dcbench

Data Centric AI

Apr 15, 2022

CONTENTS

CHAPTER

ONE

WHAT IS DCBENCH?

This benchmark evaluates the steps in your machine learning workflow beyond model training and tuning. This includes feature cleaning, slice discovery, and coreset selection. We call these "data-centric" tasks because they're focused on exploring and manipulating data – not training models. `dcbench` supports a growing number of them:

- *Minimal Data Selection*: Find the smallest subset of training data on which a fixed model architecture achieves accuracy above a threshold.
- *Slice Discovery*: Identify subgroups on which a model underperforms.
- *Data Cleaning on a Budget*: Given a fixed budget, clean input features of training data to improve model performance.

`dcbench` includes tasks that look very different from one another: the inputs and outputs of the slice discovery task are not the same as those of the minimal data cleaning task. However, we think it important that researchers and practitioners be able to run evaluations on data-centric tasks across the ML lifecycle without having to learn a bunch of different APIs or rewrite evaluation scripts.

So, `dcbench` is designed to be a common home for these diverse, but related, tasks. In `dcbench` all of these tasks are structured in a similar manner and they are supported by a common Python API that makes it easy to download data, run evaluations, and compare methods.

CHAPTER

TWO

API WALKTHROUGH

```
pip install dcbench
```

2.1 Task

dcbench supports a diverse set of data-centric tasks (*e.g.* *Slice Discovery*). You can explore the supported tasks in the documentation (*Tasks*) or via the Python API:

```
In [1]: import dcbench

In [2]: dcbench.tasks
Out[2]:
                                      name                                                ␣
↪summary
minidata             Minimal Data Selection  Given a large training dataset, what is the␣
↪sm...
slice_discovery             Slice Discovery  Machine learnings models that achieve high␣
↪ove...
budgetclean      Data Cleaning on a Budget   When it comes to data preparation, data␣
↪cleani...
```

In the dcbench API, each task is represented by a *dcbench.Task* object that can be accessed by *task_id* (*e.g.* dcbench.slice_discovery). These task objects hold metadata about the task and hold pointers to task-specific *dcbench.Problem* and *dcbench.Solution* subclasses, discussed below.

2.2 Problem

Each task features a collection of *problems* (*i.e.* instances of the task). For example, the *Slice Discovery* task includes hundreds of problems across a number of different datasets. We can explore a task's problems in dcbench:

```
In [3]: dcbench.tasks["slice_discovery"].problems
Out[3]:
           alpha  ...       target_name
p_72776   0.2000  ...  wearing_lipstick
p_72793   0.6000  ...  wearing_necklace
p_72800   0.2000  ...  wearing_necklace
p_72799   0.6000  ...  wearing_necklace
```

(continues on next page)

(continued from previous page)

```
p_72802   0.2000  ...  wearing_necklace
...          ...  ...               ...
p_121367  0.4000  ...      vehicle.n.01
p_121364  0.4000  ...      vehicle.n.01
p_118660  0.0171  ...         food.n.01
p_117634  0.0171  ...      vehicle.n.01
p_117980  0.0171  ...      vehicle.n.01

[1023 rows x 6 columns]
```

All of a task's problems share the same structure and use the same evaluation scripts. This is specified via task-specific subclasses of *dcbench.Problem* (*e.g.* *SliceDiscoveryProblem*). The problems themselves are instances of these subclasses. We can access a problem using it's id:

```
In [4]: problem = dcbench.tasks["slice_discovery"].problems["p_118919"]

In [5]: problem
Out[5]: SliceDiscoveryProblem(artifacts={'activations': 'DataPanelArtifact', 'base_
↪dataset': 'VisionDatasetArtifact', 'clip': 'DataPanelArtifact', 'model': 'ModelArtifact
↪', 'test_predictions': 'DataPanelArtifact', 'test_slices': 'DataPanelArtifact', 'val_
↪predictions': 'DataPanelArtifact'}, attributes={'alpha': 0.01709975946676697, 'dataset
↪': 'imagenet', 'n_pred_slices': 5, 'slice_category': 'rare', 'slice_names': ['hay.n.01
↪'], 'target_name': 'food.n.01'})
```

2.3 Artifact

Each *problem* is made up of a set of artifacts: a dataset with features to clean, a dataset and a model to perform error analysis on. In dcbench , these artifacts are represented by instances of *dcbench.Artifact*. We can think of each *Problem* object as a container for *Artifact* objects.

```
In [6]: problem.artifacts
Out[6]:
{'activations': <dcbench.common.artifact.DataPanelArtifact at 0x7f9feb704b90>,
 'base_dataset': <dcbench.common.artifact.VisionDatasetArtifact at 0x7f9feb704850>,
 'clip': <dcbench.common.artifact.DataPanelArtifact at 0x7f9feb704890>,
 'model': <dcbench.common.artifact.ModelArtifact at 0x7f9feb704c10>,
 'test_predictions': <dcbench.common.artifact.DataPanelArtifact at 0x7f9feb704c50>,
 'test_slices': <dcbench.common.artifact.DataPanelArtifact at 0x7f9feb704c90>,
 'val_predictions': <dcbench.common.artifact.DataPanelArtifact at 0x7f9feb704cd0>}
```

Note that *Artifact* objects don't actually hold their underlying data in memory. Instead, they hold pointers to where the *Artifact* lives in dcbench cloud storage and, if it's been downloaded, where it lives locally on disk. This makes the *Problem* objects very lightweight.

dcbench includes loading functionality for each artifact type. To load an artifact into memory we can use *load()* . Note that this will also download the artifact to disk if it hasn't yet been downloaded.

```
In [7]: problem.artifacts["model"]
Out[7]: <dcbench.common.artifact.ModelArtifact at 0x7f9feb704c10>
```

Easier yet, we can use the index operator directly on *Problem* objects to both fetch the artifact and load it into memory.

```
In [8]: problem["activations"]  # shorthand for problem.artifacts["model"].load()
Out[8]: DataPanel(nrows: 9044, ncols: 2)
```

Downloading to Disk

By default, `dcbench` downloads artifacts to `~/.dcbench` but this can be configured by creating a `dcbench-config.yaml` as described in *Configuring dcbench*. To download an *Artifact* via the Python API, use *Artifact.download()*. You can also download all the artifacts in a problem with `Problem.download()`.

2.4 Solution

CHAPTER

THREE

TASKS

3.1 Minimal Data Selection

Task Details

Task ID `minidata`

Problems 1

Given a large training dataset, what is the smallest subset you can sample that still achieves some threshold of performance.

Classes: *dcbench.MiniDataProblem* *dcbench.MiniDataSolution*

Cloud Storage

We recommend downloading Artifacts through the Python API, but you can also explore the Artifacts on the Google Cloud Console.

3.1.1 Problem Artifacts

name	type	description
`train_data`	*dcbench.DataPanelArtifact*	A DataPanel of train examples with columns `id`, `input`, and `target`.
`val_data`	*dcbench.DataPanelArtifact*	A DataPanel of validation examples with columns `id`, `input`, and `target`.
`test_data`	*dcbench.DataPanelArtifact*	A DataPanel of test examples with columns `id`, `input`, and `target`.

3.1.2 Solution Artifacts

name	type	description
`train_ids`	*dcbench.YAMLArtifact*	A list of train example ids from the `id` column of `train_data`.

3.2 Slice Discovery

Task Details

Task ID `slice_discovery`

Problems 20

Machine learnings models that achieve high overall accuracy often make systematic erors on important subgroups (or *slices*) of data. When working with high-dimensional inputs (*e.g.* images, audio) where data slices are often unlabeled, identifying underperforming slices is challenging. In this task, we'll develop automated slice discovery methods that mine unstructured data for underperforming slices.

Classes: *dcbench.SliceDiscoveryProblem* *dcbench.SliceDiscoverySolution*

Cloud Storage

We recommend downloading Artifacts through the Python API, but you can also explore the Artifacts on the Google Cloud Console.

3.2.1 Problem Artifacts

name	type	description
`val_predictions`	*dcbench.DataPanelArtifact*	A DataPanel of the model's predictions with columns *id*,`target`, and *probs*.
`test_predictions`	*dcbench.DataPanelArtifact*	A DataPanel of the model's predictions with columns *id*,`target`, and *probs*.
`test_slices`	*dcbench.DataPanelArtifact*	A DataPanel of the ground truth slice labels with columns *id*, *slices*.
`activations`	*dcbench.DataPanelArtifact*	A DataPanel of the model's activations with columns *id*,`act`
`model`	*dcbench.ModelArtifact*	A trained PyTorch model to audit.
`base_dataset`	*dcbench.VisionDatasetArtifact*	A DataPanel representing the base dataset with columns *id* and *image*.
`clip`	*dcbench.DataPanelArtifact*	A DataPanel of the image embeddings from OpenAI's CLIP model

3.2.2 Solution Artifacts

name	type	description
pred_slices	*dcbench.DataPanelArtifact*	A DataPanel of predicted slice labels with columns *id* and *pred_slices*.

3.3 Data Cleaning on a Budget

Task Details

Task ID budgetclean

Problems 144

When it comes to data preparation, data cleaning is an essential yet quite costly task. If we are given a fixed cleaning budget, the challenge is to find the training data examples that would would bring the biggest positive impact on model performance if we were to clean them.

Classes: *dcbench.BudgetcleanProblem* *dcbench.BudgetcleanSolution*

Cloud Storage

We recommend downloading Artifacts through the Python API, but you can also explore the Artifacts on the Google Cloud Console.

3.3.1 Problem Artifacts

name	type	description
X_train_dirty	*dcbench.CSVArtifact*	('Features of the dirty training dataset which we need to clean. Each dirty cell contains an embedded list of clean candidate values.',)
X_train_clean	*dcbench.CSVArtifact*	Features of the clean training dataset where each dirty value from the dirty dataset is replaced with the correct clean candidate.
y_train	*dcbench.CSVArtifact*	Labels of the training dataset.
X_val	*dcbench.CSVArtifact*	Feature of the validtion dataset which can be used to guide the cleaning optimization process.
y_val	*dcbench.CSVArtifact*	Labels of the validation dataset.
X_test	*dcbench.CSVArtifact*	('Features of the test dataset used to produce the final evaluation score of the model.',)
y_test	*dcbench.CSVArtifact*	Labels of the test dataset.

3.3.2 Solution Artifacts

name	type	description
idx_selected	*dcbench.CSVArtifact*	

CHAPTER

FOUR

INSTALLING DCBENCH

This section describes how to install the `dcbench` Python package.

```
pip install dcbench
```

Optional

Some parts of `dcbench` rely on optional dependencies. If you know which optional dependencies you'd like to install, you can do so using something like `pip install dcbench[dev]` instead. See `setup.py` for a full list of optional dependencies.

4.1 Installing from branch

To install from a specific branch use the command below, replacing `main` with the name of any branch in the dcbench repository.

```
pip install "dcbench @ git+https://github.com/data-centric-ai/dcbench@main"
```

4.2 Installing from clone

You can install from a clone of the `dcbench` repo with:

```
git clone https://github.com/data-centric-ai/dcbench.git
cd dcbench
pip install -e .
```

CHAPTER

FIVE

CONFIGURING DCBENCH

Several aspects of dcbench behavior can be configured by the user. For example, one may wish to change the directory in which dcbench downloads artifacts (by default this is ~/.dcbench).

