vision-explanation-methods

Release 0.0.7

Microsoft Corporation

Sep 14, 2023

CONTENTS:

CHAPTER

ONE

OVERVIEW

The vision-explanation-methods repository provides a set of tools and methods for generating saliency maps for object detection models, evaluating explanations, and error labeling.

The repository includes the following main components:

- **DRISE Runner**: This module provides a method for generating saliency maps for object detection models. It uses the DRISE (Detection-based Rise) method to generate saliency maps for a given image and model. The generated saliency maps can be used to understand which parts of the image are most important for the model's predictions.
- **Pointing Game**: This module provides a variety of explanation evaluation tools. It includes a method for visualizing highly salient pixels and calculating the overlap between salient pixels and ground truth bounding boxes.
- **Error Labeling**: This module provides a class for error labeling in object detection models. It includes methods for calculating Intersection over Union (IoU) scores and assigning error labels based on the IoU scores and class labels.
- **Setup**: The setup file for the vision-explanation-methods package includes the package metadata and dependencies.

The repository also includes a set of guidelines for contributing to the project, a code of conduct, and a license file.

For more detailed information about each component and how to use them, please refer to the respective sections in the documentation.

Getting Started with vision-explanation-methods

This documentation provides a guide on how to get started with the vision-explanation-methods package.

Installation

To install the vision-explanation-methods package, you need to run the setup file. The setup file for the vision-explanation-methods package is located at *python/setup.py*. This file contains metadata including the name and version of the package.

The dependencies required for the package are listed in the setup file. These include:

- numpy
- tqdm
- matplotlib<3.7.0
- ml_wrappers

To install the package, navigate to the directory containing the setup file and run the following command:

`bash python setup.py install `

Usage

The vision-explanation-methods package provides a variety of explanation evaluation tools for image explanation methods.

Generating Saliency Maps

The *DRISE_runner.py* file contains the method for generating saliency maps for object detection models. The *get_drise_saliency_map* function is used to generate the saliency map. This function takes in parameters such as the image location, model, number of classes, save name, and maximum figures.

The function returns a tuple of figure, location, and labels. The figure is a list of base64 encoded strings representing the saliency maps. The location is the path where the saliency maps are saved. The labels are a list of labels for the detected objects.

Error Labeling

The *error_labeling.py* file defines the Error Labeling Manager class. This class is used to label errors in the predictions of the model. The types of errors that can be labeled include class name errors, duplicate detection errors, background errors, and missing detection errors.

Contributing

Contributions to the vision-explanation-methods package are welcome. Please refer to the *CONTRIBUTING.md* file for guidelines on how to contribute to the project.

Code of Conduct

The vision-explanation-methods package has adopted the Microsoft Open Source Code of Conduct. Please refer to the *CODE_OF_CONDUCT.md* file for more information.

Support

For help and questions about using the vision-explanation-methods package, please refer to the *SUPPORT.md* file. This file provides information on how to file issues and get help.

License

The vision-explanation-methods package is licensed under the MIT License. Please refer to the *LICENSE.txt* file for more information.

CHAPTER

TWO

SUPPORTED MODELS AND DEPENDENCIES

2.1 Supported Models

The vision-explanation-methods package supports the following models:

- Faster R-CNN model with resnet50 backbone

2.2 Dependencies

The vision-explanation-methods package requires the following dependencies:

- numpy
- tqdm
- matplotlib<3.7.0
- ml_wrappers

2.3 Testing Dependencies

The vision-explanation-methods package requires the following dependencies for testing:

- pytest
- pytest-cov

2.4 Linting Dependencies

The vision-explanation-methods package requires the following dependencies for linting:

- autopep8==1.5.3
- flake8==4.0.1
- flake8-bugbear==21.11.29
- flake8-blind-except==0.1.1
- flake8-breakpoint
- flake8-builtins==1.5.3

- flake8-copyright==0.2.2
- flake8-docstrings==1.5.0
- flake8-logging-format==0.6.0
- flake8-pytest-style
- flake8-rst-docstrings==0.0.13
- isort

2.5 License

CHAPTER

THREE

CONTRIBUTING TO VISION-EXPLANATION-METHODS

This project welcomes contributions and suggestions. Most contributions require you to agree to a Contributor License Agreement (CLA) declaring that you have the right to, and actually do, grant us the rights to use your contribution. For details, visit https://cla.opensource.microsoft.com.

