xaitk-saliency

Release 0.7.0

Kitware, Inc.

Jun 19, 2023

CONTENTS:

CHAPTER

ONE

INTRODUCTION

The xaitk-saliency package implements a class of XAI algorithms known as *saliency algorithms*. A basic machine learning application pipeline is shown in Figure 1:

Fig. 1: Figure 1: A basic AI pipeline.

In this scenario, an AI algorithm operates on an input (text, image, etc.) to produce some sort of output (classification, detection, etc.). Saliency algorithms build on this to produce visual explanations in the form of saliency maps as shown in Figure 2:

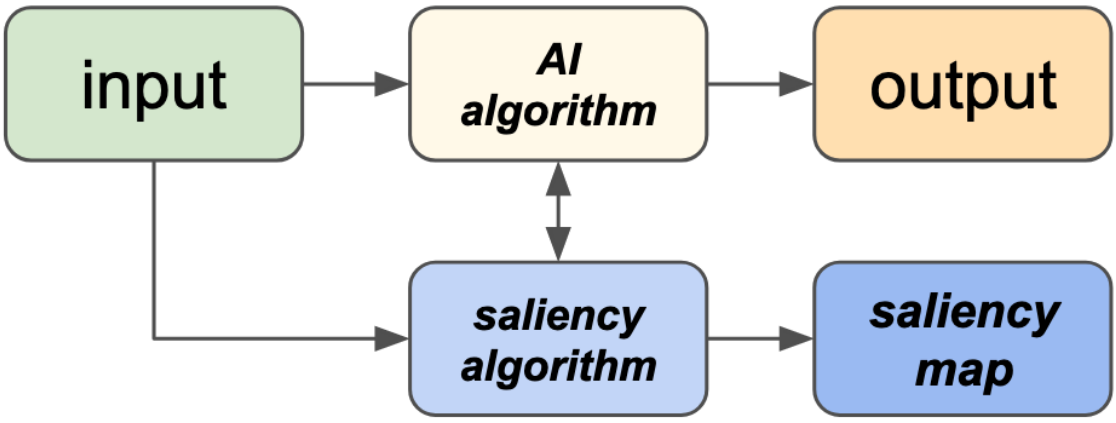

Fig. 2: Figure 2: The AI pipeline augmented with a saliency algorithm.

At a high level, saliency maps are typically colored heatmaps applied to the input, highlighting regions that are somehow significant to the AI. Figure 3 shows sample saliency maps for text and images.

Fig. 3: Figure 3: Sample saliency maps for text (left, from Tuckey et al.) and images (right, from Dong et al.).

Note: The xaitk-saliency toolkit currently focuses on providing saliency maps for images.

1.1 Image Saliency Maps: An Intuitive Introduction

Figure 4 shows a deep learning pipeline for recognizing objects in images; pixels in the image (in green) are processed by the Convolutional Neural Network (CNN, in yellow) to produce the output (in orange). Here, the system has been trained to recognize 1000 object categories. Its output is a list of 1000 numbers, one for each object type; each number is between 0 and 1, representing the system's estimate of whether or not that particular object is in the image.

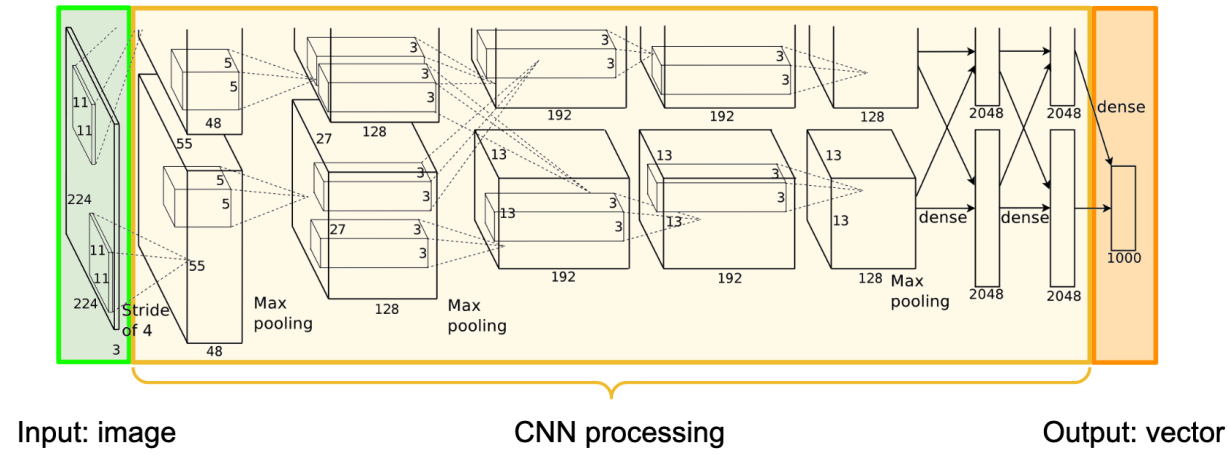

Fig. 4: Figure 4: Typical object recognition CNN architecture; the image (green, left) is processed left to right through the CNN (yellow) to produce an output vector (orange, right). Diagram from Krizhevsky et al.

This operation is "all or nothing" at both ends: the entire image must be processed, and the entire vector must be output. There is no mechanism by which a subset of the output can be traced back to a subset of the input. Yet it seems reasonable to ask questions such as:

- The input image is a camel; why did the system give a higher likelihood to "horse" than "camel"?

- The input image is a kitchen, but the system gave a high likelihood to "beach ball". What parts of the image were responsible for this?
- The input image contains two dogs, and the system gave a high likelihood for "dog". How would this change if one of the dogs wasn't in the image?
- The input image contains a dog and a cat, and the system gave a high likelihood for "cat" but not "dog". How will the system respond if the cat is removed?

At some level, these questions require a degree of *introspection*; the system must produce not only the output, but also some information about **how** the output was produced.

To avoid confusion, we need some definitions:

- The **AI algorithm, or AI,** is the algorithm whose output we are trying to explain. An AI operates according to its **AI model**; contemporary AIs are built around CNNs such as in Figure 4, examples of other models include decision trees and support vector machines.
- **Explainable AI algorithms, or XAI**, is what we attach to the AI system to generate explanations (as in Figure 2). The XAI may itself use CNNs, but these details are typically hidden from the XAI user.

To answer questions such as the ones raised above, the XAI must have some way of interacting with the AI. There are two popular approaches to this:

1) The **white-box** approach: the AI system is altered to open or expose its model. The XAI examines the state of the AI model as the AI generates its output, and uses this information to create the explanation.
2) The **black-box** approach: the AI's model is not exposed; instead, the XAI probes the AI by creating an *additional set of input images* which perturb or change the original input in some way. By comparing the original output to that for the related images, we can deduce certain aspects of the how the AI and its model behaves.

These are illustrated in Figure 5. Note how the AI algorithm is not available for inspection in the black-box model.

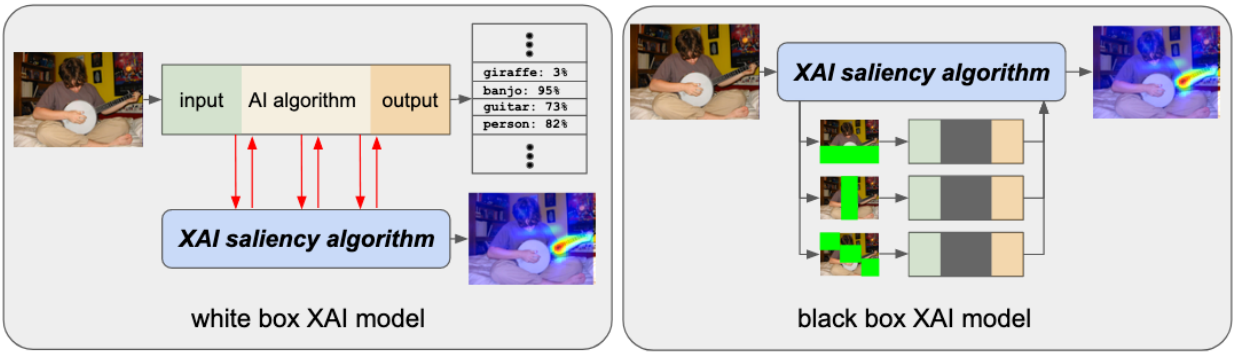

Fig. 5: Figure 5: White-box vs. black-box approaches to XAI. The white-box approach has access to the AI model; the black-box approach does not and instead repeatedly probes the AI with variations on the input image.

Note: The white-box vs. black-box distinction refers to *using the AI model after it has been created*; nothing is implied about how the model is constructed.

Let's take a look at the pros and cons of these two approaches.

1.1.1 White-Box Methods

The **white-box** approach to XAI (Figure 5, left) exposes some (or all) of the internal state of the AI model; the explanation draws a connection between this exposed state and the AI's output. Some AI methods are intrinsically introspective to the point where they are not so much "white-box" as transparent:

- In linear regression, the output is a weighted sum of input features, making it easy to separate out effects just by looking at each of the learned-feature weights.
- In a decision tree, the output is directly computed by making the comparisons through a set of nodes and branches that are part of the AI's model.

An example of a white-box explanation method for CNNs is Grad-CAM (paper, code), which exposes some (but not all) of the CNN layers to the explanation generation algorithm.

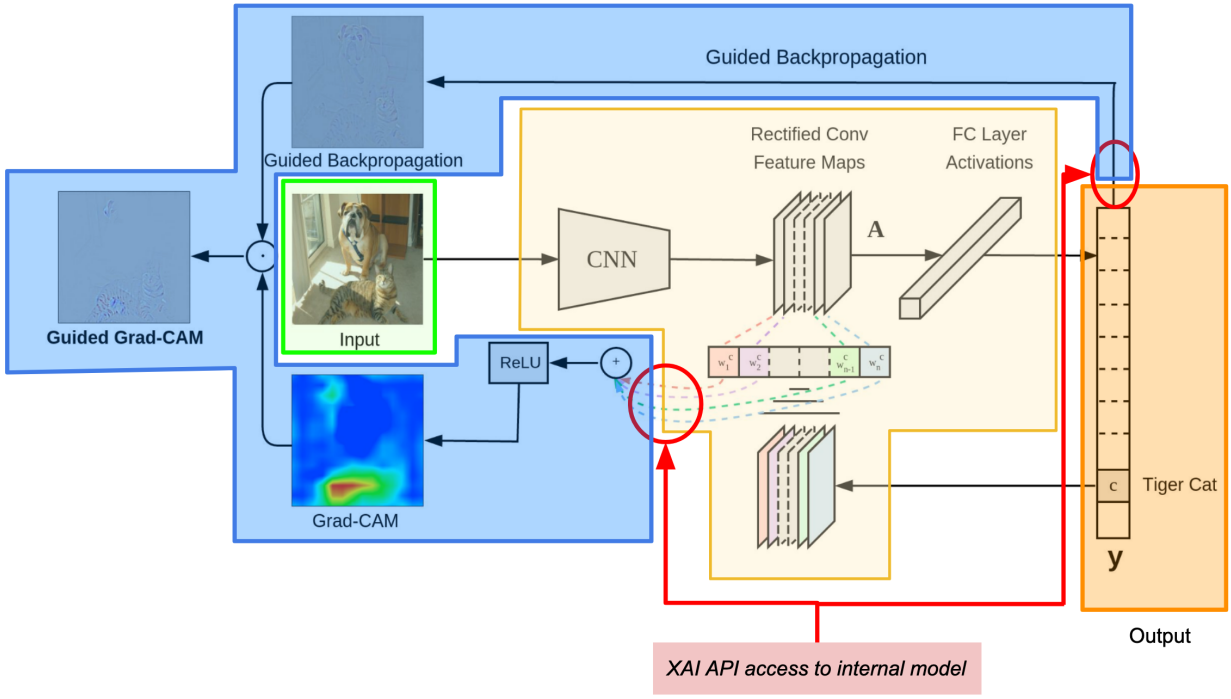

Fig. 6: Figure 6: Grad-CAM architecture. Functional regions annotated to align with Figure 4: input in green, CNN architecture in yellow, output in orange. The explanation functionality is in blue; the APIs granting access to the model are circled in red. Figure from the Grad-CAM code repo.

In Grad-CAM, first the model computes its output per the typical processing flow: input (green), through the CNN (yellow), to the output (orange.) The explanation is created via an additional processing step that uses the output and feature maps from within the CNN (made available to the XAI through the red circles) to measure and visualize the activation of those regions associated with the output (orange).

Two aspects typical of white-box methods are demonstrated here:

- **The explanation could not have been created from the output alone**. In order to operate, the explanation algorithm (blue) required access to both the output *and* the CNN internal state.
- **The XAI implementation is tightly coupled to the AI's CNN model architecture exposed by the API**. Although the *method* may be general, any particular *implementation* will expect the AI's CNN architecture to conform to the specifics of the API.

In general, pros and cons of white-box approaches are:

Pros

- A white-box XAI can choose to **leverage its tight coupling to the AI model** to maximize the information available, at the sacrifice of generalization to other AI models.
- A white-box XAI **accesses the actual AI model's computation which generated the output**. The explanation is derived directly from what the AI model computed about the input, in contrast to black-box XAIs which can only indirectly compare the output to output from slightly different inputs.
- A white-box XAI is usually more computationally efficient, since it typically only requires a single forward / backward pass through the AI model. In Figure 5, the white-box approach on the left interacts with the AI during its single processing run to produce the output; in comparison, black-box methods (such as in Figure 5 on the right) typically run the AI network multiple times.

