kwcoco

Aug 26, 2020

Contents

The Kitware COCO module defines a variant of the Microsoft COCO format, originally developed for the "collected images in context" object detection challenge. We are backwards compatible with the original module, but we also have improved implementations in several places, including segmentations and keypoints.

The `kwcoco.CocoDataset` class is capable of dynamic addition and removal of categories, images, and annotations. Has better support for keypoints and segmentation formats than the original COCO format. Despite being written in Python, this data structure is reasonably efficient.

CHAPTER 1

kwcoco package

1.1 Subpackages

1.1.1 kwcoco.cli package

Submodules

kwcoco.cli.coco_eval module

class kwcoco.cli.coco_eval.**CocoEvalCLI**

Bases: object

name = 'eval'

CLIConfig

alias of *kwcoco.coco_evaluator.CocoEvalCLIConfig*

classmethod main(*cmdline=True*, ***kw*)

Example

```
>>> import ubelt as ub
>>> from kwcoco.cli.coco_eval import *  # NOQA
>>> from kwcoco.coco_evaluator import CocoEvaluator
>>> from os.path import join
>>> import kwcoco
>>> dpath = ub.ensure_app_cache_dir('kwcoco/tests/eval')
>>> true_dset = kwcoco.CocoDataset.demo('shapes8')
>>> from kwcoco.demo.perterb import perterb_coco
>>> kwargs = {
>>>     'box_noise': 0.5,
>>>     'n_fp': (0, 10),
```

(continues on next page)

(continued from previous page)

```
>>>     'n_fn': (0, 10),
>>> }
>>> pred_dset = perterb_coco(true_dset, **kwargs)
>>> true_dset.fpath = join(dpath, 'true.mscoco.json')
>>> pred_dset.fpath = join(dpath, 'pred.mscoco.json')
>>> true_dset.dump(true_dset.fpath)
>>> pred_dset.dump(pred_dset.fpath)
>>> CocoEvalCLI.main(true_dataset=true_dset.fpath, pred_dataset=pred_dset.
↪fpath)
```

kwcoco.cli.coco_modify_categories module

class kwcoco.cli.coco_modify_categories.**CocoModifyCatsCLI**

Bases: object

Remove, rename, or coarsen categories.

name = 'modify_categories'

class CLIConfig(*data=None*, *default=None*, *cmdline=False*)

Bases: scriptconfig.config.Config

Rename or remove categories

epilog = '\n Example Usage:\n kwcoco modify_categories --help\n kwcoco modify_categ

default = {'dst': <Value(None: None)>, 'keep': <Value(None: None)>, 'keep_annot

classmethod main(*cmdline=True*, ***kw*)

Example

```
>>> # xdoctest: +SKIP
>>> kw = {'src': 'special:shapes8'}
>>> cmdline = False
>>> cls = CocoModifyCatsCLI
>>> cls.main(cmdline, **kw)
```

kwcoco.cli.coco_rebase module

kwcoco.cli.coco_show module

class kwcoco.cli.coco_show.**CocoShowCLI**

Bases: object

name = 'show'

class CLIConfig(*data=None*, *default=None*, *cmdline=False*)

Bases: scriptconfig.config.Config

Visualize a COCO image using matplotlib, optionally writing it to disk

epilog = '\n Example Usage:\n kwcoco show --help\n kwcoco show --src=special:shapes

default = {'aid': <Value(None: None)>, 'dst': <Value(None: None)>, 'gid': <Val

classmethod main(*cmdline=True*, ***kw*)

Example

```
>>> # xdoctest: +SKIP
>>> kw = {'src': 'special:shapes8'}
>>> cmdline = False
>>> cls = CocoShowCLI
>>> cls.main(cmdline, **kw)
```

kwcoco.cli.coco_split module

class kwcoco.cli.coco_split.**CocoSplitCLI**

Bases: object

name = 'split'

class CLIConfig(*data=None*, *default=None*, *cmdline=False*)

Bases: scriptconfig.config.Config

Split a single COCO dataset into two sub-datasets.

default = {'dst1': <Value(None: 'split1.mscoco.json')>, 'dst2': <Value(None: 's

epilog = '\n Example Usage:\n kwcoco split --src special:shapes8 --dst1=learn.mscoc

classmethod main(*cmdline=True*, ***kw*)

Example

```
>>> kw = {'src': 'special:shapes8',
>>>       'dst1': 'train.json', 'dst2': 'test.json'}
>>> cmdline = False
>>> cls = CocoSplitCLI
>>> cls.main(cmdline, **kw)
```

kwcoco.cli.coco_stats module

class kwcoco.cli.coco_stats.**CocoStatsCLI**

Bases: object

name = 'stats'

class CLIConfig(*data=None*, *default=None*, *cmdline=False*)

Bases: scriptconfig.config.Config

Compute summary statistics about a COCO dataset

default = {'basic': <Value(None: True)>, 'boxes': <Value(None: False)>, 'catfre

epilog = '\n Example Usage:\n kwcoco stats --src=special:shapes8\n kwcoco stats --s

classmethod main(*cmdline=True*, ***kw*)

Example

```
>>> kw = {'src': 'special:shapes8'}
>>> cmdline = False
>>> cls = CocoStatsCLI
>>> cls.main(cmdline, **kw)
```

kwcoco.cli.coco_toydata module

class kwcoco.cli.coco_toydata.**CocoToyDataCLI**

Bases: object

name = 'toydata'

class CLIConfig(*data=None*, *default=None*, *cmdline=False*)

Bases: scriptconfig.config.Config

Create COCO toydata

default = {'dst': <Value(None: 'test.mscoco.json')>, 'key': <Value(None: 'shape

epilog = '\n Example Usage:\n kwcoco toydata --key=shapes8 --dst=toydata.mscoco.jso

classmethod main(*cmdline=True*, ***kw*)

Example

```
>>> kw = {'key': 'shapes8', 'dst': 'test.json'}
>>> cmdline = False
>>> cls = CocoToyDataCLI
>>> cls.main(cmdline, **kw)
```

kwcoco.cli.coco_union module

class kwcoco.cli.coco_union.**CocoUnionCLI**

Bases: object

name = 'union'

class CLIConfig(*data=None*, *default=None*, *cmdline=False*)

Bases: scriptconfig.config.Config

Combine multiple COCO datasets into a single merged dataset.

default = {'dst': <Value(None: 'combo.mscoco.json')>, 'src': <Value(None: [])>}

epilog = '\n Example Usage:\n kwcoco union --src special:shapes8 special:shapes1 --

classmethod main(*cmdline=True*, ***kw*)

Example

```
>>> kw = {'src': ['special:shapes8', 'special:shapes1']}
>>> cmdline = False
>>> cls = CocoUnionCLI
>>> cls.main(cmdline, **kw)
```

Module contents

1.1.2 kwcoco.demo package

Submodules

kwcoco.demo.perterb module

kwcoco.demo.perterb.**perterb_coco**(*coco_dset*, ***kwargs*)

Perterbs a coco dataset

Example

```
>>> from kwcoco.demo.perterb import *  # NOQA
>>> from kwcoco.demo.perterb import _demo_construct_probs
>>> import kwcoco
>>> coco_dset = true_dset = kwcoco.CocoDataset.demo('shapes8')
>>> kwargs = {
>>>     'box_noise': 0.5,
>>>     'n_fp': 3,
>>>     'with_probs': 1,
>>> }
>>> pred_dset = perterb_coco(true_dset, **kwargs)
>>> pred_dset._check_json_serializable()
```

kwcoco.demo.toydata module

kwcoco.demo.toydata.**demodata_toy_img**(*anchors=None*, *gsize=(104*, *104)*, *categories=None*, *n_annots=(0*, *50)*, *fg_scale=0.5*, *bg_scale=0.8*, *bg_intensity=0.1*, *fg_intensity=0.9*, *gray=True*, *centerobj=None*, *exact=False*, *newstyle=True*, *rng=None*, *aux=None*)

Generate a single image with non-overlapping toy objects of available categories.