When you submit a pull request, a CLA bot will automatically determine whether you need to provide a CLA and decorate the PR appropriately (e.g., status check, comment). Simply follow the instructions provided by the bot. You will only need to do this once across all repos using our CLA.

This project has adopted the Microsoft Open Source Code of Conduct. For more information see the Code of Conduct FAQ or contact opencode@microsoft.com with any additional questions or comments.

3.1 Trademarks

This project may contain trademarks or logos for projects, products, or services. Authorized use of Microsoft trademarks or logos is subject to and must follow Microsoft's Trademark & Brand Guidelines. Use of Microsoft trademarks or logos in modified versions of this project must not cause confusion or imply Microsoft sponsorship. Any use of third-party trademarks or logos are subject to those third-party's policies.

3.2 Security

Microsoft takes the security of our software products and services seriously, which includes all source code repositories managed through our GitHub organizations. If you believe you have found a security vulnerability in any Microsoft-owned repository that meets Microsoft's definition of a security vulnerability, please report it to us as described below.

Please do not report security vulnerabilities through public GitHub issues. Instead, please report them to the Microsoft Security Response Center (MSRC) at https://msrc.microsoft.com/create-report. If you prefer to submit without logging in, send email to secure@microsoft.com. If possible, encrypt your message with our PGP key; please download it from the Microsoft Security Response Center PGP Key page.

Microsoft follows the principle of Coordinated Vulnerability Disclosure.

CHAPTER

FOUR

MICROSOFT OPEN SOURCE CODE OF CONDUCT

This project has adopted the Microsoft Open Source Code of Conduct.

Resources:

- Microsoft Open Source Code of Conduct
- Microsoft Code of Conduct FAQ
- Contact opencode@microsoft.com with questions or concerns

CHAPTER

FIVE

ERROR LABELING

The Error Labeling Manager class is defined in the `error_labeling.py` file in the `vision_explanation_methods/error_labeling` directory. This class is used to label errors in the vision-explanation-methods package.

5.1 Error Labeling Manager Class

The Error Labeling Manager class is used to label errors in the vision-explanation-methods package. It uses the Intersection over Union (IoU) threshold to determine the overlap between the predicted and true bounding boxes. The class also uses the ErrorLabelType Enum to define the types of error labels.

The ErrorLabelType Enum provides the following types of error labels:

- MISSING: The ground truth doesn't have a corresponding detection.
- BACKGROUND: The model predicted detections, but there was nothing there. This prediction must have a 0 IoU score with all ground truth detections.
- LOCALIZATION: The predicted class is correct, but the bounding box does not have sufficient overlap with the ground truth (based on the IoU threshold).
- CLASS_NAME: The predicted class is incorrect, but the bounding box is correct.
- CLASS_LOCALIZATION: Both the predicted class and bounding box are incorrect.
- DUPLICATE_DETECTION: The predicted class is correct, the bounding box is correct, but the IoU score is lower than another detection.
- MATCH: The bounding boxes overlap and the class names match.

The Error Labeling Manager class provides the following methods:

- `compute_error_labels()`: This method computes the error labels for the predicted and true bounding boxes.
- `compute_error_list()`: This method determines a complete list of errors encountered during prediction.

5.2 Error Labeling in Object Detection

In object detection, the Error Labeling Manager class is used to label errors in the predictions. The class uses the IoU threshold to determine the overlap between the predicted and true bounding boxes. The class also uses the ErrorLabelType Enum to define the types of error labels.

The ErrorLabelType Enum provides the following types of error labels:

- MISSING: The ground truth doesn't have a corresponding detection.

- BACKGROUND: The model predicted detections, but there was nothing there. This prediction must have a 0 IoU score with all ground truth detections.
- LOCALIZATION: The predicted class is correct, but the bounding box does not have sufficient overlap with the ground truth (based on the IoU threshold).
- CLASS_NAME: The predicted class is incorrect, but the bounding box is correct.
- CLASS_LOCALIZATION: Both the predicted class and bounding box are incorrect.
- DUPLICATE_DETECTION: The predicted class is correct, the bounding box is correct, but the IoU score is lower than another detection.
- MATCH: The bounding boxes overlap and the class names match.

