
medigan Documentation

Release 1.0.0

['Richard Osuala, Grzegorz Skorupko, Noussair Lazrak']

Jul 11, 2023

DESCRIPTION

1	Overview	3
1.1	Description	3
1.1.1	Aim and Scope	4
1.1.2	Core Features	4
1.1.3	Architecture and Workflows	4
1.1.4	Notes	4
1.1.5	Issues	5
1.1.6	Links	5
1.2	Generative Models	5
1.2.1	00001_DCGAN_MMG_CALC_ROI	6
1.2.2	00002_DCGAN_MMG_MASS_ROI	6
1.2.3	00003_CYCLEGAN_MMG_DENSITY_FULL	7
1.2.4	00004_PIX2PIX_MMG_MASSES_W_MASKS	8
1.2.5	00005_DCGAN_MMG_MASS_ROI	9
1.2.6	00006_WGAN_GP_MMG_MASS_ROI	9
1.2.7	00007_INPAINT_BRAIN_MRI	10
1.2.8	00008_C-DCGAN_MMG_MASSES	13
1.2.9	00009_PGGAN_POLYP_PATCHES_W_MASKS	14
1.2.10	00010_FASTGAN_POLYP_PATCHES_W_MASKS	15
1.2.11	00011_SINGAN_POLYP_PATCHES_W_MASKS	17
1.2.12	00012_C-DCGAN_MMG_MASSES	18
1.2.13	00013_CYCLEGAN_MMG_DENSITY_OPTIMAM_MLO	19
1.2.14	00014_CYCLEGAN_MMG_DENSITY_OPTIMAM_CC	20
1.2.15	00015_CYCLEGAN_MMG_DENSITY_CSAW_MLO	21
1.2.16	00016_CYCLEGAN_MMG_DENSITY_CSAW_CC	22
1.2.17	00017_DCGAN_XRAY_LUNG_NODULES	23
1.2.18	00018_WGAN_GP_XRAY_LUNG_NODULES	23
1.2.19	00019_PGGAN_CHEST_XRAY	24
1.2.20	00020_PGGAN_CHEST_XRAY	24
1.2.21	00021_CYCLEGAN_BRAIN_MRI_T1_T2	25
1.2.22	00022_WGAN_CARDIAC_AGING	26
1.3	Code Examples	28
1.3.1	Install	28
1.3.2	Generate Images	28
1.3.3	Visualize Generative Model	29
1.3.4	Search for Generative Models	29
1.3.5	Rank Generative Models	30
1.4	Modules Overview	30
1.4.1	Generators	31
1.4.2	ModelExecutor	45

1.4.3	ModelVisualizer	49
1.4.4	ModelSelector	50
1.4.5	ModelContributor	54
1.4.6	ConfigManager	59
1.5	Full Code Documentation	61
1.5.1	medigan package	61
1.6	Tests	98
1.6.1	Setup medigan for running tests	99
1.6.2	Test 1: test_medigan_imports	99
1.6.3	Test 2: test_init_generators	99
1.6.4	Test 3: test_generate_methods	99
1.6.5	Test 4: test_generate_methods_with_additional_args	100
1.6.6	Test 5: test_get_generate_method	100
1.6.7	Test 6: test_search_for_models_method	100
1.6.8	Test 7: test_find_model_and_generate_method	100
1.6.9	Test 8: test_rank_models_by_performance	100
1.6.10	Test 9: test_find_and_rank_models_by_performance	100
1.6.11	Test 10: test_find_and_rank_models_then_generate_method	101
1.6.12	Test 11: test_get_models_by_key_value_pair	101
1.7	Model Contributions	101
1.7.1	Guide: Automated Model Contribution	101
1.7.2	Guide: Manual Model Contribution	102
1.7.3	Conventions that your model should follow	106
2	Indices	107
	Python Module Index	109
	Index	111

Welcome to *medigan*: **medical** generative (**a**dversarial) **n**etworks

Let's install *medigan* and generate a few synthetic images.

```
pip install medigan
```

```
from medigan import Generators

# model 1 is "00001_DCGAN_MMG_CALC_ROI"
Generators().generate(model_id=1, install_dependencies=True)
```


OVERVIEW

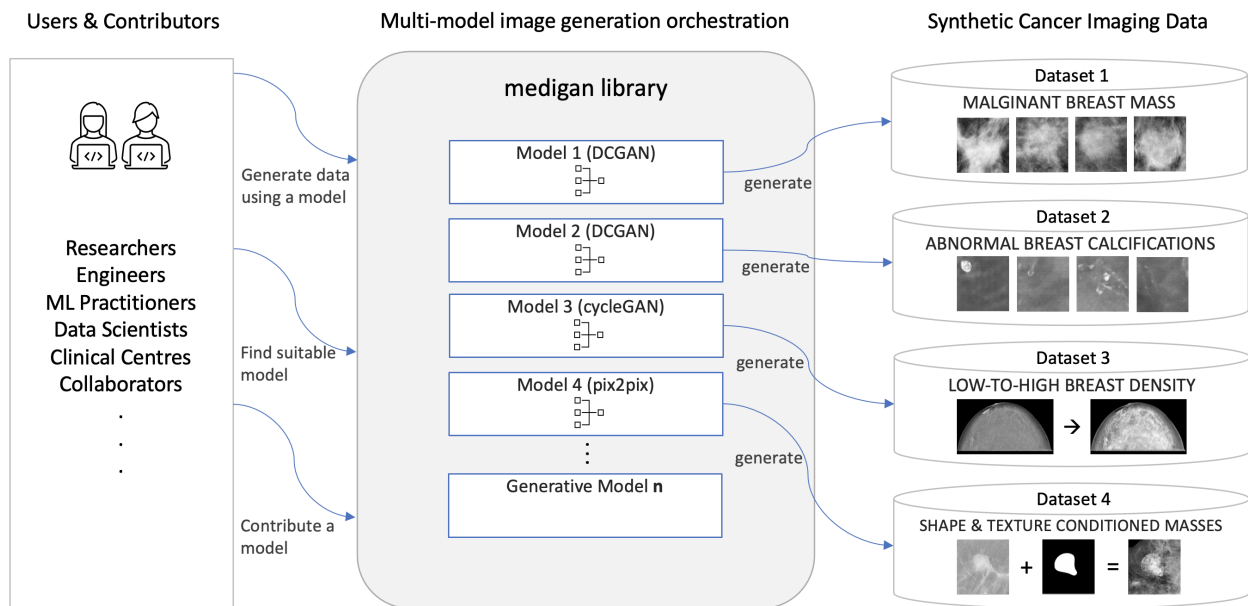


Fig. 1: Overview of *medigan* users, library, and models on the example of mammography cancer image dataset generation.

Note: Check out our [article](#) on *medigan* to get all the details.

1.1 Description

Welcome to *medigan*: **medical** generative (**adversarial**) networks

Table of Contents

- *Description*
 - *Aim and Scope*
 - *Core Features*

- *Architecture and Workflows*
- *Notes*
- *Issues*
- *Links*

1.1.1 Aim and Scope

medigan focuses on automating medical image dataset synthesis using GANs.

These datasets can again be used to train diagnostic or prognostic clinical models such as disease classification, detection and segmentation models.

Despite this current focus, *medigan*, is readily extendable to any type of modality and any type of generative model.

Note: More detail is available in the *medigan* article now available as [preprint](#). The article also includes experiments with insights into FID as generative model evaluation metric.

1.1.2 Core Features

- Researchers and ML-practitioners can conveniently use an existing model in *medigan* for synthetic data augmentation instead of having to train their own generative model each time.
- Users can search and find a model using search terms (e.g. “Mammography, 128x128, DCGAN”) or key value pairs (e.g. *key* = “modality”, *value* = “Mammography”)
- Users can explore the config and information (metrics, use-cases, modalities, etc) of each model in *medigan*
- Users can generate samples using a model
- Users can also get the `generate_method` of a model that they may want to use dynamically inside their dataloaders
- Model contributors can share and disseminate their generative models thereby augmenting their reach.

1.1.3 Architecture and Workflows

1.1.4 Notes

- Each model in *medigan* has its own dependencies listed in the `global.json` model metadata/config file. Setting `install_dependencies=True`, in the `generate()` and related methods, triggers an automatic installation of the respective model’s python dependencies (e.g. `numpy`, `torch`, etc), i.e., to the user’s active python environment.
- Running the `generate()` and related methods for synthetic data generation will trigger the download of the respective generative model to the user’s local directory from where the code is run (i.e. the current CLI path).

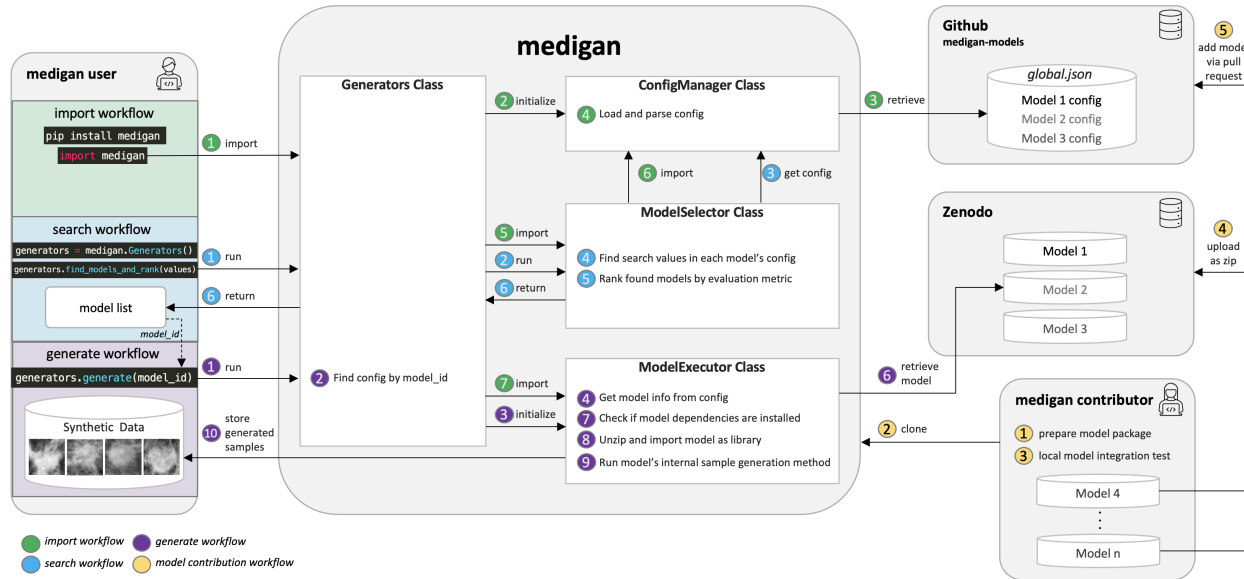


Fig. 2: Architectural overview including main workflows consisting of (a) library import and initialisation, (b) generative model search and ranking, (c) sample generation, and (d) generative model contribution.

1.1.5 Issues

In case you encounter problems while using *medigan* or would like to request additional features, please create a [new issue](#) and we will try to help.

1.1.6 Links

- [Github \(medigan library\)](#)
- [Test Pypi \(medigan library\)](#)

1.2 Generative Models

This section provides an overview of the generative models in *medigan*.

Find in the tables below for each model:

1. A **model_id**
2. A link to detailed documentation on **Zenodo**

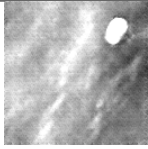
Further model information can be found in the [global.json](#) metadata.

Warning: Some of the model internal checkpoint loading functions may implicitly use the pickle module (e.g. `torch.load()`). Pickle is insecure: It is possible to construct malicious pickle data which will execute arbitrary code during unpickling ([example video](#)). While we do our best to analyse and test each model before Zenodo upload and *medigan* integration, we cannot provide a security guarantee. Be aware and run only models you trust. To further mitigate risks, we plan to integrate a malware scanning tool into *medigan*'s [CI pipeline](#).

1.2.1 00001_DCGAN_MMG_CALC_ROI

DCGAN Model for Mammogram Calcification Region of Interest Generation (Trained on INbreast).

Note: A deep convolutional generative adversarial network (DCGAN) that generates regions of interest (ROI) of mammograms containing benign and/or malignant calcifications. Pixel dimensions are 128x128. The DCGAN was trained on ROIs from the INbreast dataset (Moreira et al, 2012). The uploaded ZIP file contains the files `dcgan.pt` (model weights), `init.py` (image generation method and utils), a `README.md`, and the GAN model architecture (in pytorch) below the `/src` folder. Kernel size=6 used in DCGAN discriminator.

Output type	Modality	Model type	Output size	Base dataset	Output examples	model_id	Hosted on	Reference
Breast Calcification	mammography	dcgan	128x128	Inbreast		00001_DCGAN_MMG_CALC_ROI	ModelDB (5187714)	

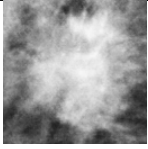
```
# create samples with this model
from medigan import Generators
Generators().generate(model_id="00001_DCGAN_MMG_CALC_ROI", image_size=128)

# model specific parameters
inputs= ["image_size: default=128, help=128 is the image size that works with the_
↪supplied checkpoint."]
```

1.2.2 00002_DCGAN_MMG_MASS_ROI

DCGAN Model for Mammogram Mass Region of Interest Generation (Trained on OPTIMAM)

Note: A deep convolutional generative adversarial network (DCGAN) that generates regions of interest (ROI) of mammograms containing benign and/or malignant masses. Pixel dimensions are 128x128. The DCGAN was trained on ROIs from the Optimam dataset (Halling-Brown et al, 2014). The uploaded ZIP file contains the files `malign_mass_gen` (model weights), and `init.py` (image generation method and pytorch GAN model architecture). Kernel size=6 used in DCGAN discriminator.

Output type	Modality	Model type	Output size	Base dataset	Output examples	model_id	Hosted on	Reference
Breast Mass	mammography	dcgan	128x128	Optimam		00002_DCGAN_MMG_MASS_ROI	ModelDB (5188557)	Alyafi et al (2019)

```
# create samples with this model
from medigan import Generators
Generators().generate(model_id="00002_DCGAN_MMG_MASS_ROI")
```

(continues on next page)

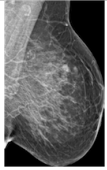
(continued from previous page)

```
# model specific parameters
inputs= []
```

1.2.3 00003_CYCLEGAN_MMG_DENSITY_FULL

CycleGAN Model for Low-to-High Breast Density Mammograms Translation (Trained on BCDR)

Note: A cycle generative adversarial network (CycleGAN) that generates mammograms with high breast density from an original mammogram e.g. with low-breast density. The CycleGAN was trained using normal (without pathologies) digital mammograms from BCDR dataset (Lopez, M. G., et al. 2012). The uploaded ZIP file contains the files CycleGAN_high_density.pth (model weights), **init.py** (image generation method and utils) and the GAN model architecture (in pytorch) below the /src folder.

Output type	Modality	Model type	Output size	Base dataset	Output examples	model_id	Hosted on	Reference
Breast Density Transfer	mammography	cycle-gan	1332x800	BCDR		00003_CYCLEGAN_MMG_DENSITY_FULL	Medigan (5547263)	Lucho et al (2022)

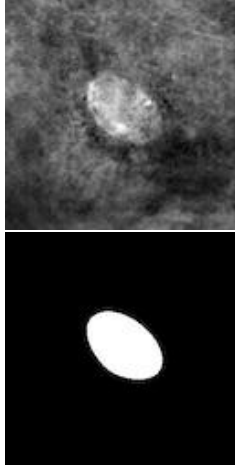
```
# create samples with this model
from medigan import Generators
Generators().generate(
    model_id="00003_CYCLEGAN_MMG_DENSITY_FULL",
    input_path="models/00003_CYCLEGAN_MMG_DENSITY_FULL/images",
    image_size=[1332, 800],
    gpu_id=0,
)

# model specific parameters
inputs= [
    "input_path: default=models/00003_CYCLEGAN_MMG_DENSITY_FULL/images, help=the path to
    ↪.png mammogram images that are translated from low to high breast density or vice versa
    ↪",
    "image_size: default=[1332, 800], help=list with image height and width. Images are
    ↪rescaled to these pixel dimensions.",
    "gpu_id: default=0, help=the gpu to run the model on.",
    "translate_all_images: default=False, help=flag to override num_samples in case the
    ↪user wishes to translate all images in the specified input_path folder."
]
```

1.2.4 00004_PIX2PIX_MMG_MASSES_W_MASKS

Generates synthetic patches given a random mask tiled with texture patches extracted from real images (Trained on BCDR)

Note: Generates synthetic patches given a random mask tiled with texture patches extracted from real images. The texture patches should be extracted from within the mass and outside the mass of a real image. Hence, some real ROIs are required to start with and its ideal for data augmentation purposes for mass segmentation.

Output type	Modality	Model type	Output size	Base dataset	Output examples	model_id	Hosted on	Reference
Breast Mass with Mask	mammography	pix2pix	256x256	BCDR		00004_PIX2PIX_MMG_MASSES_W_MASKS (7093759)		

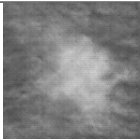
```
# create samples with this model
from medigan import Generators
Generators().generate(
    model_id="00004_PIX2PIX_MMG_MASSES_W_MASKS",
    input_path="models/00004_PIX2PIX_MMG_MASSES_W_MASKS/images",
    image_size=[256, 256],
    patch_size=[32, 32],
    shapes=['oval', 'lobulated'],
    ssim_threshold=0.2,
    gpu_id=0,
)

# model specific parameters
inputs= [
    "input_path: default=models/00004_PIX2PIX_MMG_MASSES_W_MASKS/images help=inputs that
    →are used in the pix2pix input image pool (e.g. for tiled image generation) ",
    "image_size: default=[256, 256] help=height and width of images.",
    "patch_size: default=[32, 32] help=height and width of patches (annotation size on
    →image).",
    "shapes: default=['oval', 'lobulated'] help=the type of the mask curve shapes
    →generated via bezier curves.",
    "ssim_threshold: default=0.2, help=the SSIM threshold that images must surpass to be
    →output.",
    "gpu_id: default=0 help=the gpu to run the model.",
]
```

1.2.5 00005_DCGAN_MMG_MASS_ROI

DCGAN Model for Mammogram MASS Patch Generation (Trained on BCDR)

Note: A deep convolutional generative adversarial network (DCGAN) that generates mass patches of mammograms. Pixel dimensions are 128x128. The DCGAN was trained on MMG patches from the BCDR dataset (Lopez et al, 2012). The uploaded ZIP file contains the files 500.pt (model weight), **init.py** (image generation method and utils), a requirements.txt, and the GAN model architecture (in pytorch) below the /src folder.

Output type	Modality	Model type	Output size	Base dataset	Output examples	model_id	Hosted on	Reference
Breast Mass	mammography	drgan	128x128	BCDR		00005_DCGAN_MMG_MASS_ROI	Zenodo (6555188)	Szafranska et al (2022)

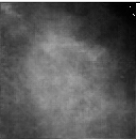
```
# create samples with this model
from medigan import Generators
Generators().generate(model_id="00005_DCGAN_MMG_MASS_ROI")

# model specific parameters
inputs= []
```

1.2.6 00006_WGANGP_MMG_MASS_ROI

WGAN-GP Model for Mammogram MASS Patch Generation (Trained on BCDR)

Note: A wasserstein generative adversarial network with gradient penalty (WGAN-GP) that generates mass patches of mammograms. Pixel dimensions are 128x128. The DCGAN was trained on MMG patches from the BCDR dataset (Lopez et al, 2012). The uploaded ZIP file contains the files 10000.pt (model weight), **init.py** (image generation method and utils), a requirements.txt, and the GAN model architecture (in pytorch) below the /src folder.

Output type	Modality	Model type	Output size	Base dataset	Output examples	model_id	Hosted on	Reference
Breast Mass	mammography	wgan-gp	128x128	BCDR		00006_WGANGP_MMG_MASS_ROI	Zenodo (6554713)	Szafranska et al (2022)

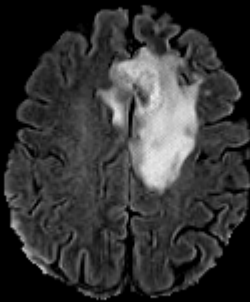
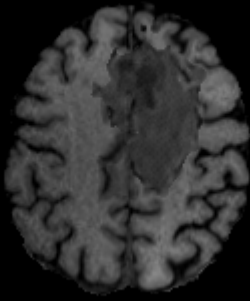
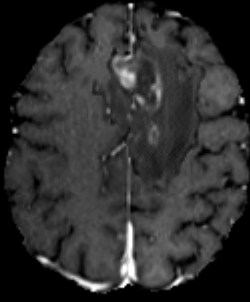
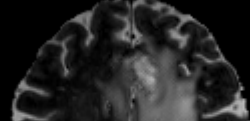
```
# create samples with this model
from medigan import Generators
Generators().generate(model_id="00006_WGANGP_MMG_MASS_ROI")

# model specific parameters
inputs= []
```

1.2.7 00007_INPAINT_BRAIN_MRI

Tumor Inpainting Model for Generation of Flair, T1, T1c, T2 Brain MRI Images (Trained on BRATS)

Note: A Generative adversarial network (GAN) for Inpainting tumors (based on concentric circle-based tumor grade masks) into multi-modal MRI images (Flair, T1, T1c, T2) with dimensions 256x256. Model was trained on BRATS MRI Dataset (Menze et al). For more information, see publication (<https://doi.org/10.1002/mp.14701>). Model comes with example input image folders. Apart from that, the uploaded ZIP file contains the model checkpoint files .pth (model weight), **init.py** (image generation method and utils), a requirements.txt, the MEDIGAN metadata.json. The proposed method synthesizes brain tumor images from normal brain images and concentric circles that are simplified tumor masks. The tumor masks are defined by complex features, such as grade, appearance, size, and location. Thus, these features of the tumor masks are condensed and simplified to concentric circles. In the proposed method, the user-defined concentric circles are converted to various tumor masks through deep neural networks. The normal brain images are masked by the tumor mask, and the masked region is inpainted with the tumor images synthesized by the deep neural networks. Also see original repository at: <https://github.com/KSH0660/BrainTumor>”

Output type	Modality	Model type	Output size	Base dataset	Output examples	model_id	hosted on	Reference
Brain Tumors on Flair, T1, T1c, T2 with Masks	brain MRI	in-paint GAN	256x256	ISBI 2018		00007	inpaint	BRAIN_MRI
								
								
								
1.2. Generative Models								11

```

# create samples with this model
from medigan import Generators
Generators().generate(
    model_id="00007_INPAINT_BRAIN_MRI",
    image_size=256,
    num_inpaints_per_sample=2,
    randomize_input_image_order=True,
    F_img_path=None,
    T1_img_path=None,
    T1c_img_path=None,
    T2_img_path=None,
    add_variations_to_mask=True,
    x_center=130,
    y_center=130,
    radius_1=10,
    radius_2=15,
    radius_3=30,
)

# model specific parameters
inputs= [
    "image_size: default=256, help=the size if height and width of the generated images.
    ↪",
    "num_inpaints_per_sample: default=2, help=the number of tumor inpaint images per MRI.
    ↪modality that is generated from the same input sample",
    "randomize_input_image_order: default=True, help=input image order is randomized.
    ↪This helps to not exclude input images if batch generation is used.",
    "F_img_path: default=None, help=The path to the folder were the input Flair MRI
    ↪images are stored.",
    "T1_img_path: default=None, help=The path to the folder were the input T1 MRI images
    ↪are stored.",
    "T1c_img_path: default=None, help=The path to the folder were the input T1c MRI
    ↪images are stored.",
    "T2_img_path: default=None, help=The path to the folder were the input T2 MRI images
    ↪are stored.",
    "add_variations_to_mask: default=True, help=This slightly varies the values of x_
    ↪center, y_center, radius_1, radius_2, radius_3. If True, the same segmentation masks
    ↪is still used to generate each of the 4 modality images. This is recommended as it
    ↪results in higher image diversity.",
    "x_center: default=130, help=the x coordinate of the concentric circle upon which
    ↪the binary mask, the tumor grade mask, and, ultimately, the generated images are based.
    ↪",
    "y_center: default=130, help=the y coordinate of the concentric circle upon which
    ↪the binary mask, the tumor grade mask, and, ultimately, the generated images are based.
    ↪",
    "radius_1: default=10, help=the radius of the first (inside second) of three
    ↪concentric circles (necrotic and non-enhancing tumor) upon which the binary mask, the
    ↪tumor grade mask, and, ultimately, the generated images are based.",
    "radius_2: default=15, help=the radius of the second (inside third) of three
    ↪concentric circles (enhancing tumor) upon which the binary mask, the tumor grade mask,
    ↪and, ultimately, the generated images are based.",
    "radius_3: default=30, help=the radius of the third of three concentric circles
    ↪(edema) upon which the binary mask, the tumor grade mask, and, ultimately, the
    ↪generated images are based."

```

(continues on next page)

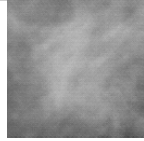
(continued from previous page)

]

1.2.8 00008_C-DCGAN_MMG_MASSES

Conditional DCGAN Model for Patch Generation of Mammogram Masses Conditioned on Biopsy Proven Malignancy Status (Trained on CBIS-DDSM)

Note: A class-conditional deep convolutional generative adversarial network that generates mass patches of mammograms that are conditioned to either be benign (1) or malignant (0). Pixel dimensions are 128x128. The Cond-DCGAN was trained on MMG patches from the CBIS-DDSM (Sawyer Lee et al, 2016). The uploaded ZIP file contains the files 1750.pt (model weight), `init.py` (image generation method and utils), a `requirements.txt`, a LICENSE file, the MEDI-GAN metadata, the used GAN training config file, a `test.sh` file to run the model, and two folders with a few generated images.

Output type	Modality	Model type	Output size	Base dataset	Output examples	model_id	Hosted on	Reference
Breast Mass (Mal/Benign)	mammography	c-dcgan	128x128	CBIS-DDSM		00008_C-DCGAN_MMG_MASSES	(6647349)	

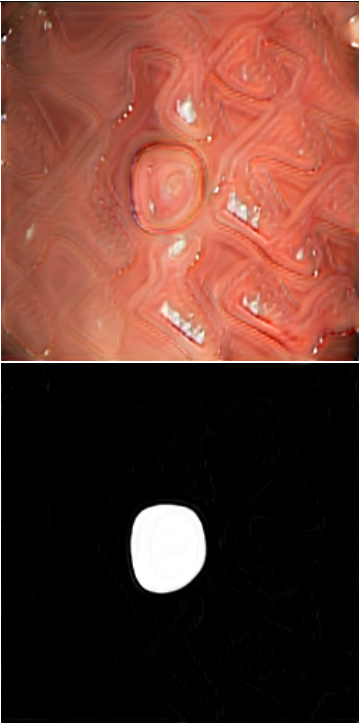
```
# create samples with this model
from medigan import Generators
Generators().generate(
    model_id="00008_C-DCGAN_MMG_MASSES",
    condition=None,
    z=None,
    is_cbisddsm_training_data=False, # False generates with weights of GAN trained on
    ↳cbis-ddsm testset
)

# model specific parameters
inputs= [
    "condition: default=None, help=Either 0, 1 or None. Condition indicates whether a
    ↳generated mass is malignant (0) or benign (1). If None, a balanced set of malignant
    ↳and benign tumor images is created.",
    "z: default=None, help=the input noise torch tensor for the generator. If None, this
    ↳option is ignored (e.g. random input vector generation)"
    "is_cbisddsm_training_data: default=True, help=Boolean indicating whether a GAN
    ↳checkpoint trained on the predefined test or train dataset (predefined by cbis-ddsm
    ↳dataset creators) should be used."
]
```

1.2.9 00009_PGGAN_POLYP_PATCHES_W_MASKS

PGGAN Model for Patch Generation of Polyps with Corresponding Segmentation Masks (Trained on HyperKvasir)

Note: A Progressively-growing generative adversarial network that generates a 4 dimensional output containing an RGB image (channels 1-3) and a segmentation mask (channel 4). The RGB images are images of polyps and the segmentation mask indicates the location and shape of the polyp on the image. Pixel dimensions are 256x256. The model was trained on gastrointestinal endoscopy imaging data from the HyperKvasir dataset by Borgli et al (2020, 'https://doi.org/10.1038/s41597-020-00622-y'). The uploaded ZIP file contains the files ProGAN_300000_g.model (model weight), **init.py** (image generation method and utils), a requirements.txt, a LICENSE file, the MEDIGAN metadata, the source code from the official repository ('https://github.com/vlbthambawita/singan-seg-polyp'), and a test.sh file to run the model, and a folder 'examples/' with a few generated images.