Parameters

- **anchors** (*ndarray*) – Nx2 base width / height of boxes
- **gsize** (*Tuple[int, int]*) – width / height of the image
- **categories** (*List[str]*) – list of category names
- **n_annots** (*Tuple | int*) – controls how many annotations are in the image. if it is a tuple, then it is interpreted as uniform random bounds
- **fg_scale** (*float*) – standard deviation of foreground intensity
- **bg_scale** (*float*) – standard deviation of background intensity

- **bg_intensity** (*float*) – mean of background intensity
- **fg_intensity** (*float*) – mean of foreground intensity
- **centerobj** (*bool*) – if 'pos', then the first annotation will be in the center of the image, if 'neg', then no annotations will be in the center.
- **exact** (*bool*) – if True, ensures that exactly the number of specified annots are generated.
- **newstyle** (*bool*) – use new-sytle mscoco format
- **rng** (*RandomState*) – the random state used to seed the process
- **aux** – if specified builds auxillary channels

CommandLine: xdoctest -m kwcoco.demo.toydata demodata_toy_img:0 –profile xdoctest -m kwcoco.demo.toydata demodata_toy_img:1 –show

Example

```
>>> from kwcoco.demo.toydata import *  # NOQA
>>> img, anns = demodata_toy_img(gsize=(32, 32), anchors=[[.3, .3]], rng=0)
>>> img['imdata'] = '<ndarray shape={}>'.format(img['imdata'].shape)
>>> print('img = {}'.format(ub.repr2(img)))
>>> print('anns = {}'.format(ub.repr2(anns, nl=2, cbr=True)))
>>> # xdoctest: +IGNORE_WANT
img = {
    'height': 32,
    'imdata': '<ndarray shape=(32, 32, 3)>',
    'width': 32,
}
anns = [{'bbox': [15, 10, 9, 8],
  'category_name': 'star',
  'keypoints': [],
  'segmentation': {'counts': '[`06j0000O20N1000e8', 'size': [32, 32]},},
 {'bbox': [11, 20, 7, 7],
  'category_name': 'star',
  'keypoints': [],
  'segmentation': {'counts': 'g;1m04N0O20N102L[=', 'size': [32, 32]},},
 {'bbox': [4, 4, 8, 6],
  'category_name': 'superstar',
  'keypoints': [{'keypoint_category': 'left_eye', 'xy': [7.25, 6.8125]}, {
↪'keypoint_category': 'right_eye', 'xy': [8.75, 6.8125]}],
  'segmentation': {'counts': 'U4210j0300O01010O00MVO0ed0', 'size': [32, 32]},},
 {'bbox': [3, 20, 6, 7],
  'category_name': 'star',
  'keypoints': [],
  'segmentation': {'counts': 'g31m04N000002L[f0', 'size': [32, 32]},},]
```

Example

```
>>> # xdoctest: +REQUIRES(--show)
>>> img, anns = demodata_toy_img(gsize=(172, 172), rng=None, aux=True)
>>> print('anns = {}'.format(ub.repr2(anns, nl=1)))
>>> import kwplot
>>> kwplot.autompl()
```

(continues on next page)

(continued from previous page)

```
>>> kwplot.imshow(img['imdata'], pnum=(1, 2, 1), fnum=1)
>>> auxdata = img['auxillary'][0]['imdata']
>>> kwplot.imshow(auxdata, pnum=(1, 2, 2), fnum=1)
>>> kwplot.show_if_requested()
```

Ignore: from kwcoco.demo.toydata import * import xinspect globals().update(xinspect.get_kwargs(demodata_toy_img))

kwcoco.demo.toydata.**demodata_toy_dset**(*gsize=(600, 600)*, *n_imgs=5*, *verbose=3*, *rng=0*, *newstyle=True*, *dpath=None*, *aux=None*, *cache=True*)

Create a toy detection problem

Parameters

- **gsize** (*Tuple*) – size of the images
- **n_img** (*int*) – number of images to generate
- **rng** (*int | RandomState*) – random number generator or seed
- **newstyle** (*bool, default=True*) – create newstyle mscoco data
- **dpath** (*str*) – path to the output image directory, defaults to using kwcoco cache dir

Returns dataset in mscoco format

Return type dict

SeeAlso: random_video_dset

CommandLine: xdoctest -m kwcoco.demo.toydata demodata_toy_dset –show

Ignore: import xdev globals().update(xdev.get_func_kwargs(demodata_toy_dset))

Todo:

- [ ] Non-homogeneous images sizes

Example

```
>>> from kwcoco.demo.toydata import *
>>> import kwcoco
>>> dataset = demodata_toy_dset(gsize=(300, 300), aux=True, cache=False)
>>> dpath = ub.ensure_app_cache_dir('kwcoco', 'toy_dset')
>>> dset = kwcoco.CocoDataset(dataset)
>>> # xdoctest: +REQUIRES(--show)
>>> print(ub.repr2(dset.dataset, nl=2))
>>> import kwplot
>>> kwplot.autompl()
>>> dset.show_image(gid=1)
>>> ub.startfile(dpath)
```

kwcoco.demo.toydata.**random_video_dset**(*num_videos=1*, *num_frames=2*, *num_tracks=2*, *anchors=None*, *gsize=(600, 600)*, *verbose=3*, *render=False*, *rng=None*)

Create a toy Coco Video Dataset

Parameters

- **num_videos** – number of videos
- **num_frames** – number of images per video
- **num_tracks** – number of tracks per video
- **gsize** – image size
- **render** (*bool | dict*) – if truthy the toy annotations are synthetically rendered. See `render_toy_image` for details.
- **rng** (*int | None | RandomState*) – random seed / state

SeeAlso: random_single_video_dset

Example

```
>>> from kwcoco.demo.toydata import *  # NOQA
>>> dset = random_video_dset(render=True, num_videos=3, num_frames=2, num_
↪tracks=10)
>>> # xdoctest: +REQUIRES(--show)
>>> dset.show_image(1, doclf=True)
>>> dset.show_image(2, doclf=True)
```

import xdev globals().update(xdev.get_func_kwargs(random_video_dset)) num_videos = 2

kwcoco.demo.toydata.**random_single_video_dset**(*gsize=(600, 600)*, *num_frames=5*, *num_tracks=3*, *tid_start=1*, *gid_start=1*, *video_id=1*, *anchors=None*, *rng=None*, *render=False*, *autobuild=True*, *verbose=3*)

Create the video scene layout of object positions.