You can see the current state of the dcbench configuration with:

```
In [1]: import dcbench

In [2]: dcbench.config
Out[2]: DCBenchConfig(local_dir='/home/docs/.dcbench', public_bucket_name='dcbench',␣
↪hidden_bucket_name='dcbench-hidden', celeba_dir='/home/docs/.dcbench/datasets/celeba',␣
↪imagenet_dir='/home/docs/.dcbench/datasets/imagenet')
```

5.1 Configuring with YAML

To change the configuration create a YAML file, like the one below:

Then set the environment variable DCBENCH_CONFIG to point to the file:

```
export DCBENCH_CONFIG="/path/to/dcbench-config.yaml"
```

If you're using a conda, you can permanently set this variable for your environment:

```
conda env config vars set DCBENCH_CONFIG="path/to/dcbench-config.yaml"
conda activate env_name  # need to reactivate the environment
```

5.2 Configuring Programmatically

You can also update the config programmatically, though unlike the YAML method above, these changes will not persist beyond the lifetime of your program.

```
dcbench.config.local_dir = "/path/to/storage"
dcbench.config.public_bucket_name = "dcbench-test"
```

CHAPTER

SIX

DCBENCH PACKAGE

6.1 Subpackages

6.1.1 dcbench.common package

Submodules

dcbench.common.artifact module

class Artifact(*artifact_id*, ***kwargs*)

Bases: abc.ABC

A pointer to a unit of data (e.g. a CSV file) that is stored locally on disk and/or in a remote GCS bucket.

In DCBench, each artifact is identified by a unique artifact ID. The only state that the *Artifact* object must maintain is this ID (`self.id`). The object does not hold the actual data in memory, making it lightweight.

Artifact is an abstract base class. Different types of artifacts (e.g. a CSV file vs. a PyTorch model) have corresponding subclasses of *Artifact* (e.g. *CSVArtifact*, *ModelArtifact*).

Tip: The vast majority of users should not call the *Artifact* constructor directly. Instead, they should either create a new artifact by calling *from_data()* or load an existing artifact from a YAML file.

The class provides utilities for accessing and managing a unit of data:

- Synchronizing the local and remote copies of a unit of data: *upload()*, *download()*
- Loading the data into memory: *load()*
- Creating new artifacts from in-memory data: *from_data()*
- Serializing the pointer artifact so it can be shared: *to_yaml()*, *from_yaml()*

Parameters **artifact_id** (*str*) – The unique artifact ID.

Return type None

id

The unique artifact ID.

Type str

classmethod from_data(*data*, *artifact_id=None*)

Create a new artifact object from raw data and save the artifact to disk in the local directory specified in the config file at `config.local_dir`.

Tip: When called on the abstract base class *Artifact*, this method will infer which artifact subclass to use. If you know exactly which artifact class you'd like to use (e.g. *DataPanelArtifact*), you should call this classmethod on that subclass.

Parameters

- **data** (*Union[mk.DataPanel, pd.DataFrame, Model]*) – The raw data that will be saved to disk.
- **artifact_id** (*str, optional*) – . Defaults to None, in which case a UUID will be generated and used.

Returns A new artifact pointing to the **:arg:`data`** that was saved to disk.

Return type *Artifact*

property local_path: str

The local path to the artifact in the local directory specified in the config file at `config.local_dir`.

property remote_url: str

The URL of the artifact in the remote GCS bucket specified in the config file at `config.public_bucket_name`.

property is_downloaded: bool

Checks if artifact is downloaded to local directory specified in the config file at `config.local_dir`.

Returns True if artifact is downloaded, False otherwise.

Return type bool

property is_uploaded: bool

Checks if artifact is uploaded to GCS bucket specified in the config file at `config.public_bucket_name`.

Returns True if artifact is uploaded, False otherwise.

Return type bool

upload(*force=False*, *bucket=None*)

Uploads artifact to a GCS bucket at `self.path`, which by default is just the artifact ID with the default extension.

Parameters

- **force** (*bool, optional*) – Force upload even if artifact is already uploaded. Defaults to False.
- **bucket** (*storage.Bucket, optional*) – The GCS bucket to which the artifact is uplioaded. Defaults to None, in which case the artifact is uploaded to the bucket speciried in the config file at config.public_bucket_name.

Return type bool

Returns bool: True if artifact was uploaded, False otherwise.

download(*force=False*)

Downloads artifact from GCS bucket to the local directory specified in the config file at `config.local_dir`. The relative path to the artifact within that directory is `self.path`, which by default is just the artifact ID with the default extension.

Parameters **force** (*bool, optional*) – Force download even if artifact is already downloaded. Defaults to False.

Returns True if artifact was downloaded, False otherwise.

Return type bool

> **Warning:** By default, the GCS cache on public urls has a max-age up to an hour. Therefore, when updating an existin artifacts, changes may not be immediately reflected in subsequent downloads.
>
> See here for more details.

DEFAULT_EXT: str = ''

isdir: bool = False

abstract load()

Load the artifact into memory from disk at `self.local_path`.

Return type *Any*

abstract save(*data*)

Save data to disk at `self.local_path`.

Parameters **data** (*Any*) –

Return type None

static from_yaml(*loader*, *node*)

This function is called by the YAML loader to convert a YAML node into an Artifact object.

It should not be called directly.

Parameters **loader** (*yaml.loader.Loader*) –

static to_yaml(*dumper*, *data*)

This function is called by the YAML dumper to convert an Artifact object into a YAML node.

It should not be called directly.

Parameters

- **dumper** (*yaml.dumper.Dumper*) –
- **data** (dcbench.common.artifact.Artifact) –

class CSVArtifact(*artifact_id*, ***kwargs*)

Bases: *dcbench.common.artifact.Artifact*

Parameters **artifact_id** (*str*) –

Return type None

DEFAULT_EXT: str = 'csv'

load()

Load the artifact into memory from disk at `self.local_path`.

Return type pandas.core.frame.DataFrame

save(*data*)

Save data to disk at `self.local_path`.

Parameters **data** (*pandas.core.frame.DataFrame*) –

Return type None

class YAMLArtifact(*artifact_id*, ***kwargs*)

Bases: *dcbench.common.artifact.Artifact*

Parameters **artifact_id** (*str*) –

Return type None

DEFAULT_EXT: str = 'yaml'

load()

Load the artifact into memory from disk at `self.local_path`.

Return type *Any*

save(*data*)

Save data to disk at `self.local_path`.

Parameters **data** (*Any*) –

Return type None

class DataPanelArtifact(*artifact_id*, ***kwargs*)

Bases: *dcbench.common.artifact.Artifact*

Parameters **artifact_id** (*str*) –

Return type None

DEFAULT_EXT: str = 'mk'

isdir: bool = True

load()

Load the artifact into memory from disk at `self.local_path`.

Return type pandas.core.frame.DataFrame

save(*data*)

Save data to disk at `self.local_path`.

Parameters **data** (*meerkat.datapanel.DataPanel*) –

Return type None

class VisionDatasetArtifact(*artifact_id*, ***kwargs*)

Bases: *dcbench.common.artifact.DataPanelArtifact*

Parameters **artifact_id** (*str*) –

Return type None

DEFAULT_EXT: str = 'mk'

isdir: bool = True

COLUMN_SUBSETS = {'celeba': ['id', 'image', 'identity', 'split'], 'imagenet': ['id', 'image', 'name', 'synset']}

classmethod from_name(*name*)

Parameters **name** (*str*) –

download(*force=False*)

Downloads artifact from GCS bucket to the local directory specified in the config file at `config.local_dir`. The relative path to the artifact within that directory is `self.path`, which by default is just the artifact ID with the default extension.

Parameters **force** (*bool, optional*) – Force download even if artifact is already downloaded. Defaults to False.

Returns True if artifact was downloaded, False otherwise.

Return type bool

Warning: By default, the GCS cache on public urls has a max-age up to an hour. Therefore, when updating an existin artifacts, changes may not be immediately reflected in subsequent downloads.

See here for more details.

class ModelArtifact(*artifact_id*, ***kwargs*)

Bases: *dcbench.common.artifact.Artifact*

Parameters **artifact_id** (*str*) –

Return type None

DEFAULT_EXT: str = 'pt'

load()

Load the artifact into memory from disk at `self.local_path`.

Return type *dcbench.common.modeling.Model*

save(*data*)

Save data to disk at `self.local_path`.

Parameters **data** (dcbench.common.modeling.Model) –

Return type None

dcbench.common.artifact_container module

class ArtifactSpec(*description: 'str'*, *artifact_type: 'type'*, *optional: 'bool' = False*)

Bases: object

Parameters

- **description** (*str*) –
- **artifact_type** (*type*) –
- **optional** (*bool*) –

Return type None

description: str

artifact_type: type

optional: bool = False

class ArtifactContainer(*artifacts*, *attributes=None*, *container_id=None*)

Bases: abc.ABC, collections.abc.Mapping, *dcbench.common.table.RowMixin*

A logical collection of artifacts and attributes (simple tags describing the container), which are useful for finding, sorting and grouping containers.

Parameters

- **artifacts** (*Mapping[str, Union[Artifact, Any]]*) – A mapping with the same keys as the *ArtifactContainer.artifact_specs* (possibly excluding optional artifacts). Each value can either be an Artifact, in which case the artifact type must match the type specified in the corresponding *ArtifactSpec*, or a raw object, in which case a new artifact of the type specified in *artifact_specs* is created from the raw object and an artifact_id is generated according to the following pattern: <task_id>/<container_type>/artifacts/<container_id>/<key>.
- **attributes** (*Mapping[str, PRIMITIVE_TYPE], optional*) – A mapping with the same keys as the *ArtifactContainer.attribute_specs* (possibly excluding optional attributes). Each value must be of the type specified in the corresponding AttributeSpec. Defaults to None.
- **container_id** (*str, optional*) – The ID of the container. Defaults to None, in which case a UUID is generated.

artifacts

A dictionary of artifacts, indexed by name.

Tip: We can use the index operator directly on *ArtifactContainer* objects to both fetch the artifact, download it if necessary, and load it into memory. For example, to load the artifact "data" into memory from a container container, we can simply call container["data"], which is equivalent to calling container.artifacts["data"].download() followed by container.artifacts["data"].load().

Type Dict[str, *Artifact*]

attributes

A dictionary of attributes, indexed by name.

Tip: Accessing attributes Atttributes can be accessed via a dot-notation (as long as the attribute name does not conflict). For example, to access the attribute "data" in a container container, we can simply call container.data.