Cons

- The flip side of tighter XAI integration to a specific AI model or class of models is **loss of generality**. An explanation technique that works for one model can be difficult to port to other AI models. Lack of generality can also make it harder to evaluate explanation algorithms across AI models.
- It may be necessary to **modify the AI model implementation** to gain access to the internal state. Depending on the environment in which the AI was developed and delivered, this problem may be trivial or insurmountable.
- Similarly, the white-box XAI may **require updating as the AI model evolves**. Tight coupling introduces a dependency which must be managed, possibly increasing development costs.

1.1.2 Black-Box Methods

One way to frame the AI pipeline in Figure 1 is that we're asking the AI a question (the input), and it gives us an answer (the output). In this setting, a white-box XAI uses its special access to the AI model to observe details of how the AI answers the question. In contrast, a **black-box** XAI (Figure 5, right) does not see any details of how the AI answers a single question; rather, **it asks the AI a series of different questions related to the original input** and bases its explanation on how these answers differ from the original answer.

This technique relies on two assumptions:

1) We have some way to generate these "related questions" based on the original input whose output we're trying to explain.
2) The AI algorithm's responses to these additional questions will somehow "add up" to an explanation for the original output.

The xaitk-saliency package deals with image-based AI; black-box XAI for images typically generate the "related questions" by **image perturbation** techniques. These repeatedly change or partially obscure the input image to create new images to run through the AI, which in turn generates the "related answers" the XAI uses to form its explanation.

Figure 7 shows the architecture for one black-box XAI algorithm, RISE (Randomized Input Sampling for Explanation). When applied to an image classification AI algorithm, RISE generates an "importance map" indicating which regions of the input are most associated with high confidence for a particular label. This is done by creating copies of the input with areas randomly obscured (shown in the red box in Figure 7). Each of these is fed through the AI; by comparing how the outputs change, RISE develops a correlation between image areas and label confidences.

Two aspects typical of black-box methods are demonstrated here:

- **The explanation does not require access to the inner workings of the AI**. RISE is black box because it only uses the AI's standard input and output pathways.

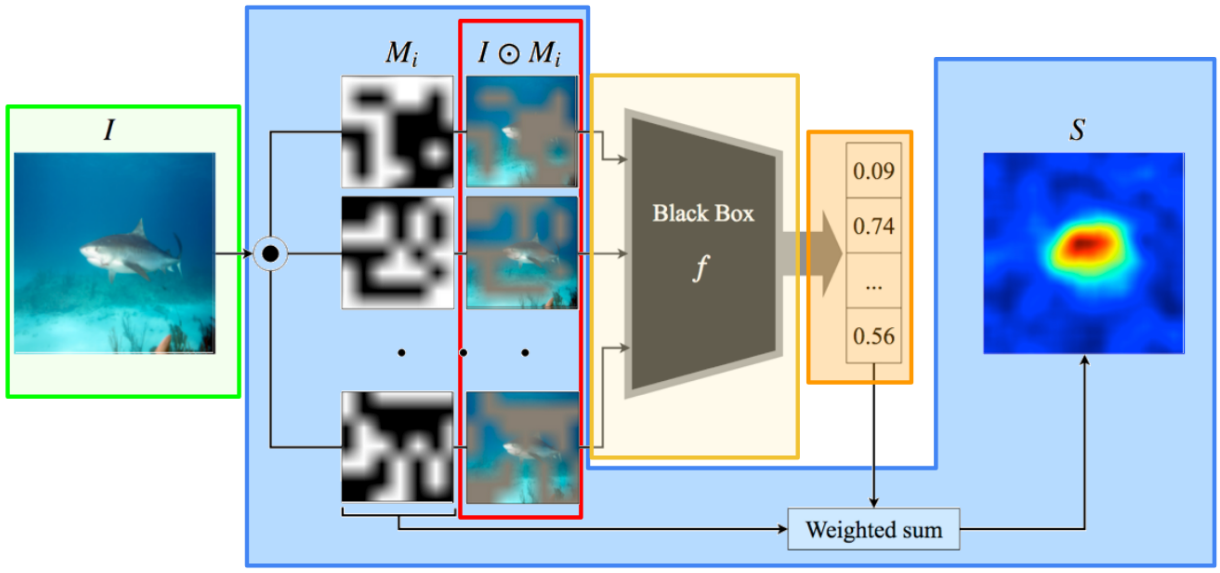

Fig. 7: Figure 7: RISE architecture. Functional regions annotated to align with Figure 4: input in green, AI in yellow, output in orange. Note that the operation of the AI is not exposed to the XAI. The XAI (in blue) creates the set of related inputs (red box) by randomly obscuring areas of the input. Figure from Petsiuk, Das, and Saenko.

- **The AI must be run many times on different inputs to generate the explanation**. In the experiments described in their paper, the RISE team used up to 8000 masked versions of a single input image to generate an explanation.

In general, pros and cons of black-box approaches are:

Pros

- A black-box XAI **does not depend on the AI method, only the inputs and outputs**. (It is said to be *model-agnostic*.) In Figure 7, the AI (in yellow) can be anything: a CNN, a decision tree, or random number generator. This independence is the primary appeal of black-box methods, and has several implications:
  - A single black-box XAI can, in theory, **operate across any number of AI implementations.** As long as the AI provides input and output as in Figure 1, it can be used with a black-box XAI.
  - Black-box XAIs are **loosely coupled** to the AIs they explain. As long as the basic I/O pathways are unchanged, the AI has more freedom to evolve at a different pace from the XAI.
  - The black-box approach **enables XAI when the AI must not be exposed**, due to security concerns, contractual agreements, etc.

Cons

- Black-box XAI approaches **require extra work** to generate and process the related inputs. As a result, they are generally slower and more resource intensive than white-box approaches.
- A black-box XAI can only **indirectly observe how the AI processes the original input**. A white-box XAI's explanation directly uses how the AI responds to the input, but for any one input, a black-box XAI can never know anything beyond the output. Processing an array of related inputs provides indirect / differential insight into the AI's *behavior*, but a black-box XAI cannot relate this behavior to anything inside the AI.

1.2 Saliency Algorithms

The xaitk-saliency package currently provides several black-box XAI algorithms. These algorithms follow a general pattern that consists of two sequential steps: **image perturbation** followed by **heatmap generation**. **Image perturbation** involves generating perturbed versions of the input image by applying a set of perturbation masks. **Heatmap generation** involves generating saliency heatmaps based on how the black-box model outputs change as a result of image perturbation. This technical design choice allows for modularization of the image perturbation and heatmap generation components of the algorithm. By formulating the algorithms in this manner, the exact operation of the black-box model is not needed by an algorithm, which is concerned only with the inputs and outputs. Additionally, the algorithm can be flexibly determined by the user; that is, the user is free to choose and configure the algorithm as needed for the problem domain.

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

Table 1: Saliency Algorithms by Task

Task	Saliency Algorithm(s)
Image classification	*Occlusion-based Saliency* [1]; *Randomized Input Sampling for Explanation (RISE)* [2]
Image similarity	*Similarity Based Saliency Maps (SBSM)* [3]
Object detection	*Detector-RISE (D-RISE)* [4]
Reinforcement learning	*Perturbation-based Saliency* [5]

1. Zeiler MD, Fergus R. Visualizing and understanding convolutional networks (2013). arXiv preprint arXiv:1311.2901. 2013.
2. Petsiuk V, Das A, Saenko K. Rise: Randomized input sampling for explanation of black-box models. arXiv preprint arXiv:1806.07421. 2018 Jun 19.
3. Dong B, Collins R, Hoogs A. Explainability for Content-Based Image Retrieval. In CVPR Workshops 2019 Jun (pp. 95-98).
4. Petsiuk V, Jain R, Manjunatha V, Morariu VI, Mehra A, Ordonez V, Saenko K. Black-box explanation of object detectors via saliency maps. arXiv preprint arXiv:2006.03204. 2020 Jun 5.
5. Greydanus S, Koul A, Dodge J, Fern A. Visualizing and understanding atari agents. In International conference on machine learning 2018 Jul 3 (pp. 1792-1801). PMLR.

CHAPTER

TWO

INSTALLATION

There are two ways to obtain the xaitk-saliency package. The simplest is to install via the **pip** command. Alternatively, the source tree can be acquired and be locally developed using Poetry (installation and usage).

2.1 From `pip`

```
$ pip install xaitk-saliency
```

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://github.com/XAITK/xaitk-saliency.git ./
$ 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 xaitk-saliency 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 `xaitk_saliency/`, 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.

CHAPTER

THREE

INTERFACES

The xaitk-saliency API consists of a number of object-oriented functor interfaces for saliency heatmap generation. These initial interfaces focus on black-box visual saliency. We define the two high-level requirements for this initial task: reference image perturbation in preparation for black-box testing, and saliency heatmap generation utilizing black-box inputs. We define a few similar interfaces for performing the saliency heatmap generation, separated by the intermediate algorithmic use cases: image similarity, classification, and object detection. 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, that some interfaces are defined for certain black-box concepts as part of the SMQTK ecosystem (e.g. in SMQTK-Descriptors, SMQTK-Classifier, SMQTK-Relevancy, 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.

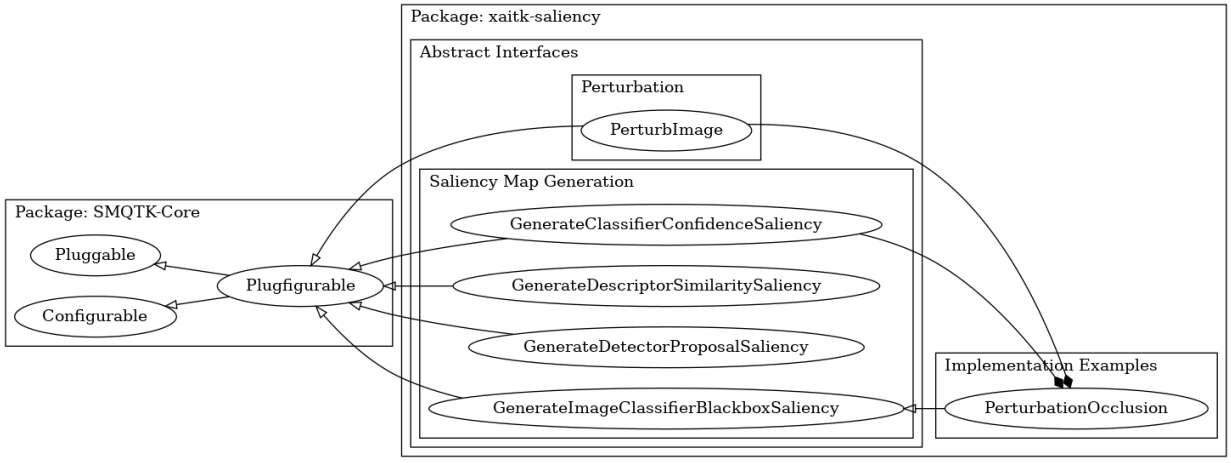

Fig. 1: Figure 1: Abstract Interface Inheritance.

3.1 Image Perturbation

The PerturbImage interface abstracts the behavior of taking a reference image and generating some number of perturbations of the image along with paired mask matrices that indicate where perturbations have occurred and to what amount.

Implementations should impart no side effects on the input image.

Immediate candidates for implementation of this interface are occlusion-based saliency algorithms [3] that perform perturbations on image pixels.

3.1.1 Interface: PerturbImage

class xaitk_saliency.interfaces.perturb_image.**PerturbImage**

Interface abstracting the behavior of taking a reference image and generating some number perturbations in the form of mask matrices indicating where perturbations should occur and to what amount.

Implementations should impart no side effects upon the input image.

abstract perturb(*ref_image: ndarray*) → ndarray

Transform an input reference image into a number of mask matrices indicating the perturbed regions.

Output mask matrix should be three-dimensional with the format [nMasks x Height x Width], sharing the same height and width to the input reference image. The implementing algorithm may determine the quantity of output masks per input image. These masks should indicate the regions in the corresponding perturbed image that have been modified. Values should be in the [0, 1] range, where a value closer to 1.0 indicates areas of the image that are unperturbed. Note that output mask matrices may be of a floating-point type to allow for fractional perturbation.

Parameters
ref_image – Reference image to generate perturbations from.

Returns
Mask matrix with shape [nMasks x Height x Width].

3.1.2 Image Occlusion via Perturbation Masks

A common intermediate step in this process is applying the generated perturbation masks to imagery to produce occluded images. We provide two utility functions as baseline implementation to perform this step:

- `xaitk_saliency.utils.masking.occlude_image_batch` - performs the transformation as a batch operation
- `xaitk_saliency.utils.masking.occlude_image_streaming` - performs the transformation in a streaming method with optional parallelization in that streaming

While the batch version is simpler and in many cases the faster of the two versions, the streaming version may be more applicable to large image masks or when a great deal of masks are being input, where in such cases the batch version would exceed available memory.

xaitk_saliency.utils.masking.**occlude_image_batch**(*ref_image: ndarray*, *masks: ndarray*, *fill: int | Sequence[int] | ndarray | None = None*, *threads: int | None = None*) → ndarray

Apply a number of input occlusion masks to the given reference image, producing a list of images equivalent in length, and parallel in order, to the input masks. This batch version will compute all occluded images and returns them all in one large matrix.

We expect the "mask" matrices and the image to be the same height and width, and for the mask matrix values to be in the [0, 1] range. In the mask matrix, values closer to 1 correspond to regions of the image that should *NOT* be occluded. E.g. a 0 in the mask will translate to *fully* occluding the corresponding location in the source image.

We optionally take in a "fill" that alpha-blend into masked regions of the input *ref_image*. *fill* may be either a scalar, sequence of scalars, or another image matrix congruent in shape to the *ref_image*. When *fill* is a scalar or a sequence of scalars, the scalars should be in the same data-type and value range as the input image. A sequence of scalars should be the same length as there are channels in the *ref_image*. When *fill* is an image matrix it should follow the format of *[H x W]* or *[H x W x C]*, should be in the same dtype and value range as *ref_image* and should match the same number of channels if channels are provided. When no fill is passed, black is used (default absence of color).