Parameters

- **gsize** (*Tuple[int, int]*) – size of the images
- **num_frames** (*int*) – number of frames in this video
- **num_tracks** (*int*) – number of tracks in this video
- **tid_start** (*int, default=1*) – track-id start index
- **gid_start** (*int, default=1*) – image-id start index
- **video_id** (*int, default=1*) – video-id of this video
- **anchors** (*ndarray | None*) – base anchor sizes of the object boxes we will generate.
- **rng** (*RandomState*) – random state / seed
- **render** (*bool | dict*) – if truthy, does the rendering according to provided params in the case of dict input.
- **autobuild** (*bool, default=True*) – prebuild coco lookup indexes
- **verbose** (*int*) – verbosity level

Todo:

- [ ] Need maximum allowed object overlap measure
- [ ] Need better parameterized path generation

Example

```
>>> from kwcoco.demo.toydata import *  # NOQA
>>> anchors = np.array([ [0.3, 0.3],  [0.1, 0.1]])
>>> dset = random_single_video_dset(render=True, num_frames=10, num_tracks=10,␣
↪anchors=anchors)
>>> # xdoctest: +REQUIRES(--show)
>>> # Show the tracks in a single image
>>> import kwplot
>>> kwplot.autompl()
>>> annots = dset.annots()
>>> tids = annots.lookup('track_id')
>>> tid_to_aids = ub.group_items(annots.aids, tids)
>>> paths = []
>>> track_boxes = []
>>> for tid, aids in tid_to_aids.items():
>>>     boxes = dset.annots(aids).boxes.to_cxywh()
>>>     path = boxes.data[:, 0:2]
>>>     paths.append(path)
>>>     track_boxes.append(boxes)
>>> import kwplot
>>> plt = kwplot.autoplt()
>>> ax = plt.gca()
>>> ax.cla()
>>> #
>>> import kwimage
>>> colors = kwimage.Color.distinct(len(track_boxes))
>>> for i, boxes in enumerate(track_boxes):
>>>     color = colors[i]
>>>     path = boxes.data[:, 0:2]
>>>     boxes.draw(color=color, centers={'radius': 0.01}, alpha=0.5)
>>>     ax.plot(path.T[0], path.T[1], 'x-', color=color)
```

Example

```
>>> from kwcoco.demo.toydata import *  # NOQA
>>> anchors = np.array([ [0.2, 0.2],  [0.1, 0.1]])
>>> gsize = np.array([(600, 600)])
>>> print(anchors * gsize)
>>> dset = random_single_video_dset(render=True, num_frames=10, anchors=anchors,␣
↪num_tracks=10)
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> plt = kwplot.autoplt()
>>> plt.clf()
>>> gids = list(dset.imgs.keys())
>>> pnums = kwplot.PlotNums(nSubplots=len(gids), nRows=1)
>>> for gid in gids:
>>>     dset.show_image(gid, pnum=pnums(), fnum=1, title=False)
>>> pnums = kwplot.PlotNums(nSubplots=len(gids))
```

kwcoco.demo.toydata.**render_toy_dataset**(*dset*, *rng*, *dpath=None*, *renderkw=None*)

Create toydata renderings for a preconstructed coco dataset.

Example

```
>>> from kwcoco.demo.toydata import *  # NOQA
>>> import kwarray
>>> rng = None
>>> rng = kwarray.ensure_rng(rng)
>>> num_tracks = 3
>>> dset = random_video_dset(rng=rng, num_videos=3, num_frames=10, num_tracks=3)
>>> dset = render_toy_dataset(dset, rng)
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> plt = kwplot.autoplt()
>>> plt.clf()
>>> gids = list(dset.imgs.keys())
>>> pnums = kwplot.PlotNums(nSubplots=len(gids), nRows=num_tracks)
>>> for gid in gids:
>>>     dset.show_image(gid, pnum=pnums(), fnum=1, title=False)
>>> pnums = kwplot.PlotNums(nSubplots=len(gids))
>>> #
>>> # for gid in gids:
>>> #     canvas = dset.draw_image(gid)
>>> #     kwplot.imshow(canvas, pnum=pnums(), fnum=2)
```

kwcoco.demo.toydata.**render_toy_image**(*dset*, *gid*, *rng=None*, *renderkw=None*)

Modifies dataset inplace, rendering synthetic annotations

Parameters

- **dset** (*CocoDataset*) – coco dataset with renderable anotations / images
- **gid** (*int*) – image to render
- **rng** (*int | None | RandomState*) – random state
- **renderkw** (*dict*) – rendering config gray (boo): gray or color images fg_scale (float): foreground noisyness (gauss std) bg_scale (float): background noisyness (gauss std) fg_intensity (float): foreground brightness (gauss mean) bg_intensity (float): background brightness (gauss mean) newstyle (bool): use new kwcoco datastructure formats with_kpts (bool): include keypoint info with_sseg (bool): include segmentation info

Example

```
>>> from kwcoco.demo.toydata import *  # NOQA
>>> gsize=(600, 600)
>>> num_frames=5
>>> verbose=3
>>> rng = None
>>> import kwarray
>>> rng = kwarray.ensure_rng(rng)
>>> dset = random_video_dset(
>>>     gsize=gsize, num_frames=num_frames, verbose=verbose, rng=rng, num_
↪videos=2)
>>> print('dset.dataset = {}'.format(ub.repr2(dset.dataset, nl=2)))
>>> gid = 1
>>> renderkw = dict(
...     gray=0,
... )
```

(continues on next page)

(continued from previous page)

```
>>> render_toy_image(dset, gid, rng, renderkw=renderkw)
>>> gid = 1
>>> canvas = dset.imgs[gid]['imdata']
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.imshow(canvas, doclf=True)
>>> dets = dset.annots(gid=gid).detections
>>> dets.draw()
```

kwcoco.demo.toydata.**random_multi_object_path**(*num_objects*, *num_frames*, *rng=None*)

num_objects = 30 num_frames = 30

from kwcoco.demo.toydata import * # NOQA paths = random_multi_object_path(num_objects, num_frames, rng)

import kwplot plt = kwplot.autoplt() ax = plt.gca() ax.cla() ax.set_xlim(-.01, 1.01) ax.set_ylim(-.01, 1.01)

rng = None

for path in paths: ax.plot(path.T[0], path.T[1], 'x-')

kwcoco.demo.toydata.**random_path**(*num*, *degree=1*, *dimension=2*, *rng=None*, *mode='walk'*)

Create a random path using a bezier curve.

