` to denote that it is a release integration branch, e.g. `release-v1.2.19` (where 19 is the incremented patch version number). Patch topic-branches should be based on this `release-...` branch. When all fix branches have been integrated, follow the *Patch Release* section above, replacing `release` branch references (merge target) to be the `release-...` integration branch. Step 6 should be to merge this release integration branch into `release` first, and then `release` into `main`, if applicable (some exceptional patches may only make sense for specific versions and don’t warrant integration into upstream `main`).

### 7.1.2 Tag New Version

Release branches are tagged in order to record where in the git tree a particular release refers to. All release tags should be in the history of the `release` and `main` branches (barring exceptional circumstances).

We prefer to use local `git tag` commands to create the release version tag, pushing the tag to upstream. The version tag should be applied to the merge commit resulting from the above described `update-to-v{NEW_VERSION}` topic-branch (“the release”).

See the example commands below, replacing `HASH` with the appropriate git commit hash, and `UPSTREAM` with the appropriate remote name. We also show how to use Poetry’s `version` command to consistently access the current, just-updated package version.

```
$ git checkout HASH
$ VERSION="v$(poetry version -s)"
$ git tag -a "$VERSION" -F docs/release_notes/"$VERSION".rst
$ git push UPSTREAM "$VERSION"
```

### 7.1.3 Draft a New Release on GitLab

After creating and pushing a new version tag, a GitLab “release” should be made.

a. Navigate to the GitLab Releases page for the nrtk repository.

b. Click the “Create a new release” button (or go here).

c. Select from the “Tag name” dropdown the tag version just created and pushed

d. Enter the version number as the title, e.g. “v1.2.3”.

e. Select the release date.

f. Copy and paste the release notes for this version into the release notes field.

g. Click the “Create Release” button to create the GitLab release!

In the future, this may be automated. See the appropriate GitLab documentation for more details.
7.2 Release Notes

7.2.1 Pending Release Notes

Updates / New Features

Fixes

7.2.2 v0.1.0

Initial release of the Natural Robustness Toolkit originally ported from prior work.

Updates / New Features

CI/CD
- Added code unit-test and coverage reporting.
- Added code coverage minimum coverage soft check (allows failure).

Documentation
- Added baseline interface and implementation documentation sections.
- Updated and added relevant documents detailing the components of our Open Source Strategy.
- Added ability to render documentation on GitLab Pages.

Examples
- Updated and added relevant documents detailing the components of our Open Source Strategy.

Examples
- Added an example notebook demonstrating NRTK perturber functionality.
- Added an example notebook demonstrating NRTK generic generator functionality.
- Added an example notebook demonstrating NRTK pybsm generator functionality.
- Added an example notebook demonstrating NRTK COCOScorer functionality.

Interfaces
- Added a `GenerateObjectDetectorBlackboxResponse` interface for generating response curves with given perturber factories, detector, and scorer.
- Added a `PerturbImage` interface for taking an image stimulus and generating a perturbed image.
- Added a `PerturbImageFactory` interface for generating `PerturbImage` instances of specified type and configuration while varying one parameter.
- Added a `ScoreDetection` interface that takes in ground-truth and predicted BBox-label pairs and generates scores based on a given metric.

Implementations
- Added an example `GenerateObjectDetectorBlackboxResponse` implementation, `SimpleGenericGenerator` which takes takes input data directly as Sequences.
- Added several `PerturbImage` implementations:
  - From opencv:
* AverageBlurPerturber: Applies average blurring to the given image stimulus.
* GaussianBlurPerturber: Applies Gaussian blurring to the given image stimulus.
* MedianBlurPerturber: Applies median blurring to the given image stimulus.

– From skimage:
  * SaltNoisePerturber, PepperNoisePerturber, SaltAndPepperNoisePerturber: Adds salt and/or pepper noise to given image stimulus.
  * GaussianNoisePerturber: Adds Gaussian-distributed additive noise to given image stimulus.
  * SpeckleNoisePerturber: Adds multiplicative (Gaussian) noise to given image stimulus.