Images output will mirror the input image format. As such, the *fill* value passed must be compatible with the input image channels for broadcasting. For example, a single channel input will not be able to be broadcast against a multi-channel *fill* input. A ValueError will be raised by the underlying numpy call in such cases.

NOTE: Due to the batch nature of this function, utilizing a fill color will consistently utilize more RAM due to the use of alpha blending

Assumptions:

- Mask input is per-pixel. Does not accept per-channel masks.
- Fill value input is in an applicable value range supported by the input image format, which is mirrored in output images.

Parameters

- **ref_image** – Reference image to generate perturbations from.
- **masks** – Mask matrix input of shape *[N x H x W]* where height and width dimensions are the same size as the input *ref_image*.
- **fill** – Optional fill for alpha-blending based on the input masks for the occluded regions as a scalar value, a per-channel sequence or a shape-matched image.
- **threads** – Optional number of threads to use for parallelism when set to a positive integer. If 0, a negative value, or *None*, work will be performed on the main-thread in-line.

Raises

ValueError – The input mask matrix was not three-dimensional, its last two dimensions did not match the shape of the input imagery, or the input fill value could not be broadcast against the input image.

Returns

A numpy array of masked images.

xaitk_saliency.utils.masking.**occlude_image_streaming**(*ref_image: ndarray*, *masks: Iterable[ndarray]*, *fill: int | Sequence[int] | ndarray | None = None*, *threads: int | None = None*) → Generator[ndarray, None, None]

Apply a number of input occlusion masks to the given reference image, producing a list of images equivalent in length, and parallel in order, to the input masks. This streaming version will return an iterator that yields occluded image matrices.

We expect the "mask" matrices and the image to be the same height and width, and for the mask matrix values to be in the [0, 1] range. In the mask matrix, values closer to 1 correspond to regions of the image that should *NOT* be occluded. E.g. a 0 in the mask will translate to *fully* occluding the corresponding location in the source image.

We optionally take in a "fill" that alpha-blend into masked regions of the input *ref_image*. *fill* may be either a scalar, sequence of scalars, or another image matrix congruent in shape to the *ref_image*. When *fill* is a scalar or a sequence of scalars, the scalars should be in the same data-type and value range as the input image. A sequence of scalars should be the same length as there are channels in the *ref_image*. When *fill* is an image matrix it should follow the format of *[H x W]* or *[H x W x C]*, should be in the same dtype and value range as *ref_image* and should match the same number of channels if channels are provided. When no fill is passed, black is used (default absence of color).

Images output will mirror the input image format. As such, the *fill* value passed must be compatible with the input image channels for broadcasting. For example, a single channel input will not be able to be broadcast against a multi-channel *fill* input. A ValueError will be raised by the underlying numpy call in such cases.

Assumptions:

- Mask input is per-pixel. Does not accept per-channel masks.
- Fill value input is in an applicable value range supported by the input image format, which is mirrored in output images.

Parameters

- **`ref_image`** – Original base image
- **`masks`** – Mask images in the [N, Height, Weight] shape format.
- **`fill`** – Optional fill for alpha-blending based on the input masks for the occluded regions as a scalar value, a per-channel sequence or a shape-matched image.
- **`threads`** – Optional number of threads to use for parallelism when set to a positive integer. If 0, a negative value, or *None*, work will be performed on the main-thread in-line.

Raises

`ValueError` – One or more input masks in the input iterable did not match shape of the input reference image.

Returns

A generator of numpy array masked images.

3.2 Heatmap Generation

These interfaces comprise a family of siblings that all perform a similar transformation, but require different standard inputs. There is no standard to rule them all without being so abstract that it would break the concept of interface abstraction, or the ability to substitute any arbitrary implementations of the interface without interrupting successful execution. Each interface is intended to handle different black-box outputs from different algorithmic categories. In the future, as additional algorithmic categories are identified for which saliency map generation is applicable, additional interfaces may be defined and added to this initial repertoire.

3.2.1 Interface: GenerateClassifierConfidenceSaliency

This interface proposes that implementations transform black-box image classification scores into saliency heatmaps. This should require a sequence of per-class confidences predicted on the reference image, a number of per-class confidences as predicted on perturbed images, as well as the masks of the reference image perturbations (as would be output from a `PerturbImage` implementation).

Implementations should use this input to generate a visual saliency heatmap for each input "class" in the input. This is both an effort to vectorize the operation for optimal performance, as well as to allow some algorithms to take advantage of differences in classification behavior for other classes to influence heatmap generation. For classifiers that generate many class label predictions, it is intended that only a subset of relevant class predictions need be provided here if computational performance is a consideration.

An immediate candidate implementation for this interface is the RISE algorithm [2] and occlusion-based saliency algorithms [3] that generate saliency heatmaps.

class
xaitk_saliency.interfaces.gen_classifier_conf_sal.**GenerateClassifierConfidenceSaliency**

Visual saliency map generation interface whose implementations transform black-box image classification scores into saliency heatmaps.

This should require a sequence of per-class confidences predicted on the reference image, a number of per-class confidences as predicted on perturbed images, as well as the masks of the reference image perturbations (as would be output from a *xaitk_saliency.interfaces.perturb_image.PerturbImage* implementation).

Implementations should use this input to generate a visual saliency heatmap for each input "class" in the input. This is both an effort to vectorize the operation for optimal performance, as well as to allow some algorithms to take advantage of differences in classification behavior for other classes to influence heatmap generation. For classifiers that generate many class label predictions, it is intended that only a subset of relevant class predictions need be provided here if computational performance is a consideration.

abstract generate(*image_conf: ndarray*, *perturbed_conf: ndarray*, *perturbed_masks: ndarray*) → ndarray

Generate a visual saliency heatmap matrix given the black-box classifier output on a reference image, the same classifier output on perturbed images and the masks of the visual perturbations.

Perturbation mask input into the *perturbed_masks* parameter here is equivalent to the perturbation mask output from a *xaitk_saliency.interfaces.perturb_image.PerturbImage.perturb()* method implementation. These should have the shape *[nMasks x H x W]*, and values in range [0, 1], where a value closer to 1 indicate areas of the image that are *unperturbed*. Note the type of values in masks can be either integer, floating point or boolean within the above range definition. Implementations are responsible for handling these expected variations.

Generated saliency heatmap matrices should be floating-point typed and be composed of values in the [-1,1] range. Positive values of the saliency heatmaps indicate regions which increase class confidence scores, while negative values indicate regions which decrease class confidence scores according to the model that generated input confidence values.

Parameters

- **image_conf** – Reference image predicted class-confidence vector, as a *numpy.ndarray*, for all classes that require saliency map generation. This should have a shape *[nClasses]*, be float-typed and with values in the [0,1] range.
- **perturbed_conf** – Perturbed image predicted class confidence matrix. Classes represented in this matrix should be congruent to classes represented in the *image_conf* vector. This should have a shape *[nMasks x nClasses]*, be float-typed and with values in the [0,1] range.

- **perturbed_masks** – Perturbation masks *numpy.ndarray* over the reference image. This should be parallel in association to the classification results input into the *perturbed_conf* parameter. This should have a shape *[nMasks x H x W]*, and values in range [0, 1], where a value closer to 1 indicate areas of the image that are *unperturbed*.

Returns

Generated visual saliency heatmap for each input class as a float-type *numpy.ndarray* of shape *[nClasses x H x W]*.

3.2.2 Interface: GenerateDescriptorSimilaritySaliency

This interface proposes that implementations require externally generated feature-vectors for two reference images between which we are trying to discern the feature-space saliency. This also requires the feature-vectors for perturbed images as well as the masks of the perturbations as would be output from a `PerturbImage` implementation. We expect perturbations to be relative to the second reference image feature-vector.

An immediate candidate implementation for this interface is the Similarity Based Saliency Maps (SBSM) algorithm [1].

class
xaitk_saliency.interfaces.gen_descriptor_sim_sal.**GenerateDescriptorSimilaritySaliency**

Visual saliency map generation interface whose implementations transform black-box feature vectors from multiple references and perturbations into saliency heatmaps.

This transformation requires a reference image, and a number of query images all translated into feature vectors via some black-box means. We are trying to discern the feature-space saliency between the reference image and each query image. This also requires the feature vectors for perturbed versions of the reference images as well as the masks of the perturbations as would be output from a *xaitk_saliency.interfaces.perturb_image.PerturbImage* implementation. The resulting saliency heatmaps are relative to the reference image.

abstract generate(*ref_descr: ndarray*, *query_descrs: ndarray*, *perturbed_descrs: ndarray*, *perturbed_masks: ndarray*) → ndarray

Generate a matrix of visual saliency heatmaps given the black-box descriptor generation output on a reference image, several query images, perturbed versions of the reference image and the masks of the visual perturbations.

Perturbation mask input into the *perturbed_masks* parameter here is equivalent to the perturbation mask output from a *xaitk_saliency.interfaces.perturb_image.PerturbImage.perturb()* method implementation. We expect perturbations to be relative to the reference image. These should have the shape *[nMasks x H x W]*, and values in range [0, 1], where a value closer to 1 indicates areas of the image that are *unperturbed*. Note the type of values in masks can be either integer, floating point or boolean within the above range definition. Implementations are responsible for handling these expected variations.

Generated saliency heatmap matrices should be floating-point typed and be composed of values in the [-1,1] range. Positive values of the saliency heatmaps indicate regions which increase image similarity scores, while negative values indicate regions which decrease image similarity scores according to the model that generated input feature vectors.

Parameters

- **ref_descr** – Reference image float feature vector, shape *[nFeats]*
- **query_descrs** – Query image float feature vectors, shape *[nQueryImgs x nFeats]*.
- **perturbed_descrs** – Feature vectors of reference image perturbations, float typed of shape *[nMasks x nFeats]*.

- **perturbed_masks** – Perturbation masks *numpy.ndarray* over the query image. This should be parallel in association to the *perturbed_descrs* parameter. This should have a shape *[nMasks x H x W]*, and values in range [0, 1], where a value closer to 1 indicates areas of the image that are *unperturbed*.

Returns

Generated saliency heatmaps as a float-typed *numpy.ndarray* with shape *[nQueryImgs x H x W]*.

3.2.3 Interface: GenerateDetectorProposalSaliency

This interface proposes that implementations transform black-box image object detection predictions into visual saliency heatmaps. This should require externally generated object detection predictions over some image, along with predictions for perturbed images and the perturbation masks for those images as would be output from a `PerturbImage` implementation. Object detection representations used here would need to encapsulate localization information (i.e. bounding box regions), class scores, and objectness scores (if applicable to the detector, such as YOLOv3). Object detections are converted into (4+1+nClasses) vectors (4 indices for bounding box locations, 1 index for objectness, and nClasses indices for different object classes).

Implementations should use this input to generate a visual saliency heatmap for each input detection. We assume that an input detection is coupled with a single truth class (or a single leaf node in a hierarchical structure). Input detections on the reference image may be drawn from ground truth or predictions as desired by the use case. As for perturbed image detections, we expect those to usually be decoupled from the source of reference image detections, which is why below we formulate the shape of perturbed image detects with `nProps` instead of `nDets` (though the value of that axis may be the same in some cases).

A candidate implementation for this interface is the D-RISE [4] algorithm.

class xaitk_saliency.interfaces.gen_detector_prop_sal.**GenerateDetectorProposalSaliency**

This interface proposes that implementations transform black-box image object detection predictions into visual saliency heatmaps. This should require externally-generated object detection predictions over some image, along with predictions for perturbed images and the perturbation masks for those images as would be output from a *xaitk_saliency.interfaces.perturb_image.PerturbImage* implementation.

Object detection representations used here would need to encapsulate localization information (i.e. bounding box regions), class scores, and objectness scores (if applicable to the detector, such as YOLOv3). Object detections are converted into (4+1+nClasses) vectors (4 indices for bounding box locations, 1 index for objectness, and nClasses indices for different object classes).

abstract generate(*ref_dets: ndarray*, *perturbed_dets: ndarray*, *perturb_masks: ndarray*) → ndarray

Generate visual saliency heatmap matrices for each reference detection, describing what visual information contributed to the associated reference detection.

We expect input detections to come from a black-box source that outputs our minimum requirements of a bounding-box, per-class scores. Objectness scores are required in our input format, but not necessarily from detection black-box methods as there is a sensible default value for this. See the *format_detection()* helper function for assistance in forming our input format, which includes this optional default fill-in. We expect objectness is a confidence score valued in the inclusive `[0,1]` range. We also expect classification scores to be in the inclusive `[0,1]` range.

We assume that an input detection is coupled with a single truth class (or a single leaf node in a hierarchical structure). Detections input as references (`ref_dets` parameter) may be either ground truth or predicted detections. As for perturbed image detections input (`perturbed_dets`), we expect the quantity of detections to be decoupled from the source of reference image detections, which is why below we formulate the shape of perturbed image detections with *nProps* instead of *nDets*.

Perturbation mask input into the *perturbed_masks* parameter here is equivalent to the perturbation mask output from a *xaitk_saliency.interfaces.perturb_image.PerturbImage.perturb()* method implementation. These should have the shape *[nMasks x H x W]*, and values in range [0, 1], where a value closer to 1 indicate areas of the image that are *unperturbed*. Note the type of values in masks can be either integer, floating point or boolean within the above range definition. Implementations are responsible for handling these expected variations.

Generated saliency heatmap matrices should be floating-point typed and be composed of values in the [-1,1] range. Positive values of the saliency heatmaps indicate regions which increase object detection scores, while negative values indicate regions which decrease object detection scores according to the model that generated input object detections.