Parameters

- **num** (*int*) – number of points in the path
- **degree** (*int, default=1*) – degree of curvieness of the path
- **dimension** (*int, default=2*) – number of spatial dimensions
- **rng** (*RandomState, default=None*) – seed

References

https://github.com/dhermes/bezier

Example

```
>>> from kwcoco.demo.toydata import *  # NOQA
>>> num = 10
>>> dimension = 2
>>> degree = 3
>>> rng = None
>>> path = random_path(num, degree, dimension, rng, mode='walk')
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> plt = kwplot.autoplt()
>>> kwplot.multi_plot(xdata=path[:, 0], ydata=path[:, 1], fnum=1, doclf=1,␣
↪xlim=(0, 1), ylim=(0, 1))
>>> kwplot.show_if_requested()
```

kwcoco.demo.toypatterns module

class kwcoco.demo.toypatterns.**CategoryPatterns**(*categories=None*, *fg_scale=0.5*, *fg_intensity=0.9*, *rng=None*)

Bases: object

Example

```
>>> self = CategoryPatterns.coerce()
>>> chip = np.zeros((100, 100, 3))
>>> offset = (20, 10)
>>> dims = (160, 140)
>>> info = self.random_category(chip, offset, dims)
>>> print('info = {}'.format(ub.repr2(info, nl=1)))
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(info['data'], pnum=(1, 2, 1), fnum=1, title='chip-space')
>>> kpts = kwimage.Points._from_coco(info['keypoints'])
>>> kpts.translate(-np.array(offset)).draw(radius=3)
>>> #####
>>> mask = kwimage.Mask.coerce(info['segmentation'])
>>> kwplot.imshow(mask.to_c_mask().data, pnum=(1, 2, 2), fnum=1, title='img-space
↪')
>>> kpts.draw(radius=3)
>>> kwplot.show_if_requested()
```

classmethod coerce(*data=None*, ***kwargs*)

Construct category patterns from either defaults or only with specific categories. Can accept either an existig category pattern object, a list of known catnames, or mscoco category dictionaries.

Example

```
>>> data = ['superstar']
>>> self = CategoryPatterns.coerce(data)
```

index(*name*)

get(*index*, *default=NoParam*)

random_category(*chip*, *xy_offset=None*, *dims=None*, *newstyle=True*)

Ignore: import xdev globals().update(xdev.get_func_kwargs(self.random_category))

Example

```
>>> from kwcoco.demo.toypatterns import *  # NOQA
>>> self = CategoryPatterns.coerce(['superstar'])
>>> chip = np.random.rand(64, 64)
>>> info = self.random_category(chip)
```

render_category(*cname*, *chip*, *xy_offset=None*, *dims=None*, *newstyle=True*)

Ignore: import xdev globals().update(xdev.get_func_kwargs(self.random_category))

Example

```
>>> self = CategoryPatterns.coerce(['superstar'])
>>> chip = np.random.rand(64, 64)
>>> info = self.render_category('superstar', chip, newstyle=True)
>>> print('info = {}'.format(ub.repr2(info, nl=-1)))
>>> info = self.render_category('superstar', chip, newstyle=False)
>>> print('info = {}'.format(ub.repr2(info, nl=-1)))
```

kwcoco.demo.toypatterns.**star**(*a*, *dtype=<class 'numpy.uint8'>*)

Generates a star shaped structuring element.

Much faster than skimage.morphology version

class kwcoco.demo.toypatterns.**Rasters**

Bases: object

static superstar()

test data patch

Ignore:

```
>>> kwplot.autompl()
>>> patch = Rasters.superstar()
>>> data = np.clip(kwimage.imscale(patch, 2.2), 0, 1)
>>> kwplot.imshow(data)
```

static eff()

test data patch

Ignore:

```
>>> kwplot.autompl()
>>> eff = kwimage.draw_text_on_image(None, 'F', (0, 1), valign='top')
>>> patch = Rasters.eff()
>>> data = np.clip(kwimage.imscale(Rasters.eff(), 2.2), 0, 1)
>>> kwplot.imshow(data)
```

Module contents

1.1.3 kwcoco.metrics package

Submodules

kwcoco.metrics.assignment module

Todo:

- **[ ] _fast_pdist_priority: Look at absolute difference in sibling entropy** when deciding whether to go up or down in the tree.
- **[ ] medschool applications true-pred matching (applicant proposing) fast** algorithm.
- **[ ] Maybe looping over truth rather than pred is faster? but it makes you** have to combine pred score / ious, which is weird.
- **[x] preallocate ndarray and use hstack to build confusion vectors?**

  - doesn't help

- **[ ] relevant classes / classes / classes-of-interest we care about needs** to be a first class member of detection metrics.

---

kwcoco.metrics.clf_report module

kwcoco.metrics.clf_report.**classification_report**(*y_true*, *y_pred*, *target_names=None*, *sample_weight=None*, *verbose=False*)

Computes a classification report which is a collection of various metrics commonly used to evaulate classification quality. This can handle binary and multiclass settings.

Note that this function does not accept probabilities or scores and must instead act on final decisions. See ovr_classification_report for a probability based report function using a one-vs-rest strategy.

This emulates the bm(cm) Matlab script written by David Powers that is used for computing bookmaker, markedness, and various other scores.

References

https://csem.flinders.edu.au/research/techreps/SIE07001.pdf https://www.mathworks.com/matlabcentral/fileexchange/5648-bm-cm-?requestedDomain=www.mathworks.com Jurman, Riccadonna, Furlanello, (2012). A Comparison of MCC and CEN

Error Measures in MultiClass Prediction

Example

```
>>> # xdoctest: +IGNORE_WANT
>>> # xdoctest: +REQUIRES(module:sklearn)
>>> y_true = [1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3]
>>> y_pred = [1, 2, 1, 3, 1, 2, 2, 3, 2, 2, 3, 3, 2, 3, 3, 3, 1, 3]
>>> target_names = None
>>> sample_weight = None
>>> report = classification_report(y_true, y_pred, verbose=0)
>>> print(report['confusion'])
pred  1  2  3  Σr
real
1     3  1  1   5
2     0  4  1   5
3     1  1  6   8
Σp    4  6  8  18
>>> print(report['metrics'])
metric    precision  recall    fpr  markedness  bookmaker    mcc  support
class
1            0.7500  0.6000 0.0769      0.6071     0.5231 0.5635        5
2            0.6667  0.8000 0.1538      0.5833     0.6462 0.6139        5
3            0.7500  0.7500 0.2000      0.5500     0.5500 0.5500        8
combined     0.7269  0.7222 0.1530      0.5751     0.5761 0.5758       18
```

Ignore:

```
>>> size = 100
>>> rng = np.random.RandomState(0)
>>> p_classes = np.array([.90, .05, .05][0:2])
>>> p_classes = p_classes / p_classes.sum()
>>> p_wrong   = np.array([.03, .01, .02][0:2])
>>> y_true = testdata_ytrue(p_classes, p_wrong, size, rng)
>>> rs = []
>>> for x in range(17):
>>>     p_wrong += .05
>>>     y_pred = testdata_ypred(y_true, p_wrong, rng)
>>>     report = classification_report(y_true, y_pred, verbose='hack')
>>>     rs.append(report)
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> import pandas as pd
>>> df = pd.DataFrame(rs).drop(['raw'], axis=1)
>>> delta = df.subtract(df['target'], axis=0)
>>> sqrd_error = np.sqrt((delta ** 2).sum(axis=0))
>>> print('Error')
>>> print(sqrd_error.sort_values())
>>> ys = df.to_dict(orient='list')
>>> kwplot.multi_plot(ydata_list=ys)
```

kwcoco.metrics.clf_report.**ovr_classification_report**(*mc_y_true*, *mc_probs*, *target_names=None*, *sample_weight=None*, *metrics=None*)

