001-getting-started

March 29, 2024

A library for benchmarking, developing and deploying deep learning anomaly detection algorithms

---

> NOTE: This notebook is originally created by @innat on Kaggle.

Anomalib: Anomalib is a deep learning library that aims to collect state-of-the-art anomaly detection algorithms for benchmarking on both public and private datasets. Anomalib provides several ready-to-use implementations of anomaly detection algorithms described in the recent literature, as well as a set of tools that facilitate the development and implementation of custom models. The library has a strong focus on image-based anomaly detection, where the goal of the algorithm is to identify anomalous images, or anomalous pixel regions within images in a dataset.

The library supports a number of image and video datasets for **benchmarking** and custom dataset support for **training/inference**. In this notebook, we will explore `anomalib` training a PADIM model on the `MVTec AD` bottle dataset and evaluating the model's performance.

0.1 Installing Anomalib

Installation can be done in two ways: (i) install via PyPI, or (ii) installing from sourc, both of which are shown below:

0.1.1 I. Install via PyPI

[ ]:
```
# Option - I: Uncomment the next line if you want to install via pip.
# %pip install anomalib
# %anomalib install -v
```

> NOTE:
>
> Although v1.0.0 is on PyPI, it may not be stable and may have bugs. It is therefore recommended to install from source.

0.1.2 II. Install from Source

This option would initially download anomalib repository from github and manually install `anomalib` from source, which is shown below:

[ ]:
```
# Option - II: Uncomment the next three lines if you want to install from the␣
↪source.
# !git clone https://github.com/openvinotoolkit/anomalib.git
# %cd anomalib
# %pip install .
# %anomalib install -v
```

Now let's verify the working directory. This is to access the datasets and configs when the notebook is run from different platforms such as local or Google Colab.

```
[ ]: import os
     from pathlib import Path

     from git.repo import Repo

     current_directory = Path.cwd()
     if current_directory.name == "000_getting_started":
         # On the assumption that, the notebook is located in
         #   ~/anomalib/notebooks/000_getting_started/
         root_directory = current_directory.parent.parent
     elif current_directory.name == "anomalib":
         # This means that the notebook is run from the main anomalib directory.
         root_directory = current_directory
     else:
         # Otherwise, we'll need to clone the anomalib repo to the␣
       ↪`current_directory`
         repo = Repo.clone_from(
             url="https://github.com/openvinotoolkit/anomalib.git",
             to_path=current_directory,
         )
         root_directory = current_directory / "anomalib"

     os.chdir(root_directory)
```

0.2 Imports

```
[ ]: from typing import Any

     import numpy as np
     from matplotlib import pyplot as plt
     from PIL import Image
     from torchvision.transforms import ToPILImage

     from anomalib import TaskType
     from anomalib.data import MVTec
     from anomalib.data.utils import read_image
     from anomalib.deploy import ExportType, OpenVINOInferencer
     from anomalib.engine import Engine
     from anomalib.models import Padim
```

0.3 Model

Currently, there are **13** anomaly detection models available in `anomalib` library. Namely,

- CFA
- CS-Flow
- CFlow
- DFKDE
- DFM
- DRAEM
- FastFlow
- Ganomaly
- Padim
- Patchcore
- Reverse Distillation
- R-KDE
- STFPM

In this tutorial, we'll be using Padim.

0.4 Dataset: MVTec AD

MVTec AD is a dataset for benchmarking anomaly detection methods with a focus on industrial inspection. It contains over **5000** high-resolution images divided into **15** different object and texture categories. Each category comprises a set of defect-free training images and a test set of images with various kinds of defects as well as images without defects. If the dataset is not located in the root datasets directory, anomalib will automatically install the dataset.

We could now import the MVtec AD dataset using its specific datamodule implemented in anomalib.

[81]:
```
datamodule = MVTec(num_workers=0)
datamodule.prepare_data()  # Downloads the dataset if it's not in the specified␣
 ↪`root` directory
datamodule.setup()  # Create train/val/test/prediction sets.

i, data = next(enumerate(datamodule.val_dataloader()))
print(data.keys())
```

```
dict_keys(['image_path', 'label', 'image', 'mask'])
```

Let's check the shapes of the input images and masks.