Type Dict[str, Attribute]

Notes

ArtifactContainer is an abstract base class, and should not be instantiated directly. There are two main groups of *ArtifactContainer* subclasses:

1. *dcbench.Problem* - A logical collection of artifacts and attributes that correspond to a specific problem to be solved.
   - Example subclasses: *dcbench.SliceDiscoveryProblem*, *dcbench.BudgetcleanProblem*
2. *dcbench.Solution* - A logical collection of artifacts and attributes that correspond to a solution to a problem.
   - Example subclasses: *dcbench.SliceDiscoverySolution*, *dcbench.BudgetcleanSolution*

A concrete (i.e. non-abstract) subclass of *ArtifactContainer* must include (1) a specification for the artifacts it holds, (2) a specification for the attributes used to tag it, and (3) a *task_id* linking the subclass to one of dcbench's tasks (see *Task*). For example, in the code block below we include such a specification in the definition of a simple container that holds a training dataset and a test dataset (see *dcbench.SliceDiscoveryProblem* for a real example):

```
class DemoContainer(ArtifactContainer):
    artifact_specs = {
        "train_dataset": ArtifactSpec(
            artifact_type=CSVArtifact,
            description="A CSV containing training data."
        ),
        "test_dataset": ArtifactSpec(
            artifact_type=CSVArtifact,
            description="A CSV containing test data."
        ),
    }
    attribute_specs = {
        "dataset_name": AttributeSpec(
            attribute_type=str,
            description="The name of the dataset."
        ),
    }
    task_id = "slice_discovery"
```

artifact_specs: Mapping[str, *ArtifactSpec*]

task_id: str

attribute_specs: Mapping[str, *AttributeSpec*] = {}

container_type: str = 'artifact_container'

property is_downloaded: bool

Checks if all of the artifacts in the container are downloaded to the local directory specified in the config file at `config.local_dir`.

Returns True if artifact is downloaded, False otherwise.

Return type bool

property is_uploaded: bool

Checks if all of the artifacts in the container are uploaded to the GCS bucket specified in the config file at `config.public_bucket_name`.

Returns True if artifact is uploaded, False otherwise.

Return type bool

`upload`(*force=False*, *bucket=None*)

Uploads all of the artifacts in the container to a GCS bucket, skipping artifacts that are already uploaded.

Parameters

- **force** (*bool, optional*) – Force upload even if an artifact is already uploaded. Defaults to False.
- **bucket** (*storage.Bucket, optional*) – The GCS bucket to which the artifacts are uploaded. Defaults to None, in which case the artifact is uploaded to the bucket speciried in the config file at config.public_bucket_name.

Returns True if any artifacts were uploaded, False otherwise.

Return type bool

`download`(*force=False*)

Downloads artifacts in the container from the GCS bucket specified in the config file at `config.public_bucket_name` to the local directory specified in the config file at `config.local_dir`. The relative path to the artifact within that directory is `self.path`, which by default is just the artifact ID with the default extension.

Parameters **force** (*bool, optional*) – Force download even if an artifact is already downloaded. Defaults to False.

Returns True if any artifacts were downloaded, False otherwise.

Return type bool

`static from_yaml`(*loader*, *node*)

This function is called by the YAML loader to convert a YAML node into an *ArtifactContainer* object.

It should not be called directly.

Parameters **loader** (*yaml.loader.Loader*) –

`static to_yaml`(*dumper*, *data*)

This function is called by the YAML dumper to convert an *ArtifactContainer* object into a YAML node.

It should not be called directly.

Parameters

- **dumper** (*yaml.dumper.Dumper*) –
- **data** (dcbench.common.artifact_container.ArtifactContainer) –

dcbench.common.method module

dcbench.common.modeling module

`class Model`(*config=None*)

Bases: pytorch_lightning.core.lightning.LightningModule

Parameters **config** (*dict*) –

`DEFAULT_CONFIG = {}`

training: bool

class ResNet(*num_classes*, *arch='resnet18'*, *dropout=0.0*, *pretrained=True*)

Bases: torchvision.models.resnet.ResNet

Parameters

- **num_classes** (*int*) –
- **arch** (*str*) –
- **dropout** (*float*) –
- **pretrained** (*bool*) –

ACTIVATION_DIMS = [64, 128, 256, 512]

ACTIVATION_WIDTH_HEIGHT = [64, 32, 16, 8]

RESNET_TO_ARCH = {'resnet18': [2, 2, 2, 2], 'resnet50': [3, 4, 6, 3]}

training: bool

default_transform(*img*)

Parameters **img** (*PIL.Image.Image*) –

default_train_transform(*img*)

Parameters **img** (*PIL.Image.Image*) –

class DenseNet(*num_classes*, *arch='densenet121'*, *pretrained=True*)

Bases: torchvision.models.densenet.DenseNet

Parameters

- **num_classes** (*int*) –
- **arch** (*str*) –
- **pretrained** (*bool*) –

DENSENET_TO_ARCH = {'densenet121': {'block_config': (6, 12, 24, 16), 'growth_rate': 32, 'num_init_features': 64}}

training: bool

class VisionClassifier(*config=None*)

Bases: *dcbench.common.modeling.Model*

Parameters **config** (*dict*) –

DEFAULT_CONFIG = {'arch': 'resnet18', 'lr': 0.0001, 'model_name': 'resnet', 'num_classes': 2, 'pretrained': True, 'train_transform': <function default_train_transform>, 'transform': <function default_transform>}

forward(*x*)

Same as torch.nn.Module.forward().

Parameters

- ***args** – Whatever you decide to pass into the forward method.
- ****kwargs** – Keyword arguments are also possible.

Returns Your model's output

training_step(*batch*, *batch_idx*)

Here you compute and return the training loss and some additional metrics for e.g. the progress bar or logger.

Parameters

- **batch** (Tensor | (Tensor, ...) | [Tensor, ...]) – The output of your DataLoader. A tensor, tuple or list.
- **batch_idx** (int) – Integer displaying index of this batch
- **optimizer_idx** (int) – When using multiple optimizers, this argument will also be present.
- **hiddens** (Any) – Passed in if **:paramref:`~pytorch_lightning.core.lightning.LightningModule.truncated_bpt** > 0.

Returns

Any of.

- Tensor - The loss tensor
- dict - A dictionary. Can include any keys, but must include the key 'loss'
- **None - Training will skip to the next batch. This is only for automatic optimization.** This is not supported for multi-GPU, TPU, IPU, or DeepSpeed.

In this step you'd normally do the forward pass and calculate the loss for a batch. You can also do fancier things like multiple forward passes or something model specific.

Example:

```
def training_step(self, batch, batch_idx):
    x, y, z = batch
    out = self.encoder(x)
    loss = self.loss(out, x)
    return loss
```

If you define multiple optimizers, this step will be called with an additional optimizer_idx parameter.

```
# Multiple optimizers (e.g.: GANs)
def training_step(self, batch, batch_idx, optimizer_idx):
    if optimizer_idx == 0:
        # do training_step with encoder
        ...
    if optimizer_idx == 1:
        # do training_step with decoder
        ...
```

If you add truncated back propagation through time you will also get an additional argument with the hidden states of the previous step.

```
# Truncated back-propagation through time
def training_step(self, batch, batch_idx, hiddens):
    # hiddens are the hidden states from the previous truncated backprop step
    out, hiddens = self.lstm(data, hiddens)
```

(continues on next page)

(continued from previous page)

```
    loss = ...
    return {"loss": loss, "hiddens": hiddens}
```

Note: The loss value shown in the progress bar is smoothed (averaged) over the last values, so it differs from the actual loss returned in train/validation step.

---

validation_step(*batch*, *batch_idx*)

Operates on a single batch of data from the validation set. In this step you'd might generate examples or calculate anything of interest like accuracy.

```
# the pseudocode for these calls
val_outs = []
for val_batch in val_data:
    out = validation_step(val_batch)
    val_outs.append(out)
validation_epoch_end(val_outs)
```

Parameters

- **batch** – The output of your `DataLoader`.
- **batch_idx** – The index of this batch.
- **dataloader_idx** – The index of the dataloader that produced this batch. (only if multiple val dataloaders used)

Returns

- Any object or value
- `None` - Validation will skip to the next batch

```
# pseudocode of order
val_outs = []
for val_batch in val_data:
    out = validation_step(val_batch)
    if defined("validation_step_end"):
        out = validation_step_end(out)
    val_outs.append(out)
val_outs = validation_epoch_end(val_outs)
```

```
# if you have one val dataloader:
def validation_step(self, batch, batch_idx):
    ...


# if you have multiple val dataloaders:
def validation_step(self, batch, batch_idx, dataloader_idx=0):
    ...
```

Examples:

```
# CASE 1: A single validation dataset
def validation_step(self, batch, batch_idx):
    x, y = batch

    # implement your own
    out = self(x)
    loss = self.loss(out, y)

    # log 6 example images
    # or generated text... or whatever
    sample_imgs = x[:6]
    grid = torchvision.utils.make_grid(sample_imgs)
    self.logger.experiment.add_image('example_images', grid, 0)

    # calculate acc
    labels_hat = torch.argmax(out, dim=1)
    val_acc = torch.sum(y == labels_hat).item() / (len(y) * 1.0)

    # log the outputs!
    self.log_dict({'val_loss': loss, 'val_acc': val_acc})
```

If you pass in multiple val dataloaders, *validation_step()* will have an additional argument. We recommend setting the default value of 0 so that you can quickly switch between single and multiple dataloaders.

```
# CASE 2: multiple validation dataloaders
def validation_step(self, batch, batch_idx, dataloader_idx=0):
    # dataloader_idx tells you which dataset this is.
    ...
```

Note: If you don't need to validate you don't need to implement this method.

Note: When the *validation_step()* is called, the model has been put in eval mode and PyTorch gradients have been disabled. At the end of validation, the model goes back to training mode and gradients are enabled.

validation_epoch_end(*outputs*)

Called at the end of the validation epoch with the outputs of all validation steps.

```
# the pseudocode for these calls
val_outs = []
for val_batch in val_data:
    out = validation_step(val_batch)
    val_outs.append(out)
validation_epoch_end(val_outs)
```

Parameters **outputs** – List of outputs you defined in *validation_step()*, or if there are multiple dataloaders, a list containing a list of outputs for each dataloader.

Returns None

Return type None

Note: If you didn't define a *validation_step()*, this won't be called.

Examples

With a single dataloader:

```
def validation_epoch_end(self, val_step_outputs):
    for out in val_step_outputs:
        ...
```

With multiple dataloaders, *outputs* will be a list of lists. The outer list contains one entry per dataloader, while the inner list contains the individual outputs of each validation step for that dataloader.

```
def validation_epoch_end(self, outputs):
    for dataloader_output_result in outputs:
        dataloader_outs = dataloader_output_result.dataloader_i_outputs

    self.log("final_metric", final_value)
```

test_epoch_end(*outputs*)

Called at the end of a test epoch with the output of all test steps.

```
# the pseudocode for these calls
test_outs = []
for test_batch in test_data:
    out = test_step(test_batch)
    test_outs.append(out)
test_epoch_end(test_outs)
```

Parameters **outputs** – List of outputs you defined in `test_step_end()`, or if there are multiple dataloaders, a list containing a list of outputs for each dataloader

Returns None

Return type None

Note: If you didn't define a *test_step()*, this won't be called.

Examples

With a single dataloader:

```
def test_epoch_end(self, outputs):
    # do something with the outputs of all test batches
    all_test_preds = test_step_outputs.predictions

    some_result = calc_all_results(all_test_preds)
    self.log(some_result)
```

With multiple dataloaders, *outputs* will be a list of lists. The outer list contains one entry per dataloader, while the inner list contains the individual outputs of each test step for that dataloader.

```
def test_epoch_end(self, outputs):
    final_value = 0
    for dataloader_outputs in outputs:
        for test_step_out in dataloader_outputs:
            # do something
            final_value += test_step_out

    self.log("final_metric", final_value)
```

test_step(*batch*, *batch_idx*)

Operates on a single batch of data from the test set. In this step you'd normally generate examples or calculate anything of interest such as accuracy.

```
# the pseudocode for these calls
test_outs = []
for test_batch in test_data:
    out = test_step(test_batch)
    test_outs.append(out)
test_epoch_end(test_outs)
```

Parameters

- **batch** – The output of your `DataLoader`.
- **batch_idx** – The index of this batch.
- **dataloader_id** – The index of the dataloader that produced this batch. (only if multiple test dataloaders used).