One-vs-rest classification report

Parameters

- **mc_y_true** – multiclass truth labels (integer label format)
- **mc_probs** – multiclass probabilities for each class [N x C]

Example

```
>>> # xdoctest: +IGNORE_WANT
>>> # xdoctest: +REQUIRES(module:sklearn)
>>> from kwcoco.metrics.clf_report import *  # NOQA
>>> y_true = [1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 0, 0, 0, 0, 0, 0, 0, 0]
>>> y_probs = np.random.rand(len(y_true), max(y_true) + 1)
>>> target_names = None
>>> sample_weight = None
>>> verbose = True
>>> report = ovr_classification_report(y_true, y_probs)
>>> print(report['ave'])
auc      0.6541
ap       0.6824
kappa    0.0963
mcc      0.1002
brier    0.2214
dtype: float64
>>> print(report['ovr'])
     auc     ap  kappa    mcc  brier  support  weight
```

(continues on next page)

(continued from previous page)

```
0 0.6062 0.6161 0.0526 0.0598 0.2608         8  0.4444
1 0.5846 0.6014 0.0000 0.0000 0.2195         5  0.2778
2 0.8000 0.8693 0.2623 0.2652 0.1602         5  0.2778
```

Ignore:

```
>>> y_true = [1, 1, 1]
>>> y_probs = np.random.rand(len(y_true), 3)
>>> target_names = None
>>> sample_weight = None
>>> verbose = True
>>> report = ovr_classification_report(y_true, y_probs)
>>> print(report['ovr'])
```

kwcoco.metrics.confusion_vectors module

class kwcoco.metrics.confusion_vectors.**ConfusionVectors**(*data*, *classes*, *probs=None*)

Bases: ubelt.util_mixins.NiceRepr

Stores information used to construct a confusion matrix. This includes corresponding vectors of predicted labels, true labels, sample weights, etc...

Variables

- **data** (*DataFrameArray*) – should at least have keys true, pred, weight
- **classes** (*Sequence | CategoryTree*) – list of category names or category graph
- **probs** (*ndarray, optional*) – probabilities for each class

Example

```
>>> # xdoctest: IGNORE_WANT
>>> from kwcoco.metrics import DetectionMetrics
>>> dmet = DetectionMetrics.demo(
>>>     nimgs=10, nboxes=(0, 10), n_fp=(0, 1), nclasses=3)
>>> cfsn_vecs = dmet.confusion_vectors()
>>> print(cfsn_vecs.data._pandas())
     pred  true   score  weight     iou  txs  pxs  gid
0       2     2 10.0000  1.0000  1.0000    0    4    0
1       2     2  7.5025  1.0000  1.0000    1    3    0
2       1     1  5.0050  1.0000  1.0000    2    2    0
3       3    -1  2.5075  1.0000 -1.0000   -1    1    0
4       2    -1  0.0100  1.0000 -1.0000   -1    0    0
5      -1     2  0.0000  1.0000 -1.0000    3   -1    0
6      -1     2  0.0000  1.0000 -1.0000    4   -1    0
7       2     2 10.0000  1.0000  1.0000    0    5    1
8       2     2  8.0020  1.0000  1.0000    1    4    1
9       1     1  6.0040  1.0000  1.0000    2    3    1
..    ...   ...     ...     ...     ...  ...  ...  ...
62     -1     2  0.0000  1.0000 -1.0000    7   -1    7
63     -1     3  0.0000  1.0000 -1.0000    8   -1    7
64     -1     1  0.0000  1.0000 -1.0000    9   -1    7
```

(continues on next page)

(continued from previous page)

```
65      1    -1 10.0000  1.0000 -1.0000   -1    0    8
66      1     1  0.0100  1.0000  1.0000    0    1    8
67      3    -1 10.0000  1.0000 -1.0000   -1    3    9
68      2     2  6.6700  1.0000  1.0000    0    2    9
69      2     2  3.3400  1.0000  1.0000    1    1    9
70      3    -1  0.0100  1.0000 -1.0000   -1    0    9
71     -1     2  0.0000  1.0000 -1.0000    2   -1    9
```

```
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> from kwcoco.metrics.confusion_vectors import ConfusionVectors
>>> cfsn_vecs = ConfusionVectors.demo(
>>>     nimgs=128, nboxes=(0, 10), n_fp=(0, 3), n_fn=(0, 3), nclasses=3)
>>> cx_to_binvecs = cfsn_vecs.binarize_ovr()
>>> measures = cx_to_binvecs.measures()['perclass']
>>> print('measures = {!r}'.format(measures))
measures = <PerClass_Measures({
    'cat_1': <Measures({'ap': 0.7501, 'auc': 0.7170, 'catname': cat_1, 'max_f1':␣
↪f1=0.77@0.41, 'max_mcc': mcc=0.71@0.44, 'nsupport': 787.0000, 'realneg_total':␣
↪594.0000, 'realpos_total': 193.0000})>,
    'cat_2': <Measures({'ap': 0.8288, 'auc': 0.8137, 'catname': cat_2, 'max_f1':␣
↪f1=0.83@0.40, 'max_mcc': mcc=0.78@0.40, 'nsupport': 787.0000, 'realneg_total':␣
↪589.0000, 'realpos_total': 198.0000})>,
    'cat_3': <Measures({'ap': 0.7536, 'auc': 0.7150, 'catname': cat_3, 'max_f1':␣
↪f1=0.77@0.40, 'max_mcc': mcc=0.71@0.42, 'nsupport': 787.0000, 'realneg_total':␣
↪578.0000, 'realpos_total': 209.0000})>,
}) at 0x7f1b9b0d6130>
```

```
>>> kwplot.figure(fnum=1, doclf=True)
>>> measures.draw(key='pr', fnum=1, pnum=(1, 3, 1))
>>> measures.draw(key='roc', fnum=1, pnum=(1, 3, 2))
>>> measures.draw(key='mcc', fnum=1, pnum=(1, 3, 3))
...
```

classmethod from_json(*state*)

classmethod demo(***kw*)

Example

```
>>> cfsn_vecs = ConfusionVectors.demo()
>>> print('cfsn_vecs = {!r}'.format(cfsn_vecs))
>>> cx_to_binvecs = cfsn_vecs.binarize_ovr()
>>> print('cx_to_binvecs = {!r}'.format(cx_to_binvecs))
```

classmethod from_arrays(*true*, *pred=None*, *score=None*, *weight=None*, *probs=None*, *classes=None*)

Construct confusion vector data structure from component arrays

Example

```
>>> import kwarray
>>> classes = ['person', 'vehicle', 'object']
>>> rng = kwarray.ensure_rng(0)
>>> true = (rng.rand(10) * len(classes)).astype(np.int)
>>> probs = rng.rand(len(true), len(classes))
>>> cfsn_vecs = ConfusionVectors.from_arrays(true=true, probs=probs,␣
↪classes=classes)
>>> cfsn_vecs.confusion_matrix()
pred     person  vehicle  object
real
person        0        0       0
vehicle       2        4       1
object        2        1       0
```

confusion_matrix(*raw=False*, *compress=False*)

Builds a confusion matrix from the confusion vectors.