Out-put type	Modality	Model type	Output size	Base dataset	Output examples	model_id	Hosted on	Reference
Polyp with Mask	endoscopy	pggan	256x256	HyperKvasir		00009_PGGAN_POLYP_PATCHES_W_MASKS	odo (6653743)	bawita et al (2022)

```
# create samples with this model
Generators().generate(
    model_id="00009_PGGAN_POLYP_PATCHES_W_MASKS",
    gpu_id=None,
    channel=128,
    z_dim=128,
    pixel_norm=False,
    img_channels=4,
    tanh=False,
    step=6,
    alpha=1,
    save_option="image_only",
    num_fakes=1000,
```

(continues on next page)

(continued from previous page)

```

)

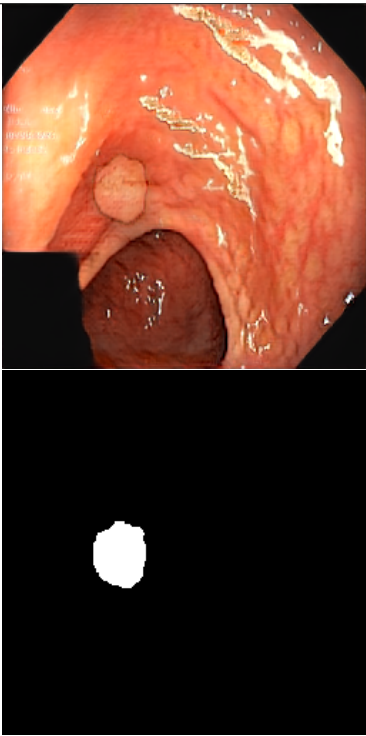
# model specific parameters
inputs=[
    "gpu_id: type=int, default=None, help=0 is the first gpu, 1 is the second gpu, etc.",
    "channel: type=int, default=128, help=determines how big the model is, smaller value_
↳ means faster training, but less capacity of the model",
    "z_dim: type=int, default=128, help=the initial latent vectors dimension, can be_
↳ smaller such as 64, if the dataset is not diverse",
    "pixel_norm: default=False, action=store_true, help=a normalization method inside_
↳ the model, you can try use it or not depends on the dataset",
    "img_channels: default=4, help=Number of channels in input data., for rgb images=3,_
↳ gray=1 etc.",
    "tanh: default=False, action=store_true, help=an output non-linearity on the output_
↳ of Generator, you can try use it or not depends on the dataset",
    "step: default=6, help=step to generate fake data. # can be 1 = 8, 2 = 16, 3 = 32, 4_
↳ = 64, 5 = 128, 6 = 256",
    "alpha: default=1, help=Progressive gan parameter to set, 0 or 1",
    "save_option: default=image_only, help=Options to save output, image_only, mask_only,
↳ image_and_mask, choices=[image_only,mask_only, image_and_mask]",
    "num_fakes: default=1000, help=Number of fakes to generate, type=int"
]

```

1.2.10 00010_FASTGAN_POLYP_PATCHES_W_MASKS

FastGAN Model for Patch Generation of Polyps with Corresponding Segmentation Masks (Trained on HyperKvasir)

Note: A Fast generative adversarial network (FastGAN) that generates a 4 dimensional output containing an RGB image (channels 1-3) and a segmentation mask (channel 4). FASTGAN is from the paper ‘Towards Faster and Stabilized GAN Training for High-fidelity Few-shot Image Synthesis’ in ICLR 2021. The RGB images are images of polyps and the segmentation mask indicates the location and shape of the polyp on the image. Pixel dimensions are 256x256. The model was trained on gastrointestinal endoscopy imaging data from the HyperKvasir dataset by Borgli et al (2020, ‘<https://doi.org/10.1038/s41597-020-00622-y>’). The uploaded ZIP file contains the files FastGAN_all_50000.pth (model weight), **init.py** (image generation method and utils), a requirements.txt, a LICENSE file, the MEDIGAN metadata, the source code from the repository (‘<https://github.com/vlbthambawita/singan-seg-polyp>’), and a test.sh file to run the model, and a folder ‘examples/’ with a few generated images.

Output type	Modality	Model type	Output size	Base dataset	Output examples	model_id	Hosted on	Reference
Polyp with Mask	endoscopy	fast-gan	256x256	Polyp-Kvasir		00010_FASTGAN_POLYP_PATCHES_W_MASKS	odo (6660711)	bawita et al (2022)

```

# create samples with this model
from medigan import Generators
Generators().generate(
    model_id="00010_FASTGAN_POLYP_PATCHES_W_MASKS",
    gpu_id=None,
    save_option="image_and_mask",
)

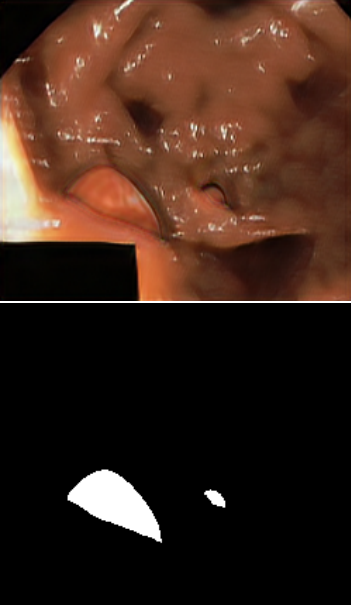
# model specific parameters
inputs=[
    "gpu_id: type=int, default=None, help=0 is the first gpu, 1 is the second gpu, etc.",
    "save_option: default=image_only, help=Options to save output, image_only, mask_only,
    → image_and_mask, choices=[image_only,mask_only, image_and_mask]"
]

```

1.2.11 00011_SINGAN_POLYP_PATCHES_W_MASKS

SinGAN Model for Patch Generation of Polyps with Corresponding Segmentation Masks (Trained on HyperKvasir)

Note: A SinGAN generative adversarial network that generates a 2dimensional output image tuple containing a 3-channel RGB image and a 3-channel segmentation mask. SinGAN is from the paper ‘SinGAN: Learning a Generative Model from a Single Natural Image’ in ICCV 2019. The width of the outputted images varies depending on the corresponding original image. The RGB images are images of polyps and the segmentation mask indicates the location and shape of the polyp on the image. Pixel dimensions are 213x256. The model was trained on gastrointestinal endoscopy imaging data from the HyperKvasir dataset by Borgli et al (2020, ‘<https://doi.org/10.1038/s41597-020-00622-y>’). The uploaded ZIP file contains the checkpoints in folder ‘SinGAN-Generated’ (model weights), **init.py** (image generation method and utils), a requirements.txt, a LICENSE file, the MEDIGAN metadata, the source code from the repository (‘<https://github.com/vlbthambawita/singan-seg-polyp>’), and a test.sh file to run the model, and a folder ‘examples/’ with a few generated images.

Out-put type	Modality	Model type	Out-put size	Base dataset	Output examples	model_id	Hosted on	Reference
Polyp with Mask	endoscopy	sin-gan	250x250	Hyper-Kvasir		00011_SINGAN_POLYP_PATCHES_W_MASKS	odo (6667944)	bawita et al (2022)

```
# create samples with this model
from medigan import Generators
Generators().generate(
    model_id="00011_SINGAN_POLYP_PATCHES_W_MASKS",
    model_files="models/00011_SINGAN_POLYP_PATCHES_W_MASKS/singan_seg_polyp/SinGAN-Generated",
    gen_start_scale=0,
    checkpoint_ids=None,
    multiple_checkpoints=False
)

# model specific parameters
inputs= [
    "model_files: default=models/00011_SINGAN_POLYP_PATCHES_W_MASKS/singan_seg_polyp/SinGAN-Generated help=the folder where the checkpoints are stored | ",
```

(continues on next page)

(continued from previous page)

```

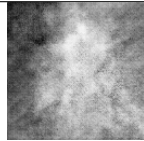
    "gen_start_scale: default=0 help=The start for scaling (progressively increasing
    ↳ generator input size) in SinGAN.",
    "checkpoint_ids: default=None help=A list of checkpoint ids that will be used for
    ↳ polyp generation. If None, all available checkpoints (i.e. 1000) or one random one
    ↳ (depending on 'multiple_checkpoints' arg) will be used.",
    "multiple_checkpoints: default=False help=A boolean indicating if all checkpoint_ids
    ↳ or one random one is used for generating images, but only in case 'checkpoint_ids
    ↳ '==None"
]

```

1.2.12 00012_C-DCGAN_MMG_MASSES

Conditional DCGAN Model for Patch Generation of Mammogram Masses Conditioned on Biopsy Proven Malignancy Status (Trained on BCDR)

Note: A class-conditional deep convolutional generative adversarial network that generates mass patches of mammograms that are conditioned to either be benign (1) or malignant (0). Pixel dimensions are 128x128. The Cond-DCGAN was trained on MMG patches from the BCDR dataset (Lopez et al, 2012). The uploaded ZIP file contains the files 1250.pt (model weight), `init.py` (image generation method and utils), a `requirements.txt`, a `LICENSE` file, the MEDIGAN metadata, the used GAN training config file, a `test.sh` file to run the model, and two folders with a few generated images.

Output type	Modality	Model type	Output size	Base dataset	Output examples	model_id	Hosted on	Reference
Breast Mass (Mal/Benign)	mammography	c-dcgan	128x128	BCDR		00012_C-DCGAN_MMG_MASSES	(6755693)	

```

# create samples with this model
from medigan import Generators
Generators().generate(
    model_id="00012_C-DCGAN_MMG_MASSES",
    condition=None,
    z=None,
)

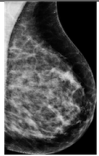
# model specific parameters
inputs= [
    "condition: default=None, help=Either 0, 1 or None. Condition indicates whether a
    ↳ generated mass is malignant (0) or benign (1). If None, a balanced set of malignant
    ↳ and benign tumor images is created.",
    "z: default=None, help=the input noise torch tensor for the generator. If None, this
    ↳ option is ignored (e.g. random input vector generation)"
]

```

1.2.13 00013_CYCLEGAN_MMG_DENSITY_OPTIMAM_MLO

CycleGAN Model for Low-to-High Breast Density Mammograms Translation of MLO VIEW (Trained on OPTIMAM)

Note: A cycle generative adversarial network (CycleGAN) that generates mammograms with high breast density from an original mammogram e.g. with low-breast density. The CycleGAN was trained using normal (without pathologies) digital mammograms from OPTIMAM dataset (Halling-Brown et al, 2014). The uploaded ZIP file contains the files CycleGAN_high_density.pth (model weights), **init.py** (image generation method and utils) and the GAN model architecture (in pytorch) below the /src folder.

Output type	Modality	Model type	Output size	Base dataset	Output examples	model_id	Hosted on	Reference
Breast Density Transfer MLO	mammography	cycle-gan	1332x800	OPTIMAM		00013_CYCLEGAN_MMG_DENSITY_OPTIMAM_MLO	Medigan (6818095)	et al (2022)

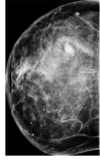
```
# create samples with this model
from medigan import Generators
Generators().generate(
    model_id="00013_CYCLEGAN_MMG_DENSITY_OPTIMAM_MLO",
    input_path="models/00013_CYCLEGAN_MMG_DENSITY_OPTIMAM_MLO/images",
    image_size=[1332, 800],
    gpu_id=0,
    translate_all_images=False,
    low_to_high=True,
)

# model specific parameters
inputs= [
    "input_path: default=models/00013_CYCLEGAN_MMG_DENSITY_OPTIMAM_MLO/images, help=the_
    ↳ path to .png mammogram images that are translated from low to high breast density or_
    ↳ vice versa",
    "image_size: default=[1332, 800], help=list with image height and width. Images are_
    ↳ rescaled to these pixel dimensions.",
    "gpu_id: default=0, help=the gpu to run the model on.",
    "translate_all_images: default=False, help=flag to override num_samples in case the_
    ↳ user wishes to translate all images in the specified input_path folder.",
    "low_to_high: default=True, help=if true, breast density is added. If false, it is_
    ↳ removed from the input image. A different generator of the cycleGAN is used based no_
    ↳ this flag."
]
```

1.2.14 00014_CYCLEGAN_MMG_DENSITY_OPTIMAM_CC

CycleGAN Model for Low-to-High Breast Density Mammograms Translation of CC VIEW (Trained on OPTIMAM)

Note: A cycle generative adversarial network (CycleGAN) that generates mammograms with high breast density from an original mammogram e.g. with low-breast density. The CycleGAN was trained using normal (without pathologies) digital mammograms from OPTIMAM dataset (Halling-Brown et al, 2014). The uploaded ZIP file contains the files CycleGAN_high_density.pth (model weights), **init.py** (image generation method and utils) and the GAN model architecture (in pytorch) below the /src folder.

Output type	Modality	Model type	Output size	Base dataset	Output examples	model_id	Hosted on	Reference
Breast Density Transfer CC	mammography	cycle-gan	1332x800	OPTIMAM		00014_CYCLEGAN_MMG_DENSITY_OPTIMAM_CC	medigan (6818103)	OPTIMAM CC cho et al (2022)

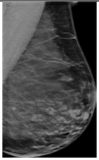
```
# create samples with this model
from medigan import Generators
Generators().generate(
    model_id="00014_CYCLEGAN_MMG_DENSITY_OPTIMAM_CC",
    input_path="models/00014_CYCLEGAN_MMG_DENSITY_OPTIMAM_CC/images",
    image_size=[1332, 800],
    gpu_id=0,
    translate_all_images=False,
    low_to_high=True,
)

# model specific parameters
inputs= [
    "input_path: default=models/00013_CYCLEGAN_MMG_DENSITY_OPTIMAM_MLO/images, help=the_
    ↳ path to .png mammogram images that are translated from low to high breast density or_
    ↳ vice versa",
    "image_size: default=[1332, 800], help=list with image height and width. Images are_
    ↳ rescaled to these pixel dimensions.",
    "gpu_id: default=0, help=the gpu to run the model on.",
    "translate_all_images: default=False, help=flag to override num_samples in case the_
    ↳ user wishes to translate all images in the specified input_path folder.",
    "low_to_high: default=True, help=if true, breast density is added. If false, it is_
    ↳ removed from the input image. A different generator of the cycleGAN is used based no_
    ↳ this flag."
]
```

1.2.15 00015_CYCLEGAN_MMG_DENSITY_CSAW_MLO

CycleGAN Model for Low-to-High Breast Density Mammograms Translation of MLO VIEW (Trained on CSAW)

Note: A cycle generative adversarial network (CycleGAN) that generates mammograms with high breast density from an original mammogram e.g. with low-breast density. The CycleGAN was trained using normal (without pathologies) digital mammograms from CSAW dataset (Dembrower et al., 2020, <https://doi.org/10.1007/s10278-019-00278-0>). The uploaded ZIP file contains the files CycleGAN_high_density.pth (model weights), **init.py** (image generation method and utils) and the GAN model architecture (in pytorch) below the /src folder.

Output type	Modality	Model type	Output size	Base dataset	Output examples	model_id	Hosted on	Reference
Breast Density Transfer MLO	mammography	cycle-gan	1332x800	CSAW		00015_CYCLEGAN_MMG_DENSITY_CSAW_MLO	(6818105)	CSAW_MLO cho et al (2022)

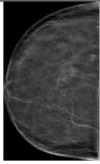
```
# create samples with this model
from medigan import Generators
Generators().generate(
    model_id="00015_CYCLEGAN_MMG_DENSITY_CSAW_MLO",
    input_path="models/00015_CYCLEGAN_MMG_DENSITY_CSAW_MLO/images",
    image_size=[1332, 800],
    gpu_id=0,
    translate_all_images=False,
    low_to_high=True,
)

# model specific parameters
inputs= [
    "input_path: default=models/00013_CYCLEGAN_MMG_DENSITY_OPTIMAM_MLO/images, help=the_
    ↳ path to .png mammogram images that are translated from low to high breast density or_
    ↳ vice versa",
    "image_size: default=[1332, 800], help=list with image height and width. Images are_
    ↳ rescaled to these pixel dimensions.",
    "gpu_id: default=0, help=the gpu to run the model on.",
    "translate_all_images: default=False, help=flag to override num_samples in case the_
    ↳ user wishes to translate all images in the specified input_path folder.",
    "low_to_high: default=True, help=if true, breast density is added. If false, it is_
    ↳ removed from the input image. A different generator of the cycleGAN is used based no_
    ↳ this flag."
]
```

1.2.16 00016_CYCLEGAN_MMG_DENSITY_CSAW_CC

CycleGAN Model for Low-to-High Breast Density Mammograms Translation of CC VIEW (Trained on CSAW)

Note: A cycle generative adversarial network (CycleGAN) that generates mammograms with high breast density from an original mammogram e.g. with low-breast density. The CycleGAN was trained using normal (without pathologies) digital mammograms from CSAW dataset (Dembrower et al., 2020, <https://doi.org/10.1007/s10278-019-00278-0>). The uploaded ZIP file contains the files CycleGAN_high_density.pth (model weights), **init.py** (image generation method and utils) and the GAN model architecture (in pytorch) below the /src folder.

Output type	Modal-ity	Model type	Out-put size	Base dataset	Output exam- ples	model_id	Hosted on	Refer- ence
Breast Den- sity Transfer CC	mam- mogra- phy	cy- cle- gan	1332x800	CSAW		00016_CYCLEGAN_MMG_DENSITY_CSAW_CC	(6818107)	CSAW-CC cho et al (2022)

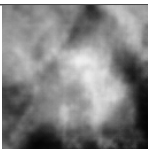
```
# create samples with this model
from medigan import Generators
Generators().generate(
    model_id="00016_CYCLEGAN_MMG_DENSITY_CSAW_CC",
    input_path="models/00016_CYCLEGAN_MMG_DENSITY_CSAW_CC/images",
    image_size=[1332, 800],
    gpu_id=0,
    translate_all_images=False,
    low_to_high=True,
)

# model specific parameters
inputs= [
    "input_path: default=models/00013_CYCLEGAN_MMG_DENSITY_OPTIMAM_MLO/images, help=the_
↳ path to .png mammogram images that are translated from low to high breast density or_
↳ vice versa",
    "image_size: default=[1332, 800], help=list with image height and width. Images are_
↳ rescaled to these pixel dimensions.",
    "gpu_id: default=0, help=the gpu to run the model on.",
    "translate_all_images: default=False, help=flag to override num_samples in case the_
↳ user wishes to translate all images in the specified input_path folder.",
    "low_to_high: default=True, help=if true, breast density is added. If false, it is_
↳ removed from the input image. A different generator of the cycleGAN is used based no_
↳ this flag."
]
```

1.2.17 00017_DCGAN_XRAY_LUNG_NODULES

DCGAN Model for Patch Generation of Lung Nodules (Trained on Node21)

Note: An unconditional deep convolutional generative adversarial network (DCGAN) that generates lung nodule regions-of-interest patches based on chest xray (CXR) images. The pixel dimension of the generated patches is 128x128. The WGAN_GP was trained on cropped patches from CXR images from the NODE21 dataset (Sogancioglu et al, 2021). The uploaded ZIP file contains the files model.pt (model weight), **init.py** (image generation method and utils), a requirements.txt, a LICENSE file, the MEDIGAN metadata.json file, the used GAN training config file, a test.sh file to run the model, and an /image folder with a few generated example images.

Output type	Modality	Model type	Output size	Base dataset	Output examples	model_id	Hosted on	Reference
Lung Nodules	chest x-ray	drgan	128x128	NODE21		00017_DCGAN_XRAY_LUNG_NODULES	medigan (6943691)	

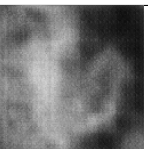
```
# create samples with this model
from medigan import Generators
Generators().generate(model_id="00017_DCGAN_XRAY_LUNG_NODULES")

# model specific parameters
inputs= []
```

1.2.18 00018_WGANGP_XRAY_LUNG_NODULES

WGAN_GP Model for Patch Generation of Lung Nodules (Trained on Node21)

Note: An unconditional wasserstein generative adversarial network with gradient penalty (WGAN_GP) that generates lung nodule regions-of-interest patches based on chest xray (CXR) images. The pixel dimension of the generated patches is 128x128. The WGAN_GP was trained on cropped patches from CXR images from the NODE21 dataset (Sogancioglu et al, 2021). The uploaded ZIP file contains the files model.pt (model weight), **init.py** (image generation method and utils), a requirements.txt, a LICENSE file, the MEDIGAN metadata.json file, the used GAN training config file, a test.sh file to run the model, and an /image folder with a few generated example images.

Output type	Modality	Model type	Output size	Base dataset	Output examples	model_id	Hosted on	Reference
Lung Nodules	chest x-ray	wgan-gp	128x128	NODE21		00018_WGANGP_XRAY_LUNG_NODULES	medigan (6943761)	

```
# create samples with this model
from medigan import Generators
Generators().generate(model_id="00018_WGANGP_XRAY_LUNG_NODULES")

# model specific parameters
inputs= []
```

1.2.19 00019_PGGAN_CHEST_XRAY

PGGAN Model for Generation of Chest XRAY (CXR) Images (Trained on ChestX-ray14 Dataset)

Note: An unconditional Progressively-growing generative adversarial network (PGGAN) that generates chest xray (CXR) images with pixel dimensions 1024x1024. The PGGAN was trained on CXR images from the ChestX-ray14 Dataset (Wang et al., 2017, Paper: <https://arxiv.org/pdf/1705.02315.pdf>, Data: <https://nihcc.app.box.com/v/ChestXray-NIHCC>). The uploaded ZIP file contains the files model.pt (model weight), **init.py** (image generation method and utils), a requirements.txt, a LICENSE file, the MEDIGAN metadata.json file, the used GAN training config file, a test.sh file to run the model, and an /image folder with a few generated example images.

Output type	Modality	Model type	Output size	Base dataset	Output examples	model_id	Hosted on	Reference
Chest Xray Images	chest x-ray	pggan	1024x1024	ChestX-ray14		00019_PGGAN_CHEST_XRAY	Medigan (6943803)	

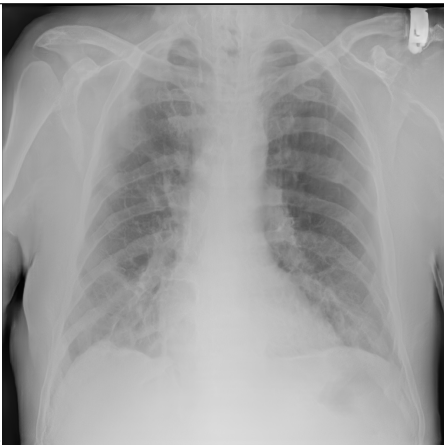
```
# create samples with this model
from medigan import Generators
Generators().generate(model_id="00019_PGGAN_CHEST_XRAY")

# model specific parameters
inputs= []
```

1.2.20 00020_PGGAN_CHEST_XRAY

PGGAN Model for Generation of Chest XRAY (CXR) Images (Trained on ChestX-ray14 Dataset)

Note: An unconditional Progressively-growing generative adversarial network (PGGAN) that generates chest xray (CXR) images with pixel dimensions 1024x1024. The PGGAN was trained on CXR images from the ChestX-ray14 Dataset (Wang et al., 2017, Paper: <https://arxiv.org/pdf/1705.02315.pdf>, Data: <https://nihcc.app.box.com/v/ChestXray-NIHCC>). The uploaded ZIP file contains the model weights checkpoint file, **init.py** (image generation method and utils), a requirements.txt, the MEDIGAN metadata.json file, a test.sh file to run the model, and an /image folder with a few generated example images.

Output type	Modality	Model type	Output size	Base dataset	Output examples	model_id	Hosted on	Reference
Chest Xray Images	chest x-ray	pg-gan	1024x1024	chestX-ray14		00020_PGGAN_CHEST_XRAY	Medo (7046280)	Le et al (2021)

```
# create samples with this model
from medigan import Generators
Generators().generate(
    model_id="00020_PGGAN_CHEST_XRAY",
    image_size=1024,
    resize_pixel_dim=None,
)

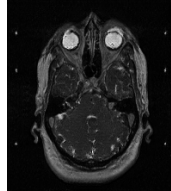
# model specific parameters
inputs = [
    "image_size: default=1024, help=the size if height and width of the generated images",
    ↪ "resize_pixel_dim: default=None, help=Resizing of generated images via the pillow_↪",
    ↪ "PIL image library."
]
```

1.2.21 00021_CYCLEGAN_BRAIN_MRI_T1_T2

CycleGAN Brain MRI T1-T2 translation (trained on CrossMoDA 2021 dataset)

Note: In recent years, deep learning models have considerably advanced the performance of segmentation tasks on Brain Magnetic Resonance Imaging (MRI). However, these models show a considerable performance drop when they are evaluated on unseen data from a different distribution. Since annotation is often a hard and costly task requiring expert supervision, it is necessary to develop ways in which existing models can be adapted to the unseen domains without any additional labelled information. In this work, we explore one such technique which extends the CycleGAN [2] architecture to generate label-preserving data in the target domain. The synthetic target domain data is used to train the nn-UNet [3] framework for the task of multi-label segmentation. The experiments are conducted and evaluated on the dataset [1] provided in the ‘Cross-Modality Domain Adaptation for Medical Image Segmentation’ challenge [23] for segmentation of vestibular schwannoma (VS) tumour and cochlea on contrast enhanced (ceT1) and high resolution (hrT2) MRI scans. In the proposed approach, our model obtains dice scores (DSC) 0.73 and 0.49 for tumour and cochlea respectively on the validation set of the dataset. This indicates the applicability of the proposed technique to

real-world problems where data may be obtained by different acquisition protocols as in [1] where hrT2 images are more reliable, safer, and lower-cost alternative to ceT1.

Output type	Modality	Model type	Output size	Base dataset	Output examples	model_id	Hosted on	Reference
Brain T1-T2 MRI Modality Transfer	brain MRI	cycle-gan	224x192	Cross-MoDA 2021		00021_CYCLEGAN_BRAIN_MRI_T1_T2	Brain-MRI-T1-T2 (7074555)	et al (2022)

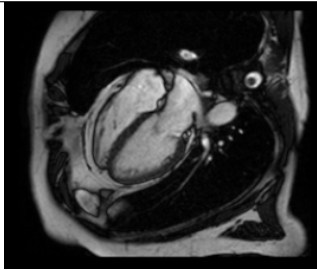
```
# create samples with this model
from medigan import Generators
Generators().generate(
    model_id="00021_CYCLEGAN_BRAIN_MRI_T1_T2",
    input_path= "models/00021_CYCLEGAN_BRAIN_MRI_T1_T2/inputs/T1", # or /T2
    image_size=[224, 192],
    gpu_id=0,
    translate_all_images=True,
    T1_to_T2=True,
)

# model specific parameters
inputs = [
    "input_path: default=models/00021_CYCLEGAN_BRAIN_MRI_T1_T2/inputs/T1, help=the path_
    ↳to .png brain MRI images that are translated from T1 to T2 or vice versa. ",
    "image_size: default=[224, 192], help=list with image height and width. ",
    "gpu_id: default=0, help=the gpu to run the model on.",
    "translate_all_images: default=False, help=flag to override num_samples in case the_
    ↳user wishes to translate all images in the specified input_path folder.",
    "T1_to_T2: default=True, help=if true, generator for T1 to T2 translation is used._
    ↳If false, the translation is done from T2 to T1 instead. Need to adjust input path in_
    ↳this case e.g. models/00021_CYCLEGAN_BRAIN_MRI_T1_T2/inputs/T2 instead of models/00021_
    ↳CYCLEGAN_BRAIN_MRI_T1_T2/inputs/T1. A different generator of the cycleGAN is used_
    ↳based on this flag."
]
```

1.2.22 00022_WGAN_CARDIAC_AGING

Conditional WGANGP Model for Cardiac Image generation with age offset (Trained on UK Biobank)

Note: A conditional wasserstein generative adversarial network with gradient penalty (WGAN_GP) that generates MRI cardiac images. The pixel dimension of the generated images is 256x256. The uploaded ZIP file contains the files model.ckpt (model weights), `init.py` (image generation method and utils), a requirements.txt, and the used GAN training config file. A sample_image.png is provided for example generation.