The Error Labeling Manager class provides the following methods:

- `compute_error_labels()`: This method computes the error labels for the predicted and true bounding boxes.
- `compute_error_list()`: This method determines a complete list of errors encountered during prediction.

Note: The Error Labeling Manager class is used in the `test_error_labeling.py` file in the `tests` directory to test the error labeling in the vision-explanation-methods package.

CHAPTER

SIX

GENERATING SALIENCY MAPS FOR OBJECT DETECTION MODELS

This section provides an overview of the methods used to generate saliency maps for object detection models in the vision-explanation-methods package.

6.1 DRISE_runner.py

The DRISE_runner.py file contains the main function for generating saliency maps, *get_drise_saliency_map()*. This function takes in an image, a model, the number of classes, and a save name as parameters. It also accepts optional parameters for the number of masks, mask resolution, mask padding, device choice, and maximum figures.

The function begins by setting the device to either CUDA or CPU, depending on the availability of a GPU. If a model is not provided, the function loads a pre-trained Faster R-CNN model with a ResNet50 backbone. The image is then converted to a tensor and passed to the model for prediction.

The function then generates saliency scores using the DRISE method. The saliency scores are filtered to remove any scores containing NaN values. If no detections are found, the function raises a ValueError.

The function then generates a list of labels and a list of figures for each detection. Each figure is a visualization of the saliency map for the corresponding detection. The figures are saved as JPEG images and returned as base64 strings.

The function finally returns a tuple containing the list of figures, the save name, and the list of labels.

6.2 DRISE_saliency()

The *DRISE_saliency()* function in the drise.py file is used to compute the DRISE saliency map. This function takes in a model, an image tensor, target detections, and the number of masks as parameters. It also accepts optional parameters for mask resolution, mask padding, device, and verbosity.

The function begins by setting the mask padding and generating a list of mask records. Each mask record contains a mask and a list of affinity scores. The affinity scores are computed by comparing the target detections with the detections made on the masked image.

The function then fuses the masks based on their affinity scores to generate the saliency map.

6.3 PointingGame Class

The PointingGame class in the pointing_game.py file provides methods for evaluating the saliency maps. The *pointing_game()* method calculates the saliency scores for a given object detection prediction. The *calculate_gt_salient_pixel_overlap()* method calculates the overlap between the salient pixels and the ground truth bounding box.

6.4 Error Labeling

The error_labeling.py file provides methods for labeling the errors in the object detection predictions. The *ErrorLabelingManager* class manages the error labeling process. The *label_errors()* method labels the errors based on the intersection over union (IoU) scores between the ground truth and predicted bounding boxes.

Note: The vision-explanation-methods package uses the DRISE method for generating saliency maps. DRISE is a black box explainability method for object detection models. It generates saliency maps by occluding parts of the image with random masks and observing the effect on the model's predictions.

CHAPTER

SEVEN

USING DRISE FOR IMAGE EXPLANATION

DRISE is a black box explainability method for object detection. It generates saliency maps for object detection models.

The DRISE method is implemented in the `drise.py` file located in the `vision_explanation_methods/explanations` directory.

The DRISE method is used in the `DRISE_runner.py` file located in the `vision_explanation_methods` directory.

The DRISE method is used in the `pointing_game.py` file located in the `vision_explanation_methods/evaluation` directory.