Parameters **raw** (*bool*) – if True uses 'pred_raw' otherwise used 'pred'

Returns

cm [the labeled confusion matrix]

(Note: we should write a efficient replacement for this use case. #remove_pandas)

Return type pd.DataFrame

CommandLine: xdoctest -m ~/code/kwcoco/kwcoco/metrics/confusion_vectors.py ConfusionVectors.confusion_matrix

Example

```
>>> from kwcoco.metrics import DetectionMetrics
>>> dmet = DetectionMetrics.demo(
>>>     nimgs=10, nboxes=(0, 10), n_fp=(0, 1), n_fn=(0, 1), nclasses=3, cls_
↪noise=.2)
>>> cfsn_vecs = dmet.confusion_vectors()
>>> cm = cfsn_vecs.confusion_matrix()
...
>>> print(cm.to_string(float_format=lambda x: '%.2f' % x))
pred        background  cat_1  cat_2  cat_3
real
background        0.00   1.00   1.00   1.00
cat_1             2.00  12.00   0.00   1.00
cat_2             2.00   0.00  14.00   1.00
cat_3             1.00   0.00   1.00  17.00
```

coarsen(*cxs*)

Creates a coarsened set of vectors

binarize_peritem(*negative_classes=None*)

Creates a binary representation useful for measuring the performance of detectors. It is assumed that scores of "positive" classes should be high and "negative" clases should be low.

Parameters **negative_classes** (*List[str | int]*) – list of negative class names or idxs, by default chooses any class with a true class index of -1. These classes should ideally have low scores.

Returns BinaryConfusionVectors

Example

```
>>> from kwcoco.metrics import DetectionMetrics
>>> dmet = DetectionMetrics.demo(
>>>     nimgs=10, nboxes=(0, 10), n_fp=(0, 1), nclasses=3)
>>> cfsn_vecs = dmet.confusion_vectors()
>>> class_idxs = list(dmet.classes.node_to_idx.values())
>>> binvecs = cfsn_vecs.binarize_peritem()
```

binarize_ovr(*mode=1*, *keyby='name'*, *ignore_classes={'ignore'}*)

Transforms cfsn_vecs into one-vs-rest BinaryConfusionVectors for each category.

Parameters

- **mode** (*int, default=1*) – 0 for heirarchy aware or 1 for voc like. MODE 0 IS PROBABLY BROKEN
- **keyby** (*int | str*) – can be cx or name
- **ignore_classes** (*Set[str]*) – category names to ignore

Returns

which behaves like Dict[int, BinaryConfusionVectors]: cx_to_binvecs

Return type *OneVsRestConfusionVectors*

Example

```
>>> cfsn_vecs = ConfusionVectors.demo()
>>> print('cfsn_vecs = {!r}'.format(cfsn_vecs))
>>> catname_to_binvecs = cfsn_vecs.binarize_ovr(keyby='name')
>>> print('catname_to_binvecs = {!r}'.format(catname_to_binvecs))
```

Notes

Consider we want to measure how well we can classify beagles.

Given a multiclass confusion vector, we need to carefully select a subset. We ignore any truth that is coarser than our current label. We also ignore any background predictions on irrelevant classes

dog | dog <- ignore coarser truths dog | cat <- ignore coarser truths dog | beagle <- ignore coarser truths cat | dog cat | cat cat | background <- ignore failures to predict unrelated classes cat | maine-coon beagle | beagle beagle | dog beagle | background beagle | cat Snoopy | beagle Snoopy | cat maine-coon | background <- ignore failures to predict unrelated classes maine-coon | beagle maine-coon | cat

Anything not marked as ignore is counted. We count anything marked as beagle or a finer grained class (e.g. Snoopy) as a positive case. All other cases are negative. The scores come from the predicted probability of beagle, which must be remembered outside the dataframe.

classification_report(*verbose=0*)

Build a classification report with various metrics.

Example

```
>>> from kwcoco.metrics.confusion_vectors import *  # NOQA
>>> cfsn_vecs = ConfusionVectors.demo()
>>> report = cfsn_vecs.classification_report(verbose=1)
```

class kwcoco.metrics.confusion_vectors.**OneVsRestConfusionVectors**(*cx_to_binvecs*, *classes*)

Bases: ubelt.util_mixins.NiceRepr

Container for multiple one-vs-rest binary confusion vectors

Variables

- **cx_to_binvecs** –
- **classes** –

Example

```
>>> from kwcoco.metrics import DetectionMetrics
>>> dmet = DetectionMetrics.demo(
>>>     nimgs=10, nboxes=(0, 10), n_fp=(0, 1), nclasses=3)
>>> cfsn_vecs = dmet.confusion_vectors()
>>> self = cfsn_vecs.binarize_ovr(keyby='name')
>>> print('self = {!r}'.format(self))
```

classmethod demo()

keys()

measures(***kwargs*)

Example

```
>>> self = OneVsRestConfusionVectors.demo()
>>> thresh_result = self.measures()['perclass']
```

ovr_classification_report()

class kwcoco.metrics.confusion_vectors.**BinaryConfusionVectors**(*data*, *cx=None*, *classes=None*)

Bases: ubelt.util_mixins.NiceRepr

Stores information about a binary classification problem. This is always with respect to a specific class, which is given by *cx* and *classes*.

The *data* DataFrameArray must contain *is_true* - if the row is an instance of class *classes[cx]* *pred_score* - the predicted probability of class *classes[cx]*, and *weight* - sample weight of the example

Example

```
>>> self = BinaryConfusionVectors.demo(n=10)
>>> print('self = {!r}'.format(self))
>>> print('pr = {}'.format(ub.repr2(self.measures())))
>>> print('roc = {}'.format(ub.repr2(self.roc())))
```

```
>>> self = BinaryConfusionVectors.demo(n=0)
>>> print('pr = {}'.format(ub.repr2(self.measures())))
>>> print('roc = {}'.format(ub.repr2(self.roc())))
```

```
>>> self = BinaryConfusionVectors.demo(n=1)
>>> print('pr = {}'.format(ub.repr2(self.measures())))
>>> print('roc = {}'.format(ub.repr2(self.roc())))
```

```
>>> self = BinaryConfusionVectors.demo(n=2)
>>> print('self = {!r}'.format(self))
>>> print('pr = {}'.format(ub.repr2(self.measures())))
>>> print('roc = {}'.format(ub.repr2(self.roc())))
```

classmethod demo(*n=10*, *p_true=0.5*, *p_error=0.2*, *rng=None*)

Create random data for tests

Example

```
>>> from kwcoco.metrics.confusion_vectors import *  # NOQA
>>> cfsn = BinaryConfusionVectors.demo(n=1000, p_error=0.1)
>>> measures = cfsn.measures()
>>> print('measures = {}'.format(ub.repr2(measures, nl=1)))
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.figure(fnum=1, pnum=(1, 2, 1))
>>> measures.draw('pr')
>>> kwplot.figure(fnum=1, pnum=(1, 2, 2))
>>> measures.draw('roc')
```

catname

draw_distribution()

precision_recall(*stabalize_thresh=7*, *stabalize_pad=7*, *method='sklearn'*)