Returns

Any of.

- Any object or value
- `None` - Testing will skip to the next batch

```
# if you have one test dataloader:
def test_step(self, batch, batch_idx):
    ...


# if you have multiple test dataloaders:
def test_step(self, batch, batch_idx, dataloader_idx=0):
    ...
```

Examples:

```
# CASE 1: A single test dataset
def test_step(self, batch, batch_idx):
    x, y = batch

    # implement your own
```

(continues on next page)

(continued from previous page)

```
    out = self(x)
    loss = self.loss(out, y)

    # log 6 example images
    # or generated text... or whatever
    sample_imgs = x[:6]
    grid = torchvision.utils.make_grid(sample_imgs)
    self.logger.experiment.add_image('example_images', grid, 0)

    # calculate acc
    labels_hat = torch.argmax(out, dim=1)
    test_acc = torch.sum(y == labels_hat).item() / (len(y) * 1.0)

    # log the outputs!
    self.log_dict({'test_loss': loss, 'test_acc': test_acc})
```

If you pass in multiple test dataloaders, *test_step()* will have an additional argument. We recommend setting the default value of 0 so that you can quickly switch between single and multiple dataloaders.

```
# CASE 2: multiple test dataloaders
def test_step(self, batch, batch_idx, dataloader_idx=0):
    # dataloader_idx tells you which dataset this is.
    ...
```

Note: If you don't need to test you don't need to implement this method.

Note: When the *test_step()* is called, the model has been put in eval mode and PyTorch gradients have been disabled. At the end of the test epoch, the model goes back to training mode and gradients are enabled.

`configure_optimizers`()

Choose what optimizers and learning-rate schedulers to use in your optimization. Normally you'd need one. But in the case of GANs or similar you might have multiple.

Returns

Any of these 6 options.

- **Single optimizer**.
- **List or Tuple** of optimizers.
- **Two lists** - The first list has multiple optimizers, and the second has multiple LR schedulers (or multiple `lr_scheduler_config`).
- **Dictionary**, with an `"optimizer"` key, and (optionally) a `"lr_scheduler"` key whose value is a single LR scheduler or `lr_scheduler_config`.
- **Tuple of dictionaries** as described above, with an optional `"frequency"` key.
- **None** - Fit will run without any optimizer.

The `lr_scheduler_config` is a dictionary which contains the scheduler and its associated configuration. The default configuration is shown below.

```
lr_scheduler_config = {
    # REQUIRED: The scheduler instance
    "scheduler": lr_scheduler,
    # The unit of the scheduler's step size, could also be 'step'.
    # 'epoch' updates the scheduler on epoch end whereas 'step'
    # updates it after a optimizer update.
    "interval": "epoch",
    # How many epochs/steps should pass between calls to
    # `scheduler.step()`. 1 corresponds to updating the learning
    # rate after every epoch/step.
    "frequency": 1,
    # Metric to to monitor for schedulers like `ReduceLROnPlateau`
    "monitor": "val_loss",
    # If set to `True`, will enforce that the value specified 'monitor'
    # is available when the scheduler is updated, thus stopping
    # training if not found. If set to `False`, it will only produce a warning
    "strict": True,
    # If using the `LearningRateMonitor` callback to monitor the
    # learning rate progress, this keyword can be used to specify
    # a custom logged name
    "name": None,
}
```

When there are schedulers in which the `.step()` method is conditioned on a value, such as the `torch.optim.lr_scheduler.ReduceLROnPlateau` scheduler, Lightning requires that the `lr_scheduler_config` contains the keyword `"monitor"` set to the metric name that the scheduler should be conditioned on.

Metrics can be made available to monitor by simply logging it using `self.log('metric_to_track', metric_val)` in your `LightningModule`.

Note: The `frequency` value specified in a dict along with the `optimizer` key is an int corresponding to the number of sequential batches optimized with the specific optimizer. It should be given to none or to all of the optimizers. There is a difference between passing multiple optimizers in a list, and passing multiple optimizers in dictionaries with a frequency of 1:

- In the former case, all optimizers will operate on the given batch in each optimization step.
- In the latter, only one optimizer will operate on the given batch at every step.

This is different from the `frequency` value specified in the `lr_scheduler_config` mentioned above.

```
def configure_optimizers(self):
    optimizer_one = torch.optim.SGD(self.model.parameters(), lr=0.01)
    optimizer_two = torch.optim.SGD(self.model.parameters(), lr=0.01)
    return [
        {"optimizer": optimizer_one, "frequency": 5},
        {"optimizer": optimizer_two, "frequency": 10},
    ]
```

In this example, the first optimizer will be used for the first 5 steps, the second optimizer for the next 10 steps and that cycle will continue. If an LR scheduler is specified for an optimizer using the `lr_scheduler` key in the above dict, the scheduler will only be updated when its optimizer is being used.

Examples:

```python
# most cases. no learning rate scheduler
def configure_optimizers(self):
    return Adam(self.parameters(), lr=1e-3)

# multiple optimizer case (e.g.: GAN)
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    return gen_opt, dis_opt

# example with learning rate schedulers
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    dis_sch = CosineAnnealing(dis_opt, T_max=10)
    return [gen_opt, dis_opt], [dis_sch]

# example with step-based learning rate schedulers
# each optimizer has its own scheduler
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    gen_sch = {
        'scheduler': ExponentialLR(gen_opt, 0.99),
        'interval': 'step'  # called after each training step
    }
    dis_sch = CosineAnnealing(dis_opt, T_max=10) # called every epoch
    return [gen_opt, dis_opt], [gen_sch, dis_sch]

# example with optimizer frequencies
# see training procedure in `Improved Training of Wasserstein GANs`, Algorithm 1
# https://arxiv.org/abs/1704.00028
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    n_critic = 5
    return (
        {'optimizer': dis_opt, 'frequency': n_critic},
        {'optimizer': gen_opt, 'frequency': 1}
    )
```

Note: Some things to know:

- Lightning calls `.backward()` and `.step()` on each optimizer and learning rate scheduler as needed.
- If you use 16-bit precision (`precision=16`), Lightning will automatically handle the optimizers.
- If you use multiple optimizers, *training_step()* will have an additional `optimizer_idx` parameter.
- If you use `torch.optim.LBFGS`, Lightning handles the closure function automatically for you.
- If you use multiple optimizers, gradients will be calculated only for the parameters of current optimizer at each training step.
- If you need to control how often those optimizers step or override the default `.step()` schedule,

override the `optimizer_step()` hook.

training: bool

trainer: Optional['pl.Trainer']

precision: int

prepare_data_per_node: bool

allow_zero_length_dataloader_with_multiple_devices: bool

dcbench.common.problem module

class Problem(*artifacts*, *attributes=None*, *container_id=None*)

Bases: *dcbench.common.artifact_container.ArtifactContainer*

A logical collection of :class:`Artifact`s and "attributes" that correspond to a specific problem to be solved.

See the walkthrough section on *Problem* for more information.

Parameters

- **artifacts** (*Mapping[str, Artifact]*) –
- **attributes** (*Mapping[str, Attribute]*) –
- **container_id** (*str*) –

container_type: str = 'problem'

name: str

summary: str

task_id: str

solution_class: type

abstract solve(***kwargs*)

Parameters **kwargs** (*Any*) –

Return type *Solution*

abstract evaluate(*solution*)

Parameters **solution** (Solution) –

Return type *Result*

class ProblemTable(*data*)

Bases: *dcbench.common.table.Table*

trial(*solver=None*)

Parameters **solver** (*Optional[Callable[[Problem], Solution]]*) –

Return type *Trial*

dcbench.common.result module

class Result(*id*, *attributes=None*)

Bases: *dcbench.common.table.RowMixin*

Parameters

- **id** (*str*) –
- **attributes** (*Mapping[str, Union[int, float, str, bool]]*) –

attribute_specs: Mapping[str, *dcbench.common.table.AttributeSpec*]

dcbench.common.solution module

class Result(*source*)

Bases: Mapping

save(*path*)

Parameters **path** (*str*) –

Return type None

static load(*path*)

Parameters **path** (*str*) –

Return type *dcbench.common.solution.Result*

class Solution(*artifacts*, *attributes=None*, *container_id=None*)

Bases: *dcbench.common.artifact_container.ArtifactContainer*

Parameters

- **artifacts** (*Mapping[str, Artifact]*) –
- **attributes** (*Mapping[str, Attribute]*) –
- **container_id** (*str*) –

container_type: str = 'solution'

dcbench.common.solve module

dcbench.common.solver module

solver(*id*, *summary*)

Parameters

- **id** (*str*) –
- **summary** (*str*) –

dcbench.common.table module

class AttributeSpec(*description: str*, *attribute_type: type*, *optional: bool = False*)

Bases: object

Parameters

- **description** (*str*) –
- **attribute_type** (*type*) –
- **optional** (*bool*) –

Return type None

description: str

attribute_type: type

optional: bool = False

class RowMixin(*id*, *attributes=None*)

Bases: object

Parameters

- **id** (*str*) –
- **attributes** (*Mapping[str, Union[int, float, str, bool]]*) –

attribute_specs: Mapping[str, *dcbench.common.table.AttributeSpec*]

property attributes: Optional[Mapping[str, Union[int, float, str, bool]]]

class RowUnion(*id*, *elements*)

Bases: *dcbench.common.table.RowMixin*

Parameters

- **id** (*str*) –
- **elements** (*Sequence[dcbench.common.table.RowMixin]*) –

attribute_specs: Mapping[str, *dcbench.common.table.AttributeSpec*]

predicate(*a*, *b*)

Parameters

- **a** (*Union[int, float, str, bool]*) –
- **b** (*Union[int, float, str, bool, slice, Sequence[Union[int, float, str, bool]]]*) –

Return type bool

class Table(*data*)

Bases: Mapping[str, *dcbench.common.table.RowMixin*]

property df

where(***kwargs*)

Parameters **kwargs** (*Union[int, float, str, bool, slice, Sequence[Union[int, float, str, bool]]]*) –

Return type *dcbench.common.table.Table*

average(**targets*, *groupby=None*, *std=False*)

Parameters

- **targets** (*str*) –
- **groupby** (*Optional[Sequence[str]]*) –
- **std** (*bool*) –

Return type *dcbench.common.table.Table*

dcbench.common.task module

class Task(*task_id*, *name*, *summary*, *problem_class*, *solution_class*, *baselines=Empty DataFrame Columns: [] Index: []*)

Bases: *dcbench.common.table.RowMixin*

Task(task_id: str, name: str, summary: str, problem_class: type, solution_class: type, baselines: dcbench.common.table.Table = Empty DataFrame Columns: [] Index: [])

Parameters

- **task_id** (*str*) –
- **name** (*str*) –
- **summary** (*str*) –
- **problem_class** (*type*) –
- **solution_class** (*type*) –
- **baselines** (dcbench.common.table.Table) –

Return type None

task_id: str

name: str

summary: str

problem_class: type

solution_class: type

baselines: *dcbench.common.table.Table* = Empty DataFrame Columns: [] Index: []

property problems_path

property local_problems_path

property remote_problems_url

write_problems(*containers*, *append=True*)

Parameters

- **containers** (*List[dcbench.common.artifact_container.ArtifactContainer]*) –
- **append** (*bool*) –

upload_problems(*include_artifacts=False*, *force=True*)

Uploads the problems to the remote storage.