7.1 DRISE Implementation

The DRISE method is implemented in the `drise.py` file. The implementation includes the following functions:

- `DRISE_saliency`: This function computes the DRISE saliency map. It takes as input an object detection model, an image tensor, a list of target detections, the number of masks to use for saliency, the resolution of the mask before scale up, the amount to pad the mask before cropping, the device to use to run the function, and a boolean indicating whether to print verbose output. It returns a list of tensors, one tensor for each image. Each tensor is of shape [D, 3, W, H], and [i ,3 W, H] is the saliency map associated with detection i.
- `DRISE_saliency_for_mlflow`: This function is similar to `DRISE_saliency`, but it is designed to work with the MLflow tracking service. It takes as input a model, an image tensor, a list of target detections, the number of masks to use for saliency, the resolution of the mask before scale up, the amount to pad the mask before cropping, the device to use to run the function, and a boolean indicating whether to print verbose output. It returns a list of tensors, one tensor for each image. Each tensor is of shape [D, 3, W, H], and [i ,3 W, H] is the saliency map associated with detection i.
- `generate_mask`: This function creates a random mask for image occlusion. It takes as input the lower resolution mask grid shape, the size of the image to be masked, and the amount to offset the mask. It returns an occlusion mask for the image, which has the same shape as the image.
- `fuse_mask`: This function masks an image tensor. It takes as input an image tensor and a mask for the image. It returns the masked image.
- `compute_affinity_scores`: This function computes the highest affinity score between two sets of detections. It takes as input a set of detections to get affinity scores for and a set of detections to score against. It returns a set of affinity scores associated with each detection.

7.2 Using DRISE

The DRISE method is used in the `DRISE_runner.py` file. The `get_drise_saliency_map` function in this file runs D-RISE on an image and visualizes the saliency maps. It takes as input the path of the image location, the model to use for D-RISE, the number of classes the model predicted, the path to save the output figure, the number of masks to use for saliency, the resolution of the mask before scale up, the amount to pad the mask before cropping, the device to use, and the maximum number of figures to generate. It returns a tuple of a list of Matplotlib figures, the path to where the output figure is saved, and a list of labels.

The DRISE method is also used in the `pointing_game.py` file. The `PointingGame` class in this file uses D-RISE to generate saliency maps for object detection models. The `pointing_game` function in this class finds saliency scores for the top 20% of salient pixels. It takes as input the filename of the image, the index of the detection to explain, the threshold for saliency, and the number of masks to use for saliency. It returns a saliency map for the image.

Note: The DRISE method requires the PyTorch library.

CHAPTER

EIGHT

SETUP AND INSTALLATION

The vision-explanation-methods package is a Python package developed by Microsoft Corporation. It provides a set of tools for explaining vision models, particularly object detection models.

8.1 Installation

The package can be installed using pip. The dependencies required for the package are:

- numpy
- tqdm
- matplotlib<3.7.0
- ml_wrappers

You can install the package and its dependencies using the following command:

```
pip install vision-explanation-methods
```

8.2 Setup

The setup file for the package is located at *python/setup.py*. This file contains the necessary information for the package setup, including the package name, version, description, author, license, and dependencies.

The package version is defined in the *vision_explanation_methods/version.py* file. The current version of the package is 0.0.7.

The package includes a README file (*README.md*) and a license file (*LICENSE.txt*). If the *LICENSE* file exists in the parent directory, it will be copied to *LICENSE.txt* during the setup process.

The package is classified under the following categories:

- Development Status :: 5 - Production/Stable
- Intended Audience :: Developers
- Intended Audience :: Science/Research
- License :: OSI Approved :: MIT License
- Programming Language :: Python :: 3
- Programming Language :: Python :: 3.7
- Programming Language :: Python :: 3.8

- Programming Language :: Python :: 3.9
- Topic :: Scientific/Engineering :: Artificial Intelligence
- Operating System :: Microsoft :: Windows
- Operating System :: MacOS
- Operating System :: POSIX :: Linux

The package is not safe for zipping.

8.3 Bug Reporting

If you encounter any bugs while using the package, you can report them using the bug report template provided in the *.github/ISSUE_TEMPLATE/bug_report.md* file. Please provide a clear and concise description of the bug, steps to reproduce the behavior, the expected behavior, and any relevant screenshots or additional context. Include information about your operating system, browser, Python version, and the version of the vision-explanation-methods package.

8.4 Feature Requests

If you have any suggestions for new features, you can submit them using the feature request template provided in the *.github/ISSUE_TEMPLATE/feature_request.md* file. Please provide a clear and concise description of the feature you want, any alternative solutions or features you've considered, and any additional context or screenshots.

8.5 Contributing

Contributions to the package are welcome. Please refer to the *README.md* file for information on how to contribute to the project.

