dbcollection Documentation

Release 0.2.6

M. Farrajota

Jul 27, 2018

Introduction

dbcollection is a library for downloading/parsing/managing datasets via simple methods. It was built from the ground up to be cross-platform (**Windows**, **Linux**, **MacOS**) and cross-language (**Python**, **Lua**, **Matlab**, etc.). This is achieved by using the popular `HDF5` file format to store (meta)data of manually parsed datasets and the power of Python for scripting. By doing so, this library can target any platform that supports Python and any language that has bindings for `HDF5`.

This package allows to easily manage and load datasets by using `HDF5` files to store metadata. By storing all the necessary metadata to disk, managing either big or small datasets has an equal or very similar impact on the system's resource usage. Also, once a dataset is setup, it is setup forever! This means users can reuse any previously set dataset as many times as needed without having to set it each time they are used.

dbcollection allows users to focus on more important tasks like prototyping new models or testing them in different datasets without having to incur the loss of spending time managing datasets or creating/modyfing scripts to load/fetch data by taking advantage of the work of the community that shared these resources.

This library contains a (growing!) list of popular datasets in computer science for many fields like **object detection**, **classification**, **human body joint detection**, **captioning**, etc. It provides a great way to quickly start hacking on a number of different tasks by skipping the boring task of learning how to set/parse datasets (and sometimes dealing with human errors in annotated data).

Also, since it has been developed with community in mind, this library should encourage users to write and share their scripts for downloading/parsing other datasets with the community.

The project's code is publicly available on GitHub and is licensed under the *MIT license*.

CHAPTER 1

Main features

Here are some of key features dbcollection provides:

- Simple API to load/download/setup/manage datasets.
- Simple API to fetch data from a dataset.
- Store and pull data from disk or from memory, you choose!
- Datasets only need to be set/processed once, so next time you use it it will load instantly!
- Cross-platform (**Windows**, **Linux**, **MacOs**).
- Cross-language (**Python**, **Lua/Torch7**, **Matlab**).
- Easily extensible to other languages that support `HDF5` files format.
- Concurrent/parallel data access thanks to `HDF5`.
- Contains a diverse (and growing!) list of popular datasets for machine-, deep-learning tasks (*object detection*, *action recognition*, *human pose estimation*, etc.)

1.1 Contents:

1.1.1 About dbcollection

What is dbcollection

`dbcollection` is a package written in **Python** that contains methods for downloading / managing datasets and to fetch data from them via simple API function calls. This package was developed with ease of use in mind and seamless reuse of data between projects. It provides to researchers a quick and easy way to deal with the headache of setting up data for training / processing algorithms when working on a new project.

Also, it provides a cross-platform, cross-language framework to share datasets between users such that anyone can quickly load/fetch data with minimal effort spent, thus taking advantage of the community effort to build and share solutions for common problems.

In summary, the main goal of this project is to help users save time and effort when developing/deploying/sharing code.

Why does this project exist

This project started during my PhD where when learning/experimenting new stuff in machine/deep learning was a constant struggle between setting up new datasets and manually parsing some of the quirks they might had before even doing the actual research I was meant to do.

Also, when moving between projects/frameworks this usually meant that I either had to reimplement it in another language or had to re-write the same boilerplate code once more (and store it in a cache file to avoid constant computations at startup). This usually meant time lost doing the same thing over and over again, hard drives filled with 'junk' and, ultimately, it bored me.

By the end of 2016 I've started writting some initial code in Lua to deal with downloading data from urls, extracting it from a myriad of compression formats like .zip, .tar, etc., process the metadata associated with the dataset in the right format and storing the resulting data into disk as cache in order to avoid re-computing it every time I'd launch a script.

This worked fairly well and helped to avoid recomputing datasets over and over again and it meant I only needed to do it once. Afterwards, I've discovered how neat `HDF5` files were and all the features they had, and this provided the missing piece I needed for building a cross-language, cross-platform metadata storing/fetching framework for general purpose data management.

What problems does it try to solve

This project tries to solve some of the problems I've identified during my research when trying out new datasets. These are some of the problems `dbcollection` tries to solve:

- Downloading, extracting and parsing different datasets without prior knowledge of their inner workings is sometimes error prone.
- Constantly writting the same boilerplate code or adapting existing one for new projects is a hassle.
- Disk space gets littered with data files everywhere as you work on different projects with no centralized storage.
- Wasted time and computer resources (mostly memory) when loading large datasets.
- Many datasets use `.json` files to distribute their (meta)data, which is fine for smaller datasets, but they are not an efficient solution to store large amounts of (meta)data for large datasets.
- Usually trying new datasets means that you will have to spend a significant portion of your time learning how to use it so you can load/parse it. And good luck if those datasets are distributed using some complicated, in house format that can only be extracted by using a specific toolbox (e.g., Caltech Pedestrian).
- Having to learn a toolboxe (and languages maybe) to fetch data for a given dataset that you just want to try it out before doing anything serious is not viable at all or, at best, a tedious task.
- If you start using a new language you'll probably write the same scripts to load/parse your datasets.

This project aims to solve most of these problems so anyone using it won't have to deal with the nightmare that is trying a new dataset. By using `dbcollection`, you can immediately start to use any dataset of your choice without much of the pain that involves using a new dataset, so you can quickly train/test your new, shiny algorithm / model in simple or complex scenarios without having to waste precious time.

How does it work

When loading a dataset, the user uses usually a single API method to fetch a data loader object. The method looks for an entry of the dataset in a cache file stored in your home directory for a matching entry. If the dataset does not exist

in cache, the system proceeds to do the following steps to download and process a dataset:

1. First, the data files are located in a pre-defined path or in a path provided by the user. If no data files are found or no one matches the specific files the package is looking for, it proceeds to download the data files to disk by using pre-defined urls with the source files. Then, the downloaded files are extracted into the same path where the files are located;
2. Next, the data files are processed and annotations are parsed and converted into numpy arrays which are stored into a `HDF5` file;
3. Then, the dataset's information is added to cache.

After these steps are done, a data loader object is generated based on the metadata `HDF5` file that contains the dataset's metadata information. This object contains several methods for querying and fetching data from the metadata file.

Since the processed metadata is stored in a `HDF5` file, it provides some key features like:

- portable across languages and operating systems;
- fast access to data;
- allows concurrent reads of the same file;
- compression;
- data can be stored in a nested way (like a folder tree)

Also, by using the `HDF5` format, it is simple to deploy an API written in other languages that support the `HDF5` format to interface with the stored metadata. For more information about this file format see the official HDF5 documentation.

Note: The **dbcollection** package creates a folder in your home directory named `~/dbcollection/` where all the metadata files are organized and stored. The contents of this folder is tracked by a `.json` file also stored in your home directory in `~/dbcollection.json` which contains all information/configurations about the stored datasets.

1.1.2 Install guide

Installing `dbcollection` is quite simple. You can do it in several ways, but we advise users to install this package via `pip`. If you want the latest features or bug fixes then you can install this package from the source.

This package is available for **Windows**, **Linux** and **MacOs**, and **Python** `>=2.7` and `>=3.5` are supported.

Note: Other Python versions bellow <3.5 may work but officially we only support these ones.

Install dbcollection via pip

Installing `dbcollection` using pip is simple. For that purpose, simply do the following command:

```
$ pip install dbcollection
```

This will install the latest version of the `dbcollection` package on your system.

Install dbcollection from source

To install the `dbcollection` package from source, you need to do the following steps:

1. Clone the repo to your hard drive:

```
$ git clone --recursive https://github.com/dbcollection/dbcollection
```

2. `cd` to the dbcollection folder and do the command:

```
$ cd dbcollection/
$ python setup.py install
```

and voilá, the package should now be installed on your system.

Other languages

There are some wrappers written for use with other languages available if you want to use this package. For now, these are the supported wrappers that emulate the functionality of this package:

- Lua/Torch7
- Matlab

1.1.3 Getting started

In this chapter you'll be introduced to the basic usage of **dbcollection** and its most commonly used features. More advanced methods and features will be addressed in the next chapters, but for now lets take a quick look at how we can start using this package.

Before you start using this package, there are a few things you should be aware in order to improve your experience when managing datasets. First, *setting up the paths* to store data/metada files is important in order to keep things organized and in the same place. Next, you should also take a look at the *available datasets* data file's size in disk in order to check if you have enough disk space to store them.

After this is done, take a look at the *contents of the package* before proceeding to learning how to use the API of this module in the *basic usage* section.

Setting up paths

dbcollection uses a json cache file stored in your home directory (`~/dbcollection.json`) to log what datasets have been loaded, what tasks have been used and where data is stored. When first installing this package, the default paths where downloaded data files and .h5 metadata files are stored are defined in the cache file to store data in `~\dbcollection\`.

You can chose to store your data files or metadata files into separate dirs on a disk or on separate disks by modifying the paths of these two fields: `default_cache_dir` and `default_download_dir`. You can either manually change the paths in the `~\dbcollection.json` file under the `info` key or you can do it by using **dbcollection**. To use the latter, you can do the following:

```
>>> # import the package
>>> import dbcollection as dbc
>>>
>>> # directly access the fields and assign new paths
>>> dbc.cache.cache_dir = 'new/cache/path'
>>> dbc.cache.download_dir = 'new/download_data/path'
```

This will change the paths where the metadata files (`cache_dir`) and the downloaded files (`download_dir`) are stored in disk. In case you want to reset the paths to the original defaults you can simply do:

```
>>> # reset the metadata cache dir to the default path
>>> dbc.cache.reset_cache_dir()
>>>
>>> # reset the data download dir to the default path
>>> dbc.cache.reset_download_dir()
```

With this you should be able to easily locate where data/metadata files are being stored in disk without having to manually specify a path everytime you setup a new dataset.

Contents of the package

The **dbcollection** package comes with several API methods and other features to help managing datasets with very little overhead. These include:

- API methods for managing datasets and fetching data;
- API for managing/querying the cache file;
- A list of available datasets;
- utility methods for parsing loaded data, load different types of data files, downloading urls, string convertions, or constructing a tree of files in a dir.

The following sections will describe in more detail these features one by one and how to use them. Next, comes a brief tutorial on how to use this package to quickly start hacking new stuff with it.

Basic usage

To use this package you first need to import it.

```
>>> import dbcollection as dbc
```

Then, to load a dataset, all you need to do is call the `load()` method with the name of the dataset you want to load. For example, lets load the `mnist` dataset.

```
>>> mnist = dbc.load('mnist')
```

This returns a *DataLoader* object which contains all necessary methods to fetch data for this dataset.

Notice the name of the dataset is all lower case. The name of a dataset must be the exact one, and to have the right one you should check the list of available datasets to see the correct name. To do this you can use the `info_datasets()` method to list all available datasets names and tasks.

```
>>> dbc.info_datasets()
---------------------------------------
Available datasets in cache for load
---------------------------------------
- mnist  ['classification']

-----------------------------------
Available datasets for download
-----------------------------------
- caltech_pedestrian  ['detection', 'detection_10x', 'detection_30x']
- cifar10  ['classification']
- cifar100  ['classification']
- coco  ['caption_2015', 'caption_2016', 'detection_2015', 'detection_2016',
↪'keypoints_2016']
```

(continues on next page)

(continued from previous page)

```
- flic  ['keypoints']
- ilsvrc2012  ['classification', 'raw256']
- inria_pedestrian  ['detection']
- leeds_sports_pose  ['keypoints', 'keypoints_original']
- leeds_sports_pose_extended  ['keypoints']
- mnist  ['classification']
- mpii_pose  ['keypoints', 'keypoints_full']
- pascal_voc_2007  ['detection']
- pascal_voc_2012  ['detection']
- ucf_101  ['recognition']
- ucf_sports  ['recognition']
```

This returns two lists, one for used datasets on your system, and the other is a list of all available datasets to download and their respective tasks for processing. Please notice that the `mnist` dataset we've just loaded has the `classification` task setup. This is due to this task being the default task that is selected if no task is specified at loading time. Also, a list of all available tasks is displayed in the **Available datasets for download** list.

Note: For more information about the available datasets and tasks see *here*.

Returning to the previous example about loading the `mnist` dataset, the resulting data loading object contains several methods to fetch data from the metadata file, as well as other information like the task name, the set splits, where the data files are located, etc.

```
>>> mnist.
mnist.data_dir          mnist.hdf5_filepath     mnist.object_field_id(  mnist.size(
mnist.db_name           mnist.info(             mnist.object_fields     mnist.task
mnist.get(              mnist.list(             mnist.root_path         mnist.test
mnist.hdf5_file         mnist.object(           mnist.sets              mnist.train
```

The API methods for fetching and querying the metadata file are quite handy. For example, to see how the metadata file is structured and what data fields it contains, you simply have to use the `info()` method in order to have an idea of how data is organized.

```
>>> mnist.info()

> Set: test
- classes,         shape = (10, 2),          dtype = uint8
- images,          shape = (10000, 28, 28),  dtype = uint8,  (in 'object_ids',␣
↪position = 0)
- labels,          shape = (10000,),         dtype = uint8,  (in 'object_ids',␣
↪position = 1)
- object_fields,   shape = (2, 7),           dtype = uint8
- object_ids,      shape = (10000, 2),       dtype = uint8

(Pre-ordered lists)
- list_images_per_class,  shape = (10, 1135),  dtype = int32

> Set: train
- classes,         shape = (10, 2),          dtype = uint8
- images,          shape = (60000, 28, 28),  dtype = uint8,  (in 'object_ids',␣
↪position = 0)
- labels,          shape = (60000,),         dtype = uint8,  (in 'object_ids',␣
↪position = 1)
- object_fields,   shape = (2, 7),           dtype = uint8
```

(continues on next page)

(continued from previous page)

```
- object_ids,      shape = (60000, 2),        dtype = uint8

(Pre-ordered lists)
- list_images_per_class,  shape = (10, 6742),  dtype = int32
```

This way, you get a general idea of how the dataset's data is split and what fields compose each set, and also their type or shape. This method and its output are described in more detail in the *fetching data* section.

To fetch data, you can use two methods to retrieve a chunk of data by using the get() and object() methods. These two are complementary to one another, but when you need to fetch data from a single field you use the `get()` method, and when you need to retrieve data from a set of fields you'll use the `object()` method. For example, lets retrieve the first 10 images from the training set of `mnist`:

```
>>> imgs = mnist.get('train', 'images', range(10))
>>> imgs.shape
(10, 28, 28)
```

Fetching data is simple! If can retrieve this same data in two other ways. The first way is to grab the train set data altogether and then using the same method:

```
>>> train = mnist.train  # get a data loader object of the train set
>>> train
SetLoader: set<train>, len<60000>
>>> imgs = train.get('images', range(10))
>>> imgs.shape
(10, 28, 28)
```

The difference here is that you can grab the train set as a separate object and do all your operations with it. Also, here you don't have to explicitly define the set to fetch data from, but you still have to define the field name.

The second way you can fetch data is by targeting the actual field you want to retrieve data from. Just like the previous examples, we can grab the first 10 images from the `mnist` train set in the following ways:

```
>>> # First way
>>> images = mnist.train.images  # get a data loader object of the images field
>>> images
FieldLoader: <HDF5 dataset "images": shape (60000, 28, 28), type "|u1">
>>> images.get(range(10))
>>> imgs.shape
(10, 28, 28)

>>> # Second way
>>> imgs = images[0:10]
>>> imgs.shape
(10, 28, 28)
```

For single fields you can do array slicing operations likewith numpy arrays. All of these operations convey the same results, and it is up to the user to decide which one fits his/hers needs best.

We've see so far how fetching data from single fields is done, but most cases you want to grab sets of related data fields like, for example, the image and label. This information is conveyd by two key fields in the metadata files that relate different fields ids with each other: the `object_fields` and `object_ids` fields. The `object_ids` field is a list of indexes of fields defined in the `object_fields` field. So, to get the right label for a given image you just need to collect the ids of each field and then fetch their data. To do this, we'll use the `object()` method to grab the ids of the fields for the *100th* item:

```
>>> # Grab the ids of the image and label fields of the 100th element
>>> ids = mnist.object('train', 99)
>>> ids
array([99,  1], dtype=uint8)
>>> # Fetch the image data
>>> img = mnist.get('train', 'images', ids[0])
>>> img.shape
(28, 28)
>>> # Fetch the label data
>>> label = mnist.get('train', 'labels', ids[1])
>>> label
0
```

Another way you can do this to get the same data, without having to manually fetch data from every field, is to use the `convert_to_value` argument in `object()` and set it to `True`. This will automatically fetch the data of all fields and return them in a list.

```
>>> # Grab the ids of the image and label fields of the 100th element
>>> (img, label) = mnist.object('train', 99, convert_to_value=True)
>>> img.shape
(28, 28)
>>> label
0
```

As you can see, this can be quite handy when multiple fields compose an object element. You'll mostly use a combination of `get()` and `object()` to fetch data from a dataset in your code, and this is all you'll probably need.

The two last methods I would like to point to are list() and size(). The `list()` method lists all data fields available for each set and the `size()` method returns the size of a field. The purpose of these methods is to mearly serve as information source for the user.

With this information, you should be able to have a sufficient understanding of how `dbcollection` works and how to take advantage of its features. In the tutorial we'll dive deeper on more advanced features and use cases that can help you get more out of this module.

1.1.4 Managing datasets

In the previous Chapter *Getting started*, you've seen basic operations on how to load a dataset, check what available datasets are in **dbcollection** and to fetch data like image tensors and labels.

In the following sections, we'll explore in more detail how to use the **dbcollection** dataset management API to deal with loading, adding or removing datasets with some simple commands.

Furthermore, other functionality like managing the cache file, finding which datasets are available for download or which tasks does it have will be covered as well.

Note: This section covers all the necessary steps / ways for you to effectively use this package for dealing with datasets.

Main operations

You have a set of operations at your disposal using the **dbcollection's** API to manage your datasets. These operations are:

- *Downloading data files from online resources*

- *Parsing annotations / metadata of datasets*
- *Loading a dataset as a data loader object*
- *Adding a custom dataset*
- *Removing a dataset or task*
- *Modifying the cache*
- *Querying the cache*
- *Displaying cache information*
- *Displaying information about available datasets*

These operations allows users to do (pretty much) anything needed for managing data files.

A word of warning for **new users**: you must take into consideration the implications of some of these operations like removing datasets data or modyfing the cache may result in permanent data loss.

You should check the section of *Best practices* if you are unclear on how to proceed in some cases in order to avoid undesired results.

Downloading data files from online resources

> **Warning:** This section deals with manually downloading source data files of datasets.
>
> For most users this information may not be necessary or relevant, and you can skip this section altogether and move to the one which deals with *loading datasets*.
>
> This is because the `load()` method automatically downloads or processes any dataset that has not been previously setup, and it is not required to manually download data files in order to load a dataset.

One use of **dbcollection** is to download data files from online sources. This removes the need to search where to get the data files from and to locate which specific resources are required.

In some cases, it is not a very challenging thing to do, but in others it can be a daunting task. By having the necessary resources defined and ready for use, you can save quite some time when dealing with this process.

To download a dataset you simply need to use the download() method from the *core API* methods and provide the name of the dataset you want to download. For example, lets download the `cifar10` dataset's data files:

```
>>> dbc.download('cifar10')
```

The data files will be stored in a folder named `cifar10/` in a pre-defined directory in disk defined by `dbc.cache.download_dir`. To change this directory you can simpy assign a new path to it: `dbc.cache.download_dir = 'my/new/path/'`.

If you want to use a different directory to store the data files you can use the `data_dir` input argument to specify the path of the data directory you want to store your files:

```
>>> dbc.download('cifar10', data_dir='some/other/path/')
```

This will also create a folder with the same name as the dataset. This is important because **dbcollection** searches for this dir names when loading the data files. If the names don't match then it proceeds to download the source files.

After all files have been downloaded, by default, they are extracted into the same folder where they have been stored.

Most source files are compressed for distribution. The `download()` method allows to extract these compressed files to disk without you having to manually do it yourself. If the source data files are all what you want to retrieve, then set the `extract_data` input argument to `False`:

```
>>> dbc.download('cifar10', extract_data=False)
```

Note: This package uses the patool module for file extraction which supports most data compression formats like TAR, ZIP or RAR.

An important aspect to mention about using this method is that, when using it to download data files of a dataset, it automatically registers in cache where the files are located for that dataset.

So, next time you want to load that dataset, you don't need to explicitly tell where the data is located in disk (if the files still exist of course).

Parsing annotations / metadata of datasets

Warning: This section deals with manually parsing annotations of datasets.

For most users this information is not relevant and you can skip this section altogether and move to the next one which deals with *loading datasets*.

This is because the `load()` method automatically downloads or processes any dataset that has not been previously setup, and it is not required to manually parse annotations of tasks of datasets in order to load a dataset.

Arguably, one of the most important functionalities of **dbcollection** is automatically processing data annotations. It is well known that manually parsing data files + annotations of different datasets is no fun.

Moreover, it is time consuming, annoying and repetitive. Also, it usually results in disks littered with various cache files which are used to store portions of the annotations accross multiple directories for some specific tasks.

dbcollection provides a way to deal with these issues. Hand-crafted scripts were developed to parse data annotations of specific tasks of datasets for you. These annotations are stored in a common format and in a single place on your disk that you can easily track.

Note: Not all annotations are necessary for day to day use, so only the most useful ones are stored.

If you happen to need an annotation that is not available in our scripts for any particular reason, please feel free to fill an issue on GitHub describing what annotation you need, why and for what task + dataset or, better yet, *contribute with a pull request*.

Processing metadata of dataset's annotations is done by using the process() method. Continuing with the previous section example, lets process the metadata files for the `cifar10` dataset:

```
>>> dbc.process('cifar10')
```

The method will process the data annotations of this dataset and stores the resulting metadata into an `HDF5` file stored in disk. By default, all metadata files are stored in your home directory in `~/dbcollection/<dataset>/<task>.h5`.

This directory is used to centralize all metadata files in disk and its path can be accessed via `dbc.cache.cache_dir`. To change the default path, simply assign a new path to it: `dbc.cache.cache_dir = new/cache/path/`.

Many datasets have many tasks to choose from and these can be listed by the `info_datasets()` method described in *this section*. To specify which task to process, we must use the `task` input argument and assign it a task name:

```
>>> dbc.process('cifar10', task='classification')
```

This processes the annotations of the `classification` task and registers them to cache.

We must point out that this example is not the most illustrative of them all because `cifar10` only has one task which is `classification`.

Every dataset has a default task and it is not required to explicitly define one. But, if you want to select a different task, you will need to provide a valid task name for processing.

Note: The `process()` method requires that the data files of a dataset have been previously downloaded and registered in the cache.

If you have not done this, please see the previous section which explains how to download data files of a dataset or see the section further on this page about manually configuring the cache if you happen to have the necessary data files in disk but on a different folder.

The next section covers the `load()` method which deals with loading datasets as data loader objects for extracting (meta)data.

Loading a dataset as a data loader object

Loading a dataset's metadata is quite simple. For that, we need to use the load() method and select a dataset to import. Lets load the `cifar10` dataset:

```
>>> cifar10 = dbc.load('cifar10')
```

To load a dataset we just need to specify its name and in return we'll get a `DataLoader` object.

```
>>> cifar10
DataLoader: "cifar10" (classification task)
```

This object contains methods to retrieve data from and attributes with information about the dataset's name, task, metadata file path, sets and other kinds of information.

```
>>> cifar10.
cifar10.data_dir          cifar10.list(             cifar10.size(
cifar10.db_name           cifar10.object(           cifar10.task
cifar10.get(              cifar10.object_field_id(  cifar10.test
cifar10.hdf5_file         cifar10.object_fields     cifar10.train
cifar10.hdf5_filepath     cifar10.root_path
cifar10.info(             cifar10.sets
```

With this `DataLoader` object, fetching data like labels, bounding boxes, images filenames, etc., is trivial.

Note: A more detailed look on how (meta)data is stored and retrieved will be described in the Chapter *Fetching data*.

When loading a dataset for the first time that is not available in disk, the `load()` method will download the dataset's data files and parse the annotations into an `HDF5` metadata file. The data and metadata files will be stored in the dirs defined in `dbc.cache.download_dir` and `dbc.cache.cache_dir`, respectively. These dirs' paths can me modified by assigning new values to them.

In cases where ou might want to download your data files into a separate directory or you have the dataset's data files available in another directory, you can use the `data_dir` input argument to specify which path to use.

Lets load the `cifar100` dataset, but this time lets store the data files into a new directory:

```
>>> cifar100 = dbc.load('cifar100', data_dir='some/new/path/')
```

If you haven't downloaded/loaded it yet, it will proceed to download and extract the files to the `some/new/path/` path provided in `data_dir` and process the default task for this dataset.

When loading/downloading a dataset for the first time, the dataset's information about where the data and the cache files are stored is registered in the `~/dbcollection.json` cache file, so that next time you load a specific dataset you'll only need to specify its name without having to provide the path where the data files are stored.

Now, lets talk about tasks of datasets. In the previous examples we've dimissed any references about which task to load in order to keep the examples simple. For some simple datasets, often there's only one task available to use, but for cases like `coco` for example, there are many different tasks that have different types of fields. For these cases, it is important to specify which task to use.

To load a specific task, you need to use the `task` input argument and assign a name of the task you want to load. If no task is specified, the default task `task="default"` is used. For the previous example, we could have done this by specifying the task name as `classification`:

```
>>> cifar100_cls = dbc.load('cifar100', task='classification')
```

The returned object contains the information of the selected task in its `__str__()` method:

```
>>> cifar100_cls
DataLoader: "cifar10" (classification task)
```

The `load()` method is probably the only method you'll ever need to use to load datasets. The Chapter *Fetching data* continues where this section stopped about dealing with (meta)data. In the following sections deal with adding and removing datasets to / from the cache and what their uses are.

Note: The *Best practices* section at the end of this page provides some tips about how to setup **dbcollection** in your system in order to never have the need to look at any other method besides `load()` for dealing with datasets.

Adding a custom dataset

The add() method is used to add custom datasets to the cache.

To add a custom dataset, you need to provide information about the `name`, `task`, `data_dir` and `file_path`. Optionally, you can add a list of `keywords` to categorize the dataset.

Lets add a custom dataset to the available datasets list for load in cache:

```
>>> dbc.add(name='new_dataset',
            task='new_task',
            data_dir='some/path/data',
            file_path='other/path/metadata/file.h5',
            keywords=('image_processing', 'classification'))
```

When loading this dataset, the `name` and `task` args are required in order to select the dataset. Then, the `HDF5` file containing the metadata is loaded as a `DataLoader` object and all data files are available in the directory path provided by `data_dir`.

This method can also be used to add additional tasks to existing datasets. For example, if we wanted to enhance the `cifar10` dataset with an extra task which contains custom metadata, you could do something like the following:

```
>>> dbc.add(name='cifar10',
            task='custom_classification',
            data_dir='default/path/dir/cifar10',
            file_path='path/to/new/metadata/file.h5')
```

You can also use the `add()` method to assign the path of the data files for a dataset. This would tell the `load()` method to search for the data files in a specific path before attempting to execute the code path to download data files.

```
>>> dbc.add(name='mnist',
            data_dir='default/path/dir/cifar10',
            task='',  # skips adding the task name
            file_path='')  # skips adding the file path for the task
```

Removing a dataset or task

Removing datasets or tasks is pretty simple. With remove(), all you need to do to remove a dataset is to provide the name of the dataset you want to remove from the cache.

```
>>> dbc.remove('cifar100')
```

If you just want to delete a task from a dataset you need to specify both the name of the dataset you want to remove from and the name of the task:

```
>>> dbc.remove('cifar100', 'classification')
```

This removes the `classification` task entry from the cache registry and it also deletes the metadata file associated to it.

However, this will not remove the data files from disk. For that, you must use the `delete_data` argument and set it to `True`.

```
>>> dbc.remove('cifar100', 'classification', delete_data=True)
```

Warning: This will permanently remove the data files stored in the `data_dir` path associated with the dataset in the cache.

Modifying the cache

There are several ways to manage / modify the cache contents in `~/dbcollection.json` or to delete / reset the cache file.

The config_cache() allows users to do the following operations:

- Change values of fields;
- Delete the cache file / directory;
- Delete the `~/dbcollection.json` cache file;
- Reset the cache file.

Lets look at some examples.

First, we can change the default cache directory path and assign it a new path:

```
>>> dbc.config_cache('default_cache_dir', 'new/path/cache/')
```

Or we could change the default directory where data files are downloaded:

```
>>> dbc.config_cache('default_download_dir', 'new/path/download/')
```

Or both at the same time:

```
>>> dbc.config_cache(field='info',
                     value={'default_cache_dir', 'new/path/cache/',
                            'default_download_dir': 'new/path/download/'})
```

Any field can be changed in the cache file just by specifying its name and the new value you want to change it with. The `field` arg looks for a string in the cache file and, if it finds a valid match, it replaces the first found match with the value provided with the `value` arg.

Other operations this method allows is to delete folders associated with the cache file or even the cache file itself. To remove just the cache folder where all the metadata `HDF5` files are stored, you need to do the following:

```
>>> dbc.config_cache(delete_cache_dir=True)
```

If you just want to remove or reset the cache file you can do the following:

```
>>> # Remove the cache file
>>> dbc.config_cache(delete_cache_file=True)

>>> # Reset the cache file (empty data)
>>> dbc.config_cache(reset_cache=True)
```

You can also bundle these arguments together to delete the cache dir and file:

```
>>> # Remove the cache file + dir
>>> dbc.config_cache(delete_cache_file=True, delete_cache_dir=True)
```

Or do it in one sweep:

```
>>> # Remove the cache file + dir
>>> dbc.config_cache(delete_cache=True)
```

Note: The `config_cache()` method is somewhat limited in its scope compared with other ways to change the cache contents (like directly modifying the cache file manually) but it has the necessary functionality to do the most common operations you might want to do like changing some fields or deleting / reseting the cache.

Warning: The `config_cache()` method should be used with extreme caution in order to not permanently delete your configurations. If you are going to use this method, please be aware of dangers of doing so.

Querying the cache

Sometimes you just need to search for some keywords in the cache and you don't want to take the effort to do it with a terminal.

The query() method allows to search for a pattern in the cache file and returns the contents of that pattern for any match. This means it can return multiple values depending on the the pattern and the configurations registered in your cache file.

For example, you might want to know what are the default paths for the cache and download dirs in your system. To retrieve this information, you can use the `query()` method and search for the `info` pattern like this:

```
>>> dbc.query('info')
[{'info': {'default_cache_dir': '/home/mf/dbcollection', 'default_download_dir':
'/home/mf/dbcollection/data/'}}]
```

We can also list all categories listed in the cache file:

```
>>> dbc.query('category')
[{'category': {'classification': ['mnist', 'cifar10'], 'detection':
['leeds_sports_pose'], 'human pose': ['leeds_sports_pose'],
'image_processing': ['leeds_sports_pose', 'cifar10'], 'keypoints':
['leeds_sports_pose']}}]
```

This can also be useful to locate the contents of a dataset.

```
>>> dbc.query('mnist')
[{'mnist': {'data_dir': '/home/mf/dbcollection/mnist/data', 'keywords':
['classification'], 'tasks': {'classification': '/home/mf/dbcollection/mnist/
classification.h5'}}}]
```

If the pattern was not found in the cache file, it return an empty dictionary.

```
>>> dbc.query('cifar1000')
[]
```

We can also retrieve information of all datasets that have the same keyword.

```
>>> dbc.query('classification')
[{'dataset': {'cifar10': {'keywords': ['classification']}}}, {'dataset':
{'mnist': {'keywords': ['classification']}}}]
```

The `query()` method is ment to do simple searches of patterns. It is always much more useful to take a look at the cache file itself, but for its scope it may provide the necessary functionality you might need.

Displaying cache information

You can display the contents of you cache file into the screen by using the info_cache() method.

It prints the cache file contents in a structured way for easier visualization.

```
>>> dbc.info_cache()
--------------
  Paths info
--------------
{
    "default_cache_dir": "/home/mf/dbcollection",
    "default_download_dir": "/home/mf/dbcollection/data/"
}

----------------
  Dataset info
```

(continues on next page)

(continued from previous page)

```
----------------
{
    "cifar10": {
        "data_dir": "/home/mf/dbcollection/data/cifar10/data",
        "keywords": [
            "image_processing",
            "classification"
        ],
        "tasks": {
            "classification": "/home/mf/dbcollection/cifar10/classification.h5"
        }
    },
    "cifar100": {
        "data_dir": "/home/mf/dbcollection/data/cifar100/data",
        "keywords": [
            "image_processing",
            "classification"
        ],
        "tasks": {
            "classification": "/home/mf/dbcollection/cifar100/classification.h5"
        }
    },
    "mnist": {
        "data_dir": "/home/mf/dbcollection/mnist/data",
        "keywords": [
            "classification"
        ],
        "tasks": {
            "classification": "/home/mf/dbcollection/mnist/classification.h5"
        }
    }
}

------------------------
  Datasets by category
------------------------

   > classification:   ['cifar10', 'cifar100', 'mnist']
   > image_processing: ['cifar10', 'cifar100']
```

You can select what sections you want to display by using the `paths_info`, `datasets_info` and `categories_info` args.

```
>>> dbc.info_cache(datasets_info=False, categories_info=False)
--------------
  Paths info
--------------
{
    "default_cache_dir": "/home/mf/dbcollection",
    "default_download_dir": "/home/mf/dbcollection/data/"
}

>>> dbc.info_cache(paths_info=False, categories_info=False)
----------------
  Dataset info
----------------
```

(continues on next page)

(continued from previous page)

```
{
    "cifar10": {
        "data_dir": "/home/mf/dbcollection/data/cifar10/data",
        "keywords": [
            "image_processing",
            "classification"
        ],
        "tasks": {
            "classification": "/home/mf/dbcollection/cifar10/classification.h5"
        }
    },
    "cifar100": {
        "data_dir": "/home/mf/dbcollection/data/cifar100/data",
        "keywords": [
            "image_processing",
            "classification"
        ],
        "tasks": {
            "classification": "/home/mf/dbcollection/cifar100/classification.h5"
        }
    },
    "mnist": {
        "data_dir": "/home/mf/dbcollection/mnist/data",
        "keywords": [
            "classification"
        ],
        "tasks": {
            "classification": "/home/mf/dbcollection/mnist/classification.h5"
        }
    }
}

>>> dbc.info_cache(paths_info=False, datasets_info=False)
------------------------
  Datasets by category
------------------------

   > classification:   ['cifar10', 'cifar100', 'mnist']
   > image_processing: ['cifar10', 'cifar100']
```

This method also allows you to select information about a single or multiple datasets.

```
>>> dbc.info_cache('cifar10')
--------------
  Paths info
--------------
{
    "default_cache_dir": "/home/mf/dbcollection",
    "default_download_dir": "/home/mf/dbcollection/data/"
}

-----------------
  Dataset info
-----------------
{
    "cifar10": {
        "data_dir": "/home/mf/dbcollection/data/cifar10/data",
```

(continues on next page)

(continued from previous page)

```
        "keywords": [
            "image_processing",
            "classification"
        ],
        "tasks": {
            "classification": "/home/mf/dbcollection/cifar10/classification.h5"
        }
    }
}

------------------------
  Datasets by category
------------------------

   > classification:   ['cifar10']
   > image_processing: ['cifar10']


>>> dbc.info_cache(['cifar10', 'cifar100'])
--------------
  Paths info
--------------
{
    "default_cache_dir": "/home/mf/dbcollection",
    "default_download_dir": "/home/mf/dbcollection/data/"
}

----------------
  Dataset info
----------------
{
    "cifar10": {
        "data_dir": "/home/mf/dbcollection/data/cifar10/data",
        "keywords": [
            "image_processing",
            "classification"
        ],
        "tasks": {
            "classification": "/home/mf/dbcollection/cifar10/classification.h5"
        }
    },
    "cifar100": {
        "data_dir": "/home/mf/dbcollection/data/cifar100/data",
        "keywords": [
            "image_processing",
            "classification"
        ],
        "tasks": {
            "classification": "/home/mf/dbcollection/cifar100/classification.h5"
        }
    }
}

------------------------
  Datasets by category
------------------------
```

(continues on next page)

(continued from previous page)

```
   > classification:   ['cifar10', 'cifar100']
   > image_processing: ['cifar10', 'cifar100']
```

Displaying information about available datasets

Some important information about **dbcollection** is the list of available datasets for load + download. Also, information about what tasks are available per dataset is of great interest.