Output type	Modality	Model type	Output size	Base dataset	Output examples	model_id	Hosted on	Reference
Cardiac image	MRI	wgan	256x256	UK Biobank		00022_WGAN_CARDIAC_AGING	CoAgInG (7494368)	Impello et al (2022)

```

# create samples with this model
from medigan import Generators
Generators().generate(
    model_id="00022_WGAN_CARDIAC_AGING",
    image_paths_input=[
        "models/00022_WGAN_CARDIAC_AGING/sample_image.png",
        "models/00022_WGAN_CARDIAC_AGING/sample_image.png",
        "models/00022_WGAN_CARDIAC_AGING/sample_image.png"
    ],
    aging_input=[
        -4,
        10,
        2
    ],
    data_type="2d",
    view="la",
    subcat="2ch",
)

# model specific parameters
inputs= [
    "input_image_paths: default=[\"models/00022_WGAN_CARDIAC_AGING/sample_image.png\"]",
    "help=List of image paths to apply aging. One channel .png images must be provided.",
    "aging_input: default=[-4] help=List of age offset values for each image.",
    "data_type: default=\"2d\" help=",
    "view: default=\"la\" help=",
    "subcat: default=\"2ch\" help="
]

```

1.3 Code Examples

Table of Contents

- *Code Examples*
 - *Install*
 - *Generate Images*
 - *Visualize Generative Model*
 - *Search for Generative Models*
 - *Rank Generative Models*

1.3.1 Install

Install *medigan* library from pypi (or github).

```
pip install medigan
```

Import *medigan* and initialize Generators

```
from medigan import Generators
generators = Generators()
```

1.3.2 Generate Images

Generate 10 samples using one (model 1 is *00001_DCGAN_MMG_CALC_ROI*) of the *medigan models* from the config.

install_dependencies signals to medigan that the user wishes to automatically install all the python dependencies (e.g. numpy, torch, etc) required to run this model (i.e. to the user's active python environment).

```
generators.generate(model_id=1, num_samples=10, install_dependencies=True)
```

Get the model's generate method and run it to generate 3 samples

```
# model 1 is "00001_DCGAN_MMG_CALC_ROI"
gen_function = generators.get_generate_function(model_id=1, num_samples=3)
gen_function()
```

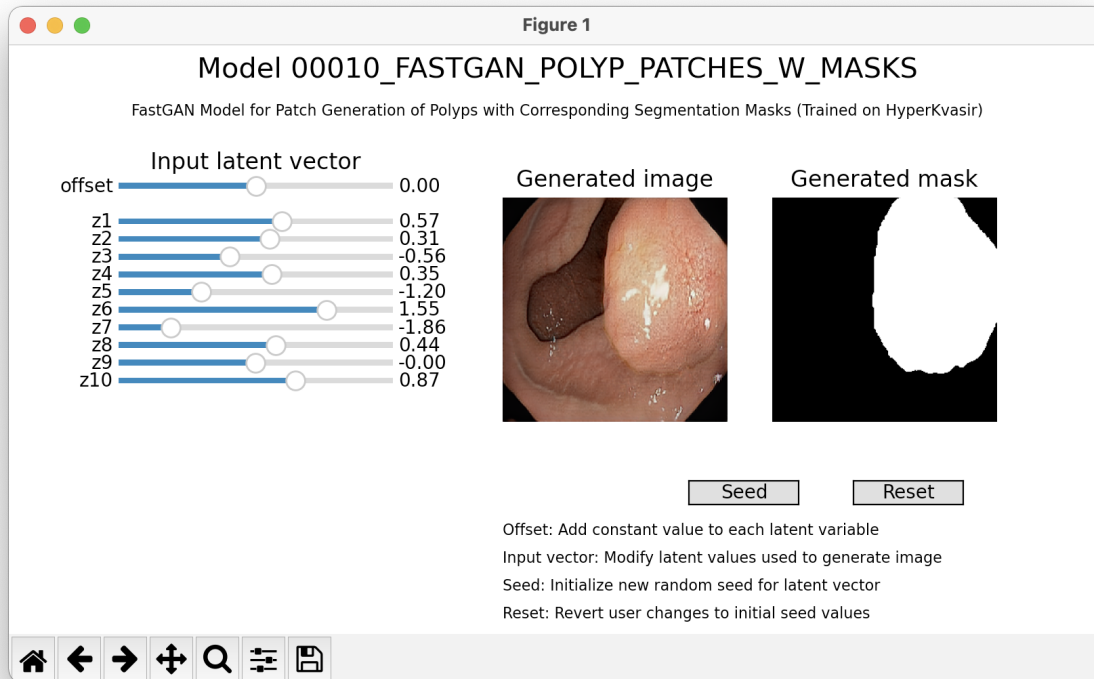
Get the model's synthetic data as torch dataloader with 3 samples

```
# model 4 is "00004_PIX2PIX_MMG_MASSES_W_MASKS"
dataloader = generators.get_as_torch_dataloader(model_id=4, num_samples=3)
```

1.3.3 Visualize Generative Model

Displays an interactive visual interface for exploration of applicable models.

```
# model 10 is "00010_FASTGAN_POLYP_PATCHES_W_MASKS"
generators.visualize(10)
```



1.3.4 Search for Generative Models

Find all models that contain a specific key-value pair in their model config.

```
key = "modality"
value = "Full-Field Mammography"
found_models = generators.get_models_by_key_value_pair(key1=key, value1=value, is_case_
↳ sensitive=False)
print(found_models)
```

Create a list of search terms and find the models that have these terms in their config.

```
values_list = ['drgan', 'Mammography', 'inbreast']
models = generators.find_matching_models_by_values(values=values_list, target_values_
↳ operator='AND', are_keys_also_matched=True, is_case_sensitive=False)
print(f'Found models: {models}')
```

Create a list of search terms, find a model and generate

```
values_list = ['dcgan', 'mMg', 'ClF', 'modalities', 'inbreast']
generators.find_model_and_generate(values=values_list, target_values_operator='AND', are_
↳keys_also_matched=True, is_case_sensitive=False, num_samples=5)
```

1.3.5 Rank Generative Models

Rank the models by a performance metric and return ranked list of models

```
ranked_models = generators.rank_models_by_performance(metric="SSIM", order="asc")
print(ranked_models)
```

Find the models, then rank them by a performance metric and return ranked list of models

```
ranked_models = generators.find_models_and_rank(values=values_list, target_values_
↳operator='AND', are_keys_also_matched=True, is_case_sensitive=False, metric="SSIM",
↳order="asc")
print(ranked_models)
```

Find the models, then rank them, and then generate samples with the best ranked model.

```
generators.find_models_rank_and_generate(values=values_list, target_values_operator='AND
↳', are_keys_also_matched=True, is_case_sensitive=False, metric="SSIM", order="asc",
↳num_samples=5)
```

1.4 Modules Overview

Table of Contents

- *Modules Overview*
 - *Generators*
 - *ModelExecutor*
 - *ModelVisualizer*
 - *ModelSelector*
 - *ModelContributor*
 - *ConfigManager*

1.4.1 Generators

medigan.generators Module

Base class providing user-library interaction methods for config management, and model selection and execution.

Classes

<code>ConfigManager([config_dict, ...])</code>	<i>ConfigManager</i> class: Downloads, loads and parses medigan's config json as dictionary.
<code>DataLoader(dataset[, batch_size, shuffle, ...])</code>	Data loader.
<code>Dataset(*args, **kwargs)</code>	An abstract class representing a <i>Dataset</i> .
<code><i>Generators</i>([config_manager, model_selector, ...])</code>	<i>Generators</i> class: Contains medigan's public methods to facilitate users' automated sample generation workflows.
<code>ModelContributor(model_id, init_py_path)</code>	<i>ModelContributor</i> class: Contributes a user's local model to the public medigan library
<code>ModelExecutor(model_id, execution_config[, ...])</code>	<i>ModelExecutor</i> class: Find config links to download models, init models as python packages, run generate methods.
<code>ModelSelector([config_manager])</code>	<i>ModelSelector</i> class: Given a config dict, gets, searches, and ranks matching models.
<code>ModelVisualizer(model_executor, config)</code>	<i>ModelVisualizer</i> class: Visualises synthetic data through a user interface.
<code>SyntheticDataset(samples[, masks, ...])</code>	A synthetic dataset containing data generated by a model of medigan
<code>Utils()</code>	Utils class containing reusable static methods.

Generators

```
class medigan.generators.Generators(config_manager: Optional[medigan.config_manager.ConfigManager]
    = None, model_selector:
    Optional[medigan.select_model.model_selector.ModelSelector] =
    None, model_executors: Optional[list] = None, model_contributors:
    Optional[list] = None, initialize_all_models: bool = False)
```

Bases: object

Generators class: Contains medigan's public methods to facilitate users' automated sample generation workflows.

Parameters

- **config_manager** (*ConfigManager*) – Provides the config dictionary, based on which *model_ids* are retrieved and models are selected and executed
- **model_selector** (*ModelSelector*) – Provides model comparison, search, and selection based on keys/values in the selection part of the config dict
- **model_executors** (*list*) – List of initialized *ModelExecutor* instances that handle model package download, init, and sample generation
- **initialize_all_models** (*bool*) – Flag indicating, if True, that one *ModelExecutor* for each *model_id* in the config dict should be initialized triggered by creation of *Generators*

class instance. Note that, if False, the *Generators* class will only initialize a *ModelExecutor* on the fly when need be i.e. when the generate method for the respective model is called.

config_manager

Provides the config dictionary, based on which model_ids are retrieved and models are selected and executed

Type *ConfigManager*

model_selector

Provides model comparison, search, and selection based on keys/values in the selection part of the config dict

Type *ModelSelector*

model_executors

List of initialized *ModelExecutor* instances that handle model package download, init, and sample generation

Type list

Methods Summary

<code>add_all_model_executors()</code>	Add <i>ModelExecutor</i> class instances for all models available in the config.
<code>add_metadata_from_file(model_id, ...)</code>	Read and parse the metadata of a local model, identified by <i>model_id</i> , from a metadata file in json format.
<code>add_metadata_from_input(model_id, ..., ...)</code>	Create a metadata dict for a local model, identified by <i>model_id</i> , given the necessary minimum metadata contents.
<code>add_model_contributor(model_id[, init_py_path])</code>	Add a <i>ModelContributor</i> instance of this <i>model_id</i> to the <i>self.model_contributors</i> list.
<code>add_model_executor(model_id[, ...])</code>	Add one <i>ModelExecutor</i> class instance corresponding to the specified <i>model_id</i> .
<code>add_model_to_config(model_id, metadata[, ...])</code>	Adding or updating a model entry in the global metadata.
<code>contribute(model_id, init_py_path, ..., ...)</code>	Implements the full model contribution workflow including model metadata generation, model test, model Zenodo upload, and medigan github issue creation.
<code>find_matching_models_by_values(values[, ...])</code>	Search for values (and keys) in model configs and return a list of each matching <i>ModelMatchCandidate</i> .
<code>find_model_and_generate(values[, ...])</code>	Search for values (and keys) in model configs to generate samples with the found model.
<code>find_model_executor_by_id(model_id)</code>	Find and return the <i>ModelExecutor</i> instance of this <i>model_id</i> in the <i>self.model_executors</i> list.
<code>find_models_and_rank(values[, ...])</code>	Search for values (and keys) in model configs, rank results and return sorted list of model dicts.
<code>find_models_rank_and_generate(values[, ...])</code>	Search for values (and keys) in model configs, rank results to generate samples with highest ranked model.
<code>generate(model_id[, num_samples, ...])</code>	Generate samples with the model corresponding to the <i>model_id</i> or return the model's generate function.

continues on next page

Table 1 – continued from previous page

<code>get_as_torch_dataloader([dataset, model_id, ...])</code>	Get torch Dataloader sampling synthetic data from medigan model.
<code>get_as_torch_dataset(model_id[, ...])</code>	Get synthetic data in a torch Dataset for specified medigan model.
<code>get_config_by_id(model_id[, config_key])</code>	Get and return the part of the config below a <i>config_key</i> for a specific <i>model_id</i> .
<code>get_generate_function(model_id[, ...])</code>	Return the model's generate function.
<code>get_model_contributor_by_id(model_id)</code>	Find and return the <i>ModelContributor</i> instance of this <i>model_id</i> in the <i>self.model_contributors</i> list.
<code>get_model_executor(model_id[, ...])</code>	Add and return the <i>ModelExecutor</i> instance of this <i>model_id</i> from the <i>self.model_executors</i> list.
<code>get_models_by_key_value_pair(key1, value1[, ...])</code>	Get and return a list of <i>model_id</i> dicts that contain the specified key value pair in their selection config.
<code>get_selection_criteria_by_id(model_id[, ...])</code>	Get and return the selection config dict for a specific <i>model_id</i> .
<code>get_selection_criteria_by_ids([model_ids, ...])</code>	Get and return a list of selection config dicts for each of the specified <i>model_ids</i> .
<code>get_selection_keys([model_id])</code>	Get and return all first level keys from the selection config dict for a specific <i>model_id</i> .
<code>get_selection_values_for_key(key[, model_id])</code>	Get and return the value of a specified key of the selection dict in the config for a specific <i>model_id</i> .
<code>is_model_executor_already_added(model_id)</code>	Check whether the <i>ModelExecutor</i> instance of this <i>model_id</i> is already in <i>self.model_executors</i> list.
<code>is_model_metadata_valid(model_id, metadata)</code>	Checking if a model's corresponding metadata is valid.
<code>list_models()</code>	Return the list of <i>model_ids</i> as strings based on config.
<code>push_to_github(model_id, github_access_token)</code>	Upload the model's metadata inside a github issue to the medigan github repository.
<code>push_to_zenodo(model_id, zenodo_access_token)</code>	Upload the model files as zip archive to a public Zenodo repository where the model will be persistently stored.
<code>rank_models_by_performance([model_ids, ...])</code>	Rank model based on a provided metric and return sorted list of model dicts.
<code>test_model(model_id[, is_local_model, ...])</code>	Test if a model generates and returns a specific number of samples in the correct format
<code>visualize(model_id[, slider_grouper, ...])</code>	Initialize and run <i>ModelVisualizer</i> of this <i>model_id</i> if it is available.

Methods Documentation

`add_all_model_executors()`

Add *ModelExecutor* class instances for all models available in the config.

Return type None

`add_metadata_from_file(model_id: str, metadata_file_path: str) → dict`

Read and parse the metadata of a local model, identified by *model_id*, from a metadata file in json format.

Parameters

- **model_id** (*str*) – The generative model's unique id
- **metadata_file_path** (*str*) – the path pointing to the metadata file

Returns Returns a dict containing the contents of parsed metadata json file.

Return type dict

add_metadata_from_input(*model_id*: str, *model_weights_name*: str, *model_weights_extension*: str, *generate_method_name*: str, *dependencies*: list, *fill_more_fields_interactively*: bool = True, *output_path*: str = 'config') → dict

Create a metadata dict for a local model, identified by *model_id*, given the necessary minimum metadata contents.

Parameters

- **model_id** (str) – The generative model’s unique id
- **model_weights_name** (str) – the name of the checkpoint file containing the model’s weights
- **model_weights_extension** (str) – the extension (e.g. .pt) of the checkpoint file containing the model’s weights
- **generate_method_name** (str) – the name of the sample generation method inside the models `__init__.py` file
- **dependencies** (list) – the list of dependencies that need to be installed via pip to run the model
- **fill_more_fields_interactively** (bool) – flag indicating whether a user will be interactively asked via command line for further input to fill out missing metadata content
- **output_path** (str) – the path where the created metadata json file will be stored

Returns Returns a dict containing the contents of the metadata json file.

Return type dict

add_model_contributor(*model_id*: str, *init_py_path*: Optional[str] = None) → [*medigan.contribute_model.model_contributor.ModelContributor*](#)

Add a *ModelContributor* instance of this *model_id* to the *self.model_contributors* list.

Parameters

- **model_id** (str) – The generative model’s unique id
- **init_py_path** (str) – The path to the local model’s `__init__.py` file needed for importing and running this model.

Returns *ModelContributor* class instance corresponding to the *model_id*

Return type [*ModelContributor*](#)

add_model_executor(*model_id*: str, *install_dependencies*: bool = False)

Add one *ModelExecutor* class instance corresponding to the specified *model_id*.

Parameters

- **model_id** (str) – The generative model’s unique id
- **install_dependencies** (bool) – flag indicating whether a generative model’s dependencies are automatically installed. Else error is raised if missing dependencies are detected.

Return type None

add_model_to_config(*model_id*: str, *metadata*: dict, *is_local_model*: Optional[bool] = None, *overwrite_existing_metadata*: bool = False, *store_new_config*: bool = True) → bool

Adding or updating a model entry in the global metadata.

Parameters

- **model_id** (str) – The generative model’s unique id
- **metadata** (dict) – The model’s corresponding metadata
- **is_local_model** (bool) – flag indicating whether the tested model is a new local user model i.e not yet part of medigan’s official models
- **overwrite_existing_metadata** (bool) – in case of *is_local_model*, flag indicating whether existing metadata for this model in medigan’s *config/global.json* should be overwritten.
- **store_new_config** (bool) – flag indicating whether the current model metadata should be stored on disk i.e. in *config/*

Returns Flag indicating whether model metadata update was successfully concluded

Return type bool

contribute(*model_id*: str, *init_py_path*: str, *github_access_token*: str, *zenodo_access_token*: str, *metadata_file_path*: Optional[str] = None, *model_weights_name*: Optional[str] = None, *model_weights_extension*: Optional[str] = None, *generate_method_name*: Optional[str] = None, *dependencies*: Optional[list] = None, *fill_more_fields_interactively*: bool = True, *overwrite_existing_metadata*: bool = False, *output_path*: str = 'config', *creator_name*: str = 'unknown name', *creator_affiliation*: str = 'unknown affiliation', *model_description*: str = "", *install_dependencies*: bool = False)

Implements the full model contribution workflow including model metadata generation, model test, model Zenodo upload, and medigan github issue creation.

Parameters

- **model_id** (str) – The generative model’s unique id
- **init_py_path** (str) – The path to the local model’s *__init__.py* file needed for importing and running this model.
- **github_access_token** (str) – a personal access token linked to your github user account, used as means of authentication
- **zenodo_access_token** (str) – a personal access token in Zenodo linked to a user account for authentication
- **metadata_file_path** (str) – the path pointing to the metadata file
- **model_weights_name** (str) – the name of the checkpoint file containing the model’s weights
- **model_weights_extension** (str) – the extension (e.g. .pt) of the checkpoint file containing the model’s weights
- **generate_method_name** (str) – the name of the sample generation method inside the models *__init__.py* file
- **dependencies** (list) – the list of dependencies that need to be installed via pip to run the model
- **fill_more_fields_interactively** (bool) – flag indicating whether a user will be interactively asked via command line for further input to fill out missing metadata content

- **overwrite_existing_metadata** (*bool*) – flag indicating whether existing metadata for this model in medigan’s *config/global.json* should be overwritten.
- **output_path** (*str*) – the path where the created metadata json file will be stored
- **creator_name** (*str*) – the creator name that will appear on the corresponding github issue
- **creator_affiliation** (*str*) – the creator affiliation that will appear on the corresponding github issue
- **model_description** (*list*) – the model_description that will appear on the corresponding github issue
- **install_dependencies** (*bool*) – flag indicating whether a generative model’s dependencies are automatically installed. Else error is raised if missing dependencies are detected.

Returns Returns the url pointing to the corresponding issue on github

Return type *str*

find_matching_models_by_values (*values: list, target_values_operator: str = 'AND', are_keys_also_matched: bool = False, is_case_sensitive: bool = False*) → *list*

Search for values (and keys) in model configs and return a list of each matching *ModelMatchCandidate*.

This function calls an identically named function in a *ModelSelector* instance.

Parameters

- **values** (*list*) – list of values used to search and find models corresponding to these *values*
- **target_values_operator** (*str*) – the operator indicating the relationship between *values* in the evaluation of model search results. Should be either “AND”, “OR”, or “XOR”.
- **are_keys_also_matched** (*bool*) – flag indicating whether, apart from values, the keys in the model config should also be searchable
- **is_case_sensitive** (*bool*) – flag indicating whether the search for values (and) keys in the model config should be case-sensitive.

Returns a list of *ModelMatchCandidate* class instances each of which was successfully matched against the search values.

Return type *list*

find_model_and_generate (*values: list, target_values_operator: str = 'AND', are_keys_also_matched: bool = False, is_case_sensitive: bool = False, num_samples: int = 30, output_path: Optional[str] = None, is_gen_function_returned: bool = False, install_dependencies: bool = False, **kwargs*)

Search for values (and keys) in model configs to generate samples with the found model.

Note that the number of found models should be ==1. Else no samples will be generated and a error is logged to console.

Parameters

- **values** (*list*) – list of values used to search and find models corresponding to these *values*
- **target_values_operator** (*str*) – the operator indicating the relationship between *values* in the evaluation of model search results. Should be either “AND”, “OR”, or “XOR”.
- **are_keys_also_matched** (*bool*) – flag indicating whether, apart from values, the keys in the model config should also be searchable

- **is_case_sensitive** (*bool*) – flag indicating whether the search for values (and) keys in the model config should be case-sensitive.
- **num_samples** (*int*) – the number of samples that will be generated
- **output_path** (*str*) – the path as str to the output folder where the generated samples will be stored
- **is_gen_function_returned** (*bool*) – flag indicating whether, instead of generating samples, the sample generation function will be returned
- **install_dependencies** (*bool*) – flag indicating whether a generative model’s dependencies are automatically installed. Else error is raised if missing dependencies are detected.
- ****kwargs** – arbitrary number of keyword arguments passed to the model’s sample generation function

Returns However, if *is_gen_function_returned* is True, it returns the internal generate function of the model.

Return type None

find_model_executor_by_id(*model_id: str*) → *medigan.execute_model.model_executor.ModelExecutor*

Find and return the *ModelExecutor* instance of this *model_id* in the *self.model_executors* list.

Parameters **model_id** (*str*) – The generative model’s unique id

Returns *ModelExecutor* class instance corresponding to the *model_id*

Return type *ModelExecutor*

find_models_and_rank(*values: list, target_values_operator: str = 'AND', are_keys_also_matched: bool = False, is_case_sensitive: bool = False, metric: str = 'SSIM', order: str = 'asc'*) → list

Search for values (and keys) in model configs, rank results and return sorted list of model dicts.

This function calls an identically named function in a *ModelSelector* instance.

Parameters

- **values** (*list*) – list of values used to search and find models corresponding to these *values*
- **target_values_operator** (*str*) – the operator indicating the relationship between *values* in the evaluation of model search results. Should be either “AND”, “OR”, or “XOR”.
- **are_keys_also_matched** (*bool*) – flag indicating whether, apart from values, the keys in the model config should also be searchable
- **is_case_sensitive** (*bool*) – flag indicating whether the search for values (and) keys in the model config should be case-sensitive.
- **metric** (*str*) – The key in the selection dict that corresponds to the metric of interest
- **order** (*str*) – the sorting order of the ranked results. Should be either “asc” (ascending) or “desc” (descending)

Returns a list of the searched and matched model dictionaries containing metric and *model_id*, sorted by metric.

Return type list

```
find_models_rank_and_generate(values: list, target_values_operator: str = 'AND',
                               are_keys_also_matched: bool = False, is_case_sensitive: bool = False,
                               metric: str = 'SSIM', order: str = 'asc', num_samples: int = 30,
                               output_path: Optional[str] = None, is_gen_function_returned: bool =
                               False, install_dependencies: bool = False, **kwargs)
```

Search for values (and keys) in model configs, rank results to generate samples with highest ranked model.

Parameters

- **values** (*list*) – list of values used to search and find models corresponding to these *values*
- **target_values_operator** (*str*) – the operator indicating the relationship between *values* in the evaluation of model search results. Should be either “AND”, “OR”, or “XOR”.
- **are_keys_also_matched** (*bool*) – flag indicating whether, apart from values, the keys in the model config should also be searchable
- **is_case_sensitive** (*bool*) – flag indicating whether the search for values (and) keys in the model config should be case-sensitive.
- **metric** (*str*) – The key in the selection dict that corresponds to the metric of interest
- **order** (*str*) – the sorting order of the ranked results. Should be either “asc” (ascending) or “desc” (descending)
- **num_samples** (*int*) – the number of samples that will be generated
- **output_path** (*str*) – the path as str to the output folder where the generated samples will be stored
- **is_gen_function_returned** (*bool*) – flag indicating whether, instead of generating samples, the sample generation function will be returned
- **install_dependencies** (*bool*) – flag indicating whether a generative model’s dependencies are automatically installed. Else error is raised if missing dependencies are detected.
- ****kwargs** – arbitrary number of keyword arguments passed to the model’s sample generation function

Returns However, if *is_gen_function_returned* is True, it returns the internal generate function of the model.

Return type None

```
generate(model_id: str, num_samples: int = 30, output_path: Optional[str] = None, save_images: bool =
         True, is_gen_function_returned: bool = False, install_dependencies: bool = False, **kwargs)
```

Generate samples with the model corresponding to the *model_id* or return the model’s generate function.

Parameters

- **model_id** (*str*) – The generative model’s unique id
- **num_samples** (*int*) – the number of samples that will be generated
- **output_path** (*str*) – the path as str to the output folder where the generated samples will be stored
- **save_images** (*bool*) – flag indicating whether generated samples are returned (i.e. as list of numpy arrays) or rather stored in file system (i.e in *output_path*)
- **is_gen_function_returned** (*bool*) – flag indicating whether, instead of generating samples, the sample generation function will be returned

- **install_dependencies** (*bool*) – flag indicating whether a generative model’s dependencies are automatically installed. Else error is raised if missing dependencies are detected.
- ****kwargs** – arbitrary number of keyword arguments passed to the model’s sample generation function

Returns Returns images as list of numpy arrays if *save_images* is False. However, if *is_gen_function_returned* is True, it returns the internal generate function of the model.

Return type list

get_as_torch_dataloader(*dataset=None, model_id: Optional[str] = None, num_samples: int = 1000, install_dependencies: bool = False, transform=None, batch_size=None, shuffle=None, sampler=None, batch_sampler=None, num_workers=None, collate_fn=None, pin_memory=None, drop_last=None, timeout=None, worker_init_fn=None, prefetch_factor: Optional[int] = None, persistent_workers: Optional[bool] = None, pin_memory_device: Optional[str] = None, **kwargs*) → torch.utils.data.dataloader.DataLoader

Get torch Dataloader sampling synthetic data from medigan model.

Dataloader combines a dataset and a sampler, and provides an iterable over the given torch dataset. Dataloader is created for synthetic data for the specified medigan model. Pytorch native parameters are set to None per default. Only those params are passed to the Dataloader() initialization function that are not None.

Parameters

- **dataset** (*Dataset*) – dataset from which to load the data.
- **model_id** – str The generative model’s unique id
- **num_samples** – int the number of samples that will be generated
- **install_dependencies** – bool flag indicating whether a generative model’s dependencies are automatically installed. Else error is raised if missing dependencies are detected.
- ****kwargs** – arbitrary number of keyword arguments passed to the model’s sample generation function (e.g. the input path for image-to-image translation models in medigan).
- **transform** – the torch data transformation functions to be applied to the data in the dataset.
- **batch_size** (*int, optional*) – how many samples per batch to load (default: None).
- **shuffle** (*bool, optional*) – set to True to have the data reshuffled at every epoch (default: None).
- **sampler** (*Sampler or Iterable, optional*) – defines the strategy to draw samples from the dataset. Can be any *Iterable* with `__len__` implemented. If specified, *shuffle* must not be specified. (default: None)
- **batch_sampler** (*Sampler or Iterable, optional*) – like *sampler*, but returns a batch of indices at a time. Mutually exclusive with *batch_size*, *shuffle*, *sampler*, and *drop_last*. (default: None)
- **num_workers** (*int, optional*) – how many subprocesses to use for data loading. 0 means that the data will be loaded in the main process. (default: None)
- **collate_fn** (*callable, optional*) – merges a list of samples to form a mini-batch of Tensor(s). Used when using batched loading from a map-style dataset. (default: None)

- **pin_memory** (*bool, optional*) – If True, the data loader will copy Tensors into CUDA pinned memory before returning them. If your data elements are a custom type, or your `collate_fn` returns a batch that is a custom type, see the example below. (default: None)
- **drop_last** (*bool, optional*) – set to True to drop the last incomplete batch, if the dataset size is not divisible by the batch size. If False and the size of dataset is not divisible by the batch size, then the last batch will be smaller. (default: None)
- **timeout** (*numeric, optional*) – if positive, the timeout value for collecting a batch from workers. Should always be non-negative. (default: None)
- **worker_init_fn** (*callable, optional*) – If not None, this will be called on each worker subprocess with the worker id (an int in `[0, num_workers - 1]`) as input, after seeding and before data loading. (default: None)
- **prefetch_factor** (*int, optional, keyword-only arg*) – Number of batches loaded in advance by each worker. 2 means there will be a total of `2 * num_workers` batches prefetched across all workers. (default: None).
- **persistent_workers** (*bool, optional*) – If True, the data loader will not shutdown the worker processes after a dataset has been consumed once. This allows to maintain the workers *Dataset* instances alive. (default: None)
- **pin_memory_device** (*str, optional*) – the device to pin memory to if `pin_memory` is True (default: None).

Returns a `torch.utils.data.DataLoader` object with data generated by model corresponding to inputted *Dataset* or *model_id*.

Return type `DataLoader`

get_as_torch_dataset(*model_id: str, num_samples: int = 100, install_dependencies: bool = False, transform=None, **kwargs*) → `torch.utils.data.dataset.Dataset`

Get synthetic data in a torch Dataset for specified medigan model.

The dataset returns a dict with keys `sample` (== image), `labels` (== condition), and `mask` (== segmentation mask). While key `'sample'` is mandatory, the other key value pairs are only returned if applicable to generative model.

Parameters

- **model_id** – str The generative model's unique id
- **num_samples** – int the number of samples that will be generated
- **install_dependencies** –
bool flag indicating whether a generative model's dependencies are automatically installed. Else error is raised if missing dependencies are detected.
- **transform** the torch data transformation functions to be applied to the data in the dataset.
- ****kwargs** – arbitrary number of keyword arguments passed to the model's sample generation function (e.g. the input path for image-to-image translation models in medigan).

Returns a `torch.utils.data.Dataset` object with data generated by model corresponding to *model_id*.

Return type `Dataset`

get_config_by_id(*model_id: str, config_key: Optional[str] = None*) → dict

Get and return the part of the config below a *config_key* for a specific *model_id*.

The `config_key` parameters can be separated by a `.` (dot) to allow for retrieval of nested config keys, e.g, `'execution.generator.name'`

This function calls an identically named function in a *ConfigManager* instance.

Parameters

- **model_id** (*str*) – The generative model's unique id
- **config_key** (*str*) – A key of interest present in the config dict

Returns a dictionary from the part of the config file corresponding to *model_id* and *config_key*.

Return type dict

get_generate_function(*model_id: str, num_samples: int = 30, output_path: Optional[str] = None, install_dependencies: bool = False, **kwargs*)

Return the model's generate function.

Relies on the *self.generate* function.