8.6 License

The package is licensed under the MIT License. The full text of the license can be found in the *LICENSE.txt* file.

CHAPTER

NINE

API REFERENCE

This section provides a detailed reference to the classes and functions in the vision-explanation-methods package.

9.1 vision_explanation_methods

Module for creating explanations for vision models.

9.2 vision_explanation_methods.explanations

Module for image explanation methods.

9.3 vision_explanation_methods.explanations.drise

Implementation of DRISE.

A black box explainability method for object detection.

vision_explanation_methods.explanations.drise.**DRISE_saliency**(*model: vision_explanation_methods.explanations.common.Gene*
image_tensor: torch.Tensor,
target_detections:
List[vision_explanation_methods.explanations.common
number_of_masks: int, *mask_res:*
Tuple[int, int] = (16, 16),
mask_padding: Optional[int] =
None, *device: str = 'cpu'*, *verbose:*
bool = False) → List[torch.Tensor]

Compute DRISE saliency map.

Parameters

- **model** (*OcclusionModelWrapper*) – Object detection model wrapped for occlusion
- **target_detections** (*List of Detection Records*) – Baseline detections to get saliency maps for
- **number_of_masks** (*int*) – Number of masks to use for saliency
- **mask_res** – Resolution of mask before scale up
- **mask_padding** – How much to pad the mask before cropping

Type Optional int

Device Device to use to run the function

Type str

Returns A list of tensors, one tensor for each image. Each tensor is of shape [D, 3, W, H], and [i ,3 W, H] is the saliency map associated with detection i.

Return type List torch.Tensor

vision_explanation_methods.explanations.drise.**DRISE_saliency_for_mlflow**(*model*, *image_tensor: pandas.core.frame.DataFrame*, *target_detections: List[vision_explanation_methods.explana*, *number_of_masks: int*, *mask_res: Tuple[int, int] = (16, 16)*, *mask_padding: Optional[int] = None*, *device: str = 'cpu'*, *verbose: bool = False*) → List[torch.Tensor]

Compute DRISE saliency map.

Parameters

- **model** (*OcclusionModelWrapper*) – Object detection model wrapped for occlusion
- **target_detections** (*List of Detection Records*) – Baseline detections to get saliency maps for
- **number_of_masks** (*int*) – Number of masks to use for saliency
- **mask_res** – Resolution of mask before scale up
- **mask_padding** – How much to pad the mask before cropping

Type Optional int

Device Device to use to run the function

Type str

Returns A list of tensors, one tensor for each image. Each tensor is of shape [D, 3, W, H], and [i ,3 W, H] is the saliency map associated with detection i.

Return type List torch.Tensor

class vision_explanation_methods.explanations.drise.**MaskAffinityRecord**(*mask: torch.Tensor*, *affinity_scores: List[torch.Tensor]*)

Bases: object

Class for keeping track of masks and associated affinity score.

Parameters

- **mask** (*torch.Tensor*) – 3xHxW mask
- **affinity_scores** (*List of Tensors*) – Scores for each detection in each image associated with mask.

get_weighted_masks() → List[torch.Tensor]
Return the masks weighted by the affinity scores.

Returns Masks weighted by affinity scores - N tensors of shape Dx3xHxW, where N is the number of images in the batch, D, is the number of detections in an image (where D changes image to image)

Return type List of Tensors

to(*device: str*)
Move affinity record to accelerator.

Parameters **device** (*String*) – Torch string describing device, e.g. 'cpu' or 'cuda:0'

vision_explanation_methods.explanations.drise.**compute_affinity_scores**(*base_detections: vision_explanation_methods.explanations.co masked_detections: vision_explanation_methods.explanations.co* → torch.Tensor

Compute highest affinity score between two sets of detections.

Parameters

- **base_detections** (*Detection Record*) – Set of detections to get affinity scores for
- **masked_detections** (*Detection Record*) – Set of detections to score against

Returns Set of affinity scores associated with each detections

Return type Tensor of shape D, where D is number of base detections

vision_explanation_methods.explanations.drise.**convert_base64_to_tensor**(*b64_img: str*, *device: str*) → torch.Tensor

Convert base64 image to tensor.