Parameters

- **include_artifacts** (*bool*) – If True, also uploads the artifacts of the problems.
- **force** (*bool*) – If True, if the problem overwrites the remote problems. Defaults to True. .. warning:

```
It is somewhat dangerous to set `force=False`, as this could lead
to remote and local problems being out of sync.
```

download_problems(*include_artifacts=False*)

Parameters **include_artifacts** (*bool*) –

property problems

dcbench.common.trial module

class Problem

Bases: object

class Solution

Bases: object

class Trial(*problems=None*, *solver=None*)

Bases: *dcbench.common.table.Table*

evaluate(*repeat=1*, *quiet=False*)

Parameters

- **repeat** (*int*) –
- **quiet** (*bool*) –

Return type *dcbench.common.trial.Trial*

save()

Return type None

dcbench.common.utils module

Module contents

class Artifact(*artifact_id*, ***kwargs*)

Bases: abc.ABC

A pointer to a unit of data (e.g. a CSV file) that is stored locally on disk and/or in a remote GCS bucket.

In DCBench, each artifact is identified by a unique artifact ID. The only state that the *Artifact* object must maintain is this ID (self.id). The object does not hold the actual data in memory, making it lightweight.

Artifact is an abstract base class. Different types of artifacts (e.g. a CSV file vs. a PyTorch model) have corresponding subclasses of *Artifact* (e.g. CSVArtifact, ModelArtifact).

Tip: The vast majority of users should not call the *Artifact* constructor directly. Instead, they should either create a new artifact by calling *from_data()* or load an existing artifact from a YAML file.

The class provides utilities for accessing and managing a unit of data:

- Synchronizing the local and remote copies of a unit of data: *upload()*, *download()*
- Loading the data into memory: *load()*
- Creating new artifacts from in-memory data: *from_data()*
- Serializing the pointer artifact so it can be shared: *to_yaml()*, *from_yaml()*

Parameters **artifact_id** (*str*) – The unique artifact ID.

Return type None

id

The unique artifact ID.

Type str

classmethod from_data(*data*, *artifact_id=None*)

Create a new artifact object from raw data and save the artifact to disk in the local directory specified in the config file at config.local_dir.

Tip: When called on the abstract base class *Artifact*, this method will infer which artifact subclass to use. If you know exactly which artifact class you'd like to use (e.g. DataPanelArtifact), you should call this classmethod on that subclass.

Parameters

- **data** (*Union[mk.DataPanel, pd.DataFrame, Model]*) – The raw data that will be saved to disk.
- **artifact_id** (*str, optional*) – . Defaults to None, in which case a UUID will be generated and used.

Returns A new artifact pointing to the **:arg:`data`** that was saved to disk.

Return type *Artifact*

property local_path: str

The local path to the artifact in the local directory specified in the config file at config.local_dir.

property remote_url: str

The URL of the artifact in the remote GCS bucket specified in the config file at config.public_bucket_name.

property is_downloaded: bool

Checks if artifact is downloaded to local directory specified in the config file at config.local_dir.

Returns True if artifact is downloaded, False otherwise.

Return type bool

property is_uploaded: bool

Checks if artifact is uploaded to GCS bucket specified in the config file at config.public_bucket_name.

Returns True if artifact is uploaded, False otherwise.

Return type bool

upload(*force=False*, *bucket=None*)

Uploads artifact to a GCS bucket at self.path, which by default is just the artifact ID with the default extension.

Parameters

- **force** (*bool, optional*) – Force upload even if artifact is already uploaded. Defaults to False.
- **bucket** (*storage.Bucket, optional*) – The GCS bucket to which the artifact is uplioaded. Defaults to None, in which case the artifact is uploaded to the bucket speciried in the config file at config.public_bucket_name.

Return type bool

Returns bool: True if artifact was uploaded, False otherwise.

download(*force=False*)

Downloads artifact from GCS bucket to the local directory specified in the config file at config.local_dir. The relative path to the artifact within that directory is self.path, which by default is just the artifact ID with the default extension.

Parameters **force** (*bool, optional*) – Force download even if artifact is already downloaded. Defaults to False.

Returns True if artifact was downloaded, False otherwise.

Return type bool

> **Warning:** By default, the GCS cache on public urls has a max-age up to an hour. Therefore, when updating an existin artifacts, changes may not be immediately reflected in subsequent downloads.
>
> See here for more details.

DEFAULT_EXT: str = ''

isdir: bool = False

abstract load()

Load the artifact into memory from disk at `self.local_path`.

Return type *Any*

abstract save(*data*)

Save data to disk at `self.local_path`.

Parameters **data** (*Any*) –

Return type None

static from_yaml(*loader*, *node*)

This function is called by the YAML loader to convert a YAML node into an Artifact object.

It should not be called directly.

Parameters **loader** (*yaml.loader.Loader*) –

static to_yaml(*dumper*, *data*)

This function is called by the YAML dumper to convert an Artifact object into a YAML node.

It should not be called directly.

Parameters

- **dumper** (*yaml.dumper.Dumper*) –
- **data** (dcbench.common.artifact.Artifact) –

class Problem(*artifacts*, *attributes=None*, *container_id=None*)

Bases: *dcbench.common.artifact_container.ArtifactContainer*

A logical collection of :class:`Artifact`s and "attributes" that correspond to a specific problem to be solved.

See the walkthrough section on *Problem* for more information.

Parameters

- **artifacts** (*Mapping[str, Artifact]*) –
- **attributes** (*Mapping[str, Attribute]*) –
- **container_id** (*str*) –

container_type: str = 'problem'

name: str

summary: str

task_id: str

solution_class: type

abstract solve(***kwargs*)

Parameters **kwargs** (*Any*) –

Return type *Solution*

abstract evaluate(*solution*)

Parameters **solution** (Solution) –

Return type *Result*

artifact_specs: Mapping[str, *ArtifactSpec*]

class Solution(*artifacts*, *attributes=None*, *container_id=None*)

Bases: *dcbench.common.artifact_container.ArtifactContainer*

Parameters

- **artifacts** (*Mapping[str, Artifact]*) –
- **attributes** (*Mapping[str, Attribute]*) –
- **container_id** (*str*) –

container_type: str = 'solution'

artifact_specs: Mapping[str, *ArtifactSpec*]

task_id: str

class Task(*task_id*, *name*, *summary*, *problem_class*, *solution_class*, *baselines=Empty DataFrame Columns: [] Index: []*)

Bases: *dcbench.common.table.RowMixin*

Task(task_id: str, name: str, summary: str, problem_class: type, solution_class: type, baselines: dcbench.common.table.Table = Empty DataFrame Columns: [] Index: [])

Parameters

- **task_id** (*str*) –
- **name** (*str*) –
- **summary** (*str*) –
- **problem_class** (*type*) –
- **solution_class** (*type*) –
- **baselines** (dcbench.common.table.Table) –

Return type None

task_id: str

name: str

summary: str

problem_class: type

solution_class: type

baselines: *dcbench.common.table.Table* = Empty DataFrame Columns: [] Index: []

property problems_path

property local_problems_path

property remote_problems_url

write_problems(*containers*, *append=True*)

Parameters

- **containers** (*List[dcbench.common.artifact_container.ArtifactContainer]*) –
- **append** (*bool*) –

upload_problems(*include_artifacts=False*, *force=True*)

Uploads the problems to the remote storage.

Parameters

- **include_artifacts** (*bool*) – If True, also uploads the artifacts of the problems.
- **force** (*bool*) – If True, if the problem overwrites the remote problems. Defaults to True. .. warning:

```
It is somewhat dangerous to set `force=False`, as this could lead
to remote and local problems being out of sync.
```

download_problems(*include_artifacts=False*)

Parameters **include_artifacts** (*bool*) –

property problems

attribute_specs: Mapping[str, *AttributeSpec*]

class Table(*data*)

Bases: Mapping[str, *dcbench.common.table.RowMixin*]

property df

where(***kwargs*)

Parameters **kwargs** (*Union[int, float, str, bool, slice, Sequence[Union[int, float, str, bool]]]*) –

Return type *dcbench.common.table.Table*

average(**targets*, *groupby=None*, *std=False*)

Parameters

- **targets** (*str*) –
- **groupby** (*Optional[Sequence[str]]*) –
- **std** (*bool*) –

Return type *dcbench.common.table.Table*

class Result(*id*, *attributes=None*)

Bases: *dcbench.common.table.RowMixin*

Parameters

- **id** (*str*) –
- **attributes** (*Mapping[str, Union[int, float, str, bool]]*) –

attribute_specs: Mapping[str, *dcbench.common.table.AttributeSpec*]

6.1.2 dcbench.tasks package

Subpackages

dcbench.tasks.budgetclean package

Submodules

dcbench.tasks.budgetclean.baselines module

random_clean(*problem*, *seed=1337*)

Parameters

- **problem** (dcbench.tasks.budgetclean.problem.BudgetcleanProblem) –
- **seed** (*int*) –

Return type *dcbench.tasks.budgetclean.problem.BudgetcleanSolution*

cp_clean(*problem*, *seed=1337*, *n_jobs=8*, *kparam=3*)

Parameters

- **problem** (dcbench.tasks.budgetclean.problem.BudgetcleanProblem) –
- **seed** (*int*) –

Return type *dcbench.tasks.budgetclean.problem.BudgetcleanSolution*

dcbench.tasks.budgetclean.common module

class Preprocessor(*num_strategy='mean'*)

Bases: object

docstring for Preprocessor.

fit(*X_train*, *y_train*, *X_full=None*)

transform(*X=None*, *y=None*)

dcbench.tasks.budgetclean.problem module

class BudgetcleanSolution(*artifacts*, *attributes=None*, *container_id=None*)

Bases: *dcbench.common.solution.Solution*

Parameters

- **artifacts** (*Mapping[str, Artifact]*) –
- **attributes** (*Mapping[str, Attribute]*) –
- **container_id** (*str*) –

artifact_specs: Mapping[str, *dcbench.common.artifact_container.ArtifactSpec*] = {'idx_selected': ArtifactSpec(description='', artifact_type=<class 'dcbench.common.artifact.CSVArtifact'>, optional=False)}

task_id: str

class BudgetcleanProblem(*artifacts*, *attributes=None*, *container_id=None*)