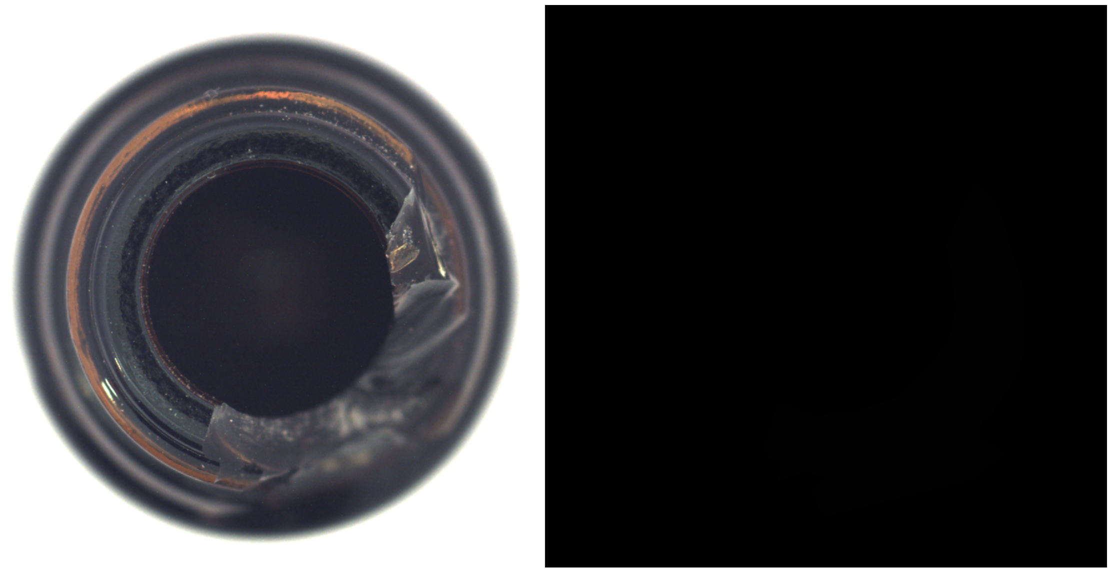
[80]:
```
print(data["image"].shape, data["mask"].shape)
```

```
torch.Size([32, 3, 900, 900]) torch.Size([32, 900, 900])
```

We could now visualize a normal and abnormal sample from the validation set.

[79]:
```
def show_image_and_mask(sample: dict[str, Any], index: int) -> Image:
    """Show an image with a mask.
    Args:
        sample (dict[str, Any]): Sample from the dataset.
        index (int): Index of the sample.

    Returns:
        Image: Output image with a mask.
    """
    # Load the image from the path
    image = Image.open(sample["image_path"][index])

    # Load the mask and convert it to RGB
    mask = ToPILImage()(sample["mask"][index]).convert("RGB")

    # Resize mask to match image size, if they differ
    if image.size != mask.size:
        mask = mask.resize(image.size)

    return Image.fromarray(np.hstack((np.array(image), np.array(mask))))


# Visualize an image with a mask
show_image_and_mask(data, index=0)
```

[79]:



0.5 Prepare Model

Let's create the Padim and train it.

[78]:
```python
# Get the model and datamodule
model = Padim()
datamodule = MVTec(num_workers=0)
```

[77]:
```python
# start training
engine = Engine(task=TaskType.SEGMENTATION)
engine.fit(model=model, datamodule=datamodule)
```

```
Trainer will use only 1 of 2 GPUs because it is running inside an interactive /
notebook environment. You may try to set `Trainer(devices=2)` but please note
that multi-GPU inside interactive / notebook environments is considered
experimental and unstable. Your mileage may vary.
GPU available: True (cuda), used: True
TPU available: False, using: 0 TPU cores
IPU available: False, using: 0 IPUs
HPU available: False, using: 0 HPUs
`Trainer(val_check_interval=1.0)` was configured so validation will run at the
end of the training epoch..
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0,1]

  | Name                  | Type                     | Params
-------------------------------------------------------------------
0 | model                 | PadimModel               | 2.8 M
1 | _transform            | Compose                  | 0
2 | normalization_metrics | MinMax                   | 0
3 | image_threshold       | F1AdaptiveThreshold      | 0
4 | pixel_threshold       | F1AdaptiveThreshold      | 0
5 | image_metrics         | AnomalibMetricCollection | 0
6 | pixel_metrics         | AnomalibMetricCollection | 0
-------------------------------------------------------------------
2.8 M     Trainable params
0         Non-trainable params
2.8 M     Total params
11.131    Total estimated model params size (MB)

Training: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

`Trainer.fit` stopped: `max_epochs=1` reached.
```