Parameters

- **model_id** (*str*) – The generative model's unique id
- **num_samples** (*int*) – the number of samples that will be generated
- **output_path** (*str*) – the path as str to the output folder where the generated samples will be stored
- **install_dependencies** (*bool*) – flag indicating whether a generative model's dependencies are automatically installed. Else error is raised if missing dependencies are detected.
- ****kwargs** – arbitrary number of keyword arguments passed to the model's sample generation function

Returns The internal reusable generate function of the generative model.

Return type function

get_model_contributor_by_id(*model_id: str*) → *medigan.contribute_model.model_contributor.ModelContributor*

Find and return the *ModelContributor* instance of this *model_id* in the *self.model_contributors* list.

Parameters **model_id** (*str*) – The generative model's unique id

Returns *ModelContributor* class instance corresponding to the *model_id*

Return type *ModelContributor*

get_model_executor(*model_id: str, install_dependencies: bool = False*) → *medigan.execute_model.model_executor.ModelExecutor*

Add and return the *ModelExecutor* instance of this *model_id* from the *self.model_executors* list.

Relies on *self.add_model_executor* and *self.find_model_executor_by_id* functions.

Parameters

- **model_id** (*str*) – The generative model's unique id
- **install_dependencies** (*bool*) – flag indicating whether a generative model's dependencies are automatically installed. Else error is raised if missing dependencies are detected.

Returns *ModelExecutor* class instance corresponding to the *model_id*

Return type *ModelExecutor*

get_models_by_key_value_pair(*key1*: str, *value1*: str, *is_case_sensitive*: bool = False) → list

Get and return a list of model_id dicts that contain the specified key value pair in their selection config.

The key param can contain ‘.’ (dot) separations to allow for retrieval of nested config keys such as ‘execution.generator.name’

This function calls an identically named function in a *ModelSelector* instance.

Parameters

- **key1** (str) – The key in the selection dict
- **value1** (str) – The value in the selection dict that corresponds to key1
- **is_case_sensitive** (bool) – flag to evaluate keys and values with case sensitivity if set to True

Returns a list of the dictionaries each containing a models id and the found key-value pair in the models config

Return type list

get_selection_criteria_by_id(*model_id*: str, *is_model_id_removed*: bool = True) → dict

Get and return the selection config dict for a specific model_id.

This function calls an identically named function in a *ModelSelector* instance.

Parameters

- **model_id** (str) – The generative model’s unique id
- **is_model_id_removed** (bool) – flag to to remove the model_ids from first level of dictionary.

Returns a dictionary corresponding to the selection config of a model

Return type dict

get_selection_criteria_by_ids(*model_ids*: Optional[list] = None, *are_model_ids_removed*: bool = True) → list

Get and return a list of selection config dicts for each of the specified model_ids.

This function calls an identically named function in a *ModelSelector* instance.

Parameters

- **model_ids** (list) – A list of generative models’ unique ids or ids abbreviated as integers (e.g. 1, 2, .. 21)
- **are_model_ids_removed** (bool) – flag to remove the model_ids from first level of dictionary.

Returns a list of dictionaries each corresponding to the selection config of a model

Return type list

get_selection_keys(*model_id*: Optional[str] = None) → list

Get and return all first level keys from the selection config dict for a specific model_id.

This function calls an identically named function in a *ModelSelector* instance.

Parameters **model_id** (str) – The generative model’s unique id

Returns a list containing the keys as strings of the selection config of the *model_id*.

Return type list

get_selection_values_for_key(*key: str, model_id: Optional[str] = None*) → list

Get and return the value of a specified key of the selection dict in the config for a specific *model_id*.

The key param can contain ‘.’ (dot) separations to allow for retrieval of nested config keys such as ‘execution.generator.name’

This function calls an identically named function in a *ModelSelector* instance.

Parameters

- **key** (*str*) – The key in the selection dict
- **model_id** (*str*) – The generative model’s unique id

Returns a list of the values that correspond to the key in the selection config of the *model_id*.

Return type list

is_model_executor_already_added(*model_id*) → bool

Check whether the *ModelExecutor* instance of this *model_id* is already in *self.model_executors* list.

Parameters **model_id** (*str*) – The generative model’s unique id

Returns indicating whether this *ModelExecutor* had been already previously added to *self.model_executors*

Return type bool

is_model_metadata_valid(*model_id: str, metadata: dict, is_local_model: bool = True*) → bool

Checking if a model’s corresponding metadata is valid.

Specific fields in the model’s metadata are mandatory. It is asserted if these key value pairs are present.

Parameters

- **model_id** (*str*) – The generative model’s unique id
- **metadata** (*dict*) – The model’s corresponding metadata
- **is_local_model** (*bool*) – flag indicating whether the tested model is a new local user model i.e not yet part of medigan’s official models

Returns Flag indicating whether the specific model’s metadata format and fields are valid

Return type bool

list_models() → list

Return the list of *model_ids* as strings based on config.

Return type list

push_to_github(*model_id: str, github_access_token: str, package_link: Optional[str] = None, creator_name: str = "", creator_affiliation: str = "", model_description: str = ""*)

Upload the model’s metadata inside a github issue to the medigan github repository.

To add your model to medigan, your metadata will be reviewed on Github and added to medigan’s official model metadata

The medigan repository issues page: <https://github.com/RichardObi/medigan/issues>

Get your Github access token here: <https://github.com/settings/tokens>

Parameters

- **model_id** (*str*) – The generative model’s unique id

- **github_access_token** (*str*) – a personal access token linked to your github user account, used as means of authentication
- **package_link** – a package link
- **creator_name** (*str*) – the creator name that will appear on the corresponding github issue
- **creator_affiliation** (*str*) – the creator affiliation that will appear on the corresponding github issue
- **model_description** (*list*) – the model_description that will appear on the corresponding github issue

Returns Returns the url pointing to the corresponding issue on github

Return type *str*

push_to_zenodo(*model_id: str, zenodo_access_token: str, creator_name: str = 'unknown name', creator_affiliation: str = 'unknown affiliation', model_description: str = ''*) → *str*

Upload the model files as zip archive to a public Zenodo repository where the model will be persistently stored.

Get your Zenodo access token here: <https://zenodo.org/account/settings/applications/tokens/new/> (Enable scopes *deposit:actions* and *deposit:write*)

Parameters

- **model_id** (*str*) – The generative model’s unique id
- **zenodo_access_token** (*str*) – a personal access token in Zenodo linked to a user account for authentication
- **creator_name** (*str*) – the creator name that will appear on the corresponding Zenodo model upload homepage
- **creator_affiliation** (*str*) – the creator affiliation that will appear on the corresponding Zenodo model upload homepage
- **model_description** (*list*) – the model_description that will appear on the corresponding Zenodo model upload homepage

Returns Returns the url pointing to the corresponding Zenodo model upload homepage

Return type *str*

rank_models_by_performance(*model_ids: Optional[list] = None, metric: str = 'SSIM', order: str = 'asc'*) → *list*

Rank model based on a provided metric and return sorted list of model dicts.

The metric param can contain ‘.’ (dot) separations to allow for retrieval of nested metric config keys such as ‘downstream_task.CLF.accuracy’

This function calls an identically named function in a *ModelSelector* instance.

Parameters

- **model_ids** (*list*) – only evaluate the *model_ids* in this list. If none, evaluate all available *model_ids*
- **metric** (*str*) – The key in the selection dict that corresponds to the metric of interest
- **order** (*str*) – the sorting order of the ranked results. Should be either “asc” (ascending) or “desc” (descending)

Returns a list of model dictionaries containing metric and *model_id*, sorted by metric.

Return type list

test_model(*model_id*: str, *is_local_model*: bool = True, *overwrite_existing_metadata*: bool = False, *store_new_config*: bool = True, *num_samples*: int = 3, *install_dependencies*: bool = False)

Test if a model generates and returns a specific number of samples in the correct format

Parameters

- **model_id** (str) – The generative model’s unique id
- **is_local_model** (bool) – flag indicating whether the tested model is a new local user model i.e not yet part of medigan’s official models
- **overwrite_existing_metadata** (bool) – in case of *is_local_model*, flag indicating whether existing metadata for this model in medigan’s *config/global.json* should be overwritten.
- **store_new_config** (bool) – flag indicating whether the current model metadata should be stored on disk i.e. in config/
- **num_samples** (int) – the number of samples that will be generated
- **install_dependencies** (bool) – flag indicating whether a generative model’s dependencies are automatically installed. Else error is raised if missing dependencies are detected.

visualize(*model_id*: str, *slider_grouper*: int = 10, *auto_close*: bool = False, *install_dependencies*: bool = False) → None

Initialize and run *ModelVisualizer* of this *model_id* if it is available. It allows to visualize a sample from the model’s output. UI window will pop up allowing the user to control the generation parameters (conditional and unconditional ones).

Parameters

- **model_id** (str) – The generative model’s unique id to visualize.
- **slider_grouper** (int) – Number of input parameters to group together within one slider.
- **auto_close** (bool) – Flag for closing the user interface automatically after time. Used while testing.
- **install_dependencies** (bool) – flag indicating whether a generative model’s dependencies are automatically installed. Else error is raised if missing dependencies are detected.

1.4.2 ModelExecutor

medigan.execute_model.model_executor Module

Model executor class that downloads models, loads them as python packages, and runs their generate functions.

Classes

<code>ModelExecutor(model_id, execution_config[, ...])</code>	<i>ModelExecutor</i> class: Find config links to download models, init models as python packages, run generate methods.
<code>Path(*args, **kwargs)</code>	PurePath subclass that can make system calls.
<code>Utils()</code>	Utils class containing reusable static methods.
<code>tqdm(*_, **_)</code>	Decorate an iterable object, returning an iterator which acts exactly like the original iterable, but prints a dynamically updating progressbar every time a value is requested.

ModelExecutor

```
class medigan.execute_model.model_executor.ModelExecutor(model_id: str, execution_config: dict,
                                                         download_package: bool = True,
                                                         install_dependencies: bool = False)
```

Bases: object

ModelExecutor class: Find config links to download models, init models as python packages, run generate methods.

Parameters

- **model_id** (*str*) – The generative model’s unique id
- **execution_config** (*dict*) – The part of the config below the ‘execution’ key
- **download_package** (*bool*) – Flag indicating, if True, that the model’s package should be downloaded instead of using an existing one that was downloaded previously
- **install_dependencies** (*bool*) – flag indicating whether a generative model’s dependencies are automatically installed. Else error is raised if missing dependencies are detected.

model_id

The generative model’s unique id

Type str

execution_config

The part of the config below the ‘execution’ key

Type dict

download_package

Flag indicating, if True, that the model’s package should be downloaded instead of using an existing one that was downloaded previously

Type bool

image_size

Pixel dimension of the generated samples, where images are assumed to have the same width and height

Type int

dependencies

List of the dependencies of a models python package.

Type list

model_name

Name of the generative model

Type str

model_extension

File extension of the generative model's weights file.

Type str

package_name

Name of the model's python package i.e. the name of the model's zip file and unzipped package folder

Type str

package_link

The link to the zipped model package. Note: Convention is to host models on Zenodo (reason: static doi content)

Type str

generate_method_name

The name of the model's generate method inside the model package. This method is called to generate samples.

Type str

generate_method_args

The args of the model's generate method inside the model package

Type dict

serialised_model_file_path

Path as string to the generative model's weights file

Type str

package_path

Path as string to the generative model's python package containing an `__init__.py` file

Type str

deserialized_model_as_lib

The generative model's package imported as python library. Generate method inside this library can be called.

Methods Summary

<code>generate([num_samples, output_path, ...])</code>	Generate samples using the generative model or return the model's generate function.
<code>is_model_already_unpacked()</code>	Check if a valid path to the model files exists and, if so, set the <i>package_path</i>

Methods Documentation

generate(*num_samples*: int = 20, *output_path*: Optional[str] = None, *save_images*: bool = True, *is_gen_function_returned*: bool = False, *batch_size*: int = 32, ***kwargs*)

Generate samples using the generative model or return the model's generate function.

The name and additional parameters of the generate function of the respective generative model are retrieved from the model's *execution_config*.

Parameters

- **num_samples** (int) – the number of samples that will be generated
- **output_path** (str) – the path as str to the output folder where the generated samples will be stored
- **save_images** (bool) – flag indicating whether generated samples are returned (i.e. as list of numpy arrays) or rather stored in file system (i.e in *output_path*)
- **is_gen_function_returned** (bool) – flag indicating whether, instead of generating samples, the sample generation function will be returned
- **batch_size** (int) – the batch size for the sample generation function
- ****kwargs** – arbitrary number of keyword arguments passed to the model's sample generation function

Returns Returns images as list of numpy arrays if *save_images* is False. However, if *is_gen_function_returned* is True, it returns the internal generate function of the model.

Return type list

Raises Exception – If the generate method of the model does not exist, cannot be called, or is called with missing params, or if the sample generation inside the model package returns an exception.

is_model_already_unpacked() → bool

Check if a valid path to the model files exists and, if so, set the *package_path*

1.4.3 ModelVisualizer

medigan.model_visualizer Module

ModelVisualizer class providing visualizing corresponding model input and model output changes.

Classes

<code>Button(ax, label[, image, color, hovercolor])</code>	A GUI neutral button.
<code>ModelVisualizer(model_executor, config)</code>	<i>ModelVisualizer</i> class: Visualises synthetic data through a user interface.
<code>Slider(ax, label, valmin, valmax[, valinit, ...])</code>	A slider representing a floating point range.

ModelVisualizer

class `medigan.model_visualizer.ModelVisualizer`(*model_executor*, *config*: *None*)

Bases: `object`

ModelVisualizer class: Visualises synthetic data through a user interface. Depending on a model, it is possible to control the input latent vector values and conditional input.

Parameters

- **model_executor** (`ModelExecutor`) – The generative model’s executor object
- **config** (*dict*) – The config dict containing the model metadata

model_executor

The generative model’s executor object

Type `ModelExecutor`

input_latent_vector_size

Size of the latent vector used as an input for generation

Type `int`

conditional

Flag for models with conditional input

Type `bool`

condition

Value of the conditinal input to the model

Type `Union[int, float]`

max_input_value

Absolute value used for setting latent values input range

Type `float`

Methods Summary

<code>visualize([slider_grouper, auto_close])</code>	Visualize the model's output.
--	-------------------------------

Methods Documentation

visualize(*slider_grouper: int = 10, auto_close=False*)

Visualize the model's output. This method is called by the user. It opens up a user interface with available controls.

Parameters

- **slider_grouper** (*int*) – Number of input parameters to group together within one slider.
- **auto_close** (*bool*) – Flag for closing the user interface automatically after time. Used while testing.

Return type None

1.4.4 ModelSelector

medigan.select_model.model_selector Module

Model selection class that describes, finds, compares, and ranks generative models.

Classes

<code>ConfigManager([config_dict, ...])</code>	<i>ConfigManager</i> class: Downloads, loads and parses medigan's config json as dictionary.
<code>MatchedEntry(key, value[, matching_element])</code>	<i>MatchedEntry</i> class: One target key-value pair that matches with a model's selection config.
<code>ModelMatchCandidate(model_id, target_values)</code>	<i>ModelMatchCandidate</i> class: A prospectively matching model given the target values as model search params.
<code>ModelSelector([config_manager])</code>	<i>ModelSelector</i> class: Given a config dict, gets, searches, and ranks matching models.
<code>Utils()</code>	Utils class containing reusable static methods.

ModelSelector

class `medigan.select_model.model_selector.ModelSelector`(*config_manager: Optional[medigan.config_manager.ConfigManager] = None*)

Bases: `object`

ModelSelector class: Given a config dict, gets, searches, and ranks matching models.

Parameters `config_manager` (`ConfigManager`) – Provides the config dictionary, based on which models are selected and compared.

`config_manager`

Provides the config dictionary, based on which models are selected and compared.

Type `ConfigManager`

`model_selection_dicts`

Contains a dictionary for each model id that consists of the *model_id* and the selection config of that model

Type list

Methods Summary

<code>find_matching_models_by_values(values[, ...])</code>	Search for values (and keys) in model configs and return a list of each matching <i>ModelMatchCandidate</i> .
<code>find_models_and_rank(values[, ...])</code>	Search for values (and keys) in model configs, rank results and return sorted list of model dicts.
<code>get_models_by_key_value_pair(key1, value1[, ...])</code>	Get and return a list of <i>model_id</i> dicts that contain the specified key value pair in their selection config.
<code>get_selection_criteria_by_id(model_id[, ...])</code>	Get and return the selection config dict for a specific <i>model_id</i> .
<code>get_selection_criteria_by_ids([model_ids, ...])</code>	Get and return a list of selection config dicts for each of the specified <i>model_ids</i> .
<code>get_selection_keys([model_id])</code>	Get and return all first level keys from the selection config dict for a specific <i>model_id</i> .
<code>get_selection_values_for_key(key[, model_id])</code>	Get and return the value of a specified key of the selection dict in the config for a specific <i>model_id</i> .
<code>rank_models_by_performance([model_ids, ...])</code>	Rank model based on a provided metric and return sorted list of model dicts.
<code>recursive_search_for_values(search_dict, ...)</code>	Do a recursive search to match values in the <i>search_dict</i> with values (and keys) in a model's config.

Methods Documentation

find_matching_models_by_values (*values*: list, *target_values_operator*: str = 'AND', *are_keys_also_matched*: bool = False, *is_case_sensitive*: bool = False) → list

Search for values (and keys) in model configs and return a list of each matching *ModelMatchCandidate*.

Uses `self.recursive_search_for_values` to recursively populate each *ModelMatchCandidate* with *MatchedEntry* instances. After populating, each *ModelMatchCandidate* is evaluated based on the provided *target_values_operator* and *values* list using *ModelMatchCandidate.check_if_is_match*.

Parameters

- **values** (list) – list of values used to search and find models corresponding to these values
- **target_values_operator** (str) – the operator indicating the relationship between *values* in the evaluation of model search results. Should be either “AND”, “OR”, or “XOR”.
- **are_keys_also_matched** (bool) – flag indicating whether, apart from values, the keys in the model config should also be searchable

- **is_case_sensitive** (*bool*) – flag indicating whether the search for values (and) keys in the model config should be case-sensitive.

Returns a list of *ModelMatchCandidate* class instances each of which was successfully matched against the search values.

Return type list

find_models_and_rank(*values: list, target_values_operator: str = 'AND', are_keys_also_matched: bool = False, is_case_sensitive: bool = False, metric: str = 'SSIM', order: str = 'asc'*) → list

Search for values (and keys) in model configs, rank results and return sorted list of model dicts.

Parameters

- **values** (*list*) – list of values used to search and find models corresponding to these *values*
- **target_values_operator** (*str*) – the operator indicating the relationship between *values* in the evaluation of model search results. Should be either “AND”, “OR”, or “XOR”.
- **are_keys_also_matched** (*bool*) – flag indicating whether, apart from *values*, the keys in the model config should also be searchable
- **is_case_sensitive** (*bool*) – flag indicating whether the search for values (and) keys in the model config should be case-sensitive.
- **metric** (*str*) – The key in the selection dict that corresponds to the *metric* of interest
- **order** (*str*) – the sorting order of the ranked results. Should be either “asc” (ascending) or “desc” (descending)

Returns a list of the searched and matched model dictionaries containing *metric* and *model_id*, sorted by *metric*.

Return type list

get_models_by_key_value_pair(*key1: str, value1: str, is_case_sensitive: bool = False*) → list

Get and return a list of *model_id* dicts that contain the specified key value pair in their selection config.

The key param can contain ‘.’ (dot) separations to allow for retrieval of nested config keys such as ‘execution.generator.name’. If *key1* points to a list, any value in the list that matches *value1* is accepted.

Parameters

- **key1** (*str*) – The key in the selection dict
- **value1** (*str*) – The value in the selection dict that corresponds to *key1*
- **is_case_sensitive** (*bool*) – flag to evaluate keys and values with case sensitivity if set to True

Returns a list of the dictionaries each containing a model’s *model_id* and the found key-value pair in the models config

Return type list

get_selection_criteria_by_id(*model_id: str, is_model_id_removed: bool = True*) → dict

Get and return the selection config dict for a specific *model_id*.

Parameters

- **model_id** (*str*) – The generative model’s unique id
- **is_model_id_removed** (*bool*) – flag to to remove the *model_id* from first level of each dictionary.

Returns a dictionary corresponding to the selection config of a model

Return type dict

get_selection_criteria_by_ids(*model_ids*: Optional[list] = None, *are_model_ids_removed*: bool = True) → list

Get and return a list of selection config dicts for each of the specified *model_ids*.

Parameters

- **model_ids** (*list*) – A list of generative models’ unique ids
- **are_model_ids_removed** (*bool*) – flag to remove the *model_ids* from first level of dictionary.

Returns a list of dictionaries each corresponding to the selection config of a model

Return type list

get_selection_keys(*model_id*: Optional[str] = None) → list

Get and return all first level keys from the selection config dict for a specific *model_id*.

Parameters **model_id** (*str*) – The generative model’s unique id

Returns a list containing the keys as strings of the selection config of the *model_id*.

Return type list

get_selection_values_for_key(*key*: str, *model_id*: Optional[str] = None) → list

Get and return the value of a specified key of the selection dict in the config for a specific *model_id*.

The key param can contain ‘.’ (dot) separations to allow for retrieval of nested config keys such as ‘execution.generator.name’

Parameters

- **key** (*str*) – The key in the selection dict
- **model_id** (*str*) – The generative model’s unique id

Returns a list of the values that correspond to the key in the selection config of the *model_id*.

Return type list

rank_models_by_performance(*model_ids*: Optional[list] = None, *metric*: str = 'SSIM', *order*: str = 'asc') → list

Rank model based on a provided metric and return sorted list of model dicts.

The metric param can contain ‘.’ (dot) separations to allow for retrieval via nested metric config keys such as ‘downstream_task.CLF.accuracy’. If the value found for the metric key is of type list, then the largest value in the list is used for ranking if *order* is descending, while the smallest value is used if *order* is ascending.

Parameters

- **model_ids** (*list*) – only evaluate the *model_ids* in this list. If none, evaluate all available *model_ids*
- **metric** (*str*) – The key in the selection dict that corresponds to the metric of interest
- **order** (*str*) – the sorting order of the ranked results. Should be either “asc” (ascending) or “desc” (descending)

Returns a list of model dictionaries containing metric and *model_id*, sorted by *metric*.

Return type list

recursive_search_for_values(*search_dict*: dict, *model_match_candidate*: medigan.select_model.model_match_candidate.ModelMatchCandidate)

Do a recursive search to match values in the *search_dict* with values (and keys) in a model's config.

The provided *ModelMatchCandidate* instance is recursively populated with *MatchedEntry* instances. A *MatchedEntry* instance contains a key-value pair found in the model's config that matches with one search term of interest.

The search terms of interest are stored in *ModelMatchCandidate.target_values*. The model's selection config is provided in the 'search_dict'.

To traverse the *search_dict*, the value for each key in the *search_dict* is retrieved.

- If that value is of type dictionary, the function calls itself with that nested dictionary as new *search_dict*.
- If that value is of type list, each value in the list is compared with each search term of interest in *ModelMatchCandidate.target_values*.
- If the value of the *search_dict* is of another type (i.e. str), it is compared with each search term of interest in *ModelMatchCandidate.target_values*.

Parameters

- **search_dict** (*dict*) – contains keys and values from a model's config that are matched against a set of search values.
- **model_match_candidate** (*ModelMatchCandidate*) – a class instance representing a model to be prepared for evaluation (populated with *MatchedEntry* objects), as to whether it is a match given its search values (*self.target_values*).

Returns a *ModelMatchCandidate* class instance that has been populated with *MatchedEntry* class instances.

Return type list

1.4.5 ModelContributor

medigan.contribute_model.model_contributor Module

Model contributor class that tests models, creates metadata entries, uploads and contributes them to medigan.

Classes

GithubModelUploader(model_id, access_token)	<i>GithubModelUploader</i> class: Pushes the metadata of a user's model to the medigan repo, where it creates a dedicated github issue.
<i>ModelContributor</i> (model_id, init_py_path)	<i>ModelContributor</i> class: Contributes a user's local model to the public medigan library
Path(*args, **kwargs)	PurePath subclass that can make system calls.
Utils()	Utils class containing reusable static methods.
ZenodoModelUploader(model_id, access_token)	<i>ZenodoModelUploader</i> class: Uploads a user's model via API to Zenodo, here it is permanently stored with DOI.

ModelContributor

class medigan.contribute_model.model_contributor.**ModelContributor**(*model_id: str, init_py_path: str*)

Bases: object

ModelContributor class: Contributes a user's local model to the public medigan library

Parameters

- **model_id** (*str*) – The generative model's unique id
- **init_py_path** (*str*) – The path to the local model's `__init__.py` file needed for importing and running this model.

model_id

The generative model's unique id

Type str

init_py_path

The path to the local model's `__init__.py` file needed for importing and running this model.

Type str

package_path

Path as string to the generative model's python package

Type str

package_name

Name of the model's python package i.e. the name of the model's zip file and unzipped package folder

Type str

metadata_file_path

Path as string to the generative model's metadata file e.g. default is relative path to package root.

Type str

zenodo_model_uploader

An instance of the *ZenodoModelUploader* class

Type str

github_model_uploader

An instance of the *GithubModelUploader* class.

Type str

Methods Summary

<code>add_metadata_from_file(metadata_file_path)</code>	Read and parse the metadata of a local model, identified by <i>model_id</i> , from a metadata file in json format.
<code>add_metadata_from_input(...)</code>	Create a metadata dict for a local model, identified by <i>model_id</i> , given the necessary minimum metadata contents.
<code>is_value_for_key_already_set(key, metadata, ...)</code>	Check if the value of a <i>key</i> in a <i>metadata</i> dictionary is already set and e.g.
<code>load_metadata_template()</code>	Loads and parses (json to dict) a default medigan metadata template.
<code>push_to_github(access_token[, package_link, ...])</code>	Upload the model's metadata inside a github issue to the medigan github repository.
<code>push_to_zenodo(access_token, creator_name, ...)</code>	Upload the model files as zip archive to a public Zenodo repository where the model will be persistently stored.
<code>validate_and_update_model_weights_path()</code>	Check if the model files can be found in the <i>package_path</i> or based on the <i>path_to_metadata</i> .
<code>validate_init_py_path(init_py_path)</code>	Asserts whether the <i>init_py_path</i> exists and points to a valid <code>__init__.py</code> correct file.
<code>validate_local_model_import()</code>	Check if the model package in the <i>package_path</i> can be imported as python library using <code>importlib</code> .
<code>validate_model_id(model_id[, max_chars, ...])</code>	Asserts if the <i>model_id</i> is in the correct format and has a valid length

Methods Documentation

add_metadata_from_file(*metadata_file_path*) → dict

Read and parse the metadata of a local model, identified by *model_id*, from a metadata file in json format.

Parameters

- **model_id** (*str*) – The generative model's unique id
- **metadata_file_path** (*str*) – the path pointing to the metadata file

Returns Returns a dict containing the contents of parsed metadata json file.

Return type dict

add_metadata_from_input(*model_weights_name*: Optional[*str*] = None, *model_weights_extension*: Optional[*str*] = None, *generate_method_name*: Optional[*str*] = None, *dependencies*: list = [], *fill_more_fields_interactively*: bool = True, *output_path*: *str* = 'config')

Create a metadata dict for a local model, identified by *model_id*, given the necessary minimum metadata contents.

Parameters

- **model_id** (*str*) – The generative model's unique id
- **model_weights_name** (*str*) – the name of the checkpoint file containing the model's weights
- **model_weights_extension** (*str*) – the extension (e.g. .pt) of the checkpoint file containing the model's weights

- **generate_method_name** (*str*) – the name of the sample generation method inside the models `__init__.py` file
- **dependencies** (*list*) – the list of dependencies that need to be installed via pip to run the model.
- **fill_more_fields_interactively** (*bool*) – flag indicating whether a user will be interactively asked via command line for further input to fill out missing metadata content
- **output_path** (*str*) – the path where the created metadata json file will be stored.

Returns Returns a dict containing the contents of the metadata json file.

Return type dict

is_value_for_key_already_set(*key: str, metadata: dict, nested_key*) → bool

Check if the value of a *key* in a *metadata* dictionary is already set and e.g. not an empty string, dict or list.

Parameters

- **key** (*str*) – The key in the currently traversed part of the model’s metadata dictionary
- **metadata** (*dict*) – The currently traversed part of the model’s metadata dictionary
- **nested_key** (*str*) – the *nested_key* indicates which subpart of the model’s metadata we are currently traversing

Returns Flag indicating whether a value exists for the *key* in the dict

Return type bool

load_metadata_template() → dict

Loads and parses (json to dict) a default medigan metadata template.

Returns Returns the metadata template as dict

Return type dict

push_to_github(*access_token: str, package_link: Optional[str] = None, creator_name: str = "", creator_affiliation: str = "", model_description: str = ""*)

Upload the model’s metadata inside a github issue to the medigan github repository.

To add your model to medigan, your metadata will be reviewed on Github and added to medigan’s official model metadata

The medigan repository issues page: <https://github.com/RichardObi/medigan/issues>

Get your Github access token here: <https://github.com/settings/tokens>

Parameters

- **access_token** (*str*) – a personal access token linked to your github user account, used as means of authentication
- **package_link** – a package link
- **creator_name** (*str*) – the creator name that will appear on the corresponding github issue
- **creator_affiliation** (*str*) – the creator affiliation that will appear on the corresponding github issue
- **model_description** (*list*) – the model_description that will appear on the corresponding github issue

Returns Returns the url pointing to the corresponding issue on github

Return type str

push_to_zenodo(*access_token: str, creator_name: str, creator_affiliation: str, model_description: str = ""*)

Upload the model files as zip archive to a public Zenodo repository where the model will be persistently stored.