This is achieved via the info_datasets() method that lists what datasets are available for load in your system and what datasets and tasks are available for download.

```
>>> dbc.info_datasets()
---------------------------------------
  Available datasets in cache for load
---------------------------------------
  - cifar10  ['classification']
  - cifar100  ['classification']
  - mnist  ['classification']

-----------------------------------
  Available datasets for download
-----------------------------------
  - caltech_pedestrian  ['detection', 'detection_10x', 'detection_30x']
  - cifar10  ['classification']
  - cifar100  ['classification']
  - coco  ['caption_2015', 'caption_2016', 'detection_2015', 'detection_2016',
↪'keypoints_2016']
  - flic  ['keypoints']
  - ilsvrc2012  ['classification', 'raw256']
  - inria_pedestrian  ['detection']
  - leeds_sports_pose  ['keypoints', 'keypoints_original']
  - leeds_sports_pose_extended  ['keypoints']
  - mnist  ['classification']
  - mpii_pose  ['keypoints', 'keypoints_full']
  - pascal_voc_2007  ['detection']
  - pascal_voc_2012  ['detection']
  - ucf_101  ['recognition']
  - ucf_sports  ['recognition']
```

You can define what information you want to print to the screen via the args `show_downloaded` and `show_available`.

```
>>> dbc.info_datasets(show_available=False)
---------------------------------------
  Available datasets in cache for load
---------------------------------------
  - cifar10  ['classification']
  - cifar100  ['classification']
  - mnist  ['classification']


>>> dbc.info_datasets(show_downloaded=False)
-----------------------------------
  Available datasets for download
-----------------------------------
```

(continues on next page)

(continued from previous page)

```
  - caltech_pedestrian  ['detection', 'detection_10x', 'detection_30x']
  - cifar10  ['classification']
  - cifar100  ['classification']
  - coco  ['caption_2015', 'caption_2016', 'detection_2015', 'detection_2016',
↪'keypoints_2016']
  - flic  ['keypoints']
  - ilsvrc2012  ['classification', 'raw256']
  - inria_pedestrian  ['detection']
  - leeds_sports_pose  ['keypoints', 'keypoints_original']
  - leeds_sports_pose_extended  ['keypoints']
  - mnist  ['classification']
  - mpii_pose  ['keypoints', 'keypoints_full']
  - pascal_voc_2007  ['detection']
  - pascal_voc_2012  ['detection']
  - ucf_101  ['recognition']
  - ucf_sports  ['recognition']
```

Also, you can list only the datasets that match a certain pattern.

```
>>> dbc.info_datasets('pascal', show_downloaded=False)
-----------------------------------
  Available datasets for download
-----------------------------------
  - pascal_voc_2007  ['detection']
  - pascal_voc_2012  ['detection']
```

Best practices

These are some of the recommended practices when dealing with managing datasets using the **dbcollection** package.

There are several ways to achieve the same result and these are some of the most common practices that you might encounter when using this package.

Before downloading / loading a dataset

There are some things you should do prior to use the `download()` and `load()` methods if you don't like the default setup. For example, it is always useful to specify where you want the data files to be stored in your system.

It is best practice to store all your files in the same directory. If you have an SSD drive but you don't want to store eveything there, it is best to use the `data_dir` argument tom specify uses cases, while keeping everything else under the same directory.

To do this, set the paths for the `default_download_dir` in your cache file. The recommended way to do this is the following:

```
>>> dbc.cache.download_dir = 'new/path/to/download/data/'
```

You can also do this for the cache dir where the metadata files are stored, but it shouldn't be a big deal unless you are really keen for quick data accesses. Likewise, you should use the following way to set the path of the cache dir:

```
>>> dbc.cache.cache_dir = 'new/path/to/cache/metadata/'
```

Removing datasets from cache

The easiest and the less error prone way to remove a dataset from cache is by using the `remove()` method. This will parse the cache file and remove any information regarding the specific dataset you are looking to remove.

The same is valid for removing a task of a dataset. Or the data files in your system.

Resetting / deleting the cache file

If you must delete or reset the contents of your file, there are two recommended ways for you to do this:

1. Using the `config_cache()` method;
2. Manually deleting the file from your filesystem.

Both will accomplish the same goal, so feel free to chose the one that fits you best.

Changing fields / values of the cache

Here I strongly recommend you to do this process by hand. This means opening the `~/dbcollection.json` file and manually changing the field or value you want. This may be easier to do or to understand what is actually being changed, and it should be less error prone for most users.

Also, this isn't a common operation todo, so take this advice with a grain of salt.

Checking the contents of the cache file

The contents of the `~/dbcollection.json` cache file may be hard to read when you have many many datasets registered, so it is best to use the `info_cache()` because it produces much nicer outputs for the cache, and you always define what data you want to visualize.

Checking what datasets are available for download

I strongly recommend you to check the documentation in order to see what datasets are available for download / use.

The `info_datasets()` does list all available datasets + tasks in the **dbcollection** package and it is fine to see the list of available datasets. However, the documentation has much more information about them.

1.1.5 Fetching data

Apart from managing datasets, another goal of **dbcollection** is to provide a way to easily pull data samples of a dataset in a straight-forward manner. Just like managing datasets, fetching data is also very easy!

To accomplish this feat, we make use of the `HDF5` file format. This enables us to overcome some issues related with other commonly used data formats like `.json`, `.csv` or plain `.txt` for storing data:

- The `HDF5` file structure is easy to understand;
- Easy to use API;
- Data accesses from disk are fast;
- It uses a file handler so it does not need to load the entire file to memory;
- Efficient in fetching chunks of data;

- It offers lots of useful features like data compression.

Another useful feature of this file format is how the data is structured. Internally, it acts like a file system in the sense that data is stored hierarchically, so you'll have your data structure well defined.

Also, since you don't need to load the entire file into memory, you save:

- **Resources**: big datasets will have the same impact on the system's memory as small datasets;
- **Time**: because only a file handler is required to access data, loading a dataset is a quick process.

This Chapter deals with retrieving data from a dataset. In the following sections, we'll address how data is structured and how to fetch (and parse) data samples from a dataset. Also, best practices about retrieving data using this package are detailed at the end of this page.

Data structure of a dataset in an HDF5 file

Before proceeding on explaining how to retrieve data from a dataset, it is important to explain how data is stored inside an `HDF5` file.

The data file resembles a file system, so lets assume that the root of this file system is defined by the `/` symbol.

```
/
...
```

Datasets are usually split into several sets of data which are normally used for training, validation and testing. Some might have more splits, other fewer, but generally this is how they are setup.

To better explain this, we'll use the *MNIST* dataset as an example to describe how datasets are structured inside an `HDF5` file.

The `mnist` dataset contains two set splits: **train** and **test**. In an `HDF5` file these are called `Groups` and they are basically folders inside the file.

```
/
├── train/
│   ...
└── test/
    ...
```

Now, for each split, several data fields compose the metadata / annotations of the dataset. These can be images, labels, filenames, bounding boxes, ordered lists, etc., and they convey the available information to be retrieved by the user. In an `HDF5` file these are called `Datasets` and they store arrays of data as fields.

The `mnist` dataset contains the following fields for each set:

```
/
├── train/
│   ├── classes
│   ├── images
│   ├── labels
│   ├── object_fields
│   ├── object_ids
│   └── list_images_per_class
└── test/
    ├── classes
    ├── images
```

(continues on next page)

(continued from previous page)

```
    ├── labels
    ├── object_fields
    ├── object_ids
    └── list_images_per_class
```

As you can see, data is stored in a hierarchical way inside a metadata file.

Note: Notice that both sets have the same fields. This is not the case for other datasets. For more information about how a certain dataset is structured see the *Available datasets* Chapter.

Now, there are some aspects that need to be addressed about some of fields of these sets.

For clarity sake, lets only consider the `train` set of this example. We could split these fields into three categories:

1. **data fields**,
2. **object fields**
3. **organized list(s)**.

The **data fields** category represents the actual data contents of the dataset. For the `mnist` example, there only exists information about the `labels`, `classes` and `images` tensors. Each is a N-dimensional array of data where each row corresponds to a sample of data, and the dimensionality of these arrays varies between types of fields.

The **object fields** category is made of special crafted fields that exist in all datasets in this package. They are basically aggregators of data fields, for example tables of databases that aggregate foreign keys of other tables.

Here, we have two fields that do this job for us: `object_fields` and `object_ids`. `object_fields` is an 2D array that contains an ordered list of the set's field names. These names are used for fetching data of these data fields by some API methods, but, more importantly, it shows how data is structured in the `object_ids` field. This last field is also an 2D array but it contains indexes of data fields instead. It is this field that correlates different labels / classes for different images for this example. For other datasets, for example, it is this field that links image files with labels with bounding boxes, etc. In the following sections we'll see more clearly the role of these two fields in fetching data.

Lastly, the **organized list(s)** category corresponds to pre-ordered, pre-computed lists that may be helpful for some use cases. For example, for object detection scenarios, having a list of order bounding boxes per image may be useful for selecting only one box per image when creating batches of data. The number of these list fields varies from dataset to dataset, but their use case should be easy to understand just by looking at its name.

In summary, it is important to understand how datasets are structure before proceeding to retrieve data from them. Also, every dataset has its data structured in its own way, but the relationship between them is known via two special fields (`object_fields` and `object_ids`). With this knowledge, you should now be ready to tackle how to fetch data from the metadata files associated to a given task of a dataset.

Note: If you want to know more about how the available datasets in this package are structured, please see the *Available datasets* Chapter for more information about them.

Retrieving data from a dataset

To retrieve data from a dataset, we must first load it.

In this section we'll continue using the `mnist` dataset as our example for explaining how we can retrieve data samples for this dataset.

Loading a dataset

This section has been explained in detail in previously Chapters. Therefore, lets load the `mnist` dataset in the simplest way possible using the `load()` method:

```
>>> mnist = dbc.load('mnist')
```

When selecting a dataset, the `load()` method returns a `Dataloader` object that contains a series of methods and attributes that will be used to query and store data.

The DataLoader object

Printing this data loader object prints the name of the dataset that is associated with and which task was selected.

```
>>> print(mnist)
DataLoader: "mnist" (classification task)
```

Now, lets take a better look what attributes and methods this object contains:

```
>>> mnist.
mnist.data_dir          mnist.list(             mnist.size(
mnist.db_name           mnist.object(           mnist.task
mnist.get(              mnist.object_field_id(  mnist.test
mnist.hdf5_file         mnist.object_fields     mnist.train
mnist.hdf5_filepath     mnist.root_path
mnist.info(             mnist.sets
```

It contains the following attributes:

- `data_dir`: Directory path where the source data files are stored in disk;
- `db_name`: Name of the dataset;
- `task`: Name of the task;
- `object_fields`: Data field names for each set;
- `sets`: List of names of set splits (train, test).

These attributes provide useful information about the loaded dataset. The `sets` and `object_fields` attributes provide relevant information about the number and name of the set splits and the data fields that each set contains, respectively. This is useful information when retrieving data using the `DataLoader` API methods.

The API methods for fetching data or information of data for this object are the following:

- `get()`: Retrieves data from the dataset's `HDF5` metadata file;
- `object()`: Retrieves a list of all fields' indexes/values of an object composition;
- `object_field_id()`: Retrieves the index position of a field in the `object_ids` list;
- `list()`: List of all field names of a set;
- `size()`: Size of a field;
- `info()`: Prints information about all data fields of a set.

The first two methods are used to fetch data samples from the `HDF5` metadata file. The other methods provide information about the data fields.

Regarding fetching data, both `get()` and `object` methods return data samples, but their purpose differs slightly enough that it justifies having two of such methods. `get` is used to fetch data of single fields, while `object` is used to collect data from multiple fields that compose an 'object'.

In the next subsection we'll see more clearly this difference between these two methods.

Note: For more information, see the *DataLoader* section in the Reference manual.

Fetching data using the get() and object() API methods

Now, lets proceed to retrieve data using these two API methods.

Lets sample the first 10 images from the training set.

```
>>> imgs = mnist.get('train', 'images', range(10))
>>> type(imgs)
<class 'numpy.ndarray'>
>>> imgs.shape
(10, 28, 28)
```

Retrieving the first 10 images from the `mnist` dataset is very simple! You just need to provide the name of the set and the name of the data field you want to retrieve data from and the indices of the samples.

In turn, this returns a `numpy.ndarray` with the images' data. The same procedure is done to retrieve data from the other data fields.

If we wanted to return an image and the label associated with it for a given 'object', we would need to determine the indices of each field so we could fetch the correct samples. This is how you would do this to return the 100th sample object:

```
>>> # First, see what fields compose the 'object_ids' field
>>> mnist.object_fields['train']
('images', 'labels')
>>> # Next, get the indices of the fields for the 100th sample object
>>> ids = mnist.get('train', 'object_ids', 99)
>>> ids
array([99,  1], dtype=int32)
>>> # Then, fetch the data of the 'images' field
>>> img = mnist.get('train', 'images', ids[0])
>>> img.shape
(28, 28)
>>> # Finally, fetch the data of the 'labels' field
>>> lbl = mnist.get('train', 'labels', ids[1])
>>> lbl
1
```

This took quite a few steps to do: first you have to find the name of the fields that compose the 'object', then find the ids for each field and then retrieve the data for each sample.

We can write the same example in fewer lines using the `object()` method and obtain the same results.

```
>>> # Just to show which fields compose the 'object_ids' field
>>> mnist.object_fields['train']
('images', 'labels')
>>> # Fetch the data in a single command using 'object()'
>>> (img, lbl) = mnist.object('train', 99, convert_to_value=True)
```

(continues on next page)

(continued from previous page)

```
>>> img.shape
(28, 28)
>>> lbl
1
```

As you can see, it is much simpler to fetch data this way. The `object()` method receives the set name and the sample object index we want to fetch. If you don't set `convert_to_value=True`, the method will only return the indexes of the fields.

With these methods, you can input an index or a list of indexes and retrieve data for any data field existing in a set. The values on this lists don't need to be contiguous (thanks to `h5py`).

For example, fetching the first 5 even images is just a matter of passing the right list:

```
>>> imgs = mnist.get('train', 'images', [0, 2, 4, 6, 8])
>>> imgs.shape
(5, 28, 28)
```

Or, if you want to get all images, you don't need to pass any index:

```
>>> imgs = mnist.get('train', 'images')
>>> imgs.shape
(60000, 28, 28)
```

These methods are quite flexible about what format of inputs they receive, just as long as the input contains valid value ranges.

Fetching data by accessing data fields directly

There is another way to fetch data besides using `get()` and `object()`. This is done by accessing directly the data fields themselves. To explain this better lets take a look again at the attributes of the `DataLoader` object that we've seen before.

```
>>> mnist.
mnist.data_dir          mnist.list(             mnist.size(
mnist.db_name           mnist.object(           mnist.task
mnist.get(              mnist.object_field_id(  mnist.test  <---
mnist.hdf5_file         mnist.object_fields     mnist.train  <---
mnist.hdf5_filepath     mnist.root_path
mnist.info(             mnist.sets
```

Note these two attributes highlighted here. These attributes refer to the set splits of the dataset, and are object of type *SetLoader*:

```
>>> mnist.sets
('train', 'test')
>>> mnist.train
SetLoader: set<train>, len<60000>
>>> mnist.test
SetLoader: set<test>, len<10000>
```

The set groups in an `HDF5` file are converted when loading a `DataLoader` object into attributes objects of type `SetLoader`. These also contain their own set of attributes and methods:

```
>>> mnist.train.
mnist.train.classes                 mnist.train.list_images_per_class
mnist.train.data                    mnist.train.nelems
mnist.train.fields                  mnist.train.object(
mnist.train.get(                    mnist.train.object_field_id(
mnist.train.images                  mnist.train.object_fields
mnist.train.info(                   mnist.train.object_ids
mnist.train.labels                  mnist.train.set
mnist.train.list(                   mnist.train.size(
```

As you can see, these objects also contain the same methods available in `DataLoader` objects and some other attributes that are the data fields of this set.

The only difference between these methods and the ones from `DataLoader` is that these do not require you to specify the set name to select data from:

```
>>> img = mnist.train.get('images', 0)
>>> img.shape
(28, 28)
```

The attribute fields of `SetLoader` objects like, for example, `mnist.train.classes` or `mnist.train.list_images_per_class` are also special objects of type `FieldLoader`.

Lets look at the `classes` field:

```
>>> train.mnist.classes
FieldLoader: <HDF5 dataset "classes": shape (10, 11), type "|u1">

>>> mnist.train.classes.
mnist.train.classes.data                mnist.train.classes.object_field_id(
mnist.train.classes.fillvalue           mnist.train.classes.set
mnist.train.classes.get(                mnist.train.classes.shape
mnist.train.classes.info(               mnist.train.classes.size(
mnist.train.classes.name                mnist.train.classes.type
mnist.train.classes.obj_id
```

Two important things to mention here. First, when printing the data field, you can see it is of type `HDF5 dataset`, which means that this field's data is retrieved from disk. Second, some methods available in `DataLoader` and `SetLoader` are also available here.

Note: For more information about how to transfer data to memory see the *Allocating data to memory* section at the end of this page.

The `FieldLoader` object contains all necessary information about a data field like `shape`, `type`, `fillvalue` (used for padding arrays), among others. The `data` attribute stores the data buffer that is used to fetch data samples.

To fetch data samples from `FieldLoader` objects, you can use common slicing operations like with `numpy.ndarrays` or `HDF5 dataset`:

```
>>> mnist.train.classes[0]
array([ 97, 105, 114, 112, 108,  97, 110, 101,   0,   0,   0], dtype=uint8)

>>> mnist.train.classes.data[0]
array([ 97, 105, 114, 112, 108,  97, 110, 101,   0,   0,   0], dtype=uint8)

>>> mnist.train.classes.get(0)
array([ 97, 105, 114, 112, 108,  97, 110, 101,   0,   0,   0], dtype=uint8)
```

All slicing operations used with **numpy** arrays are supported. This is because **h5py** uses **numpy** as the backend. Therefore, you can apply any operations to the output sample that you would do with **numpy** arrays.

```
>>> mnist.train.classes[0:5]
array([[ 97, 105, 114, 112, 108,  97, 110, 101,   0,   0,   0],
       [ 97, 117, 116, 111, 109, 111,  98, 105, 108, 101,   0],
       [ 98, 105, 114, 100,   0,   0,   0,   0,   0,   0,   0],
       [ 99,  97, 116,   0,   0,   0,   0,   0,   0,   0,   0],
       [100, 101, 101, 114,   0,   0,   0,   0,   0,   0,   0]], dtype=uint8)

>>> mnist.train.classes[0:5][1:3]
array([[ 97, 117, 116, 111, 109, 111,  98, 105, 108, 101,   0],
       [ 98, 105, 114, 100,   0,   0,   0,   0,   0,   0,   0]], dtype=uint8)

>>> mnist.train.classes[:, 1:2]
array([[105],
       [117],
       [105],
       [ 97],
       [101],
       [111],
       [114],
       [111],
       [104],
       [114]], dtype=uint8)
```

At this point, you should be have mastered everything you may need to know about fetching data using **dbcollection**.

Up until now, we've been focusing on retrieving data samples. However, most of the times you are using this package's API methods is to try to understand / visualize how data is structured and what type is a field composed of.

In the next section, we'll see some handy methods that enables us to visualize how data is structured in a comprehensible format using the REPL.

Visualizing information about sets and data fields

A good way to see how a dataset is structured is to use the `info()` method.

The `info()` method displays information about the sets that compose the dataset and the data fields of each set. It shows the `shape` and `type` of the fields, and also it the fields are **linked** in `object_ids` and in which position.

```
>>> mnist.info()

> Set: test
   - classes,         shape = (10, 2),          dtype = uint8
   - images,          shape = (10000, 28, 28),  dtype = uint8,  (in 'object_ids',␣
↪position = 0)
   - labels,          shape = (10000,),         dtype = uint8,  (in 'object_ids',␣
↪position = 1)
   - object_fields,   shape = (2, 7),           dtype = uint8
   - object_ids,      shape = (10000, 2),       dtype = uint8

   (Pre-ordered lists)
   - list_images_per_class,  shape = (10, 1135),  dtype = int32

> Set: train
   - classes,         shape = (10, 2),          dtype = uint8
   - images,          shape = (60000, 28, 28),  dtype = uint8,  (in 'object_ids',␣
↪position = 0)
```

(continues on next page)

(continued from previous page)

```
   - labels,          shape = (60000,),          dtype = uint8,   (in 'object_ids',␣
↪position = 1)
   - object_fields,   shape = (2, 7),            dtype = uint8
   - object_ids,      shape = (60000, 2),        dtype = uint8

   (Pre-ordered lists)
   - list_images_per_class,   shape = (10, 6742),   dtype = int32
```

With this method, you can display information of a single set if you want. For that, you need to pass the name of the set as input:

```
>>> mnist.info('test')

> Set: test
   - classes,         shape = (10, 2),           dtype = uint8
   - images,          shape = (10000, 28, 28),   dtype = uint8,   (in 'object_ids',␣
↪position = 0)
   - labels,          shape = (10000,),          dtype = uint8,   (in 'object_ids',␣
↪position = 1)
   - object_fields,   shape = (2, 7),            dtype = uint8
   - object_ids,      shape = (10000, 2),        dtype = uint8

   (Pre-ordered lists)
   - list_images_per_class,   shape = (10, 1135),   dtype = int32
```

If you are calling this method from an attribute field that is a `SetLoader` object, you don't need to specify the name of the set.

```
>>> mnist.test.info()

> Set: test
   - classes,         shape = (10, 2),           dtype = uint8
   - images,          shape = (10000, 28, 28),   dtype = uint8,   (in 'object_ids',␣
↪position = 0)
   - labels,          shape = (10000,),          dtype = uint8,   (in 'object_ids',␣
↪position = 1)
   - object_fields,   shape = (2, 7),            dtype = uint8
   - object_ids,      shape = (10000, 2),        dtype = uint8

   (Pre-ordered lists)
   - list_images_per_class,   shape = (10, 1135),   dtype = int32
```

You can even use this method on data fields!

```
>>> mnist.test.images.info()
Field: images,   shape = (10000, 28, 28),   dtype = uint8,   (in 'object_ids', position␣
↪= None)

>>> mnist.test.labels.info()
Field: labels,   shape = (10000,),   dtype = uint8,   (in 'object_ids', position = 1)
```

Along with the `info()` method, you have access to two additional methods: `size()` and `list()`.

`size()` returns a dictionary or a tuple of the `shape` of a data field. As with the `info()` method, you can use it in several ways:

```
>>> mnist.size()
{'train': (60000, 2), 'test': (10000, 2)}

>>> mnist.size('train')
(60000, 2)

>>> mnist.size('train', 'images')
(60000, 28, 28)

>>> mnist.test.size()
(10000, 2)

>>> mnist.test.images.size()
(10000, 28, 28)
```

The `list()` method also returns a dictionary or a tuple, but it contains the names of all data fields of a set or sets.

```
>>> mnist.list()
{'train': ('classes', 'labels', 'object_fields', 'object_ids', 'images',
'list_images_per_class'), 'test': ('classes', 'labels', 'object_fields',
'object_ids', 'images', 'list_images_per_class')}

>>> mnist.list('train')
('classes', 'labels', 'object_fields', 'object_ids', 'images',
'list_images_per_class')
```

These three methods enables you to quickly visualize the structure of a dataset and its data, and also to get the shape or names of some data fields with these simple commands.

Next comes the *Parsing data* section. This section shows how to deal with some quirks of the way data is stored in `HDF5` files. This is very important w.r.t. **dbcollection** because some trade-offs had to be made regarding data allocation into arrays.

Parsing data

> **Warning:** This section contains valuable information about how data itself is stored inside `HDF5` files.
>
> It is important that you read this section so you know why some data is stored the way it is and how to parse it correctly using some utility methods provided by the **dbcollection** package.

Since **dbcollection** uses **h5py** to store data into files in the `HDF5` format, some arrays my contain padding values.

Because of some technicalities in dealing with strings or lists, storing information into **numpy** arrays required us to use some form of padding in order to have evenly shaped arrays.

This did not impact the size of the metadata files with the extra information, but you still have to remove this padding effects from data in order to have the correct information.

Regarding strings and list of strings, these had to be converted into ASCII format in order to be stored. This information is explicitly available within the package / API itself, but it is well documented in the *Available datasets* Chapter for each available dataset on this package.

Unpadding data

Lets look how we can remove padding values from data. For example, lets consider that we have a data sample that is a **numpy** array and it contains some padding elements in it:

```
>>> import numpy as np
>>> my_list = np.array([[1,2,3,-1,-1],[5,6,-1,-1,-1],[1,-1,-1,-1,-1]], dtype=np.int32)
>>> my_list
array([[ 1,  2,  3, -1, -1],
       [ 5,  6, -1, -1, -1],
       [ 1, -1, -1, -1, -1]], dtype=int32)
```

In order to remove padding effects from this list (or data in general), you first need to know which padding value was used to fill it. This information can be obtain from the documentation or you can get this from the `fillvalue` field of `FieldLoader` objects.

```
>>> # for example
>>> mnist.train.classes.fillvalue
0
```

In this example, lets assume we know the fill value. Also, we can see that the fill value `-1` was used to pad this array because *1)* it sits at the right-hand side of the array and *2)* there are several of them in a sequence. This is a strong indicator that this value might be used for padding an array.

To remove this padding information, there is a method you can use from the `dbcollection.utils.pad` module for unpadding lists: *unpad_list()*. It receives a list and a value as input and outputs a parsed list.

```
>>> # import the method
>>> from dbcollection.utils.pad import unpad_list

>>> # same list as in the previous example
>>> my_list.tolist()
[[1,2,3,-1,-1],[5,6,-1,-1,-1],[1,-1,-1,-1,-1]]

>>> # unpad the list
>>> unpad_list(my_list.tolist(), -1)  # input must be a list
[[1, 2, 3], [5, 6], [1]]
```

That's it. We removed the padding effect off of a list.

Removing padding information from data is not hard, but it is necessary. In the next sub-section, we'll take a look at converting / decoding ASCII data to strings, which is something you'll be doing more frequently.

Note: For more information about utility methods available in this package, see the *Utils reference section*.

String<->ASCII convertion

String data is the most ubiquitous annotation information available. Class names, category names, file path + names, etc., exists in some for or shape in pretty much all datasets annotations.

Since we encoded string data to ASCII, we need to convert it back to string format in order to have any utility for our use.

In the `dbcollection.utils.string_ascii` module there are methods available for converting strings to ASCII and vice-versa. Here we'll only need to use the *convert_ascii_to_str()* method. It receives a `numpy.ndarray`

as input and returns a string or list of strings.

Lets use the `classes` field from the `mnist` dataset as example:

```
>>> mnist.train.classes[:]
array([[48,  0],
       [49,  0],
       [50,  0],
       [51,  0],
       [52,  0],
       [53,  0],
       [54,  0],
       [55,  0],
       [56,  0],
       [57,  0]], dtype=uint8)
```

ASCII encoded strings are always of type `np.uint8` and are padded with zeros. To convert this array back to strings, we must import the method from the `dbcollection.utils` module:

```
>>> from dbcollection.utils.string_ascii import convert_ascii_to_str as tostr_
```

This is usually the alias this method is given for being shorter to type. Finally, lets convert the array back to a string.

```
>>> tostr_(mnist.train.classes[:])
['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
```

That's it. Converting ASCII to string is easy and there are tool available at your disposal to make this process as trivial as possible.

Moving data from memory<->disk

An extra feature at your disposal is the ability to allocate a data field into memory. For cases where the dataset is very small (`mnist`, `cifar10`), you can simply move the data to RAM. To do this you just need to set the `to_memory` attribute to `True` of `FielLoader` data objects.

```
>>> # in disk
>>> mnist.train.images
FieldLoader: <HDF5 dataset "images": shape (60000, 28, 28), type "|u1">

>>> # move data to memory
>>> mnist.train.images.to_memory = True

>>> # in memory
>>> mnist.train.images
FieldLoader: <numpy.ndarray "images": shape (60000, 28, 28), type uint8>
```

To revert the process you just need to set `to_memory` to `False`.

```
>>> # move data back to disk
>>> mnist.train.images.to_memory = False

>>> # in disk
>>> mnist.train.images
FieldLoader: <HDF5 dataset "images": shape (60000, 28, 28), type "|u1">
```

Best practices

Here's a list of best practices you might want to consider when fetching data:

- Use the `get()` and `object()` API methods for fetching data. They are simple to use and your code will be easier to understand. Also, when fetching data between different sets, all you need to do is use a different set name and your code remains the same.
- Check the documentation first if you are new to **dbcollection**. When you become more confortable using this package, then you'll probably only need to use the `info()` method for visualizing data.
- Before parsing a field, use the available information about the particular dataset you are working with in the *Available datasets* Chapter in the documentation. Each dataset has its own manual which provides detailed information about what fields it has, what's their shape, what padding value do they have, their purpose of use, the type of data, or even if it is a string encoded as ASCII.
- Concerning allocating data into memory, check the size of the data field before loading it into memory. All data fields in `HDF5` files are compressed and the size they occupy in disk may be misleading. Proceed with caution!
- The documentation is your friend! :)

1.1.6 Managing the cache

This Chapter addresses managing / configurating the cache registry of used datasets.

The `~/dbcollection.json` cache file located in your home directory is the central registry of **dbcollection**. Here is stored the information about what datasets have been downloaded / parsed, which tasks are listed for use, where is data stored and what categories exist.

It is important to keep this file in your system's home directory because there is where **dbcollection** tried to locate it. If it is not found, an empty one will be generated.

A situation that might take you to manually configure this cache file, opposed to letting the package do it for you, is when you have moved data around to other paths in your disk and you need to fix this in the cache file. Here we'll see how to deal with this kind of scenarios.

In this Chapter we'll see the most important operations you may want / need to do to configure the cache file in your system.

Cache's structure

The `~/dbcollection.json` cache file is composed of three main sections:

- `info`: holds default paths configurations;
- `datasets`: list of datasets information like data path, tasks and keywords;
- `category`: list of datasets grouped by categories (e.g., classification, keypoints, object_detection, etc.).

For example, this is how a cache file is generally formatted:

```
{
    "info": {
         "default_cache_dir": "/home/mf/dbcollection",
         "default_download_dir": "/home/mf/dbcollection/downloaded_data"
    },
    "datasets": {
        "mnist": {
             "data_dir": "/home/mf/dbcollection/downloaded_data/mnist/data",
```

(continues on next page)

(continued from previous page)

```
            "keywords": [
                "image_processing",
                "classification",
            ],
            "tasks": {
                "classification": "/home/mf/dbcollection/mnist/classification.h5"
            }
        }
    },
    "category": {
         "classification": [
             "mnist"
         ],
         "image_processing": [
             "mnist"
         ],
    }
}
```

The `info` section contains default paths of where the `HDF5` metadata files and source data files are stored.

The `datasets` section stores the metadata information for each dataset loaded in the system. Here you'll find what tasks have been processed, where the source data files are stored and what keywords define them.

The `category` section contains all categories defined in the `keywords` field of each dataset and groups them by name. The main purpose for this section is to help find similar datasets for a certain category.

Accessing the cache's contents

To access the cache's contents you can:

- Open the `~/dbcollection.json` cache file in the filesystem;
- Use the `config_cache()`, `query()` and `info_cache()` methods;
- Use the `.cache` attribute which is loaded when importing the package.

Opening the cache file is the easiest way to visualize and modify the contents of the cache.

The `config_cache()`, `query()` and `info_cache()` methods are useful when simple operations like displaying the cache contents or modifying a field is required.

The last way to access the cache is by accessing the `.cache` attribute of the package. When importing **dbcollection**, the cache file is automatically loaded into memory as a *CacheManager* object.

```
>>> import dbcollection as dbc
>>> dbc.cache
<dbcollection.core.cache.CacheManager object at 0x7f2875878550>
```

This object contains methods attributes and methods for managing the cache in a deeper level compared to the previous ways to deal with the cache registry.

Here is the list of attributes and methods that compose the `CacheManager` object:

```
>>> dbc.cache.
dbc.cache.__class__(                    dbc.cache._set_download_dir(
dbc.cache.__delattr__(                  dbc.cache.add_data(
dbc.cache.__dict__                      dbc.cache.add_keywords(
```

(continues on next page)

(continued from previous page)

```
dbc.cache.__dir__(                    dbc.cache.cache_dir
dbc.cache.__doc__                     dbc.cache.cache_filename
dbc.cache.__eq__(                     dbc.cache.check_dataset_name(
dbc.cache.__format__(                 dbc.cache.clear(
dbc.cache.__ge__(                     dbc.cache.create_os_home_dir(
dbc.cache.__getattribute__(           dbc.cache.data
dbc.cache.__gt__(                     dbc.cache.delete_category_entry(
dbc.cache.__hash__(                   dbc.cache.delete_dataset(
dbc.cache.__init__(                   dbc.cache.delete_dataset_cache(
dbc.cache.__le__(                     dbc.cache.delete_entry(
dbc.cache.__lt__(                     dbc.cache.delete_task(
dbc.cache.__module__                  dbc.cache.download_dir
dbc.cache.__ne__(                     dbc.cache.exists_dataset(
dbc.cache.__new__(                    dbc.cache.exists_task(
dbc.cache.__reduce__(                 dbc.cache.get_dataset_storage_paths(
dbc.cache.__reduce_ex__(              dbc.cache.get_task_cache_path(
dbc.cache.__repr__(                   dbc.cache.info(
dbc.cache.__setattr__(                dbc.cache.is_empty(
dbc.cache.__sizeof__(                 dbc.cache.is_test
dbc.cache.__str__(                    dbc.cache.modify_field(
dbc.cache.__subclasshook__(           dbc.cache.read_data_cache(
dbc.cache.__weakref__                 dbc.cache.read_data_cache_file(
dbc.cache._cache_dir                  dbc.cache.reload_cache(
dbc.cache._default_cache_dir_path(    dbc.cache.reset_cache(
dbc.cache._empty_data(                dbc.cache.reset_cache_dir(
dbc.cache._get_cache_dir(             dbc.cache.reset_download_dir(
dbc.cache._get_download_dir(          dbc.cache.update(
dbc.cache._os_remove(                 dbc.cache.write_data_cache(
dbc.cache._set_cache_dir(
```

The cache's contents are stored in a dictionary under the `.data` attribute. Although this way of accessing the contents of the cache is a bit more complex than the other two, it does provide some functionality that is very useful for certain cases.

In the following sections we'll take a look at the most common operations that you might need to know to manage **dbcollection**'s cache like adding, modifying or deleting a dataset or task or reseting path defaults.

Note: The `CacheManager` object contains many methods for specific actions, and, to learn more about them, it is encouraged to take a look at the **reference manual** for more details about them. Only the most important ones will be covered in this Chapter.

Displaying the cache's contents

To visualize the cache's contents you can:

- Open the cache file in your filesystem;
- Use `dbc.info_cache()` or `dbc.cache.info()` methods to display the contents to the screen;

These two methods display the same information about the cache, so use whichever method you prefer.

Basic operations

The most common operations you may need are *adding*, *modifying*, *deleting* and *querying* the cache.

There are other available operations you can do, but on this section we'll focus on these four basic operations which should cover most use cases when dealing with the cache registry.

Note: All following operations can be done by manually opening the file modyfing its contents. Here, we'll only focus on doing these operations using the available attributes / methods in **dbcollection**.

Getting information about a dataset

To fetch the cache's contents about a dataset, you can access the `.data` attribute. For example, to fetch the data about the `mnist` dataset, you can do the following:

```
>>> dbc.cache.data['dataset']['mnist']
{'data_dir': '/home/mf/dbcollection/downloaded_data/mnist/data', 'tasks':
{'classification': '/home/mf/dbcollection/mnist/classification.h5'}, 'keywords':
'classification'}
```

Adding a dataset

Inserting a data into the cache is done by using the `add_data()` method. It requires the name and the data in order to insert it into the registry. This will create an entry under the `datasets` and `category` sections.

```
>>> # add a custom dataset
>>> dbc.cache.add_data('new_dataset',
                       {'data_dir': '/some/new/dir',
                        'tasks': {'some_task': '/path/to/task/some_task.h5'},
                        'keywords': ['list', 'of', 'keywords']})
```

Adding a task

To add a task to an existing dataset, you can proceed in two ways.

1. The first way is to use the `.data` attribute field and insert a new task into the dictionary:

   ```
   >>> # add a new task to mnist
   >>> dbc.cache.data['dataset']['mnist']['tasks'].update({'new_task': 'path/to/new/
   ↪task.h5'})
   >>> dbc.cache.write_data_cache(dbc.cache.data)  # write the changes to disk
   ```

2. The second way is by using the `.add_data()` method:

   ```
   >>> # get mnist data
   >>> mnist_metadata = dbc.cache.data['dataset']['mnist']
   >>> # add a new task
   >>> mnist_metadata['tasks'].update({'new_task': 'path/to/new/task.h5'})
   >>> dbc.cache.add_data('mnist', mnist_metadata, is_append=True)
   ```

Removing a dataset

Removing dataset entrys is pretty simple. To do this use the `delete_dataset()` method to remove the dataset from the cache:

```
>>> dbc.cache.delete_dataset('mnist')
```

Please note that this will also remove the dataset's directory in disk.

Warning: You cannot remove a dataset simply by removing the entry from `.data`'s dictionary. This would require you to write the changes to disk and you would also need to remove all the registries of the dataset from `category`.

Removing a task

Just like removing datasets, to remove a task you simply need to call the `delete_task()` method. This method requires you to specify the dataset's name and the task you want to remove. This will remove the task entry from the cache and its file from disk.

```
>>> dbc.cache.delete_dataset('mnist', 'classification')
```

Modifying data

The process of modifying data is simillar to assigning new information to the cache.