0.6 Validation

[76]:
```python
# load best model from checkpoint before evaluating
test_results = engine.test(
    model=model,
    datamodule=datamodule,
    ckpt_path=engine.trainer.checkpoint_callback.best_model_path,
)
```

```
Restoring states from the checkpoint path at /home/djameln/anomalib/lightning_lo
gs/version_144/checkpoints/epoch=0-step=7.ckpt
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0,1]
Loaded model weights from the checkpoint at /home/djameln/anomalib/lightning_log
s/version_144/checkpoints/epoch=0-step=7.ckpt
```

```
Testing: |          | 0/?  [00:00<?, ?it/s]
```

```
        Test metric                 DataLoader 0

        image_AUROC              0.9992063641548157
       image_F1Score             0.9921259880065918
        pixel_AUROC              0.9842503070831299
       pixel_F1Score             0.7291697859764099
```

[75]:
```
print(test_results)
```

```
[{'pixel_AUROC': 0.9842503070831299, 'pixel_F1Score': 0.7291697859764099,
'image_AUROC': 0.9992063641548157, 'image_F1Score': 0.9921259880065918}]
```

0.7 OpenVINO Inference

Now that we trained and tested a model, we could check a single inference result using OpenVINO inferencer object. This will demonstrate how a trained model could be used for inference.

Before we can use OpenVINO inference, let's export the model to OpenVINO format first.

[74]:
```
engine.export(
    model=model,
    export_type=ExportType.OPENVINO,
)
```

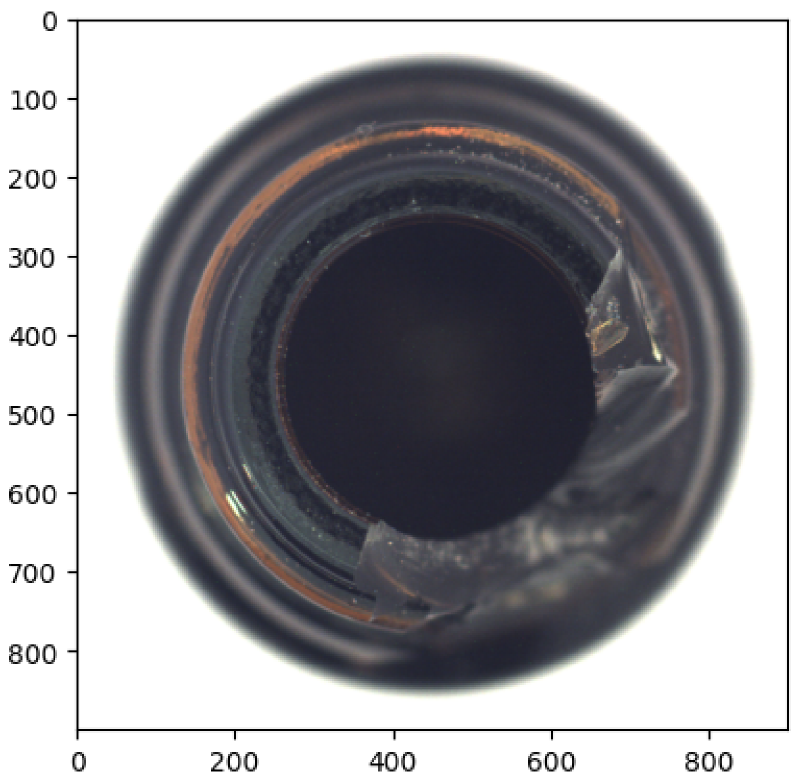
[74]:
```
PosixPath('/home/djameln/anomalib/weights/openvino/model.xml')
```

0.7.1 Load a Test Image

Let's read an image from the test set and perform inference using OpenVINO inferencer.