– From PIL:
  * BrightnessPerturber: Adjusts given image stimulus brightness.
  * ColorPerturber: Adjusts given image stimulus color balance.
  * ContrastPerturber: Adjust given image stimulus contrast.
  * SharpnessPerturber: Adjust given image stimulus sharpness.

– NOPPerturber: Serves as a pass-through NOP perturber to test interface functionality.

– From xaitk-irt:
  * PybsmSensor: Holds sensor parameters for pyBSM perturbation.
  * PybsmScenario: Holds target and environment parameters for pyBSM perturbation.
  * PybsmPerturber: Matches PerturbImage interface and is used by pybsm factories.
  * _PybsmPerturbImageFactory: Base PyBSM perturber factory.
  * CustomPybsmPerturbImageFactory: Original “custom” pybsm perturber factory.

- Added a PerturbImageFactory implementation, StepPerturbImageFactory, which is a simple implementation that varies a chosen parameter from start to stop by the given step value.
- Added ScoreDetections implementations
  – NOPScorer: Serves as a pass-through NOP scorer to test interface functionality.
  – RandomScorer: Generates random score values and serves as a test for reproducibility.
  – COCOScorer: Generates detection scores for a specific statistic index using the converted COCO format data.
  – ClassAgnosticPixelwiseIoUScorer: Generates pixelwise IoU scores in a class agnostic way.

**Fixes**

- Changed the default git client poetry uses from dulwich to system-installed git.
7.2.3 v0.2.0

This minor release adds interfaces related to classification operations, additions to the documentation, and updates to abstract and pinned dependencies.

Updates / New Features

Documentation
- Added an Introduction section to provide background and conceptual information about nrtk.
- Added an inheritance diagram which visualizes the layout of nrtk interfaces and implementations and included in interface documentation page.
- Added Getting Started page.
- Added Read the Docs configuration files
- Added a style sheet to guide future documentation and text updates.

Interfaces
- Added a GenerateClassifierBlackboxResponse interface for generating response curves with given per-turber factories, classifier, and scorer.
- Added a ScoreClassification interface that takes in ground-truth and predicted classifications and generates scores based on a given metric.

Package
- Updated pybsm dependency to no longer refer to a specific repository and defer to the configured index (PYPI).
- Added license metadata to reference Apache 2.0.
- Updated locked versions for a couple packages that had their previously-locked versions yanked from PYPI.

Security
- Upgraded Pillow>=10.0

7.2.4 v0.2.1

This patch release adds a valid email address to be attributed to the author of the package.

Fixes

Package
- Fixed author email address to be a valid address (nrtk@kitware.com).
7.2.5 v0.3.0

This minor release updates the main opencv dependency of the package to opencv-python and adds a conditional opencv-python-headless installation job to CI.

Updates / New Features

Documentation

- Updated README.md under examples.

CI/CD

- Added a conditional case of opencv-python-headless installation.

Package

- Updated main opencv dependency from opencv-python-headless to opencv-python.

7.2.6 v0.3.1

A couple bug fixes, a CI optimization, and updated text.

Updates / New Features

- Optimized opencv-python-headless usage in .gitlab-ci.yml.
- Text updated to properly reflect style guide.

Fixes

- Capped pybsm to <0.2.0 to prevent incompatible package breaking changes.
- Added py.typed to the source directory.

7.2.7 v0.3.2

Addresses numerous vulnerabilities and adds automated SAST scanning to CI/CD.

Updates / New Features

CI/CD

- Added SAST scanning to CI/CD.

Dependency Vulnerabilities

- Upgraded Pillow to Pillow>=10.2.0
- Replaced jupyter and ipython with notebook>=7.0.7, which also future proofed jupyter notebook to current package name.
Fixes

- Replaced asserts with appropriate exceptions.