The easiest way to do this is by changing the contents of `.data` and writting the changes back to disk:

```
>>> # do stuff to data
>>> dbc.cache.data['info']['default_download_dir'] = 'new/save/dir'
>>> # write changes to disk
>>> dbc.cache.write_data_cache(dbc.cache.data)
```

Reset the cache

There might come a time where you need to reset the configurations of your cache.

For whatever reason you may need to do this, you just need to use the `reset_cache()` method and it will reset the contents of your cache file, leaving it empty of information about datasets and restoring the default paths for the cache downloaded files diretories.

```
>>> dbc.cache.reset_cache(force_reset=True)
```

Note: You have to explicitly set `force_reset` to `True`. This is a failsafe mechanism to avoid unintended resets of the cache.

Check if a dataset exists

To see if a dataset exists you can:

1. Check if the name exists in `.data`.

```
>>> 'mnist' in dbc.cache.data['datasets']
True
```

2. Use the `exists_dataset()` method.

```
>>> dbc.cache.exists_dataset('mnist')
True
```

Check if a task exists

To check if a task exists for a dataset you can:

1. Transverse the dataset's metadata in `.data` and look for a particular task name.

```
>>> 'classification' in dbc.cache.data['datasets']['mnist']['tasks']
True
```

2. Use the `exists_task()` method.

```
>>> dbc.cache.exists_task('mnist', 'classification')
True
```

Other useful operations

Here are a few other operations that will be very useful for you to use.

Change the default metadata cache directory

Changing the directory where the `HDF5` metadata files are stored may be usefull if you want to store these files in another disk like an SSD.

To do this, you simply need to assign a new path to the `.cache_dir` attribute field and it will automatically register it both in memory and in disk.

```
>>> dbc.cache.cache_dir = 'new/path/cache/'
```

Also, you can check what the current path where the cache data is stored by printing this field:

```
>>> dbc.cache.cache_dir
'new/path/cache/'
```

Change the default download directory path

Like with the cache directory, you can also change the default path where source data files of datasets are stored via the `.download_dir` attribute field.

Just assign a new path to it to change where the source files are stored in disk:

```
>>> dbc.cache.download_dir = 'new/save/path/download/data/'
```

Likewise, to see check what is the default path where downloaded data files are stored just print this field:

```
>>> dbc.cache.download_dir
'new/save/path/download/data/'
```

Reloading the cache

Consider the following: you've changed something in the cache file (manually or by some other way) but you want to discard them and get back the previous cache state.

This is achieved by useing the `reload_cache()` method which loads the cache's contents in disk back to memory, regenerating the previous information.

```
>>> dbc.cache.reload_cache()
```

1.1.7 Creating a new dataset

Creating a new dataset requires you to set the download, setup and parse scripts in a certain way. In order to properly setup a new dataset, you need to accomplish these three following steps:

1. First, create a directory under the `datasets/` dir so it can be loaded to the list of available datasets.
2. Then, set up a `__init__.py` file for storing the configurations of the new dataset (urls, keywords, tasks) and to create unique `.py` files for each task your dataset will have.
3. Finally, include a `README.rst` documentation file for the dataset in the same directory. This should provide a 'how to use' manual for users to understand the data structure of the `HDF5` metadata files.

With these steps you are done with creating a dataset. The package will automatically search for directories under `datasets/` which have a specific class in the `__init__.py` file and includes it into a list of datasets.

In the following sections we'll take a closer look on how to properly configure and set up these files and directories.

Setting up the dataset's directory

To create a new dataset you need to create a directory with the same name of the dataset you want it be called under the `datasets/` directory.

You can use multiple nested directories to store your dataset in cases where other related datasets will be grouped under a common dir. For example, the `cifar10` and `cifar100` datasets are stored under the `cifar/` directory:

```
dbcollection/
├── core/
│       ...
├── datasets/
│   ├── cifar/
│   │   ├── cifar10/
│   │   └── cifar100/
│   └── ...
├── tests/
│       ...
│
└── utils/
        ...
```

This enables the package to group several different (or similar) versions of the same dataset under the same parent folder for organizational purposes, thus maintaing a clear structure for future additions of datasets in the root `datasets/` directory that may contain many related datasets.

Note: Please use **snake_case** when naming the directory by using all characters in lower case and separated by an underscore. See the *Coding guidelines* section for more information.

Setting up the `__init__.py` file

Inside the folder there should be a `__init__.py` and one or more files containing code to parse the data for a given task.

The file contains information about the urls, tasks and keywords of the dataset. To set up these configs you need to import a *BaseDataset* class to define these parameters. To set this class you can use the following setup:

```
"""
Brief description of the dataset
"""

from dbcollection.datasets import BaseDataset
from .taskfile import TaskName

urls = ('http://url.something',)
keywords = ('some', 'keywords', 'for', 'grouping')
tasks = {'some_task_name': TaskName}
default_task = 'some_task_name'

class Dataset(BaseDataset):
    """Name of the dataset."""
    urls = urls
    keywords = keywords
    tasks = tasks
    default_task = default_task
```

To setup the new dataset's class you need to:

1. Write a short docstring indicating the purpose of the dataset (for image classification, object detection, action recognition, natural language processing, etc.);
2. Import `BaseDataset` from `dbcollection.datasets`;
3. Import the task classes from their corresponding files. In the above example, the task `TaskName` was imported from `taskfile.py` which must be located in the same directory as the `__init__.py` file;
4. Define a class called `Dataset` that inherits from `BaseDataset`. (It is required to name the class as `Dataset` in order for it to be included in the dataset list along with the rest of the other datasets);
5. Define a docstring for the class with the name of the dataset;
6. Set the `urls`, `keywords`, `tasks` and `default_task` fields. The `urls` field contains a list of urls of the source data files to be downloaded. The `keywords` field contains a list of strings which are used for helping in searching/grouping datasets for cataloging the existing datasets and their functionality. The `tasks` field is a dictionary containing all available tasks constructors for the dataset, and the `default_task` field indicates which task should be loaded as the default if no name is set when loading/processing a dataset.

Note: The task(s) file(s) does/do not require to have the same name of the task, but it is best practice to use the same name as the task to avoid confusion.

Additional information about setting up URLs for different sources

When configuring an url for download, you can either specify a string or a dict.

If using a string, the url of the file to download will be stored with the same filename as the url. Lets take the case of downloading data files for the `cifar10` dataset. To download the data file, you simply need to set the urls list as the following example:

```
urls = ('https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz',)
```

This will download the above url and save it to disk as `cifar-10-python.tar.gz`.

To download more files simply add url strings to the list and they will be sequentially downloaded and stored to disk.

We could write the previous example as using a dictionary instead of a string:

```
urls = ({'url': 'https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz'},)
```

This way contains additional options that you can use for dealing with urls. For example, if a MD5 hash checksum is available, you can use it to validate the integrity of the downloaded file:

```
urls = ({'url': 'https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz',
         'md5hash': 'c58f30108f718f92721af3b95e74349a'},)
```

We could also change the name of the saved filename. For that, we need to set a `save_name` field with the name of the file we would like to store the url data to disk.

```
urls = ({'url': 'https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz',
         'md5hash': 'c58f30108f718f92721af3b95e74349a',
         'save_name': 'cifar-10.tar.gz'},)
```

In some cases, for example, due to file names clashing, you may need to extract the downloaded url files into a different directory instead of the directory where the data file is stored. You can use the `extract_dir` field to specify a child directory name to extract these files into a separate directory in order to avoid the previous problem:

```
urls = ({'url': 'https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz',
         'md5hash': 'c58f30108f718f92721af3b95e74349a',
         'save_name': 'cifar-10.tar.gz',
         'extract_dir': 'extract_cifar10'},)
```

Another use case you may want to know is how to download url files from google drive. Fortunately for you, this is implemented here via the `googledrive` field. Note that this requires you to specify a filename in `save_name`. For example, downloading an url from google drive you can do the following:

```
urls = ({'googledrive': '0B4K3PZp8xXDJN0Fpb0piVjQ3Y3M',
         'save_name': 'flic.zip'},)
```

Note: For more information about downloading urls see the *Url download* section in the Reference manual.

Creating a task for parsing annotations

To create a script to parse / process annotations for a specific task you need to create a file (or as many files as you need) in the same directory as the `__init__.py` file.

Here you'll load and process all your annotations and define how data will be stored in the `HDF5` metadata file. The following template shows a basic setup of such task file which you can use as guidance when creating your own. You should also take a look at how other datasets' tasks are setup in order to have a better grasp on how to setup yours.

```
"""
Name of the dataset and the task
"""


# import necessary packages here
from __future__ import print_function, division  # for python 2.7 compatibility
import os
import numpy as np
...

# import the BaseTask class for inheriting its methods
from dbcollection.datasets import BaseTask

# import additional utility methods for parsing strings, padding lists or storing data
from dbcollection.utils.string_ascii import convert_str_to_ascii as str2ascii
from dbcollection.utils.pad import pad_list
from dbcollection.utils.hdf5 import hdf5_write_data


class Classification(BaseTask):
    """Name of the dataset + task."""

    # metadata filename of the task
    filename_h5 = 'classification'

    # Main method to load data from files
    def load_data(self):
        """
        Load the data from the files.
        """
        # Load annotations or setup other data fields here

    # Main method to store data to the HDF5 metadata file
    def add_data_to_default(self, hdf5_handler, data, set_name=None):
        """
        Add data of a set to the default group.

        For each field, the data is organized into a single big matrix.
        """
        # Store metadata here

    # optional method for storing data in the raw format (can leave as blank)
    def add_data_to_source(self, hdf5_handler, data, set_name=None):
        """
        Store data annotations in a nested tree fashion.

        It closely follows the tree structure of the data.
        """
        pass
```

Note: Check out how `cifar10` or `mnist` are setup for a basic script on how to parse annotations and structure how

data is stored in the HDF5 metadata file.

How to store data into HDF5 files

One important note about storing data into HDF5 files is how to do it. Due to the way the **dbcollection's** API is defined, you must need to store all information of a field into a single array. This way, each row indicated a sample and the columns the structure of the data itself.

For most cases, you will need to pad data in order to have arrays of the same shape. This has been discussed in *this section here* which you should take a look if you have questions about (un)padding data. Also, you should take a look at *Padding* section in the Reference manual.

Another relevant information to mention is how to save the data fields into the HDF5 file. You can either use h5py syntax to allocate the data fields in the right position or you can use the *hdf5_write_data()* from dbcollection.utils.hdf5 to simplify this process.

Besides these utility methods there are other useful ones in dbcollection.utils module that you should take a look when creating your own task, specially the *utils* section in the Reference manual for a list of available methods.

Writting the README.rst documentation file

When creating a dataset, it is very important to provide a small documentation of the structure and information of the data.

When creating such manual, it is good practice to follow a common format to keep things consistent.

The following scheme details a template format on how to write a README.rst documentation file in the reStructuredText format for a new dataset and what information to provide to the end user in order to describe how to use it.

```
.. _<dataset_name>_readme:

==============
<dataset_name>
==============

Brief description of the dataset's format / main features.


Use cases
=========

Main use cases.

(E.g., Image classification.)


Properties
==========

- ``name``: <dataset_name> (same as the directory)
- ``keywords``: "some", "keywords".
- ``dataset size``: size of the source data files in disk (e.g., 11,6 MB)
- ``is downloadable``: **yes**, **no** or **partial**
- ``tasks``:
```

(continues on next page)

(continued from previous page)

```
    - :ref:`<task_name1> <link_task1>`: **(default)**  <-- indicates that it is the␣
↪default task
        - ``primary use``: main use case (e.g., image classification)
        - ``description``: brief description of the annotations available
        - ``sets``: list of set splits (e.g., train, test)
        - ``metadata file size in disk``: size of the task's metadata file in disk (e.
↪g., 6,8 MB)
        - ``has annotations``: **yes** or **no**
            - ``which``:
                - brief description of the annotation (e.g., labels for each image␣
↪class/category.)
                ...

    - :ref:`<task_name2> <link_task2>`:
       ...
    - :ref:`<task_name3> <link_task3>`:
       ...


Metadata structure (HDF5)
=========================

.. _link_task1::

Task: <task_name1>
--------------------

::

    /
    ├── <set1>/
    │   ├── field1         # dtype=np.uint8, shape=(10,2)    (note: string in ASCII␣
↪format)
    │   ├── field2         # dtype=np.uint8, shape=(60000,28,28)
    │   ├── field3         # dtype=np.uint8, shape=(60000,)
    │   ├── object_fields  # dtype=np.uint8, shape=(2,7)     (note: string in ASCII␣
↪format)
    │   ├── object_ids     # dtype=np.int32, shape=(60000,2)
    │   └── list_field1_per_field2   # dtype=np.int32, shape=(10,6742))
    │
    └── <set2>/
        ├── field1         # dtype=np.uint8, shape=(10,2)    (note: string in ASCII␣
↪format)
        ├── field2         # dtype=np.uint8, shape=(10000,28,28)
        ├── field3         # dtype=np.uint8, shape=(10000,)
        ├── object_fields  # dtype=np.uint8, shape=(2,7)     (note: string in ASCII␣
↪format)
        ├── object_ids     # dtype=np.int32, shape=(10000,2)
        └── list_field1_per_field2   # dtype=np.int32, shape=(10,1742))


Fields
^^^^^^

- ``<field1>``: <description of the field> (e.g., class names)
    - ``available in``: <sets> (e.g., train, test)
    - ``dtype``: <numpy data type> (e.g., np.uint8)
```

(continues on next page)

(continued from previous page)

```
    - ``is padded``: True or False
    - ``fill value``: 0 , 1, -1, etc.
    - ``note``: an important note about this data field ( e.g., strings stored in␣
↪ASCII format)
- ``<field2>``: <description of the field>
    - ``available in``: <sets>
    - ``dtype``: <numpy data type>
    - ``is padded``: True or False
    - ``fill value``: 0 , 1, -1, etc.
    - ``note``: pre-ordered list (another example)
- ...

Disclaimer
==========

Disclaimer about the creators of the dataset.

Info of the website where the dataset was retrieved from.
It should containg a link to the original website/source.
```

1.1.8 core

Core API methods/classes.

Dataset management

API methods for managing datasets.

This module contains several methods for easy management of datasets. These include methods for:

- downloading datasets' data files from online sources (urls)
- processing/parsing data files + annotations into a HDF5 file to store metadata information
- loading dataset's metadata into a data loader object
- add/remove datasets to/from cache
- managing the cache file
- querying the cache file for some dataset/keyword
- displaying information about available datasets in cache or for download

These methods compose the core API for dealing with dataset management. Users should be able to take advantage of most functionality by using only these functions to manage and query their datasets in a simple and easy way.

Methods

download

dbcollection.core.api.download.**download**(*name*, *data_dir=''*, *extract_data=True*, *verbose=True*)

Download a dataset data to disk.

This method will download a dataset's data files to disk. After download, it updates the cache file with the dataset's name and path where the data is stored.

Parameters

- **name** (*str*) – Name of the dataset.
- **data_dir** (*str, optional*) – Directory path to store the downloaded data.
- **extract_data** (*bool, optional*) – Extracts/unpacks the data files (if true).
- **verbose** (*bool, optional*) – Displays text information (if true).

Examples

Download the CIFAR10 dataset to disk.

```
>>> import dbcollection as dbc
>>> dbc.download('cifar10')
```

process

dbcollection.core.api.process.**process**(*name*, *task='default'*, *verbose=True*)

Process a dataset's metadata and stores it to file.

The data is stored a a HSF5 file for each task composing the dataset's tasks.

Parameters

- **name** (*str*) – Name of the dataset.
- **task** (*str, optional*) – Name of the task to process.
- **verbose** (*bool, optional*) – Displays text information (if true).

Raises KeyError – If a task does not exist for a dataset.

Examples

```
>>> import dbcollection as dbc
```

Download the CIFAR10 dataset to disk.

```
>>> dbc.process('cifar10', task='classification', verbose=False)
```

load

dbcollection.core.api.load.**load**(*name*, *task='default'*, *data_dir=''*, *verbose=True*)

Returns a metadata loader of a dataset.

Returns a loader with the necessary functions to manage the selected dataset.

Parameters

- **name** (*str*) – Name of the dataset.
- **task** (*str, optional*) – Name of the task to load.

- **data_dir** (*str, optional*) – Directory path to store the downloaded data.
- **verbose** (*bool, optional*) – Displays text information (if true).

Returns Data loader class.

Return type *DataLoader*

Raises Exception – If dataset is not available for loading.

Examples

Load the MNIST dataset.

```
>>> import dbcollection as dbc
>>> mnist = dbc.load('mnist')
>>> print('Dataset name: ', mnist.db_name)
Dataset name:  mnist
```

add

dbcollection.core.api.add.**add**(*name*, *task*, *data_dir*, *hdf5_filename*, *categories=()*, *verbose=True*, *force_overwrite=False*)

Add a dataset/task to the list of available datasets for loading.

Parameters

- **name** (*str*) – Name of the dataset.
- **task** (*str*) – Name of the task to load.
- **data_dir** (*str*) – Path of the stored data in disk.
- **hdf5_filename** (*str*) – Path to the metadata HDF5 file.
- **categories** (*list, optional*) – List of keyword strings to categorize the dataset.
- **verbose** (*bool, optional*) – Displays text information (if true).
- **force_overwrite** (*bool, optional*) – Forces the overwrite of data in cache

Examples

Add a dataset manually to dbcollection.

```
>>> import dbcollection as dbc
>>> dbc.add('new_db', 'new_task', 'new/path/db', 'newdb.h5', ['new_category'])
>>> dbc.query('new_db')
{'new_db': {'tasks': {'new_task': 'newdb.h5'}, 'data_dir': 'new/path/db',
↪'keywords':
['new_category']}}
```

remove

dbcollection.core.api.remove.**remove**(*name*, *task=''*, *delete_data=False*, *verbose=True*)

Remove/delete a dataset and/or task from the cache.

Removes the dataset's information registry from the dbcollection.json cache file. The dataset's data files remain in disk if 'delete_data' is not enabled. If you intended to remove the data files as well, the 'delete_data' input arg must be set to 'True' in order to also remove the data files.

Moreover, if the intended action is to remove a task of the dataset from the cache registry, this can be achieved by specifying the task name to be deleted. This effectively removes only that task entry for the dataset. Note that deleting a task results in removing the associated HDF5 metadata file from disk.

Parameters

- **name** (*str*) – Name of the dataset to delete.
- **task** (*str, optional*) – Name of the task to delete.
- **delete_data** (*bool, optional*) – Delete all data files from disk for this dataset if True.
- **verbose** (*bool, optional*) – Displays text information (if true).

Examples

Remove a dataset from the list.

```
>>> import dbcollection as dbc
>>> # add a dataset
>>> dbc.add('new_db', 'new_task', 'new/path/db', 'newdb.h5', ['new_category'])
>>> dbc.query('new_db')
{'new_db': {'tasks': {'new_task': 'newdb.h5'}, 'data_dir': 'new/path/db',
'keywords': ['new_category']}}
>>> dbc.remove('new_db')  # remove the dataset
Removed 'new_db' dataset: cache=True, disk=False
>>> dbc.query('new_db')  # check if the dataset info was removed (retrieves an␣
↪empty dict)
{}
```

config_cache

query

info_cache

info_datasets

fetch_list_datasets

Classes

DownloadAPI

class dbcollection.core.api.download.**DownloadAPI**(*name*, *data_dir*, *extract_data*, *verbose*)

Dataset download API class.

This class contains methods to correctly download a dataset's data files to disk.

Parameters

- **name** (*str*) – Name of the dataset.
- **data_dir** (*str*) – Directory path to store the downloaded data.
- **extract_data** (*bool*) – Extracts/unpacks the data files (if true).
- **verbose** (*bool*) – Displays text information (if true).

Variables

- **name** (*str*) – Name of the dataset.
- **data_dir** (*str*) – Directory path to store the downloaded data.
- **save_data_dir** (*str*) – Data files save dir path.
- **save_cache_dir** (*str*) – Cache save dir path.
- ***extract_data*** (*bool*) – Flag to extract data (if True).
- **verbose** (*bool*) – Flag to display text information (if true).
- **cache_manager** (*CacheManager*) – Cache manager object.

create_dir(*path*)
Create a directory in the disk.

download_dataset()
Download the dataset to disk.

get_download_data_dir_from_cache()
Create a dir path from the cache information for this dataset.

run()
Main method.

update_cache()
Update the cache manager information for this dataset.

ProcessAPI

class dbcollection.core.api.process.**ProcessAPI**(*name*, *task*, *verbose*)
Dataset metadata process API class.

This class contains methods to correctly process the dataset's data files and convert their metadata to disk.

Parameters

- **name** (*str*) – Name of the dataset.
- **task** (*str*) – Name of the task to process.
- **verbose** (*bool*) – Displays text information (if true).

Variables

- **name** (*str*) – Name of the dataset.
- **task** (*str*) – Name of the task to process.
- **verbose** (*bool*) – Displays text information (if true).
- ***extract_data*** (*bool*) – Flag to extract data (if True).
- **cache_manager** (*CacheManager*) – Cache manager object.

Raises `KeyError` – If a task does not exist for a dataset.

check_if_task_exists_in_database(*task*)
Check if task exists in the list of available tasks for processing.

create_dir(*path*)
Create a directory in the disk.

exists_task(*task*)
Checks if a task exists for a dataset.

get_default_task()
Returns the default task for this dataset.

parse_task_name(*task*)
Parse the input task string.

process_dataset()
Process the dataset's metadata.

run()
Main method.

update_cache(*task_info*)
Update the cache manager information for this dataset.

LoadAPI

class dbcollection.core.api.load.**LoadAPI**(*name*, *task*, *data_dir*, *verbose*)
Dataset load API class.

This class contains methods to correctly load a dataset's metadata as a data loader object.

Parameters

- **name** (*str*) – Name of the dataset.
- **task** (*str*) – Name of the task to load.
- **data_dir** (*str*) – Directory path to store the downloaded data.
- **verbose** (*bool*) – Displays text information (if true).

Variables

- **name** (*str*) – Name of the dataset.
- **task** (*str*) – Name of the task to load.
- **data_dir** (*str*) – Directory path to store the downloaded data.
- **verbose** (*bool*) – Displays text information (if true).
- **cache_manager** (*CacheManager*) – Cache manager object.
- **available_datasets_list** (*list*) – List of available datast names for download.

download_dataset()
Download the dataset to disk.

get_data_loader()
Return a DataLoader object.

parse_task_name(*task*)
Validate the task name.

process_dataset()
Process the dataset's metadata.

run()
Main method.

AddAPI

class dbcollection.core.api.add.**AddAPI**(*name*, *task*, *data_dir*, *hdf5_filename*, *categories*, *verbose*, *force_overwrite*)
Add dataset API class.

This class contains methods to correctly register a dataset in the cache.

Parameters

- **name** (*str*) – Name of the dataset.
- **task** (*str*) – Name of the task to load.
- **data_dir** (*str*) – Path of the stored data in disk.
- **hdf5_filename** (*str*) – Path to the metadata HDF5 file.
- **categories** (*tuple*) – List of keyword strings to categorize the dataset.
- **verbose** (*bool*) – Displays text information.
- **force_overwrite** (*bool*) – Forces the overwrite of data in cache

Variables

- **name** (*str*) – Name of the dataset.
- **task** (*str*) – Name of the task to load.
- **data_dir** (*str*) – Path of the stored data in disk.
- **hdf5_filename** (*bool*) – Path to the metadata HDF5 file.
- **categories** (*tuple*) – Tuple of keyword strings to categorize the dataset.
- **verbose** (*bool*) – Displays text information.
- **force_overwrite** (*bool*) – Forces the overwrite of data in cache
- **cache_manager** (*CacheManager*) – Cache manager object.

run()
Main method.

RemoveAPI

class dbcollection.core.api.remove.**RemoveAPI**(*name*, *task*, *delete_data*, *verbose*)
Dataset remove API class.

This class contains methods to remove a dataset registry from cache. Also, it can remove the dataset's files from disk if needed.

Parameters

- **name** (*str*) – Name of the dataset to delete.
- **task** (*str, optional*) – Name of the task to delete.

- **delete_data** (*bool*) – Delete all data files from disk for this dataset if True.

Variables

- **name** (*str*) – Name of the dataset to delete.
- **task** (*str*) – Name of the task to delete.
- **delete_data** (*bool*) – Delete all data files from disk for this dataset if True.
- **cache_manager** (*CacheManager*) – Cache manager object.

exists_dataset()
Return True if a dataset name exists in the cache.

print_msg_registry_removal()
Prints to screen the success message.

remove_dataset()
Removes the dataset from cache (and disk if selected).

remove_dataset_data_files_from_disk()
Removes the directory containing the data files from disk.

remove_dataset_entry_from_cache()
Removes the dataset registry from cache.

remove_dataset_registry()
Removes the dataset registry from cache.

remove_registry_from_cache()
Remove the dataset or task from cache.

remove_task_registry()
Remove the task registry for this dataset from cache.

run()
Main method.

ConfigAPI

QueryAPI

InfoCacheAPI

InfoDatasetAPI

Cache management

CacheManager

Data loading

Dataset's metadata loader classes.

DataLoader

class dbcollection.core.loader.**DataLoader**(*name*, *task*, *data_dir*, *hdf5_filepath*)

Dataset metadata loader class.

This class contains several methods to fetch data from a hdf5 file by using simple, easy to use functions for (meta)data handling.

Parameters

- **name** (*str*) – Name of the dataset.
- **task** (*str*) – Name of the task.
- **data_dir** (*str*) – Path of the dataset's data directory on disk.
- **hdf5_filepath** (*str*) – Path of the metadata cache file stored on disk.

Variables

- **db_name** (*str*) – Name of the dataset.
- **task** (*str*) – Name of the task.
- **data_dir** (*str*) – Path of the dataset's data directory on disk.
- **hdf5_filepath** (*str*) – Path of the hdf5 metadata file stored on disk.
- **hdf5_file** (*h5py._hl.files.File*) – hdf5 file object handler.
- **root_path** (*str*) – Default data group of the hdf5 file.
- **sets** (*tuple*) – List of names of set splits (e.g. train, test, val, etc.)
- **object_fields** (*dict*) – Data field names for each set split.

get(*set_name*, *field*, *index=None*, *convert_to_str=False*)

Retrieves data from the dataset's hdf5 metadata file.

This method retrieves the i'th data from the hdf5 file with the same 'field' name. Also, it is possible to retrieve multiple values by inserting a list/tuple of number values as indexes.

Parameters

- **set_name** (*str*) – Name of the set.
- **field** (*str*) – Name of the data field.
- **idx** (*int/list/tuple, optional*) – Index number of the field. If it is a list, returns the data for all the value indexes of that list.
- **convert_to_str** (*bool, optional*) – Convert the output data into a string. Warning: output must be of type np.uint8

Returns Numpy array containing the field's data. If convert_to_str is set to True, it returns a string or list of strings.

Return type np.ndarray/list/str

Raises KeyError – If set name is not valid or does not exist.

info(*set_name=None*)

Prints information about all data fields of a set.

Displays information of all fields of a set group inside the hdf5 metadata file. This information contains the name of the field, as well as the size/shape of the data, the data type and if the field is contained in the 'object_ids' list.

If no 'set_name' is provided, it displays information for all available sets.

This method only shows the most useful information about a set/fields internals, which should be enough for most users in helping to determine how to use/handle a specific dataset with little effort.

Parameters **set_name** (*str, optional*) – Name of the set.

Raises KeyError – If set name is not valid or does not exist.

list (*set_name=None*)
List of all field names of a set.

Parameters **set_name** (*str, optional*) – Name of the set.

Returns List of all data fields of the dataset.

Return type list/dict

Raises KeyError – If set name is not valid or does not exist.

object (*set_name*, *index=None*, *convert_to_value=False*)
Retrieves a list of all fields' indexes/values of an object composition.

Retrieves the data's ids or contents of all fields of an object.

It basically works as calling the get() method for each individual field and then groups all values into a list w.r.t. the corresponding order of the fields.

Parameters

- **set_name** (*str*) – Name of the set.
- **index** (*int/list/tuple, optional*) – Index number of the field. If it is a list, returns the data for all the value indexes of that list. If no index is used, it returns the entire data field array.
- **convert_to_value** (*bool, optional*) – If False, outputs a list of indexes. If True, it outputs a list of arrays/values instead of indexes.

Returns List of indexes of the data fields available in 'object_fields'. If convert_to_value is set to True, it returns a list of data instead of indexes.

Return type list

Raises KeyError – If set name is not valid or does not exist.

object_field_id (*set_name*, *field*)
Retrieves the index position of a field in the 'object_ids' list.

This method returns the position of a field in the 'object_ids' object. If the field is not contained in this object, it returns a null value.

Parameters

- **set_name** (*str*) – Name of the set.
- **field** (*str*) – Name of the field in the metadata file.

Returns Index of the field in the 'object_ids' list.

Return type int

Raises KeyError – If set name is not valid or does not exist.

size (*set_name=None*, *field='object_ids'*)
Size of a field.

Returns the number of the elements of a field.

Parameters

- **set_name** (*str, optional*) – Name of the set.
- **field** (*str, optional*) – Name of the field in the metadata file.

Returns Returns the size of a field.

Return type list/dict

Raises KeyError – If set name is not valid or does not exist.

SetLoader

class dbcollection.core.loader.**SetLoader**(*hdf5_group*)
Set metadata loader class.

This class contains several methods to fetch data from a specific set (group) in a hdf5 file. It contains useful information about a specific group and also several methods to fetch data.

Parameters **hdf5_group** (*h5py._hl.group.Group*) – hdf5 group object handler.

Variables

- **hdf5_group** (*h5py._hl.group.Group*) – hdf5 group object handler.
- **set** (*str*) – Name of the set.
- **fields** (*tuple*) – List of all field names of the set.
- **object_fields** (*tuple*) – List of all field names of the set contained by the 'object_ids' list.
- **nelems** (*int*) – Number of rows in 'object_ids'.

__len__()

Returns Number of elements

Return type int

get(*field*, *index=None*, *convert_to_str=False*)
Retrieves data from the dataset's hdf5 metadata file.

This method retrieves the i'th data from the hdf5 file with the same 'field' name. Also, it is possible to retrieve multiple values by inserting a list/tuple of number values as indexes.

Parameters

- **field** (*str*) – Field name.
- **index** (*int/list/tuple, optional*) – Index number of the field. If it is a list, returns the data for all the value indexes of that list.
- **convert_to_str** (*bool, optional*) – Convert the output data into a string. Warning: output must be of type np.uint8

Returns Numpy array containing the field's data. If convert_to_str is set to True, it returns a string or list of strings.

Return type np.ndarray/list/str

Raises KeyError – If the field does not exist in the list.

info()
Prints information about the data fields of a set.

Displays information of all fields available like field name, size and shape of all sets. If a 'set_name' is provided, it displays only the information for that specific set.

This method provides the necessary information about a data set internals to help determine how to use/handle a specific field.

list()
List of all field names.

Returns List of all data fields of the dataset.

Return type list

object(*index=None*, *convert_to_value=False*)
Retrieves a list of all fields' indexes/values of an object composition.

Retrieves the data's ids or contents of all fields of an object.

It basically works as calling the get() method for each individual field and then groups all values into a list w.r.t. the corresponding order of the fields.

Parameters

- **index** (*int/list/tuple, optional*) – Index number of the field. If it is a list, returns the data for all the value indexes of that list. If no index is used, it returns the entire data field array.
- **convert_to_value** (*bool, optional*) – If False, outputs a list of indexes. If True, it outputs a list of arrays/values instead of indexes.

Returns Returns a list of indexes or, if convert_to_value is True, a list of data arrays/values.

Return type list

object_field_id(*field*)
Retrieves the index position of a field in the 'object_ids' list.

This method returns the position of a field in the 'object_ids' object. If the field is not contained in this object, it returns a null value.

Parameters **field** (*str*) – Name of the field in the metadata file.

Returns Index of the field in the 'object_ids' list.

Return type int

Raises KeyError – If field does not exists in the list of object fields.

size(*field='object_ids'*)
Size of a field.

Returns the number of the elements of a field.

Parameters **field** (*str, optional*) – Name of the field in the metadata file.

Returns Returns the size of the field.

Return type tuple

Raises KeyError – If field is invalid or does not exist in the fields dict.

FieldLoader

class dbcollection.core.loader.**FieldLoader**(*hdf5_field*, *obj_id=None*)
Field metadata loader class.

This class contains several methods to fetch data from a specific field of a set (group) in a hdf5 file. It contains useful information about the field and also several methods to fetch data.

Parameters

- **hdf5_field** (*h5py._hl.dataset.Dataset*) – hdf5 field object handler.
- **obj_id** (*int, optional*) – Position of the field in 'object_fields'.

Variables

- **data** (*h5py._hl.dataset.Dataset*) – hdf5 group object handler.
- **set** (*str*) – Name of the set.
- **name** (*str*) – Name of the field.
- **type** (*type*) – Type of the field's data.
- **shape** (*tuple*) – Shape of the field's data.
- **fillvalue** (*int*) – Value used to pad arrays when storing the data in the hdf5 file.
- **obj_id** (*int*) – Identifier of the field if contained in the 'object_ids' list.

__getitem__(*index*)

Parameters **index** (*int*) – Index

Returns Numpy data array.

Return type np.ndarray

__len__()

Returns Number of samples

Return type int

get(*index=None*, *convert_to_str=False*)
Retrieves data of the field from the dataset's hdf5 metadata file.

This method retrieves the i'th data from the hdf5 file. Also, it is possible to retrieve multiple values by inserting a list/tuple of number values as indexes.

Parameters

- **index** (*int/list/tuple, optional*) – Index number of he field. If it is a list, returns the data for all the value indexes of that list.
- **convert_to_str** (*bool, optional*) – Convert the output data into a string. Warning: output must be of type np.uint8

Returns Numpy array containing the field's data. If convert_to_str is set to True, it returns a string or list of strings.

Return type np.ndarray/list/str

Note: When using lists/tuples of indexes, this method sorts the list and removes duplicate values. This is because the h5py api requires the indexing elements to be in increasing order when retrieving data.

---

info(*verbose=True*)
Prints information about the field.

Displays information like name, size and shape of the field.

Parameters **verbose** (*bool, optional*) – If true, display extra information about the field.

object_field_id()
Retrieves the index position of the field in the 'object_ids' list.

This method returns the position of the field in the 'object_ids' object. If the field is not contained in this object, it returns a null value.

Returns Index of the field in the 'object_ids' list.

Return type int

size()
Size of the field.

Returns the number of the elements of the field.

Returns Returns the size of the field.

Return type tuple

to_memory
Modifies how data is accessed and stored.

Accessing data from a field can be done in two ways: memory or disk. To enable data allocation and access from memory requires the user to specify a boolean. If set to True, data is allocated to a numpy ndarray and all accesses are done in memory. Otherwise, data is kept in disk and accesses are done using the HDF5 object handler.

1.1.9 datasets

This module contains scripts to download/process all datasets available in dbcollection.

These scripts are self contained, meaning they can be imported and used to manually setup a dataset.

Constructors: Classes

BaseDataset

class dbcollection.datasets.**BaseDataset**(*data_path*, *cache_path*, *extract_data=True*, *verbose=True*)
Base class for download/processing a dataset.

Parameters

- **data_path** (*str*) – Path to the data directory.
- **cache_path** (*str*) – Path to the cache file
- **extract_data** (*bool, optional*) – Extracts the downloaded files if they are compacted.
- **verbose** (*bool*) – Be verbose

Variables

- **data_path** (*str*) – Path to the data directory.
- **cache_path** (*str*) – Path to the cache file
- ***extract_data*** (*bool, optional*) – Extracts the downloaded files if they are compacted.
- **verbose** (*bool*) – Be verbose
- **urls** (*list*) – List of URL links to download.
- **keywords** (*list*) – List of keywords.
- **tasks** (*dict*) – Dataset's tasks.
- **default_task** (*str*) – Default task name.

download()
Download and extract files to disk.

Returns A list of keywords.

Return type tuple

get_task_constructor(*task*)
Returns the class constructor for the input task.

Parameters **task** (*str*) – Task name.

Returns

- *str* – Task name.
- *str* – Task's ending suffix (if any).
- *BaseTask* – Constructor to process the metadata of a task.

parse_task_name(*task*)
Parses the task string to look for key suffixes.

Parameters **task** (*str*) – Task name.

Returns Returns a task name without the '_s' suffix.

Return type str

process(*task='default'*)
Processes the metadata of a task.

Parameters **task** (*str, optional*) – Task name.

Returns Returns a dictionary with the task name as key and the filename as value.

Return type dict

BaseTask

class dbcollection.datasets.**BaseTask**(*data_path*, *cache_path*, *suffix=None*, *verbose=True*)
Base class for processing a task of a dataset.

Parameters

- **data_path** (*str*) – Path to the data directory.
- **cache_path** (*str*) – Path to the cache file
- **suffix** (*str, optional*) – Suffix to select optional properties for a task.

- **verbose** (*bool, optional*) – Be verbose.