[73]:
```
image_path = root_directory / "datasets/MVTec/bottle/test/broken_large/000.png"
image = read_image(path="./datasets/MVTec/bottle/test/broken_large/000.png")
plt.imshow(image)
```

[73]:
```
<matplotlib.image.AxesImage at 0x7fd239fcc460>
```



0.7.2 Load the OpenVINO Model

By default, the output files are saved into `results` directory. Let's check where the OpenVINO model is stored.

[72]:
```
output_path = Path(engine.trainer.default_root_dir)
print(output_path)
```

```
/home/djameln/anomalib
```

[71]:
```
openvino_model_path = output_path / "weights" / "openvino" / "model.bin"
metadata = output_path / "weights" / "openvino" / "metadata.json"
print(openvino_model_path.exists(), metadata.exists())
```

```
True True
```

[70]:
```
inferencer = OpenVINOInferencer(
    path=openvino_model_path,  # Path to the OpenVINO IR model.
    metadata=metadata,  # Path to the metadata file.
    device="CPU",  # We would like to run it on an Intel CPU.
)
```

0.7.3 Perform Inference

Predicting an image using OpenVINO inferencer is as simple as calling `predict` method.

```
[69]: predictions = inferencer.predict(image=image_path)
```

where `predictions` contain any relevant information regarding the task type. For example, predictions for a segmentation model could contain image, anomaly maps, predicted scores, labels or masks.

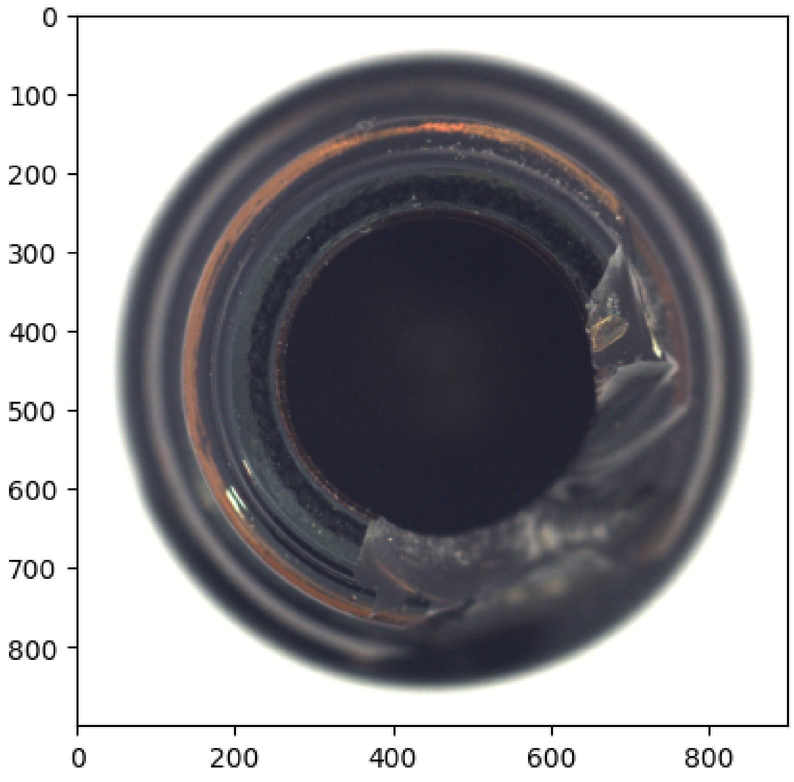
0.7.4 Visualizing Inference Results

```
[68]: print(predictions.pred_score, predictions.pred_label)
```

```
0.8962510235051898 True
```