7.2.8 v0.4.0

Updated package code, dependency, and tests to use `pybsm>=0.2.0`.

Updates / New Features

- Updated `pybsm` to `0.2.0`.

Fixes

7.2.9 v0.4.1

Created a few fixes required for downstream `nrtk-cdao` entrypoint.

Updates / New Features

Fixes

- Corrected configuration semantics for pyBSM implementations.
- Exposed all plugins for discoverability.

7.2.10 v0.4.2

Added support for `Python 3.12`.

Updates / New Features

Python Version Support

- Added support for `py3.12`

CI/CD

- Updated CI test matrix to support `py3.12`

Fixes

7.2.11 v0.5.0

Updated to use newer version of `pybsm` which included an improvement to Fourier sampling.
Updates / New Features

- Added Sphinx document rendering for MRs. The docs pages can be accessed by clicking the “View App” button located in the merge request page under the test pipeline section.
- Updated to use pybsm’s new Fourier sampling method

Fixes

- Fix incorrect version range for numpy.

7.2.12 v0.5.1

Hotfix to address a numpy dependency resolution issue.

Updates / New Features

Fixes

- Changed supported Python version from ^3.8.1 to >=3.8.1,<4.0.0 to address a numpy dependency conflict.

7.2.13 v0.5.2

A hotfix to correct a numpy dependency resolution issue.

Updates / New Features

- Updated to use pybsm>=0.4.1.

Fixes

- Fixed numpy dependency versions for downstream resolution.

7.2.14 v0.5.3

Attempting to patch downstream dependency resolution issues.

Updates / New Features

- Updated to use pybsm>=0.4 to support wider range of dependencies.
- Lowered the minimum numpy allowed to 1.22.
Fixes

7.2.15 v0.6.0

Added a new pyBSM perturber to nrtk from pyBSM’s JitterOTF.

Updates / New Features

Added implementation of JitterOTF perturber

Fixes

- Added missing docstring for intTime
- Added numpy hinge for python 3.12

7.2.16 v0.7.0

Added support for JitterOTFPerturber Plugfigurability rehydration and dehydration.

Updates / New Features

- Removed perturber interface restriction which required that input image dimensions be maintained. Note perturbers which modify image dimensions (including rotations) should be used with caution as scoring can be impacted if groundtruth isn’t similarly transformed.
- Updated git lfs to properly track large files in any directory
- Added functionality to dehydrate and rehydrate JitterOTFPerturber objects with config files

Fixes
MISCELLANEOUS

8.1 Creating a Public Release Request

8.1.1 Shared Storage

This Google Drive folder is used to store submitted and approved public release requests.

- The Submitted folder is used to store public release request packages that are or are intended to be submitted, and not approved yet.
- The Approved folder is used to store public release request packages that have been approved. These folders should have a copy-of/reference-to the approving document.

Folders underneath Submitted and Approved folders generally take the form of the request submission date in YYYY-MM-DD format, e.g. 2022-12-02.

8.1.2 Submission Components

The following are the critical components to a public release submission that we should place into an applicable sub-folder under Submitted:

- Summary Document
- Code Package

Summary Document

Create a google doc under the appropriate “Submitted” sub-folder and title it “Summary”. This should contain a summarizing paragraph or two, and then a list of changes since the last public release. Basically the change notes since the last public release request submission. Draw from the release notes in the docs/release_notes/ files.

The summary document to be sent is usually a Word document (.docx)

Example document.
**Code Package**

This should be a git archive of the codebase at the point which we want to get released.

In the repository, checkout the branch that is to be submitted for release approval. Ideally this is the current `main/master` branch.

Create a tag to mark where in the repository the submission, and future approval, applies to. This is important as we will need to know where this is to know what is appropriate to expose when the request is approved. The tag should be of the form `public-release-request-YYYYMMDD`, obviously replacing the `YYYYMMDD` with the request submission date. This date should additionally match the dates mentioned above for the submission request folder to just keep everything sane and matching.