Get your Zenodo access token here: <https://zenodo.org/account/settings/applications/tokens/new/> (Enable scopes *deposit:actions* and *deposit:write*)

Parameters

- **access_token** (*str*) – a personal access token in Zenodo linked to a user account for authentication
- **creator_name** (*str*) – the creator name that will appear on the corresponding Zenodo model upload homepage
- **creator_affiliation** (*str*) – the creator affiliation that will appear on the corresponding Zenodo model upload homepage
- **model_description** (*list*) – the model_description that will appear on the corresponding Zenodo model upload homepage

Returns Returns the url pointing to the corresponding Zenodo model upload homepage

Return type *str*

validate_and_update_model_weights_path() → *dict*

Check if the model files can be found in the *package_path* or based on the *path_to_metadata*.

Ideally, the user provided *package_path* and the *path_to_metadata* should both point to the same model package containing weights, config, license, etc. Here we check both of these paths to find the model weights.

Returns Returns the metadata after updating the path to the model's checkpoint's weights

Return type *dict*

validate_init_py_path(*init_py_path*) → *bool*

Asserts whether the *init_py_path* exists and points to a valid *__init__.py* correct file.

Parameters **init_py_path** (*str*) – The path to the local model's *__init__.py* file needed for importing and running this model.

validate_local_model_import()

Check if the model package in the *package_path* can be imported as python library using *importlib*.

validate_model_id(*model_id: str, max_chars: int = 30, min_chars: int = 13*) → *bool*

Asserts if the *model_id* is in the correct format and has a valid length

Parameters

- **model_id** (*str*) – The generative model's unique id
- **max_chars** (*int*) – the maximum of chars allowed in the *model_id*
- **min_chars** (*int*) – the minimum of chars allowed in the *model_id*

Returns Returns flag indicating whether the *model_id* is correctly formatted.

Return type *bool*

1.4.6 ConfigManager

medigan.config_manager Module

Config manager class that downloads, ingests, parses, and prepares the config information for all models.

Classes

<code>ConfigManager([config_dict, ...])</code>	<i>ConfigManager</i> class: Downloads, loads and parses medigan's config json as dictionary.
<code>Path(*args, **kwargs)</code>	PurePath subclass that can make system calls.
<code>Utils()</code>	Utils class containing reusable static methods.

ConfigManager

class medigan.config_manager.**ConfigManager**(*config_dict: Optional[dict] = None, is_new_download_forced: bool = False*)

Bases: object

ConfigManager class: Downloads, loads and parses medigan's config json as dictionary.

Parameters

- **config_dict** (*dict*) – Optionally provides the config dictionary if already loaded and parsed in a previous process.
- **is_new_download_forced** (*bool*) – Flags, if True, that a new config file should be downloaded from the config link instead of parsing an existing file.

config_dict

Optionally provides the config dictionary if already loaded and parsed in a previous process.

Type dict

is_new_download_forced

Flags, if True, that a new config file should be downloaded from the config link instead of parsing an existing file.

Type bool

model_ids

Lists the unique id's of the generative models specified in the *config_dict*

Type list

is_config_loaded

Flags if the loading and parsing of the config file was successful (True) or not (False).

Type bool

Methods Summary

<code>add_model_to_config(model_id, metadata[, ...])</code>	Adding or updating a model entry in the global metadata.
<code>get_config_by_id(model_id[, config_key])</code>	From <i>config_manager</i> , get and return the part of the config below a <i>config_key</i> for a specific <i>model_id</i> .
<code>is_model_in_config(model_id)</code>	Checking if a <i>model_id</i> is present in the global model metadata file
<code>is_model_metadata_valid(model_id, metadata)</code>	Checking if a model's corresponding metadata is valid.
<code>load_config_file([is_new_download_forced])</code>	Load a config file and return boolean flag indicating success of loading process.

Methods Documentation

add_model_to_config(*model_id*: str, *metadata*: dict, *is_local_model*: bool = True, *overwrite_existing_metadata*: bool = False, *store_new_config*: bool = True) → bool

Adding or updating a model entry in the global metadata.

Parameters

- **model_id** (str) – The generative model's unique id
- **metadata** (dict) – The model's corresponding metadata
- **is_local_model** (bool) – flag indicating whether the tested model is a new local user model i.e not yet part of medigan's official models
- **overwrite_existing_metadata** (bool) – in case of *is_local_model*, flag indicating whether existing metadata for this model in medigan's *config/global.json* should be overwritten.
- **store_new_config** (bool) – flag indicating whether the current model metadata should be stored on disk i.e. in *config/*

Returns Flag indicating whether model metadata update was successfully concluded

Return type bool

get_config_by_id(*model_id*: str, *config_key*: Optional[str] = None) → dict

From *config_manager*, get and return the part of the config below a *config_key* for a specific *model_id*.

The key param can contain '.' (dot) separations to allow for retrieval of nested config keys such as 'execution.generator.name'

Parameters

- **model_id** (str) – The generative model's unique id
- **config_key** (str) – A key of interest present in the config dict

Returns a dictionary from the part of the config file corresponding to *model_id* and *config_key*.

Return type dict

is_model_in_config(*model_id*: str) → bool

Checking if a *model_id* is present in the global model metadata file

Parameters **model_id** (str) – The generative model's unique id

Returns Flag indicating whether a *model_id* is present in global model metadata

Return type bool

is_model_metadata_valid(*model_id*: str, *metadata*: dict, *is_local_model*: bool = True) → bool

Checking if a model's corresponding metadata is valid.

Specific fields in the model's metadata are mandatory. It is asserted if these key value pairs are present.

Parameters

- **model_id** (str) – The generative model's unique id
- **metadata** (dict) – The model's corresponding metadata
- **is_local_model** (bool) – flag indicating whether the tested model is a new local user model i.e not yet part of medigan's official models

Returns Flag indicating whether the specific model's metadata format and fields are valid

Return type bool

load_config_file(*is_new_download_forced*: bool = False) → bool

Load a config file and return boolean flag indicating success of loading process.

If the config file is not present in *medigan.CONSTANTS.CONFIG_FILE_FOLDER*, it is per default downloaded from the web resource specified in *medigan.CONSTANTS.CONFIG_FILE_URL*.

Parameters **is_new_download_forced** (bool) – Forces new download of config file even if the file has been downloaded before.

Returns a boolean flag indicating true only if the config file was loaded successfully.

Return type bool

1.5 Full Code Documentation

1.5.1 medigan package

Subpackages

medigan.contribute_model package

Submodules

medigan.contribute_model.base_model_uploader module

Base Model uploader class that uploads models to medigan associated data storage services.

class *medigan.contribute_model.base_model_uploader.BaseModelUploader*(*model_id*: str, *metadata*: dict)

Bases: object

BaseModelUploader class: Uploads a user's model and metadata to third party storage to allow its inclusion into the medigan library.

Parameters

- **model_id** (*str*) – The generative model’s unique id
- **metadata** (*dict*) – The model’s corresponding metadata

model_id

The generative model’s unique id

Type *str*

metadata

The model’s corresponding metadata

Type *dict*

push()**medigan.contribute_model.github_model_uploader module**

Github Model uploader class that uploads the metadata of a new model to the medigan github repository.

```
class medigan.contribute_model.github_model_uploader.GithubModelUploader(model_id: str,
                                                                           access_token: str)
```

Bases: [*medigan.contribute_model.base_model_uploader.BaseModelUploader*](#)

GithubModelUploader class: Pushes the metadata of a user’s model to the medigan repo, where it creates a dedicated github issue.

Parameters

- **model_id** (*str*) – The generative model’s unique id
- **access_token** (*str*) – a personal access token linked to your github user account, used as means of authentication

model_id

The generative model’s unique id

Type *str*

access_token

a personal access token linked to your github user account, used as means of authentication

Type *str*

```
add_package_link_to_metadata(metadata: dict, package_link: Optional[str] = None, is_update_forced:
                             bool = False) → dict
```

Update *package_link* in the model’s metadata if current *package_link* does not containing a valid url.

Parameters

- **metadata** (*dict*) – The model’s corresponding metadata
- **package_link** (*str*) – the new package link to used to replace the old one
- **is_update_forced** (*bool*) – flag to update metadata even though metadata already contains a valid url in its *package_link*

Returns Returns the updated metadata dict with replaced *package_link* if replacement was applicable.

Return type *dict*

```
push(metadata: dict, package_link: Optional[str] = None, creator_name: str = 'n.a.', creator_affiliation: str = 'n.a.', model_description: str = 'n.a.')
```

Upload the model's metadata inside a github issue to the medigan github repository.

To add your model to medigan, your metadata will be reviewed on Github and added to medigan's official model metadata

The medigan repository issues page: <https://github.com/RichardObi/medigan/issues>

Get your Github access token here: <https://github.com/settings/tokens>

Parameters

- **metadata** (*dict*) – The model's corresponding metadata
- **package_link** – a package link
- **creator_name** (*str*) – the creator name that will appear on the corresponding github issue
- **creator_affiliation** (*str*) – the creator affiliation that will appear on the corresponding github issue
- **model_description** (*list*) – the model_description that will appear on the corresponding github issue

Returns Returns the url pointing to the corresponding issue on github

Return type str

medigan.contribute_model.model_contributor module

Model contributor class that tests models, creates metadata entries, uploads and contributes them to medigan.

```
class medigan.contribute_model.model_contributor.ModelContributor(model_id: str, init_py_path: str)
```

Bases: object

ModelContributor class: Contributes a user's local model to the public medigan library

Parameters

- **model_id** (*str*) – The generative model's unique id
- **init_py_path** (*str*) – The path to the local model's `__init__.py` file needed for importing and running this model.

model_id

The generative model's unique id

Type str

init_py_path

The path to the local model's `__init__.py` file needed for importing and running this model.

Type str

package_path

Path as string to the generative model's python package

Type str

package_name

Name of the model's python package i.e. the name of the model's zip file and unzipped package folder

Type str

metadata_file_path

Path as string to the generative model's metadata file e.g. default is relative path to package root.

Type str

zenodo_model_uploader

An instance of the *ZenodoModelUploader* class

Type str

github_model_uploader

An instance of the *GithubModelUploader* class.

Type str

add_metadata_from_file(*metadata_file_path*) → dict

Read and parse the metadata of a local model, identified by *model_id*, from a metadata file in json format.

Parameters

- **model_id** (*str*) – The generative model's unique id
- **metadata_file_path** (*str*) – the path pointing to the metadata file

Returns Returns a dict containing the contents of parsed metadata json file.

Return type dict

add_metadata_from_input(*model_weights_name*: Optional[*str*] = None, *model_weights_extension*: Optional[*str*] = None, *generate_method_name*: Optional[*str*] = None, *dependencies*: list = [], *fill_more_fields_interactively*: bool = True, *output_path*: *str* = 'config')

Create a metadata dict for a local model, identified by *model_id*, given the necessary minimum metadata contents.

Parameters

- **model_id** (*str*) – The generative model's unique id
- **model_weights_name** (*str*) – the name of the checkpoint file containing the model's weights
- **model_weights_extension** (*str*) – the extension (e.g. .pt) of the checkpoint file containing the model's weights
- **generate_method_name** (*str*) – the name of the sample generation method inside the models `__init__.py` file
- **dependencies** (*list*) – the list of dependencies that need to be installed via pip to run the model.
- **fill_more_fields_interactively** (*bool*) – flag indicating whether a user will be interactively asked via command line for further input to fill out missing metadata content
- **output_path** (*str*) – the path where the created metadata json file will be stored.

Returns Returns a dict containing the contents of the metadata json file.

Return type dict

is_value_for_key_already_set(*key: str, metadata: dict, nested_key*) → bool

Check if the value of a *key* in a *metadata* dictionary is already set and e.g. not an empty string, dict or list.

Parameters

- **key** (*str*) – The key in the currently traversed part of the model’s metadata dictionary
- **metadata** (*dict*) – The currently traversed part of the model’s metadata dictionary
- **nested_key** (*str*) – the *nested_key* indicates which subpart of the model’s metadata we are currently traversing

Returns Flag indicating whether a value exists for the *key* in the dict

Return type bool

load_metadata_template() → dict

Loads and parses (json to dict) a default medigan metadata template.

Returns Returns the metadata template as dict

Return type dict

push_to_github(*access_token: str, package_link: Optional[str] = None, creator_name: str = "", creator_affiliation: str = "", model_description: str = ""*)

Upload the model’s metadata inside a github issue to the medigan github repository.

To add your model to medigan, your metadata will be reviewed on Github and added to medigan’s official model metadata

The medigan repository issues page: <https://github.com/RichardObi/medigan/issues>

Get your Github access token here: <https://github.com/settings/tokens>

Parameters

- **access_token** (*str*) – a personal access token linked to your github user account, used as means of authentication
- **package_link** – a package link
- **creator_name** (*str*) – the creator name that will appear on the corresponding github issue
- **creator_affiliation** (*str*) – the creator affiliation that will appear on the corresponding github issue
- **model_description** (*list*) – the *model_description* that will appear on the corresponding github issue

Returns Returns the url pointing to the corresponding issue on github

Return type str

push_to_zenodo(*access_token: str, creator_name: str, creator_affiliation: str, model_description: str = ""*)

Upload the model files as zip archive to a public Zenodo repository where the model will be persistently stored.

Get your Zenodo access token here: <https://zenodo.org/account/settings/applications/tokens/new/> (Enable scopes *deposit:actions* and *deposit:write*)

Parameters

- **access_token** (*str*) – a personal access token in Zenodo linked to a user account for authentication

- **creator_name** (*str*) – the creator name that will appear on the corresponding Zenodo model upload homepage
- **creator_affiliation** (*str*) – the creator affiliation that will appear on the corresponding Zenodo model upload homepage
- **model_description** (*list*) – the model_description that will appear on the corresponding Zenodo model upload homepage

Returns Returns the url pointing to the corresponding Zenodo model upload homepage

Return type *str*

validate_and_update_model_weights_path() → *dict*

Check if the model files can be found in the *package_path* or based on the *path_to_metadata*.

Ideally, the user provided *package_path* and the *path_to_metadata* should both point to the same model package containing weights, config, license, etc. Here we check both of these paths to find the model weights.

Returns Returns the metadata after updating the path to the model's checkpoint's weights

Return type *dict*

validate_init_py_path(*init_py_path*) → *bool*

Asserts whether the *init_py_path* exists and points to a valid *__init__.py* correct file.

Parameters **init_py_path** (*str*) – The path to the local model's *__init__.py* file needed for importing and running this model.

validate_local_model_import()

Check if the model package in the *package_path* can be imported as python library using *importlib*.

validate_model_id(*model_id: str, max_chars: int = 30, min_chars: int = 13*) → *bool*

Asserts if the *model_id* is in the correct format and has a valid length

Parameters

- **model_id** (*str*) – The generative model's unique id
- **max_chars** (*int*) – the maximum of chars allowed in the *model_id*
- **min_chars** (*int*) – the minimum of chars allowed in the *model_id*

Returns Returns flag indicating whether the *model_id* is correctly formatted.

Return type *bool*

medigan.contribute_model.zenodo_model_uploader module

Zenodo Model uploader class that uploads models to medigan associated data storage on Zenodo.

class `medigan.contribute_model.zenodo_model_uploader.ZenodoModelUploader`(*model_id*,
access_token)

Bases: `medigan.contribute_model.base_model_uploader.BaseModelUploader`

ZenodoModelUploader class: Uploads a user's model via API to Zenodo, here it is permanently stored with DOI.

Parameters

- **model_id** (*str*) – The generative model's unique id

- **access_token** (*str*) – a personal access token in Zenodo linked to a user account for authentication

model_id

The generative model's unique id

Type *str*

access_token

a personal access token in Zenodo linked to a user account for authentication

Type *str*

create_upload_description(*metadata: dict, model_description: str = ""*) → *str*

Create a string containing the textual description that will accompany the upload model files.

The string contains tags and a text retrieved from the description subsection of the model metadata.

Parameters

- **metadata** (*dict*) – The model's corresponding metadata
- **model_description** (*str*) – the model_description that will appear on the corresponding Zenodo model upload homepage

Returns Returns the textual description of the model upload

Return type *str*

create_upload_json_data(*creator_name: str, creator_affiliation: str, description: str = ""*) → *dict*

Create some descriptive data in dict format to be uploaded and stored alongside the model files.

Parameters

- **creator_name** (*str*) – the creator name that will appear on the corresponding Zenodo model upload homepage
- **creator_affiliation** (*str*) – the creator affiliation that will appear on the corresponding Zenodo model upload homepage
- **description** (*str*) – the model_description that will appear on the corresponding Zenodo model upload homepage

Returns Returns the descriptive data in dictionary structure describing the model upload.

Return type *dict*

empty_upload() → *dict*

Upload an empty placeholder entry to Zenodo as is required to retrieve a *deposition_id* and *bucket_url*.

deposition_id and *bucket_url* are needed for file upload and publishing in the subsequent upload steps.

Returns Returns the response retrieved via the Zenodo API

Return type *dict*

locate_or_create_model_zip_file(*package_path: str, package_name: str*) -> (*<class 'str'>*, *<class 'str'>*)

If not possible to locate, create a zipped python package of the model.

Parameters

- **package_path** (*str*) – Path as string to the generative model's python package containing an *__init__.py* file

- **package_name** (*str*) – Name of the model’s python package i.e. the name of the model’s zip file and unzipped package folder

Returns Returns a tuple containing two strings: The *filename* and the *file_path* of and to the zipped python package

Return type tuple

publish(*deposition_id: str*) → dict

Publish a zenodo upload.

This makes the upload official, as it will then be publicly accessible and persistently stored on Zenodo with associated DOI.

Parameters **deposition_id** (*str*) – The deposition id assigned by Zenodo to the uploaded model file

Returns Returns the response retrieved via the Zenodo API

Return type dict

push(*metadata: dict, package_path: str, package_name: str, creator_name: str, creator_affiliation: str, model_description: str = ""*)

Upload the model files as zip archive to a public Zenodo repository where the model will be persistently stored.

Get your Zenodo access token here: <https://zenodo.org/account/settings/applications/tokens/new/> (Enable scopes *deposit:actions* and *deposit:write*)

Parameters

- **metadata** (*dict*) – The model’s corresponding metadata
- **package_path** (*dict*) – The path to the packaged model files
- **package_name** (*dict*) – The name of the packaged model files
- **creator_name** (*str*) – the creator name that will appear on the corresponding Zenodo model upload homepage
- **creator_affiliation** (*str*) – the creator affiliation that will appear on the corresponding Zenodo model upload homepage
- **model_description** (*list*) – the model_description that will appear on the corresponding Zenodo model upload homepage

Returns Returns the url pointing to the corresponding Zenodo model upload homepage

Return type str

upload(*file_path: str, filename: str, bucket_url: str*) → dict

Upload a file to Zenodo entry of the uploaded model files.

Parameters

- **file_path** (*str*) – The path of the file that is uploaded to Zenodo
- **filename** (*str*) – The name of the file that is uploaded to Zenodo
- **bucket_url** (*str*) – The bucket url used in the PUT request to upload the data file.

Returns Returns the response retrieved via the Zenodo API

Return type dict

upload_descriptive_data(*deposition_id: str, data: dict*) → dict

Upload textual descriptive data to be associated and added to the Zenodo entry of the uploaded model files.

Parameters

- **deposition_id** (*str*) – The deposition id assigned by Zenodo to the uploaded model file
- **data** (*dict*) – The descriptive information that will to be uploaded to Zenodo and associated with the `deposition_id`

Returns Returns the response retrieved via the Zenodo API

Return type dict

Module contents

medigan.execute_model package

Submodules

medigan.execute_model.install_model_dependencies module

Functionality for automated installation of a model's python package dependencies.

`medigan.execute_model.install_model_dependencies.install_model`(*model_id: str, config_manager: Optional[medigan.config_manager.ConfigManager] = None, execution_config: Optional[dict] = None*)

installing the dependencies required for this model as stated in config

`medigan.execute_model.install_model_dependencies.parse_args`() → `argparse.Namespace`

medigan.execute_model.model_executor module

Model executor class that downloads models, loads them as python packages, and runs their generate functions.

class `medigan.execute_model.model_executor.ModelExecutor`(*model_id: str, execution_config: dict, download_package: bool = True, install_dependencies: bool = False*)

Bases: `object`

ModelExecutor class: Find config links to download models, init models as python packages, run generate methods.

Parameters

- **model_id** (*str*) – The generative model's unique id
- **execution_config** (*dict*) – The part of the config below the 'execution' key
- **download_package** (*bool*) – Flag indicating, if True, that the model's package should be downloaded instead of using an existing one that was downloaded previously
- **install_dependencies** (*bool*) – flag indicating whether a generative model's dependencies are automatically installed. Else error is raised if missing dependencies are detected.

model_id

The generative model's unique id

Type str

execution_config

The part of the config below the 'execution' key

Type dict

download_package

Flag indicating, if True, that the model's package should be downloaded instead of using an existing one that was downloaded previously

Type bool

image_size

Pixel dimension of the generated samples, where images are assumed to have the same width and height

Type int

dependencies

List of the dependencies of a models python package.

Type list

model_name

Name of the generative model

Type str

model_extension

File extension of the generative model's weights file.

Type str

package_name

Name of the model's python package i.e. the name of the model's zip file and unzipped package folder

Type str

package_link

The link to the zipped model package. Note: Convention is to host models on Zenodo (reason: static doi content)

Type str

generate_method_name

The name of the model's generate method inside the model package. This method is called to generate samples.

Type str

generate_method_args

The args of the model's generate method inside the model package

Type dict

serialised_model_file_path

Path as string to the generative model's weights file

Type str

package_path

Path as string to the generative model's python package containing an `__init__.py` file

Type str

deserialized_model_as_lib

The generative model's package imported as python library. Generate method inside this library can be called.

generate(*num_samples: int = 20, output_path: Optional[str] = None, save_images: bool = True, is_gen_function_returned: bool = False, batch_size: int = 32, **kwargs*)

Generate samples using the generative model or return the model's generate function.

The name and additional parameters of the generate function of the respective generative model are retrieved from the model's *execution_config*.

Parameters

- **num_samples** (*int*) – the number of samples that will be generated
- **output_path** (*str*) – the path as str to the output folder where the generated samples will be stored
- **save_images** (*bool*) – flag indicating whether generated samples are returned (i.e. as list of numpy arrays) or rather stored in file system (i.e in *output_path*)
- **is_gen_function_returned** (*bool*) – flag indicating whether, instead of generating samples, the sample generation function will be returned
- **batch_size** (*int*) – the batch size for the sample generation function
- ****kwargs** – arbitrary number of keyword arguments passed to the model's sample generation function

Returns Returns images as list of numpy arrays if *save_images* is False. However, if *is_gen_function_returned* is True, it returns the internal generate function of the model.

Return type list

Raises Exception – If the generate method of the model does not exist, cannot be called, or is called with missing params, or if the sample generation inside the model package returns an exception.

is_model_already_unpacked() → bool

Check if a valid path to the model files exists and, if so, set the *package_path*

medigan.execute_model.synthetic_dataset module

SyntheticDataset allows to return a generative model as torch dataset.

class medigan.execute_model.synthetic_dataset.**SyntheticDataset**(*samples, masks=None, other_imaging_output=None, labels=None, transform=None*)

Bases: torch.utils.data.dataset.Dataset

A synthetic dataset containing data generated by a model of medigan

Parameters

- **samples** (*list*) – List of data points in the dataset e.g. generated images as numpy array.
- **masks** (*list*) – List of segmentation masks, if applicable, pertaining to the *samples* items

- **other_imaging_output** (*list*) – List of other imaging output produced by the generative model (e.g. specific types of other masks/modalities)
- **labels** (*list*) – list of labels, if applicable, pertaining to the *samples* items
- **transform** – torch compose transform functions that are applied to the torch dataset.

samples

List of data points in the dataset e.g. generated images as numpy array.

Type list

masks

List of segmentation masks, if applicable, pertaining to the *samples* items

Type list

other_imaging_output

List of other imaging output produced by the generative model (e.g. specific types of other masks/modalities)

Type list

labels

list of labels, if applicable, pertaining to the *samples* items

Type list

transform

torch compose transform functions that are applied to the torch dataset.

Module contents

medigan.select_model package

Submodules

medigan.select_model.matched_entry module

MatchedEntry class that represents one match of a key value pair of a model's config against a search query.

class medigan.select_model.matched_entry.**MatchedEntry**(*key: str, value, matching_element: Optional[str] = None*)

Bases: object

MatchedEntry class: One target key-value pair that matches with a model's selection config.

Parameters

- **key** (*str*) – string that represents the matched key in model selection dict
- **value** – represents the key's matched value in the model selection dict
- **matching_element** (*str*) – string that was used to match the search value

key

string that represents the matched key in model selection dict

Type str

value

represents the key's matched value in the model selection dict

matching_element

string that was used to match the search value

Type str

medigan.select_model.model_match_candidate module

ModelMatchCandidate class that holds data to evaluate if a generative model matches against a search query.

```
class medigan.select_model.model_match_candidate.ModelMatchCandidate(model_id: str,
                                                                    target_values: list,
                                                                    target_values_operator:
                                                                    str = 'AND',
                                                                    is_case_sensitive: bool =
                                                                    False,
                                                                    are_keys_also_matched:
                                                                    bool = False, is_match:
                                                                    bool = False)
```

Bases: object

ModelMatchCandidate class: A prospectively matching model given the target values as model search params.

Parameters

- **model_id** (*str*) – The generative model's unique id
- **target_values** (*list*) – list of target values used to evaluate if a *ModelMatchCandidate* instance is a match
- **target_values_operator** (*str*) – the operator indicating the relationship between *values* in the evaluation of *ModelMatchCandidate* instances. Should be either “AND”, “OR”, or “XOR”.
- **is_case_sensitive** (*bool*) – flag indicating whether the matching of *values* (and) keys should be case-sensitive
- **are_keys_also_matched** (*bool*) – flag indicating whether, apart from *values*, keys should also be matched
- **is_match** (*bool*) – flag indicating whether the *ModelMatchCandidate* instance is a match

model_id

The generative model's unique id

Type str

target_values

list of target values used to evaluate if a *ModelMatchCandidate* instance is a match

Type list

target_values_operator

the operator indicating the relationship between *values* in the evaluation of *ModelMatchCandidate* instances. Should be either “AND”, “OR”, or “XOR”.

Type str

is_case_sensitive

flag indicating whether the matching of *values* (and) keys should be case-sensitive

Type bool

are_keys_also_matched

flag indicating whether, apart from values, keys should also be matched

Type bool

matched_entries

contains iteratively added *MatchedEntry* class instances. Each of the *MatchedEntry* instances indicates a match between one of the user specified values in *self.target_values* and the selection config keys or *values* of the model of this *ModelMatchCandidate*.

Type list

is_match

flag indicating whether the *ModelMatchCandidate* instance is a match

Type bool

add_matched_entry(*matched_entry*: [medigan.select_model.matched_entry.MatchedEntry](#)) → None

Add a *MatchedEntry* instance to the *matched_entries* list.

check_if_is_match() → bool

Evaluates whether the model represented by this instance is a match given search values and operator.

The matching element from each *MatchEntry* of this instance ('self.matched_entries') are retrieved. To be a match, this instance ('self') needs to fulfill the requirement of the operator, which can be of value 'AND', or 'OR', or 'XOR'. For example, the default 'AND' requires that each search value ('self.target_values') has at least one corresponding *MatchEntry*, while in 'OR' only one of the search values needs to have been matched by a corresponding *MatchedEntry*.

Returns flag that, only if True, indicates that this instance is a match given the search values and operator.

Return type bool

get_all_matching_elements() → list

Get the matching element from each of the *MatchedEntry* instances in the *matched_entries* list.

Returns list of all matching elements (i.e. string that matched a search value) from each *MatchedEntry* in *matched_entries*

Return type list

medigan.select_model.model_selector module

Model selection class that describes, finds, compares, and ranks generative models.

class `medigan.select_model.model_selector.ModelSelector`(*config_manager*: *Optional*[[medigan.config_manager.ConfigManager](#)] = None)

Bases: object

ModelSelector class: Given a config dict, gets, searches, and ranks matching models.

Parameters **config_manager** ([ConfigManager](#)) – Provides the config dictionary, based on which models are selected and compared.

config_manager

Provides the config dictionary, based on which models are selected and compared.

Type *ConfigManager*

model_selection_dicts

Contains a dictionary for each model id that consists of the *model_id* and the selection config of that model

Type list

find_matching_models_by_values(*values*: list, *target_values_operator*: str = 'AND',
are_keys_also_matched: bool = False, *is_case_sensitive*: bool =
 False) → list

Search for values (and keys) in model configs and return a list of each matching *ModelMatchCandidate*.

Uses *self.recursive_search_for_values* to recursively populate each *ModelMatchCandidate* with *MatchedEntry* instances. After populating, each *ModelMatchCandidate* is evaluated based on the provided *target_values_operator* and *values* list using *ModelMatchCandidate.check_if_is_match*.