Parameters

- **ref_dets** – Detections, objectness and class scores on a reference image as a float-typed array with shape *[nDets x (4+1+nClasses)]*.
- **perturbed_dets** – Object detections, objectness and class scores for perturbed variations of the reference image. We expect this to be a float-types array with shape *[nMasks x nProps x (4+1+nClasses)]*.
- **perturb_masks** – Perturbation masks *numpy.ndarray* over the reference image. This should be parallel in association to the detection propositions input into the *perturbed_dets* parameter. This should have a shape *[nMasks x H x W]*, and values in range [0, 1], where a value closer to 1 indicate areas of the image that are *unperturbed*.

Returns

A visual saliency heatmap matrix describing each input reference detection. These will be float-typed arrays with shape *[nDets x H x W]*.

Detection formatting helper

The *GenerateDetectorProposalSaliency.generate()* method takes in a specifically formatted matrix that combines three aspects of common detector model outputs: * bounding boxes * objectness scores * classification scores

We provide a helper function to merge distinct output data into the unified format.

xaitk_saliency.utils.detection.**format_detection**(*bbox_mat: ndarray*, *classification_mat: ndarray*, *objectness: ndarray | None = None*) → ndarray

Combine detection and classification output, with optional objectness output, into the combined format required for *GenerateDetectorProposalSaliency.generate()* *_dets input parameters.

We enforce some shape consistency so that we can create a valid output matrix. The input bounding box matrix should be of shape `[nDets x 4]`, the classification matrix should be of shape `[nDets x nClasses]`, and the objectness vector, if provided, should be of size `nDets`.

If an `objectness` score vector is not provided, we assume a vector of 1's.

The output of this function is a matrix that is of shape `[nDets x (4+1+nClasses)]`. This is the result of horizontally stacking the input in bbox, objectness and classification order. The output matrix data-type will follow numpy's rules about safe-casting given the combination of input matrix types.

In exceptions about shape mismatches, index 0 refers to the `bbox_mat` input, index 1 refers to the `objectness` vector, and index 2 refers to the `classification_mat`.

Parameters

- **bbox_mat** – Matrix of bounding boxes. This matrix should have the shape `[nDets x 4]`. The format of each row-vector is not important but generally expected to be `[left, top, right, bottom]` pixel coordinates. This matrix must be of a type that is float-castable.

- **classification_mat** – Matrix of classification scores from the detector or detection classifier. This should have the shape of `[nDets x nClasses]`. This matrix must be of a type that is float-castable.
- **objectness** – Optional vector of objectness scores for input detections. This is optional as not all detection models output this aspect. When provided, this should be a vector of ints/floats of size `nDets` to match the other parameter shapes.

Raises

ValueError – When input matrix shapes are mismatched such that they cannot be horizontally stacked.

Returns

Matrix combining bounding box, objectness and class confidences.

3.3 End-to-End Saliency Generation

Unlike the previous saliency heatmap generation interfaces, this interface uses a black-box classifier as input along with a reference image to generate visual saliency heatmaps.

A candidate implementation for this interface is the `PerturbationOcclusion` implementation or one of its sub-implementations (`RISEStack` or `SlidingWindowStack`).

3.3.1 Interface: GenerateImageClassifierBlackboxSaliency

class xaitk_saliency.interfaces.gen_image_classifier_blackbox_sal.**GenerateImageClassifierBlackboxSaliency**

This interface for algorithms takes a reference image and an image classifier black-box algorithm, then generates a number of visual saliency heatmap matrices, one for each class output by the classifier black box.

A classifier black box needs to be input, which requires some specification in how to operate the black box. The *smqtk_classifier.ClassifyImage* abstract interface is used to provide a minimal form that a black-box classifier requires: be able to classify an image into confidences for some number of class labels.

Generates a visual saliency heatmap for each input class as a float-type *numpy.ndarray* of shape *[nClasses x H x W]*.

generate(*ref_image: ndarray*, *blackbox: ClassifyImage*) → ndarray

Generates per-class visual saliency heatmaps for some classifier black box over some image of interest.

The input reference image is expected to be in matrix form and be in either a *H x W* or *H x W x C* shape format.

Output saliency map matrix should be (1) in the shape *nClasses x H x W*, (2) floating-point typed, and (3) composed of values in the *[-1, 1]* range. *nClasses* should be the quantity of unique class labels output by the given classifier black box. While specific algorithms determine the quantity of heatmaps returned, the height and width of returned heatmaps should be consistent with the input image, i.e. the *H* and *W* dimensions should match in size to the reference image's *H* and *W* dimensions. Positive values of the saliency heatmaps indicate regions that increase respective class confidence scores, while negative values indicate regions that decrease respective class confidence scores according to the given black-box classifier.

Parameters

- **ref_image** – Reference image over which visual saliency heatmaps will be generated.
- **blackbox** – The black-box classifier handle to perform arbitrary operations on in order to deduce visual saliency.

Raises

ShapeMismatchError – The implementation result visual saliency heatmap matrix did not have matching height and width components to the reference image.

Returns

A number of visual saliency heatmaps equivalent in number to the quantity of class labels output by the black-box classifier.

3.3.2 Interface: GenerateImageSimilarityBlackboxSaliency

class xaitk_saliency.interfaces.gen_image_similarity_blackbox_sal.
GenerateImageSimilarityBlackboxSaliency

This interface describes the generation of visual saliency heatmaps based on the similarity of a reference image to a number of query images. Similarity is deduced from the output of a black-box image feature vector generator that transforms each image to an embedding space.

The resulting saliency maps are relative to the reference image. As such, each map denotes regions in the reference image that make it more or less similar to the corresponding query image.

The *smqtk_descriptors.ImageDescriptorGenerator* interface is used to provide a common format for image feature vector generation.

generate(*ref_image: ndarray*, *query_images: Sequence[ndarray]*, *blackbox: ImageDescriptorGenerator*) → ndarray

Generates visual saliency maps based on the similarity of the reference image to each query image determined by the output of the blackbox feature vector generator.

The input reference image is expected to be a matrix in either a *H x W* or *H x W x C* shape format. The input query images should be a sequence of matrices, each of which should also be in either *H x W* or *H x W x C* format. Each query image is not required to have the same shape, however.

The output saliency map matrix should be (1) of shape *nQueryImgs x H x W* with matching height and width to the reference image, (2) floating-point typed, and (3) composed of values in the *[-1, 1]* range.

The *(0, 1]* range is intended to describe regions that are positively salient, and the *[-1, 0)* range is intended to describe regions that are negatively salient. Positive values of each saliency heatmap indicate regions of the reference image that increase its similarity to the corresponding query image, while negative values indicate regions that actively decrease its similarity to the corresponding query image.

Similarity is determined by the output of the feature vector generator and implementation specifics.

Parameters

- **ref_image** – Reference image to compute saliency for.
- **query_images** – Query images to compare the reference image to.
- **blackbox** – Black-box image feature vector generator.

Raises

ShapeMismatchError – The implementation's resulting heatmap matrix did not have matching height and width components to the reference image.

Returns

A matrix of saliency heatmaps relative to the reference image with shape *nQueryImgs x H x W*.

3.3.3 Interface: GenerateObjectDetectorBlackboxSaliency

class xaitk_saliency.interfaces.gen_object_detector_blackbox_sal.
GenerateObjectDetectorBlackboxSaliency

This interface describes the generation of visual saliency heatmaps for input object detections with respect to a given black box object detection and classification model.

This transformation requires reference detections to focus on explaining, and the image those detections were drawn from. For compatibility, the input detections specification are split into three separate inputs: bounding boxes, scores, and objectness. A visual saliency heatmap is generated for each input detection.

The *smqtk_detection.DetectImageObjects* abstract interface is used to provide a common format for a black-box object detector.

generate(*ref_image: ndarray*, *bboxes: ndarray*, *scores: ndarray*, *blackbox: DetectImageObjects*, *objectness: ndarray | None = None*) → ndarray

Generate per-detection visual saliency heatmaps for some object detector black-box over some input reference detections from some input reference image.

The input reference image is expected to be a matrix in either *[H x W]* or *[H x W x C]* shape format.

The reference detections are represented by three separate inputs: bounding boxes, scores, and objectness. The input bounding boxes are expected to be a matrix with shape *[nDets x 4]* where each row is the bounding box of a single detection in xyxy format. The input scores are expected to be a matrix with shape *[nDets x nClasses]* where each row is the scores for each class for a single detection. The order of each class score should match the order returned by the input black-box algorithm. The optional input objectness is expected to be a vector of length *nDets* containing the objectness score (single float value) for each reference detection. If this is not provided, it is assumed that each detection has an objectness score of 1.

If your detections consist of a single class prediction and confidence score instead of scores for each class, it is best practice to replace the objectness score with the confidence score and use a one-hot encoding of the prediction as the class scores.

The output saliency map matrix should be (1) in the shape *[nDets x H x W]* where *H* and *W* are the height and width respectively of the input reference image, (2) floating-point typed, and (3) composed of values in the *[-1, 1]* range.

The *(0, 1]* range is intended to describe regions that are positively salient, and the *[-1, 0)* range is intended to describe regions that are negatively salient. Positive values of the saliency heatmaps indicate regions that increase detection locality and class confidence scores, while negative values indicate regions that actively decrease detection locality and class confidence scores.

Parameters

- **ref_image** – Reference image that the input reference detections belong to.
- **bboxes** – The bounding boxes in xyxy format of the reference detections to generate visual saliency maps for. This should be a matrix with shape *[nDets x 4]*.
- **scores** – The class scores of the reference detections to generate visual saliency maps for. This should be a matrix with shape *[nDets x nClasses]*. The order of the scores should match that returned by the input black-box detection algorithm.
- **blackbox** – The black-box object detector to perform arbitrary operations on in order to deduce visual saliency.
- **objectness** – Optional objectness score for each reference detection. This should be a vector of length *nDets*. If not provided, it is assumed that each detection has an objectness score of 1.

Raises

- **ValueError** – The input reference image had an unexpected number of dimensions.
- **ValueError** – The input bounding boxes had a width other than 4.
- **ValueError** – The input bounding boxes, scores, and/or objectness scores do not match in quantity.
- **ShapeMismatchError** – The implementation result visual saliency heatmap matrix did not have matching height and width components to the reference image.
- **ShapeMismatchError** – The quantity of resulting heatmaps did not match the quantity of input reference detections.

Returns

A number of visual saliency heatmaps, one for each input reference detection. This is a single matrix of shape *[nDets x H x W]* where *H* and *W* are the height and width respectively of the input reference image.

3.4 Code Examples

For Jupyter Notebook examples of xaitk-saliency interfaces, see the `examples/` directory of the project.

3.5 References

1. Dong B, Collins R, Hoogs A. Explainability for Content-Based Image Retrieval. InCVPR Workshops 2019 Jun (pp. 95-98).
2. Petsiuk V, Das A, Saenko K. Rise: Randomized input sampling for explanation of black-box models. arXiv preprint arXiv:1806.07421. 2018 Jun 19.
3. Zeiler MD, Fergus R. Visualizing and understanding convolutional networks (2013). arXiv preprint arXiv:1311.2901. 2013.
4. Petsiuk V, Jain R, Manjunatha V, Morariu VI, Mehra A, Ordonez V, Saenko K. Black-box explanation of object detectors via saliency maps. arXiv preprint arXiv:2006.03204. 2020 Jun 5.

CHAPTER

FOUR

IMPLEMENTATIONS

Included with this toolkit are a number of implementations for the interfaces described in the previous section. Unlike the interfaces, which declare operation and use case, implementations provide variations on *how* to satisfy the interface-defined use case, varying trade-offs, or results implications.

4.1 Image Perturbation

4.1.1 Class: RandomGrid

class xaitk_saliency.impls.perturb_image.random_grid.**RandomGrid**(*n: int*, *s: Tuple[int, int]*, *p1: float*, *seed: int | None = None*, *threads: int | None = None*)

Generate masks using a random grid of set cell size. If the chosen cell size does not divide an image evenly, then the grid is over-sized and the resulting mask is centered and cropped. Each mask is also shifted randomly by a maximum of half the cell size in both x and y.

This method is based on RISE (http://bmvc2018.org/contents/papers/1064.pdf) but aims to address the changing cell size, given images of different sizes, aspect of that implementation. This method keeps cell size constant and instead adjusts the overall grid size for different sized images.

Parameters

- **n** – Number of masks to generate.
- **s** – Dimensions of the grid cells in pixels. E.g. (3, 4) would use a grid of 3x4 pixel cells.
- **p1** – Probability of a grid cell being set to 1 (not occluded). This should be a float value in the [0, 1] range.
- **seed** – A seed to use for the random number generator, allowing for masks to be reproduced.
- **threads** – Number of threads to use when generating masks. If this is <=0 or None, no threading is used and processing is performed in-line serially.

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(*ref_img: ndarray*) → ndarray

Transform an input reference image into a number of mask matrices indicating the perturbed regions.

Output mask matrix should be three-dimensional with the format [nMasks x Height x Width], sharing the same height and width to the input reference image. The implementing algorithm may determine the quantity of output masks per input image. These masks should indicate the regions in the corresponding perturbed image that have been modified. Values should be in the [0, 1] range, where a value closer to 1.0 indicates areas of the image that are unperturbed. Note that output mask matrices may be of a floating-point type to allow for fractional perturbation.

Parameters
ref_image – Reference image to generate perturbations from.

Returns
Mask matrix with shape [nMasks x Height x Width].