Variables

- **data_path** (*str*) – Path to the data directory.
- **cache_path** (*str*) – Path to the cache file
- **suffix** (*str, optional*) – Suffix to select optional properties for a task.
- **verbose** (*bool, optional*) – Be verbose.
- **filename_h5** (*str*) – hdf5 metadata file name.

add_data_to_default(*handler*, *data*, *set_name=None*)
Add data of a set to the default group.

For each field, the data is organized into a single big matrix.

Parameters

- **hdf5_handler** (*h5py._hl.group.Group*) – hdf5 group object handler.
- **data** (*list/dict*) – List or dict containing the data annotations of a particular set or sets.
- **set_name** (*str*) – Set name.

add_data_to_source(*hdf5_handler*, *data*, *set_name=None*)
Store data annotations in a nested tree fashion.

It closely follows the tree structure of the data.

Parameters

- **hdf5_handler** (*h5py._hl.group.Group*) – hdf5 group object handler.
- **data** (*list/dict*) – List or dict containing the data annotations of a particular set or sets.
- **set_name** (*str*) – Set name.

load_data()
Load data of the dataset (create a generator).

Load data from annnotations and split it to corresponding sets (train, val, test, etc.)

process_metadata()
Process metadata and store it in a hdf5 file.

run()
Run task processing.

1.1.10 utils

Utility methods for url download, file extraction, data padding and parsing, testing, etc.

Also, all third-party submodules are located under this module.

URL download

Download functions.

dbcollection.utils.url.**check_if_url_files_exist**(*urls*, *save_dir*)
Evaluates if all url filenames exist on disk.

Parameters

- **urls** (*list/tuple/dict*) – URL paths.
- **dir_save** (*str*) – Directory to store the downloaded data.

dbcollection.utils.url.**download_extract_urls**(*urls*, *save_dir*, *extract_data=True*, *verbose=True*)
Download urls + extract files to disk.

Parameters

- **urls** (*list/tuple/dict*) – URL paths.
- **dir_save** (*str*) – Directory to store the downloaded data.
- **extract_data** (*bool, optional*) – Extracts/unpacks the data files (if true).
- **verbose** (*bool, optional*) – Display messages on screen if set to True.

dbcollection.utils.url.**extract_archive_file**(*filename*, *save_dir*)
Extracts a file archive's data to a directory.

Parameters

- **filename** (*str*) – File name + path of the archive file.
- **dir_save** (*str*) – Directory to extract the file archive.

class dbcollection.utils.url.**URL**
URL manager class.

class dbcollection.utils.url.**URLDownload**
Download an URL using the requests module.

class dbcollection.utils.url.**URLDownloadGoogleDrive**
Download an URL from Google Drive.

File loading

Library to load different types of file into memory.

dbcollection.utils.file_load.**load_json**(*fname*)
Loads a json file to memory.

Parameters **fname** (*str*) – File name + path.

Returns Data structure of the input json file.

Return type dict/list

dbcollection.utils.file_load.**load_matlab**(*fname*)
Loads a matlab file to memory.

Parameters **fname** (*str*) – File name + path.

Returns Data structure of the input matlab file.

Return type dict/list

dbcollection.utils.file_load.**load_pickle**(*fname*)
Loads a pickle file to memory.

Parameters **fname** (*str*) – File name + path.

Returns Data structure of the input file.

Return type dict/list

dbcollection.utils.file_load.**load_txt**(*fname*, *mode='r'*)
Loads a .txt file to memory.

Parameters

- **fname** (*str*) – File name + path.
- **mode** (*str, optional*) – File open mode.

Returns

Return type list of strings

dbcollection.utils.file_load.**load_xml**(*fname*)
Loads and parses a xml file to a dictionary.

Parameters **fname** (*str*) – File name + path.

Returns Dictionary of the input file's data structure.

Return type dict

Padding

Library of methods for padding/unpadding lists or lists of lists with fill values.

dbcollection.utils.pad.**pad_list**(*listA*, *val=-1*, *length=None*)
Pad list of lists with 'val' such that all lists have the same length.

Parameters

- **listA** (*list*) – List of lists of different sizes.
- **val** (*number, optional*) – Value to pad the lists.
- **length** (*number, optional*) – Total length of the list.

Returns A list of lists with the same same.

Return type list

Examples

Pad an uneven list of lists with a value.

```
>>> from dbcollection.utils.pad import pad_list
>>> pad_list([[0,1,2,3],[45,6],[7,8],[9]])  # pad with -1 (default)
[[0, 1, 2, 3], [4, 5, 6, -1], [7, 8, -1, -1], [9-1, -1, -1]]
>>> pad_list([[1,2],[3,4]])  # does nothing
[[1, 2], [3, 4]]
>>> pad_list([[],[1],[3,4,5]], 0)  # pad lists with 0
[[0, 0, 0], [1, 0, 0], [3, 4, 5]]
>>> pad_list([[],[1],[3,4,5]], 0, 6)  # pad lists with 0 of size 6
[[0, 0, 0, 0, 0, 0], [1, 0, 0, 0, 0, 0], [3, 4, 5, 0, 0, 0]]
```

dbcollection.utils.pad.**unpad_list**(*listA*, *val=-1*)
Unpad list of lists with which has values equal to 'val'.

Parameters

- **listA** (*list*) – List of lists of equal sizes.
- **val** (*number, optional*) – Value to unpad the lists.

Returns A list of lists without the padding values.

Return type list

Examples

Remove the padding values of a list of lists.

```
>>> from dbcollection.utils.pad import unpad_list
>>> unpad_list([[1,2,3,-1,-1],[5,6,-1,-1,-1]])
[[1, 2, 3], [5, 6]]
>>> unpad_list([[5,0,-1],[1,2,3,4,5]], 5)
[[0, -1], [1, 2, 3, 4]]
```

dbcollection.utils.pad.**squeeze_list**(*listA*, *val=-1*)

Compact a list of lists into a single list.

Squeezes (spaghettify) a list of lists into a single list. The lists are concatenated into a single one, and to separate them it is used a separating value to mark the split location when unsqueezing the list.

Parameters

- **listA** (*list*) – List of lists.
- **val** (*number, optional*) – Value to separate the lists.

Returns A list with all lists concatenated into one.

Return type list

Examples

Compact a list of lists into a single list.

```
>>> from dbcollection.utils.pad import squeeze_list
>>> squeeze_list([[1,2], [3], [4,5,6]], -1)
[1, 2, -1, 3, -1, 4, 5, 6]
```

dbcollection.utils.pad.**unsqueeze_list**(*listA*, *val=-1*)

Unpacks a list into a list of lists.

Returns a list of lists by splitting the input list into 'N' lists when encounters an element equal to 'val'. Empty lists resulting of trailing values at the end of the list are discarded.

Source: https://stackoverflow.com/questions/4322705/split-a-list-into-nested-lists-on-a-value

Parameters

- **listA** (*list*) – A list.
- **val** (*int/float, optional*) – Value to separate the lists.

Returns A list of lists.

Return type list

Examples

Unpack a list into a list of lists.

```
>>> from dbcollection.utils.pad import unsqueeze_list
>>> unsqueeze_list([1, 2, -1, 3, -1, 4, 5, 6], -1)
[[1, 2], [3], [4, 5, 6]]
```

String<->ASCII

String-to-ascii and ascii-to-string convertion methods.

dbcollection.utils.string_ascii.**convert_str_to_ascii**(*inp_str*)

Convert a list of strings into an ascii encoded numpy array.

Converts a string or list of strings to a numpy array. The array size is defined by the size of string plus one. This is needed for ascii to str convertion in lua using ffi.string() which expects a 0 at the end of an array.

If a list of strings is used, the size of the array is defined by the size of the longest string (plus one), and zero padded to maitain the array shape.

Parameters **inp_str** (*str/list/tuple*) – String or list of strings to convert to an ascii array.

Returns Single/multi-dimensional array of ASCII encoded strings.

Return type np.ndarray

Examples

Example1: Convert a string to a numpy array encoded into ASCII values.

```
>>> from dbcollection.utils.string_ascii import convertstr_to_ascii
>>> convertstr_to_ascii('string1')
array([115, 116, 114, 105, 110, 103,  49,   0], dtype=uint8)
```

Example2: Convert a list of lists into an ASCII array.

```
>>> from dbcollection.utils.string_ascii import convertstr_to_ascii
>>> convertstr_to_ascii(['string1', 'string2', 'string3'])
array([[115, 116, 114, 105, 110, 103,  49,   0],
    [115, 116, 114, 105, 110, 103,  50,   0],
    [115, 116, 114, 105, 110, 103,  51,   0]], dtype=uint8)
```

dbcollection.utils.string_ascii.**convert_ascii_to_str**(*input_array*)

Convert a numpy array to a string (or a list of strings)

Parameters **input_array** (*np.ndarray*) – Array of strings encoded in ASCII format.

Returns String or list of strings.

Return type str/list

Examples

Convert a numpy array to a string.

```
>>> from dbcollection.utils.string_ascii import convert_ascii_to_str
>>> import numpy as np
>>> # ascii format of 'string1'
>>> tensor = np.array([[115, 116, 114, 105, 110, 103, 49, 0]], dtype=np.uint8)
>>> convert_ascii_to_str(tensor)
['string1']
```

dbcollection.utils.string_ascii.**str_to_ascii**(*input_str*)

Converts a string to an ascii encoded numpy array.

Converts a single string of characters into a numpy array coded as ascii.

Parameters **input_str** (*str*) – String data.

Returns Uni-dimensional array of char values encoded in ASCII format.

Return type np.ndarray

Examples

Convert a string to numpy array.

```
>>> from dbcollection.utils.string_ascii import str_to_ascii
>>> str_to_ascii('string1')
array([115, 116, 114, 105, 110, 103,  49], dtype=uint8)
```

dbcollection.utils.string_ascii.**ascii_to_str**(*input_array*)

Converts an ascii encoded numpy array to a string.

Parameters **input_array** (*np.ndarray*) – Input array vector (should be of type dtype=numpy.uint8)

Returns Single string.

Return type str

Examples

Convert a numpy array to string.

```
>>> import numpy as np
>>> from dbcollection.utils.string_ascii import ascii_to_str
>>> ascii_to_str(np.array([115, 116, 114, 105, 110, 103,  49], dtype=uint8))
'string1'
```

HDF5

hdf5 utility functions.

dbcollection.utils.hdf5.**hdf5_write_data**(*h5_handler*, *field_name*, *data*, *dtype=None*, *chunks=True*, *compression='gzip'*, *compression_opts=4*, *fillvalue=-1*)

Write/store data into a hdf5 file.

Parameters

- **h5_handler** (*h5py._hl.group.Group*) – Handler for an HDF5 group object.

- **field_name** (*str*) – Field name.
- **data** (*np.ndarray*) – Data array.
- **dtype** (*np.dtype, optional*) – Data type.
- **chunks** (*bool, optional*) – Store data as chunks if True.
- **compression** (*str, optional*) – Compression algorithm type.
- **compression_opts** (*int, optional*) – Compression option (range: [1,10])
- **fillvalue** (*int/float, optional*) – Value to pad the data.

Returns Handler for an HDF5 dataset object.

Return type h5py._hl.dataset.Dataset

Dir db constructor

This module contains methods for parsing directories

dbcollection.utils.os_dir.**construct_dataset_from_dir**(*dir_path*, *verbose=True*)

Build a dataset from a directory.

This method creates a dataset from a root folder. The first child folders compose the dataset's partition into train/val/test/etc. Then, child folders of these compose the dataset's classes and all files inside correspond to the data.

Parameters

- **dir_path** (*str*) – Directory path to create the dataset structure from.
- **verbose** (*bool, optional*) – Prints messages to the screen (if True).

Returns Dataset structure.

Return type dict

dbcollection.utils.os_dir.**construct_set_from_dir**(*dir_path*, *verbose=True*)

Build a dataset from a directory.

This method creates a dataset from a root folder. The first child folders compose the dataset's classes and all files inside correspond to the data.

Parameters

- **dir_path** (*str*) – Directory path to create the set structure from.
- **verbose** (*bool, optional*) – Prints messages to the screen (if True).

Returns Set structure with keys as class names and values as image filenames.

Return type dict

dbcollection.utils.os_dir.**dir_get_size**(*dir_path*)

Returns the number of files and subfolders in a directory.

Parameters **dir_path** (*str*) – Directory path.

Returns

- *int* – Number of files in the folder.
- *int* – Number of folders in the path.

Test

Test utility functions/classes.

TestBaseDB

class dbcollection.utils.test.**TestBaseDB**(*name*, *task*, *data_dir*, *verbose=True*)

Test Class for loading datasets.

Parameters

- **name** (*str*) – Name of the dataset.
- **task** (*str*) – Name of the task.
- **data_dir** (*str*) – Path of the dataset's data directory on disk.
- **verbose** (*bool, optional*) – Be verbose.

Variables

- **name** (*str*) – Name of the dataset.
- **task** (*str*) – Name of the task.
- **data_dir** (*str*) – Path of the dataset's data directory on disk.
- **verbose** (*bool*) – Be verbose.

delete_cache()

Delete all cache data + dir

download(*extract_data=True*)

Download a dataset to disk.

Parameters **extract_data** (*bool*) – Flag signaling to extract data to disk (if True).

list_datasets()

Print dbcollection info

load()

Return a data loader object for a dataset.

Returns A data loader object of a dataset.

Return type *DataLoader*

print_info(*loader*)

Print information about the dataset to the screen

Parameters **loader** (DataLoader) – Data loader object of a dataset.

process()

Process dataset

run(*mode*)

Run the test script.

Parameters **mode** (*str*) – Task name to execute.

Raises Exception – If an invalid mode was inserted.

TestDatasetGenerator

Timeout

class dbcollection.utils.test.**Timeout**(*sec*)
Timeout class using ALARM signal.

exception Timeout

Third party modules

Third-party modules used by dbcollection.

caltech_pedestrian_extractor

Extract images (.seq to .jpg) and annotation files (.vbb to .json) from the Caltech Pedestrian Dataset.

dbcollection.utils.db.caltech_pedestrian_extractor.converter.**extract_data**(*data_path*, *save_path*, *sets=None*)
Extract image and annotation data from .vbb and .seq files.

Parameters

- **data_path** (*str*) – Directory path of data files.
- **save_path** (*str*) – Directory path to store the extracted data.
- **sets** (*str/list/tuple, optional*) – List of set names to extract.

Raises TypeError – If sets input arg is not a string, list or tuple.

1.1.11 Available datasets

Here you can find a list of all available datasets for load/download on this package. The majority of these datasets are for computer vision tasks, but other tasks such as natural language processing are being added to this list. If you have any suggestion for a well needed dataset, please feel free to write an issue on GitHub detailing the new proposal or submit a pull request with an implementation of your proposal. For more information about contributing to this project, please check out the Contributing section of the documentation.

The following list of datasets contains detailed information about how datasets are stored, along with other properties that you may find useful to know about when using them.

Caltech Pedestrian

The **Caltech Pedestrian Dataset consists** of approximately 10 hours of 640x480 30Hz video taken from a vehicle driving through regular traffic in an urban environment. About 250,000 frames (in 137 approximately minute long segments) with a total of 350,000 bounding boxes and 2300 unique pedestrians were annotated.

The annotation includes temporal correspondence between bounding boxes and detailed occlusion labels.

Use cases

Pedestrian detection in images/videos.

Properties

- `name`: caltech_pedestrian
- `keywords`: image_processing, detection, pedestrian
- `dataset size`: 11,9 GB
- `is downloadable`: **yes**
- **`tasks:`**
  - **detection: (default)**
    - `primary use`: object detection
    - `description`: Contains image filenames, classes and bounding box annotations for pedestrian detection in images/videos.
    - `sets`: train, test
    - `metadata file size in disk`: 728,4 kB
    - **`has annotations:` yes**
      - **`which:`**

        labels for each class/category.

        bounding box of pedestrians.

        occlusion % of annotated pedestrians.
  - **detection_10x:**
    - `primary use`: object detection
    - `description`: Contains image filenames, classes and bounding box annotations for pedestrian detection in images/videos.
    - `sets`: train, test
    - `metadata file size in disk`: 6,2 MB
    - **`has annotations:` yes**
      - **`which:`**

        labels for each class/category.

        bounding box of pedestrians.

        occlusion % of annotated pedestrians.
  - **detection_30x:**
    - `primary use`: object detection
    - `description`: Contains image filenames, classes and bounding box annotations for pedestrian detection in images/videos.
    - `sets`: train, test
    - `metadata file size in disk`: 17,4 MB
    - **`has annotations:` yes**
      - **`which:`**

        labels for each class/category.

bounding box of pedestrians.

occlusion % of annotated pedestrians.

Metadata structure (HDF5)

Task: detection

```
/
├── train/
│   ├── image_filenames   # dtype=np.uint8, shape=(4250,90)  (note: string in ASCII␣
↪format)
│   ├── classes           # dtype=np.uint8, shape=(4,10)     (note: string in ASCII␣
↪format)
│   ├── boxes             # dtype=np.float, shape=(6313,4)
│   ├── boxesv            # dtype=np.float, shape=(6313,4)
│   ├── id                # dtype=np.int32, shape=(6313,)
│   ├── occlusion         # dtype=np.float, shape=(6313,)
│   ├── object_fields     # dtype=np.uint8, shape=(6,16)     (note: string in ASCII␣
↪format)
│   ├── object_ids        # dtype=np.int32, shape=(6313,6)
│   ├── list_image_filenames_per_class   # dtype=np.int32, shape=(4,5033))
│   ├── list_boxes_per_image             # dtype=np.int32, shape=(4250,22))
│   ├── list_boxesv_per_image            # dtype=np.int32, shape=(4250,22))
│   ├── list_object_ids_per_image        # dtype=np.int32, shape=(4250,22))
│   └── list_objects_ids_per_class       # dtype=np.int32, shape=(4,5033))
│
└── test/
    ├── image_filenames   # dtype=np.uint8, shape=(4024,90)  (note: string in ASCII␣
↪format)
    ├── classes           # dtype=np.uint8, shape=(4,10)     (note: string in ASCII␣
↪format)
    ├── boxes             # dtype=np.float, shape=(5109,4)
    ├── boxesv            # dtype=np.float, shape=(5109,4)
    ├── id                # dtype=np.int32, shape=(5109,)
    ├── occlusion         # dtype=np.float, shape=(5109,)
    ├── object_fields     # dtype=np.uint8, shape=(6,16)     (note: string in ASCII␣
↪format)
    ├── object_ids        # dtype=np.int32, shape=(5109,6)
    ├── list_image_filenames_per_class   # dtype=np.int32, shape=(4,2010))
    ├── list_boxes_per_image             # dtype=np.int32, shape=(4024,13))
    ├── list_boxesv_per_image            # dtype=np.int32, shape=(4024,13))
    ├── list_object_ids_per_image        # dtype=np.int32, shape=(4024,13))
    └── list_objects_ids_per_class       # dtype=np.int32, shape=(4,4371))
```

Fields

- **`image_filenames`: image file path+names**
  - `available in`: train, test
  - `dtype`: np.uint8
  - `is padded`: True
  - `fill value`: 0

  - `note`: strings stored in ASCII format

- **`classes`: class names**
  - `available in`: train, test
  - `dtype`: np.uint8
  - `is padded`: True
  - `fill value`: 0
  - `note`: strings stored in ASCII format

- **`boxes`: bounding boxes**
  - `available in`: train, test
  - `dtype`: np.float
  - `is padded`: False
  - `fill value`: -1
  - `note`: bbox format (x1,y1,x2,y2)

- **`boxesv`: bounding boxes (visible)**
  - `available in`: train, test
  - `dtype`: np.float
  - `is padded`: False
  - `fill value`: -1
  - `note`: bbox format (x1,y1,x2,y2)

- **`id`: label ids**
  - `available in`: train, test
  - `dtype`: np.int32
  - `is padded`: False
  - `fill value`: -1

- **`occlusion`: occlusion percentage**
  - `available in`: train, test
  - `dtype`: np.float
  - `is padded`: False
  - `fill value`: -1

- **`object_fields`: list of field names of the object id list**
  - `available in`: train, test
  - `dtype`: np.uint8
  - `is padded`: True
  - `fill value`: 0
  - `note`: strings stored in ASCII format
  - `note`: key field (*field name* aggregator)

- **object_ids: list of field ids**
  - available in: train, test
  - dtype: np.int32
  - is padded: False
  - fill value: -1
  - note: key field (*field id* aggregator)
- **list_image_filenames_per_class: list of image per class**
  - available in: train, test
  - dtype: np.int32
  - is padded: True
  - fill value: -1
  - note: pre-ordered list
- **list_boxes_per_image: list of bounding boxes per image**
  - available in: train, test
  - dtype: np.int32
  - is padded: True
  - fill value: -1
  - note: pre-ordered list
- **list_boxesv_per_image: list of (visible) bounding boxes per image**
  - available in: train, test
  - dtype: np.int32
  - is padded: True
  - fill value: -1
  - note: pre-ordered list
- **list_object_ids_per_image: list of object ids per image**
  - available in: train, test
  - dtype: np.int32
  - is padded: True
  - fill value: -1
  - note: pre-ordered list
- **list_objects_ids_per_class: list of object ids per class**
  - available in: train, test
  - dtype: np.int32
  - is padded: True
  - fill value: -1
  - note: pre-ordered list

Task: detection_10x

```
/
├── train/
│   ├── image_filenames   # dtype=np.uint8, shape=(42782,90)  (note: string in ASCII␣
↪format)
│   ├── classes           # dtype=np.uint8, shape=(4,10)     (note: string in ASCII␣
↪format)
│   ├── boxes             # dtype=np.float, shape=(63538,4)
│   ├── boxesv            # dtype=np.float, shape=(63538,4)
│   ├── id                # dtype=np.int32, shape=(63538,)
│   ├── occlusion         # dtype=np.float, shape=(63538,)
│   ├── object_fields     # dtype=np.uint8, shape=(6,16)     (note: string in ASCII␣
↪format)
│   ├── object_ids        # dtype=np.int32, shape=(63538,6)
│   ├── list_image_filenames_per_class   # dtype=np.int32, shape=(4,20422))
│   ├── list_boxes_per_image             # dtype=np.int32, shape=(42782,22))
│   ├── list_boxesv_per_image            # dtype=np.int32, shape=(42782,22))
│   ├── list_object_ids_per_image        # dtype=np.int32, shape=(42782,22))
│   └── list_objects_ids_per_class       # dtype=np.int32, shape=(4,50605))
│
└── test/
    ├── image_filenames   # dtype=np.uint8, shape=(40465,90)  (note: string in ASCII␣
↪format)
    ├── classes           # dtype=np.uint8, shape=(4,10)     (note: string in ASCII␣
↪format)
    ├── boxes             # dtype=np.float, shape=(51079,4)
    ├── boxesv            # dtype=np.float, shape=(51079,4)
    ├── id                # dtype=np.int32, shape=(51079,)
    ├── occlusion         # dtype=np.float, shape=(51079,)
    ├── object_fields     # dtype=np.uint8, shape=(6,16)     (note: string in ASCII␣
↪format)
    ├── object_ids        # dtype=np.int32, shape=(51079,6)
    ├── list_image_filenames_per_class   # dtype=np.int32, shape=(4,20173))
    ├── list_boxes_per_image             # dtype=np.int32, shape=(40465,14))
    ├── list_boxesv_per_image            # dtype=np.int32, shape=(40465,14))
    ├── list_object_ids_per_image        # dtype=np.int32, shape=(40465,14))
    └── list_objects_ids_per_class       # dtype=np.int32, shape=(4,43748))
```

Fields

- **`image_filenames`: image file path+names**
  - `available in`: train, test
  - `dtype`: np.uint8
  - `is padded`: True
  - `fill value`: 0
  - `note`: strings stored in ASCII format
- **`classes`: class names**
  - `available in`: train, test
  - `dtype`: np.uint8

    - `is padded`: True
    - `fill value`: 0
    - `note`: strings stored in ASCII format
- **`boxes`: bounding boxes**
    - `available in`: train, test
    - `dtype`: np.float
    - `is padded`: False
    - `fill value`: -1
    - `note`: bbox format (x1,y1,x2,y2)
- **`boxesv`: bounding boxes (visible)**
    - `available in`: train, test
    - `dtype`: np.float
    - `is padded`: False
    - `fill value`: -1
    - `note`: bbox format (x1,y1,x2,y2)
- **`id`: label ids**
    - `available in`: train, test
    - `dtype`: np.int32
    - `is padded`: False
    - `fill value`: -1
- **`occlusion`: occlusion percentage**
    - `available in`: train, test
    - `dtype`: np.float
    - `is padded`: False
    - `fill value`: -1
- **`object_fields`: list of field names of the object id list**
    - `available in`: train, test
    - `dtype`: np.uint8
    - `is padded`: True
    - `fill value`: 0
    - `note`: strings stored in ASCII format
    - `note`: key field (*field name* aggregator)
- **`object_ids`: list of field ids**
    - `available in`: train, test
    - `dtype`: np.int32
    - `is padded`: False

    - `fill value`: -1
    - `note`: key field (*field id* aggregator)
- **`list_image_filenames_per_class`: list of image per class**
    - `available in`: train, test
    - `dtype`: np.int32
    - `is padded`: True
    - `fill value`: -1
    - `note`: pre-ordered list
- **`list_boxes_per_image`: list of bounding boxes per image**
    - `available in`: train, test
    - `dtype`: np.int32
    - `is padded`: True
    - `fill value`: -1
    - `note`: pre-ordered list
- **`list_boxesv_per_image`: list of (visible) bounding boxes per image**
    - `available in`: train, test
    - `dtype`: np.int32
    - `is padded`: True
    - `fill value`: -1
    - `note`: pre-ordered list
- **`list_object_ids_per_image`: list of object ids per image**
    - `available in`: train, test
    - `dtype`: np.int32
    - `is padded`: True
    - `fill value`: -1
    - `note`: pre-ordered list
- **`list_objects_ids_per_class`: list of object ids per class**
    - `available in`: train, test
    - `dtype`: np.int32
    - `is padded`: True
    - `fill value`: -1
    - `note`: pre-ordered list

Task: detection_30x

```
/
├── train/
│   ├── image_filenames   # dtype=np.uint8, shape=(128419,90)  (note: string in ASCII␣
↪format)
│   ├── classes           # dtype=np.uint8, shape=(4,10)       (note: string in ASCII␣
↪format)
│   ├── boxes             # dtype=np.float, shape=(190598,4)
│   ├── boxesv            # dtype=np.float, shape=(190598,4)
│   ├── id                # dtype=np.int32, shape=(190598,)
│   ├── occlusion         # dtype=np.float, shape=(190598,)
│   ├── object_fields     # dtype=np.uint8, shape=(6,16)       (note: string in ASCII␣
↪format)
│   ├── object_ids        # dtype=np.int32, shape=(190598,6)
│   ├── list_image_filenames_per_class   # dtype=np.int32, shape=(4,61274))
│   ├── list_boxes_per_image             # dtype=np.int32, shape=(128419,22))
│   ├── list_boxesv_per_image            # dtype=np.int32, shape=(128419,22))
│   ├── list_object_ids_per_image        # dtype=np.int32, shape=(128419,22))
│   └── list_objects_ids_per_class       # dtype=np.int32, shape=(4,151768))
│
└── test/
    ├── image_filenames   # dtype=np.uint8, shape=(121465,90)  (note: string in ASCII␣
↪format)
    ├── classes           # dtype=np.uint8, shape=(4,10)       (note: string in ASCII␣
↪format)
    ├── boxes             # dtype=np.float, shape=(153305,4)
    ├── boxesv            # dtype=np.float, shape=(153305,4)
    ├── id                # dtype=np.int32, shape=(153305,)
    ├── occlusion         # dtype=np.float, shape=(153305,)
    ├── object_fields     # dtype=np.uint8, shape=(6,16)       (note: string in ASCII␣
↪format)
    ├── object_ids        # dtype=np.int32, shape=(153305,6)
    ├── list_image_filenames_per_class   # dtype=np.int32, shape=(4,60537))
    ├── list_boxes_per_image             # dtype=np.int32, shape=(121465,14))
    ├── list_boxesv_per_image            # dtype=np.int32, shape=(121465,14))
    ├── list_object_ids_per_image        # dtype=np.int32, shape=(121465,14))
    └── list_objects_ids_per_class       # dtype=np.int32, shape=(4,131273))
```

Fields

- **`image_filenames`: image file path+names**
  - `available in`: train, test
  - `dtype`: np.uint8
  - `is padded`: True
  - `fill value`: 0
  - `note`: strings stored in ASCII format
- **`classes`: class names**
  - `available in`: train, test
  - `dtype`: np.uint8
  - `is padded`: True
  - `fill value`: 0

    - `note`: strings stored in ASCII format

- **`boxes`: bounding boxes**
    - `available in`: train, test
    - `dtype`: np.float
    - `is padded`: False
    - `fill value`: -1
    - `note`: bbox format (x1,y1,x2,y2)
- **`boxesv`: bounding boxes (visible)**
    - `available in`: train, test
    - `dtype`: np.float
    - `is padded`: False
    - `fill value`: -1
    - `note`: bbox format (x1,y1,x2,y2)
- **`id`: label ids**
    - `available in`: train, test
    - `dtype`: np.int32
    - `is padded`: False
    - `fill value`: -1
- **`occlusion`: occlusion percentage**
    - `available in`: train, test
    - `dtype`: np.float
    - `is padded`: False
    - `fill value`: -1
- **`object_fields`: list of field names of the object id list**
    - `available in`: train, test
    - `dtype`: np.uint8
    - `is padded`: True
    - `fill value`: 0
    - `note`: strings stored in ASCII format
    - `note`: key field (*field name* aggregator)
- **`object_ids`: list of field ids**
    - `available in`: train, test
    - `dtype`: np.int32
    - `is padded`: False
    - `fill value`: -1
    - `note`: key field (*field id* aggregator)

- **list_image_filenames_per_class: list of image per class**
  - available in: train, test
  - dtype: np.int32
  - is padded: True
  - fill value: -1
  - note: pre-ordered list
- **list_boxes_per_image: list of bounding boxes per image**
  - available in: train, test
  - dtype: np.int32
  - is padded: True
  - fill value: -1
  - note: pre-ordered list
- **list_boxesv_per_image: list of (visible) bounding boxes per image**
  - available in: train, test
  - dtype: np.int32
  - is padded: True
  - fill value: -1
  - note: pre-ordered list
- **list_object_ids_per_image: list of object ids per image**
  - available in: train, test
  - dtype: np.int32
  - is padded: True
  - fill value: -1
  - note: pre-ordered list
- **list_objects_ids_per_class: list of object ids per class**
  - available in: train, test
  - dtype: np.int32
  - is padded: True
  - fill value: -1
  - note: pre-ordered list

Disclaimer

CIFAR-10

The **CIFAR-10** dataset consists of 60000 32x32 colour images in 10 classes, with 6000 images per class. There are 50000 training images and 10000 test images.

Use cases

Image classification.

Properties

- `name`: cifar10
- `keywords`: image_processing, classification
- `dataset size`: 170,5 MB
- `is downloadable`: **yes**
- `tasks`: *classification (default)*

Tasks

classification (default)

- *How to use*
- *Properties*
- *HDF5 file structure*
- *Fields*

How to use

```
>>> # import the package
>>> import dbcollection as dbc
>>>
>>> # load the dataset
>>> cifar10 = dbc.load('cifar10')
>>> cifar10
DataLoader: "cifar10" (classification task)
```

Properties

- `primary use`: image classification
- `description`: Contains image tensors and label annotations for image classification.
- `sets`: train, test
- `metadata file size in disk`: 178,3 MB
- **`has annotations`: yes**

        - **which:**
            * labels for each image class/category.

- **available fields:**
    - *classes*
    - *images*
    - *labels*
    - *object_fields*
    - *object_ids*
    - *list_images_per_class*

HDF5 file structure

```
/
├── train/
│   ├── classes        # dtype=np.uint8, shape=(10,11)  (note: string in ASCII format)
│   ├── images         # dtype=np.uint8, shape=(50000,32,32,3)
│   ├── labels         # dtype=np.uint8, shape=(50000,)
│   ├── object_fields  # dtype=np.uint8, shape=(2,8)    (note: string in ASCII format)
│   ├── object_ids     # dtype=np.int32, shape=(50000,2)
│   └── list_images_per_class   # dtype=np.int32, shape=(10,5000))
│
└── test/
    ├── classes        # dtype=np.uint8, shape=(10,11)  (note: string in ASCII format)
    ├── images         # dtype=np.uint8, shape=(10000,32,32,3)
    ├── labels         # dtype=np.uint8, shape=(10000,)
    ├── object_fields  # dtype=np.uint8, shape=(2,8)    (note: string in ASCII format)
    ├── object_ids     # dtype=np.int32, shape=(10000,2)
    └── list_images_per_class   # dtype=np.int32, shape=(10,1000))
```

Fields

- **`classes`: class names**
    - `available in`: train, test
    - `dtype`: np.uint8
    - `is padded`: True
    - `fill value`: 0
    - `note`: strings stored in ASCII format
- **`images`: images tensor**
    - `available in`: train, test
    - `dtype`: np.uint8
    - `is padded`: False
    - `fill value`: -1
- **`labels`: class ids**

        - `available in`: train, test
        - `dtype`: np.uint8
        - `is padded`: False
        - `fill value`: -1
- **`object_fields`: list of field names of the object id list**
        - `available in`: train, test
        - `dtype`: np.uint8
        - `is padded`: True
        - `fill value`: 0
        - `note`: strings stored in ASCII format
        - `note`: key field (*field name* aggregator)
- **`object_ids`: list of field ids**
        - `available in`: train, test
        - `dtype`: np.int32
        - `is padded`: False
        - `fill value`: -1
        - `note`: key field (*field id* aggregator)
- **`list_images_per_class`: list of image ids per class**
        - `available in`: train, test
        - `dtype`: np.int32
        - `is padded`: True
        - `fill value`: -1
        - `note`: pre-ordered list

Disclaimer

CIFAR-100

This dataset is just like the **CIFAR-10**, except it has 100 classes containing 600 images each. There are 500 training images and 100 testing images per class. The 100 classes in the **CIFAR-100** are grouped into 20 superclasses. Each image comes with a "fine" label (the class to which it belongs) and a "coarse" label (the superclass to which it belongs).

Use cases

Image classification.

Properties

- `name`: cifar100
- `keywords`: image_processing, classification
- `dataset size`: 355,3 MB
- `is downloadable`: **yes**
- `tasks`: *classification (default)*

Tasks

classification (default)

- *How to use*
- *Properties*
- *HDF5 file structure*
- *Fields*

How to use

```
>>> # import the package
>>> import dbcollection as dbc
>>>
>>> # load the dataset
>>> cifar100 = dbc.load('cifar100')
>>> cifar100
DataLoader: "cifar100" (classification task)
```

Properties

- `primary use`: image classification
- `description`: Contains image tensors and label annotations for image classification.
- `sets`: train, test
- `metadata file size in disk`: 177,8 MB
- **`has annotations`: yes**
    - **which:**
        * labels for each image class/category.
- **`available fields`:**
    - *classes*
    - *superclasses*
    - *images*
    - *labels*

- *coarse_labels*
- *object_fields*
- *object_ids*
- *list_images_per_class*
- *list_images_per_superclass*

HDF5 file structure

```
/
├── train/
│   ├── classes        # dtype=np.uint8, shape=(100,18)  (note: string in ASCII␣
↪format)
│   ├── superclasses   # dtype=np.uint8, shape=(20,31)   (note: string in ASCII␣
↪format)
│   ├── images         # dtype=np.uint8, shape=(50000,32,32,3)
│   ├── labels         # dtype=np.uint8, shape=(50000,)
│   ├── coarse_labels  # dtype=np.uint8, shape=(50000,)
│   ├── object_fields  # dtype=np.uint8, shape=(3,13)    (note: string in ASCII␣
↪format)
│   ├── object_ids     # dtype=np.int32, shape=(50000,3)
│   ├── list_images_per_class        # dtype=np.int32, shape=(100,500))
│   └── list_images_per_superclass   # dtype=np.int32, shape=(20,2500))
│
└── test/
    ├── classes        # dtype=np.uint8, shape=(100,18)  (note: string in ASCII␣
↪format)
    ├── superclasses   # dtype=np.uint8, shape=(20,31)   (note: string in ASCII␣
↪format)
    ├── images         # dtype=np.uint8, shape=(10000,32,32,3)
    ├── labels         # dtype=np.uint8, shape=(10000,)
    ├── coarse_labels  # dtype=np.uint8, shape=(10000,)
    ├── object_fields  # dtype=np.uint8, shape=(3,13)    (note: string in ASCII␣
↪format)
    ├── object_ids     # dtype=np.int32, shape=(10000,3)
    ├── list_images_per_class        # dtype=np.int32, shape=(100,100))
    └── list_images_per_superclass   # dtype=np.int32, shape=(20,500))
```

Fields

- **`classes`: class descriptions**
  - `available in`: train, test
  - `dtype`: np.uint8
  - `is padded`: True
  - `fill value`: 0
  - `note`: strings stored in ASCII format
- **`superclasses`: super class names. It is composed of groups of classes per super class**
  - `available in`: train, test

  - dtype: np.uint8
  - is padded: True
  - fill value: 0
  - note: strings stored in ASCII format

- **images: images tensor**
  - available in: train, test
  - dtype: np.uint8
  - is padded: False
  - fill value: -1

- **labels: class ids**
  - available in: train, test
  - dtype: np.uint8
  - is padded: False
  - fill value: -1

- **coarse_labels: superclass ids**
  - available in: train, test
  - dtype: np.uint8
  - is padded: False
  - fill value: -1

- **object_fields: list of field names of the object id list**
  - available in: train, test
  - dtype: np.uint8
  - is padded: True
  - fill value: 0
  - note: strings stored in ASCII format
  - note: key field (*field name* aggregator)

- **object_ids: list of field ids**
  - available in: train, test
  - dtype: np.int32
  - is padded: False
  - fill value: -1
  - note: key field (*field id* aggregator)

- **list_images_per_class: list of image ids per class**
  - available in: train, test
  - dtype: np.int32
  - is padded: True

        - `fill value`: -1
        - `note`: pre-ordered list
- **`list_images_per_superclass`: list of image ids per superclass**
        - `available in`: train, test
        - `dtype`: np.int32
        - `is padded`: True
        - `fill value`: -1
        - `note`: pre-ordered list

Disclaimer

All rights reserved to the original creators of **CIFAR-100**.