Parameters

- **values** (*list*) – list of values used to search and find models corresponding to these values
- **target_values_operator** (*str*) – the operator indicating the relationship between *values* in the evaluation of model search results. Should be either “AND”, “OR”, or “XOR”.
- **are_keys_also_matched** (*bool*) – flag indicating whether, apart from values, the keys in the model config should also be searchable
- **is_case_sensitive** (*bool*) – flag indicating whether the search for values (and) keys in the model config should be case-sensitive.

Returns a list of *ModelMatchCandidate* class instances each of which was successfully matched against the search values.

Return type list

find_models_and_rank(*values*: list, *target_values_operator*: str = 'AND', *are_keys_also_matched*: bool =
 False, *is_case_sensitive*: bool = False, *metric*: str = 'SSIM', *order*: str = 'asc') →
 list

Search for values (and keys) in model configs, rank results and return sorted list of model dicts.

Parameters

- **values** (*list*) – list of values used to search and find models corresponding to these *values*
- **target_values_operator** (*str*) – the operator indicating the relationship between *values* in the evaluation of model search results. Should be either “AND”, “OR”, or “XOR”.
- **are_keys_also_matched** (*bool*) – flag indicating whether, apart from *values*, the keys in the model config should also be searchable
- **is_case_sensitive** (*bool*) – flag indicating whether the search for values (and) keys in the model config should be case-sensitive.
- **metric** (*str*) – The key in the selection dict that corresponds to the *metric* of interest
- **order** (*str*) – the sorting order of the ranked results. Should be either “asc” (ascending) or “desc” (descending)

Returns a list of the searched and matched model dictionaries containing *metric* and *model_id*, sorted by *metric*.

Return type list

get_models_by_key_value_pair(*key1*: str, *value1*: str, *is_case_sensitive*: bool = False) → list

Get and return a list of *model_id* dicts that contain the specified key value pair in their selection config.

The key param can contain ‘.’ (dot) separations to allow for retrieval of nested config keys such as ‘execution.generator.name’. If *key1* points to a list, any value in the list that matches *value1* is accepted.

Parameters

- **key1** (str) – The key in the selection dict
- **value1** (str) – The value in the selection dict that corresponds to *key1*
- **is_case_sensitive** (bool) – flag to evaluate keys and values with case sensitivity if set to True

Returns a list of the dictionaries each containing a model’s *model_id* and the found key-value pair in the models config

Return type list

get_selection_criteria_by_id(*model_id*: str, *is_model_id_removed*: bool = True) → dict

Get and return the selection config dict for a specific *model_id*.

Parameters

- **model_id** (str) – The generative model’s unique id
- **is_model_id_removed** (bool) – flag to to remove the *model_id* from first level of each dictionary.

Returns a dictionary corresponding to the selection config of a model

Return type dict

get_selection_criteria_by_ids(*model_ids*: Optional[list] = None, *are_model_ids_removed*: bool = True) → list

Get and return a list of selection config dicts for each of the specified *model_ids*.

Parameters

- **model_ids** (list) – A list of generative models’ unique ids
- **are_model_ids_removed** (bool) – flag to remove the *model_ids* from first level of dictionary.

Returns a list of dictionaries each corresponding to the selection config of a model

Return type list

get_selection_keys(*model_id*: Optional[str] = None) → list

Get and return all first level keys from the selection config dict for a specific *model_id*.

Parameters **model_id** (str) – The generative model’s unique id

Returns a list containing the keys as strings of the selection config of the *model_id*.

Return type list

get_selection_values_for_key(*key*: str, *model_id*: Optional[str] = None) → list

Get and return the value of a specified key of the selection dict in the config for a specific *model_id*.

The key param can contain ‘.’ (dot) separations to allow for retrieval of nested config keys such as ‘execution.generator.name’

Parameters

- **key** (*str*) – The key in the selection dict
- **model_id** (*str*) – The generative model’s unique id

Returns a list of the values that correspond to the key in the selection config of the *model_id*.

Return type list

rank_models_by_performance(*model_ids: Optional[list] = None, metric: str = 'SSIM', order: str = 'asc'*)
→ list

Rank model based on a provided metric and return sorted list of model dicts.

The metric param can contain ‘.’ (dot) separations to allow for retrieval via nested metric config keys such as ‘downstream_task.CLF.accuracy’. If the value found for the metric key is of type list, then the largest value in the list is used for ranking if *order* is descending, while the smallest value is used if *order* is ascending.

Parameters

- **model_ids** (*list*) – only evaluate the *model_ids* in this list. If none, evaluate all available *model_ids*
- **metric** (*str*) – The key in the selection dict that corresponds to the metric of interest
- **order** (*str*) – the sorting order of the ranked results. Should be either “asc” (ascending) or “desc” (descending)

Returns a list of model dictionaries containing metric and *model_id*, sorted by *metric*.

Return type list

recursive_search_for_values(*search_dict: dict, model_match_candidate:*
medigan.select_model.model_match_candidate.ModelMatchCandidate)

Do a recursive search to match values in the *search_dict* with values (and keys) in a model’s config.

The provided *ModelMatchCandidate* instance is recursively populated with *MatchedEntry* instances. A *MatchedEntry* instance contains a key-value pair found in the model’s config that matches with one search term of interest.

The search terms of interest are stored in *ModelMatchCandidate.target_values*. The model’s selection config is provided in the ‘search_dict’.

To traverse the *search_dict*, the value for each key in the *search_dict* is retrieved.

- If that value is of type dictionary, the function calls itself with that nested dictionary as new *search_dict*.
- If that value is of type list, each value in the list is compared with each search term of interest in *ModelMatchCandidate.target_values*.
- If the value of the *search_dict* is of another type (i.e. *str*), it is compared with each search term of interest in *ModelMatchCandidate.target_values*.

Parameters

- **search_dict** (*dict*) – contains keys and values from a model’s config that are matched against a set of search values.
- **model_match_candidate** (*ModelMatchCandidate*) – a class instance representing a model to be prepared for evaluation (populated with *MatchedEntry* objects), as to whether it is a match given its search values (*self.target_values*).

Returns a *ModelMatchCandidate* class instance that has been populated with *MatchedEntry* class instances.

Return type list

Module contents

Submodules

medigan.config_manager module

Config manager class that downloads, ingests, parses, and prepares the config information for all models.

class medigan.config_manager.**ConfigManager**(*config_dict: Optional[dict] = None*,
is_new_download_forced: bool = False)

Bases: object

ConfigManager class: Downloads, loads and parses medigan’s config json as dictionary.

Parameters

- **config_dict** (*dict*) – Optionally provides the config dictionary if already loaded and parsed in a previous process.
- **is_new_download_forced** (*bool*) – Flags, if True, that a new config file should be downloaded from the config link instead of parsing an existing file.

config_dict

Optionally provides the config dictionary if already loaded and parsed in a previous process.

Type dict

is_new_download_forced

Flags, if True, that a new config file should be downloaded from the config link instead of parsing an existing file.

Type bool

model_ids

Lists the unique id’s of the generative models specified in the *config_dict*

Type list

is_config_loaded

Flags if the loading and parsing of the config file was successful (True) or not (False).

Type bool

add_model_to_config(*model_id: str, metadata: dict, is_local_model: bool = True*,
overwrite_existing_metadata: bool = False, store_new_config: bool = True) → bool

Adding or updating a model entry in the global metadata.

Parameters

- **model_id** (*str*) – The generative model’s unique id
- **metadata** (*dict*) – The model’s corresponding metadata
- **is_local_model** (*bool*) – flag indicating whether the tested model is a new local user model i.e not yet part of medigan’s official models
- **overwrite_existing_metadata** (*bool*) – in case of *is_local_model*, flag indicating whether existing metadata for this model in medigan’s *config/global.json* should be overwritten.

- **store_new_config** (*bool*) – flag indicating whether the current model metadata should be stored on disk i.e. in config/

Returns Flag indicating whether model metadata update was successfully concluded

Return type bool

get_config_by_id(*model_id: str, config_key: Optional[str] = None*) → dict

From *config_manager*, get and return the part of the config below a *config_key* for a specific *model_id*.

The key param can contain ‘.’ (dot) separations to allow for retrieval of nested config keys such as ‘execution.generator.name’

Parameters

- **model_id** (*str*) – The generative model’s unique id
- **config_key** (*str*) – A key of interest present in the config dict

Returns a dictionary from the part of the config file corresponding to *model_id* and *config_key*.

Return type dict

is_model_in_config(*model_id: str*) → bool

Checking if a *model_id* is present in the global model metadata file

Parameters **model_id** (*str*) – The generative model’s unique id

Returns Flag indicating whether a *model_id* is present in global model metadata

Return type bool

is_model_metadata_valid(*model_id: str, metadata: dict, is_local_model: bool = True*) → bool

Checking if a model’s corresponding metadata is valid.

Specific fields in the model’s metadata are mandatory. It is asserted if these key value pairs are present.

Parameters

- **model_id** (*str*) – The generative model’s unique id
- **metadata** (*dict*) – The model’s corresponding metadata
- **is_local_model** (*bool*) – flag indicating whether the tested model is a new local user model i.e not yet part of medigan’s official models

Returns Flag indicating whether the specific model’s metadata format and fields are valid

Return type bool

load_config_file(*is_new_download_forced: bool = False*) → bool

Load a config file and return boolean flag indicating success of loading process.

If the config file is not present in *medigan.CONSTANTS.CONFIG_FILE_FOLDER*, it is per default downloaded from the web resource specified in *medigan.CONSTANTS.CONFIG_FILE_URL*.

Parameters **is_new_download_forced** (*bool*) – Forces new download of config file even if the file has been downloaded before.

Returns a boolean flag indicating true only if the config file was loaded successfully.

Return type bool

match_model_id(*provided_model_id: str*) → bool

Replacing a model_id acronym (e.g. 00005 or 5) with the unique *model_id* present in the model metadata

Parameters **provided_model_id** (*str*) – The user-provided model_id that might be shorter (e.g. “00005” or “5”) than the real unique model id

Returns If matched, returning the unique *model_id* present in global model metadata.

Return type str

medigan.constants module

Global constants of the medigan library

medigan.constants.CONFIG_FILE_FOLDER = 'config'

Name and extensions of config file.

medigan.constants.CONFIG_FILE_KEY_DEPENDENCIES = 'dependencies'

Below the execution dict, the key under which the package link of a model is present in the config file. Note: The model packages are per convention stored on Zenodo where they retrieve a static DOI avoiding security issues due to static non-modifiable content on Zenodo. Zenodo also helps to maintain clarity of who the owners and contributors of each generative model (and its IP) in medigan are.

medigan.constants.CONFIG_FILE_KEY_DESCRIPTION = 'description'

Below the selection dict, the key under which the performance dictionary of a model is nested in the config file.

medigan.constants.CONFIG_FILE_KEY_EXECUTION = 'execution'

The key under which the selection dictionary of a model is nested in the config file.

medigan.constants.CONFIG_FILE_KEY_GENERATE = 'generate_method'

Below the execution dict, the key under which the exact name of a model's generate() function is present.

medigan.constants.CONFIG_FILE_KEY_GENERATE_ARGS = 'args'

Below the execution dict, the key under which an array of mandatory base arguments of any model's generate() function is present.

medigan.constants.CONFIG_FILE_KEY_GENERATE_ARGS_BASE = 'base'

Below the execution dict, the key under which a nested dict of key-value pairs of model specific custom arguments of a model's generate() function are present.

medigan.constants.CONFIG_FILE_KEY_GENERATE_ARGS_CUSTOM = 'custom'

Below the execution dict, the key under which the model_file argument value of any model's generate() function is present.

medigan.constants.CONFIG_FILE_KEY_GENERATE_ARGS_INPUT_LATENT_VECTOR_SIZE = 'input_latent_vector_size'

Below the selectoin dict, the key under which the tags (list of strings) is present.

medigan.constants.CONFIG_FILE_KEY_GENERATE_ARGS_MODEL_FILE = 'model_file'

Below the execution dict, the key under which the num_samples argument value of any model's generate() function is present.

medigan.constants.CONFIG_FILE_KEY_GENERATE_ARGS_NUM_SAMPLES = 'num_samples'

Below the execution dict, the key under which the output_path argument value of any model's generate() function is present.

`medigan.constants.CONFIG_FILE_KEY_GENERATE_ARGS_OUTPUT_PATH = 'output_path'`

Below the execution dict, the key under which the save images boolean flag argument value of any model's `generate()` function is present.

`medigan.constants.CONFIG_FILE_KEY_GENERATE_ARGS_SAVE_IMAGES = 'save_images'`

Below the execution dict, the key under which the random input_latent_vector_size argument value of model's `generate()` function is present.

`medigan.constants.CONFIG_FILE_KEY_GENERATE_NAME = 'name'`

Below the execution dict, the key under which a nested dict with info on the arguments of a model's `generate()` function is present.

`medigan.constants.CONFIG_FILE_KEY_GENERATOR = 'generator'`

Below the execution dict, the key under which a model's generator's is present in the config file.

`medigan.constants.CONFIG_FILE_KEY_GENERATOR_NAME = 'name'`

Below the execution dict, the key under which a model's image_size is present in the config file.

`medigan.constants.CONFIG_FILE_KEY_IMAGE_SIZE = 'image_size'`

Below the execution dict, the key under which a model's name is present in the config file. This is the name of the weights file!

`medigan.constants.CONFIG_FILE_KEY_MODEL_EXTENSION = 'extension'`

Below the execution dict, the key under which the package_name of a model is present in the config file.

`medigan.constants.CONFIG_FILE_KEY_MODEL_NAME = 'model_name'`

Below the execution dict, the key under which a nested dict with info on the model's `generate()` function is present.

`medigan.constants.CONFIG_FILE_KEY_PACKAGE_LINK = 'package_link'`

Below the execution dict, the key under which the extension of a model is present in the config file.

`medigan.constants.CONFIG_FILE_KEY_PACKAGE_NAME = 'package_name'`

Below the execution dict, the key under which the package_name of a model is present in the config file.

`medigan.constants.CONFIG_FILE_KEY_PERFORMANCE = 'performance'`

Below the execution dict, the key under which the dependencies dictionary of a model is nested in the config file.

`medigan.constants.CONFIG_FILE_KEY_SELECTION = 'selection'`

The key under which the description dictionary of a model is nested in the config file.

`medigan.constants.CONFIG_FILE_KEY_TAGS = 'tags'`

The filetype of any of the generative model's python packages after download and before unpacking.

`medigan.constants.CONFIG_FILE_NAME_AND_EXTENSION = 'global.json'`

The key under which the execution dictionary of a model is nested in the config file.

`medigan.constants.CONFIG_FILE_URL =`

`'https://raw.githubusercontent.com/RichardObi/medigan/main/config/global.json'`

Folder path that will be created to locally store the config file.

`medigan.constants.CONFIG_TEMPLATE_FILE_NAME_AND_EXTENSION = 'template.json'`

Download link to template.json file.

`medigan.constants.CONFIG_TEMPLATE_FILE_URL =`

`'https://raw.githubusercontent.com/RichardObi/medigan/main/templates/template.json'`

Name and extensions of template of config file.

`medigan.constants.DEFAULT_OUTPUT_FOLDER = 'output'`

The folder containing an `__init__.py` file is a python module.

`medigan.constants.GITHUB_REPO = 'RichardObi/medigan'`

The assignee of the Github Issue when adding a model to medigan

`medigan.constants.GITHUB_TITLE = 'Model Integration Request for medigan'`

The repository of the Github Issue when adding a model to medigan

`medigan.constants.INIT_PY_FILE = '__init__.py'`

Name and extensions of template of config file.

`medigan.constants.MODEL_FOLDER = 'models'`

To add a model, please create pull request in this github repo.

Type Static link to the config of medigan. Note

`medigan.constants.MODEL_ID = 'model_id'`

The default path to a folder under which the outputs of the medigan package (i.e. generated samples) are stored.

`medigan.constants.PACKAGE_EXTENSION = '.zip'`

The string describing a model's unique id in medigan's data structures.

`medigan.constants.TEMPLATE_FOLDER = 'templates'`

The line break in the Zenodo description that appears together with the pushed model on Zenodo

`medigan.constants.ZENODO_API_URL = 'https://zenodo.org/api/deposit/depositions'`

The HEADER for Zenodo REST API requests

`medigan.constants.ZENODO_GENERIC_MODEL_DESCRIPTION = "<p>Usage:</p><p>This GAN is used as part of the medigan library. This GANs metadata is therefore stored in and retrieved from medigan's config file. medigan is an open-source Python library on Github that allows developers and researchers to easily add synthetic imaging data into their model training pipelines. medigan is documented here and can be used via pip install:</p><pre><code class='language-python'>pip install medigan</code></pre><p>To run this model in medigan, use the following commands.</p><pre><code class='language-python'> from medigan import Generators </code></pre><pre><code class='language-python'> generators = Generators() </code></pre><pre><code class='language-python'> generators.generate(model_id='YOUR_MODEL_ID', num_samples=10)</code></pre><p> </p>"`

The REST API to interact with Zenodo

`medigan.constants.ZENODO_HEADERS = {'Content-Type': 'application/json'}`

The title of the Github Issue when adding a model to medigan

`medigan.constants.ZENODO_LINE_BREAK = '<p> </p>'`

A generic description appended to model uploads that are automatically uploaded to zenodo via Zenodo API call in medigan

medigan.exceptions module

Custom exceptions to handle module specific error and facilitate bug fixes and debugging.

medigan.generators module

Base class providing user-library interaction methods for config management, and model selection and execution.

```
class medigan.generators.Generators(config_manager: Optional[medigan.config_manager.ConfigManager]
                                     = None, model_selector:
                                     Optional[medigan.select_model.model_selector.ModelSelector] =
                                     None, model_executors: Optional[list] = None, model_contributors:
                                     Optional[list] = None, initialize_all_models: bool = False)
```

Bases: object

Generators class: Contains medigan's public methods to facilitate users' automated sample generation workflows.

Parameters

- **config_manager** (*ConfigManager*) – Provides the config dictionary, based on which *model_ids* are retrieved and models are selected and executed
- **model_selector** (*ModelSelector*) – Provides model comparison, search, and selection based on keys/values in the selection part of the config dict
- **model_executors** (*list*) – List of initialized *ModelExecutor* instances that handle model package download, init, and sample generation
- **initialize_all_models** (*bool*) – Flag indicating, if True, that one *ModelExecutor* for each *model_id* in the config dict should be initialized triggered by creation of *Generators* class instance. Note that, if False, the *Generators* class will only initialize a *ModelExecutor* on the fly when need be i.e. when the generate method for the respective model is called.

config_manager

Provides the config dictionary, based on which *model_ids* are retrieved and models are selected and executed

Type *ConfigManager*

model_selector

Provides model comparison, search, and selection based on keys/values in the selection part of the config dict

Type *ModelSelector*

model_executors

List of initialized *ModelExecutor* instances that handle model package download, init, and sample generation

Type list

add_all_model_executors()

Add *ModelExecutor* class instances for all models available in the config.

Return type None

add_metadata_from_file(*model_id: str, metadata_file_path: str*) → dict

Read and parse the metadata of a local model, identified by *model_id*, from a metadata file in json format.

Parameters

- **model_id** (*str*) – The generative model’s unique id
- **metadata_file_path** (*str*) – the path pointing to the metadata file

Returns Returns a dict containing the contents of parsed metadata json file.

Return type dict

add_metadata_from_input(*model_id: str, model_weights_name: str, model_weights_extension: str, generate_method_name: str, dependencies: list, fill_more_fields_interactively: bool = True, output_path: str = 'config'*) → dict

Create a metadata dict for a local model, identified by *model_id*, given the necessary minimum metadata contents.

Parameters

- **model_id** (*str*) – The generative model’s unique id
- **model_weights_name** (*str*) – the name of the checkpoint file containing the model’s weights
- **model_weights_extension** (*str*) – the extension (e.g. .pt) of the checkpoint file containing the model’s weights
- **generate_method_name** (*str*) – the name of the sample generation method inside the models `__init__.py` file
- **dependencies** (*list*) – the list of dependencies that need to be installed via pip to run the model
- **fill_more_fields_interactively** (*bool*) – flag indicating whether a user will be interactively asked via command line for further input to fill out missing metadata content
- **output_path** (*str*) – the path where the created metadata json file will be stored

Returns Returns a dict containing the contents of the metadata json file.

Return type dict

add_model_contributor(*model_id: str, init_py_path: Optional[str] = None*) → [*medigan.contribute_model.model_contributor.ModelContributor*](#)

Add a *ModelContributor* instance of this *model_id* to the *self.model_contributors* list.

Parameters

- **model_id** (*str*) – The generative model’s unique id
- **init_py_path** (*str*) – The path to the local model’s `__init__.py` file needed for importing and running this model.

Returns *ModelContributor* class instance corresponding to the *model_id*

Return type [*ModelContributor*](#)

add_model_executor(*model_id: str, install_dependencies: bool = False*)

Add one *ModelExecutor* class instance corresponding to the specified *model_id*.

Parameters

- **model_id** (*str*) – The generative model’s unique id

- **install_dependencies** (*bool*) – flag indicating whether a generative model’s dependencies are automatically installed. Else error is raised if missing dependencies are detected.

Return type None

add_model_to_config(*model_id: str, metadata: dict, is_local_model: Optional[bool] = None, overwrite_existing_metadata: bool = False, store_new_config: bool = True*) → bool

Adding or updating a model entry in the global metadata.

Parameters

- **model_id** (*str*) – The generative model’s unique id
- **metadata** (*dict*) – The model’s corresponding metadata
- **is_local_model** (*bool*) – flag indicating whether the tested model is a new local user model i.e not yet part of medigan’s official models
- **overwrite_existing_metadata** (*bool*) – in case of *is_local_model*, flag indicating whether existing metadata for this model in medigan’s *config/global.json* should be overwritten.
- **store_new_config** (*bool*) – flag indicating whether the current model metadata should be stored on disk i.e. in *config/*

Returns Flag indicating whether model metadata update was successfully concluded

Return type bool

contribute(*model_id: str, init_py_path: str, github_access_token: str, zenodo_access_token: str, metadata_file_path: Optional[str] = None, model_weights_name: Optional[str] = None, model_weights_extension: Optional[str] = None, generate_method_name: Optional[str] = None, dependencies: Optional[list] = None, fill_more_fields_interactively: bool = True, overwrite_existing_metadata: bool = False, output_path: str = 'config', creator_name: str = 'unknown name', creator_affiliation: str = 'unknown affiliation', model_description: str = '', install_dependencies: bool = False*)

Implements the full model contribution workflow including model metadata generation, model test, model Zenodo upload, and medigan github issue creation.

Parameters

- **model_id** (*str*) – The generative model’s unique id
- **init_py_path** (*str*) – The path to the local model’s `__init__.py` file needed for importing and running this model.
- **github_access_token** (*str*) – a personal access token linked to your github user account, used as means of authentication
- **zenodo_access_token** (*str*) – a personal access token in Zenodo linked to a user account for authentication
- **metadata_file_path** (*str*) – the path pointing to the metadata file
- **model_weights_name** (*str*) – the name of the checkpoint file containing the model’s weights
- **model_weights_extension** (*str*) – the extension (e.g. .pt) of the checkpoint file containing the model’s weights
- **generate_method_name** (*str*) – the name of the sample generation method inside the models `__init__.py` file

- **dependencies** (*list*) – the list of dependencies that need to be installed via pip to run the model
- **fill_more_fields_interactively** (*bool*) – flag indicating whether a user will be interactively asked via command line for further input to fill out missing metadata content
- **overwrite_existing_metadata** (*bool*) – flag indicating whether existing metadata for this model in medigan’s *config/global.json* should be overwritten.
- **output_path** (*str*) – the path where the created metadata json file will be stored
- **creator_name** (*str*) – the creator name that will appear on the corresponding github issue
- **creator_affiliation** (*str*) – the creator affiliation that will appear on the corresponding github issue
- **model_description** (*list*) – the model_description that will appear on the corresponding github issue
- **install_dependencies** (*bool*) – flag indicating whether a generative model’s dependencies are automatically installed. Else error is raised if missing dependencies are detected.

Returns Returns the url pointing to the corresponding issue on github

Return type *str*

find_matching_models_by_values(*values: list, target_values_operator: str = 'AND', are_keys_also_matched: bool = False, is_case_sensitive: bool = False*) → *list*

Search for values (and keys) in model configs and return a list of each matching *ModelMatchCandidate*.

This function calls an identically named function in a *ModelSelector* instance.

Parameters

- **values** (*list*) – list of values used to search and find models corresponding to these *values*
- **target_values_operator** (*str*) – the operator indicating the relationship between *values* in the evaluation of model search results. Should be either “AND”, “OR”, or “XOR”.
- **are_keys_also_matched** (*bool*) – flag indicating whether, apart from values, the keys in the model config should also be searchable
- **is_case_sensitive** (*bool*) – flag indicating whether the search for values (and) keys in the model config should be case-sensitive.

Returns a list of *ModelMatchCandidate* class instances each of which was successfully matched against the search values.

Return type *list*

find_model_and_generate(*values: list, target_values_operator: str = 'AND', are_keys_also_matched: bool = False, is_case_sensitive: bool = False, num_samples: int = 30, output_path: Optional[str] = None, is_gen_function_returned: bool = False, install_dependencies: bool = False, **kwargs*)

Search for values (and keys) in model configs to generate samples with the found model.

Note that the number of found models should be ==1. Else no samples will be generated and a error is logged to console.

Parameters

- **values** (*list*) – list of values used to search and find models corresponding to these *values*

- **target_values_operator** (*str*) – the operator indicating the relationship between *values* in the evaluation of model search results. Should be either “AND”, “OR”, or “XOR”.
- **are_keys_also_matched** (*bool*) – flag indicating whether, apart from values, the keys in the model config should also be searchable
- **is_case_sensitive** (*bool*) – flag indicating whether the search for values (and) keys in the model config should be case-sensitive.
- **num_samples** (*int*) – the number of samples that will be generated
- **output_path** (*str*) – the path as str to the output folder where the generated samples will be stored
- **is_gen_function_returned** (*bool*) – flag indicating whether, instead of generating samples, the sample generation function will be returned
- **install_dependencies** (*bool*) – flag indicating whether a generative model’s dependencies are automatically installed. Else error is raised if missing dependencies are detected.
- ****kwargs** – arbitrary number of keyword arguments passed to the model’s sample generation function

Returns However, if *is_gen_function_returned* is True, it returns the internal generate function of the model.

Return type None

find_model_executor_by_id(*model_id: str*) → *medigan.execute_model.model_executor.ModelExecutor*

Find and return the *ModelExecutor* instance of this *model_id* in the *self.model_executors* list.

Parameters *model_id* (*str*) – The generative model’s unique id

Returns *ModelExecutor* class instance corresponding to the *model_id*

Return type *ModelExecutor*

find_models_and_rank(*values: list, target_values_operator: str = 'AND', are_keys_also_matched: bool = False, is_case_sensitive: bool = False, metric: str = 'SSIM', order: str = 'asc'*) → list

Search for values (and keys) in model configs, rank results and return sorted list of model dicts.

This function calls an identically named function in a *ModelSelector* instance.

Parameters

- **values** (*list*) – list of values used to search and find models corresponding to these *values*
- **target_values_operator** (*str*) – the operator indicating the relationship between *values* in the evaluation of model search results. Should be either “AND”, “OR”, or “XOR”.
- **are_keys_also_matched** (*bool*) – flag indicating whether, apart from values, the keys in the model config should also be searchable
- **is_case_sensitive** (*bool*) – flag indicating whether the search for values (and) keys in the model config should be case-sensitive.
- **metric** (*str*) – The key in the selection dict that corresponds to the metric of interest
- **order** (*str*) – the sorting order of the ranked results. Should be either “asc” (ascending) or “desc” (descending)

Returns a list of the searched and matched model dictionaries containing metric and model_id, sorted by metric.

Return type list

find_models_rank_and_generate(*values: list, target_values_operator: str = 'AND', are_keys_also_matched: bool = False, is_case_sensitive: bool = False, metric: str = 'SSIM', order: str = 'asc', num_samples: int = 30, output_path: Optional[str] = None, is_gen_function_returned: bool = False, install_dependencies: bool = False, **kwargs*)

Search for values (and keys) in model configs, rank results to generate samples with highest ranked model.

Parameters

- **values** (*list*) – list of values used to search and find models corresponding to these *values*
- **target_values_operator** (*str*) – the operator indicating the relationship between *values* in the evaluation of model search results. Should be either “AND”, “OR”, or “XOR”.
- **are_keys_also_matched** (*bool*) – flag indicating whether, apart from values, the keys in the model config should also be searchable
- **is_case_sensitive** (*bool*) – flag indicating whether the search for values (and) keys in the model config should be case-sensitive.
- **metric** (*str*) – The key in the selection dict that corresponds to the metric of interest
- **order** (*str*) – the sorting order of the ranked results. Should be either “asc” (ascending) or “desc” (descending)
- **num_samples** (*int*) – the number of samples that will be generated
- **output_path** (*str*) – the path as str to the output folder where the generated samples will be stored
- **is_gen_function_returned** (*bool*) – flag indicating whether, instead of generating samples, the sample generation function will be returned
- **install_dependencies** (*bool*) – flag indicating whether a generative model’s dependencies are automatically installed. Else error is raised if missing dependencies are detected.
- ****kwargs** – arbitrary number of keyword arguments passed to the model’s sample generation function

Returns However, if *is_gen_function_returned* is True, it returns the internal generate function of the model.