Parameters

- **b64_img** (*str*) – Base64 encoded image
- **device** (*str*) – Torch string describing device, e.g. "cpu" or "cuda:0"

Returns Image tensor

Return type Tensor

vision_explanation_methods.explanations.drise.**convert_tensor_to_base64**(*img_tens: torch.Tensor*) → Tuple[str, Tuple[int, int]]

Convert image tensor to base64 string.

Parameters **img_tens** (*Tensor*) – Image tensor

Returns Base64 encoded image

Return type str

vision_explanation_methods.explanations.drise.**fuse_mask**(*img_tensor: torch.Tensor*, *mask: torch.Tensor*) → torch.Tensor

Mask an image tensor.

Parameters

- **img_tensor** (*Tensor*) – Image to be masked
- **mask** (*Tensor*) – Mask for image

Returns Masked image

Return type Tensor

vision_explanation_methods.explanations.drise.**generate_mask**(*base_size: Tuple[int, int]*, *img_size: Tuple[int, int]*, *padding: int*, *device: str*) → torch.Tensor

Create a random mask for image occlusion.

Parameters

- **base_size** (*Tuple (int, int)*) – Lower resolution mask grid shape
- **img_size** (*Tuple (int, int)*) – Size of image to be masked (hxw)
- **padding** (*int*) – Amount to offset mask
- **device** (*String*) – Torch string describing device, e.g. 'cpu' or 'cuda:0'

Returns Occlusion mask for image, same shape as image

Return type Tensor

vision_explanation_methods.explanations.drise.**saliency_fusion**(*affinity_records: List[*vision_explanation_methods.explanations.drise.M *device: str*, *normalize: Optional[bool] = True*, *verbose: bool = False*) → torch.Tensor

Create a fused mask based on the affinity scores of the different masks.

Parameters

- **affinity_records** (*List of affinity records*) – List of affinity records computed for mask
- **device** (*String*) – Torch string describing device, e.g. 'cpu' or 'cuda:0'
- **normalize** – Normalize the image by subtracting off the average affinity score (optional), defaults to true

Type bool

Returns List of saliency maps - one list of maps for each image in batch, and one map per detection in each image

Return type List of Tensors - one tensor for each image, and each tensor of shape Dx3xHxW, where D is the number of detections in that image.

9.4 vision_explanation_methods.evaluation

Module for evaluation.

9.5 vision_explanation_methods.evaluation.pointing_game

Defines a variety of explanation evaluation tools.

class vision_explanation_methods.evaluation.pointing_game.**PointingGame**(*model: Any*, *device='auto'*)

Bases: object

A class for the high energy pointing game.

calculate_gt_salient_pixel_overlap(*saliency_scores: List[torch.Tensor]*, *gt_bbox: List*)

Calculate percent of overlap between salient pixels and gt bbox.

Formula: number of salient pixels in the gt bbox / number of pixels in the gt bbox

Parameters

- **saliency_scores** (*List[Tensor]*) – 2D matrix representing the saliency scores of each pixel in an image
- **gt_bbox** (*List*) – bounding box for ground truth prediction

Returns return percent of salient pixel overlap with the ground truth

Return type Float

pointing_game(*imagelocation: str*, *index: int*, *threshold: float = 0.8*, *num_masks: int = 100*)

Calculate the saliency scores for a given object detection prediction.

The calculated value is a matrix of saliency scores. Values below the threshold are set to -1. The goal here is to filter out insignificant saliency scores, and identify highly salient pixels. That is why it is called a pointing game - we want to "point", i.e. identify, all highly salient pixels. That way we can easily determine if these highly salient pixels overlap with the gt bounding box.

Parameters

- **imagelocation** (*str*) – Path of the image location
- **index** (*int*) – Index of the desired object within the given image to evaluate
- **threshold** (*float*) – threshold between 0 and 1 to determine saliency of a pixel. If saliency score is below the threshold, then the score is set to -1
- **num_masks** (*int*) – number of masks to run drise with

Returns 2d matrix of highly salient pixels

Return type List[Tensor]

visualize_highly_salient_pixels(*img*, *saliency_scores*, *gt_bbox: Optional[List] = None*)

Create figure of highly salient pixels.