For information about the dataset and its terms of use, please see this link.

COCO - Common Objects in Context

The Microsoft Common Objects in COntext (MS COCO) dataset contains 91 common object categories with 82 of them having more than 5,000 labeled instances. In total the dataset has 2,500,000 labeled instances in 328,000 images.

Use cases

Object detection, segmentation, captioning and human body joint detection.

Properties

- `name`: coco
- `keywords`: image_processing, detection, keypoint, captions, human, pose
- `dataset size`: 40,3 GB
- `is downloadable`: **yes**
- **`tasks`:**
        - **detection_2015: (default)**
            * `primary use`: object detection
            * `description`: Contains image filenames, classes, bounding box and segmentation mask annotations for object detection in images.
            * `sets`: train, val, test
            * `metadata file size in disk`: 243,6 MB
            * **`has annotations`: yes**
                · **`which`:**

                image filenames

                object categories and supercategories

bounding box of pedestrians.

occlusion % of annotated pedestrians.

segmentation masks

- **detection_2016:**
  - `primary use`: object detection
  - `description`: Contains image filenames, classes, bounding box and segmentation mask annotations for object detection in images.
  - `sets`: train, val, test, test_dev
  - `metadata file size in disk`: 244,7 MB
  - **`has annotations`: yes**
    - **`which`:**

      image filenames

      object categories and supercategories

      bounding box of pedestrians.

      occlusion % of annotated pedestrians.

      segmentation masks

- **caption_2015:**
  - `primary use`: image captioning
  - `description`: Contains image filenames and captions for image captioning.
  - `sets`: train, val, test
  - `metadata file size in disk`: 21,9 MB
  - **`has annotations`: yes**
    - **`which`:**

      image filenames

      captions

- **caption_2016:**
  - `primary use`: image captioning
  - `description`: Contains image filenames and captions for image captioning.
  - `sets`: train, val, test, test_dev
  - `metadata file size in disk`: 23,0 MB
  - **`has annotations`: yes**
    - **`which`:**

      image filenames

      captions

- **keypoints_2016:**
  - `primary use`: human body joint detection

* `description`: Contains image filenames, classes, bounding box and segmentation mask annotations for object detection in images.
* `sets`: train, val, test, test_dev
* `metadata file size in disk`: 106,6 MB
* **`has annotations`: yes**
    - **`which`:**

      image filenames

      object categories and supercategories

      bounding box of pedestrians.

      occlusion % of annotated pedestrians.

      segmentation masks

      body joint keypoints

      skeleton

Metadata structure (HDF5)

Task: detection_2015

```
/
├── train/
│   ├── image_filenames       # dtype=np.uint8, shape=(82783,74)    (note: string in␣
↪ASCII format)
│   ├── category              # dtype=np.uint8, shape=(80,15)       (note: string in␣
↪ASCII format)
│   ├── supercategory         # dtype=np.uint8, shape=(12,11)       (note: string in␣
↪ASCII format)
│   ├── coco_annotations_ids  # dtype=np.int32, shape=(604907,)
│   ├── coco_categories_ids   # dtype=np.int32, shape=(80,)
│   ├── coco_images_ids       # dtype=np.int32, shape=(82783,)
│   ├── coco_urls             # dtype=np.uint8, shape=(82783,32)    (note: string in␣
↪ASCII format)
│   ├── image_id              # dtype=np.int32, shape=(82783,)
│   ├── category_id           # dtype=np.int32, shape=(80,)
│   ├── annotation_id         # dtype=np.int32, shape=(604907,)
│   ├── width                 # dtype=np.int32, shape=(82783,)
│   ├── height                # dtype=np.int32, shape=(82783,)
│   ├── boxes                 # dtype=np.float, shape=(604907,4)
│   ├── iscrowd               # dtype=np.uint8, shape=(2,)
│   ├── segmentation          # dtype=np.float, shape=(604907,10043)
│   ├── area                  # dtype=np.int32, shape=(604907,)
│   ├── object_fields         # dtype=np.uint8, shape=(13,16)       (note: string in␣
↪ASCII format)
│   ├── object_ids            # dtype=np.int32, shape=(604907,13)
│   ├── list_boxes_per_image                      # dtype=np.int32, shape=(82783,93))
│   ├── list_image_filenames_per_category         # dtype=np.int32, shape=(80,45174))
│   ├── list_image_filenames_per_supercategory    # dtype=np.int32, shape=(12,45174))
│   ├── list_object_ids_per_image                 # dtype=np.int32, shape=(82783,93))
│   ├── list_objects_ids_per_category             # dtype=np.int32, shape=(80,185316))
│   └── list_objects_ids_per_supercategory        # dtype=np.int32, shape=(12,185316))
```

(continues on next page)

(continued from previous page)

```
│
├── val/
│   ├── image_filenames       # dtype=np.uint8, shape=(40504,74)   (note: string in␣
↪ASCII format)
│   ├── category              # dtype=np.uint8, shape=(80,15)      (note: string in␣
↪ASCII format)
│   ├── supercategory         # dtype=np.uint8, shape=(12,11)      (note: string in␣
↪ASCII format)
│   ├── coco_annotations_ids  # dtype=np.int32, shape=(291875,)
│   ├── coco_categories_ids   # dtype=np.int32, shape=(80,)
│   ├── coco_images_ids       # dtype=np.int32, shape=(40504,)
│   ├── coco_urls             # dtype=np.uint8, shape=(40504,32)   (note: string in␣
↪ASCII format)
│   ├── image_id              # dtype=np.int32, shape=(40504,)
│   ├── category_id           # dtype=np.int32, shape=(80,)
│   ├── annotation_id         # dtype=np.int32, shape=(291875,)
│   ├── width                 # dtype=np.int32, shape=(40504,)
│   ├── height                # dtype=np.int32, shape=(40504,)
│   ├── boxes                 # dtype=np.float, shape=(291875,4)
│   ├── iscrowd               # dtype=np.uint8, shape=(2,)
│   ├── segmentation          # dtype=np.float, shape=(291875,7237)
│   ├── area                  # dtype=np.int32, shape=(291875,)
│   ├── object_fields         # dtype=np.uint8, shape=(13,16)      (note: string in␣
↪ASCII format)
│   ├── object_ids            # dtype=np.int32, shape=(291875,13)
│   ├── list_boxes_per_image                      # dtype=np.int32, shape=(40504,93))
│   ├── list_image_filenames_per_category         # dtype=np.int32, shape=(80,21634))
│   ├── list_image_filenames_per_supercategory    # dtype=np.int32, shape=(12,21634))
│   ├── list_object_ids_per_image                 # dtype=np.int32, shape=(40504,93))
│   ├── list_objects_ids_per_category             # dtype=np.int32, shape=(80,88153))
│   └── list_objects_ids_per_supercategory        # dtype=np.int32, shape=(12,88153))
│
└── test/
    ├── image_filenames       # dtype=np.uint8, shape=(40775,72)   (note: string in␣
↪ASCII format)
    ├── category              # dtype=np.uint8, shape=(80,15)      (note: string in␣
↪ASCII format)
    ├── supercategory         # dtype=np.uint8, shape=(12,11)      (note: string in␣
↪ASCII format)
    ├── coco_categories_ids   # dtype=np.int32, shape=(80,)
    ├── coco_images_ids       # dtype=np.int32, shape=(40775,)
    ├── coco_urls             # dtype=np.uint8, shape=(40775,32)   (note: string in␣
↪ASCII format)
    ├── image_id              # dtype=np.int32, shape=(40775,)
    ├── category_id           # dtype=np.int32, shape=(80,)
    ├── width                 # dtype=np.int32, shape=(40775,)
    ├── height                # dtype=np.int32, shape=(40775,)
    ├── object_fields         # dtype=np.uint8, shape=(4,16)       (note: string in␣
↪ASCII format)
    ├── object_ids            # dtype=np.int32, shape=(40775,4)
    └── list_object_ids_per_image   # dtype=np.int32, shape=(40775,1))
```

Fields

- **`image_filenames`: image file path+names**

  - `available in`: train, val, test
  - `dtype`: np.uint8
  - `is padded`: True
  - `fill value`: 0
  - `note`: strings stored in ASCII format
- **`category`: category names**
  - `available in`: train, val, test
  - `dtype`: np.uint8
  - `is padded`: True
  - `fill value`: 0
  - `note`: strings stored in ASCII format
- **`supercategory`: super category names**
  - `available in`: train, val, test
  - `dtype`: np.uint8
  - `is padded`: True
  - `fill value`: 0
  - `note`: strings stored in ASCII format
- **`coco_annotations_ids`: reference to coco annotation ids (useful for evaluating on coco)**
  - `available in`: train, val
  - `dtype`: np.int32
  - `is padded`: False
  - `fill value`: -1
- **`coco_categories_ids`: reference to coco category ids (useful for evaluating on coco)**
  - `available in`: train, val, test
  - `dtype`: np.int32
  - `is padded`: False
  - `fill value`: -1
- **`coco_images_ids`: reference to coco image filename ids (useful for evaluating on coco)**
  - `available in`: train, val, test
  - `dtype`: np.int32
  - `is padded`: False
  - `fill value`: -1
- **`coco_urls`: coco urls**
  - `available in`: train, val, test
  - `dtype`: np.uint8
  - `is padded`: True

        - `fill value`: 0
        - `note`: strings stored in ASCII format
- **`image_id`: image filename ids**
        - `available in`: train, val, test
        - `dtype`: np.int32
        - `is padded`: False
        - `fill value`: -1
- **`category_id`: category ids**
        - `available in`: train, val, test
        - `dtype`: np.int32
        - `is padded`: False
        - `fill value`: -1
- **`annotation_id`: annotation ids**
        - `available in`: train, val
        - `dtype`: np.int32
        - `is padded`: False
        - `fill value`: -1
- **`width`: image width**
        - `available in`: train, val, test
        - `dtype`: np.int32
        - `is padded`: False
        - `fill value`: -1
- **`height`: image height**
        - `available in`: train, val, test
        - `dtype`: np.int32
        - `is padded`: False
        - `fill value`: -1
- **`boxes`: bounding box**
        - `available in`: train, val
        - `dtype`: np.float
        - `is padded`: False
        - `fill value`: -1
        - `note`: bbox format (x1,y1,x2,y2)
- **`iscrowd`: is crowd (0 - False, 1 - True)**
        - `available in`: train, val
        - `dtype`: np.uint8

  - `is padded`: False
  - `fill value`: -1

- **`segmentation`: segmentation mask**
  - `available in`: train, val
  - `dtype`: np.float
  - `is padded`: True
  - `fill value`: -1
  - `note`: the masks come in 3 different formats, but they are mostly lists of lists. These have been packed (vectorized) into an array with a single dimension in order to be stored in the HDF5 metadata file. To unpack these arrays to their original format, use the `unsqueeze_list()` method in `dbcollection.utils.pad`.

- **`area`: object area**
  - `available in`: train, val
  - `dtype`: np.int32
  - `is padded`: False
  - `fill value`: -1

- **`object_fields`: list of field names of the object id list**
  - `available in`: train, val, test
  - `dtype`: np.uint8
  - `is padded`: True
  - `fill value`: 0
  - `note`: strings stored in ASCII format
  - `note`: key field (*field name* aggregator)

- **`object_ids`: list of field ids**
  - `available in`: train, val, test
  - `dtype`: np.int32
  - `is padded`: False
  - `fill value`: -1
  - `note`: key field (*field id* aggregator)

- **`list_boxes_per_image`: list of bounding boxes per image**
  - `available in`: train, val
  - `dtype`: np.int32
  - `is padded`: True
  - `fill value`: -1
  - `note`: pre-ordered list

- **`list_image_filenames_per_category`: list of image filenames per category**
  - `available in`: train, val

        - `dtype`: np.int32
        - `is padded`: True
        - `fill value`: -1
        - `note`: pre-ordered list

- **`list_image_filenames_per_supercategory`: list of image filenames per supercategory**
        - `available in`: train, val
        - `dtype`: np.int32
        - `is padded`: True
        - `fill value`: -1
        - `note`: pre-ordered list

- **`list_object_ids_per_image`: list of object ids per image**
        - `available in`: train, val, test
        - `dtype`: np.int32
        - `is padded`: True
        - `fill value`: -1
        - `note`: pre-ordered list

- **`list_objects_ids_per_category`: list of object ids per category**
        - `available in`: train, val
        - `dtype`: np.int32
        - `is padded`: True
        - `fill value`: -1
        - `note`: pre-ordered list

- **`list_objects_ids_per_supercategory`: list of object ids per supercategory**
        - `available in`: train, val
        - `dtype`: np.int32
        - `is padded`: True
        - `fill value`: -1
        - `note`: pre-ordered list

Task: detection_2016

```
/
├── train/
│   ├── image_filenames       # dtype=np.uint8, shape=(82783,74)    (note: string in␣
↪ASCII format)
│   ├── category              # dtype=np.uint8, shape=(80,15)       (note: string in␣
↪ASCII format)
│   ├── supercategory         # dtype=np.uint8, shape=(12,11)       (note: string in␣
↪ASCII format)
```

(continues on next page)

(continued from previous page)

```
│   ├── coco_annotations_ids  # dtype=np.int32, shape=(604907,)
│   ├── coco_categories_ids   # dtype=np.int32, shape=(80,)
│   ├── coco_images_ids       # dtype=np.int32, shape=(82783,)
│   ├── coco_urls             # dtype=np.uint8, shape=(82783,32)   (note: string in␣
↪ASCII format)
│   ├── image_id              # dtype=np.int32, shape=(82783,)
│   ├── category_id           # dtype=np.int32, shape=(80,)
│   ├── annotation_id         # dtype=np.int32, shape=(604907,)
│   ├── width                 # dtype=np.int32, shape=(82783,)
│   ├── height                # dtype=np.int32, shape=(82783,)
│   ├── boxes                 # dtype=np.float, shape=(604907,4)
│   ├── iscrowd               # dtype=np.uint8, shape=(2,)
│   ├── segmentation          # dtype=np.float, shape=(604907,10043)
│   ├── area                  # dtype=np.int32, shape=(604907,)
│   ├── object_fields         # dtype=np.uint8, shape=(13,16)      (note: string in␣
↪ASCII format)
│   ├── object_ids            # dtype=np.int32, shape=(604907,13)
│   ├── list_boxes_per_image                      # dtype=np.int32, shape=(82783,93))
│   ├── list_image_filenames_per_category         # dtype=np.int32, shape=(80,45174))
│   ├── list_image_filenames_per_supercategory    # dtype=np.int32, shape=(12,45174))
│   ├── list_object_ids_per_image                 # dtype=np.int32, shape=(82783,93))
│   ├── list_objects_ids_per_category             # dtype=np.int32, shape=(80,185316))
│   └── list_objects_ids_per_supercategory        # dtype=np.int32, shape=(12,185316))
│
├── val/
│   ├── image_filenames       # dtype=np.uint8, shape=(40504,74)   (note: string in␣
↪ASCII format)
│   ├── category              # dtype=np.uint8, shape=(80,15)      (note: string in␣
↪ASCII format)
│   ├── supercategory         # dtype=np.uint8, shape=(12,11)      (note: string in␣
↪ASCII format)
│   ├── coco_annotations_ids  # dtype=np.int32, shape=(291875,)
│   ├── coco_categories_ids   # dtype=np.int32, shape=(80,)
│   ├── coco_images_ids       # dtype=np.int32, shape=(40504,)
│   ├── coco_urls             # dtype=np.uint8, shape=(40504,32)   (note: string in␣
↪ASCII format)
│   ├── image_id              # dtype=np.int32, shape=(40504,)
│   ├── category_id           # dtype=np.int32, shape=(80,)
│   ├── annotation_id         # dtype=np.int32, shape=(291875,)
│   ├── width                 # dtype=np.int32, shape=(40504,)
│   ├── height                # dtype=np.int32, shape=(40504,)
│   ├── boxes                 # dtype=np.float, shape=(291875,4)
│   ├── iscrowd               # dtype=np.uint8, shape=(2,)
│   ├── segmentation          # dtype=np.float, shape=(291875,7237)
│   ├── area                  # dtype=np.int32, shape=(291875,)
│   ├── object_fields         # dtype=np.uint8, shape=(13,16)      (note: string in␣
↪ASCII format)
│   ├── object_ids            # dtype=np.int32, shape=(291875,13)
│   ├── list_boxes_per_image                      # dtype=np.int32, shape=(40504,93))
│   ├── list_image_filenames_per_category         # dtype=np.int32, shape=(80,21634))
│   ├── list_image_filenames_per_supercategory    # dtype=np.int32, shape=(12,21634))
│   ├── list_object_ids_per_image                 # dtype=np.int32, shape=(40504,93))
│   ├── list_objects_ids_per_category             # dtype=np.int32, shape=(80,88153))
│   └── list_objects_ids_per_supercategory        # dtype=np.int32, shape=(12,88153))
│
├── test/
│   ├── image_filenames       # dtype=np.uint8, shape=(81434,72)   (note: string in␣
↪ASCII format)
```

(continues on next page)

(continued from previous page)

```
│   ├── category              # dtype=np.uint8, shape=(80,15)       (note: string in␣
↪ASCII format)
│   ├── supercategory         # dtype=np.uint8, shape=(12,11)       (note: string in␣
↪ASCII format)
│   ├── coco_categories_ids   # dtype=np.int32, shape=(80,)
│   ├── coco_images_ids       # dtype=np.int32, shape=(81434,)
│   ├── coco_urls             # dtype=np.uint8, shape=(81434,32)    (note: string in␣
↪ASCII format)
│   ├── image_id              # dtype=np.int32, shape=(81434,)
│   ├── category_id           # dtype=np.int32, shape=(80,)
│   ├── width                 # dtype=np.int32, shape=(81434,)
│   ├── height                # dtype=np.int32, shape=(81434,)
│   ├── object_fields         # dtype=np.uint8, shape=(4,16)       (note: string in␣
↪ASCII format)
│   ├── object_ids            # dtype=np.int32, shape=(81434,4)
│   └── list_object_ids_per_image   # dtype=np.int32, shape=(81434,1))
│
└── test_dev/
    ├── image_filenames       # dtype=np.uint8, shape=(20288,72)    (note: string in␣
↪ASCII format)
    ├── category              # dtype=np.uint8, shape=(80,15)       (note: string in␣
↪ASCII format)
    ├── supercategory         # dtype=np.uint8, shape=(12,11)       (note: string in␣
↪ASCII format)
    ├── coco_categories_ids   # dtype=np.int32, shape=(80,)
    ├── coco_images_ids       # dtype=np.int32, shape=(20288,)
    ├── coco_urls             # dtype=np.uint8, shape=(20288,32)    (note: string in␣
↪ASCII format)
    ├── image_id              # dtype=np.int32, shape=(20288,)
    ├── category_id           # dtype=np.int32, shape=(80,)
    ├── width                 # dtype=np.int32, shape=(20288,)
    ├── height                # dtype=np.int32, shape=(20288,)
    ├── object_fields         # dtype=np.uint8, shape=(4,16)       (note: string in␣
↪ASCII format)
    ├── object_ids            # dtype=np.int32, shape=(20288,4)
    └── list_object_ids_per_image   # dtype=np.int32, shape=(20288,1))
```

Fields

- **`image_filenames`: image file path+names**
  - `available in`: train, val, test, test_dev
  - `dtype`: np.uint8
  - `is padded`: True
  - `fill value`: 0
  - `note`: strings stored in ASCII format
- **`category`: category names**
  - `available in`: train, val, test, test_dev
  - `dtype`: np.uint8
  - `is padded`: True

        - `fill value`: 0
        - `note`: strings stored in ASCII format
- **`supercategory`: super category names**
        - `available in`: train, val, test, test_dev
        - `dtype`: np.uint8
        - `is padded`: True
        - `fill value`: 0
        - `note`: strings stored in ASCII format
- **`coco_annotations_ids`: reference to coco annotation ids (useful for evaluating on coco)**
        - `available in`: train, val
        - `dtype`: np.int32
        - `is padded`: False
        - `fill value`: -1
- **`coco_categories_ids`: reference to coco category ids (useful for evaluating on coco)**
        - `available in`: train, val, test, test_dev
        - `dtype`: np.int32
        - `is padded`: False
        - `fill value`: -1
- **`coco_images_ids`: reference to coco image filename ids (useful for evaluating on coco)**
        - `available in`: train, val, test, test_dev
        - `dtype`: np.int32
        - `is padded`: False
        - `fill value`: -1
- **`coco_urls`: coco urls**
        - `available in`: train, val, test, test_dev
        - `dtype`: np.uint8
        - `is padded`: True
        - `fill value`: 0
        - `note`: strings stored in ASCII format
- **`image_id`: image filename ids**
        - `available in`: train, val, test, test_dev
        - `dtype`: np.int32
        - `is padded`: False
        - `fill value`: -1
- **`category_id`: category ids**
        - `available in`: train, val, test, test_dev

  - `dtype`: np.int32
  - `is padded`: False
  - `fill value`: -1

- **`annotation_id`: annotation ids**
  - `available in`: train, val
  - `dtype`: np.int32
  - `is padded`: False
  - `fill value`: -1

- **`width`: image width**
  - `available in`: train, val, test, test_dev
  - `dtype`: np.int32
  - `is padded`: False
  - `fill value`: -1

- **`height`: image height**
  - `available in`: train, val, test, test_dev
  - `dtype`: np.int32
  - `is padded`: False
  - `fill value`: -1

- **`boxes`: bounding box**
  - `available in`: train, val
  - `dtype`: np.float
  - `is padded`: False
  - `fill value`: -1
  - `note`: bbox format (x1,y1,x2,y2)

- **`iscrowd`: is crowd (0 - False, 1 - True)**
  - `available in`: train, val
  - `dtype`: np.uint8
  - `is padded`: False
  - `fill value`: -1

- **`segmentation`: segmentation mask**
  - `available in`: train, val
  - `dtype`: np.float
  - `is padded`: True
  - `fill value`: -1

  - `note`: the masks come in 3 different formats, but they are mostly lists of lists. These have been packed (vectorized) into an array with a single dimension in order to be stored in the HDF5 metadata file. To unpack these arrays to their original format, use the `unsqueeze_list()` method in `dbcollection.utils.pad`.

- **`area`: object area**
  - `available in`: train, val
  - `dtype`: np.int32
  - `is padded`: False
  - `fill value`: -1
- **`object_fields`: list of field names of the object id list**
  - `available in`: train, val, test, test_dev
  - `dtype`: np.uint8
  - `is padded`: True
  - `fill value`: 0
  - `note`: strings stored in ASCII format
  - `note`: key field (*field name* aggregator)
- **`object_ids`: list of field ids**
  - `available in`: train, val, test, test_dev
  - `dtype`: np.int32
  - `is padded`: False
  - `fill value`: -1
  - `note`: key field (*field id* aggregator)
- **`list_boxes_per_image`: list of bounding boxes per image**
  - `available in`: train, val
  - `dtype`: np.int32
  - `is padded`: True
  - `fill value`: -1
  - `note`: pre-ordered list
- **`list_image_filenames_per_category`: list of image filenames per category**
  - `available in`: train, val
  - `dtype`: np.int32
  - `is padded`: True
  - `fill value`: -1
  - `note`: pre-ordered list
- **`list_image_filenames_per_supercategory`: list of image filenames per supercategory**
  - `available in`: train, val
  - `dtype`: np.int32

        - `is padded`: True
        - `fill value`: -1
        - `note`: pre-ordered list
- **`list_object_ids_per_image`: list of object ids per image**
        - `available in`: train, val, test, test_dev
        - `dtype`: np.int32
        - `is padded`: True
        - `fill value`: -1
        - `note`: pre-ordered list
- **`list_objects_ids_per_category`: list of object ids per category**
        - `available in`: train, val
        - `dtype`: np.int32
        - `is padded`: True
        - `fill value`: -1
        - `note`: pre-ordered list
- **`list_objects_ids_per_supercategory`: list of object ids per supercategory**
        - `available in`: train, val
        - `dtype`: np.int32
        - `is padded`: True
        - `fill value`: -1
        - `note`: pre-ordered list

Task: caption_2015

```
/
├── train/
│   ├── image_filenames       # dtype=np.uint8, shape=(82783,74)    (note: string in␣
↪ASCII format)
│   ├── captions              # dtype=np.uint8, shape=(414133,251)    (note: string in␣
↪ASCII format)
│   ├── coco_images_ids       # dtype=np.int32, shape=(82783,)
│   ├── coco_urls             # dtype=np.uint8, shape=(82783,32)    (note: string in␣
↪ASCII format)
│   ├── image_id              # dtype=np.int32, shape=(82783,)
│   ├── width                 # dtype=np.int32, shape=(82783,)
│   ├── height                # dtype=np.int32, shape=(82783,)
│   ├── object_fields         # dtype=np.uint8, shape=(5,16)       (note: string in␣
↪ASCII format)
│   ├── object_ids            # dtype=np.int32, shape=(414133,5)
│   ├── list_object_ids_per_image    # dtype=np.int32, shape=(82783,7))
│   └── list_captions_per_image      # dtype=np.int32, shape=(82783,7))
│
├── val/
```

(continues on next page)

(continued from previous page)

```
│    ├── image_filenames       # dtype=np.uint8, shape=(40504,70)    (note: string in␣
↪ASCII format)
│    ├── captions              # dtype=np.uint8, shape=(202654,74)    (note: string in␣
↪ASCII format)
│    ├── coco_images_ids       # dtype=np.int32, shape=(40504,)
│    ├── coco_urls             # dtype=np.uint8, shape=(40504,32)    (note: string in␣
↪ASCII format)
│    ├── image_id              # dtype=np.int32, shape=(40504,)
│    ├── width                 # dtype=np.int32, shape=(40504,)
│    ├── height                # dtype=np.int32, shape=(40504,)
│    ├── object_fields         # dtype=np.uint8, shape=(5,16)       (note: string in␣
↪ASCII format)
│    ├── object_ids            # dtype=np.int32, shape=(202654,5)
│    ├── list_object_ids_per_image   # dtype=np.int32, shape=(40504,7))
│    └── list_captions_per_image     # dtype=np.int32, shape=(40504,7))
│
└── test/
    ├── image_filenames       # dtype=np.uint8, shape=(40775,72)    (note: string in␣
↪ASCII format)
    ├── category              # dtype=np.uint8, shape=(80,15)       (note: string in␣
↪ASCII format)
    ├── supercategory         # dtype=np.uint8, shape=(12,11)       (note: string in␣
↪ASCII format)
    ├── coco_categories_ids   # dtype=np.int32, shape=(80,)
    ├── coco_images_ids       # dtype=np.int32, shape=(40775,)
    ├── coco_urls             # dtype=np.uint8, shape=(40775,32)    (note: string in␣
↪ASCII format)
    ├── image_id              # dtype=np.int32, shape=(40775,)
    ├── category_id           # dtype=np.int32, shape=(80,)
    ├── width                 # dtype=np.int32, shape=(40775,)
    ├── height                # dtype=np.int32, shape=(40775,)
    ├── object_fields         # dtype=np.uint8, shape=(4,16)       (note: string in␣
↪ASCII format)
    ├── object_ids            # dtype=np.int32, shape=(40775,4)
    └── list_object_ids_per_image   # dtype=np.int32, shape=(40775,1))
```

Fields

- **`image_filenames`: image file path+names**
  - `available in`: train, val, test
  - `dtype`: np.uint8
  - `is padded`: True
  - `fill value`: 0
  - `note`: strings stored in ASCII format
- **`captions`: image captions**
  - `available in`: train, val
  - `dtype`: np.uint8
  - `is padded`: True
  - `fill value`: 0

  - `note`: strings stored in ASCII format

- **`category`: category names**
  - `available in`: test
  - `dtype`: np.uint8
  - `is padded`: True
  - `fill value`: 0
  - `note`: strings stored in ASCII format
- **`supercategory`: super category names**
  - `available in`: test
  - `dtype`: np.uint8
  - `is padded`: True
  - `fill value`: 0
  - `note`: strings stored in ASCII format
- **`coco_categories_ids`: reference to coco category ids (useful for evaluating on coco)**
  - `available in`: test
  - `dtype`: np.int32
  - `is padded`: False
  - `fill value`: -1
- **`coco_images_ids`: reference to coco image filename ids (useful for evaluating on coco)**
  - `available in`: train, val, test
  - `dtype`: np.int32
  - `is padded`: False
  - `fill value`: -1
- **`coco_urls`: coco urls**
  - `available in`: train, val, test
  - `dtype`: np.uint8
  - `is padded`: True
  - `fill value`: 0
  - `note`: strings stored in ASCII format
- **`image_id`: image filename ids**
  - `available in`: train, val, test
  - `dtype`: np.int32
  - `is padded`: False
  - `fill value`: -1
- **`category_id`: category ids**
  - `available in`: test

  - `dtype`: np.int32
  - `is padded`: False
  - `fill value`: -1

- **`width`: image width**
  - `available in`: train, val, test
  - `dtype`: np.int32
  - `is padded`: False
  - `fill value`: -1

- **`height`: image height**
  - `available in`: train, val, test
  - `dtype`: np.int32
  - `is padded`: False
  - `fill value`: -1

- **`object_fields`: list of field names of the object id list**
  - `available in`: train, val, test
  - `dtype`: np.uint8
  - `is padded`: True
  - `fill value`: 0
  - `note`: strings stored in ASCII format
  - `note`: key field (*field name* aggregator)

- **`object_ids`: list of field ids**
  - `available in`: train, val, test
  - `dtype`: np.int32
  - `is padded`: False
  - `fill value`: -1
  - `note`: key field (*field id* aggregator)

- **`list_object_ids_per_image`: list of object ids per image**
  - `available in`: train, val, test
  - `dtype`: np.int32
  - `is padded`: True
  - `fill value`: -1
  - `note`: pre-ordered list

- **`list_captions_per_image`: list of captions per image**
  - `available in`: train, val
  - `dtype`: np.int32
  - `is padded`: True

- `fill value`: -1
- `note`: pre-ordered list

Task: caption_2016

```
/
├── train/
│   ├── image_filenames       # dtype=np.uint8, shape=(82783,74)   (note: string in␣
↪ASCII format)
│   ├── captions              # dtype=np.uint8, shape=(414133,251)   (note: string in␣
↪ASCII format)
│   ├── coco_images_ids       # dtype=np.int32, shape=(82783,)
│   ├── coco_urls             # dtype=np.uint8, shape=(82783,32)   (note: string in␣
↪ASCII format)
│   ├── image_id              # dtype=np.int32, shape=(82783,)
│   ├── width                 # dtype=np.int32, shape=(82783,)
│   ├── height                # dtype=np.int32, shape=(82783,)
│   ├── object_fields         # dtype=np.uint8, shape=(5,16)       (note: string in␣
↪ASCII format)
│   ├── object_ids            # dtype=np.int32, shape=(414133,5)
│   ├── list_object_ids_per_image   # dtype=np.int32, shape=(82783,7))
│   └── list_captions_per_image     # dtype=np.int32, shape=(82783,7))
│
├── val/
│   ├── image_filenames       # dtype=np.uint8, shape=(40504,70)   (note: string in␣
↪ASCII format)
│   ├── captions              # dtype=np.uint8, shape=(202654,74)   (note: string in␣
↪ASCII format)
│   ├── coco_images_ids       # dtype=np.int32, shape=(40504,)
│   ├── coco_urls             # dtype=np.uint8, shape=(40504,32)   (note: string in␣
↪ASCII format)
│   ├── image_id              # dtype=np.int32, shape=(40504,)
│   ├── width                 # dtype=np.int32, shape=(40504,)
│   ├── height                # dtype=np.int32, shape=(40504,)
│   ├── object_fields         # dtype=np.uint8, shape=(5,16)       (note: string in␣
↪ASCII format)
│   ├── object_ids            # dtype=np.int32, shape=(202654,5)
│   ├── list_object_ids_per_image   # dtype=np.int32, shape=(40504,7))
│   └── list_captions_per_image     # dtype=np.int32, shape=(40504,7))
│
├── test/
│   ├── image_filenames       # dtype=np.uint8, shape=(81434,72)   (note: string in␣
↪ASCII format)
│   ├── category              # dtype=np.uint8, shape=(80,15)       (note: string in␣
↪ASCII format)
│   ├── supercategory         # dtype=np.uint8, shape=(12,11)       (note: string in␣
↪ASCII format)
│   ├── coco_categories_ids   # dtype=np.int32, shape=(80,)
│   ├── coco_images_ids       # dtype=np.int32, shape=(81434,)
│   ├── coco_urls             # dtype=np.uint8, shape=(81434,32)   (note: string in␣
↪ASCII format)
│   ├── image_id              # dtype=np.int32, shape=(81434,)
│   ├── category_id           # dtype=np.int32, shape=(80,)
│   ├── width                 # dtype=np.int32, shape=(81434,)
│   ├── height                # dtype=np.int32, shape=(81434,)
│   ├── object_fields         # dtype=np.uint8, shape=(4,16)       (note: string in␣
↪ASCII format)
```

(continues on next page)

(continued from previous page)

```
│    ├── object_ids            # dtype=np.int32, shape=(81434,4)
│    └── list_object_ids_per_image   # dtype=np.int32, shape=(81434,1))
│
└── test_dev/
    ├── image_filenames       # dtype=np.uint8, shape=(20288,72)    (note: string in␣
↪ASCII format)
    ├── category              # dtype=np.uint8, shape=(80,15)       (note: string in␣
↪ASCII format)
    ├── supercategory         # dtype=np.uint8, shape=(12,11)       (note: string in␣
↪ASCII format)
    ├── coco_categories_ids   # dtype=np.int32, shape=(80,)
    ├── coco_images_ids       # dtype=np.int32, shape=(20288,)
    ├── coco_urls             # dtype=np.uint8, shape=(20288,32)    (note: string in␣
↪ASCII format)
    ├── image_id              # dtype=np.int32, shape=(20288,)
    ├── category_id           # dtype=np.int32, shape=(80,)
    ├── width                 # dtype=np.int32, shape=(20288,)
    ├── height                # dtype=np.int32, shape=(20288,)
    ├── object_fields         # dtype=np.uint8, shape=(4,16)       (note: string in␣
↪ASCII format)
    ├── object_ids            # dtype=np.int32, shape=(20288,4)
    └── list_object_ids_per_image   # dtype=np.int32, shape=(20288,1))
```

Fields

- **`image_filenames`: image file path+names**
  - `available in`: train, val, test, test_dev
  - `dtype`: np.uint8
  - `is padded`: True
  - `fill value`: 0
  - `note`: strings stored in ASCII format
- **`captions`: image captions**
  - `available in`: train, val
  - `dtype`: np.uint8
  - `is padded`: True
  - `fill value`: 0
  - `note`: strings stored in ASCII format
- **`category`: category names**
  - `available in`: test, test_dev
  - `dtype`: np.uint8
  - `is padded`: True
  - `fill value`: 0
  - `note`: strings stored in ASCII format
- **`supercategory`: super category names**

  - `available in`: test, test_dev
  - `dtype`: np.uint8
  - `is padded`: True
  - `fill value`: 0
  - `note`: strings stored in ASCII format
- **`coco_categories_ids`: reference to coco category ids (useful for evaluating on coco)**
  - `available in`: test, test_dev
  - `dtype`: np.int32
  - `is padded`: False
  - `fill value`: -1
- **`coco_images_ids`: reference to coco image filename ids (useful for evaluating on coco)**
  - `available in`: train, val, test, test_dev
  - `dtype`: np.int32
  - `is padded`: False
  - `fill value`: -1
- **`coco_urls`: coco urls**
  - `available in`: train, val, test, test_dev
  - `dtype`: np.uint8
  - `is padded`: True
  - `fill value`: 0
  - `note`: strings stored in ASCII format
- **`image_id`: image filename ids**
  - `available in`: train, val, test, test_dev
  - `dtype`: np.int32
  - `is padded`: False
  - `fill value`: -1
- **`category_id`: category ids**
  - `available in`: test, test_dev
  - `dtype`: np.int32
  - `is padded`: False
  - `fill value`: -1
- **`width`: image width**
  - `available in`: train, val, test, test_dev
  - `dtype`: np.int32
  - `is padded`: False
  - `fill value`: -1