Return type None

generate(*model_id: str, num_samples: int = 30, output_path: Optional[str] = None, save_images: bool = True, is_gen_function_returned: bool = False, install_dependencies: bool = False, **kwargs*)

Generate samples with the model corresponding to the *model_id* or return the model’s generate function.

Parameters

- **model_id** (*str*) – The generative model’s unique id
- **num_samples** (*int*) – the number of samples that will be generated
- **output_path** (*str*) – the path as str to the output folder where the generated samples will be stored

- **save_images** (*bool*) – flag indicating whether generated samples are returned (i.e. as list of numpy arrays) or rather stored in file system (i.e in *output_path*)
- **is_gen_function_returned** (*bool*) – flag indicating whether, instead of generating samples, the sample generation function will be returned
- **install_dependencies** (*bool*) – flag indicating whether a generative model’s dependencies are automatically installed. Else error is raised if missing dependencies are detected.
- ****kwargs** – arbitrary number of keyword arguments passed to the model’s sample generation function

Returns Returns images as list of numpy arrays if *save_images* is False. However, if *is_gen_function_returned* is True, it returns the internal generate function of the model.

Return type list

get_as_torch_dataloader(*dataset=None, model_id: Optional[str] = None, num_samples: int = 1000, install_dependencies: bool = False, transform=None, batch_size=None, shuffle=None, sampler=None, batch_sampler=None, num_workers=None, collate_fn=None, pin_memory=None, drop_last=None, timeout=None, worker_init_fn=None, prefetch_factor: Optional[int] = None, persistent_workers: Optional[bool] = None, pin_memory_device: Optional[str] = None, **kwargs*) → torch.utils.data.dataloader.DataLoader

Get torch Dataloader sampling synthetic data from medigan model.

Dataloader combines a dataset and a sampler, and provides an iterable over the given torch dataset. Dataloader is created for synthetic data for the specified medigan model. Pytorch native parameters are set to None per default. Only those params are passed to the Dataloader() initialization function that are not None.

Parameters

- **dataset** (*Dataset*) – dataset from which to load the data.
- **model_id** – str The generative model’s unique id
- **num_samples** – int the number of samples that will be generated
- **install_dependencies** – bool flag indicating whether a generative model’s dependencies are automatically installed. Else error is raised if missing dependencies are detected.
- ****kwargs** – arbitrary number of keyword arguments passed to the model’s sample generation function (e.g. the input path for image-to-image translation models in medigan).
- **transform** – the torch data transformation functions to be applied to the data in the dataset.
- **batch_size** (*int, optional*) – how many samples per batch to load (default: None).
- **shuffle** (*bool, optional*) – set to True to have the data reshuffled at every epoch (default: None).
- **sampler** (*Sampler or Iterable, optional*) – defines the strategy to draw samples from the dataset. Can be any *Iterable* with `__len__` implemented. If specified, *shuffle* must not be specified. (default: None)
- **batch_sampler** (*Sampler or Iterable, optional*) – like sampler, but returns a batch of indices at a time. Mutually exclusive with *batch_size*, *shuffle*, *sampler*, and *drop_last*. (default: None)
- **num_workers** (*int, optional*) – how many subprocesses to use for data loading. 0 means that the data will be loaded in the main process. (default: None)

- **collate_fn** (*callable, optional*) – merges a list of samples to form a mini-batch of Tensor(s). Used when using batched loading from a map-style dataset. (default: `None`)
- **pin_memory** (*bool, optional*) – If `True`, the data loader will copy Tensors into CUDA pinned memory before returning them. If your data elements are a custom type, or your `collate_fn` returns a batch that is a custom type, see the example below. (default: `None`)
- **drop_last** (*bool, optional*) – set to `True` to drop the last incomplete batch, if the dataset size is not divisible by the batch size. If `False` and the size of dataset is not divisible by the batch size, then the last batch will be smaller. (default: `None`)
- **timeout** (*numeric, optional*) – if positive, the timeout value for collecting a batch from workers. Should always be non-negative. (default: `None`)
- **worker_init_fn** (*callable, optional*) – If not `None`, this will be called on each worker subprocess with the worker id (an int in `[0, num_workers - 1]`) as input, after seeding and before data loading. (default: `None`)
- **prefetch_factor** (*int, optional, keyword-only arg*) – Number of batches loaded in advance by each worker. 2 means there will be a total of `2 * num_workers` batches prefetched across all workers. (default: `None`).
- **persistent_workers** (*bool, optional*) – If `True`, the data loader will not shutdown the worker processes after a dataset has been consumed once. This allows to maintain the workers *Dataset* instances alive. (default: `None`)
- **pin_memory_device** (*str, optional*) – the device to pin memory to if `pin_memory` is `True` (default: `None`).

Returns a `torch.utils.data.DataLoader` object with data generated by model corresponding to inputted *Dataset* or *model_id*.

Return type `DataLoader`

get_as_torch_dataset (*model_id: str, num_samples: int = 100, install_dependencies: bool = False, transform=None, **kwargs*) → `torch.utils.data.dataset.Dataset`

Get synthetic data in a torch Dataset for specified medigan model.

The dataset returns a dict with keys `sample` (== image), `labels` (== condition), and `mask` (== segmentation mask). While key `'sample'` is mandatory, the other key value pairs are only returned if applicable to generative model.

Parameters

- **model_id** – str The generative model’s unique id
- **num_samples** – int the number of samples that will be generated
- **install_dependencies** –
bool flag indicating whether a generative model’s dependencies are automatically installed. Else error is raised if missing dependencies are detected.
- **transform** the torch data transformation functions to be applied to the data in the dataset.
- ****kwargs** – arbitrary number of keyword arguments passed to the model’s sample generation function (e.g. the input path for image-to-image translation models in medigan).

Returns a `torch.utils.data.Dataset` object with data generated by model corresponding to *model_id*.

Return type `Dataset`

get_config_by_id(*model_id*: str, *config_key*: Optional[str] = None) → dict

Get and return the part of the config below a *config_key* for a specific *model_id*.

The *config_key* parameters can be separated by a ‘.’ (dot) to allow for retrieval of nested config keys, e.g, ‘execution.generator.name’

This function calls an identically named function in a *ConfigManager* instance.

Parameters

- **model_id** (str) – The generative model’s unique id
- **config_key** (str) – A key of interest present in the config dict

Returns a dictionary from the part of the config file corresponding to *model_id* and *config_key*.

Return type dict

get_generate_function(*model_id*: str, *num_samples*: int = 30, *output_path*: Optional[str] = None, *install_dependencies*: bool = False, ***kwargs*)

Return the model’s generate function.

Relies on the *self.generate* function.

Parameters

- **model_id** (str) – The generative model’s unique id
- **num_samples** (int) – the number of samples that will be generated
- **output_path** (str) – the path as str to the output folder where the generated samples will be stored
- **install_dependencies** (bool) – flag indicating whether a generative model’s dependencies are automatically installed. Else error is raised if missing dependencies are detected.
- ****kwargs** – arbitrary number of keyword arguments passed to the model’s sample generation function

Returns The internal reusable generate function of the generative model.

Return type function

get_model_contributor_by_id(*model_id*: str) →

medigan.contribute_model.model_contributor.ModelContributor

Find and return the *ModelContributor* instance of this *model_id* in the *self.model_contributors* list.

Parameters **model_id** (str) – The generative model’s unique id

Returns *ModelContributor* class instance corresponding to the *model_id*

Return type *ModelContributor*

get_model_executor(*model_id*: str, *install_dependencies*: bool = False) →

medigan.execute_model.model_executor.ModelExecutor

Add and return the *ModelExecutor* instance of this *model_id* from the *self.model_executors* list.

Relies on *self.add_model_executor* and *self.find_model_executor_by_id* functions.

Parameters

- **model_id** (str) – The generative model’s unique id

- **install_dependencies** (*bool*) – flag indicating whether a generative model’s dependencies are automatically installed. Else error is raised if missing dependencies are detected.

Returns *ModelExecutor* class instance corresponding to the *model_id*

Return type *ModelExecutor*

get_models_by_key_value_pair(*key1: str, value1: str, is_case_sensitive: bool = False*) → list

Get and return a list of *model_id* dicts that contain the specified key value pair in their selection config.

The key param can contain ‘.’ (dot) separations to allow for retrieval of nested config keys such as ‘execution.generator.name’

This function calls an identically named function in a *ModelSelector* instance.

Parameters

- **key1** (*str*) – The key in the selection dict
- **value1** (*str*) – The value in the selection dict that corresponds to *key1*
- **is_case_sensitive** (*bool*) – flag to evaluate keys and values with case sensitivity if set to True

Returns a list of the dictionaries each containing a models id and the found key-value pair in the models config

Return type list

get_selection_criteria_by_id(*model_id: str, is_model_id_removed: bool = True*) → dict

Get and return the selection config dict for a specific *model_id*.

This function calls an identically named function in a *ModelSelector* instance.

Parameters

- **model_id** (*str*) – The generative model’s unique id
- **is_model_id_removed** (*bool*) – flag to to remove the *model_ids* from first level of dictionary.

Returns a dictionary corresponding to the selection config of a model

Return type dict

get_selection_criteria_by_ids(*model_ids: Optional[list] = None, are_model_ids_removed: bool = True*) → list

Get and return a list of selection config dicts for each of the specified *model_ids*.

This function calls an identically named function in a *ModelSelector* instance.

Parameters

- **model_ids** (*list*) – A list of generative models’ unique ids or ids abbreviated as integers (e.g. 1, 2, .. 21)
- **are_model_ids_removed** (*bool*) – flag to remove the *model_ids* from first level of dictionary.

Returns a list of dictionaries each corresponding to the selection config of a model

Return type list

get_selection_keys(*model_id*: *Optional[str] = None*) → list

Get and return all first level keys from the selection config dict for a specific *model_id*.

This function calls an identically named function in a *ModelSelector* instance.

Parameters **model_id** (*str*) – The generative model’s unique id

Returns a list containing the keys as strings of the selection config of the *model_id*.

Return type list

get_selection_values_for_key(*key*: *str*, *model_id*: *Optional[str] = None*) → list

Get and return the value of a specified key of the selection dict in the config for a specific *model_id*.

The key param can contain ‘.’ (dot) separations to allow for retrieval of nested config keys such as ‘execution.generator.name’

This function calls an identically named function in a *ModelSelector* instance.

Parameters

- **key** (*str*) – The key in the selection dict
- **model_id** (*str*) – The generative model’s unique id

Returns a list of the values that correspond to the key in the selection config of the *model_id*.

Return type list

is_model_executor_already_added(*model_id*) → bool

Check whether the *ModelExecutor* instance of this *model_id* is already in *self.model_executors* list.

Parameters **model_id** (*str*) – The generative model’s unique id

Returns indicating whether this *ModelExecutor* had been already previously added to *self.model_executors*

Return type bool

is_model_metadata_valid(*model_id*: *str*, *metadata*: *dict*, *is_local_model*: *bool = True*) → bool

Checking if a model’s corresponding metadata is valid.

Specific fields in the model’s metadata are mandatory. It is asserted if these key value pairs are present.

Parameters

- **model_id** (*str*) – The generative model’s unique id
- **metadata** (*dict*) – The model’s corresponding metadata
- **is_local_model** (*bool*) – flag indicating whether the tested model is a new local user model i.e not yet part of medigan’s official models

Returns Flag indicating whether the specific model’s metadata format and fields are valid

Return type bool

list_models() → list

Return the list of *model_ids* as strings based on config.

Return type list

push_to_github(*model_id*: str, *github_access_token*: str, *package_link*: Optional[str] = None, *creator_name*: str = "", *creator_affiliation*: str = "", *model_description*: str = "")

Upload the model's metadata inside a github issue to the medigan github repository.

To add your model to medigan, your metadata will be reviewed on Github and added to medigan's official model metadata

The medigan repository issues page: <https://github.com/RichardObi/medigan/issues>

Get your Github access token here: <https://github.com/settings/tokens>

Parameters

- **model_id** (str) – The generative model's unique id
- **github_access_token** (str) – a personal access token linked to your github user account, used as means of authentication
- **package_link** – a package link
- **creator_name** (str) – the creator name that will appear on the corresponding github issue
- **creator_affiliation** (str) – the creator affiliation that will appear on the corresponding github issue
- **model_description** (list) – the model_description that will appear on the corresponding github issue

Returns Returns the url pointing to the corresponding issue on github

Return type str

push_to_zenodo(*model_id*: str, *zenodo_access_token*: str, *creator_name*: str = 'unknown name', *creator_affiliation*: str = 'unknown affiliation', *model_description*: str = "") → str

Upload the model files as zip archive to a public Zenodo repository where the model will be persistently stored.

Get your Zenodo access token here: <https://zenodo.org/account/settings/applications/tokens/new/> (Enable scopes *deposit:actions* and *deposit:write*)

Parameters

- **model_id** (str) – The generative model's unique id
- **zenodo_access_token** (str) – a personal access token in Zenodo linked to a user account for authentication
- **creator_name** (str) – the creator name that will appear on the corresponding Zenodo model upload homepage
- **creator_affiliation** (str) – the creator affiliation that will appear on the corresponding Zenodo model upload homepage
- **model_description** (list) – the model_description that will appear on the corresponding Zenodo model upload homepage

Returns Returns the url pointing to the corresponding Zenodo model upload homepage

Return type str

rank_models_by_performance(*model_ids*: Optional[list] = None, *metric*: str = 'SSIM', *order*: str = 'asc') → list

Rank model based on a provided metric and return sorted list of model dicts.

The metric param can contain ‘.’ (dot) separations to allow for retrieval of nested metric config keys such as ‘downstream_task.CLF.accuracy’

This function calls an identically named function in a *ModelSelector* instance.

Parameters

- **model_ids** (*list*) – only evaluate the *model_ids* in this list. If none, evaluate all available *model_ids*
- **metric** (*str*) – The key in the selection dict that corresponds to the metric of interest
- **order** (*str*) – the sorting order of the ranked results. Should be either “asc” (ascending) or “desc” (descending)

Returns a list of model dictionaries containing metric and *model_id*, sorted by metric.

Return type list

test_model(*model_id: str, is_local_model: bool = True, overwrite_existing_metadata: bool = False, store_new_config: bool = True, num_samples: int = 3, install_dependencies: bool = False*)

Test if a model generates and returns a specific number of samples in the correct format

Parameters

- **model_id** (*str*) – The generative model’s unique id
- **is_local_model** (*bool*) – flag indicating whether the tested model is a new local user model i.e not yet part of medigan’s official models
- **overwrite_existing_metadata** (*bool*) – in case of *is_local_model*, flag indicating whether existing metadata for this model in medigan’s *config/global.json* should be overwritten.
- **store_new_config** (*bool*) – flag indicating whether the current model metadata should be stored on disk i.e. in *config/*
- **num_samples** (*int*) – the number of samples that will be generated
- **install_dependencies** (*bool*) – flag indicating whether a generative model’s dependencies are automatically installed. Else error is raised if missing dependencies are detected.

visualize(*model_id: str, slider_grouper: int = 10, auto_close: bool = False, install_dependencies: bool = False*) → None

Initialize and run *ModelVisualizer* of this *model_id* if it is available. It allows to visualize a sample from the model’s output. UI window will pop up allowing the user to control the generation parameters (conditional and unconditional ones).

Parameters

- **model_id** (*str*) – The generative model’s unique id to visualize.
- **slider_grouper** (*int*) – Number of input parameters to group together within one slider.
- **auto_close** (*bool*) – Flag for closing the user interface automatically after time. Used while testing.
- **install_dependencies** (*bool*) – flag indicating whether a generative model’s dependencies are automatically installed. Else error is raised if missing dependencies are detected.

medigan.model_visualizer module

ModelVisualizer class providing visualizing corresponding model input and model output changes.

class medigan.model_visualizer.**ModelVisualizer**(*model_executor*, *config*: None)

Bases: object

ModelVisualizer class: Visualises synthetic data through a user interface. Depending on a model, it is possible to control the input latent vector values and conditional input.

Parameters

- **model_executor** (*ModelExecutor*) – The generative model’s executor object
- **config** (*dict*) – The config dict containing the model metadata

model_executor

The generative model’s executor object

Type *ModelExecutor*

input_latent_vector_size

Size of the latent vector used as an input for generation

Type int

conditional

Flag for models with conditional input

Type bool

condition

Value of the conditinal input to the model

Type Union[int, float]

max_input_value

Absolute value used for setting latent values input range

Type float

visualize(*slider_grouper*: int = 10, *auto_close*=False)

Visualize the model’s output. This method is called by the user. It opens up a user interface with available controls.

Parameters

- **slider_grouper** (*int*) – Number of input parameters to group together within one slider.
- **auto_close** (*bool*) – Flag for closing the user interface automatically after time. Used while testing.

Return type None

medigan.utils module

Utils class providing generalized reusable functions for I/O, parsing, sorting, type conversions, etc.

class medigan.utils.Utils

Bases: object

Utils class containing reusable static methods.

static **call_without_removable_params**(*my_callable*, *removable_param_values*: list = [None],
***params*)

call a callable without passing parameters that contain any of the *removable_param_values* as value.

static **copy**(*source_path*: pathlib.Path, *target_path*: str = '/')

copy a folder or file from *source_path* to *target_path*

static **deep_get**(*base_dict*: dict, *key*: str)

Split the key by “.” to get value in nested dictionary.

static **dict_to_lowercase**(*target_dict*: dict, *string_conversion*: bool = True) → dict

transform values and keys in dict to lowercase, optionally with string conversion of the values.

Warning: Does not convert nested dicts in the *target_dict*, but rather removes them from return object.

static **download_file**(*download_link*: str, *path_as_string*: str, *file_extension*: str = '.json')

download a file using the *requests* lib and store in *path_as_string*

static **has_more_than_n_diff_pixel_values**(*img*: numpy.ndarray, *n*: int = 4) → bool

This function checks whether an image contains more than *n* different pixel values.

This helps to differentiate between segmentation masks and actual images.

static **is_file_in**(*folder_path*: str, *filename*: str) → bool

Checks if a file is inside a folder

static **is_file_located_or_downloaded**(*path_as_string*: str, *download_if_not_found*: bool = True,
download_link: Optional[str] = None,
is_new_download_forced: bool = False,
allow_local_path_as_url: bool = True) → bool

check if is file in *path_as_string* and optionally download the file (again).

static **is_url_valid**(*the_url*: str) → bool

Checks if a url is valid using *urllib.parse.urlparse*

static **list_to_lowercase**(*target_list*: list) → list

string conversion and lower-casing of values in list.

trade-off: String conversion for increased robustness > type failure detection

static **makedirs**(*path_as_string*: str) → bool

create folder in *path_as_string* if not already created.

static **order_dict_by_value**(*dict_list*, *key*: str, *order*: str = 'asc', *sort_algorithm*='bubbleSort') → list

Sorting a list of dicts by the values of a specific key in the dict using a sorting algorithm.

- This function is deprecated. You may use Python List *sort()* with *key*=lambda function instead.

static **read_in_json**(*path_as_string*) → dict

read a .json file and return as dict

```
static split_images_and_masks_no_ordering(data: list, num_samples: int, max_nested_arrays: int = 2) → [<class 'numpy.ndarray'>, <class 'numpy.ndarray'>]
```

Extracts and separates the masks from the images if a model returns both in the same np.ndarray.

This extendable function assumes that, in data, a mask follows the image that it corresponds to or vice versa.

- This function is deprecated. Please use *split_images_masks_and_labels* instead.

```
static split_images_masks_and_labels(data: list, num_samples: int, max_nested_arrays: int = 2) → [<class 'list'>, <class 'list'>, <class 'list'>, <class 'list'>]
```

Separates the data (sample, mask, other_imaging_data, label) returned by a generative model

This functions expects a list of tuples as input *data* and assumes that each tuple contains sample, mask, other_imaging_data, label at index positions [0], [1], [2], and [3] respectively.

samples, masks, and imaging data are expected to be of type np.ndarray and labels of type “str”.

For example, this extendable function assumes that, in data, a mask follows the image that it corresponds to or vice versa.

```
static store_dict_as(dictionary, extension: str = '.json', output_path: str = 'config', filename: str = 'metadata.json')
```

store a Python dictionary in file system as variable filetype.

```
static unzip_and_return_unzipped_path(package_path: str)
```

if not already dir, unzip an archive with *Utils.unzip_archive*. Return path to unzipped dir/file

```
static unzip_archive(source_path: pathlib.Path, target_path: str = './')
```

unzip a .zip archive in the *target_path*

Module contents

medigan is a modular Python library for automating synthetic dataset generation.

1.6 Tests

Table of Contents

- *Tests*
 - *Setup medigan for running tests*
 - *Test 1: test_medigan_imports*
 - *Test 2: test_init_generators*
 - *Test 3: test_generate_methods*
 - *Test 4: test_generate_methods_with_additional_args*
 - *Test 5: test_get_generate_method*
 - *Test 6: test_search_for_models_method*
 - *Test 7: test_find_model_and_generate_method*

- Test 8: `test_rank_models_by_performance`
- Test 9: `test_find_and_rank_models_by_performance`
- Test 10: `test_find_and_rank_models_then_generate_method`
- Test 11: `test_get_models_by_key_value_pair`

Automated continuous integration (CI) tests ([GitHub actions](#)) are triggered by commits to the medigan repository. These CI tests can be found [here](#).

Apart from that, to facilitate testing if *medigan* is setup correctly and whether all of the features in *medigan* work as desired, the following set of automated test cases is provided. Below, each test function is described and a command is provided to run each test.

1.6.1 Setup medigan for running tests

Open your command line, and clone *medigan* from Github with:

```
git clone https://github.com/RichardObi/medigan.git
cd medigan
```

To install dependencies and to setup and activate a virtual environment, run:

```
pip install pipenv
pipenv install
pipenv shell
```

1.6.2 Test 1: test_medigan_imports

This test checks if *medigan* can be imported correctly.

```
python -m tests.tests TestMediganMethods.test_medigan_imports
```

1.6.3 Test 2: test_init_generators

This test checks if the central *generators* class can be initialised correctly.

```
python -m tests.tests TestMediganMethods.test_init_generators
```

1.6.4 Test 3: test_generate_methods

This test examines whether samples can be created with any of three example generative models (1, 2, 3) in *medigan*.

```
python -m tests.tests TestMediganMethods.test_generate_methods
```

1.6.5 Test 4: test_generate_methods_with_additional_args

Additional key-value pair arguments (kwargs) can be provided to the *generate()* method of a generative model. This test checks if these additional arguments are passed correctly to the generate method and whether the generate method's returned result corresponds to the passed arguments.

```
python -m tests.tests TestMediganMethods.test_generate_methods_with_additional_args
```

1.6.6 Test 5: test_get_generate_method

The *generate()* method of any of the generative models in *medigan* can be returned. This makes it easier to integrate the *generate()* function dynamically into users' data processing and training pipelines i.e. avoiding it to reload the model weights each time it is called. This test tests if the *generate()* method is successfully returned and usable thereafter.

```
python -m tests.tests TestMediganMethods.test_get_generate_method
```

1.6.7 Test 6: test_search_for_models_method

The tested function searches for a model by matching provided key words with the information in the model's *config*. This test checks whether the expected models are found accordingly.

```
python -m tests.tests TestMediganMethods.test_search_for_models_method
```

1.6.8 Test 7: test_find_model_and_generate_method

After searching and finding one specific model, the tested function generates samples with that model. This test checks whether the expected model is found and whether samples are generated accordingly.

```
python -m tests.tests TestMediganMethods.test_find_model_and_generate_method
```

1.6.9 Test 8: test_rank_models_by_performance

Provided a list of model ids, the tested function ranks these models by a performance metric. The performance metrics are stored in the models' *config*. This test checks whether the ranking worked and whether the expected model is ranked the highest.

```
python -m tests.tests TestMediganMethods.test_rank_models_by_performance
```

1.6.10 Test 9: test_find_and_rank_models_by_performance

After searching and finding various models, the tested function ranks these models by a performance metric. This test checks whether the expected model is found, and whether it is the highest ranked one and whether it generated samples accordingly.

```
python -m tests.tests TestMediganMethods.test_find_and_rank_models_by_performance
```

1.6.11 Test 10: test_find_and_rank_models_then_generate_method

After searching and finding various models, the tested function ranks these models by a performance metric and generates samples with the highest ranked model. This test checks whether the expected model is found, whether it is the highest ranked one and whether it generated samples accordingly.

```
python -m tests.tests TestMediganMethods.test_find_and_rank_models_then_generate_method
```

1.6.12 Test 11: test_get_models_by_key_value_pair

After receiving a key value pair, the tested function returns all models that have that key-value pair in their model `config`. This test checks whether the expected models are found and returned correctly.

```
python -m tests.tests TestMediganMethods.test_get_models_by_key_value_pair
```

1.7 Model Contributions

We are happy that you are considering contributing your model to *medigan*. This will make your model accessible to the community and our users can easily integrate your synthetic data into their training pipelines and experiments.

1.7.1 Guide: Automated Model Contribution

Create an `__init__.py` file in your model's root folder.

Next, run the following code to contribute your model to *medigan*.

- Your model will be stored on [Zenodo](#).
- Also, a Github [issue](#) will be created to add your model's metadata to medigan's `global.json`.
- To do so, please provide a github access token ([get one here](#)) and a zenodo access token ([get one here](#)), as shown below. After creation, the zenodo access token may take a few minutes before being recognized in zenodo API calls.

```
from medigan import Generators
gen = Generators()

# Contribute your model
gen.contribute(
    model_id = "00100_YOUR_MODEL", # assign an ID
    init_py_path = "path/ending/with/__init__.py",
    model_weights_name = "100000",
    model_weights_extension = ".pt",
    generate_method_name = "generate", # in __init__.py
    dependencies = ["numpy", "torch"],
    creator_name = "YOUR_NAME",
    creator_affiliation = "YOUR_AFFILIATION",
    zenodo_access_token = 'ZENODO_ACCESS_TOKEN',
    github_access_token = 'GITHUB_ACCESS_TOKEN',
)
```

1.7.2 Guide: Manual Model Contribution

In the following, you find a step-by-step guide on how to contribute your generative model to *medigan*. In case you encounter problems during this process feel free to reach out by creating an issue [here](#) and we will try to help. Checkout the figure below that shows the main components of the model contribution workflow depicted in yellow (d).

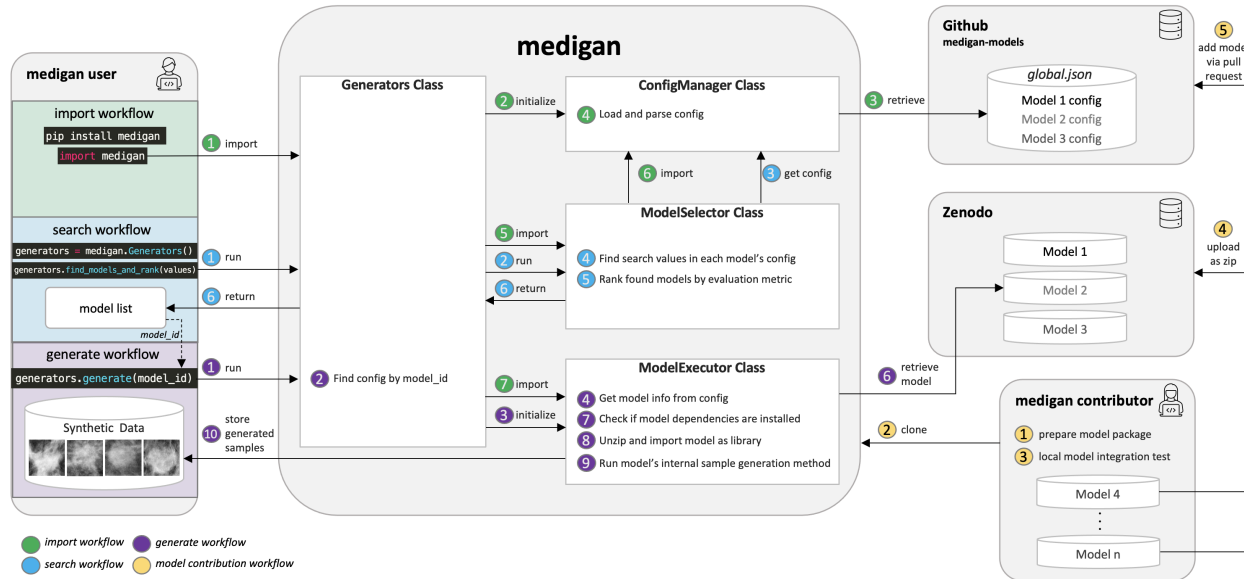


Fig. 3: Architectural overview including main workflows consisting of (a) library import and initialisation, (b) generative model search and ranking, (c) sample generation, and (d) generative model contribution.

If you are here, you have recently developed a generative model such as a GAN, VAE, Diffusion Model, etc and you would like to boost your model's impact, reusability, dissemination by uploading it to *medigan*. We are delighted and will assist you in adding your model.

1. Firstly, let's create the needed files:

To add your model you will need the following files.