Deprecated, all information lives in measures now

roc(*fp_cutoff=None*, *stabalize_thresh=7*, *stabalize_pad=7*)

Deprecated, all information lives in measures now

measures

memoization decorator for a method that respects args and kwargs

References

http://code.activestate.com/recipes/577452-a-memoize-decorator-for-instance-methods/

Example

```
>>> import ubelt as ub
>>> closure = {'a': 'b', 'c': 'd'}
>>> incr = [0]
>>> class Foo(object):
```

(continues on next page)

(continued from previous page)

```
>>>     @memoize_method
>>>     def foo_memo(self, key):
>>>         value = closure[key]
>>>         incr[0] += 1
>>>         return value
>>>     def foo(self, key):
>>>         value = closure[key]
>>>         incr[0] += 1
>>>         return value
>>> self = Foo()
>>> assert self.foo('a') == 'b' and self.foo('c') == 'd'
>>> assert incr[0] == 2
>>> print('Call memoized version')
>>> assert self.foo_memo('a') == 'b' and self.foo_memo('c') == 'd'
>>> assert incr[0] == 4
>>> assert self.foo_memo('a') == 'b' and self.foo_memo('c') == 'd'
>>> print('Counter should no longer increase')
>>> assert incr[0] == 4
>>> print('Closure changes result without memoization')
>>> closure = {'a': 0, 'c': 1}
>>> assert self.foo('a') == 0 and self.foo('c') == 1
>>> assert incr[0] == 6
>>> assert self.foo_memo('a') == 'b' and self.foo_memo('c') == 'd'
>>> print('Constructing a new object should get a new cache')
>>> self2 = Foo()
>>> self2.foo_memo('a')
>>> assert incr[0] == 7
>>> self2.foo_memo('a')
>>> assert incr[0] == 7
```

class kwcoco.metrics.confusion_vectors.**Measures**(*roc_info*)

Bases: ubelt.util_mixins.NiceRepr, *kwcoco.metrics.util.DictProxy*

Example

```
>>> from kwcoco.metrics.confusion_vectors import *  # NOQA
>>> binvecs = BinaryConfusionVectors.demo(n=100, p_error=0.5)
>>> self = binvecs.measures()
>>> print('self = {!r}'.format(self))
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> self.draw(doclf=True)
>>> self.draw(key='pr',  pnum=(1, 2, 1))
>>> self.draw(key='roc', pnum=(1, 2, 2))
>>> kwplot.show_if_requested()
```

catname

summary()

draw(*key=None*, *prefix=''*, ***kw*)

Example

```
>>> cfsn_vecs = ConfusionVectors.demo()
>>> ovr_cfsn = cfsn_vecs.binarize_ovr(keyby='name')
>>> self = ovr_cfsn.measures()['perclass']
>>> self.draw('mcc', doclf=True, fnum=1)
>>> self.draw('pr', doclf=1, fnum=2)
>>> self.draw('roc', doclf=1, fnum=3)
```

summary_plot (*fnum=1*, *title=''*)

Example

```
>>> from kwcoco.metrics.confusion_vectors import *  # NOQA
>>> cfsn_vecs = ConfusionVectors.demo(n=100, p_error=0.5)
>>> binvecs = cfsn_vecs.binarize_peritem()
>>> self = binvecs.measures()
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> self.summary_plot()
>>> kwplot.show_if_requested()
```

class kwcoco.metrics.confusion_vectors.**PerClass_Measures**(*cx_to_info*)

Bases: ubelt.util_mixins.NiceRepr, *kwcoco.metrics.util.DictProxy*

summary()

draw (*key='mcc'*, *prefix=''*, ***kw*)

Example

```
>>> cfsn_vecs = ConfusionVectors.demo()
>>> ovr_cfsn = cfsn_vecs.binarize_ovr(keyby='name')
>>> self = ovr_cfsn.measures()['perclass']
>>> self.draw('mcc', doclf=True, fnum=1)
>>> self.draw('pr', doclf=1, fnum=2)
>>> self.draw('roc', doclf=1, fnum=3)
```

draw_roc (*prefix=''*, ***kw*)

draw_pr (*prefix=''*, ***kw*)

summary_plot (*fnum=1*, *title=''*)

CommandLine: python ~/code/kwcoco/kwcoco/metrics/confusion_vectors.py PerClass_Measures.summary_plot --show

Example

```
>>> from kwcoco.metrics.confusion_vectors import *  # NOQA
>>> from kwcoco.metrics.detect_metrics import DetectionMetrics
>>> dmet = DetectionMetrics.demo(
>>>     n_fp=(0, 5), n_fn=(0, 5), nimgs=128, nboxes=(0, 10),
```

(continues on next page)

(continued from previous page)

```
>>>     nclasses=3)
>>> cfsn_vecs = dmet.confusion_vectors()
>>> ovr_cfsn = cfsn_vecs.binarize_ovr(keyby='name')
>>> self = ovr_cfsn.measures()['perclass']
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> self.summary_plot(title='demo summary_plot ovr')
>>> kwplot.show_if_requested()
```

kwcoco.metrics.detect_metrics module

class kwcoco.metrics.detect_metrics.**DetectionMetrics**(*classes=None*)

Bases: ubelt.util_mixins.NiceRepr

Variables

- **gid_to_true_dets** (*Dict*) – maps image ids to truth
- **gid_to_pred_dets** (*Dict*) – maps image ids to predictions
- **classes** (CategoryTree) – category coder

Example

```
>>> dmet = DetectionMetrics.demo(
>>>     nimgs=100, nboxes=(0, 3), n_fp=(0, 1), nclasses=8, score_noise=0.9,␣
↪hacked=False)
>>> print(dmet.score_kwcoco(bias=0, compat='mutex', prioritize='iou')['mAP'])
...
>>> # NOTE: IN GENERAL NETHARN AND VOC ARE NOT THE SAME
>>> print(dmet.score_voc(bias=0)['mAP'])
0.8582...
>>> #print(dmet.score_coco()['mAP'])
```

clear()

classmethod from_coco(*true_coco*, *pred_coco*, *gids=None*, *verbose=0*)

Create detection metrics from two coco files representing the truth and predictions.