Create a git archive while on this tag with. Consider the following example, setting the `YYYYMMDD` with the appropriate value to match tag to archive:

```bash
$ YYYYMMDD="20221202"
$ git archive --format tar.gz -o "nrtk-$YYYYMMDD-$(git rev-parse HEAD).tar.gz" public-release-request-$YYYYMMDD
```

### 8.2 Style Sheet

This document provides writing guidance and lists the appropriate spelling of words and phrases to use within this repository’s documentation and examples for consistency purposes.

- **active vs. passive voice** - use active voice whenever possible and appropriate (e.g. “Use the Submitted folder to store requests” *not* “The Submitted folder is used to store requests”)

- **black box** (noun) or **black-box** (adj); *not* blackbox

- **GitLab** - use CamelCase; *not* Gitlab

- **ground truth** (noun) or **ground-truth** (adj); *not* groundtruth

- Headings - use Title Capitalization and succinct descriptions (e.g. “Example Notebook for the XYZ Implementation” *not* “Example Notebook to demonstrate the usage of the XYZ implementation”) and follow with an intro sentence, at least between headings (i.e. don’t stack headers)

- Instructional language - should be direct, imperative, active, and straightforward (e.g. “To use the package, install the files in your Python environment” *not* “In order to use the package, files could be installed in your Python environment”)

- **nrtk/NRTK** - lowercase when referring to the nrtk package or repository; uppercase when referring to the NRTK platform at a higher level

- **open source** (noun/adj) - no hyphen

- **please** - do not use

- Release notes - use sentences with end punctuation

- **set up** (verb) or **setup** (noun/adj); *not* set-up

- **TOC** include a table of contents (TOC) for each Jupyter notebook

- **use case** (noun) - no hyphen
CHAPTER
NINE

INDICES AND TABLES

• genindex
• modindex
• search
INDEX

Symbols
__call__() (nrtk.impls.perturb_image.pybsm.perturber.PybsmPerturber method), 22
__call__() (nrtk.interfaces.perturb_image_factory.PerturbImageFactory method), 16
__call__() (nrtk.interfaces.gen_object_detector_blackbox_response.GenerateObjectDetectorBlackboxResponse method), 18
__call__() (nrtk.interfaces.perturb_image.PerturbImage method), 15
__call__() (nrtk.interfaces.score_detections.ScoreDetections method), 17
__getitem__() (nrtk.impls.gen_object_detector_blackbox_response.simple_generic_generator.SimpleGenericGenerator method), 35
__getitem__() (nrtk.impls.gen_object_detector_blackbox_response.simple_pybsm_generator.SimplePybsmGenerator method), 36
__getitem__() (nrtk.interfaces.gen_object_detector_blackbox_response.GenerateObjectDetectorBlackboxResponse method), 18
__getitem__() (nrtk.interfaces.perturb_image_factory.PerturbImageFactory method), 16
__init__() (nrtk.impls.gen_object_detector_blackbox_response.simple_generic_generator.SimpleGenericGenerator method), 36
__init__() (nrtk.impls.gen_object_detector_blackbox_response.simple_pybsm_generator.SimplePybsmGenerator method), 37
__init__() (nrtk.impls.perturb_image.generic.PIL.enhance.SharpnessPerturber method), 31
__init__() (nrtk.impls.perturb_image.generic.cv2.blur.AverageBlurPerturber method), 19
__init__() (nrtk.impls.perturb_image.generic.cv2.blur.GaussianBlurPerturber method), 20
__init__() (nrtk.impls.perturb_image.generic.cv2.blur.MedianBlurPerturber method), 21
__init__() (nrtk.impls.perturb_image.pybsm.perturber.PybsmPerturber method), 22
__init__() (nrtk.impls.perturb_image_factory.generic.step.StepPerturbImageFactory method), 32
__init__() (nrtk.impls.perturb_image_factory.pybsm.CustomPybsmPerturbImageFactory method), 32
__init__() (nrtk.interfaces.perturb_image_factory.PerturbImageFactory method), 16
__iter__() (nrtk.interfaces.perturb_image_factory.PerturbImageFactory method), 16
__len__() (nrtk.impls.gen_object_detector_blackbox_response.simple_pybsm_generator.SimplePybsmGenerator method), 37
__len__() (nrtk.impls.perturb_image_factory.PerturbImageFactory method), 16
__next__() (nrtk.interfaces.perturb_image_factory.PerturbImageFactory method), 16
__repr__() (nrtk.impls.perturb_image.pybsm.perturber.PybsmPerturber method), 23
__str__() (nrtk.impls.perturb_image.pybsm.perturber.PybsmPerturber method), 23