4.1.2 Class: RISEGrid

class xaitk_saliency.impls.perturb_image.rise.**RISEGrid**(*n: int*, *s: int*, *p1: float*, *seed: int | None = None*, *threads: int | None = 4*)

Based on Petsiuk et. al: http://bmvc2018.org/contents/papers/1064.pdf

Implementation is borrowed from the original authors: https://github.com/eclique/RISE/blob/master/explanations.py

Generate a set of random binary masks

Parameters

- **n** – Number of random masks used in the algorithm. E.g. 1000.
- **s** – Spatial resolution of the small masking grid. E.g. 8. Assumes square grid.
- **p1** – Probability of the grid cell being set to 1 (otherwise 0). This should be a float value in the [0, 1] range. E.g. 0.5.
- **seed** – A seed to pass into the constructed random number generator to allow for reproducibility
- **threads** – The number of threads to utilize when generating masks. If this is <=0 or None, no threading is used and processing is performed in-line serially.

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(*ref_image: ndarray*) → ndarray

Transform an input reference image into a number of mask matrices indicating the perturbed regions.

Output mask matrix should be three-dimensional with the format [nMasks x Height x Width], sharing the same height and width to the input reference image. The implementing algorithm may determine the quantity of output masks per input image. These masks should indicate the regions in the corresponding perturbed image that have been modified. Values should be in the [0, 1] range, where a value closer to 1.0 indicates areas of the image that are unperturbed. Note that output mask matrices may be of a floating-point type to allow for fractional perturbation.

Parameters
ref_image – Reference image to generate perturbations from.

Returns
Mask matrix with shape [nMasks x Height x Width].

4.1.3 Class: SlidingRadial

class xaitk_saliency.impls.perturb_image.sliding_radial.**SlidingRadial**(*radius: Tuple[float, float] = (50, 50)*, *stride: Tuple[int, int] = (20, 20)*, *sigma: Tuple[float, float] | None = None*)

Produce perturbation matrices generated by sliding a radial occlusion area with configured radius over the area of an image. When the two radius values are the same, circular masks are generated; otherwise, elliptical masks are generated. Passing sigma values will apply a Gaussian filter to the mask, blurring it. This results in a smooth transition from full occlusion in the center of the radial to no occlusion at the edge.

Due to the geometry of sliding radials, if the stride given does not evenly divide the radial size along the applicable axis, then the result plane of values when summing the generated masks will not be even.

Related, if the stride is set to be larger than the radial diameter, the resulting plane of summed values will also not be even, as there be increasingly long valleys of unperturbed space between masked regions.

The generated masks are boolean if no blurring is used, otherwise the masks will be of floating-point type in the [0, 1] range.

Parameters

- **radius** – The radius of the occlusion area in pixels as a tuple with format *(radius_y, radius_x)*.
- **stride** – The striding step in pixels for the center of the radial as a tuple with format *(height_step, width_step)*.
- **sigma** – The sigma values for the Gaussian filter applied to masks in pixels as a tuple with format *(sigma_y, sigma_x)*.

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(*ref_image: ndarray*) → ndarray

Transform an input reference image into a number of mask matrices indicating the perturbed regions.

Output mask matrix should be three-dimensional with the format [nMasks x Height x Width], sharing the same height and width to the input reference image. The implementing algorithm may determine the quantity of output masks per input image. These masks should indicate the regions in the corresponding perturbed image that have been modified. Values should be in the [0, 1] range, where a value closer to 1.0 indicates areas of the image that are unperturbed. Note that output mask matrices may be of a floating-point type to allow for fractional perturbation.

Parameters
ref_image – Reference image to generate perturbations from.

Returns
Mask matrix with shape [nMasks x Height x Width].

4.1.4 Class: SlidingWindow

class xaitk_saliency.impls.perturb_image.sliding_window.**SlidingWindow**(*window_size: Tuple[int, int] = (50, 50)*, *stride: Tuple[int, int] = (20, 20)*)

Produce perturbation matrices based on hard, block-y occlusion areas as generated by sliding a window of a configured size over the area of an image.

Due to the geometry of sliding windows, if the stride given does not evenly divide the window size along the applicable axis, then the result plane of values when summing the generated masks will not be even.

Related, if the stride is set to be larger than the window size, the resulting plane of summed values will also not be even, as there be increasingly long valleys of unperturbed space between masked regions.

Parameters

- **window_size** – The block window size in pixels as a tuple with format *(height, width)*.
- **stride** – The sliding window striding step in pixels as a tuple with format *(height_step, width_step)*.

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):
```

(continues on next page)

(continued from previous page)

```
...     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(*ref_image: ndarray*) → ndarray

Transform an input reference image into a number of mask matrices indicating the perturbed regions.

Output mask matrix should be three-dimensional with the format [nMasks x Height x Width], sharing the same height and width to the input reference image. The implementing algorithm may determine the quantity of output masks per input image. These masks should indicate the regions in the corresponding perturbed image that have been modified. Values should be in the [0, 1] range, where a value closer to 1.0 indicates areas of the image that are unperturbed. Note that output mask matrices may be of a floating-point type to allow for fractional perturbation.

Parameters
ref_image – Reference image to generate perturbations from.

Returns
Mask matrix with shape [nMasks x Height x Width].

4.2 Heatmap Generation

4.2.1 Class: DRISEScoring

class xaitk_saliency.impls.gen_detector_prop_sal.drise_scoring.**DRISEScoring**

This D-RISE implementation transforms black-box object detector predictions into visual saliency heatmaps. Specifically, we make use of perturbed detections generated using the *RISEGrid* image perturbation class and a similarity metric that captures both the localization and categorization aspects of object detection.

Object detection representations used here would need to encapsulate localization information (i.e. bounding box regions), class scores, and objectness scores (if applicable to the detector, such as YOLOv3). Object detections are converted into (4+1+nClasses) vectors (4 indices for bounding box locations, 1 index for objectness, and nClasses indices for different object classes).

If your detections consist of a single class prediction and confidence score instead of scores for each class, it is best practice to replace the objectness score with the confidence score and use a one-hot encoding of the prediction as the class scores.

Based on Petsiuk et al: https://arxiv.org/abs/2006.03204

generate(*ref_dets: ndarray*, *perturbed_dets: ndarray*, *perturbed_masks: ndarray*) → ndarray

Generate visual saliency heatmap matrices for each reference detection, describing what visual information contributed to the associated reference detection.

We expect input detections to come from a black-box source that outputs our minimum requirements of a bounding-box, per-class scores. Objectness scores are required in our input format, but not necessarily from detection black-box methods as there is a sensible default value for this. See the *format_detection()* helper function for assistance in forming our input format, which includes this optional default fill-in. We

expect objectness is a confidence score valued in the inclusive `[0,1]` range. We also expect classification scores to be in the inclusive `[0,1]` range.

We assume that an input detection is coupled with a single truth class (or a single leaf node in a hierarchical structure). Detections input as references (`ref_dets` parameter) may be either ground truth or predicted detections. As for perturbed image detections input (`perturbed_dets`), we expect the quantity of detections to be decoupled from the source of reference image detections, which is why below we formulate the shape of perturbed image detections with *nProps* instead of *nDets*.

Perturbation mask input into the *perturbed_masks* parameter here is equivalent to the perturbation mask output from a *xaitk_saliency.interfaces.perturb_image.PerturbImage.perturb()* method implementation. These should have the shape *[nMasks x H x W]*, and values in range [0, 1], where a value closer to 1 indicate areas of the image that are *unperturbed*. Note the type of values in masks can be either integer, floating point or boolean within the above range definition. Implementations are responsible for handling these expected variations.

Generated saliency heatmap matrices should be floating-point typed and be composed of values in the [-1,1] range. Positive values of the saliency heatmaps indicate regions which increase object detection scores, while negative values indicate regions which decrease object detection scores according to the model that generated input object detections.

Parameters

- **ref_dets** – Detections, objectness and class scores on a reference image as a float-typed array with shape *[nDets x (4+1+nClasses)]*.
- **perturbed_dets** – Object detections, objectness and class scores for perturbed variations of the reference image. We expect this to be a float-types array with shape *[nMasks x nProps x (4+1+nClasses)]*.
- **perturb_masks** – Perturbation masks *numpy.ndarray* over the reference image. This should be parallel in association to the detection propositions input into the *perturbed_dets* parameter. This should have a shape *[nMasks x H x W]*, and values in range [0, 1], where a value closer to 1 indicate areas of the image that are *unperturbed*.

Returns

A visual saliency heatmap matrix describing each input reference detection. These will be float-typed arrays with shape *[nDets x H x W]*.

get_config() → dict

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

iou(*box_a: ndarray*, *box_b: ndarray*) → ndarray

Compute the intersection over union (IoU) of two sets of boxes.

E.g.:

A ∩ B / A ∪ B = A ∩ B / (area(A) + area(B) - A ∩ B)

Parameters

- **box_a** – (np.array) bounding boxes, Shape: [A,4]
- **box_b** – (np.array) bounding boxes, Shape: [B,4]

Returns

iou(np.array), Shape: [A,B].

4.2.2 Class: OcclusionScoring

class xaitk_saliency.impls.gen_classifier_conf_sal.occlusion_scoring.**OcclusionScoring**

This saliency implementation transforms black-box image classification scores into saliency heatmaps. This should require a sequence of per-class confidences predicted on the reference image, a number of per-class confidences as predicted on perturbed images, as well as the masks of the reference image perturbations (as would be output from a *PerturbImage* implementation).

The perturbation masks used by the following implementation are expected to be of type integer. Masks containing values of type float are rounded to the nearest value and binarized with value 1 replacing values greater than or equal to half of the maximum value in mask after rounding while 0 replaces the rest.

generate(*image_conf: ndarray*, *perturbed_conf: ndarray*, *perturbed_masks: ndarray*) → ndarray

Generate a visual saliency heatmap matrix given the black-box classifier output on a reference image, the same classifier output on perturbed images and the masks of the visual perturbations.

Perturbation mask input into the *perturbed_masks* parameter here is equivalent to the perturbation mask output from a *xaitk_saliency.interfaces.perturb_image.PerturbImage.perturb()* method implementation. These should have the shape *[nMasks x H x W]*, and values in range [0, 1], where a value closer to 1 indicate areas of the image that are *unperturbed*. Note the type of values in masks can be either integer, floating point or boolean within the above range definition. Implementations are responsible for handling these expected variations.

Generated saliency heatmap matrices should be floating-point typed and be composed of values in the [-1,1] range. Positive values of the saliency heatmaps indicate regions which increase class confidence scores, while negative values indicate regions which decrease class confidence scores according to the model that generated input confidence values.

Parameters

- **image_conf** – Reference image predicted class-confidence vector, as a *numpy.ndarray*, for all classes that require saliency map generation. This should have a shape *[nClasses]*, be float-typed and with values in the [0,1] range.
- **perturbed_conf** – Perturbed image predicted class confidence matrix. Classes represented in this matrix should be congruent to classes represented in the *image_conf* vector. This should have a shape *[nMasks x nClasses]*, be float-typed and with values in the [0,1] range.
- **perturbed_masks** – Perturbation masks *numpy.ndarray* over the reference image. This should be parallel in association to the classification results input into the *perturbed_conf* parameter. This should have a shape *[nMasks x H x W]*, and values in range [0, 1], where a value closer to 1 indicate areas of the image that are *unperturbed*.

Returns
Generated visual saliency heatmap for each input class as a float-type *numpy.ndarray* of shape *[nClasses x H x W]*.

`get_config`() → dict

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

4.2.3 Class: RISEScoring

`class` xaitk_saliency.impls.gen_classifier_conf_sal.rise_scoring.`RISEScoring`(*p1: float = 0.0*)

Saliency map generation based on the original RISE implementation. This version utilizes only the input perturbed image confidence predictions and does not utilize reference image confidences. This implementation also takes influence from debiased RISE and may take an optional debias probability, *p1* (0 by default). In the original paper this is paired with the same probability used in RISE perturbation mask generation (see the *p1* parameter in *xaitk_saliency.impls.perturb_image.rise.RISEGrid*).

Based on Hatakeyama et. al: https://openaccess.thecvf.com/content/ACCV2020/papers/Hatakeyama_Visualizing_Color-wise_Saliency_of_Black-Box_Image_Classification_Models_ACCV_2020_paper.pdf

Generate RISE-based saliency maps with optional p1 de-biasing.

Parameters
p1 – De-biasing parameter based on the masking probability. This should be a float value in the [0, 1] range.

Raises
ValueError – Input *p1* was not in the [0,1] range.

`generate`(*image_conf: ndarray*, *perturbed_conf: ndarray*, *perturbed_masks: ndarray*) → ndarray

Generate a visual saliency heatmap matrix given the black-box classifier output on a reference image, the same classifier output on perturbed images and the masks of the visual perturbations.

Perturbation mask input into the *perturbed_masks* parameter here is equivalent to the perturbation mask output from a *xaitk_saliency.interfaces.perturb_image.PerturbImage.perturb()* method implementation. These should have the shape *[nMasks x H x W]*, and values in range [0, 1], where a value closer to 1 indicate areas of the image that are *unperturbed*. Note the type of values in masks can be either integer, floating point or boolean within the above range definition. Implementations are responsible for handling these expected variations.

Generated saliency heatmap matrices should be floating-point typed and be composed of values in the [-1,1] range. Positive values of the saliency heatmaps indicate regions which increase class confidence scores, while negative values indicate regions which decrease class confidence scores according to the model that generated input confidence values.

Parameters

- **image_conf** – Reference image predicted class-confidence vector, as a *numpy.ndarray*, for all classes that require saliency map generation. This should have a shape *[nClasses]*, be float-typed and with values in the [0,1] range.
- **perturbed_conf** – Perturbed image predicted class confidence matrix. Classes represented in this matrix should be congruent to classes represented in the *image_conf* vector. This should have a shape *[nMasks x nClasses]*, be float-typed and with values in the [0,1] range.
- **perturbed_masks** – Perturbation masks *numpy.ndarray* over the reference image. This should be parallel in association to the classification results input into the *perturbed_conf* parameter. This should have a shape *[nMasks x H x W]*, and values in range [0, 1], where a value closer to 1 indicate areas of the image that are *unperturbed*.