- **height: image height**
  - available in: train, val, test, test_dev
  - dtype: np.int32
  - is padded: False
  - fill value: -1
- **object_fields: list of field names of the object id list**
  - available in: train, val, test, test_dev
  - dtype: np.uint8
  - is padded: True
  - fill value: 0
  - note: strings stored in ASCII format
  - note: key field (*field name* aggregator)
- **object_ids: list of field ids**
  - available in: train, val, test, test_dev
  - dtype: np.int32
  - is padded: False
  - fill value: -1
  - note: key field (*field id* aggregator)
- **list_object_ids_per_image: list of object ids per image**
  - available in: train, val, test, test_dev
  - dtype: np.int32
  - is padded: True
  - fill value: -1
  - note: pre-ordered list
- **list_captions_per_image: list of captions per image**
  - available in: train, val
  - dtype: np.int32
  - is padded: True
  - fill value: -1
  - note: pre-ordered list

Task: keypoints_2016

```
/
├── train/
│   ├── image_filenames       # dtype=np.uint8, shape=(82783,74)   (note: string in␣
↪ASCII format)
│   ├── category              # dtype=np.uint8, shape=(80,15)      (note: string in␣
↪ASCII format)
```

(continues on next page)

(continued from previous page)

```
│   ├── supercategory          # dtype=np.uint8, shape=(12,11)       (note: string in␣
↪ASCII format)
│   ├── coco_annotations_ids   # dtype=np.int32, shape=(185316,)
│   ├── coco_categories_ids    # dtype=np.int32, shape=(80,)
│   ├── coco_images_ids        # dtype=np.int32, shape=(82783,)
│   ├── coco_urls              # dtype=np.uint8, shape=(82783,32)    (note: string in␣
↪ASCII format)
│   ├── image_id               # dtype=np.int32, shape=(82783,)
│   ├── category_id            # dtype=np.int32, shape=(80,)
│   ├── annotation_id          # dtype=np.int32, shape=(185316,)
│   ├── width                  # dtype=np.int32, shape=(82783,)
│   ├── height                 # dtype=np.int32, shape=(82783,)
│   ├── boxes                  # dtype=np.float, shape=(185316,4)
│   ├── iscrowd                # dtype=np.uint8, shape=(2,)
│   ├── segmentation           # dtype=np.float, shape=(185316,10043)
│   ├── area                   # dtype=np.int32, shape=(185316,)
│   ├── keypoint_names         # dtype=np.uint8, shape=(17,15)       (note: string in␣
↪ASCII format)
│   ├── keypoints              # dtype=np.int32, shape=(185316,51)
│   ├── num_keypoints          # dtype=np.uint8, shape=(18,)
│   ├── skeleton               # dtype=np.uint8, shape=(19,2)
│   ├── object_fields          # dtype=np.uint8, shape=(13,16)       (note: string in␣
↪ASCII format)
│   ├── object_ids             # dtype=np.int32, shape=(185316,13)
│   ├── list_boxes_per_image                      # dtype=np.int32, shape=(82783,20))
│   ├── list_image_filenames_per_num_keypoints    # dtype=np.int32, shape=(17,45174))
│   ├── list_keypoints_per_image                  # dtype=np.int32, shape=(82783,20))
│   ├── list_object_ids_per_image                 # dtype=np.int32, shape=(82783,20))
│   └── list_object_ids_per_keypoint              # dtype=np.int32, shape=(17,92701))
│
├── val/
│   ├── image_filenames        # dtype=np.uint8, shape=(40504,70)    (note: string in␣
↪ASCII format)
│   ├── category               # dtype=np.uint8, shape=(80,15)       (note: string in␣
↪ASCII format)
│   ├── supercategory          # dtype=np.uint8, shape=(12,11)       (note: string in␣
↪ASCII format)
│   ├── coco_annotations_ids   # dtype=np.int32, shape=(88153,)
│   ├── coco_categories_ids    # dtype=np.int32, shape=(80,)
│   ├── coco_images_ids        # dtype=np.int32, shape=(40504,)
│   ├── coco_urls              # dtype=np.uint8, shape=(40504,32)    (note: string in␣
↪ASCII format)
│   ├── image_id               # dtype=np.int32, shape=(40504,)
│   ├── category_id            # dtype=np.int32, shape=(80,)
│   ├── annotation_id          # dtype=np.int32, shape=(88153,)
│   ├── width                  # dtype=np.int32, shape=(40504,)
│   ├── height                 # dtype=np.int32, shape=(40504,)
│   ├── boxes                  # dtype=np.float, shape=(88153,4)
│   ├── iscrowd                # dtype=np.uint8, shape=(2,)
│   ├── segmentation           # dtype=np.float, shape=(88153,6121)
│   ├── area                   # dtype=np.int32, shape=(88153,)
│   ├── keypoint_names         # dtype=np.uint8, shape=(17,15)       (note: string in␣
↪ASCII format)
│   ├── keypoints              # dtype=np.int32, shape=(88153,51)
│   ├── num_keypoints          # dtype=np.uint8, shape=(18,)
│   ├── skeleton               # dtype=np.uint8, shape=(19,2)
│   ├── object_fields          # dtype=np.uint8, shape=(13,16)       (note: string in␣
↪ASCII format)
```

(continues on next page)

(continued from previous page)

```
│   ├── object_ids            # dtype=np.int32, shape=(88153,13)
│   ├── list_boxes_per_image                    # dtype=np.int32, shape=(40504,16))
│   ├── list_image_filenames_per_num_keypoints  # dtype=np.int32, shape=(17,21634))
│   ├── list_keypoints_per_image                # dtype=np.int32, shape=(40504,16))
│   ├── list_object_ids_per_image               # dtype=np.int32, shape=(40504,16))
│   └── list_object_ids_per_keypoint            # dtype=np.int32, shape=(17,43971))
│
├── test/
│   ├── image_filenames       # dtype=np.uint8, shape=(81434,72)    (note: string in␣
↪ASCII format)
│   ├── category              # dtype=np.uint8, shape=(80,15)       (note: string in␣
↪ASCII format)
│   ├── supercategory         # dtype=np.uint8, shape=(12,11)       (note: string in␣
↪ASCII format)
│   ├── coco_categories_ids   # dtype=np.int32, shape=(80,)
│   ├── coco_images_ids       # dtype=np.int32, shape=(81434,)
│   ├── coco_urls             # dtype=np.uint8, shape=(81434,32)    (note: string in␣
↪ASCII format)
│   ├── image_id              # dtype=np.int32, shape=(81434,)
│   ├── category_id           # dtype=np.int32, shape=(80,)
│   ├── width                 # dtype=np.int32, shape=(81434,)
│   ├── height                # dtype=np.int32, shape=(81434,)
│   ├── object_fields         # dtype=np.uint8, shape=(4,16)       (note: string in␣
↪ASCII format)
│   ├── object_ids            # dtype=np.int32, shape=(81434,4)
│   └── list_object_ids_per_image   # dtype=np.int32, shape=(81434,1))
│
└── test_dev/
    ├── image_filenames       # dtype=np.uint8, shape=(20288,72)    (note: string in␣
↪ASCII format)
    ├── category              # dtype=np.uint8, shape=(80,15)       (note: string in␣
↪ASCII format)
    ├── supercategory         # dtype=np.uint8, shape=(12,11)       (note: string in␣
↪ASCII format)
    ├── coco_categories_ids   # dtype=np.int32, shape=(80,)
    ├── coco_images_ids       # dtype=np.int32, shape=(20288,)
    ├── coco_urls             # dtype=np.uint8, shape=(20288,32)    (note: string in␣
↪ASCII format)
    ├── image_id              # dtype=np.int32, shape=(20288,)
    ├── category_id           # dtype=np.int32, shape=(80,)
    ├── width                 # dtype=np.int32, shape=(20288,)
    ├── height                # dtype=np.int32, shape=(20288,)
    ├── object_fields         # dtype=np.uint8, shape=(4,16)       (note: string in␣
↪ASCII format)
    ├── object_ids            # dtype=np.int32, shape=(20288,4)
    └── list_object_ids_per_image   # dtype=np.int32, shape=(20288,1))
```

Fields

- **`image_filenames`: image file path+names**
  - `available in`: train, val, test, test_dev
  - `dtype`: np.uint8
  - `is padded`: True

        - `fill value`: 0
        - `note`: strings stored in ASCII format
- **`category`: category names**
        - `available in`: train, val, test, test_dev
        - `dtype`: np.uint8
        - `is padded`: True
        - `fill value`: 0
        - `note`: strings stored in ASCII format
- **`supercategory`: super category names**
        - `available in`: train, val, test, test_dev
        - `dtype`: np.uint8
        - `is padded`: True
        - `fill value`: 0
        - `note`: strings stored in ASCII format
- **`coco_annotations_ids`: reference to coco annotation ids (useful for evaluating on coco)**
        - `available in`: train, val
        - `dtype`: np.int32
        - `is padded`: False
        - `fill value`: -1
- **`coco_categories_ids`: reference to coco category ids (useful for evaluating on coco)**
        - `available in`: train, val, test, test_dev
        - `dtype`: np.int32
        - `is padded`: False
        - `fill value`: -1
- **`coco_images_ids`: reference to coco image filename ids (useful for evaluating on coco)**
        - `available in`: train, val, test, test_dev
        - `dtype`: np.int32
        - `is padded`: False
        - `fill value`: -1
- **`coco_urls`: coco urls**
        - `available in`: train, val, test, test_dev
        - `dtype`: np.uint8
        - `is padded`: True
        - `fill value`: 0
        - `note`: strings stored in ASCII format
- **`image_id`: image filename ids**

        - `available in`: train, val, test, test_dev
        - `dtype`: np.int32
        - `is padded`: False
        - `fill value`: -1
- **`category_id`: category ids**
        - `available in`: train, val, test, test_dev
        - `dtype`: np.int32
        - `is padded`: False
        - `fill value`: -1
- **`annotation_id`: annotation ids**
        - `available in`: train, val
        - `dtype`: np.int32
        - `is padded`: False
        - `fill value`: -1
- **`width`: image width**
        - `available in`: train, val, test, test_dev
        - `dtype`: np.int32
        - `is padded`: False
        - `fill value`: -1
- **`height`: image height**
        - `available in`: train, val, test, test_dev
        - `dtype`: np.int32
        - `is padded`: False
        - `fill value`: -1
- **`boxes`: bounding box**
        - `available in`: train, val
        - `dtype`: np.float
        - `is padded`: False
        - `fill value`: -1
        - `note`: bbox format (x1,y1,x2,y2)
- **`iscrowd`: is crowd (0 - False, 1 - True)**
        - `available in`: train, val
        - `dtype`: np.uint8
        - `is padded`: False
        - `fill value`: -1
- **`segmentation`: segmentation mask**

  - `available in`: train, val
  - `dtype`: np.float
  - `is padded`: True
  - `fill value`: -1
  - `note`: the masks come in 3 different formats, but they are mostly lists of lists. These have been packed (vectorized) into an array with a single dimension in order to be stored in the HDF5 metadata file. To unpack these arrays to their original format, use the `unsqueeze_list()` method in `dbcollection.utils.pad`.

- **`area`: object area**
  - `available in`: train, val
  - `dtype`: np.int32
  - `is padded`: False
  - `fill value`: -1
- **`keypoint_names`: body joint names**
  - `available in`: train, val
  - `dtype`: np.uint8
  - `is padded`: True
  - `fill value`: 0
  - `note`: strings stored in ASCII format
- **`keypoints`: body joint coordinates**
  - `available in`: train, val
  - `dtype`: np.int32
  - `is padded`: False
  - `fill value`: -1
  - `note`: coordinates format [x1,y1,is_visible,x2,y2,is_visible, ...]
- **`num_keypoints`: number of body joints**
  - `available in`: train, val
  - `dtype`: np.uint8
  - `is padded`: False
  - `fill value`: -1
- **`skeleton`: pairwise body joints**
  - `available in`: train, val
  - `dtype`: np.uint8
  - `is padded`: False
  - `fill value`: -1
- **`object_fields`: list of field names of the object id list**
  - `available in`: train, val, test, test_dev

  - `dtype`: np.uint8
  - `is padded`: True
  - `fill value`: 0
  - `note`: strings stored in ASCII format
  - `note`: key field (*field name* aggregator)
- **`object_ids`: list of field ids**
  - `available in`: train, val, test, test_dev
  - `dtype`: np.int32
  - `is padded`: False
  - `fill value`: -1
  - `note`: key field (*field id* aggregator)
- **`list_boxes_per_image`: list of bounding boxes per image**
  - `available in`: train, val
  - `dtype`: np.int32
  - `is padded`: True
  - `fill value`: -1
  - `note`: pre-ordered list
- **`list_image_filenames_per_category`: list of image filenames per category**
  - `available in`: train, val
  - `dtype`: np.int32
  - `is padded`: True
  - `fill value`: -1
  - `note`: pre-ordered list
- **`list_image_filenames_per_supercategory`: list of image filenames per supercategory**
  - `available in`: train, val
  - `dtype`: np.int32
  - `is padded`: True
  - `fill value`: -1
  - `note`: pre-ordered list
- **`list_object_ids_per_image`: list of object ids per image**
  - `available in`: train, val, test, test_dev
  - `dtype`: np.int32
  - `is padded`: True
  - `fill value`: -1
  - `note`: pre-ordered list
- **`list_objects_ids_per_category`: list of object ids per category**

  - `available in`: train, val
  - `dtype`: np.int32
  - `is padded`: True
  - `fill value`: -1
  - `note`: pre-ordered list
- **`list_objects_ids_per_supercategory`: list of object ids per supercategory**
  - `available in`: train, val
  - `dtype`: np.int32
  - `is padded`: True
  - `fill value`: -1
  - `note`: pre-ordered list

Disclaimer

All rights reserved to the original creators of **MS COCO**.

For information about the dataset and its terms of use, please see this link.

FLIC - Frames Labeled In Cinema

Collection of 5003 images automatically from popular Hollywood movies. It contains annotations of upper body joints only. It has 3987 images for training and 1016 for testing.

Use cases

Human body joint detection.

Properties

- `name`: flic
- `keywords`: image_processing, detection, human_pose, keypoints
- `dataset size`: 300,3 MB
- `is downloadable`: **yes**
- **`tasks`:**
  - **keypoints: (default)**
    - `primary use`: image classification
    - `description`: Contains keypoint coordinates (upper body joints only) and bounding boxes of the torso for body joint detection.
    - `sets`: train, test
    - `metadata file size`: 582,0 kB
    - **`has annotations`: yes**

- **which:**

  image filenames

  upper body joint coordinates

  torso bounding box

Metadata structure (HDF5)

Task: classification

```
/
├── train/
│   ├── image_filenames   # dtype=np.uint8, shape=(3987,108)  (note: string in ASCII␣
↪format)
│   ├── movienames        # dtype=np.uint8, shape=(3987,31)   (note: string in ASCII␣
↪format)
│   ├── width             # dtype=np.int32, shape=(3987,)
│   ├── height            # dtype=np.int32, shape=(3987,)
│   ├── torso_boxes       # dtype=np.float, shape=(3987,4)
│   ├── keypoints         # dtype=np.float, shape=(3987,11,3)
│   ├── keypoint_names    # dtype=np.uint8, shape=(11,15)    (note: string in ASCII␣
↪format)
│   ├── object_fields     # dtype=np.uint8, shape=(5,16)     (note: string in ASCII␣
↪format)
│   └── object_ids        # dtype=np.int32, shape=(3987,5)
│
└── test/
    ├── image_filenames   # dtype=np.uint8, shape=(1016,108)  (note: string in ASCII␣
↪format)
    ├── movienames        # dtype=np.uint8, shape=(1016,31)   (note: string in ASCII␣
↪format)
    ├── width             # dtype=np.int32, shape=(1016,)
    ├── height            # dtype=np.int32, shape=(1016,)
    ├── torso_boxes       # dtype=np.float, shape=(1016,4)
    ├── keypoints         # dtype=np.float, shape=(1016,11,3)
    ├── keypoint_names    # dtype=np.uint8, shape=(11,15)    (note: string in ASCII␣
↪format)
    ├── object_fields     # dtype=np.uint8, shape=(5,16)     (note: string in ASCII␣
↪format)
    └── object_ids        # dtype=np.int32, shape=(1016,5)
```

Fields

- **`image_filenames`: image file path + name**
  - `available in`: train, test
  - `dtype`: np.uint8
  - `is padded`: True
  - `fill value`: 0
  - `note`: strings stored in ASCII format
- **`movienames`: name of the movie where the image was taken from**

        - `available in`: train, test
        - `dtype`: np.uint8
        - `is padded`: True
        - `fill value`: 0
        - `note`: strings stored in ASCII format
- **`width`: image width**
        - `available in`: train, test
        - `dtype`: np.int32
        - `is padded`: False
        - `fill value`: -1
- **`height`: image height**
        - `available in`: train, test
        - `dtype`: np.int32
        - `is padded`: False
        - `fill value`: -1
- **`torso_boxes`: torso bounding box**
        - `available in`: train, test
        - `dtype`: np.float
        - `is padded`: False
        - `fill value`: -1
        - `note`: bbox format [x1,y1,x2,y2]
- **`keypoints`: body joint coordinates**
        - `available in`: train, test
        - `dtype`: np.float
        - `is padded`: False
        - `fill value`: -1
        - `note`: keypoint format [x1,y1,is_visible]
- **`keypoint_names`: body joint name**
        - `available in`: train, test
        - `dtype`: np.uint8
        - `is padded`: True
        - `fill value`: 0
        - `note`: strings stored in ASCII format
- **`object_fields`: list of field names of the object id list**
        - `available in`: train, test
        - `dtype`: np.uint8

    - `is padded`: True
    - `fill value`: 0
    - `note`: strings stored in ASCII format
    - `note`: key field (*field name* aggregator)
- **`object_ids`: list of field ids**
    - `available in`: train, test
    - `dtype`: np.int32
    - `is padded`: False
    - `fill value`: -1
    - `note`: key field (*field id* aggregator)

Disclaimer

All rights reserved to the original creators of **Frames Labeled In Cinema**.

For information about the dataset and its terms of use, please see this link.

ILSVRC2012 - Imagenet Large Scale Visual Recognition Challenge 2012

ImageNet is an image database organized according to the **WordNet** hierarchy (currently only the nouns), in which each node of the hierarchy is depicted by hundreds and thousands of images.

The **Large Scale Visual Recognition Challenge 2012 (ILSVRC2012)** is a subset of the large hand-labeled ImageNet dataset (10,000,000 labeled images depicting 10,000+ object categories). The training data is a subset of **ImageNet** containing the 1000 categories and 1.2 million images

Use cases

Image classification.

Properties

- `name`: ilsvrc2012
- `keywords`: image_processing, classification
- `dataset size`: 154,6 GB
- **`is downloadable`: no**
    - `data setup`: create a folder or symlink with the name `ilsvrc2012/` where you have stored and unpacked the data files and, when loading the dataset, use the `data_dir` input argument to specify the data's folder path.
- **`tasks`:**
    - **classification: (default)**
        * `primary use`: image classification

* `description`: Contains image filenames and label annotations for image classification.
* `sets`: train, val
* `metadata file size in disk`: 6,8 MB
* **`has annotations`: yes**
  - **`which`:**

    labels for each image class/category.

    descriptions for each class/category.

– **raw256:**

* `primary use`: image classification
* `description`: Contains image filenames and label annotations for image classification.
* `sets`: train, val
* `metadata file size in disk`: 6,8 MB
* **`has annotations`: yes**
  - **`which`:**

    labels for each image class/category.

    descriptions for each class/category.

Metadata structure (HDF5)

Task: classification

```
/
├── train/
│   ├── image_filenames   # dtype=np.uint8, shape=(1281166,76)  (note: string in␣
↪ASCII format)
│   ├── classes           # dtype=np.uint8, shape=(1000,10)       (note: string in␣
↪ASCII format)
│   ├── labels            # dtype=np.uint8, shape=(1000,122)
│   ├── descriptions      # dtype=np.uint8, shape=(1000,256)       (note: string in␣
↪ASCII format)
│   ├── object_fields     # dtype=np.uint8, shape=(2,16)         (note: string in␣
↪ASCII format)
│   ├── object_ids        # dtype=np.int32, shape=(1281166,2)
│   └── list_image_filenames_per_class   # dtype=np.int32, shape=(1000,1300))
│
└── val/
    ├── image_filenames   # dtype=np.uint8, shape=(50000,67)  (note: string in ASCII␣
↪format)
    ├── classes           # dtype=np.uint8, shape=(1000,10)       (note: string in␣
↪ASCII format)
    ├── labels            # dtype=np.uint8, shape=(1000,122)
    ├── descriptions      # dtype=np.uint8, shape=(1000,256)       (note: string in␣
↪ASCII format)
    ├── object_fields     # dtype=np.uint8, shape=(2,16)         (note: string in␣
↪ASCII format)
```

(continues on next page)

(continued from previous page)

```
    ├── object_ids        # dtype=np.int32, shape=(50000,2)
    └── list_image_filenames_per_class   # dtype=np.int32, shape=(1000,50))
```

Fields

- **`images`: image file path + name**
  - `available in`: train, val
  - `dtype`: np.uint8
  - `is padded`: True
  - `fill value`: 0
  - `note`: strings stored in ASCII format
- **`classes`: class names**
  - `available in`: train, val
  - `dtype`: np.uint8
  - `is padded`: True
  - `fill value`: 0
  - `note`: strings stored in ASCII format
- **`labels`: label names**
  - `available in`: train, val
  - `dtype`: np.uint8
  - `is padded`: True
  - `fill value`: 0
  - `note`: strings stored in ASCII format
- **`descriptions`: class descriptions**
  - `available in`: train, val
  - `dtype`: np.uint8
  - `is padded`: True
  - `fill value`: 0
  - `note`: strings stored in ASCII format
- **`object_fields`: list of field names of the object id list**
  - `available in`: train, val
  - `dtype`: np.uint8
  - `is padded`: True
  - `fill value`: 0
  - `note`: strings stored in ASCII format
  - `note`: key field (*field name* aggregator)

- **object_ids: list of field ids**
  - available in: train, val
  - dtype: np.int32
  - is padded: False
  - fill value: -1
  - note: key field (*field id* aggregator)
- **list_image_filenames_per_class: list of image filenames per class**
  - available in: train, val
  - dtype: np.int32
  - is padded: True
  - fill value: -1
  - note: pre-ordered list

Task: raw256

```
/
├── train/
│   ├── image_filenames   # dtype=np.uint8, shape=(1281166,76)  (note: string in␣
↪ASCII format)
│   ├── classes           # dtype=np.uint8, shape=(1000,10)       (note: string in␣
↪ASCII format)
│   ├── labels            # dtype=np.uint8, shape=(1000,122)
│   ├── descriptions      # dtype=np.uint8, shape=(1000,256)       (note: string in␣
↪ASCII format)
│   ├── object_fields     # dtype=np.uint8, shape=(2,16)         (note: string in␣
↪ASCII format)
│   ├── object_ids        # dtype=np.int32, shape=(1281166,2)
│   └── list_image_filenames_per_class   # dtype=np.int32, shape=(1000,1300))
│
└── val/
    ├── image_filenames   # dtype=np.uint8, shape=(50000,67)  (note: string in ASCII␣
↪format)
    ├── classes           # dtype=np.uint8, shape=(1000,10)       (note: string in␣
↪ASCII format)
    ├── labels            # dtype=np.uint8, shape=(1000,122)
    ├── descriptions      # dtype=np.uint8, shape=(1000,256)       (note: string in␣
↪ASCII format)
    ├── object_fields     # dtype=np.uint8, shape=(2,16)         (note: string in␣
↪ASCII format)
    ├── object_ids        # dtype=np.int32, shape=(50000,2)
    └── list_image_filenames_per_class   # dtype=np.int32, shape=(1000,50))
```

Fields

- **images: image file path + name**
  - available in: train, val
  - dtype: np.uint8

  - `is padded`: True
  - `fill value`: 0
  - `note`: strings stored in ASCII format
- **`classes`: class names**
  - `available in`: train, val
  - `dtype`: np.uint8
  - `is padded`: True
  - `fill value`: 0
  - `note`: strings stored in ASCII format
- **`labels`: label names**
  - `available in`: train, val
  - `dtype`: np.uint8
  - `is padded`: True
  - `fill value`: 0
  - `note`: strings stored in ASCII format
- **`descriptions`: class descriptions**
  - `available in`: train, val
  - `dtype`: np.uint8
  - `is padded`: True
  - `fill value`: 0
  - `note`: strings stored in ASCII format
- **`object_fields`: list of field names of the object id list**
  - `available in`: train, val
  - `dtype`: np.uint8
  - `is padded`: True
  - `fill value`: 0
  - `note`: strings stored in ASCII format
  - `note`: key field (*field name* aggregator)
- **`object_ids`: list of field ids**
  - `available in`: train, val
  - `dtype`: np.int32
  - `is padded`: False
  - `fill value`: -1
  - `note`: key field (*field id* aggregator)
- **`list_image_filenames_per_class`: list of image filenames per class**
  - `available in`: train, val

- `dtype`: np.int32
- `is padded`: True
- `fill value`: -1
- `note`: pre-ordered list

Disclaimer

All rights reserved to the original creators of **ILSVRC2012**.

For information about the dataset and its terms of use, please see this link.

INRIA Pedestrian

The **INRIA person** dataset is popular in the Pedestrian Detection community, both for training detectors and reporting results.

It consists of 614 person detections for training and 288 for testing.

Note: The data files available for download are the ones distributed in here.

Use cases

Pedestrian detection in images.

Properties

- `name`: inria_pedestrian
- `keywords`: image_processing, detection, pedestrian
- `dataset size`: 1,1 GB
- `is downloadable`: **yes**
- **`tasks:`**
  - **detection: (default)**
    - `primary use`: object detection
    - `description`: Contains image filenames, classes and bounding box annotations for pedestrian detection in images/videos.
    - `sets`: train, test
    - `metadata file size in disk`: 139,1 kB
    - **`has annotations:` yes**
      - **`which:`**

        labels for each class/category.

        bounding box of pedestrians.

occlusion % of annotated pedestrians.

Metadata structure (HDF5)

Task: detection

```
/
├── train/
│   ├── image_filenames   # dtype=np.uint8, shape=(1832,88)  (note: string in ASCII␣
↪format)
│   ├── classes           # dtype=np.uint8, shape=(4,10)     (note: string in ASCII␣
↪format)
│   ├── boxes             # dtype=np.float, shape=(1237,4)
│   ├── boxesv            # dtype=np.float, shape=(1237,4)
│   ├── id                # dtype=np.int32, shape=(1237,)
│   ├── occlusion         # dtype=np.float, shape=(1237,)
│   ├── object_fields     # dtype=np.uint8, shape=(6,16)     (note: string in ASCII␣
↪format)
│   ├── object_ids        # dtype=np.int32, shape=(1237,6)
│   ├── list_image_filenames_per_class   # dtype=np.int32, shape=(4,614))
│   ├── list_boxes_per_image             # dtype=np.int32, shape=(1832,12))
│   ├── list_boxesv_per_image            # dtype=np.int32, shape=(1832,12))
│   ├── list_object_ids_per_image        # dtype=np.int32, shape=(1832,12))
│   └── list_objects_ids_per_class       # dtype=np.int32, shape=(4,1237))
│
└── test/
    ├── image_filenames   # dtype=np.uint8, shape=(741,88)  (note: string in ASCII␣
↪format)
    ├── classes           # dtype=np.uint8, shape=(4,10)     (note: string in ASCII␣
↪format)
    ├── boxes             # dtype=np.float, shape=(589,4)
    ├── boxesv            # dtype=np.float, shape=(589,4)
    ├── id                # dtype=np.int32, shape=(589,)
    ├── occlusion         # dtype=np.float, shape=(589,)
    ├── object_fields     # dtype=np.uint8, shape=(6,16)     (note: string in ASCII␣
↪format)
    ├── object_ids        # dtype=np.int32, shape=(589,6)
    ├── list_image_filenames_per_class   # dtype=np.int32, shape=(4,288))
    ├── list_boxes_per_image             # dtype=np.int32, shape=(741,16))
    ├── list_boxesv_per_image            # dtype=np.int32, shape=(741,16))
    ├── list_object_ids_per_image        # dtype=np.int32, shape=(741,16))
    └── list_objects_ids_per_class       # dtype=np.int32, shape=(4,589))
```

Fields

- **`image_filenames`: image file path+names**
  - `available in`: train, test
  - `dtype`: np.uint8
  - `is padded`: True
  - `fill value`: 0
  - `note`: strings stored in ASCII format

- **classes: class names**
  - available in: train, test
  - dtype: np.uint8
  - is padded: True
  - fill value: 0
  - note: strings stored in ASCII format
- **boxes: bounding boxes**
  - available in: train, test
  - dtype: np.float
  - is padded: False
  - fill value: -1
  - note: bbox format (x1,y1,x2,y2)
- **boxesv: bounding boxes (visible)**
  - available in: train, test
  - dtype: np.float
  - is padded: False
  - fill value: -1
  - note: bbox format (x1,y1,x2,y2)
- **id: label ids**
  - available in: train, test
  - dtype: np.int32
  - is padded: False
  - fill value: -1
- **occlusion: occlusion percentage**
  - available in: train, test
  - dtype: np.float
  - is padded: False
  - fill value: -1
- **object_fields: list of field names of the object id list**
  - available in: train, test
  - dtype: np.uint8
  - is padded: True
  - fill value: 0
  - note: strings stored in ASCII format
  - note: key field (*field name* aggregator)
- **object_ids: list of field ids**

  - `available in`: train, test
  - `dtype`: np.int32
  - `is padded`: False
  - `fill value`: -1
  - `note`: key field (*field id* aggregator)
- **`list_image_filenames_per_class`: list of image per class**
  - `available in`: train, test
  - `dtype`: np.int32
  - `is padded`: True
  - `fill value`: -1
  - `note`: pre-ordered list
- **`list_boxes_per_image`: list of bounding boxes per image**
  - `available in`: train, test
  - `dtype`: np.int32
  - `is padded`: True
  - `fill value`: -1
  - `note`: pre-ordered list
- **`list_boxesv_per_image`: list of (visible) bounding boxes per image**
  - `available in`: train, test
  - `dtype`: np.int32
  - `is padded`: True
  - `fill value`: -1
  - `note`: pre-ordered list
- **`list_object_ids_per_image`: list of object ids per image**
  - `available in`: train, test
  - `dtype`: np.int32
  - `is padded`: True
  - `fill value`: -1
  - `note`: pre-ordered list
- **`list_objects_ids_per_class`: list of object ids per class**
  - `available in`: train, test
  - `dtype`: np.int32
  - `is padded`: True
  - `fill value`: -1
  - `note`: pre-ordered list

Disclaimer

All rights reserved to the original creators of **INRIA Pedestrian Dataset**.

For information about the dataset and its terms of use, please see this link.

LSP - Leeds Sports Pose

The **Leeds Sports Pose** dataset contains 2000 pose annotated images of mostly sports people gathered from Flickr using the tags shown above. The images have been scaled such that the most prominent person is roughly 150 pixels in length. Each image has been annotated with 14 joint locations. Left and right joints are consistently labelled from a person-centric viewpoint.

Use cases

Human body joint detection.

Properties

- `name`: leeds_sports_pose
- `keywords`: image_processing, detection, human_pose, keypoints
- `dataset size`: 264,2 MB
- `is downloadable`: **yes**
- **`tasks:`**
  - **keypoints: (default)**
    - `primary use`: human body joint detection
    - `description`: Contains image files and body parts keypoint coordinates for detecting human body joints in images
    - `sets`: train, test
    - `metadata file size in disk`: 473,7 kB
    - **`has annotations:` yes**
      - **`which:`**

        body joint keypoints
  - **keypoints_original:**
    - `primary use`: human body joint detection
    - `description`: Contains image files and body parts keypoint coordinates for detecting human body joints in images
    - `sets`: train, test
    - `metadata file size in disk`: 396,0 kB
    - **`has annotations:` yes**
      - **`which:`**

body joint keypoints

Note: The `keypoints_original` task is essentially the same as `keypoints`, but contains full size images instead of crops of persons.

Metadata structure (HDF5)

Task: keypoints

```
/
├── train/
│   ├── image_filenames   # dtype=np.uint8, shape=(1000,83)  (note: string in ASCII␣
↪format)
│   ├── keypoint_names    # dtype=np.uint8, shape=(14,15)    (note: string in ASCII␣
↪format)
│   ├── keypoints         # dtype=np.float, shape=(1000,14,3)
│   ├── object_fields     # dtype=np.uint8, shape=(2,16)     (note: string in ASCII␣
↪format)
│   └── object_ids        # dtype=np.int32, shape=(1000,2)
│
└── test/
    ├── image_filenames   # dtype=np.uint8, shape=(1000,83)  (note: string in ASCII␣
↪format)
    ├── keypoint_names    # dtype=np.uint8, shape=(14,15)    (note: string in ASCII␣
↪format)
    ├── keypoints         # dtype=np.float, shape=(1000,14,3)
    ├── object_fields     # dtype=np.uint8, shape=(2,16)     (note: string in ASCII␣
↪format)
    └── object_ids        # dtype=np.int32, shape=(1000,2)
```

Fields

- **`image_filenames`: image file path+name**
  - `available in`: train, test
  - `dtype`: np.uint8
  - `is padded`: True
  - `fill value`: 0
  - `note`: strings stored in ASCII format
- **`keypoint_names`: body joint names**
  - `available in`: train, test
  - `dtype`: np.uint8
  - `is padded`: True
  - `fill value`: 0
  - `note`: strings stored in ASCII format
- **`keypoints`: keypoint coordinates**

  - `available in`: train, test
  - `dtype`: np.float
  - `is padded`: False
  - `fill value`: -1
  - `note`: keypoint format [x1,y1,is_visible]
- **`object_fields`: list of field names of the object id list**
  - `available in`: train, test
  - `dtype`: np.uint8
  - `is padded`: True
  - `fill value`: 0
  - `note`: strings stored in ASCII format
  - `note`: key field (*field name* aggregator)
- **`object_ids`: list of field ids**
  - `available in`: train, test
  - `dtype`: np.int32
  - `is padded`: False
  - `fill value`: -1
  - `note`: key field (*field id* aggregator)

Task: keypoints_original

```
/
├── train/
│   ├── image_filenames   # dtype=np.uint8, shape=(1000,83)  (note: string in ASCII␣
↪format)
│   ├── keypoint_names    # dtype=np.uint8, shape=(14,15)    (note: string in ASCII␣
↪format)
│   ├── keypoints         # dtype=np.float, shape=(1000,14,3)
│   ├── object_fields     # dtype=np.uint8, shape=(2,16)     (note: string in ASCII␣
↪format)
│   └── object_ids        # dtype=np.int32, shape=(1000,2)
│
└── test/
    ├── image_filenames   # dtype=np.uint8, shape=(1000,83)  (note: string in ASCII␣
↪format)
    ├── keypoint_names    # dtype=np.uint8, shape=(14,15)    (note: string in ASCII␣
↪format)
    ├── keypoints         # dtype=np.float, shape=(1000,14,3)
    ├── object_fields     # dtype=np.uint8, shape=(2,16)     (note: string in ASCII␣
↪format)
    └── object_ids        # dtype=np.int32, shape=(1000,2)
```

Fields

- **image_filenames: image file path+name**
  - available in: train, test
  - dtype: np.uint8
  - is padded: True
  - fill value: 0
  - note: strings stored in ASCII format
- **keypoint_names: body joint names**
  - available in: train, test
  - dtype: np.uint8
  - is padded: True
  - fill value: 0
  - note: strings stored in ASCII format
- **keypoints: keypoint coordinates**
  - available in: train, test
  - dtype: np.float
  - is padded: False
  - fill value: -1
  - note: keypoint format [x1,y1,is_visible]
- **object_fields: list of field names of the object id list**
  - available in: train, test
  - dtype: np.uint8
  - is padded: True
  - fill value: 0
  - note: strings stored in ASCII format
  - note: key field (*field name* aggregator)
- **object_ids: list of field ids**
  - available in: train, test
  - dtype: np.int32
  - is padded: False
  - fill value: -1
  - note: key field (*field id* aggregator)

Disclaimer

All rights reserved to the original creators of **Leeds Sports Pose**.

For information about the dataset and its terms of use, please see this link.

LSPe - Leeds Sports Pose Extended

The **Leeds Sports Pose extended** dataset contains 10,000 images gathered from Flickr searches for the tags 'parkour', 'gymnastics', and 'athletics' and consists of poses deemed to be challenging to estimate. Each image has a corresponding annotation gathered from Amazon Mechanical Turk and as such cannot be guaranteed to be highly accurate. The images have been scaled such that the annotated person is roughly 150 pixels in length. Each image has been annotated with up to 14 visible joint locations.