1. A checkpoint file that contains your trained model weights (e.g., the `state_dict` in pytorch)
2. An `__init__.py` file that contains functions that
 - load the weights file (let's call that one `weights.pt`)
 - initialize your model with these weights
 - generate samples with the initialized model.
3. Now that you have the `weights.pt` and the `__init__.py`, let's check if we can make them work together.
 1. Run your `__init__.py`'s generate function using e.g. `python -m __init__.py generate`
 2. Check whether your model did load the weights effectively and whether synthetic samples were generated as expected.
4. Apply some necessary adjustments to your model package, particularly to your `__init__.py`:
 1. We have some templates that you can use to guide the adjustments described below

- If you are using a model that generates samples **without** image input (e.g., noise-to-image): Download Model 00002 from [here](#). Unzip it and open the `__init__.py` that contains an example `generate()` method.
 - If you are using a model that generates samples **with** image input (e.g., image-to-image): Download Model 00003 from [here](#). Unzip it and open the `__init__.py` that contains an example `generate()` method.
2. Please note that user's of *medigan* models may add your model to their preprocessing or AI training pipelines. Please make sure that your model, hence, runs efficiently. For instance, your model should load the weights only once even though the `generate()` function is called multiple times.
 3. Please make sure your model is able to run both on gpu and on cpu and your code automatically detects on which one to run.
 4. Please replace all `print()` functions from your code with `logging.debug()` (for this you need to `import logging`).
 5. Please add appropriate error handling using `try`, `except` blocks on each possible source of error in your code. `raise` the error in your `generate()` function to allow *medigan* to handle it and pass it to the user.
 6. If your generative model needs some input images, provide a few example images in a folder called `/images`. Users may test your model with these example images before feeding their own input images to your model.
 7. There are a few parameters of the `generate()` that are mandatory in *medigan* and others that you can set optionally.

- **Mandatory:**

- `model_file`: string, the path where your `generate()` method will find its weight file
- `output_path`: string, the path where our `generate()` method should store the generated images
- `save_images`: boolean, whether your `generate()` method should store generated samples in `output_path` or return them as numpy arrays.
- `num_samples`: int, the number of samples that should be generated.

- **Optional:**

- `input_path`: string, the path where our `generate()` method finds images that should be used as input into the generative model (i.e. in image-to-image translation).
- `image_size`: array, that contains image height, width, and, optionally, also depth.
- `translate_all_images`: boolean, in image-to-image translation, if `True`, this overwrites the `num_samples` and instead translates all images found in `input_path`.
- `gpu_id`: int, if a user has various GPUs available, the user can specify which one of them to use to run your generative model.

2. **Secondly, test your model locally:** Okay, now that we know which files we need, let's test them using a local version of *medigan*.

1. Let's start by cloning *medigan* e.g. using the command line: `git clone https://github.com/RichardObi/medigan.git`
2. Next, `cd` into *medigan*, install the dependencies of *medigan*, and create a virtual environment.

You can do so running these commands:

- `cd medigan`
- `pip install pipenv`
- `pipenv install`
- `pipenv shell`

3. Now that you have your environment up and running, please run the following command to download the config file.

- `python -m tests.tests TestMediganMethods.test_init_generators`

4. In the folder `/config`, you should now see a file called `global.json`. In this file each model's metadata is stored.

- Please add the metadata for your model at the bottom of the `global.json` file.
- To add the metadata, you can use the metadata of model `00001` in `global.json` as example.
- **Copy the metadata of model 00001 and add it to the bottom of `global.json`. Then adjust each entry in the**

- The `model_id` should follow the convention `NNNNN_TTTTTT_MMMM_AAAAA_GGGG` (N = Number of model, T = Type of model, M = Modality, A = Anatomic/Ailment Information, G = Generated Sample Type information i.e. full for full image or roi for region of interest)
- The field `package_link` (under `execution`) should point to a local zip file `NAME_OF_YOUR_MODEL_PACKAGE.zip` of your model package.
- json entries below `execution` are important and needed to run the model in *medigan*, e.g. the name and parameters of a `generate()` function in the `__init__.py`
- json entries below `selection` are important to enable users to search and rank the model compared to other models in *medigan*, e.g. the performance indicators such as SSIM, MSE, PSNR, etc.
- json entries below `description` are to allow tracing back the origin and metadata of the model and allow users to get further information about the model, e.g. license, related publications, etc.

5. You are almost done! It's Testing Time!

- Run a local test using the following code:

```
from medigan import Generators
gen = Generators()
gen.generate(model_id="YOUR_MODEL_ID")

# Test a few variations.
test_dict = {"translate_all_images": True, "SOME_OTHER_OPTIONAL_
↳PARAMS": True}
gen.generate(model_id="YOUR_MODEL_ID", num_samples=100, output_path=
↳"here", save_images=True, **test_dict)
```

- If your code runs well with different settings/params, congratulations, you have made it! You integrated your model as a package into *medigan* and are now ready for the final steps.

3. Thirdly, upload your model:

1. Package and upload your model to Zenodo - home to your model's code and documentation.

1. First, check if your model package folder contains an `__init__.py`, a `weights` file, a `license` file, and optionally other files.
2. The next step is to zip this folder. To do so (e.g., on MACOS) you may `cd` into the folder and use the following commands (while removing hidden OS system files):

```
find . -name ".DS_Store" -delete
zip -r NAME_OF_YOUR_MODEL_PACKAGE.zip . -x ".*" -x "__MACOSX"
```

Now that you have your model package zipped and ready, note that *medigan* model's are commonly stored in Z

- they get a DOI
 - the content of their package is non editable i.e. no file modifications/updates without new DOI.
 - This helps to avoid security issues as package content remains static after the model is tested, verified, and added to *medigan*.
 - Zenodo has a close to unlimited storage capacity for research data/software.
 - Also, the authorship/ownership of the model are clear
 - There is transparent licensing.
 - Each model is versioned in Zenodo with different DOIs.
 - A model documentation and contact information can be added.
1. Checkout this example of our model 00001 on Zenodo. You can use the Zenodo documentation of this model as template for your own model upload.
 2. Now, let's go to the [Zenodo](#) website.
 3. Click on New Upload (if you don't have an account, you can quickly create one e.g., using your [ORCID](#))
 4. Fill in the metadata fields for your model and upload the model package zip file (i.e. drag and drop).
 5. Click on Save and Submit. Congratulations your model is now on Zenodo! Good job!

4. Finally, add your model to `medigan's` model metadata:

Last step!!! Your model is on Zenodo and tested locally. Now we can officially add it to *medigan*. Remember the `global.json` that you created locally to test your model? It is time for glory for this file.

1. Now, clone the *medigan-models* repository (the home of *medigan's global.json*) e.g. by using `git clone https://github.com/RichardObi/medigan-models.git`
2. Create and checkout a new local branch `git checkout -b mynewbranch`
3. Open the `global.json` in your cloned local *medigan-models*
4. Edit the `global.json` file and add your model's entry at the bottom, and save.
5. Note that this is the time to replace the value of `package_link` from your local model file path to your new Zenodo model URL. To get this URL, go to the Zenodo page of your model, and scroll down to Files, where you see a download button. Copy the url link that this button points to, which is your `package_link`.
6. Commit the new file (`git add .`, `git commit -m "added model YOUR_MODEL_ID."`) and push your branch (`git push`).

7. Lastly, go to the repository [medigan-models](#) and create a pull request that merges your recently pushed branch into `main`.
8. That's it!!! Your pull request will be evaluated asap. Once approved your model is officially part of *medigan*!

If you have suggestions on improvements for our model contribution process, please take a minute and let us know [here](#).

1.7.3 Conventions that your model should follow

- Your model should have a `generate` method with the params `model_file:str`, `num_samples:int`, `save_images:bool`, and `output_path:str` (see ``template (templates/examples/__init__.py>`_`)`)
- Also, the model should do simple error handling, run flexibly on either `gpu` or `cpu`, use `logging` instead of `prints`, and create some sort of synthetic data.

We hope to welcome you model soon to *medigan*! If you need support, please let us now [here](#).

INDICES

- `genindex`
- `modindex`
- `search`

PYTHON MODULE INDEX

m

- medigan, 98
- medigan.config_manager, 59
- medigan.constants, 80
- medigan.contribute_model, 69
 - medigan.contribute_model.base_model_uploader, 61
 - medigan.contribute_model.github_model_uploader, 62
 - medigan.contribute_model.model_contributor, 54
 - medigan.contribute_model.zenodo_model_uploader, 66
- medigan.exceptions, 83
- medigan.execute_model, 72
 - medigan.execute_model.install_model_dependencies, 69
 - medigan.execute_model.model_executor, 45
 - medigan.execute_model.synthetic_dataset, 71
- medigan.generators, 31
- medigan.model_visualizer, 49
- medigan.select_model, 78
 - medigan.select_model.matched_entry, 72
 - medigan.select_model.model_match_candidate, 73
 - medigan.select_model.model_selector, 50
- medigan.utils, 97

A

`access_token` (*medigan.contribute_model.github_model_uploader.GithubModelUploader* attribute), 62

`access_token` (*medigan.contribute_model.zenodo_model_uploader.ZenodoModelUploader* attribute), 67

`add_all_model_executors()` (*medigan.generators.Generators* method), 33, 83

`add_matched_entry()` (*medigan.select_model.model_match_candidate.ModelMatchCandidate* method), 74

`add_metadata_from_file()` (*medigan.contribute_model.model_contributor.ModelContributor* attribute), 49, 96 method), 56, 64

`add_metadata_from_file()` (*medigan.generators.Generators* method), 33, 83

`add_metadata_from_input()` (*medigan.contribute_model.model_contributor.ModelContributor* method), 56, 64

`add_metadata_from_input()` (*medigan.generators.Generators* method), 34, 84

`add_model_contributor()` (*medigan.generators.Generators* method), 34, 84

`add_model_executor()` (*medigan.generators.Generators* method), 34, 84

`add_model_to_config()` (*medigan.config_manager.ConfigManager* method), 60, 78

`add_model_to_config()` (*medigan.generators.Generators* method), 34, 85

`add_package_link_to_metadata()` (*medigan.contribute_model.github_model_uploader.GithubModelUploader* method), 62

`are_keys_also_matched` (*medigan.select_model.model_match_candidate.ModelMatchCandidate* attribute), 74

B

`BaseModelUploader` class in *medigan.contribute_model.base_model_uploader*,

C

`call_without_removable_params()` (*medigan.utils.Utils* static method), 97

`check_if_is_match()` (*medigan.select_model.model_match_candidate.ModelMatchCandidate* method), 74

`condition` (*medigan.model_visualizer.ModelVisualizer* attribute), 49, 96

`conditional` (*medigan.model_visualizer.ModelVisualizer* attribute), 49, 96

`config_dict` (*medigan.config_manager.ConfigManager* attribute), 59, 78

`CONFIG_FILE_FOLDER` (in module *medigan.constants*), 80

`CONFIG_FILE_KEY_DEPENDENCIES` (in module *medigan.constants*), 80

`CONFIG_FILE_KEY_DESCRIPTION` (in module *medigan.constants*), 80

`CONFIG_FILE_KEY_EXECUTION` (in module *medigan.constants*), 80

`CONFIG_FILE_KEY_GENERATE` (in module *medigan.constants*), 80

`CONFIG_FILE_KEY_GENERATE_ARGS` (in module *medigan.constants*), 80

`CONFIG_FILE_KEY_GENERATE_ARGS_BASE` (in module *medigan.constants*), 80

`CONFIG_FILE_KEY_GENERATE_ARGS_CUSTOM` (in module *medigan.constants*), 80

`CONFIG_FILE_KEY_GENERATE_ARGS_INPUT_LATENT_VECTOR_SIZE` (in module *medigan.constants*), 80

`CONFIG_FILE_KEY_GENERATE_ARGS_MODEL_FILE` (in module *medigan.constants*), 80

`CONFIG_FILE_KEY_GENERATE_ARGS_NUM_SAMPLES` (in module *medigan.constants*), 80

`CONFIG_FILE_KEY_GENERATE_ARGS_OUTPUT_PATH` (in module *medigan.constants*), 80

`CONFIG_FILE_KEY_GENERATE_ARGS_SAVE_IMAGES` (in module *medigan.constants*), 80

- `module medigan.constants`), 81
- `CONFIG_FILE_KEY_GENERATE_NAME` (in `module medigan.constants`), 81
- `CONFIG_FILE_KEY_GENERATOR` (in `module medigan.constants`), 81
- `CONFIG_FILE_KEY_GENERATOR_NAME` (in `module medigan.constants`), 81
- `CONFIG_FILE_KEY_IMAGE_SIZE` (in `module medigan.constants`), 81
- `CONFIG_FILE_KEY_MODEL_EXTENSION` (in `module medigan.constants`), 81
- `CONFIG_FILE_KEY_MODEL_NAME` (in `module medigan.constants`), 81
- `CONFIG_FILE_KEY_PACKAGE_LINK` (in `module medigan.constants`), 81
- `CONFIG_FILE_KEY_PACKAGE_NAME` (in `module medigan.constants`), 81
- `CONFIG_FILE_KEY_PERFORMANCE` (in `module medigan.constants`), 81
- `CONFIG_FILE_KEY_SELECTION` (in `module medigan.constants`), 81
- `CONFIG_FILE_KEY_TAGS` (in `module medigan.constants`), 81
- `CONFIG_FILE_NAME_AND_EXTENSION` (in `module medigan.constants`), 81
- `CONFIG_FILE_URL` (in `module medigan.constants`), 81
- `config_manager` (`medigan.generators.Generators` attribute), 32, 83
- `config_manager` (`medigan.select_model.model_selector.ModelSelector` attribute), 51, 74
- `CONFIG_TEMPLATE_FILE_NAME_AND_EXTENSION` (in `module medigan.constants`), 81
- `CONFIG_TEMPLATE_FILE_URL` (in `module medigan.constants`), 81
- `ConfigManager` (class in `medigan.config_manager`), 59, 78
- `contribute()` (`medigan.generators.Generators` method), 35, 85
- `copy()` (`medigan.utils.Utils` static method), 97
- `create_upload_description()` (`medigan.contribute_model.zenodo_model_uploader.ZenodoModelUploader` method), 67
- `create_upload_json_data()` (`medigan.contribute_model.zenodo_model_uploader.ZenodoModelUploader` method), 67
- D**
- `deep_get()` (`medigan.utils.Utils` static method), 97
- `DEFAULT_OUTPUT_FOLDER` (in `module medigan.constants`), 81
- `dependencies` (`medigan.execute_model.model_executor.ModelExecutor` attribute), 46, 70
- `deserialized_model_as_lib` (`medigan.execute_model.model_executor.ModelExecutor` attribute), 47, 71
- `dict_to_lowercase()` (`medigan.utils.Utils` static method), 97
- `download_file()` (`medigan.utils.Utils` static method), 97
- `download_package` (`medigan.execute_model.model_executor.ModelExecutor` attribute), 46, 70
- E**
- `empty_upload()` (`medigan.contribute_model.zenodo_model_uploader.ZenodoModelUploader` method), 67
- `execution_config` (`medigan.execute_model.model_executor.ModelExecutor` attribute), 46, 70
- F**
- `find_matching_models_by_values()` (`medigan.generators.Generators` method), 36, 86
- `find_matching_models_by_values()` (`medigan.select_model.model_selector.ModelSelector` method), 51, 75
- `find_model_and_generate()` (`medigan.generators.Generators` method), 36, 86
- `find_model_executor_by_id()` (`medigan.generators.Generators` method), 37, 87
- `find_models_and_rank()` (`medigan.generators.Generators` method), 37, 87
- `find_models_and_rank()` (`medigan.select_model.model_selector.ModelSelector` method), 52, 75
- `find_models_rank_and_generate()` (`medigan.generators.Generators` method), 37, 88
- G**
- `generate()` (`medigan.execute_model.model_executor.ModelExecutor` method), 48, 71
- `generate()` (`medigan.generators.Generators` method), 38, 88
- `generate_method_args` (`medigan.execute_model.model_executor.ModelExecutor` attribute), 47, 70
- `generate_method_name` (`medigan.execute_model.model_executor.ModelExecutor` attribute), 47, 70
- `Generators` (class in `medigan.generators`), 31, 83

[get_all_matching_elements\(\)](#) (*medigan.select_model.model_match_candidate.ModelMatchCandidate* attribute), 55, 64
[get_as_torch_data_loader\(\)](#) (*medigan.generators.Generators* method), 39, 89
[get_as_torch_dataset\(\)](#) (*medigan.generators.Generators* method), 40, 90
[get_config_by_id\(\)](#) (*medigan.config_manager.ConfigManager* method), 60, 79
[get_config_by_id\(\)](#) (*medigan.generators.Generators* method), 40, 90
[get_generate_function\(\)](#) (*medigan.generators.Generators* method), 41, 91
[get_model_contributor_by_id\(\)](#) (*medigan.generators.Generators* method), 41, 91
[get_model_executor\(\)](#) (*medigan.generators.Generators* method), 41, 91
[get_models_by_key_value_pair\(\)](#) (*medigan.generators.Generators* method), 42, 92
[get_models_by_key_value_pair\(\)](#) (*medigan.select_model.model_selector.ModelSelector* method), 52, 75
[get_selection_criteria_by_id\(\)](#) (*medigan.generators.Generators* method), 42, 92
[get_selection_criteria_by_id\(\)](#) (*medigan.select_model.model_selector.ModelSelector* method), 52, 76
[get_selection_criteria_by_ids\(\)](#) (*medigan.generators.Generators* method), 42, 92
[get_selection_criteria_by_ids\(\)](#) (*medigan.select_model.model_selector.ModelSelector* method), 53, 76
[get_selection_keys\(\)](#) (*medigan.generators.Generators* method), 42, 92
[get_selection_keys\(\)](#) (*medigan.select_model.model_selector.ModelSelector* method), 53, 76
[get_selection_values_for_key\(\)](#) (*medigan.generators.Generators* method), 43, 93
[get_selection_values_for_key\(\)](#) (*medigan.select_model.model_selector.ModelSelector* method), 53, 76
[github_model_uploader](#) (*medigan.contribute_model.model_contributor.ModelContributor* attribute), 55, 64
[GITHUB_REPO](#) (in module *medigan.constants*), 82
[GITHUB_TITLE](#) (in module *medigan.constants*), 82
[GithubModelUploader](#) (class in *medigan.contribute_model.github_model_uploader*), 62

H

[has_more_than_n_diff_pixel_values\(\)](#) (*medigan.utils.Utils* static method), 97

I

[image_size](#) (*medigan.execute_model.model_executor.ModelExecutor* attribute), 46, 70
[INIT_PY_FILE](#) (in module *medigan.constants*), 82
[init_py_path](#) (*medigan.contribute_model.model_contributor.ModelContributor* attribute), 55, 63
[input_latent_vector_size](#) (*medigan.model_visualizer.ModelVisualizer* attribute), 49, 96
[install_model\(\)](#) (in module *medigan.execute_model.install_model_dependencies*), 69
[is_case_sensitive](#) (*medigan.select_model.model_match_candidate.ModelMatchCandidate* attribute), 73
[is_config_loaded](#) (*medigan.config_manager.ConfigManager* attribute), 59, 78
[is_file_in\(\)](#) (*medigan.utils.Utils* static method), 97
[is_file_located_or_downloaded\(\)](#) (*medigan.utils.Utils* static method), 97
[is_match](#) (*medigan.select_model.model_match_candidate.ModelMatchCandidate* attribute), 74
[is_model_already_unpacked\(\)](#) (*medigan.execute_model.model_executor.ModelExecutor* method), 48, 71
[is_model_executor_already_added\(\)](#) (*medigan.generators.Generators* method), 43, 93
[is_model_in_config\(\)](#) (*medigan.config_manager.ConfigManager* method), 60, 79
[is_model_metadata_valid\(\)](#) (*medigan.config_manager.ConfigManager* method), 61, 79
[is_model_metadata_valid\(\)](#) (*medigan.generators.Generators* method), 43, 93
[is_new_download_forced](#) (*medigan.config_manager.ConfigManager* attribute), 59, 78
[is_url_valid\(\)](#) (*medigan.utils.Utils* static method), 97

`is_value_for_key_already_set()` (medigan.contribute_model.model_contributor.ModelContributor module), 57, 64

K

`key` (medigan.select_model.matched_entry.MatchedEntry attribute), 72

L

`labels` (medigan.execute_model.synthetic_dataset.SyntheticDataset attribute), 72

`list_models()` (medigan.generators.Generators module), 43, 93

`list_to_lowercase()` (medigan.utils.Utils static method), 97

`load_config_file()` (medigan.config_manager.ConfigManager module), 61, 79

`load_metadata_template()` (medigan.contribute_model.model_contributor.ModelContributor module), 57, 65

`locate_or_create_model_zip_file()` (medigan.contribute_model.zenodo_model_uploader.ZenodoModelUploader module), 67

M

`masks` (medigan.execute_model.synthetic_dataset.SyntheticDataset attribute), 72

`match_model_id()` (medigan.config_manager.ConfigManager module), 79

`matched_entries` (medigan.select_model.model_match_candidate.ModelMatchCandidate attribute), 74

`MatchedEntry` (class in medigan.select_model.matched_entry), 72

`matching_element` (medigan.select_model.matched_entry.MatchedEntry attribute), 73

`max_input_value` (medigan.model_visualizer.ModelVisualizer attribute), 49, 96

`medigan` module, 98

`medigan.config_manager` module, 59, 78

`medigan.constants` module, 80

`medigan.contribute_model` module, 69

`medigan.contribute_model.base_model_uploader` module, 61

`medigan.contribute_model.github_model_uploader` module, 62

`medigan.contribute_model.model_contributor` module, 54, 63

`medigan.contribute_model.zenodo_model_uploader` module, 66

`medigan.exceptions` module, 83

`medigan.execute_model` module, 72

`medigan.execute_model.install_model_dependencies` module, 69

`medigan.execute_model.model_executor` module, 45, 69

`medigan.execute_model.synthetic_dataset` module, 71

`medigan.generators` module, 31, 83

`medigan.model_visualizer` module, 49, 96

`medigan.select_model` module, 78

`medigan.select_model.matched_entry` module, 72

`medigan.select_model.model_match_candidate` module, 73

`medigan.select_model.model_selector` module, 50, 74

`medigan.utils` module, 97

`metadata` (medigan.contribute_model.base_model_uploader.BaseModelUploader attribute), 62

`metadata_file_path` (medigan.contribute_model.model_contributor.ModelContributor attribute), 55, 64

`makedirs()` (medigan.utils.Utils static method), 97

`model_executor` (medigan.model_visualizer.ModelVisualizer attribute), 49, 96

`model_executors` (medigan.generators.Generators attribute), 32, 83

`model_extension` (medigan.execute_model.model_executor.ModelExecutor attribute), 47, 70

`MODEL_FOLDER` (in module medigan.constants), 82

`MODEL_ID` (in module medigan.constants), 82

`model_id` (medigan.contribute_model.base_model_uploader.BaseModelUploader attribute), 62

`model_id` (medigan.contribute_model.github_model_uploader.GithubModelUploader attribute), 62

`model_id` (medigan.contribute_model.model_contributor.ModelContributor attribute), 55, 63

`model_id` (medigan.contribute_model.zenodo_model_uploader.ZenodoModelUploader attribute), 67

`model_id` (medigan.execute_model.model_executor.ModelExecutor attribute), 46, 69

`model_id` (`medigan.select_model.model_match_candidate.ModelMatchCandidate` attribute), 73
`model_ids` (`medigan.config_manager.ConfigManager` attribute), 59, 78
`model_name` (`medigan.execute_model.model_executor.ModelExecutor` attribute), 47, 70
`model_selection_dicts` (`medigan.select_model.model_selector.ModelSelector` attribute), 51, 75
`model_selector` (`medigan.generators.Generators` attribute), 32, 83
`ModelContributor` (class in `medigan.contribute_model.model_contributor`), 55, 63
`ModelExecutor` (class in `medigan.execute_model.model_executor`), 46, 69
`ModelMatchCandidate` (class in `medigan.select_model.model_match_candidate`), 73
`ModelSelector` (class in `medigan.select_model.model_selector`), 50, 74
`ModelVisualizer` (class in `medigan.model_visualizer`), 49, 96
module
 `medigan`, 98
 `medigan.config_manager`, 59, 78
 `medigan.constants`, 80
 `medigan.contribute_model`, 69
 `medigan.contribute_model.base_model_uploader`, 61
 `medigan.contribute_model.github_model_uploader`, 62
 `medigan.contribute_model.model_contributor`, 54, 63
 `medigan.contribute_model.zenodo_model_uploader`, 66
 `medigan.exceptions`, 83
 `medigan.execute_model`, 72
 `medigan.execute_model.install_model_dependencies`, 69
 `medigan.execute_model.model_executor`, 45, 69
 `medigan.execute_model.synthetic_dataset`, 71
 `medigan.generators`, 31, 83
 `medigan.model_visualizer`, 49, 96
 `medigan.select_model`, 78
 `medigan.select_model.matched_entry`, 72
 `medigan.select_model.model_match_candidate`, 73
 `medigan.select_model.model_selector`, 50, 74
 `medigan.utils`, 97
`order_dict_by_value()` (`medigan.utils.Utils` static method), 97
`other_imaging_output` (`medigan.execute_model.synthetic_dataset.SyntheticDataset` attribute), 72

P

`PACKAGE_EXTENSION` (in module `medigan.constants`), 82
`package_link` (`medigan.execute_model.model_executor.ModelExecutor` attribute), 47, 70
`package_name` (`medigan.contribute_model.model_contributor.ModelContributor` attribute), 55, 63
`package_name` (`medigan.execute_model.model_executor.ModelExecutor` attribute), 47, 70
`package_path` (`medigan.contribute_model.model_contributor.ModelContributor` attribute), 55, 63
`package_path` (`medigan.execute_model.model_executor.ModelExecutor` attribute), 47, 70
`parse_args()` (in module `medigan.execute_model.install_model_dependencies`), 69
`publish()` (`medigan.contribute_model.zenodo_model_uploader.ZenodoModelUploader` method), 68
`push()` (`medigan.contribute_model.base_model_uploader.BaseModelUploader` method), 62
`push()` (`medigan.contribute_model.github_model_uploader.GithubModelUploader` method), 63
`push()` (`medigan.contribute_model.zenodo_model_uploader.ZenodoModelUploader` method), 68
`push_to_github()` (`medigan.contribute_model.model_contributor.ModelContributor` method), 57, 65
`push_to_github()` (`medigan.generators.Generators` method), 43, 93
`push_to_zenodo()` (`medigan.contribute_model.model_contributor.ModelContributor` method), 58, 65
`push_to_zenodo()` (`medigan.generators.Generators` method), 44, 94

R

`rank_models_by_performance()` (`medigan.generators.Generators` method), 44, 94
`rank_models_by_performance()` (`medigan.select_model.model_selector.ModelSelector` method), 53, 77
`read_in_json()` (`medigan.utils.Utils` static method), 97
`recursive_search_for_values()` (`medigan.select_model.model_selector.ModelSelector` method), 53, 77

S

`samples` (*medigan.execute_model.synthetic_dataset.SyntheticDataset* attribute), 72

`serialised_model_file_path` (*medigan.execute_model.model_executor.ModelExecutor* attribute), 47, 70

`split_images_and_masks_no_ordering()` (*medigan.utils.Utils* static method), 97

`split_images_masks_and_labels()` (*medigan.utils.Utils* static method), 98

`store_dict_as()` (*medigan.utils.Utils* static method), 98

`SyntheticDataset` (class in *medigan.execute_model.synthetic_dataset*), 71

`value` (*medigan.select_model.matched_entry.MatchedEntry* attribute), 72

`visualize()` (*medigan.generators.Generators* method), 45, 95

`visualize()` (*medigan.model_visualizer.ModelVisualizer* method), 50, 96

Z

`ZENODO_API_URL` (in module *medigan.constants*), 82

`ZENODO_GENERIC_MODEL_DESCRIPTION` (in module *medigan.constants*), 82

`ZENODO_HEADERS` (in module *medigan.constants*), 82

`ZENODO_LINE_BREAK` (in module *medigan.constants*), 82

`zenodo_model_uploader` (*medigan.contribute_model.model_contributor.ModelContributor* attribute), 55, 64

T

`target_values` (*medigan.select_model.model_match_candidate.ModelMatchCandidate* attribute), 73

`target_values_operator` (*medigan.select_model.model_match_candidate.ModelMatchCandidate* attribute), 73

`TEMPLATE_FOLDER` (in module *medigan.constants*), 82

`test_model()` (*medigan.generators.Generators* method), 45, 95

`transform` (*medigan.execute_model.synthetic_dataset.SyntheticDataset* attribute), 72

`ZenodoModelUploader` (class in *medigan.contribute_model.zenodo_model_uploader*), 66

U

`unzip_and_return_unzipped_path()` (*medigan.utils.Utils* static method), 98

`unzip_archive()` (*medigan.utils.Utils* static method), 98

`upload()` (*medigan.contribute_model.zenodo_model_uploader.ZenodoModelUploader* method), 68

`upload_descriptive_data()` (*medigan.contribute_model.zenodo_model_uploader.ZenodoModelUploader* method), 68

`Utils` (class in *medigan.utils*), 97

V

`validate_and_update_model_weights_path()` (*medigan.contribute_model.model_contributor.ModelContributor* method), 58, 66

`validate_init_py_path()` (*medigan.contribute_model.model_contributor.ModelContributor* method), 58, 66

`validate_local_model_import()` (*medigan.contribute_model.model_contributor.ModelContributor* method), 58, 66

`validate_model_id()` (*medigan.contribute_model.model_contributor.ModelContributor* method), 58, 66