Returns

Generated visual saliency heatmap for each input class as a float-type *numpy.ndarray* of shape *[nClasses x H x W]*.

`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

4.2.4 Class: SimilarityScoring

`class` xaitk_saliency.impls.gen_descriptor_sim_sal.similarity_scoring.`SimilarityScoring`(*proximity_metric: str = 'euclidean'*)

This saliency implementation transforms proximity in feature space into saliency heatmaps. This should require feature vectors for the reference image, for each query image, and for perturbed versions of the reference image, as well as the masks of the reference image perturbations (as would be output from a *PerturbImage* implementation).

The resulting saliency maps are relative to the reference image. As such, each map denotes regions in the reference image that make it more or less similar to the corresponding query image.

Parameters

proximity_metric – The type of comparison metric used to determine proximity in feature space. The type of comparison metric supported is restricted by scipy's cdist() function. The following metrics are supported in scipy.

'braycurtis', 'canberra', 'chebyshev', 'cityblock', 'correlation', 'cosine', 'dice', 'euclidean', 'hamming', 'jaccard', 'jensenshannon', 'kulsinski', 'mahalanobis', 'matching', 'minkowski',

'rogerstanimoto', 'russellrao', 'seuclidean', 'sokalmichener', 'sokalsneath', 'sqeuclidean', 'wminkowski', 'yule'.

generate(*ref_descr: ndarray*, *query_descrs: ndarray*, *perturbed_descrs: ndarray*, *perturbed_masks: ndarray*) → ndarray

Generate a matrix of visual saliency heatmaps given the black-box descriptor generation output on a reference image, several query images, perturbed versions of the reference image and the masks of the visual perturbations.

Perturbation mask input into the *perturbed_masks* parameter here is equivalent to the perturbation mask output from a `xaitk_saliency.interfaces.perturb_image.PerturbImage.perturb()` method implementation. We expect perturbations to be relative to the reference image. These should have the shape *[nMasks x H x W]*, and values in range [0, 1], where a value closer to 1 indicates areas of the image that are *unperturbed*. Note the type of values in masks can be either integer, floating point or boolean within the above range definition. Implementations are responsible for handling these expected variations.

Generated saliency heatmap matrices should be floating-point typed and be composed of values in the [-1,1] range. Positive values of the saliency heatmaps indicate regions which increase image similarity scores, while negative values indicate regions which decrease image similarity scores according to the model that generated input feature vectors.

Parameters

- **ref_descr** – Reference image float feature vector, shape *[nFeats]*
- **query_descrs** – Query image float feature vectors, shape *[nQueryImgs x nFeats]*.
- **perturbed_descrs** – Feature vectors of reference image perturbations, float typed of shape *[nMasks x nFeats]*.
- **perturbed_masks** – Perturbation masks *numpy.ndarray* over the query image. This should be parallel in association to the *perturbed_descrs* parameter. This should have a shape *[nMasks x H x W]*, and values in range [0, 1], where a value closer to 1 indicates areas of the image that are *unperturbed*.

Returns

Generated saliency heatmaps as a float-typed *numpy.ndarray* with shape *[nQueryImgs x H x W]*.

get_config() → dict

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

4.2.5 Class: SquaredDifferenceScoring

class xaitk_saliency.impls.gen_classifier_conf_sal.squared_difference_scoring.**SquaredDifferenceScoring**

This saliency implementation transforms black-box confidence predictions from a classification-style network into saliency heatmaps. This should require a sequence of classification scores predicted on the reference image, a number of classification scores predicted on perturbed images, as well as the masks of the reference image perturbations (as would be output from a *PerturbImage* implementation).

This implementation uses the squared difference of the reference scores and the perturbed scores to compute the saliency maps. This gives an indication of general saliency without distinguishing between positive and negative. The resulting maps are normalized between the range [0,1].

Based on Greydanus et. al: https://arxiv.org/abs/1711.00138

generate(*reference: ndarray*, *perturbed: ndarray*, *perturbed_masks: ndarray*) → ndarray

Generate a visual saliency heatmap matrix given the black-box classifier output on a reference image, the same classifier output on perturbed images and the masks of the visual perturbations.

Perturbation mask input into the *perturbed_masks* parameter here is equivalent to the perturbation mask output from a *xaitk_saliency.interfaces.perturb_image.PerturbImage.perturb()* method implementation. These should have the shape *[nMasks x H x W]*, and values in range [0, 1], where a value closer to 1 indicate areas of the image that are *unperturbed*. Note the type of values in masks can be either integer, floating point or boolean within the above range definition. Implementations are responsible for handling these expected variations.

Generated saliency heatmap matrices should be floating-point typed and be composed of values in the [-1,1] range. Positive values of the saliency heatmaps indicate regions which increase class confidence scores, while negative values indicate regions which decrease class confidence scores according to the model that generated input confidence values.

Parameters

- **image_conf** – Reference image predicted class-confidence vector, as a *numpy.ndarray*, for all classes that require saliency map generation. This should have a shape *[nClasses]*, be float-typed and with values in the [0,1] range.
- **perturbed_conf** – Perturbed image predicted class confidence matrix. Classes represented in this matrix should be congruent to classes represented in the *image_conf* vector. This should have a shape *[nMasks x nClasses]*, be float-typed and with values in the [0,1] range.
- **perturbed_masks** – Perturbation masks *numpy.ndarray* over the reference image. This should be parallel in association to the classification results input into the *perturbed_conf* parameter. This should have a shape *[nMasks x H x W]*, and values in range [0, 1], where a value closer to 1 indicate areas of the image that are *unperturbed*.

Returns

Generated visual saliency heatmap for each input class as a float-type *numpy.ndarray* of shape *[nClasses x H x W]*.

get_config() → dict

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

4.3 End-to-End Saliency Generation

4.3.1 Image Classification

Class: PerturbationOcclusion

class xaitk_saliency.impls.gen_image_classifier_blackbox_sal.occlusion_based.**PerturbationOcclusion**(*perturber: PerturbImage*, *generator: GenerateClassifierConfidenceSalien*, *threads: int = 0*)

Generator composed of modular perturbation and occlusion-based algorithms.

This implementation exposes a public attribute *fill*. This may be set to a scalar or sequence value to indicate a color that should be used for filling occluded areas as determined by the given *PerturbImage* implementation. This is a parameter to be set during runtime as this is most often driven by the black-box algorithm used, if at all.

Parameters

- **perturber** – PerturbImage implementation instance for generating masks that will dictate occlusion.
- **generator** – Implementation instance for generating saliency masks given occlusion masks and classifier outputs.
- **threads** – Optional number threads to use to enable parallelism in applying perturbation masks to an input image. If 0, a negative value, or *None*, work will be performed on the main-thread in-line.

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

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)

        #
        # 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 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'}
```

Class: RISEStack

class xaitk_saliency.impls.gen_image_classifier_blackbox_sal.rise.**RISEStack**(*n: int*, *s: int*, *p1: float*, *seed: int | None = None*, *threads: int = 0*, *debiased: bool = True*)

Encapsulation of the perturbation-occlusion method using specifically the RISE implementations of the component algorithms.

This more specifically encapsulates the original RISE method as presented in their paper and code. See references in the `RISEGrid` and `RISEScoring` documentation.

This implementation shares the *p1* probability with the internal *RISEScoring* instance use, effectively causing this implementation to utilize debiased RISE.

Parameters

- **n** – Number of random masks used in the algorithm. E.g. 1000.
- **s** – Spatial resolution of the small masking grid. E.g. 8. Assumes square grid.
- **p1** – Probability of the grid cell being set to 1 (otherwise 0). This should be a float value in the [0, 1] range. E.g. 0.5.
- **seed** – A seed to pass into the constructed random number generator to allow for reproducibility
- **threads** – The number of threads to utilize when generating masks. If this is <=0 or None, no threading is used and processing is performed in-line serially.
- **debiased** – If we should pass the provided debiasing parameter to the RISE saliency map generation algorithm. See the `RISEScoring()` documentation for more details on debiasing.

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

Class: SlidingWindowStack

class xaitk_saliency.impls.gen_image_classifier_blackbox_sal.slidingwindow.**SlidingWindowStack**(*window_size: Tuple[int, int] = (50, 50)*, *stride: Tuple[int, int] = (20, 20)*, *threads: int = 0*)

Encapsulation of the perturbation-occlusion method using specifically sliding windows and the occlusion-scoring method. See the `SlidingWindow` and `OcclusionScoring` documentation for more details.

Parameters

- **window_size** – The block window size as a tuple with format *(height, width)*.
- **stride** – The sliding window striding step as a tuple with format *(height_step, width_step)*.
- **threads** – Optional number threads to use to enable parallelism in applying perturbation masks to an input image. If 0, a negative value, or *None*, work will be performed on the main-thread in-line.

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'}
```

4.3.2 Image Similarity

Class: PerturbationOcclusion

class xaitk_saliency.impls.gen_image_similarity_blackbox_sal.occlusion_based.**PerturbationOcclusion**(*perturber: PerturbImage*, *generator: GenerateDescriptorSimilaritySaliency*, *fill: int | Sequence | ndarray | None = None*, *threads: int | None = None*)

Image similarity saliency generator composed of modular perturbation and occlusion-based algorithms.

This implementation exposes its *fill* attribute as public. This allows it to be set during runtime as this is most often driven by the black-box algorithm used, if at all.

Parameters

- **perturber** – *PerturbImage* implementation instance for generating occlusion masks.
- **generator** – *GenerateDescriptorSimilaritySaliency* implementation instance for generating saliency masks given occlusion masks and image feature vector generator outputs.
- **fill** – Optional fill for alpha-blending the occluded regions based on the masks generated by the given perturber. Can be a scalar value, a per-channel sequence or a shape-matched image.
- **threads** – Optional number threads to use to enable parallelism in applying perturbation masks to an input image. If 0, a negative value, or *None*, work will be performed on the main-thread in-line.

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

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)

        #
        # 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 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'}
```

Class: SBSMStack

class xaitk_saliency.impls.gen_image_similarity_blackbox_sal.sbsm.**SBSMStack**(*window_size: Tuple[int, int] = (50, 50)*, *stride: Tuple[int, int] = (20, 20)*, *proximity_metric: str = 'euclidean'*, *fill: int | Sequence[int] | ndarray | None = None*, *threads: int | None = None*)

Encapsulation of the perturbation-occlusion method using specifically the sliding window image perturbation and similarity scoring algorithms to generate similarity-based visual saliency maps. See the documentation of `SlidingWindow` and `SimilarityScoring` for details.

Parameters

- **window_size** – The block window size as a tuple with format *(height, width)*.
- **stride** – The sliding window striding step as a tuple with format *(height_step, width_step)*.
- **proximity_metric** – The type of comparison metric used to determine proximity in feature space. The type of comparison metric supported is restricted by scipy's cdist() function. The following metrics are supported in scipy.

  'braycurtis', 'canberra', 'chebyshev', 'cityblock', 'correlation', 'cosine', 'dice', 'euclidean', 'hamming', 'jaccard', 'jensenshannon', 'kulsinski', 'mahalanobis', 'matching', 'minkowski', 'rogerstanimoto', 'russellrao', 'seuclidean', 'sokalmichener', 'sokalsneath', 'sqeuclidean', 'wminkowski', 'yule'.

- **threads** – Optional number threads to use to enable parallelism in applying perturbation masks to an input image. If 0, a negative value, or *None*, work will be performed on the main-thread in-line.

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'}
```

4.3.3 Object Detection

Class: PerturbationOcclusion

class xaitk_saliency.impls.gen_object_detector_blackbox_sal.occlusion_based.**PerturbationOcclusion**(*perturber: PerturbImage*, *generator: GenerateDetectorProposalSaliency*, *fill: int | Sequence | ndarray | None = None*, *threads: int | None = 0*)

Generator composed of modular perturbation and occlusion-based algorithms.

This implementation exposes its *fill* attribute as public. This allows it to be set during runtime as this is most often driven by the black-box algorithm used, if at all.

Parameters

- **perturber** – *PerturbImage* implementation instance for generating occlusion masks.
- **generator** – *GenerateDetectorProposalSaliency* implementation instance for generating saliency masks given occlusion masks and black-box detector outputs.
- **fill** – Optional fill for alpha-blending the occluded regions based on the masks generated by the given perturber. Can be a scalar value, a per-channel sequence or a shape-matched image.
- **threads** – Optional number threads to use to enable parallelism in applying perturbation masks to an input image. If 0, a negative value, or *None*, work will be performed on the main-thread in-line.

Class: DRISEStack

class xaitk_saliency.impls.gen_object_detector_blackbox_sal.drise.**DRISEStack**(*n: int*, *s: int*, *p1: float*, *seed: int | None = None*, *fill: int | Sequence[int] | ndarray | None = None*, *threads: int | None = 0*)

Encapsulation of the perturbation-occlusion method using the RISE image perturbation and DRISE scoring algorithms to generate visual saliency maps for object detections. See references in the `RISEGrid` and `DRISEScoring` documentation.

Parameters

- **n** – Number of random masks used in the algorithm.
- **s** – Spatial resolution of the small masking grid. Assumes square grid.
- **p1** – Probability of the grid cell being set to 1 (not occluded). This should be a float value in the [0, 1] range.
- **seed** – A seed to pass to the constructed random number generator to allow for reproducibility.
- **fill** – Optional fill for alpha-blending the occluded regions based on the masks generated by the *RISEGrid* perturber. Can be a scalar value, a per-channel sequence or a shape-matched image.
- **threads** – The number of threads to utilize when generating masks. If this is <=0 or None, no threading is used and processing is performed in-line serially.

CHAPTER

FIVE

REVIEW PROCESS

The process for reviewing and integrating branches into xaitk-saliency is described below.

For guidelines on contributing to xaitk-saliency, see `CONTRIBUTING.md`.