Bases: *dcbench.common.problem.Problem*

Parameters

- **artifacts** (*Mapping[str, Artifact]*) –
- **attributes** (*Mapping[str, Attribute]*) –
- **container_id** (*str*) –

artifact_specs: Mapping[str, *dcbench.common.artifact_container.ArtifactSpec*] = {'X_test': ArtifactSpec(description=('Features of the test dataset used to produce the final evaluation score of the model.',), artifact_type=<class 'dcbench.common.artifact.CSVArtifact'>, optional=False), 'X_train_clean': ArtifactSpec(description='Features of the clean training dataset where each dirty value from the dirty dataset is replaced with the correct clean candidate.', artifact_type=<class 'dcbench.common.artifact.CSVArtifact'>, optional=False), 'X_train_dirty': ArtifactSpec(description=('Features of the dirty training dataset which we need to clean. Each dirty cell contains an embedded list of clean candidate values.',), artifact_type=<class 'dcbench.common.artifact.CSVArtifact'>, optional=False), 'X_val': ArtifactSpec(description='Feature of the validtion dataset which can be used to guide the cleaning optimization process.', artifact_type=<class 'dcbench.common.artifact.CSVArtifact'>, optional=False), 'y_test': ArtifactSpec(description='Labels of the test dataset.', artifact_type=<class 'dcbench.common.artifact.CSVArtifact'>, optional=False), 'y_train': ArtifactSpec(description='Labels of the training dataset.', artifact_type=<class 'dcbench.common.artifact.CSVArtifact'>, optional=False), 'y_val': ArtifactSpec(description='Labels of the validation dataset.', artifact_type=<class 'dcbench.common.artifact.CSVArtifact'>, optional=False)}

attribute_specs: Mapping[str, *AttributeSpec*] = {'budget': AttributeSpec(description='TODO', attribute_type=<class 'float'>, optional=False), 'dataset': AttributeSpec(description='TODO', attribute_type=<class 'str'>, optional=False), 'mode': AttributeSpec(description='TODO', attribute_type=<class 'str'>, optional=False), 'model': AttributeSpec(description='TODO', attribute_type=<class 'str'>, optional=False)}

task_id: str = 'budgetclean'

classmethod list()

classmethod from_id(*scenario_id*)

Parameters **scenario_id** (*str*) –

solve(*idx_selected*, ***kwargs*)

Parameters

- **idx_selected** (*Any*) –
- **kwargs** (*Any*) –

Return type *dcbench.common.solution.Solution*

evaluate(*solution*)

Parameters **solution** (dcbench.tasks.budgetclean.problem.BudgetcleanSolution) –

Return type *dcbench.common.result.Result*

name: str

summary: str

solution_class: type

Module contents

dcbench.tasks.minidata package

Submodules

dcbench.tasks.minidata.unagi_configs module

Module contents

class MiniDataSolution(*artifacts*, *attributes=None*, *container_id=None*)

Bases: *dcbench.common.solution.Solution*

Parameters

- **artifacts** (*Mapping[str, Artifact]*) –
- **attributes** (*Mapping[str, Attribute]*) –
- **container_id** (*str*) –

artifact_specs: Mapping[str, *dcbench.common.artifact_container.ArtifactSpec*] = {'train_ids': ArtifactSpec(description='A list of train example ids from the ``id`` column of ``train_data``.', artifact_type=<class 'dcbench.common.artifact.YAMLArtifact'>, optional=False)}

task_id: str = 'minidata'

classmethod from_ids(*train_ids*, *problem_id*)

Parameters

- **train_ids** (*Sequence[str]*) –
- **problem_id** (*str*) –

class MiniDataProblem(*artifacts*, *attributes=None*, *container_id=None*)

Bases: *dcbench.common.problem.Problem*

Parameters

- **artifacts** (*Mapping[str, Artifact]*) –
- **attributes** (*Mapping[str, Attribute]*) –
- **container_id** (*str*) –

artifact_specs: Mapping[str, *dcbench.common.artifact_container.ArtifactSpec*] = {'test_data': ArtifactSpec(description='A DataPanel of test examples with columns ``id``, ``input``, and ``target``.', artifact_type=<class 'dcbench.common.artifact.DataPanelArtifact'>, optional=False), 'train_data': ArtifactSpec(description='A DataPanel of train examples with columns ``id``, ``input``, and ``target``.', artifact_type=<class 'dcbench.common.artifact.DataPanelArtifact'>, optional=False), 'val_data': ArtifactSpec(description='A DataPanel of validation examples with columns ``id``, ``input``, and ``target``.', artifact_type=<class 'dcbench.common.artifact.DataPanelArtifact'>, optional=False)}

task_id: str = 'minidata'

solve(*idx_selected*, ***kwargs*)

Parameters

- **idx_selected** (*Any*) –
- **kwargs** (*Any*) –

Return type *dcbench.common.solution.Solution*

evaluate(*solution*)

Parameters **solution** (dcbench.common.solution.Solution) –

name: str

summary: str

solution_class: type

dcbench.tasks.slice_discovery package

Submodules

dcbench.tasks.slice_discovery.baselines module

confusion_sdm(*problem*)

A simple slice discovery method that returns a slice corresponding to each cell of the confusion matrix. For example, for a binary prediction task, this sdm will return 4 slices corresponding to true positives, false positives, true negatives and false negatives.

Parameters **problem** (SliceDiscoveryProblem) – The slice discovery problem.

Returns The predicted slices.

Return type *SliceDiscoverySolution*

domino_sdm(*problem*)

Parameters **problem** (dcbench.tasks.slice_discovery.problem.SliceDiscoveryProblem) –

Return type *dcbench.tasks.slice_discovery.problem.SliceDiscoverySolution*

dcbench.tasks.slice_discovery.metrics module

precision_at_k(*slice*, *pred_slice*, *k=25*)

Parameters

- **slice** (*numpy.ndarray*) –
- **pred_slice** (*numpy.ndarray*) –
- **k** (*int*) –

recall_at_k(*slice*, *pred_slice*, *k=25*)

Parameters

- **slice** (*numpy.ndarray*) –
- **pred_slice** (*numpy.ndarray*) –
- **k** (*int*) –

compute_metrics(*slices*, *pred_slices*)

[summary]

Parameters

- **slices** (*np.ndarray*) – [description]
- **pred_slices** (*np.ndarray*) – [description]

Returns [description]

Return type dict

dcbench.tasks.slice_discovery.problem module

class SliceDiscoverySolution(*artifacts*, *attributes=None*, *container_id=None*)

Bases: *dcbench.common.solution.Solution*

Parameters

- **artifacts** (*Mapping[str, Artifact]*) –
- **attributes** (*Mapping[str, Attribute]*) –
- **container_id** (*str*) –

artifact_specs: Mapping[str, *dcbench.common.artifact_container.ArtifactSpec*] = {'pred_slices': ArtifactSpec(description='A DataPanel of predicted slice labels with columns `id` and `pred_slices`.', artifact_type=<class 'dcbench.common.artifact.DataPanelArtifact'>, optional=False)}

attribute_specs: Mapping[str, *AttributeSpec*] = {'problem_id': AttributeSpec(description='A unique identifier for this problem.', attribute_type=<class 'str'>, optional=False)}

task_id: str = 'slice_discovery'

class SliceDiscoveryProblem(*artifacts*, *attributes=None*, *container_id=None*)

Bases: *dcbench.common.problem.Problem*

Parameters

- **artifacts** (*Mapping[str, Artifact]*) –
- **attributes** (*Mapping[str, Attribute]*) –
- **container_id** (*str*) –

artifact_specs: Mapping[str, *dcbench.common.artifact_container.ArtifactSpec*] = {'activations': ArtifactSpec(description="A DataPanel of the model's activations with columns `id`,`act`", artifact_type=<class 'dcbench.common.artifact.DataPanelArtifact'>, optional=False), 'base_dataset': ArtifactSpec(description='A DataPanel representing the base dataset with columns `id` and `image`.', artifact_type=<class 'dcbench.common.artifact.VisionDatasetArtifact'>, optional=False), 'clip': ArtifactSpec(description="A DataPanel of the image embeddings from OpenAI's CLIP model", artifact_type=<class 'dcbench.common.artifact.DataPanelArtifact'>, optional=False), 'model': ArtifactSpec(description='A trained PyTorch model to audit.', artifact_type=<class 'dcbench.common.artifact.ModelArtifact'>, optional=False), 'test_predictions': ArtifactSpec(description="A DataPanel of the model's predictions with columns `id`,`target`, and `probs.`", artifact_type=<class 'dcbench.common.artifact.DataPanelArtifact'>, optional=False), 'test_slices': ArtifactSpec(description='A DataPanel of the ground truth slice labels with columns `id`, `slices`.', artifact_type=<class 'dcbench.common.artifact.DataPanelArtifact'>, optional=False), 'val_predictions': ArtifactSpec(description="A DataPanel of the model's predictions with columns `id`,`target`, and `probs.`", artifact_type=<class 'dcbench.common.artifact.DataPanelArtifact'>, optional=False)}

attribute_specs: Mapping[str, *AttributeSpec*] = {'alpha': AttributeSpec(description='The alpha parameter for the AUC metric.', attribute_type=<class 'float'>, optional=False), 'dataset': AttributeSpec(description='The name of the dataset being audited.', attribute_type=<class 'str'>, optional=False), 'n_pred_slices': AttributeSpec(description='The number of slice predictions that each slice discovery method can return.', attribute_type=<class 'int'>, optional=False), 'slice_category': AttributeSpec(description='The type of slice .', attribute_type=<class 'str'>, optional=False), 'slice_names': AttributeSpec(description='The names of the slices in the dataset.', attribute_type=<class 'list'>, optional=False), 'target_name': AttributeSpec(description='The name of the target column in the dataset.', attribute_type=<class 'str'>, optional=False)}

task_id: str = 'slice_discovery'

solve(*pred_slices_dp*)

Parameters **pred_slices_dp** (*meerkat.datapanel.DataPanel*) –

Return type *dcbench.tasks.slice_discovery.problem.SliceDiscoverySolution*

evaluate(*solution*)

Parameters **solution** (dcbench.tasks.slice_discovery.problem.SliceDiscoverySolution) –

Return type dict

name: str

summary: str

solution_class: type

Module contents

confusion_sdm(*problem*)

A simple slice discovery method that returns a slice corresponding to each cell of the confusion matrix. For example, for a binary prediction task, this sdm will return 4 slices corresponding to true positives, false positives, true negatives and false negatives.

Parameters **problem** (SliceDiscoveryProblem) – The slice discovery problem.

Returns The predicted slices.