Use cases

Human body joint detection.

Properties

- `name`: leeds_sports_pose_extended
- `keywords`: image_processing, detection, human_pose, keypoints
- `dataset size`: 206,2 MB
- `is downloadable`: **yes**
- **`tasks:`**
  - **keypoints: (default)**
    - `primary use`: human body joint detection
    - `description`: Contains image files and body parts keypoint coordinates for detecting human body joints in images
    - `sets`: train, test
    - `metadata file size in disk`: 473,7 kB
    - **`has annotations:` yes**
      - **`which:`**

        body joint keypoints

Note: This dataset is essentially the same as the `leeds_sports_pose` but contains more training samples.

Metadata structure (HDF5)

Task: keypoints

```
/
├── train/
│   ├── image_filenames   # dtype=np.uint8, shape=(11000,104)  (note: string in ASCII␣
↪format)
│   ├── keypoint_names    # dtype=np.uint8, shape=(14,15)    (note: string in ASCII␣
↪format)
│   ├── keypoints         # dtype=np.float, shape=(11000,14,3)
│   ├── object_fields     # dtype=np.uint8, shape=(2,16)     (note: string in ASCII␣
↪format)
│   └── object_ids        # dtype=np.int32, shape=(11000,2)
│
└── test/
    ├── image_filenames   # dtype=np.uint8, shape=(1000,104)  (note: string in ASCII␣
↪format)
    ├── keypoint_names    # dtype=np.uint8, shape=(14,15)    (note: string in ASCII␣
↪format)
    ├── keypoints         # dtype=np.float, shape=(1000,14,3)
    ├── object_fields     # dtype=np.uint8, shape=(2,16)     (note: string in ASCII␣
↪format)
    └── object_ids        # dtype=np.int32, shape=(1000,2)
```

Fields

- **`image_filenames`: image file path+name**
  - `available in`: train, test
  - `dtype`: np.uint8
  - `is padded`: True
  - `fill value`: 0
  - `note`: strings stored in ASCII format
- **`keypoint_names`: body joint names**
  - `available in`: train, test
  - `dtype`: np.uint8
  - `is padded`: True
  - `fill value`: 0
  - `note`: strings stored in ASCII format
- **`keypoints`: keypoint coordinates**
  - `available in`: train, test
  - `dtype`: np.float
  - `is padded`: False
  - `fill value`: -1
  - `note`: keypoint format [x1,y1,is_visible]

- **object_fields: list of field names of the object id list**
  - available in: train, test
  - dtype: np.uint8
  - is padded: True
  - fill value: 0
  - note: strings stored in ASCII format
  - note: key field (*field name* aggregator)
- **object_ids: list of field ids**
  - available in: train, test
  - dtype: np.int32
  - is padded: False
  - fill value: -1
  - note: key field (*field id* aggregator)

Disclaimer

All rights reserved to the original creators of **Leeds Sports Pose extended**.

For information about the dataset and its terms of use, please see this link.

MNIST Handwritten Digit Database

The **MNIST** database of handwritten digits has a training set of 60,000 examples, and a test set of 10,000 examples. It is a subset of a larger set available from **MNIST**. The digits have been size-normalized and centered in a fixed-size image.

Use cases

Image classification.

Properties

- name: mnist
- keywords: image_processing, classification
- dataset size: 11,6 MB
- is downloadable: **yes**
- tasks: *classification (default)*

Tasks

classification (default)

How to use

```
>>> # import the package
>>> import dbcollection as dbc
>>>
>>> # load the dataset
>>> mnist = dbc.load('mnist', 'classification')
>>> mnist
DataLoader: "mnist" (classification task)
```

Properties

- `primary use`: image classification
- `description`: Contains image tensors and label annotations for image classification.
- `sets`: train, test
- `metadata file size in disk`: 6,8 MB
- **`has annotations`: yes**
  - **`which`:**
    - labels for each image class/category.
- **`available fields`:**
  - *classes*
  - *images*
  - *labels*
  - *object_fields*
  - *object_ids*
  - *list_images_per_class*

HDF5 file structure

```
/
├── train/
│   ├── classes        # dtype=np.uint8, shape=(10,2)   (note: string in ASCII format)
│   ├── images         # dtype=np.uint8, shape=(60000,28,28)
│   ├── labels         # dtype=np.uint8, shape=(60000,)
│   ├── object_fields  # dtype=np.uint8, shape=(2,7)    (note: string in ASCII format)
│   ├── object_ids     # dtype=np.int32, shape=(60000,2)
│   └── list_images_per_class   # dtype=np.int32, shape=(10,6742))
│
└── test/
    ├── classes        # dtype=np.uint8, shape=(10,2)  (note: string in ASCII format)
    ├── images         # dtype=np.uint8, shape=(10000,28,28)
    ├── labels         # dtype=np.uint8, shape=(10000,)
    ├── object_fields  # dtype=np.uint8, shape=(2,7)    (note: string in ASCII format)
    ├── object_ids     # dtype=np.int32, shape=(10000,2)
    └── list_images_per_class   # dtype=np.int32, shape=(10,1135))
```

Fields

- **`classes`: class names**
  - `available in`: train, test
  - `dtype`: np.uint8
  - `is padded`: True
  - `fill value`: 0
  - `note`: strings stored in ASCII format
- **`images`: images tensor**
  - `available in`: train, test
  - `dtype`: np.uint8
  - `is padded`: False
  - `fill value`: -1
- **`labels`: class ids**
  - `available in`: train, test
  - `dtype`: np.uint8
  - `is padded`: False
  - `fill value`: -1
- **`object_fields`: list of field names of the object id list**
  - `available in`: train, test
  - `dtype`: np.uint8
  - `is padded`: True
  - `fill value`: 0
  - `note`: strings stored in ASCII format
  - `note`: key field (*field name* aggregator)

- **object_ids: list of field ids**
  - available in: train, test
  - dtype: np.int32
  - is padded: False
  - fill value: -1
  - note: key field (*field id* aggregator)
- **list_images_per_class: list of image ids per class**
  - available in: train, test
  - dtype: np.int32
  - is padded: True
  - fill value: -1
  - note: pre-ordered list

Disclaimer

MPII Human Pose

MPII Human Pose dataset is a state of the art benchmark for evaluation of articulated human pose estimation. The dataset includes around 25K images containing over 40K people with annotated body joints. The images were systematically collected using an established taxonomy of every day human activities. Overall the dataset covers 410 human activities and each image is provided with an activity label.

Use cases

Human body joint detection.

Properties

- name: mpii_pose
- keywords: image_processing, detection, human_pose, keypoints
- dataset size: 12,1 GB
- is downloadable: **yes**
- **tasks:**
  - keypoints (default)
  - keypoints_clean (TODO)

Tasks

keypoints (default)

How to use

```
>>> # import the package
>>> import dbcollection as dbc
>>>
>>> # load the dataset
>>> mnist = dbc.load('mpii_pose', 'keypoints')
>>> mnist
DataLoader: "mpii_pose" (keypoints task)
```

Properties

- `primary use`: body joint prediction / classification
- `description`: Contains single human body pose annotations for body joint prediction / classification.
- `sets`: train, train01, val01, test
- `metadata file size in disk`: 7 MB
- **`has annotations`: yes**
    - **which:**
        - labels for each activity.
        - frame position of an image from the original video
        - head bounding box coordinates
        - body joint coordinates (x, y) and visibility
        - center coordinates (x, y) of a single person detection
        - scale of the person detection w.r.t. 200px height detections
        - is detection of sufficiently separated individuals?
        - acivities
        - category names
        - keypoint labels
        - video names / ids
- **`available fields`:**
    - *activity_id*

- *activity_name*
- *category_name*
- *frame_sec*
- *head_bbox*
- *image_filenames*
- *keypoint_labels*
- *keypoints*
- *object_fields*
- *object_ids*
- *objpos*
- *scale*
- *video_id*
- *video_name*
- *list_keypoints_per_image*
- *list_single_person_per_image*

HDF5 file structure

```
/
├── train/
│   ├── activity_id       # dtype=np.int32, shape=(29116,),
│   ├── activity_name     # dtype=np.uint8, shape=(29116,101)  (note: string in ASCII␣
↪format)
│   ├── category_name     # dtype=np.uint8, shape=(29116,23)   (note: string in ASCII␣
↪format)
│   ├── frame_sec         # dtype=np.int32, shape=(29116,)
│   ├── head_bbox         # dtype=np.float, shape=(29116,4)
│   ├── image_filenames   # dtype=np.uint8, shape=(29116,21)   (note: string in ASCII␣
↪format)
│   ├── keypoint_labels   # dtype=np.uint8, shape=(16,15)      (note: string in ASCII␣
↪format)
│   ├── keypoints         # dtype=np.float, shape=(29116,16,3)
│   ├── object_fields     # dtype=np.uint8, shape=(13,16)       (note: string in␣
↪ASCII format)
│   ├── object_ids        # dtype=np.int32, shape=(29116,13)
│   ├── objpos            # dtype=np.float, shape=(29116,2)
│   ├── scales            # dtype=np.float, shape=(29116,)
│   ├── video_ids         # dtype=np.int32, shape=(29116,)
│   ├── video_names       # dtype=np.uint8, shape=(29116,12)    (note: string in␣
↪ASCII format)
│   ├── list_keypoints_per_image       # dtype=np.int32, shape=(18079,17)
│   └── list_single_person_per_image   # dtype=np.int32, shape=(18079,1))
│
├── train01/
│   ├── activity_id       # dtype=np.int32, shape=(20310,),
│   ├── activity_name     # dtype=np.uint8, shape=(20310,101)  (note: string in ASCII␣
↪format)
```

(continues on next page)

(continued from previous page)

```
│   ├── category_name      # dtype=np.uint8, shape=(20310,23)    (note: string in ASCII␣
↪format)
│   ├── frame_sec          # dtype=np.int32, shape=(20310,)
│   ├── head_bbox          # dtype=np.float, shape=(20310,4)
│   ├── image_filenames    # dtype=np.uint8, shape=(20310,21)    (note: string in ASCII␣
↪format)
│   ├── keypoint_labels    # dtype=np.uint8, shape=(16,15)       (note: string in ASCII␣
↪format)
│   ├── keypoints          # dtype=np.float, shape=(20310,16,3)
│   ├── object_fields      # dtype=np.uint8, shape=(13,16)        (note: string in␣
↪ASCII format)
│   ├── object_ids         # dtype=np.int32, shape=(20310,13)
│   ├── objpos             # dtype=np.float, shape=(20310,2)
│   ├── scales             # dtype=np.float, shape=(20310,)
│   ├── video_ids          # dtype=np.int32, shape=(20310,)
│   ├── video_names        # dtype=np.uint8, shape=(20310,12)     (note: string in␣
↪ASCII format)
│   ├── list_keypoints_per_image       # dtype=np.int32, shape=(12656,17)
│   └── list_single_person_per_image   # dtype=np.int32, shape=(12656,1))
│
├── val01/
│   ├── activity_id        # dtype=np.int32, shape=(8806,),
│   ├── activity_name      # dtype=np.uint8, shape=(8806,101)  (note: string in ASCII␣
↪format)
│   ├── category_name      # dtype=np.uint8, shape=(8806,23)    (note: string in ASCII␣
↪format)
│   ├── frame_sec          # dtype=np.int32, shape=(8806,)
│   ├── head_bbox          # dtype=np.float, shape=(8806,4)
│   ├── image_filenames    # dtype=np.uint8, shape=(8806,21)    (note: string in ASCII␣
↪format)
│   ├── keypoint_labels    # dtype=np.uint8, shape=(16,15)       (note: string in ASCII␣
↪format)
│   ├── keypoints          # dtype=np.float, shape=(8806,16,3)
│   ├── object_fields      # dtype=np.uint8, shape=(13,16)        (note: string in␣
↪ASCII format)
│   ├── object_ids         # dtype=np.int32, shape=(8806,13)
│   ├── objpos             # dtype=np.float, shape=(8806,2)
│   ├── scales             # dtype=np.float, shape=(8806,)
│   ├── video_ids          # dtype=np.int32, shape=(8806,)
│   ├── video_names        # dtype=np.uint8, shape=(8806,12)     (note: string in ASCII␣
↪format)
│   ├── list_keypoints_per_image       # dtype=np.int32, shape=(5423,17)
│   └── list_single_person_per_image   # dtype=np.int32, shape=(5423,7))
│
└── test/
    ├── activity_id        # dtype=np.int32, shape=(11776,),
    ├── activity_name      # dtype=np.uint8, shape=(11776,101)  (note: string in ASCII␣
↪format)
    ├── category_name      # dtype=np.uint8, shape=(11776,23)    (note: string in ASCII␣
↪format)
    ├── frame_sec          # dtype=np.int32, shape=(11776,)
    ├── image_filenames    # dtype=np.uint8, shape=(11776,21)    (note: string in ASCII␣
↪format)
    ├── keypoint_labels    # dtype=np.uint8, shape=(16,15)       (note: string in ASCII␣
↪format)
    ├── object_fields      # dtype=np.uint8, shape=(13,16)        (note: string in␣
↪ASCII format)
```

(continues on next page)

(continued from previous page)

```
    ├── object_ids         # dtype=np.int32, shape=(11776,13)
    ├── objpos             # dtype=np.float, shape=(11776,2)
    ├── scales             # dtype=np.float, shape=(11776,)
    ├── video_ids          # dtype=np.int32, shape=(11776,)
    ├── video_names        # dtype=np.uint8, shape=(11776,12)    (note: string in␣
↪ASCII format)
    └── list_single_person_per_image   # dtype=np.int32, shape=(6908,7))
```

Fields

- **activity_id: activity ids**
  - available in: train, train01, val01, test
  - dtype: np.int32
  - is padded: False
  - fill value: -1
- **activity_name: activity names**
  - available in: train, train01, val01, test
  - dtype: np.uint8
  - is padded: True
  - fill value: 0
  - note: strings stored in ASCII format
- **category_name: category names**
  - available in: train, train01, val01, test
  - dtype: np.uint8
  - is padded: True
  - fill value: 0
  - note: strings stored in ASCII format
- **frame_sec: image position in video, in seconds**
  - available in: train, train01, val01, test
  - dtype: np.int32
  - is padded: False
  - fill value: -1
- **head_bbox: head bounding box coordinates**
  - available in: train, train01, val01
  - dtype: np.float
  - is padded: False
  - fill value: -1
  - note: bbox format [x1,y1,x2,y2]

- **image_filenames: image file name + path**
  - available in: train, train01, val01, test
  - dtype: np.uint8
  - is padded: True
  - fill value: 0
  - note: strings stored in ASCII format
- **keypoint_labels: body joint names**
  - available in: train, train01, val01, test
  - dtype: np.uint8
  - is padded: True
  - fill value: 0
  - note: strings stored in ASCII format
- **keypoints: body joint coordinates (x, y)**
  - available in: train, train01, val01
  - dtype: np.float
  - is padded: False
  - fill value: -1
  - note: keypoint format [x1, y1, is_visible]
- **object_fields: list of field names of the object id list**
  - available in: train, train01, val01, test
  - dtype: np.uint8
  - is padded: True
  - fill value: 0
  - note: strings stored in ASCII format
  - note: key field (*field name* aggregator)
- **object_ids: list of field ids**
  - available in: train, train01, val01, test
  - dtype: np.int32
  - is padded: False
  - fill value: -1
  - note: key field (*field id* aggregator)
- **objpos: person / detection center coordinates**
  - available in: train, train01, val01, test
  - dtype: np.float
  - is padded: False
  - fill value: -1

    - `note`: position format [x, y]

- **`scale`: person scale w.r.t. 200px height**
    - `available in`: train, train01, val01, test
    - `dtype`: np.float
    - `is padded`: False
    - `fill value`: -1
- **`video_id`: video index**
    - `available in`: train, train01, val01, test
    - `dtype`: np.int32
    - `is padded`: True
    - `fill value`: -1
- **`video_name`: video name**
    - `available in`: train, train01, val01, test
    - `dtype`: np.uint8
    - `is padded`: True
    - `fill value`: 0
    - `note`: strings stored in ASCII format
- **`list_keypoints_per_image`: list of available body joints ids per image**
    - `available in`: train, train01, val01
    - `dtype`: np.int32
    - `is padded`: True
    - `fill value`: -1
    - `note`: pre-ordered list
- **`list_single_person_per_image`: list of single person detection ids per image**
    - `available in`: train, train01, val01, test
    - `dtype`: np.int32
    - `is padded`: True
    - `fill value`: -1
    - `note`: pre-ordered list

PASCAL VOC2007 - The PASCAL Visual Object Classes Challenge 2007

The **PASCAL Visual Object Classes Challenge 2007** goal is to recognize objects from a number of visual object classes in realistic scenes (i.e. not pre-segmented objects). There are two main tasks (classification and detection) and two additional competitions (segmentation and person layout).

Note: For now, only the detection task is implemented. Submittions of the remaining tasks are highly appreciated if you are considering contributing to this project.

Use cases

Image classification, detection, segmentation and person pose detection.

Properties

- `name`: pascal_voc_2007
- `keywords`: image_processing, object_detection
- `dataset size`: 911,1 MB
- `is downloadable`: **yes**
- **`tasks:`**
  - **detection: (default)**
    - `primary use`: image classification
    - `description`: Contains image filenames, label and bounding box coordinates annotations for object detection.
    - `sets`: train, val, trainval, test
    - `metadata file size in disk`: 1,4 MB
    - **`has annotations:` yes**
      - **`which:`**

        labels for each image class/category.

        bounding box coordinates

Metadata structure (HDF5)

Task: detection

```
/
├── train/
│   ├── boxes              # dtype=np.float, shape=(7844,4)
│   ├── category_id        # dtype=np.int32, shape=(20,)
│   ├── classes            # dtype=np.uint8, shape=(20,12)     (note: string in ASCII␣
↪format)
│   ├── difficult          # dtype=np.int32, shape=(2,)
│   ├── id                 # dtype=np.int32, shape=(7844,)
│   ├── image_filenames    # dtype=np.uint8, shape=(2501,83)   (note: string in ASCII␣
↪format)
│   ├── image_id           # dtype=np.int32, shape=(2501,)
│   ├── sizes              # dtype=np.int32, shape=(2501,3)
│   ├── truncated          # dtype=np.int32, shape=(2,)
│   ├── object_fields      # dtype=np.uint8, shape=(6,16)      (note: string in ASCII␣
↪format)
│   ├── object_ids         # dtype=np.int32, shape=(7844,6)
│   ├── list_boxes_per_image             # dtype=np.int32, shape=(2501,37)
│   ├── list_image_filenames_per_class   # dtype=np.int32, shape=(20,1070)
│   ├── list_object_ids_per_image        # dtype=np.int32, shape=(2501,37)
```

(continues on next page)

(continued from previous page)

```
│   ├── list_object_ids_difficult        # dtype=np.int32, shape=(1543,)
│   ├── list_object_ids_no_difficult     # dtype=np.int32, shape=(6301,)
│   ├── list_object_ids_truncated        # dtype=np.int32, shape=(4108,)
│   ├── list_object_ids_no_truncated     # dtype=np.int32, shape=(3736,)
│   └── list_object_ids_per_class        # dtype=np.int32, shape=(20,2705)
│
├── val/
│   ├── boxes              # dtype=np.float, shape=(7818,4)
│   ├── category_id        # dtype=np.int32, shape=(20,)
│   ├── classes            # dtype=np.uint8, shape=(20,12)     (note: string in ASCII␣
↪format)
│   ├── difficult          # dtype=np.int32, shape=(2,)
│   ├── id                 # dtype=np.int32, shape=(7818,)
│   ├── image_filenames    # dtype=np.uint8, shape=(2510,83)   (note: string in ASCII␣
↪format)
│   ├── image_id           # dtype=np.int32, shape=(2510,)
│   ├── sizes              # dtype=np.int32, shape=(2510,3)
│   ├── truncated          # dtype=np.int32, shape=(2,)
│   ├── object_fields      # dtype=np.uint8, shape=(6,16)      (note: string in ASCII␣
↪format)
│   ├── object_ids         # dtype=np.int32, shape=(7818,6)
│   ├── list_boxes_per_image             # dtype=np.int32, shape=(2510,42)
│   ├── list_image_filenames_per_class   # dtype=np.int32, shape=(20,1025)
│   ├── list_object_ids_per_image        # dtype=np.int32, shape=(2510,42)
│   ├── list_object_ids_difficult        # dtype=np.int32, shape=(1511,)
│   ├── list_object_ids_no_difficult     # dtype=np.int32, shape=(6307,)
│   ├── list_object_ids_truncated        # dtype=np.int32, shape=(4036,)
│   ├── list_object_ids_no_truncated     # dtype=np.int32, shape=(3782,)
│   └── list_object_ids_per_class        # dtype=np.int32, shape=(20,2742)
│
├── trainval/
│   ├── boxes              # dtype=np.float, shape=(15662,4)
│   ├── category_id        # dtype=np.int32, shape=(20,)
│   ├── classes            # dtype=np.uint8, shape=(20,12)     (note: string in ASCII␣
↪format)
│   ├── difficult          # dtype=np.int32, shape=(2,)
│   ├── id                 # dtype=np.int32, shape=(15662,)
│   ├── image_filenames    # dtype=np.uint8, shape=(5011,83)   (note: string in ASCII␣
↪format)
│   ├── image_id           # dtype=np.int32, shape=(5011,)
│   ├── sizes              # dtype=np.int32, shape=(5011,3)
│   ├── truncated          # dtype=np.int32, shape=(2,)
│   ├── object_fields      # dtype=np.uint8, shape=(6,16)      (note: string in ASCII␣
↪format)
│   ├── object_ids         # dtype=np.int32, shape=(15662,6)
│   ├── list_boxes_per_image             # dtype=np.int32, shape=(5011,42)
│   ├── list_image_filenames_per_class   # dtype=np.int32, shape=(20,2095)
│   ├── list_object_ids_per_image        # dtype=np.int32, shape=(5011,42)
│   ├── list_object_ids_difficult        # dtype=np.int32, shape=(3054,)
│   ├── list_object_ids_no_difficult     # dtype=np.int32, shape=(12608,)
│   ├── list_object_ids_truncated        # dtype=np.int32, shape=(8144,)
│   ├── list_object_ids_no_truncated     # dtype=np.int32, shape=(7518,)
│   └── list_object_ids_per_class        # dtype=np.int32, shape=(20,5447)
│
└── test/
    ├── boxes              # dtype=np.float, shape=(14976,4)
    ├── category_id        # dtype=np.int32, shape=(20,)
```

(continues on next page)

(continued from previous page)

```
    ├── classes           # dtype=np.uint8, shape=(20,12)     (note: string in ASCII␣
↪format)
    ├── difficult         # dtype=np.int32, shape=(2,)
    ├── id                # dtype=np.int32, shape=(14976,)
    ├── image_filenames   # dtype=np.uint8, shape=(4952,83)   (note: string in ASCII␣
↪format)
    ├── image_id          # dtype=np.int32, shape=(4952,)
    ├── sizes             # dtype=np.int32, shape=(4952,3)
    ├── truncated         # dtype=np.int32, shape=(2,)
    ├── object_fields     # dtype=np.uint8, shape=(6,16)      (note: string in ASCII␣
↪format)
    ├── object_ids        # dtype=np.int32, shape=(14976,6)
    ├── list_boxes_per_image             # dtype=np.int32, shape=(4952,41)
    ├── list_image_filenames_per_class   # dtype=np.int32, shape=(20,2097)
    ├── list_object_ids_per_image        # dtype=np.int32, shape=(4952,41)
    ├── list_object_ids_difficult        # dtype=np.int32, shape=(2944,)
    ├── list_object_ids_no_difficult     # dtype=np.int32, shape=(12032,)
    ├── list_object_ids_truncated        # dtype=np.int32, shape=(1824,)
    ├── list_object_ids_no_truncated     # dtype=np.int32, shape=(7152,)
    └── list_object_ids_per_class        # dtype=np.int32, shape=(20,5227)
```

Fields

- **boxes: bounding box coordinates**
  - available in: train, val, trainval, test
  - dtype: np.float
  - is padded: False
  - fill value: -1
  - note: bbox format [x1,y1,x2,y2]
- **category_id: category id**
  - available in: train, val, trainval, test
  - dtype: np.int32
  - is padded: False
  - fill value: -1
- **classes: class names**
  - available in: train, val, trainval, test
  - dtype: np.uint8
  - is padded: True
  - fill value: 0
  - note: strings stored in ASCII format
- **difficult: is difficult**
  - available in: train, val, trainval, test
  - dtype: np.int32

  - `is padded`: False
  - `fill value`: -1

- **`id`: object id**
  - `available in`: train, val, trainval, test
  - `dtype`: np.int32
  - `is padded`: False
  - `fill value`: -1

- **`image_filenames`: image file path+name**
  - `available in`: train, val, trainval, test
  - `dtype`: np.uint8
  - `is padded`: True
  - `fill value`: 0
  - `note`: strings stored in ASCII format

- **`image_id`: image id**
  - `available in`: train, val, trainval, test
  - `dtype`: np.int32
  - `is padded`: False
  - `fill value`: -1

- **`sizes`: image size**
  - `available in`: train, val, trainval, test
  - `dtype`: np.int32
  - `is padded`: False
  - `fill value`: -1
  - `note`: size format [width, height, depth]

- **`truncated`: is truncated**
  - `available in`: train, val, trainval, test
  - `dtype`: np.int32
  - `is padded`: False
  - `fill value`: -1

- **`object_fields`: list of field names of the object id list**
  - `available in`: train, val, trainval, test
  - `dtype`: np.uint8
  - `is padded`: True
  - `fill value`: 0
  - `note`: strings stored in ASCII format
  - `note`: key field (*field name* aggregator)

- **object_ids: list of field ids**
  - available in: train, val, trainval, test
  - dtype: np.int32
  - is padded: False
  - fill value: -1
  - note: key field (*field id* aggregator)
- **list_boxes_per_image: list of bounding boxes per image**
  - available in: train, val, trainval, test
  - dtype: np.int32
  - is padded: True
  - fill value: -1
  - note: pre-ordered list
- **list_image_filenames_per_class: list of image filenames ids per class**
  - available in: train, val, trainval, test
  - dtype: np.int32
  - is padded: True
  - fill value: -1
  - note: pre-ordered list
- **list_object_ids_per_image: list of object ids per image**
  - available in: train, val, trainval, test
  - dtype: np.int32
  - is padded: True
  - fill value: -1
  - note: pre-ordered list
- **list_object_ids_difficult: list of object ids for difficult objects**
  - available in: train, val, trainval, test
  - dtype: np.int32
  - is padded: True
  - fill value: -1
  - note: pre-ordered list
- **list_object_ids_no_difficult: list of object ids for not difficult objects**
  - available in: train, val, trainval, test
  - dtype: np.int32
  - is padded: True
  - fill value: -1
  - note: pre-ordered list

- **list_object_ids_truncated: list of object ids for truncated objects**
  - available in: train, val, trainval, test
  - dtype: np.int32
  - is padded: True
  - fill value: -1
  - note: pre-ordered list
- **list_object_ids_no_truncated: list of object ids for not truncated objects**
  - available in: train, val, trainval, test
  - dtype: np.int32
  - is padded: True
  - fill value: -1
  - note: pre-ordered list
- **list_object_ids_per_class: list of object ids per class**
  - available in: train, val, trainval, test
  - dtype: np.int32
  - is padded: True
  - fill value: -1
  - note: pre-ordered list

Disclaimer

All rights reserved to the original creators of **PASCAL VOC2007**.

For information about the dataset and its terms of use, please see this link.

PASCAL VOC2012 - The PASCAL Visual Object Classes Challenge 2012

The **PASCAL Visual Object Classes Challenge 2012** goal is to recognize objects from a number of visual object classes in realistic scenes (i.e. not pre-segmented objects). There are two main tasks (classification and detection) and two additional competitions (segmentation and action classification).

Note: For now, only the detection task is implemented. Submittions of the remaining tasks are highly appreciated if you are considering contributing to this project.

Use cases

Image classification, detection, segmentation and action classification.

Properties

- `name`: pascal_voc_2012
- `keywords`: image_processing, object_detection
- `dataset size`: 2,0 GB + 1,9 GB
- **`is downloadable`: partially**
  - `data setup`: manually download and unpack the test set into a folder or symlink dir named `pascal_voc_2012` before loading the dataset.
- **`tasks`:**
  - **detection: (default)**
    - `primary use`: image classification
    - `description`: Contains image filenames, label and bounding box coordinates annotations for object detection.
    - `sets`: train, val, trainval, test
    - `metadata file size in disk`: 1,9 MB
    - **`has annotations`: yes**
      - **`which`:**

        labels for each image class/category.

        bounding box coordinates

Note: For the PASCAL VOC2012 dataset, contrary to VOC2007, the test set is only available for registered users in the dataset's original website. You can manually download it and add it along with the trainval data which is publicly available for download. The test set is **optional** if you cannot/do not want to download it from the source. Everything will work just fine, with the exception that you won't be able to access the test data. However, if you want to use it, this is the recommended way to do so:

1. download the dataset using `dbc.download('pascal_voc_2012', data_dir='some/path/dir/')`
2. inside the created dir (it will have the same name as the dataset), put the test data file (zip) and unpack it.
3. when loading the dataset's metadata with the `load()` method, during setup it will automatically detect if the test set is available or not.

Metadata structure (HDF5)

Task: detection

```
/
├── train/
│   ├── boxes             # dtype=np.float, shape=(15774,4)
│   ├── category_id       # dtype=np.int32, shape=(20,)
│   ├── classes           # dtype=np.uint8, shape=(20,12)     (note: string in ASCII␣
↪format)
│   ├── difficult         # dtype=np.int32, shape=(2,)
```

(continues on next page)

(continued from previous page)

```
│   ├── id                # dtype=np.int32, shape=(15774,)
│   ├── image_filenames   # dtype=np.uint8, shape=(5717,88)   (note: string in ASCII␣
↪format)
│   ├── image_id          # dtype=np.int32, shape=(5717,)
│   ├── sizes             # dtype=np.int32, shape=(5717,3)
│   ├── truncated         # dtype=np.int32, shape=(2,)
│   ├── object_fields     # dtype=np.uint8, shape=(6,16)      (note: string in ASCII␣
↪format)
│   ├── object_ids        # dtype=np.int32, shape=(15774,6)
│   ├── list_boxes_per_image             # dtype=np.int32, shape=(5717,56)
│   ├── list_image_filenames_per_class   # dtype=np.int32, shape=(20,2142)
│   ├── list_object_ids_per_image        # dtype=np.int32, shape=(5717,56)
│   ├── list_object_ids_difficult        # dtype=np.int32, shape=(2165,)
│   ├── list_object_ids_no_difficult     # dtype=np.int32, shape=(13609,)
│   ├── list_object_ids_truncated        # dtype=np.int32, shape=(8102,)
│   ├── list_object_ids_no_truncated     # dtype=np.int32, shape=(7672,)
│   └── list_object_ids_per_class        # dtype=np.int32, shape=(20,5019)
│
├── val/
│   ├── boxes             # dtype=np.float, shape=(15787,4)
│   ├── category_id       # dtype=np.int32, shape=(20,)
│   ├── classes           # dtype=np.uint8, shape=(20,12)     (note: string in ASCII␣
↪format)
│   ├── difficult         # dtype=np.int32, shape=(2,)
│   ├── id                # dtype=np.int32, shape=(15787,)
│   ├── image_filenames   # dtype=np.uint8, shape=(5823,88)   (note: string in ASCII␣
↪format)
│   ├── image_id          # dtype=np.int32, shape=(5823,)
│   ├── sizes             # dtype=np.int32, shape=(5823,3)
│   ├── truncated         # dtype=np.int32, shape=(2,)
│   ├── object_fields     # dtype=np.uint8, shape=(6,16)      (note: string in ASCII␣
↪format)
│   ├── object_ids        # dtype=np.int32, shape=(15787,6)
│   ├── list_boxes_per_image             # dtype=np.int32, shape=(5823,42)
│   ├── list_image_filenames_per_class   # dtype=np.int32, shape=(20,2232)
│   ├── list_object_ids_per_image        # dtype=np.int32, shape=(5823,42)
│   ├── list_object_ids_difficult        # dtype=np.int32, shape=(1946,)
│   ├── list_object_ids_no_difficult     # dtype=np.int32, shape=(13841,)
│   ├── list_object_ids_truncated        # dtype=np.int32, shape=(8288,)
│   ├── list_object_ids_no_truncated     # dtype=np.int32, shape=(7499,)
│   └── list_object_ids_per_class        # dtype=np.int32, shape=(20,5110)
│
├── trainval/
│   ├── boxes             # dtype=np.float, shape=(31561,4)
│   ├── category_id       # dtype=np.int32, shape=(20,)
│   ├── classes           # dtype=np.uint8, shape=(20,12)     (note: string in ASCII␣
↪format)
│   ├── difficult         # dtype=np.int32, shape=(2,)
│   ├── id                # dtype=np.int32, shape=(34561,)
│   ├── image_filenames   # dtype=np.uint8, shape=(11540,88)   (note: string in ASCII␣
↪format)
│   ├── image_id          # dtype=np.int32, shape=(11540,)
│   ├── sizes             # dtype=np.int32, shape=(11540,3)
│   ├── truncated         # dtype=np.int32, shape=(2,)
│   ├── object_fields     # dtype=np.uint8, shape=(6,16)      (note: string in ASCII␣
↪format)
│   ├── object_ids        # dtype=np.int32, shape=(31561,6)
```

(continues on next page)

(continued from previous page)

```
|    ├── list_boxes_per_image             # dtype=np.int32, shape=(11540,56)
|    ├── list_image_filenames_per_class   # dtype=np.int32, shape=(20,4374)
|    ├── list_object_ids_per_image        # dtype=np.int32, shape=(11540,56)
|    ├── list_object_ids_difficult        # dtype=np.int32, shape=(4111,)
|    ├── list_object_ids_no_difficult     # dtype=np.int32, shape=(27450,)
|    ├── list_object_ids_truncated        # dtype=np.int32, shape=(16390,)
|    ├── list_object_ids_no_truncated     # dtype=np.int32, shape=(15171,)
|    └── list_object_ids_per_class        # dtype=np.int32, shape=(20,10129)
|
└── test/
    ├── id                # dtype=np.int32, shape=(10991,)
    ├── image_filenames   # dtype=np.uint8, shape=(10991,88)   (note: string in ASCII␣
↪format)
    ├── object_fields     # dtype=np.uint8, shape=(16,)        (note: string in ASCII␣
↪format)
    └── object_ids        # dtype=np.int32, shape=(10991,1)
```

Fields

- **`boxes`: bounding box coordinates**
  - `available in`: train, val, trainval
  - `dtype`: np.float
  - `is padded`: False
  - `fill value`: -1
  - `note`: bbox format [x1,y1,x2,y2]
- **`category_id`: category id**
  - `available in`: train, val, trainval
  - `dtype`: np.int32
  - `is padded`: False
  - `fill value`: -1
- **`classes`: class names**
  - `available in`: train, val, trainval
  - `dtype`: np.uint8
  - `is padded`: True
  - `fill value`: 0
  - `note`: strings stored in ASCII format
- **`difficult`: is difficult**
  - `available in`: train, val, trainval
  - `dtype`: np.int32
  - `is padded`: False
  - `fill value`: -1

- **id: object id**
  - available in: train, val, trainval, test
  - dtype: np.int32
  - is padded: False
  - fill value: -1
- **image_filenames: image file path+name**
  - available in: train, val, trainval, test
  - dtype: np.uint8
  - is padded: True
  - fill value: 0
  - note: strings stored in ASCII format
- **image_id: image id**
  - available in: train, val, trainval
  - dtype: np.int32
  - is padded: False
  - fill value: -1
- **sizes: image size**
  - available in: train, val, trainval
  - dtype: np.int32
  - is padded: False
  - fill value: -1
  - note: size format [width, height, depth]
- **truncated: is truncated**
  - available in: train, val, trainval
  - dtype: np.int32
  - is padded: False
  - fill value: -1
- **object_fields: list of field names of the object id list**
  - available in: train, val, trainval, test
  - dtype: np.uint8
  - is padded: True
  - fill value: 0
  - note: strings stored in ASCII format
  - note: key field (*field name* aggregator)
- **object_ids: list of field ids**
  - available in: train, val, trainval, test

  - `dtype`: np.int32
  - `is padded`: False
  - `fill value`: -1
  - `note`: key field (*field id* aggregator)
- **`list_boxes_per_image`: list of bounding boxes per image**
  - `available in`: train, val, trainval
  - `dtype`: np.int32
  - `is padded`: True
  - `fill value`: -1
  - `note`: pre-ordered list
- **`list_image_filenames_per_class`: list of image filenames ids per class**
  - `available in`: train, val, trainval
  - `dtype`: np.int32
  - `is padded`: True
  - `fill value`: -1
  - `note`: pre-ordered list
- **`list_object_ids_per_image`: list of object ids per image**
  - `available in`: train, val, trainval
  - `dtype`: np.int32
  - `is padded`: True
  - `fill value`: -1
  - `note`: pre-ordered list
- **`list_object_ids_difficult`: list of object ids for difficult objects**
  - `available in`: train, val, trainval
  - `dtype`: np.int32
  - `is padded`: True
  - `fill value`: -1
  - `note`: pre-ordered list
- **`list_object_ids_no_difficult`: list of object ids for not difficult objects**
  - `available in`: train, val, trainval
  - `dtype`: np.int32
  - `is padded`: True
  - `fill value`: -1
  - `note`: pre-ordered list
- **`list_object_ids_truncated`: list of object ids for truncated objects**
  - `available in`: train, val, trainval

        - `dtype`: np.int32
        - `is padded`: True
        - `fill value`: -1
        - `note`: pre-ordered list
- **`list_object_ids_no_truncated`: list of object ids for not truncated objects**
        - `available in`: train, val, trainval
        - `dtype`: np.int32
        - `is padded`: True
        - `fill value`: -1
        - `note`: pre-ordered list
- **`list_object_ids_per_class`: list of object ids per class**
        - `available in`: train, val, trainval
        - `dtype`: np.int32
        - `is padded`: True
        - `fill value`: -1
        - `note`: pre-ordered list

Disclaimer

All rights reserved to the original creators of **PASCAL VOC2012**.