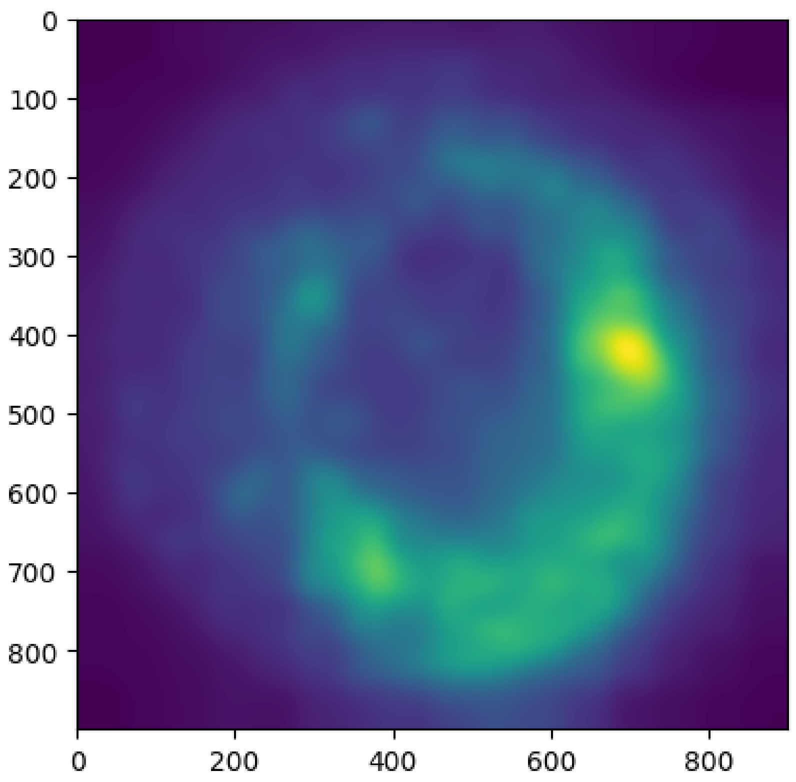
```
[67]: # Visualize the original image
plt.imshow(predictions.image)
```

```
[67]: <matplotlib.image.AxesImage at 0x7fd239f02080>
```



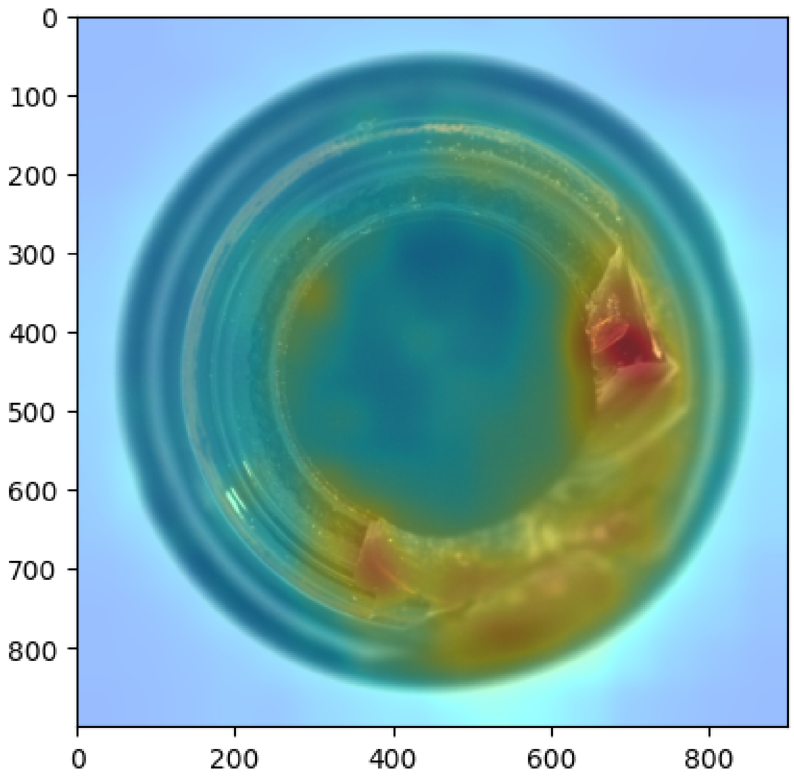
```
[66]: # Visualize the raw anomaly maps predicted by the model.
plt.imshow(predictions.anomaly_map)
```