Return type *SliceDiscoverySolution*

domino_sdm(*problem*)

Parameters **problem** (dcbench.tasks.slice_discovery.problem.SliceDiscoveryProblem) –

Return type *dcbench.tasks.slice_discovery.problem.SliceDiscoverySolution*

class SliceDiscoveryProblem(*artifacts*, *attributes=None*, *container_id=None*)

Bases: *dcbench.common.problem.Problem*

Parameters

- **artifacts** (*Mapping[str, Artifact]*) –
- **attributes** (*Mapping[str, Attribute]*) –
- **container_id** (*str*) –

artifact_specs: Mapping[str, *dcbench.common.artifact_container.ArtifactSpec*] =
{'activations': ArtifactSpec(description="A DataPanel of the model's activations with columns `id`,`act`", artifact_type=<class 'dcbench.common.artifact.DataPanelArtifact'>, optional=False), 'base_dataset': ArtifactSpec(description='A DataPanel representing the base dataset with columns `id` and `image`.', artifact_type=<class 'dcbench.common.artifact.VisionDatasetArtifact'>, optional=False), 'clip': ArtifactSpec(description="A DataPanel of the image embeddings from OpenAI's CLIP model", artifact_type=<class 'dcbench.common.artifact.DataPanelArtifact'>, optional=False), 'model': ArtifactSpec(description='A trained PyTorch model to audit.', artifact_type=<class 'dcbench.common.artifact.ModelArtifact'>, optional=False), 'test_predictions': ArtifactSpec(description="A DataPanel of the model's predictions with columns `id`,`target`, and `probs.`", artifact_type=<class 'dcbench.common.artifact.DataPanelArtifact'>, optional=False), 'test_slices': ArtifactSpec(description='A DataPanel of the ground truth slice labels with columns `id`, `slices`.', artifact_type=<class 'dcbench.common.artifact.DataPanelArtifact'>, optional=False), 'val_predictions': ArtifactSpec(description="A DataPanel of the model's predictions with columns `id`,`target`, and `probs.`", artifact_type=<class 'dcbench.common.artifact.DataPanelArtifact'>, optional=False)}

attribute_specs: Mapping[str, *AttributeSpec*] = {'alpha': AttributeSpec(description='The alpha parameter for the AUC metric.', attribute_type=<class 'float'>, optional=False), 'dataset': AttributeSpec(description='The name of the dataset being audited.', attribute_type=<class 'str'>, optional=False), 'n_pred_slices': AttributeSpec(description='The number of slice predictions that each slice discovery method can return.', attribute_type=<class 'int'>, optional=False), 'slice_category': AttributeSpec(description='The type of slice .', attribute_type=<class 'str'>, optional=False), 'slice_names': AttributeSpec(description='The names of the slices in the dataset.', attribute_type=<class 'list'>, optional=False), 'target_name': AttributeSpec(description='The name of the target column in the dataset.', attribute_type=<class 'str'>, optional=False)}

task_id: str = 'slice_discovery'

solve(*pred_slices_dp*)

Parameters **pred_slices_dp** (*meerkat.datapanel.DataPanel*) –

Return type *dcbench.tasks.slice_discovery.problem.SliceDiscoverySolution*

evaluate(*solution*)

Parameters **solution** (dcbench.tasks.slice_discovery.problem.SliceDiscoverySolution) –

Return type dict

name: str

summary: str

solution_class: type

class SliceDiscoverySolution(*artifacts*, *attributes=None*, *container_id=None*)

Bases: *dcbench.common.solution.Solution*

Parameters

- **artifacts** (*Mapping[str, Artifact]*) –
- **attributes** (*Mapping[str, Attribute]*) –
- **container_id** (*str*) –

artifact_specs: Mapping[str, *dcbench.common.artifact_container.ArtifactSpec*] = {'pred_slices': ArtifactSpec(description='A DataPanel of predicted slice labels with columns `id` and `pred_slices`.', artifact_type=<class 'dcbench.common.artifact.DataPanelArtifact'>, optional=False)}

attribute_specs: Mapping[str, *AttributeSpec*] = {'problem_id': AttributeSpec(description='A unique identifier for this problem.', attribute_type=<class 'str'>, optional=False)}

task_id: str = 'slice_discovery'

Module contents

6.2 Submodules

6.3 dcbench.config module

get_config_location()

get_config()

class DCBenchConfig(*local_dir: str = '/home/docs/.dcbench'*, *public_bucket_name: str = 'dcbench'*, *hidden_bucket_name: str = 'dcbench-hidden'*, *celeba_dir: str = '/home/docs/.dcbench/datasets/celeba'*, *imagenet_dir: str = '/home/docs/.dcbench/datasets/imagenet'*)

Bases: object

Parameters

- **local_dir** (*str*) –
- **public_bucket_name** (*str*) –
- **hidden_bucket_name** (*str*) –
- **celeba_dir** (*str*) –
- **imagenet_dir** (*str*) –

Return type None

local_dir: str = '/home/docs/.dcbench'

public_bucket_name: str = 'dcbench'

hidden_bucket_name: str = 'dcbench-hidden'

property public_remote_url

property hidden_remote_url

celeba_dir: str = '/home/docs/.dcbench/datasets/celeba'

imagenet_dir: str = '/home/docs/.dcbench/datasets/imagenet'

6.4 dcbench.constants module

6.5 dcbench.version module

6.6 Module contents

The dcbench module is a collection for benchmarks that test various apsects of data preparation and handling in the context of AI workflows.

class Artifact(*artifact_id*, ***kwargs*)

Bases: abc.ABC

A pointer to a unit of data (e.g. a CSV file) that is stored locally on disk and/or in a remote GCS bucket.

In DCBench, each artifact is identified by a unique artifact ID. The only state that the *Artifact* object must maintain is this ID (self.id). The object does not hold the actual data in memory, making it lightweight.

Artifact is an abstract base class. Different types of artifacts (e.g. a CSV file vs. a PyTorch model) have corresponding subclasses of *Artifact* (e.g. *CSVArtifact*, *ModelArtifact*).

Tip: The vast majority of users should not call the *Artifact* constructor directly. Instead, they should either create a new artifact by calling *from_data()* or load an existing artifact from a YAML file.

The class provides utilities for accessing and managing a unit of data:

- Synchronizing the local and remote copies of a unit of data: *upload()*, *download()*
- Loading the data into memory: *load()*
- Creating new artifacts from in-memory data: *from_data()*
- Serializing the pointer artifact so it can be shared: *to_yaml()*, *from_yaml()*

Parameters **artifact_id** (*str*) – The unique artifact ID.

Return type None

id

The unique artifact ID.

Type str

classmethod from_data(*data*, *artifact_id=None*)

Create a new artifact object from raw data and save the artifact to disk in the local directory specified in the config file at config.local_dir.

Tip: When called on the abstract base class *Artifact*, this method will infer which artifact subclass to use. If you know exactly which artifact class you'd like to use (e.g. *DataPanelArtifact*), you should call this classmethod on that subclass.

Parameters

- **data** (*Union[mk.DataPanel, pd.DataFrame, Model]*) – The raw data that will be saved to disk.
- **artifact_id** (*str, optional*) – . Defaults to None, in which case a UUID will be generated and used.

Returns A new artifact pointing to the **:arg:`data`** that was saved to disk.

Return type *Artifact*

property local_path: str

The local path to the artifact in the local directory specified in the config file at config.local_dir.

property remote_url: str

The URL of the artifact in the remote GCS bucket specified in the config file at config.public_bucket_name.

property is_downloaded: bool

Checks if artifact is downloaded to local directory specified in the config file at config.local_dir.

Returns True if artifact is downloaded, False otherwise.

Return type bool

property is_uploaded: bool

Checks if artifact is uploaded to GCS bucket specified in the config file at config.public_bucket_name.

Returns True if artifact is uploaded, False otherwise.

Return type bool

upload(*force=False*, *bucket=None*)

Uploads artifact to a GCS bucket at self.path, which by default is just the artifact ID with the default extension.

Parameters

- **force** (*bool, optional*) – Force upload even if artifact is already uploaded. Defaults to False.
- **bucket** (*storage.Bucket, optional*) – The GCS bucket to which the artifact is uplioaded. Defaults to None, in which case the artifact is uploaded to the bucket speciried in the config file at config.public_bucket_name.

Return type bool

Returns bool: True if artifact was uploaded, False otherwise.

download(*force=False*)

Downloads artifact from GCS bucket to the local directory specified in the config file at config.local_dir. The relative path to the artifact within that directory is self.path, which by default is just the artifact ID with the default extension.

Parameters **force** (*bool, optional*) – Force download even if artifact is already downloaded. Defaults to False.

Returns True if artifact was downloaded, False otherwise.

Return type bool

> **Warning:** By default, the GCS cache on public urls has a max-age up to an hour. Therefore, when updating an existin artifacts, changes may not be immediately reflected in subsequent downloads.
>
> See here for more details.

DEFAULT_EXT: str = ''

isdir: bool = False

abstract load()

Load the artifact into memory from disk at self.local_path.

Return type *Any*

abstract save(*data*)

Save data to disk at `self.local_path`.

Parameters **data** (*Any*) –

Return type None

static from_yaml(*loader*, *node*)

This function is called by the YAML loader to convert a YAML node into an Artifact object.

It should not be called directly.

Parameters **loader** (*yaml.loader.Loader*) –

static to_yaml(*dumper*, *data*)

This function is called by the YAML dumper to convert an Artifact object into a YAML node.

It should not be called directly.

Parameters

- **dumper** (*yaml.dumper.Dumper*) –
- **data** (dcbench.common.artifact.Artifact) –

class Problem(*artifacts*, *attributes=None*, *container_id=None*)

Bases: *dcbench.common.artifact_container.ArtifactContainer*

A logical collection of :class:`Artifact`s and "attributes" that correspond to a specific problem to be solved.

See the walkthrough section on *Problem* for more information.

Parameters

- **artifacts** (*Mapping[str, Artifact]*) –
- **attributes** (*Mapping[str, Attribute]*) –
- **container_id** (*str*) –

container_type: str = 'problem'

name: str

summary: str

task_id: str

solution_class: type

abstract solve(***kwargs*)

Parameters **kwargs** (*Any*) –

Return type *Solution*

abstract evaluate(*solution*)

Parameters **solution** (Solution) –

Return type *Result*

class Solution(*artifacts*, *attributes=None*, *container_id=None*)

Bases: *dcbench.common.artifact_container.ArtifactContainer*

Parameters

- **artifacts** (*Mapping[str, Artifact]*) –
- **attributes** (*Mapping[str, Attribute]*) –
- **container_id** (*str*) –

container_type: str = 'solution'

class BudgetcleanProblem(*artifacts*, *attributes=None*, *container_id=None*)

Bases: *dcbench.common.problem.Problem*

Parameters

- **artifacts** (*Mapping[str, Artifact]*) –
- **attributes** (*Mapping[str, Attribute]*) –
- **container_id** (*str*) –

artifact_specs: Mapping[str, *dcbench.common.artifact_container.ArtifactSpec*] = {'X_test': ArtifactSpec(description=('Features of the test dataset used to produce the final evaluation score of the model.',), artifact_type=<class 'dcbench.common.artifact.CSVArtifact'>, optional=False), 'X_train_clean': ArtifactSpec(description='Features of the clean training dataset where each dirty value from the dirty dataset is replaced with the correct clean candidate.', artifact_type=<class 'dcbench.common.artifact.CSVArtifact'>, optional=False), 'X_train_dirty': ArtifactSpec(description=('Features of the dirty training dataset which we need to clean. Each dirty cell contains an embedded list of clean candidate values.',), artifact_type=<class 'dcbench.common.artifact.CSVArtifact'>, optional=False), 'X_val': ArtifactSpec(description='Feature of the validtion dataset which can be used to guide the cleaning optimization process.', artifact_type=<class 'dcbench.common.artifact.CSVArtifact'>, optional=False), 'y_test': ArtifactSpec(description='Labels of the test dataset.', artifact_type=<class 'dcbench.common.artifact.CSVArtifact'>, optional=False), 'y_train': ArtifactSpec(description='Labels of the training dataset.', artifact_type=<class 'dcbench.common.artifact.CSVArtifact'>, optional=False), 'y_val': ArtifactSpec(description='Labels of the validation dataset.', artifact_type=<class 'dcbench.common.artifact.CSVArtifact'>, optional=False)}

attribute_specs: Mapping[str, *AttributeSpec*] = {'budget': AttributeSpec(description='TODO', attribute_type=<class 'float'>, optional=False), 'dataset': AttributeSpec(description='TODO', attribute_type=<class 'str'>, optional=False), 'mode': AttributeSpec(description='TODO', attribute_type=<class 'str'>, optional=False), 'model': AttributeSpec(description='TODO', attribute_type=<class 'str'>, optional=False)}

task_id: str = 'budgetclean'

classmethod list()

classmethod from_id(*scenario_id*)