For information about the dataset and its terms of use, please see this link.

UCF101 - Action Recognition

UCF101 is an action recognition data set of realistic action videos, collected from YouTube, having 101 action categories. This data set is an extension of **UCF50** data set which has 50 action categories.

With 13320 videos from 101 action categories, **UCF101** gives the largest diversity in terms of actions and with the presence of large variations in camera motion, object appearance and pose, object scale, viewpoint, cluttered background, illumination conditions, etc, it is the most challenging data set to date.

The videos in 101 action categories are grouped into 25 groups, where each group can consist of 4-7 videos of an action.

Use cases

Human action recognition in videos.

Properties

- `name`: ucf_101
- `keywords`: image_processing, recognition, activity, human, single_person

- `dataset size`: 6,9 GB
- `is downloadable`: **yes**
- **`tasks`:**
  - **recognition: (default)**
    - `primary use`: action recognition in videos
    - `description`: Contains videos and action label annotations for action recognition
    - `sets`: train01, train02, train03, test01, test02, test03
    - `metadata file size in disk`: 14,5 MB
    - **`has annotations`: yes**
      - **`which`:**

        activity labels for each video.

Metadata structure (HDF5)

Task: recognition

```
/
├── train01/
│   ├── activities        # dtype=np.uint8, shape=(101,19)        (note: string in␣
↪ASCII format)
│   ├── image_filenames   # dtype=np.uint8, shape=(1788425,113)   (note: string in␣
↪ASCII format)
│   ├── total_frames      # dtype=np.int32, shape=(9537,)
│   ├── video_filenames   # dtype=np.uint8, shape=(9537,60)
│   ├── videos            # dtype=np.uint8, shape=(9537,29)       (note: string in␣
↪ASCII format)
│   ├── object_fields     # dtype=np.uint8, shape=(5,31)          (note: string in␣
↪ASCII format)
│   ├── object_ids        # dtype=np.int32, shape=(9537,5)
│   ├── list_image_filenames_per_video   # dtype=np.int32, shape=(9537,1776)
│   └── list_videos_per_activity         # dtype=np.int32, shape=(101,121)
│
├── test01/
│   ├── activities        # dtype=np.uint8, shape=(101,19)       (note: string in␣
↪ASCII format)
│   ├── image_filenames   # dtype=np.uint8, shape=(697865,113)   (note: string in␣
↪ASCII format)
│   ├── total_frames      # dtype=np.int32, shape=(3783,)
│   ├── video_filenames   # dtype=np.uint8, shape=(3783,60)
│   ├── videos            # dtype=np.uint8, shape=(3783,29)      (note: string in␣
↪ASCII format)
│   ├── object_fields     # dtype=np.uint8, shape=(5,31)         (note: string in␣
↪ASCII format)
│   ├── object_ids        # dtype=np.int32, shape=(3783,5)
│   ├── list_image_filenames_per_video   # dtype=np.int32, shape=(3783,900)
│   └── list_videos_per_activity         # dtype=np.int32, shape=(101,49)
│
├── train02/
│   ├── activities        # dtype=np.uint8, shape=(101,19)        (note: string in␣
↪ASCII format)
```

(continues on next page)

(continued from previous page)

```
│   ├── image_filenames   # dtype=np.uint8, shape=(1791290,113)   (note: string in␣
↪ASCII format)
│   ├── total_frames      # dtype=np.int32, shape=(9586,)
│   ├── video_filenames   # dtype=np.uint8, shape=(9586,60)
│   ├── videos            # dtype=np.uint8, shape=(9586,29)       (note: string in␣
↪ASCII format)
│   ├── object_fields     # dtype=np.uint8, shape=(5,31)          (note: string in␣
↪ASCII format)
│   ├── object_ids        # dtype=np.int32, shape=(9586,5)
│   ├── list_image_filenames_per_video   # dtype=np.int32, shape=(9586,1776)
│   └── list_videos_per_activity         # dtype=np.int32, shape=(101,122)
│
├── test02/
│   ├── activities        # dtype=np.uint8, shape=(101,19)       (note: string in␣
↪ASCII format)
│   ├── image_filenames   # dtype=np.uint8, shape=(695000,113)   (note: string in␣
↪ASCII format)
│   ├── total_frames      # dtype=np.int32, shape=(3734,)
│   ├── video_filenames   # dtype=np.uint8, shape=(3734,60)
│   ├── videos            # dtype=np.uint8, shape=(3734,29)      (note: string in␣
↪ASCII format)
│   ├── object_fields     # dtype=np.uint8, shape=(5,31)         (note: string in␣
↪ASCII format)
│   ├── object_ids        # dtype=np.int32, shape=(3734,5)
│   ├── list_image_filenames_per_video   # dtype=np.int32, shape=(3734,833)
│   └── list_videos_per_activity         # dtype=np.int32, shape=(101,49)
│
├── train03/
│   ├── activities        # dtype=np.uint8, shape=(101,19)        (note: string in␣
↪ASCII format)
│   ├── image_filenames   # dtype=np.uint8, shape=(1786111,113)   (note: string in␣
↪ASCII format)
│   ├── total_frames      # dtype=np.int32, shape=(9624,)
│   ├── video_filenames   # dtype=np.uint8, shape=(9624,60)
│   ├── videos            # dtype=np.uint8, shape=(9624,29)       (note: string in␣
↪ASCII format)
│   ├── object_fields     # dtype=np.uint8, shape=(5,31)          (note: string in␣
↪ASCII format)
│   ├── object_ids        # dtype=np.int32, shape=(9624,5)
│   ├── list_image_filenames_per_video   # dtype=np.int32, shape=(9624,900)
│   └── list_videos_per_activity         # dtype=np.int32, shape=(101,124)
│
└── test03/
    ├── activities        # dtype=np.uint8, shape=(101,19)       (note: string in␣
↪ASCII format)
    ├── image_filenames   # dtype=np.uint8, shape=(700157,113)   (note: string in␣
↪ASCII format)
    ├── total_frames      # dtype=np.int32, shape=(3696,)
    ├── video_filenames   # dtype=np.uint8, shape=(3696,60)
    ├── videos            # dtype=np.uint8, shape=(3696,29)      (note: string in␣
↪ASCII format)
    ├── object_fields     # dtype=np.uint8, shape=(5,31)         (note: string in␣
↪ASCII format)
    ├── object_ids        # dtype=np.int32, shape=(3696,5)
    ├── list_image_filenames_per_video   # dtype=np.int32, shape=(3696,1776)
    └── list_videos_per_activity         # dtype=np.int32, shape=(101,48)
```

Fields

- **activities: activity names**
  - available in: train01, train02, train03, test01, test02, test03
  - dtype: np.uint8
  - is padded: True
  - fill value: 0
  - note: strings stored in ASCII format
- **image_filenames: image file path+name**
  - available in: train01, train02, train03, test01, test02, test03
  - dtype: np.uint8
  - is padded: True
  - fill value: 0
  - note: strings stored in ASCII format
- **total_frames: number of frames per video**
  - available in: train01, train02, train03, test01, test02, test03
  - dtype: np.int32
  - is padded: False
  - fill value: -1
- **videos: video name**
  - available in: train01, train02, train03, test01, test02, test03
  - dtype: np.uint8
  - is padded: True
  - fill value: 0
  - note: strings stored in ASCII format
- **video_filenames: video file path+name**
  - available in: train01, train02, train03, test01, test02, test03
  - dtype: np.uint8
  - is padded: True
  - fill value: 0
  - note: strings stored in ASCII format
- **object_fields: list of field names of the object id list**
  - available in: train01, train02, train03, test01, test02, test03
  - dtype: np.uint8
  - is padded: True
  - fill value: 0
  - note: strings stored in ASCII format

    - `note`: key field (*field name* aggregator)
- **`object_ids`: list of field ids**
    - `available in`: train01, train02, train03, test01, test02, test03
    - `dtype`: np.int32
    - `is padded`: False
    - `fill value`: -1
    - `note`: key field (*field id* aggregator)
- **`list_image_filenames_per_video`: list of image ids per video**
    - `available in`: train01, train02, train03, test01, test02, test03
    - `dtype`: np.int32
    - `is padded`: True
    - `fill value`: -1
    - `note`: pre-ordered list
- **`list_videos_per_activity`: list of video ids per activity**
    - `available in`: train01, train02, train03, test01, test02, test03
    - `dtype`: np.int32
    - `is padded`: True
    - `fill value`: -1
    - `note`: pre-ordered list

Disclaimer

UCF Sports Action

UCF Sports dataset consists of a set of actions collected from various sports which are typically featured on broadcast television channels such as the BBC and ESPN. The video sequences were obtained from a wide range of stock footage websites including BBC Motion gallery and GettyImages.

The dataset includes a total of 150 sequences with the resolution of 720 x 480. The collection represents a natural pool of actions featured in a wide range of scenes and viewpoints.

Use cases

Human action recognition in videos.

Properties

- `name`: ucf_sports
- `keywords`: image_processing, recognition, detection, activity, human, single_person
- `dataset size`: 1,8 GB
- `is downloadable`: **yes**
- **`tasks:`**
  - **recognition: (default)**
    - `primary use`: action recognition in videos
    - `description`: Contains videos and action label annotations for action recognition
    - `sets`: train01, train02, train03, train04, train05, test01, test02, test03, test04, test05
    - `metadata file size in disk`: 1,0 MB
    - **`has annotations:` yes**
      - **`which:`**

        activity labels for each video.

Metadata structure (HDF5)

Task: recognition

```
/
├── train01/
│   ├── activities        # dtype=np.uint8, shape=(10,14)     (note: string in ASCII␣
↪format)
│   ├── boxes             # dtype=np.int32, shape=(6548,4)
│   ├── image_filenames   # dtype=np.uint8, shape=(6548,74)   (note: string in ASCII␣
↪format)
│   ├── videos            # dtype=np.uint8, shape=(103,24)    (note: string in ASCII␣
↪format)
│   ├── object_fields     # dtype=np.uint8, shape=(4,16)      (note: string in ASCII␣
↪format)
│   ├── object_ids        # dtype=np.int32, shape=(6548,4)
│   ├── list_boxes_per_video        # dtype=np.int32, shape=(103,127)
│   ├── list_filenames_per_video    # dtype=np.int32, shape=(103,127)
│   ├── list_object_ids_per_video   # dtype=np.int32, shape=(103,127)
│   └── list_videos_per_activity    # dtype=np.int32, shape=(10,15)
│
├── test01/
│   ├── activities        # dtype=np.uint8, shape=(10,14)     (note: string in ASCII␣
↪format)
│   ├── boxes             # dtype=np.int32, shape=(3032,4)
│   ├── image_filenames   # dtype=np.uint8, shape=(3032,76)   (note: string in ASCII␣
↪format)
│   ├── videos            # dtype=np.uint8, shape=(47,24)     (note: string in ASCII␣
↪format)
│   ├── object_fields     # dtype=np.uint8, shape=(4,16)      (note: string in ASCII␣
↪format)
```

(continues on next page)

(continued from previous page)

```
│   ├── object_ids        # dtype=np.int32, shape=(3032,4)
│   ├── list_boxes_per_video        # dtype=np.int32, shape=(47,144)
│   ├── list_filenames_per_video    # dtype=np.int32, shape=(47,144)
│   ├── list_object_ids_per_video   # dtype=np.int32, shape=(47,144)
│   └── list_videos_per_activity    # dtype=np.int32, shape=(10,7)
│
├── train02/
│   ├── activities        # dtype=np.uint8, shape=(10,14)     (note: string in ASCII␣
↪format)
│   ├── boxes             # dtype=np.int32, shape=(6529,4)
│   ├── image_filenames   # dtype=np.uint8, shape=(6529,76)   (note: string in ASCII␣
↪format)
│   ├── videos            # dtype=np.uint8, shape=(103,24)    (note: string in ASCII␣
↪format)
│   ├── object_fields     # dtype=np.uint8, shape=(4,16)      (note: string in ASCII␣
↪format)
│   ├── object_ids        # dtype=np.int32, shape=(6529,4)
│   ├── list_boxes_per_video        # dtype=np.int32, shape=(103,144)
│   ├── list_filenames_per_video    # dtype=np.int32, shape=(103,144)
│   ├── list_object_ids_per_video   # dtype=np.int32, shape=(103,144)
│   └── list_videos_per_activity    # dtype=np.int32, shape=(10,15)
│
├── test02/
│   ├── activities        # dtype=np.uint8, shape=(10,14)     (note: string in ASCII␣
↪format)
│   ├── boxes             # dtype=np.int32, shape=(3051,4)
│   ├── image_filenames   # dtype=np.uint8, shape=(3051,74)   (note: string in ASCII␣
↪format)
│   ├── videos            # dtype=np.uint8, shape=(47,24)     (note: string in ASCII␣
↪format)
│   ├── object_fields     # dtype=np.uint8, shape=(4,16)      (note: string in ASCII␣
↪format)
│   ├── object_ids        # dtype=np.int32, shape=(3051,4)
│   ├── list_boxes_per_video        # dtype=np.int32, shape=(47,123)
│   ├── list_filenames_per_video    # dtype=np.int32, shape=(47,123)
│   ├── list_object_ids_per_video   # dtype=np.int32, shape=(47,123)
│   └── list_videos_per_activity    # dtype=np.int32, shape=(10,7)
│
├── train03/
│   ├── activities        # dtype=np.uint8, shape=(10,14)     (note: string in ASCII␣
↪format)
│   ├── boxes             # dtype=np.int32, shape=(6537,4)
│   ├── image_filenames   # dtype=np.uint8, shape=(6537,74)   (note: string in ASCII␣
↪format)
│   ├── videos            # dtype=np.uint8, shape=(103,24)    (note: string in ASCII␣
↪format)
│   ├── object_fields     # dtype=np.uint8, shape=(4,16)      (note: string in ASCII␣
↪format)
│   ├── object_ids        # dtype=np.int32, shape=(6537,4)
│   ├── list_boxes_per_video        # dtype=np.int32, shape=(103,144)
│   ├── list_filenames_per_video    # dtype=np.int32, shape=(103,144)
│   ├── list_object_ids_per_video   # dtype=np.int32, shape=(103,144)
│   └── list_videos_per_activity    # dtype=np.int32, shape=(10,15)
│
├── test03/
│   ├── activities        # dtype=np.uint8, shape=(10,14)     (note: string in ASCII␣
↪format)
```

(continues on next page)

(continued from previous page)

```
│   ├── boxes             # dtype=np.int32, shape=(3034,4)
│   ├── image_filenames   # dtype=np.uint8, shape=(3034,76)   (note: string in ASCII␣
↪format)
│   ├── videos            # dtype=np.uint8, shape=(47,24)     (note: string in ASCII␣
↪format)
│   ├── object_fields     # dtype=np.uint8, shape=(4,16)      (note: string in ASCII␣
↪format)
│   ├── object_ids        # dtype=np.int32, shape=(3034,4)
│   ├── list_boxes_per_video        # dtype=np.int32, shape=(47,127)
│   ├── list_filenames_per_video    # dtype=np.int32, shape=(47,127)
│   ├── list_object_ids_per_video   # dtype=np.int32, shape=(47,127)
│   └── list_videos_per_activity    # dtype=np.int32, shape=(10,7)
│
├── train04/
│   ├── activities        # dtype=np.uint8, shape=(10,14)     (note: string in ASCII␣
↪format)
│   ├── boxes             # dtype=np.int32, shape=(6520,4)
│   ├── image_filenames   # dtype=np.uint8, shape=(6520,74)   (note: string in ASCII␣
↪format)
│   ├── videos            # dtype=np.uint8, shape=(103,24)    (note: string in ASCII␣
↪format)
│   ├── object_fields     # dtype=np.uint8, shape=(4,16)      (note: string in ASCII␣
↪format)
│   ├── object_ids        # dtype=np.int32, shape=(6520,4)
│   ├── list_boxes_per_video        # dtype=np.int32, shape=(103,127)
│   ├── list_filenames_per_video    # dtype=np.int32, shape=(103,127)
│   ├── list_object_ids_per_video   # dtype=np.int32, shape=(103,127)
│   └── list_videos_per_activity    # dtype=np.int32, shape=(10,15)
│
├── test04/
│   ├── activities        # dtype=np.uint8, shape=(10,14)     (note: string in ASCII␣
↪format)
│   ├── boxes             # dtype=np.int32, shape=(3060,4)
│   ├── image_filenames   # dtype=np.uint8, shape=(3060,73)   (note: string in ASCII␣
↪format)
│   ├── videos            # dtype=np.uint8, shape=(47,24)     (note: string in ASCII␣
↪format)
│   ├── object_fields     # dtype=np.uint8, shape=(4,16)      (note: string in ASCII␣
↪format)
│   ├── object_ids        # dtype=np.int32, shape=(3060,4)
│   ├── list_boxes_per_video        # dtype=np.int32, shape=(47,144)
│   ├── list_filenames_per_video    # dtype=np.int32, shape=(47,144)
│   ├── list_object_ids_per_video   # dtype=np.int32, shape=(47,144)
│   └── list_videos_per_activity    # dtype=np.int32, shape=(10,7)
│
├── train05/
│   ├── activities        # dtype=np.uint8, shape=(10,14)     (note: string in ASCII␣
↪format)
│   ├── boxes             # dtype=np.int32, shape=(6542,4)
│   ├── image_filenames   # dtype=np.uint8, shape=(6542,76)   (note: string in ASCII␣
↪format)
│   ├── videos            # dtype=np.uint8, shape=(103,24)    (note: string in ASCII␣
↪format)
│   ├── object_fields     # dtype=np.uint8, shape=(4,16)      (note: string in ASCII␣
↪format)
│   ├── object_ids        # dtype=np.int32, shape=(6542,4)
│   ├── list_boxes_per_video        # dtype=np.int32, shape=(103,144)
```

(continues on next page)

(continued from previous page)

```
│   ├── list_filenames_per_video    # dtype=np.int32, shape=(103,144)
│   ├── list_object_ids_per_video   # dtype=np.int32, shape=(103,144)
│   └── list_videos_per_activity    # dtype=np.int32, shape=(10,15)
│
└── test05/
    ├── activities        # dtype=np.uint8, shape=(10,14)     (note: string in ASCII␣
↪format)
    ├── boxes             # dtype=np.int32, shape=(3038,4)
    ├── image_filenames   # dtype=np.uint8, shape=(3038,75)   (note: string in ASCII␣
↪format)
    ├── videos            # dtype=np.uint8, shape=(47,24)     (note: string in ASCII␣
↪format)
    ├── object_fields     # dtype=np.uint8, shape=(4,16)      (note: string in ASCII␣
↪format)
    ├── object_ids        # dtype=np.int32, shape=(3038,4)
    ├── list_boxes_per_video        # dtype=np.int32, shape=(47,127)
    ├── list_filenames_per_video    # dtype=np.int32, shape=(47,127)
    ├── list_object_ids_per_video   # dtype=np.int32, shape=(47,127)
    └── list_videos_per_activity    # dtype=np.int32, shape=(10,7)
```

Fields

- **`activities`: activity names**
  - `available in`: train01, train02, train03, train04, train05, test01, test02, test03, test04, test05
  - `dtype`: np.uint8
  - `is padded`: True
  - `fill value`: 0
  - `note`: strings stored in ASCII format
- **`image_filenames`: image file path+name**
  - `available in`: train01, train02, train03, train04, train05, test01, test02, test03, test04, test05
  - `dtype`: np.uint8
  - `is padded`: True
  - `fill value`: 0
  - `note`: strings stored in ASCII format
- **`boxes`: bounding box coordinates**
  - `available in`: train01, train02, train03, train04, train05, test01, test02, test03, test04, test05
  - `dtype`: np.int32
  - `is padded`: False
  - `fill value`: -1
  - `note`: bbox format [x1,y1,x2,y2]
- **`videos`: video name**
  - `available in`: train01, train02, train03, train04, train05, test01, test02, test03, test04, test05
  - `dtype`: np.uint8

        - `is padded`: True
        - `fill value`: 0
        - `note`: strings stored in ASCII format
- **`object_fields`: list of field names of the object id list**
        - `available in`: train01, train02, train03, train04, train05, test01, test02, test03, test04, test05
        - `dtype`: np.uint8
        - `is padded`: True
        - `fill value`: 0
        - `note`: strings stored in ASCII format
        - `note`: key field (*field name* aggregator)
- **`object_ids`: list of field ids**
        - `available in`: train01, train02, train03, train04, train05, test01, test02, test03, test04, test05
        - `dtype`: np.int32
        - `is padded`: False
        - `fill value`: -1
        - `note`: key field (*field id* aggregator)
- **`list_boxes_per_video`: list of bounding box ids per video**
        - `available in`: train01, train02, train03, train04, train05, test01, test02, test03, test04, test05
        - `dtype`: np.int32
        - `is padded`: True
        - `fill value`: -1
        - `note`: pre-ordered list
- **`list_filenames_per_video`: list of image ids per video**
        - `available in`: train01, train02, train03, train04, train05, test01, test02, test03, test04, test05
        - `dtype`: np.int32
        - `is padded`: True
        - `fill value`: -1
        - `note`: pre-ordered list
- **`list_object_ids_per_video`: list of object ids per video**
        - `available in`: train01, train02, train03, train04, train05, test01, test02, test03, test04, test05
        - `dtype`: np.int32
        - `is padded`: True
        - `fill value`: -1
        - `note`: pre-ordered list
- **`list_videos_per_activity`: list of video ids per activity**
        - `available in`: train01, train02, train03, train04, train05, test01, test02, test03, test04, test05

- `dtype`: np.int32
- `is padded`: True
- `fill value`: -1
- `note`: pre-ordered list

Disclaimer

All rights reserved to the original creators of **UCF-Sports**.

For information about the dataset and its terms of use, please see this link.

Note: All datasets have been parsed by hand and contain most of their relevant annotations. For some, there are also some versions where badly/wrongly annotated information has been discarded. This may be due to wrongly annotated data or because some fields don't provide any useful information as is. If you would like to know more, check out the datasets information available on this page or check the dataset's source website you would like to learn more about.

Warning: Many datasets have a tree structure of folder of how their annotations is stored. Of the available datasets provided by this project, this sturctured information is available in the form of `HDF5` groups in each set under a main group named `raw/`. By default, this information is disabled, so in order to grab this information you can use a special suffix attached at the end of a task name to enable this. To do so, simply append to the end of the task name the suffix `_s`. Note that if a dataset does not have this data setup, it will simply skip this step.

1.1.12 How to contribute

If you are considering to contribute to this project, first let me take the time to thank you for your time! And there are plenty of ways you can help!

Please take a moment to review this document in order to make the contribution process easy and confortable for everyone involved.

Following these guidelines is a great way to effectively contribute to the project and help the community.

This section describes a way to contribute to the project. You are welcome to provide with code or ideas to help improve the functionality this package offers. You can also help by answering questions or helping out with documentation if you prefer.

Note: The main goal behind contributing to this project is to provide tools for the community to help accelerate research and share code with others in a simple and easy way.

Contributing to the project

There are many ways to contribute to the dbcollection project: finding bugs, submitting pull requests, reporting issues and creating suggestions.

Using the issue tracker

The issue tracker is the preferred channel for *bug reports*, *features requests* and *submitting pull requests*. You can also use Stack Overflow to get feedback about your questions.

Feature requests

Feature requests are welcome to be filed. The purpose of feature requests is for others who are looking to implement a feature are aware of the interest in the feature. At the time of writting, this project is solely maintained by a single maintainer (me, on my free time), so please be considerate if it takes a little longer to get feedback on your requests.

Pull requests

Good pull requests - patches, improvements, new features, new datasets - are a fantastic help. They should remain focused in scope and avoid containing unrelated commits.

To enable us to quickly review and accept your pull requests, always create one pull request per issue and link the issue in the pull request. Never merge multiple requests in one unless they have the same root cause. Be sure to follow our *Coding Guidelines* and *Testing Guidelines* and keep code changes as small as possible. Avoid pure formatting changes to code that has not been modified otherwise. Pull requests should contain tests whenever possible.

You can use the following process to create a pull request for this project:

1. Fork the project, clone your fork, and configure the remotes:

```
$ # Clone your fork of the repo into the current directory
$ git clone https://github.com/<your-username>/dbcollection.git
$ # Navigate to the newly cloned directory
$ cd dbcollection
$ # Assign the original repo to a remote called "upstream"
$ git remote add upstream https://github.com/dbcollection/dbcollection.git
```

2. If you have cloned the repository a while ago, get the latest changes from upstream:

```
$ git checkout master
$ git pull upstream master
```

3. Create a new topic branch (off the main project development branch) to contain your feature, change, or fix:

```
$ git checkout -b <topic-branch-name>
```

4. Commit your changes in logical chunks. Use Git's interactive rebase feature to tidy up/organize your commits before making them public. This helps to keep the commit history in logical blocks and clean. For example:

    - If you are adding a new function or a dataset, keep the function/dataset + tests + doc to a single commit unless logically warranted.
    - If you are fixing a bug, keep the bugfix to a single commit unless logically warranted.

    Note: If you are fairly new to git or not very familiar with git conventions, please try to adhere to these git commit message guidelines before committing to the project.

5. Locally merge (or rebase) the upstream development branch into your topic branch:

```
$ git pull [--rebase] upstream master
```

6. Push your topic branch up to your fork:

```
$ git push origin <topic-branch-name>
```

7. Open a Pull Request with a clear title and description.

Where to contribute

Check out the full issues list for a list of all potential areas for contributions (if any). Note that just because an issue exists in the repository does not mean we will accept every contribution to the core project. There are several reasons to not accept a pull request like:

- Performance - One of dbcollection's main concerns is to deliver a simple, yet moderately fast dataset manager. This means it should perform fast enough when downloading/parsing data.
- User experience - Since the goal is to make user's life easy, the use experience should feel confortable enough to encourage using the package. This means that the interface should provide enough information of what's going on, but not overwhelm the user with unnecessary information of what's going on the background.
- Architectural - The community and/or feature owner needs to agree with any architectural impact a change may make. Things like new language APIs *should* be discussed with and agreed upon by the feature owner.

To improve the chances to get a pull request merged you should select an issue that is labelled with the help-wanted or bug labels. If the issue you want to work on is not labelled with `help-wanted` or `bug`, you can start a conversation with the issue owner asking whether an external contribution will be considered.

Suggestions

We're also interested in your feedback for the future of this project. You can submit a suggestion or feature request through the issue tracker. To make this process more effective, we're asking that these include more information to help define them more clearly.

1.1.13 Submitting bugs and suggestions

The dbcollection project tracks issues and feature requests using the GitHub issue tracker.

Before submitting an issue

First, please do a search in open issues to see if the issue or feature request has already been filed. Use this query to search for the most popular feature requests.

If you find your issue already exists, make relevant comments. If you cannot find an existing issue that describes your bug or feature, submit an issue using the guidelines below.

Writing good bug reports and feature requests

A bug is a *demonstrable problem* that is caused by the code in the repository. Good bug reports are extremely helpful.

File a single issue per problem and feature request:

- Do not enumerate multiple bugs or feature requests in the same issue.

- Do not add your issue as a comment to an existing issue unless it's for the identical input. Many issues may look similar, but have different causes.

A good bug report should be easily reproducible. The more information you can provide, the more likely someone will be successful reproducing the issue and finding a fix.

When submitting a report, please try to be as detailed as possible. Please consider including the following with each issue:

- What is your environment?
- What steps will reproduce the issue?
- What OS do you experience the problem?
- What would you expect to be the outcome?
- A code snippet that demonstrates the issue or a link to a code repository we can easily pull down onto our machine to recreate the issue.

All these details will help alot when attempting to fix any potential bugs.

Contributing fixes

If you are interested in fixing issues and contributing directly to the project's code base, please see the document *How to Contribute*.

1.1.14 Feedback channels

This project wants to have its presence on more channels than on GitHub but, for now, we only use the GitHub repo page for assistance.

Github issues

Github issues should be used for bugs and feature requests. How to submit good bugs and feature requests is described in *How to Contribute*.

Note: We don't want to use Github Issues for general 'how-to' questions about dbcollection. Instead, these general 'how-to' questions may/will be redirected to Stack Overflow.

Gitter

We have a Gitter chat! If you prefer interacting with other for asking questions about your issues via a chat room, then you are more than welcome to join us in our gitter room and ask any questions you like. Remember to always be polite with others and help out if you can.

Warning: Don't post huge blobs of code on the chat. If you need to, create a gist with the code sample and share the link instead.

1.1.15 Code organization

dbcollection consists of a modular code written in Python that can be easily extended to other languages. It contains core apy methods, a list of dataset constructors and utility functions in separate folders. These compose the core functionality and features of the dbcollection package which can be used to write other APIS using other languages like Lua or Matlab.

Additionally, the main repository contains other relevant components of the project worth mentioning like the *docs* and *notebooks* directories.

dbcollection

The `dbcollection/` directory contains the main project's files. It is partitioned into four other directories where the *core api functions*, *available datasets*, *unit/functional tests* and *utility functions* are stored.

```
dbcollection/
    core/
    datasets/
    tests/
    utils/
    __init__.py
    _version.py
```

core/

All api methods and classes are stored in this folder.

datasets/

All datasets are stored in this dir, where each dataset is stored in a separate folder with the same name. Related datasets may be stored in different subfolders under the same dir. For more information see the GitHub repository.

tests/

All tests are organized under the `tests/` directory by language and functionality. The project uses two types of tests to check for bugs: unit tests and functional tests. Unit tests are used to test the core functions of the package and functional tests are used to test the execution of downloading and installing a dataset.

The directory is organized with the same dir structure as the dbcollection/ dir. It has the following structure:

```
tests/
    core/
        test_api.py
        test_cache.py
        test_db.py
        test_loader.py
        ...
    functional/
        download/
            cifar10.py
            cifar100.py
            mnist.py
```

(continues on next page)

(continued from previous page)

```
            ...
        load/
            cifar10.py
            cifar100.py
            mnist.py
            ...
        process/
            cifar10.py
            cifar100.py
            mnist.py
            ...
    utils/
        test_pad.py
        test_string_ascii.py
        ...
```

utils/

The utility functions dir contains methods to load files, download urls, extract data, parse strings, manage cache data, etc. Additional functionality should be added in this folder.

Documentation

The `docs/` directory contains the documentation files. We use Sphinx to build our documentation and Read The Docs to host it. The structure of `docs/` is similar to most docs using Sphinx:

```
docs/
    build/
    source/
    make.bat
    Makefile
```

Notebooks

The `notebooks/` directory contains tutorials/demos/guides on using `dbcollection` as a IPython/Jupyter Notebook. These notebooks show how to use the package and show how it can be integrated with your code/research without too much hassle in a simple and interactive way.

To keep it simple, all notebooks are stored under `notebooks/`. All notebooks names should follow this convention `<type>_<language>_<goal>.ipynb` (lower-case) where:

- `type`: This indicates what is the intent of the notebook. You should use a descriptive word that unanimously explains what the notebook is all about. You can use one of these following attributes to categorize the purpose of the notebook: example, tutorial, demo, guide, etc.
- `language`: Target language of the notebook.
- `goal`: What's the end goal of the notebook. This can be a single or multiple words separated by an underscore and it should brifly describe what is purpose of the notebook.

Examples of names:

```
tutorial_python_dbcollection_api.ipynb
tutorial_python_dbcollection_tensorflow.ipynb
example_matlab_install_cifar10.ipynb
demo_lua_mnist.ipynb
```

1.1.16 Coding guidelines

We follow PEP8 as our basic style guideline for all Python code. The following sections describe key aspects of how the code is structured.

Note: For other languages, we try to follow a similar style like in `PEP8`. Exception to this rule goes to languages where the general trend diverges too much from the basic guideline used here (e.g., Java), so adopting those conventions is allowed.

Cross-compatible code

Not all functions are available between versions, so it's important to write code that will be compatible from Python 2.7 through the most recent version of Python 3.

Indentation

Use 4 spaces for identation. Don't use tabs.

Names

- **Variables, functions, methods, packages, modules**
  - `lower_case_with_underscores`
- **Classes and Exceptions**
  - `CapWords`
- **Protected methods and internal functions**
  - `_single_leading_underscore(self, ...)`
- **Private methods**
  - `__double_leading_underscore(self, ...)`
- **Constants**
  - `ALL_CAPS_WITH_UNDERSCORES`

DocStrings

We follow Numpy's docstring style for documenting methods and classes. For core API functions and methods, please try to follow as close as possible the conventions used in the code. For methods defining datasets, a simple docstring is allowed.

Use one-line docstrings for obvious functions.

```
"""Return the pathname of ``foo``."""
```

Multiline docstrings should include

- Summary line
- Use case, if appropriate
- Parameters
- Return type and semantics, unless `None` is returned
- Exceptions (if any is raised)

```
def function_with_types_in_docstring(param1, param2):
    """Example function with types documented in the docstring.

    `PEP 484`_ type annotations are supported. If attribute, parameter, and
    return types are annotated according to `PEP 484`_, they do not need to be
    included in the docstring:

    Parameters
    ----------
    param1 : int
        The first parameter.
    param2 : str
        The second parameter.

    Returns
    -------
    bool
        True if successful, False otherwise.

    .. _PEP 484:
        https://www.python.org/dev/peps/pep-0484/

    """
```

1.1.17 Testing guidelines

First off - thank you for writing test cases - they're really important.

Moreover, testing is an important part of submitted code. You should test your code by unit/functional tests following our testing guidelines. Note that we are using the `pytest` and the `mock` packages for testing, so install these packages before writing your code:

```
$ pip install pytest mock
```

Typical imports

```
import pytest
import mock
import dbcollection
```

Making your tests behave well

Test cases are run after every change (as does Travis), so it's important that you make your tests well-behaved. With this in mind, it's important that your test cases cover the functionality of your addition, so that when others make changes, they can be confident that they aren't introducing errors in your code.

Also, strive to fully test your code, but don't get too obsess over the coverage score.

Unit tests

We use `pytest` for unit testing our code. This framework makes it easy to write small tests and it has an automatic test discovery mechanism. This requires that tests have to be written in a certain way:

- `pytest` will run all files in the current directory and its subdirectories of the form `test_*.py` or `*_test.py`.
- From those files, collected test items require functions or methods outside of a class to start with the prefix `test_`.

Note: For more information about `pytest` testing practices see its documentation.

Besides the conventions required by `pytest`, use these general testing guidelines when writting tests:

- Use long, descriptive names.
- Focus on one bit of functionality.
- Should be fast, but a slow test is better than no test.
- All tests must pass. Moreover, don't let incomplete tests pass.

Functional tests

Functional tests are higher level tests that should be used to test new dataset implementations to check for errors, and are necessary for every submission of a new dataset.

When naming functional tests methods, use the convention `<api_method>_<dataset_name>` and store them in the appropriate language folder in the `tests/` directory.

Any functional tests must follow a common guideline for clarity purposes. Please consider using the following convention when writting your tests:

```
#!/usr/bin/env python3

"""
Test loading cifar10.
"""

import os
from dbcollection.utils.test import TestBaseDB


# setup
name = 'cifar10'
task = 'classification'
data_dir = ''
verbose = True
```

(continues on next page)

(continued from previous page)

```
# Run tester
tester = TestBaseDB(name, task, data_dir, verbose)
tester.run('load')
```

Hook up travis-ci

We use travis for testings the entire library across various python versions. If you hook up your fork to run travis, then it is displayed prominently whether your pull request passes or fails the testing suite. This is incredibly helpful.

If it shows that it passes, great! We can consider merging. If there's a failure, this let's you and us know there is something wrong, and needs some attention before it can be considered for merging.

Sometimes Travis will say a change failed for reasons unrelated to your pull request. For example there could be a build error or network error. To get Travis to retest your pull request, do the following:

```
$ git commit --amend -C HEAD
$ git push origin <yourbranch> -f
```

1.1.18 License

1.2 Indices and tables

- genindex
- modindex

Python Module Index

d

Index

Symbols

A

B

C

D

E

F

G

H

I

L

O

P

R

S

T

U