[66]: <matplotlib.image.AxesImage at 0x7fd39c2540a0>



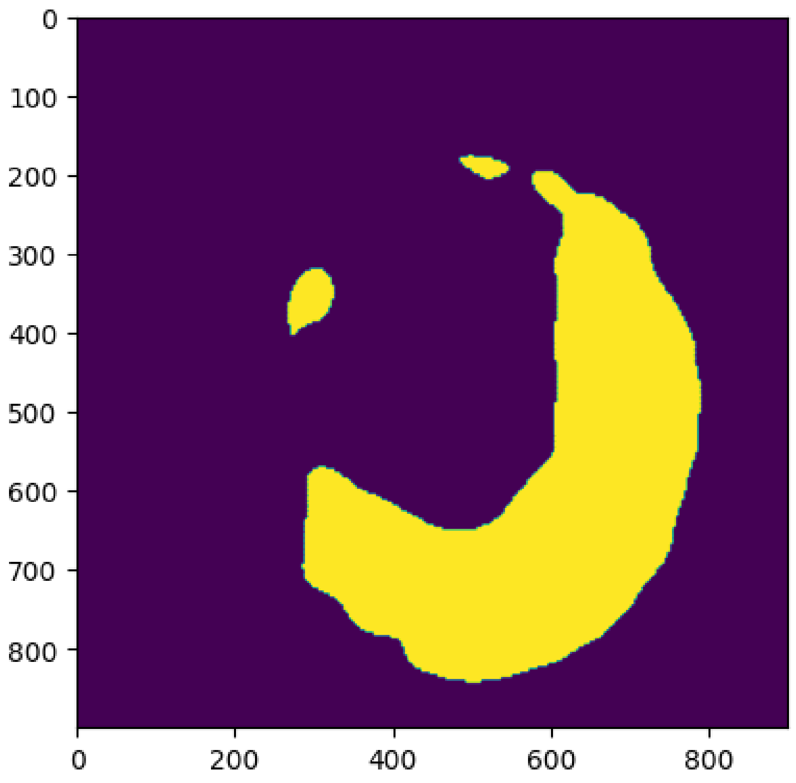
[65]:
```
# Visualize the heatmaps, on which raw anomaly map is overlayed on the original␣
  ↪image.
plt.imshow(predictions.heat_map)
```

[65]: <matplotlib.image.AxesImage at 0x7fd2a3a0ae30>



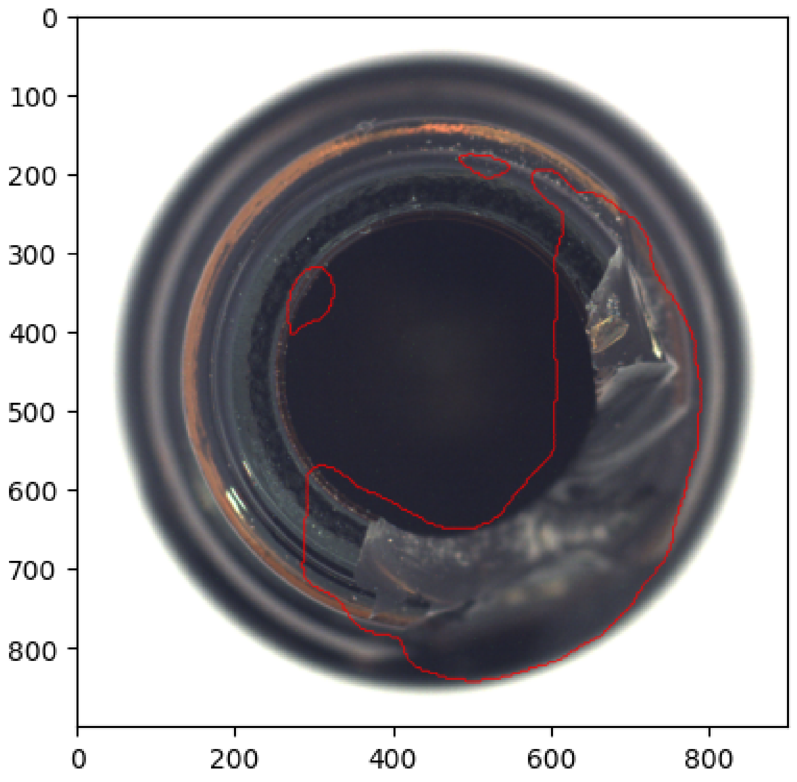
```
[64]: # Visualize the segmentation mask.
      plt.imshow(predictions.pred_mask)
```

```
[64]: <matplotlib.image.AxesImage at 0x7fd223f42020>
```



```
[63]: # Visualize the segmentation mask with the original image.
      plt.imshow(predictions.segmentations)
```

```
[63]: <matplotlib.image.AxesImage at 0x7fd1f83fb280>
```



This wraps the `getting_started` notebook. There are a lot more functionalities that could be explored in the library. Please refer to the documentation for more details.