Parameters

- **img** (*PIL.Image*) – PIL test image
- **saliency_scores** (*List[Tensor]*) – 2D matrix representing the saliency scores of each pixel in an image
- **gt_bbox** (*List*) – bounding box for ground truth prediction. if none then no ground truth bounding box is drawn

Returns Overlay of the saliency scores on top of the image

Return type Figure

9.6 vision_explanation_methods.error_labeling

Module for error labeling.

9.7 vision_explanation_methods.error_labeling.error_labeling

Defines the Error Labeling Manager class.

class vision_explanation_methods.error_labeling.error_labeling.**ErrorLabelType**(*value*)

Bases: enum.Enum

Enum providing types of error labels.

If none, then the detection is not an error. It is a correct prediction.

BACKGROUND = 'background'

CLASS_LOCALIZATION = 'class_localization'

CLASS_NAME = 'class_name'

DUPLICATE_DETECTION = 'duplicate_detection'

LOCALIZATION = 'localization'

MATCH = 'match'

MISSING = 'missing'

class vision_explanation_methods.error_labeling.error_labeling.**ErrorLabeling**(*task_type: str*, *pred_y: list*, *true_y: list*, *iou_threshold: float = 0.5*)

Bases: object

Defines a wrapper class of Error Labeling for vision scenario.

Only supported for object detection at this point.

compute_error_labels()

Compute labels for errors in an object detection prediction.

Note: if a row does not have a match, that means that there is a missing gt detection

Returns 2d matrix of error labels

Return type NDArray

compute_error_list()

Determine a complete list of errors encountered during prediction.

Note that it is possible to have more errors than actual objects in an image (because we account for missing detections and duplicate detections).

Returns list of error labels

Return type list

9.8 vision_explanation_methods.DRISE_runner

Method for generating saliency maps for object detection models.

vision_explanation_methods.DRISE_runner.**get_drise_saliency_map**(*imagelocation: str*, *model: Optional[object]*, *numclasses: int*, *savename: str*, *nummasks: int = 25*, *maskres: Tuple[int, int] = (4, 4)*, *maskpadding: Optional[int] = None*, *devicechoice: Optional[str] = None*, *max_figures: Optional[int] = None*)

Run D-RISE on image and visualize the saliency maps.

Parameters

- **imagelocation** (*str*) – Path of the image location
- **model** (*PyTorch model*) – Input model for D-RISE. If None, Faster R-CNN model will be used.
- **numclasses** (*int*) – Number of classes model predicted
- **savename** (*str*) – Path of the saved output figure
- **nummasks** (*int*) – Number of masks to use for saliency
- **maskres** (*Tuple of ints*) – Resolution of mask before scale up
- **maskpadding** – How much to pad the mask before cropping
- **max_figures** – max figure # if memory limitations.

Type Optional int

Type Optional int

Returns Tuple of Matplotlib figure list, path to where the output figure is saved, list of labels

Return type Tuple of - list of Matplotlib figures, str, list

vision_explanation_methods.DRISE_runner.**get_instance_segmentation_model**(*num_classes: int*)

Load in pre-trained Faster R-CNN model with resnet50 backbone.

Parameters **num_classes** (*int*) – Number of classes model predicted

Returns Faster R-CNN PyTorch model

Return type PyTorch model

vision_explanation_methods.DRISE_runner.**plot_img_bbox**(*ax: matplotlib.axes._subplots.AxesSubplot*, *box: numpy.ndarray*, *label: str*, *color: str*)

Plot predicted bounding box and label on the D-RISE saliency map.

Parameters

- **ax** (*Matplotlib AxesSubplot*) – Axis on which the d-rise saliency map was plotted
- **box** (*numpy.ndarray*) – Bounding box the model predicted
- **label** (*str*) – Label the model predicted
- **color** (*single letter color string*) – Color of the bounding box based on predicted label

Returns Axis with the predicted bounding box and label plotted on top of d-rise saliency map

Return type Matplotlib AxesSubplot

9.9 vision_explanation_methods.version

Metadata including name and version of package.

CHAPTER

TEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

V

INDEX

P

S

T

V