For guidelines on the release process for xaitk-saliency, see Release Process and Notes.

The review process consists of the following steps:

5.1 Pull Request

A PR is initiated by a user intending to integrate a branch from their forked repository. Before the branch is integrated into the xaitk-saliency 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.

5.1.1 Workflow Status

The submitter must set the status of their PR.

Draft

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

Open

Indicates that a PR is ready for review and that the submitter of the PR thinks that the branch is ready to be merged. If the submitter is still working on the PR and simply wants feedback, they must request it and leave their branch marked as a `Draft`.

Closed

Indicates that the PR is resolved or discarded.

5.2 Continuous Integration

The following checks are included in the automated portion of the review process. These are run as part of the CI/CD pipeline driven by GitHub actions. The success or failure of each may be seen in-line in a submitted PR in the "Checks" section of the PR view.

5.2.1 ghostflow

Runs basic checks on the commits submitted in a PR. Should ghostflow find any issues with the build, the diagnostics are written out, prompting the submitter to correct the reported issues. If there are no issues, or just warnings, ghostflow simply reports a successful build. The branch should usually pass this check before it can be merged. In the rare case that a PR is not subject to a change note, then failure of this check, specifically in regard to that lower level check, may be ignored by the reviewer.

Some reports such as whitespace issues will need to be corrected by rewriting the commit. This is generally handled by performing `git commit --fixup=...` commits and performing a `git rebase -i --autosquash ...` rebase afterwards, or a more simple global squash if appropriate.

5.2.2 LGTM Analysis

Runs a more advanced code analysis tool over the branch that can address issues that the submitter might not have noticed. Should LGTM find an issue, it will write a comment on the PR and provide a link to the LGTM website for more detailed information. The comment should be addressed by the submitter before continuing to submit for review. Ideally a submitted PR adds no new issues as reported by LGTM.

Passage of this check is not strictly required but highly encouraged.

5.2.3 SonarCloud Code Analysis

Similar to LGTM, this service performs a more in-depth analysis of the code. This should provide the same output as a SonarQube scan.

Passage of this check is not strictly required but highly encouraged.

5.2.4 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.

5.2.5 MyPy

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.

5.2.6 Unittests

Runs the unittests created under `tests/` as well as any doc-tests found in doc-strings 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.

5.2.7 CodeCov

This check reports aggregate code coverage as reported from output of the unittest jobs. This check requires that all test code be "covered" (i.e. there is no dead-code in the tests) and that a minimum coverage bar is met for package code changed or added in the PR. The configuration for this may be found in the `codecov.yml` file in the repository root.

Passage of these checks is strictly required.

5.2.8 ReadTheDocs Documentation Build

This check ensures that the documentation portion of the package is buildable by the current host ReadTheDocs.org.

Passage of these checks is strictly required.

5.2.9 Example Notebooks Execution

This check executes included example notebooks to ensure their proper functionality with the package with respect to a pull 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 this check is not strictly required but highly encouraged.

5.3 Human Review

Once the automatic checks are either resolved or addressed, the submitted PR will need to go through a human review. Reviewers should add comments to provide feedback and raise potential issues. Should the PR pass their review, the reviewer should then indicate that it has their approval using the GitHub review interface to flag the PR as `Approved`.

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

If a PR is not a draft and has an approving review, it may be merged at any time.

5.3.1 Notebooks

The default preference is that all Jupyter Notebooks be included in execution of the Notebook CI workflow (here: `.github/workflows/ci-example-notebooks.yml`). If a notebook is added, 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 specific notebooks should be added to the relevant section in `examples/README.md`.

5.4 Resolving a Branch

5.4.1 Merge

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

5.4.2 Close

If it is decided that the PR will not be integrated into xaitk-saliency, then it can be closed through GitHub.

CHAPTER

SIX

RELEASE PROCESS AND NOTES

6.1 Steps of the xaitk-saliency 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.

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

6.1.1 Create and Merge Version Update Branch

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 `master` 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.

   ```
   $ # 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 `master` locally and push the updated `master` to upstream. (Replace "upstream" in the example below with your applicable remote name.)

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

8. *Draft a new release on GitHub* for the new version.
9. Update version reference in the XAITK/xaitk.github.io home page to 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 `master` locally and push the updated `master` to upstream. (Replace "upstream" in the example below with your applicable remote name.)

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

8. *Draft a new release on GitHub* for the new version.
9. If this patch release now represents the highest version of the package, update version reference in the XAITK/xaitk.github.io home page to the new version.

Patching an Older Release

When patching a major/minor release that is not the latest pair, 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. Patch topic-branches should be based on this 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 `master`, if applicable (some patches may only make sense for specific versions).

6.1.2 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 `master` 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 package version.

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

6.1.3 Draft a new release on Github

After creating and pushing a new version tag, a GitHub "release" should be made.

a. Navigate to the GitHub Releases page for the xaitk-saliency repository.

b. Click the "Draft a new release" button (or go here).

c. Select from the "Choose a tag" drop down the tag version just created and pushed

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

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

f. Select the "This is a pre-release" check-box if applicable.

g. Click the "Publish Release" button to create the GitHub release!

6.2 Release Notes

6.2.1 v0.2.0

This initial alpha release introduces the xaitk-saliency toolkit for computing visual saliency heat-maps for input imagery over based on black-box operations.

Updates / New Features

CI

- Added properties file for SonarQube scans.
- Add CodeCov integration.

Documentation

- Updated the "Occlusion Saliency" notebook to flow smoother and include un-commentable RISE perturbation algorithm option. The narrative has been more explicitly tuned to follow an "application" narrative.
- Add miscellaneous documentation on how to run a local SonarQube scan and experimental documentation on setting up scanning as a CI workflow job.

Interfaces

- Add new interfaces in accordance to the v0.2 API draft.
  - Added to doc-strings to expand on detail around saliency heatmap return value range and meaning.
  - Updated image perturbation interface to function in a streaming iterator fashion instead of in-bulk as a means of performance optimization as well as to allow it to function on larger image sizes and larger perturbation quantities at the same time.
- Removed old interface classes "ImageSaliencyMapGenerator", "SaliencyBlackbox" and "ImageSaliencyAugmenter".

Implementations

- Add new occlusion based classifier scoring in accordance to the v0.2 API draft for ImageClassifierSaliencyMapGenerator.
- Add new RISE based perturbation algorithm in accordance to the v0.2 API draft for PerturbImage
- Add new similarity based scoring algorithm in accordance to the v0.2 API draft for ImageSimilaritySaliencyMapGenerator
- Remove old "stub" implementations in transitioning to the new API draft
  - Removed "LogitImageSaliencyAugmenter" implementation class.
  - Removed "LogitImageSaliencyMapGenerator" implementation class.
  - Removed old RISE implementation classes.
  - Removed old SBSM implementation classes.

Fixes

- Update Read the Docs documentation link in README
- Address various "code smells" as reported by SonarQube/SonarCloud.

6.2.2 v0.3.0

This minor release provides updates to example notebooks, interface naming updating for consistency, as well as added a "high level" interface for visual saliency map generation from black-box classifiers.

Updates / New Features

Documentation

- Add "SuperPixelSaliency" notebook to demonstrate use of arbitrary perturbation masks based on superpixels for saliency map generation.
- Add "VIAME_OcclusionSaliency" notebook demonstrating integration with VIAME toolkit based on saliency map generation for a fish classification task.
- Add "covid_classification" notebook demonstrating integration with MONAI based on saliency map generation for a COVID-19 X-ray classification task.
- Updated notebook demonstration for `SimilarityScoring` usage to better track the notebook structures across the repo.
- Introduce class naming philosophy in the `CONTRIBUTING.md` file.
- Updated notebooks to make use of `GenerateImageClassifierBlackboxSaliency` interface/implementations where appropriate.
- Updated wording of the `SuperPixelSaliency.ipynb` notebook as to cover a use-case where the `GenerateImageClassifierBlackboxSaliency` interface API is not appropriate, i.e. already have masks computed.
- Added support for doc building on Windows by adding a platform check so that "make.bat" is called for and not "make html".

Interfaces

- Update `PerturbImage` to only output perturbation masks, dropping physical image perturbation output. Since this output generation was the same across all known implementations, that part has been split out into a utility function.
- Added support for positive and negative saliency values as output by the saliency map generation interfaces.
- Updated `PerturbImage` interface to take in *numpy.ndarray* as the image data structure.
- Added new, higher-level `GenerateImageClassifierBlackboxSaliency` interface for transforming an image and black-box classifier into visual saliency maps.
- Renamed `ImageClassifierSaliencyMapGenerator` interface to be `GenerateClassifierConfidenceSaliency`.
- Renamed `ImageSimilaritySaliencyMapGenerator` interface to be `GenerateDescriptorSimilaritySaliency`.
- Renamed `ImageDetectionSaliencyMapGenerator` interface to be `GenerateDetectorProposalSaliency`.

Implementations

- Add `RISEScoring` implementation, with the ability to also compute a de-biased form of RISE with an optional input parameter.
- Add `RISEStack` implementation of the `GenerateImageClassifierBlackboxSaliency` interface as a simple way to invoke the combination of RISE component algorithms.
- Add `SlidingWindowStack` implementation of the `GenerateImageClassifierBlackboxSaliency` interface as a simple way to invoke the combination of the Sliding Window perturbation method and the occlusion-based based scoring method.

Misc.

- Update locked dependency versions to latest defined by abstract requirements.

Utils

- Masking
  - Added utility functions for occluded image generation that was previously duplicated across `PerturbImage` implementations. Added both batch and streaming versions of this utility.

Fixes

Documentation

- Fixed misspelled "miscellaneous" file.

Implementations

- Fix saliency map normalization in both `OcclusionScoring` as well as `SimilarityScoring` to disallow cross-class pollution in the norm.

Misc.

- Fixed up module naming inconsistencies.

6.2.3 v0.3.1

This patch release focuses on removing some listed dependencies that are in reality unused.

Fixes

CI

- Enable the running of CI for branches and PRs targeting release branches.

Package

- Remove required dependencies that are in fact not used.

6.2.4 v0.4.0

This minor release expands our documentation and examples pool. We additionally provide an the D-RISE implementation for the `GenerateDetectorProposalSaliency` interface.

Updates / New Features

CI

- Added workflow for test running *some* example notebooks.
- Update CodeCov action used to version 2.

Documentation

- Added text discussing white box methods to `introduction.rst`.
- Added some review process documentation.

- Add initial FAQ documentation file.
- Add background material for saliency maps to `introduction.rst`.
- Added API docs section, which includes descriptions of all interfaces.
- Added content to the `CONTRIBUTING.md` file on:
  - including notes here for added updates, features and fixes
  - Jupyter notebook CI workflow inclusion
- Add implementations section.
- Update example Jupyter notebooks to work with Google Colab.

Examples

- Add example notebook using classifier-based interfaces and implementations with scikit-learn on the MNIST dataset.
- Edited notebook examples.

Implementations

- Add `DRISEScoring` implementation of the `GenerateDetectorProposalSaliency` interface using detection output and associated occlusion masks.
- Add `SlidingRadial` implementation of the `PerturbImage` interface that slides radial occlusion areas across an image.

Tests

- Removed use of `unittest.TestCase` as it is not utilized directly in any way that PyTest does not provide.

Utilities

- Add type annotation, documentation and unit-tests for using image matrices as the fill option instead of just a solid color.
- Add `format_detection` helper function to form the input for `GenerateDetectorProposalSaliency` from separated components.
- Add example notebook showing the use of `SlidingRadial` perturbation and the use of `occlude_image_batch` with blurred-image alpha blending.

Fixes

Implementations

- Fixed `ValueError` messages raised in the `SimilarityScoring` implementation. Added unittests to check the raising and message content.

6.2.5 v0.5.0

Updates / New Features

CI

- Updated notebooks CI workflow to include notebook data caching.

Documentation

- Added text discussing black box methods to `introduction.rst`.
- Added a section to `introduction.rst` that describes the links between saliency algorithms and implementations.
- Edited all text.
- Update top-level `README.md` file to have more useful content.
- Update misc. doc on local SonarQube scanning.

Examples

- Add example notebook for saliency on Atari deep RL agent, including updates on top of the original work to normalize saliency maps and conform to our API standards.
- Add example demonstrating saliency map generation for COCO formatted serialized detections.
- Updated examples to all use a common data sub-directory when downloading or saving generated data.

Implementations

- Add `SquaredDifferenceScoring` implementation of the `GenerateClassifierConfidenceSaliency` interface that uses squared difference.
- Add `RandomGrid` implementation of `PerturbImage`. This generates masks of randomly occluded cells with a given size in pixels.

Utilities

- Add `gen_coco_sal` function to compute saliency maps for detections in a `kwcoco` dataset, with accompanying cli script `sal-on-coco-dets` which does this on a COCO formatted json file and writes saliency maps to disk.
- Add multi-threaded functionality to `occlude_image_batch` utility.

Containerization

- Added Dockerfile and compose file that create base xaitk_saliency image.

Fixes

Build

- Fix incorrect specification of actually-optional *papermill* in relation to its intended inclusion in the *example_deps* extra.
- Update patch version of Pillow transitive dependency locked in the `poetry.lock` file to address CVE-2021-23437.
- Update the developer dependency and locked version of ipython to address a security vulnerability.

Implementations

- Fix incorrect cosine similarity computation and mask inversion in implementation of `DRISEScoring` detector saliency.

Examples

- Updated example Jupyter notebooks with more consistent dependency checks and also fixed minor header formatting issues.

Tests

- Fix deprecation warnings around the use of `numpy.random.random_integers`.

Utilities

- Fix `xaitk_saliency.utils.detection.format_detection` to not upcast the data type output when `objectness is None`.
- Fix `xaitk_saliency.utils.masking.weight_regions_by_scalar` to not upcast the data type output when `inv_masks is True`.
- Update `xaitk_saliency.utils.masking.weight_regions_by_scalar` to not use fully vectorized operation which significantly improves efficiency.

6.2.6 v0.6.0

This minor release notably adds new high-level interfaces for black-box object detector and image similarity saliency generation. We provide some reference implementations for these interfaces, notably the D-RISE and SBSM algorithms, respectively.

Other improvements include the addition of more examples notebooks, improvements/fixes to existing implementations and a revision to the CLI object detection saliency generation tool.

See below for more details.

Updates / New Features

CI

- Added the ATARI example notebook to the list of notebooks to run during CI.

Documentation

- Update saliency algorithms table with perturbation-based saliency for reinforcement learning and add corresponding section to README.
- Added a lighter color version of the logo that will appear better is both light- and dark-theme contexts. The main README file has been updated to refer to this image.
- Added introductory sentence to the style sheet document.
- Updated the release process to be incrementally more comprehensive and now includes the specification of a `release` branch with better patch release instructions. This also now includes a step to update the version referenced in the xaitk.org source.

Examples

- Updated demo resource download links from Google Drive to data.kitware.com
- Added example using saliency to qualitatively compare two object detection models.
- Updated SimilarityScoring example to use new high-level image similarity saliency interface and follow new similarity interface inputs.

Interfaces

- Added new high-level interface for black-box object detector saliency, `GenerateObjectDetectorBlackboxSaliency`.
- Updated image similarity interface `GenerateDescriptorSimilaritySaliency` to accept multiple query images and compute a saliency map for each one, relative to the reference image.
- Added new high-level interface for image similarity saliency, `GenerateImageSimilarityBlackboxSaliency`.

Implementations

- Added three `GenerateObjectDetectorBlackboxSaliency` implementations: the generic `PerturbationOcclusion`, and two usable classes `DRISEStack` and `RandomGridStack`.
- Updated behavior of the `SlidingWindow` `PerturbImage` implementation. For a given stride, the number of masks generated is now agnostic to the window size.
- Updated `SimilarityScoring` to return [N x H x W] instead of [1 x H x W] saliency heatmaps matrix. This is inline with the similarity interface update.
- Added two implementations of `GenerateImageSimilarityBlackboxSaliency`: `PerturbationOcclusion` and `SBSMStack`.

Misc.

- Updated *poetry-core* build backend to version *1.0.8*, which now supports *pip* editable installs (*pip install -e .*).

Utils

- Updated COCO utility functions to use new high-level detector interface. *gen_coco_sal()* is now deprecated in exchange for *parse_coco_dset()* which parses a *kwcoco.CocoDataset* object into the inputs used with an implementation of *GenerateObjectDetectorBlackboxSaliency*.

Fixes

Dependency Versions

- Update pinned jupyter notebook transitive dependency version due to vulnerability warning.

Examples

- Fixed inconsistency of dependency package installs at the head of the `examples/SerializedDetectionSaliency.ipynb` notebook.

6.2.7 v0.6.1

This patch specializes the release CI workflow for this organization and repository as opposed to previously relying on the remote workflow.

Fixes

CI

- Fix the publish workflow to use appropriate values and version for the containing org and this repository.
- Update CI workflows to also run for `update-to-v*` branches.

6.2.8 v0.7.0

This minor release updates the mimumum supported python to *python = "^3.8"*, addresses dependency vulnerabilities, and updates typing to conform with current mypy and pytest standards.

Updates / New Features

Build

- New minimum supported python changed to *python = "^3.8"*.

CI

- Updated codecov action version to 3.
- Added explicit use of codecov token to facilitate successful coverage submission.
- Updated publish/release CI workflow.

Dependencies

- Updated `notebook` dependency due to a vulnerability alert.
- Periodic update of locked dep versions within abstract version constraints.
- Updated sphinx versions to fix local documentation building issue.
- Updated python minimum requirement to 3.8 (up from 3.6). This involved a number of updates and bifurcations of abstract requirements, an update to pinned versions for development/CI, and expansion of CI to cover python versions 3.10 and 3.11 (latest current release).

Fixes

Docs

- Added missing step to the release process about creating the release on GitHub's Releases section.

Examples

- Added a note to each example about restarting the runtime for compatibility with Colab, as well as a step to create a data directory if necessary.

CHAPTER

SEVEN

DESIGN DECISIONS

7.1 Concrete Dependencies and Updating

This package, like many others, has dependencies that need to be fulfilled for proper functionality to be satisfied. While these general requirements, found in the `pyproject.toml` file, are known as "abstract" requirements, we also choose to codify "concrete" requirements here via the `poetry.lock` file. This distinction is described well by [Stufft], using the `setuptools`-based resources which are parallel in use to the above respectively described files.

While xaitk-saliency is a library, we choose to retain concrete dependencies via the `poetry.lock` file to maintain consistency of environment across developers as well as CI processes. This falls in conceptual line with the "Developing Reusable Things or How Not to Repeat Yourself" section from [Stufft]. However, we are still a "library" and desire to make sure that we work with the "latest" versions of our listed abstract dependencies (within some reasonable time window). Currently, such concrete "version bumps" happen in the form of periodic update branches that update the `poetry.lock` file via a `poetry update` call. These updates are submitted as PRs to the upstream repository and allow the standard suite of CI checks to be performed to make sure the updated versions do not break anything. The timing of such updates are currently not concretely scheduled, nor are they specifically tied to events, but more on an "every so often" cadence that is relatively more frequent than versioned releases.

7.2 Image Format

We choose to use the `numpy.ndarray` data structure for our image representation in this toolkit. Earlier, we utilized the Pillow package's `PIL.Image.Image` data structure but encountered issues in certain use cases regarding large images, images with non-standard quantities of channels (e.g. > 3) or with imagery consisting of 12 or 16-bit valuation. Additionally, other popular and highly utilized packages in the Python community, like OpenCV, Scikit-Image, and PyTorch to name a few, utilize raw `numpy.ndarray` matrices as the container for image data.

CHAPTER

EIGHT

FREQUENTLY ASKED QUESTIONS

8.1 What is xaitk-saliency?

The xaitk-saliency package is an open source, explainable AI framework and toolkit for visual saliency algorithm interfaces and implementations, built for analytics and autonomy applications.

8.2 Who are the target audiences for this package?

This package is primarily a software API whose target audiences are:

- data scientists and researchers who want to apply XAI algorithms
- software engineers who want to integrate XAI algorithms into a larger system

8.3 Can I contribute to xaitk-saliency?

Of course! Before adding in your own contributions, we recommend reading the `CONTRIBUTING.md` file.

Additionally, you can contribute by helping review any outstanding branches. For guidelines on reviewing, see Review Process.

CHAPTER

NINE

MISCELLANEOUS DOCUMENTATION

9.1 Local SonarQube Testing

Follow the Try Out SonarQube documentation with the "From the Docker image" method. For a more robust docker-based setup, follow the install server docs from the "Installing SonarQube from the Docker Image" section. There is a docker-compose example there that may be adapted. The SonarQube DockerHub page contains information on host requirements needed to run the server ("Docker Host Requirements" section).

As part of the setup of the repository/project you will need to create a token for running scans with. The below assumes that you have saved this locally somewhere for reference, like the example file *~/sonarqube-local-token*.

From the above you will learn that you will need to run the `sonar-scanner` separately from running the local server. Documentation and references to acquire the scanner maybe found on their SonarScanner docs page.

Locally, we have had success running a script as per the following:

```
#! /bin/bash
SOURCE_DIR="$(cd "$(dirname "${BASH_SOURCE[0]}")" && pwd)"
docker run \
  --rm \
  --network host \
  -e SONAR_HOST_URL="http://localhost:9000" \
  -e SONAR_LOGIN="$(cat "$HOME"/sonarqube-local-token)" \
  -v "${SOURCE_DIR}:${SOURCE_DIR}" \
  -w "${SOURCE_DIR}" \
  sonarsource/sonar-scanner-cli
```

9.2 Setting Up xaitk-saliency with SonarCloud

9.2.1 Create a SonarCloud Organization

This will house our repository ("project") dashboard and is parallel to the GitHub organization.

- Go to SonarCloud
- Click the "+" button in the upper right near user drop-down.
- Select "Create a new organization".
- Follow instructions.

  - Part of this will be "Installing" the SonarCloud app at the organization level. Select either "All repositories" or select repositories to enable SonarCloud access to. Currently we have selected access only to the xaitk-saliency repository.

9.2.2 Create a Project for xaitk-saliency

- Go to SonarCloud.
- Click the "+" button in the upper right near user drop-down.
- Select the "XAITK" organization created above.
- Check our repo in the repos listed.

By default this will enable automatic scanning for common situations including PR submissions and master-branch updates. This is currently acceptable however this does not report coverage information due to the SonarCloud automatic scanning not having access to the unit test coverage reports generated during the GitHub Actions-based CI workflow. In the future we may want to switch to performing the SonarQube scanning action inside our CI workflow, however currently there is the hurdle where PRs from forks do not have the appropriate access to submit scan reports without learning private security tokens. The following section is provided as purely experimental/historical information.

9.2.3 Set Analysis Method to GitHub Actions

NOTE: *THIS IS NOT THE CURRENT CONFIGURATION.* This documentation here is notionally provided as it was written during experimentation and maybe has future value if we attempted again.

This is less setting something up but more deactivating the default setting to use SonarCloud Automatic Analysis. We chose to do this as SonarCloud automatic analysis has no ability to consider unittest code coverage, as this is only a byproduct of the CI unittests.

- Go to project page (e.g. https://sonarcloud.io/dashboard?id=XAITK_xaitk-saliency)
- Administration –> Analysis Method
- Toggle off "SonarCloud Automatic Analysis"

We "enabled" the GitHub action method by explicitly configuring a job in our GitHub CI workflow to use the SonarCloud GitHub Action to run the scanner and submit the report.

```yaml
jobs:

  # Add the following step to the end of the `unittests` job.
  unittests:
    steps:
      - name: Upload test coverage artifact
        uses: actions/upload-artifact@v2
        with:
          name: test-coverage-${{ matrix.python-version }}-${{ matrix.opt-extra }}
          path: coverage.xml

  # For analysis of codebase and submission to sonarcloud.io.
  sonarcloud:
    runs-on: ubuntu-latest

    # Requires coverage info generated from a previous unit-test run.
    needs: unittests
```

(continues on next page)

(continued from previous page)

```
  steps:
    - uses: actions/checkout@v2
      with:
        fetch-depth: 0
    - name: Pull coverage XML from test run
      uses: actions/download-artifact@v2
      with:
        # Just one version of the coverage. Sonar can't merge a matrix of
        # coverages yet (2021-06).
        name: test-coverage-3.7-
    - name: Munge coverage.xml source path for SonarCloud container use
      # Bridge the gap between what pytest-cov outputs and SonarCloud repo
      # source directory assumptions of "/github/workspace/".
      # We check that the coverage source line is as expected before sed call.
      run: |
        grep -E '^\s*<source>.*</source>$' coverage.xml
        sed -Ei 's|^(\s*)<source>.*</source>|\1<source>/github/workspace/xaitk_saliency
↪</source>|g' coverage.xml
    - name: SonarCloud Scan
      uses: sonarsource/sonarcloud-github-action@master
      env:
        GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
        SONAR_TOKEN: ${{ secrets.SONAR_TOKEN }}
```

This *requires* that a `SONAR_TOKEN` secret to be defined in the github *repository* settings to be accessed within the workflow. The application to the *repository* is important because PRs from forks will not have access the secret defined in the upstream repository, thus the job will fail for those fork-based PRs.

The value for this secret is from a SonarCloud personal security token (see below on how to make one of these). Currently, Paul Tunison (paul.tunison@kitware.com) holds the security token that is set to the `SONAR_TOKEN` secret in the upstream xaitk-saliency repository on GitHub. In the future this may be changed by a repo admin, as described below.

Create a Personal Security Token

- Go to SonarCloud.
- At the drop-down user option in the upper right –> select “My Account”.
- Click “Security” tab.
- Enter the descriptive label for the token in the editable box –> click “Generate”.
- Retain one-time-exposed value of token appropriately.

Set GitHub Repository `SONAR_TOKEN` Secret

- Go to the XAITK-Saliency repository page.
- Click on "Settings" –> "Secrets"
- If no existing `SONAR_TOKEN` secret, click on the "New repository secret" in the upper right.
  - This will open a new page to enter the name of the secret, which should be "SONAR_TOKEN" and a space to paste the value of the secret, which should be the token hash as generated above in *Create a personal security token*.
- Otherwise, update the existing secret value by clicking on the "Update" button to the right of the secret entry.
  - This will open a new page to enter a new value for the existing `SONAR_TOKEN` secret (i.e. cannot change the name of the secret). There should be a space to paste the value of the secret, which should be the token hash as generated above in *Create a personal security token*.

9.3 Style Sheet

This document lists the appropriate spelling of words and phrases to be used within this repo's code and documentation for consistency purposes.

- *black box* (noun) or *black-box* (adj); not blackbox (avoid)
- *D-RISE* not DRISE
- *end user* (noun) not end-user (preferably just "user")
- Headings – use Title Capitalization and follow with an intro sentence
- *heatmap* (noun/adj), not heat-map or heat map
- limited use of "please"
- *open source* (no hyphen)
- *PyTorch* vs Pytorch
- *set up* (verb) or *setup* (noun/adj); not set-up
- *TOC* should be included for each Jupyter notebook
- *use case* (noun) not use-case
- *visualizing* not visualising
- *white box* (noun) or *white-box* (adj); not whitebox (avoid)
- *xaitk-saliency* (all lowercase)

CHAPTER

TEN

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[Stufft] https://caremad.io/posts/2013/07/setup-vs-requirement/

INDEX