A
AverageBlurPerturber (class in nrtk.impls.perturb_image.generic.cv2.blur), 19
B
BrightnessPerturber (class in nrtk.impls.perturb_image.generic.PIL.enhance), 19
C
ClassAgnosticPixelwiseIoUScorer (class in nrtk.impls.score_detections.class_agnostic_pixelwise_iou_scorer), 33
COCOScorer (class in nrtk.impls.score_detections.coco_scorer), 34
ColorPerturber (class in nrtk.impls.perturb_image.generic.PIL.enhance), 20
ContrastPerturber (class in nrtk.impls.perturb_image.generic.PIL.enhance), 20

E
from_config() (nrtk.impls.perturb_image.pybsm.perturber.PybsmPerturber class method), 23
from_config() (nrtk.impls.perturb_image.pybsm.sensor.PybsmSensor class method), 28

F
__iter__() (nrtk.interfaces.perturb_image_factory.PerturbImageFactory method), 16

G
__call__() (nrtk.interfaces.perturb_image_factory.PerturbImageFactory method), 16

L
__len__() (nrtk.impls.gen_object_detector_blackbox_response.simple_pybsm_generator.SimplePybsmGenerator method), 37

57
RandomScorer (class in nrtk.impls.score_detections.random_scorer), 35

SaltAndPepperNoisePerturber (class in nrtk.impls.perturb_image.generic.skimage.random_noise), 30

SaltNoisePerturber (class in nrtk.impls.perturb_image.generic.skimage.random_noise), 31

cscore() (nrtk.impls.score_detections.class_agnostic_pixelwise_iou_scorer.ClassAgnosticPixelwiseIoUScorer method), 33
cscore() (nrtk.impls.score_detections.coco_scorer.COCOScorer method), 34
cscore() (nrtk.impls.score_detections.nop_scorer.NOPScorer method), 34
cscore() (nrtk.impls.score_detections.random_scorer.RandomScorer method), 35
cscore() (nrtk.interfaces.score_detections.ScoreDetections method), 17

ScoreDetections (class in nrtk.interfaces.score_detections), 17

SharpnessPerturber (class in nrtk.impls.perturb_image.generic.PIL.enhance), 31

SimpleGenericGenerator (class in nrtk.impls.gen_object_detector_blackbox_response.simple_generic_generator), 35

SimplePybsmGenerator (class in nrtk.impls.gen_object_detector_blackbox_response.simple_pybsm_generator), 36

SpeckleNoisePerturber (class in nrtk.impls.perturb_image.generic.skimage.random_noise), 31

StepPerturbImageFactory (class in nrtk.impls.perturb_image_factory.generic.step), 32

theta_key (nrtk.impls.perturb_image_factory.generic.step.StepPerturbImageFactory property), 33

theta_key (nrtk.interfaces.perturb_image_factory.PerturbImageFactory property), 17

thetas (nrtk.impls.perturb_image_factory.generic.step.StepPerturbImageFactory property), 33

thetas (nrtk.interfaces.perturb_image_factory.PerturbImageFactory property), 17