Parameters **scenario_id** (*str*) –

solve(*idx_selected*, ***kwargs*)

Parameters

- **idx_selected** (*Any*) –
- **kwargs** (*Any*) –

Return type *dcbench.common.solution.Solution*

evaluate(*solution*)

Parameters **solution** (dcbench.tasks.budgetclean.problem.BudgetcleanSolution) –

Return type *dcbench.common.result.Result*

class MiniDataProblem(*artifacts*, *attributes=None*, *container_id=None*)

Bases: *dcbench.common.problem.Problem*

Parameters

- **artifacts** (*Mapping[str, Artifact]*) –
- **attributes** (*Mapping[str, Attribute]*) –
- **container_id** (*str*) –

artifact_specs: Mapping[str, *dcbench.common.artifact_container.ArtifactSpec*] = {'test_data': ArtifactSpec(description='A DataPanel of test examples with columns ``id``, ``input``, and ``target``.', artifact_type=<class 'dcbench.common.artifact.DataPanelArtifact'>, optional=False), 'train_data': ArtifactSpec(description='A DataPanel of train examples with columns ``id``, ``input``, and ``target``.', artifact_type=<class 'dcbench.common.artifact.DataPanelArtifact'>, optional=False), 'val_data': ArtifactSpec(description='A DataPanel of validation examples with columns ``id``, ``input``, and ``target``.', artifact_type=<class 'dcbench.common.artifact.DataPanelArtifact'>, optional=False)}

task_id: str = 'minidata'

solve(*idx_selected*, ***kwargs*)

Parameters

- **idx_selected** (*Any*) –
- **kwargs** (*Any*) –

Return type *dcbench.common.solution.Solution*

evaluate(*solution*)

Parameters **solution** (dcbench.common.solution.Solution) –

class SliceDiscoveryProblem(*artifacts*, *attributes=None*, *container_id=None*)

Bases: *dcbench.common.problem.Problem*

Parameters

- **artifacts** (*Mapping[str, Artifact]*) –
- **attributes** (*Mapping[str, Attribute]*) –
- **container_id** (*str*) –

artifact_specs: Mapping[str, *dcbench.common.artifact_container.ArtifactSpec*] = {'activations': ArtifactSpec(description="A DataPanel of the model's activations with columns `id`,`act`", artifact_type=<class 'dcbench.common.artifact.DataPanelArtifact'>, optional=False), 'base_dataset': ArtifactSpec(description='A DataPanel representing the base dataset with columns `id` and `image`.', artifact_type=<class 'dcbench.common.artifact.VisionDatasetArtifact'>, optional=False), 'clip': ArtifactSpec(description="A DataPanel of the image embeddings from OpenAI's CLIP model", artifact_type=<class 'dcbench.common.artifact.DataPanelArtifact'>, optional=False), 'model': ArtifactSpec(description='A trained PyTorch model to audit.', artifact_type=<class 'dcbench.common.artifact.ModelArtifact'>, optional=False), 'test_predictions': ArtifactSpec(description="A DataPanel of the model's predictions with columns `id`,`target`, and `probs.`", artifact_type=<class 'dcbench.common.artifact.DataPanelArtifact'>, optional=False), 'test_slices': ArtifactSpec(description='A DataPanel of the ground truth slice labels with columns `id`, `slices`.', artifact_type=<class 'dcbench.common.artifact.DataPanelArtifact'>, optional=False), 'val_predictions': ArtifactSpec(description="A DataPanel of the model's predictions with columns `id`,`target`, and `probs.`", artifact_type=<class 'dcbench.common.artifact.DataPanelArtifact'>, optional=False)}

attribute_specs: Mapping[str, *AttributeSpec*] = {'alpha': AttributeSpec(description='The alpha parameter for the AUC metric.', attribute_type=<class 'float'>, optional=False), 'dataset': AttributeSpec(description='The name of the dataset being audited.', attribute_type=<class 'str'>, optional=False), 'n_pred_slices': AttributeSpec(description='The number of slice predictions that each slice discovery method can return.', attribute_type=<class 'int'>, optional=False), 'slice_category': AttributeSpec(description='The type of slice .', attribute_type=<class 'str'>, optional=False), 'slice_names': AttributeSpec(description='The names of the slices in the dataset.', attribute_type=<class 'list'>, optional=False), 'target_name': AttributeSpec(description='The name of the target column in the dataset.', attribute_type=<class 'str'>, optional=False)}

task_id: str = 'slice_discovery'

solve(*pred_slices_dp*)

Parameters **pred_slices_dp** (*meerkat.datapanel.DataPanel*) –

Return type *dcbench.tasks.slice_discovery.problem.SliceDiscoverySolution*

evaluate(*solution*)

Parameters **solution** (dcbench.tasks.slice_discovery.problem.SliceDiscoverySolution) –

Return type dict

class BudgetcleanSolution(*artifacts*, *attributes=None*, *container_id=None*)

Bases: *dcbench.common.solution.Solution*

Parameters

- **artifacts** (*Mapping[str, Artifact]*) –
- **attributes** (*Mapping[str, Attribute]*) –
- **container_id** (*str*) –

artifact_specs: Mapping[str, *dcbench.common.artifact_container.ArtifactSpec*] = {'idx_selected': ArtifactSpec(description='', artifact_type=<class 'dcbench.common.artifact.CSVArtifact'>, optional=False)}

class MiniDataSolution(*artifacts*, *attributes=None*, *container_id=None*)

Bases: *dcbench.common.solution.Solution*

Parameters

- **artifacts** (*Mapping[str, Artifact]*) –
- **attributes** (*Mapping[str, Attribute]*) –
- **container_id** (*str*) –

artifact_specs: Mapping[str, *dcbench.common.artifact_container.ArtifactSpec*] = {'train_ids': ArtifactSpec(description='A list of train example ids from the ``id`` column of ``train_data``.', artifact_type=<class 'dcbench.common.artifact.YAMLArtifact'>, optional=False)}

task_id: str = 'minidata'

classmethod from_ids(*train_ids*, *problem_id*)

Parameters

- **train_ids** (*Sequence[str]*) –
- **problem_id** (*str*) –

class SliceDiscoverySolution(*artifacts*, *attributes=None*, *container_id=None*)

Bases: *dcbench.common.solution.Solution*

Parameters

- **artifacts** (*Mapping[str, Artifact]*) –
- **attributes** (*Mapping[str, Attribute]*) –
- **container_id** (*str*) –

artifact_specs: Mapping[str, *dcbench.common.artifact_container.ArtifactSpec*] = {'pred_slices': ArtifactSpec(description='A DataPanel of predicted slice labels with columns `id` and `pred_slices`.', artifact_type=<class 'dcbench.common.artifact.DataPanelArtifact'>, optional=False)}

attribute_specs: Mapping[str, *AttributeSpec*] = {'problem_id': AttributeSpec(description='A unique identifier for this problem.', attribute_type=<class 'str'>, optional=False)}

task_id: str = 'slice_discovery'

class Task(*task_id*, *name*, *summary*, *problem_class*, *solution_class*, *baselines=Empty DataFrame Columns: [] Index: []*)

Bases: *dcbench.common.table.RowMixin*

Task(task_id: str, name: str, summary: str, problem_class: type, solution_class: type, baselines: dcbench.common.table.Table = Empty DataFrame Columns: [] Index: [])

Parameters

- **task_id** (*str*) –

- **name** (*str*) –
- **summary** (*str*) –
- **problem_class** (*type*) –
- **solution_class** (*type*) –
- **baselines** (dcbench.common.table.Table) –

Return type None

task_id: str

name: str

summary: str

problem_class: type

solution_class: type

baselines: *dcbench.common.table.Table* = Empty DataFrame Columns: [] Index: []

property problems_path

property local_problems_path

property remote_problems_url

write_problems(*containers*, *append=True*)

Parameters

- **containers** (*List[*dcbench.common.artifact_container.ArtifactContainer*]*) –
- **append** (*bool*) –

upload_problems(*include_artifacts=False*, *force=True*)

Uploads the problems to the remote storage.

Parameters

- **include_artifacts** (*bool*) – If True, also uploads the artifacts of the problems.
- **force** (*bool*) – If True, if the problem overwrites the remote problems. Defaults to True. .. warning:

```
It is somewhat dangerous to set `force=False`, as this could lead
to remote and local problems being out of sync.
```

download_problems(*include_artifacts=False*)

Parameters **include_artifacts** (*bool*) –

property problems

class ModelArtifact(*artifact_id*, ***kwargs*)

Bases: *dcbench.common.artifact.Artifact*

Parameters **artifact_id** (*str*) –

Return type None

DEFAULT_EXT: str = 'pt'

load()

Load the artifact into memory from disk at `self.local_path`.

Return type *dcbench.common.modeling.Model*

save(*data*)

Save data to disk at `self.local_path`.

Parameters **data** (`dcbench.common.modeling.Model`) –

Return type None

class YAMLArtifact(*artifact_id*, ***kwargs*)

Bases: *dcbench.common.artifact.Artifact*

Parameters **artifact_id** (*str*) –

Return type None

DEFAULT_EXT: str = 'yaml'

load()

Load the artifact into memory from disk at `self.local_path`.

Return type *Any*

save(*data*)

Save data to disk at `self.local_path`.

Parameters **data** (*Any*) –

Return type None

class DataPanelArtifact(*artifact_id*, ***kwargs*)

Bases: *dcbench.common.artifact.Artifact*

Parameters **artifact_id** (*str*) –

Return type None

DEFAULT_EXT: str = 'mk'

isdir: bool = True

load()

Load the artifact into memory from disk at `self.local_path`.

Return type pandas.core.frame.DataFrame

save(*data*)

Save data to disk at `self.local_path`.

Parameters **data** (*meerkat.datapanel.DataPanel*) –

Return type None

class VisionDatasetArtifact(*artifact_id*, ***kwargs*)

Bases: *dcbench.common.artifact.DataPanelArtifact*

Parameters **artifact_id** (*str*) –

Return type None

DEFAULT_EXT: str = 'mk'

isdir: bool = True

COLUMN_SUBSETS = {'celeba': ['id', 'image', 'identity', 'split'], 'imagenet': ['id', 'image', 'name', 'synset']}

classmethod from_name(*name*)

Parameters **name** (*str*) –

download(*force=False*)

Downloads artifact from GCS bucket to the local directory specified in the config file at `config.local_dir`. The relative path to the artifact within that directory is `self.path`, which by default is just the artifact ID with the default extension.

Parameters **force** (*bool, optional*) – Force download even if artifact is already downloaded. Defaults to False.

Returns True if artifact was downloaded, False otherwise.

Return type bool

Warning: By default, the GCS cache on public urls has a max-age up to an hour. Therefore, when updating an existin artifacts, changes may not be immediately reflected in subsequent downloads.

See here for more details.

class CSVArtifact(*artifact_id*, ***kwargs*)

Bases: *dcbench.common.artifact.Artifact*

Parameters **artifact_id** (*str*) –

Return type None

DEFAULT_EXT: str = 'csv'

load()

Load the artifact into memory from disk at `self.local_path`.

Return type pandas.core.frame.DataFrame

save(*data*)

Save data to disk at `self.local_path`.

Parameters **data** (*pandas.core.frame.DataFrame*) –

Return type None

PYTHON MODULE INDEX

d

INDEX

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W

Y