Parameters

- **true_coco** (*kwcoco.CocoDataset*)
- **pred_coco** (*kwcoco.CocoDataset*)

Example

```
>>> import kwcoco
>>> true_coco = kwcoco.CocoDataset.demo('shapes')
>>> pred_coco = true_coco
>>> self = DetectionMetrics.from_coco(true_coco, pred_coco)
>>> self.score_voc()
```

add_predictions(*pred_dets*, *imgname=None*, *gid=None*)
Register/Add predicted detections for an image

Parameters

- **pred_dets** (*Detections*) – predicted detections
- **imgname** (*str*) – a unique string to identify the image
- **gid** (*int, optional*) – the integer image id if known

add_truth(*true_dets*, *imgname=None*, *gid=None*)
Register/Add groundtruth detections for an image

Parameters

- **true_dets** (*Detections*) – groundtruth
- **imgname** (*str*) – a unique string to identify the image
- **gid** (*int, optional*) – the integer image id if known

true_detections(*gid*)
gets Detections representation for groundtruth in an image

pred_detections(*gid*)
gets Detections representation for predictions in an image

confusion_vectors(*ovthresh=0.5*, *bias=0*, *gids=None*, *compat='all'*, *prioritize='iou'*, *ignore_classes='ignore'*, *background_class=NoParam*, *verbose='auto'*, *workers=0*, *track_probs='try'*)
Assigns predicted boxes to the true boxes so we can transform the detection problem into a classification problem for scoring.

Parameters

- **ovthresh** (*float, default=0.5*) – bounding box overlap iou threshold required for assignment
- **bias** (*float, default=0.0*) – for computing bounding box overlap, either 1 or 0
- **gids** (*List[int], default=None*) – which subset of images ids to compute confusion metrics on. If not specified all images are used.
- **compat** (*str, default='all'*) – can be ('ancestors' | 'mutex' | 'all'). determines which pred boxes are allowed to match which true boxes. If 'mutex', then pred boxes can only match true boxes of the same class. If 'ancestors', then pred boxes can match true boxes that match or have a coarser label. If 'all', then any pred can match any true, regardless of its category label.
- **prioritize** (*str, default='iou'*) – can be ('iou' | 'class' | 'correct') determines which box to assign to if mutiple true boxes overlap a predicted box. if prioritize is iou, then the true box with maximum iou (above ovthresh) will be chosen. If prioritize is class, then it will prefer matching a compatible class above a higher iou. If prioritize is correct, then ancestors of the true class are preferred over descendents of the true class, over unreleated classes.
- **ignore_classes** (*set, default={'ignore'}*) – class names indicating ignore regions
- **background_class** (*str, default=ub.NoParam*) – Name of the background class. If unspecified we try to determine it with heuristics. A value of None means there is no background class.
- **verbose** (*int, default='auto'*) – verbosity flag. In auto mode, verbose=1 if len(gids) > 1000.

- **workers** (*int, default=0*) – number of parallel assignment processes
- **track_probs** (*str, default='try'*) – can be 'try', 'force', or False. if truthy, we assume probabilities for multiple classes are available.

Ignore: globals().update(xdev.get_func_kwargs(dmet.confusion_vectors))

score_kwant (*ovthresh=0.5*)
Scores the detections using kwant

score_kwcoco (*ovthresh=0.5*, *bias=0*, *gids=None*, *compat='all'*, *prioritize='iou'*)
our scoring method

score_voc (*ovthresh=0.5*, *bias=1*, *method='voc2012'*, *gids=None*, *ignore_classes='ignore'*)
score using voc method

Example

```
>>> dmet = DetectionMetrics.demo(
>>>     nimgs=100, nboxes=(0, 3), n_fp=(0, 1), nclasses=8,
>>>     score_noise=.5)
>>> print(dmet.score_voc()['mAP'])
0.9399...
```

score_coco (*verbose=0*)
score using ms-coco method

Example

```
>>> # xdoctest: +REQUIRES(--pycocotools)
>>> dmet = DetectionMetrics.demo(
>>>     nimgs=100, nboxes=(0, 3), n_fp=(0, 1), nclasses=8)
>>> print(dmet.score_coco()['mAP'])
0.711016...
```

classmethod demo (***kwargs*)
Creates random true boxes and predicted boxes that have some noisy offset from the truth.

Kwargs: nclasses (int, default=1): number of foreground classes. nimgs (int, default=1): number of images in the coco datasts. nboxes (int, default=1): boxes per image. n_fp (int, default=0): number of false positives. n_fn (int, default=0): number of false negatives. box_noise (float, default=0): std of a normal distribution used to

perterb both box location and box size.

cls_noise (float, default=0): probability that a class label will change. Must be within 0 and 1.

anchors (ndarray, default=None): used to create random boxes null_pred (bool, default=0):

if True, predicted classes are returned as null, which means only localization scoring is suitable.

with_probs (bool, default=1): if True, includes per-class probabilities with predictions

Example

```
>>> kwargs = {}
>>> # Seed the RNG
>>> kwargs['rng'] = 0
>>> # Size parameters determine how big the data is
>>> kwargs['nimgs'] = 5
>>> kwargs['nboxes'] = 7
>>> kwargs['nclasses'] = 11
>>> # Noise parameters perterb predictions further from the truth
>>> kwargs['n_fp'] = 3
>>> kwargs['box_noise'] = 0.1
>>> kwargs['cls_noise'] = 0.5
>>> dmet = DetectionMetrics.demo(**kwargs)
>>> print('dmet.classes = {}'.format(dmet.classes))
dmet.classes = <CategoryTree(nNodes=12, maxDepth=3, maxBreadth=4...)>
>>> # Can grab kwimage.Detection object for any image
>>> print(dmet.true_detections(gid=0))
<Detections(4)>
>>> print(dmet.pred_detections(gid=0))
<Detections(7)>
```

Example

```
>>> # Test case with null predicted categories
>>> dmet = DetectionMetrics.demo(nimgs=30, null_pred=1, nclasses=3,
>>>                              nboxes=10, n_fp=10, box_noise=0.3,
>>>                              with_probs=False)
>>> dmet.gid_to_pred_dets[0].data
>>> dmet.gid_to_true_dets[0].data
>>> cfsn_vecs = dmet.confusion_vectors()
>>> binvecs_ovr = cfsn_vecs.binarize_ovr()
>>> binvecs_per = cfsn_vecs.binarize_peritem()
>>> measures_per = binvecs_per.measures()
>>> measures_ovr = binvecs_ovr.measures()
>>> print('measures_per = {!r}'.format(measures_per))
>>> print('measures_ovr = {!r}'.format(measures_ovr))
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> pr_per.draw(fnum=1)
>>> measures_ovr['perclass'].draw(key='pr', fnum=2)
```

summarize(*out_dpath=None*, *plot=False*, *title=''*)

Example

```
>>> from kwcoco.metrics.confusion_vectors import *  # NOQA
>>> from kwcoco.metrics.detect_metrics import DetectionMetrics
>>> dmet = DetectionMetrics.demo(
>>>     n_fp=(0, 128), n_fn=(0, 4), nimgs=512, nboxes=(0, 32),
>>>     nclasses=3, rng=0)
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
```

(continues on next page)

(continued from previous page)

```
>>> kwplot.autompl()
>>> dmet.summarize(plot=True, title='DetectionMetrics summary demo')
>>> kwplot.show_if_requested()
```

kwcoco.metrics.detect_metrics.**eval_detections_cli**(***kw*)

CommandLine: xdoctest -m ~/code/kwcoco/kwcoco/metrics/detect_metrics.py eval_detections_cli

kwcoco.metrics.drawing module

kwcoco.metrics.drawing.**draw_roc**(*roc_info*, *prefix=''*, *fnum=1*, ***kw*)

NOTE: There needs to be enough negative examples for using ROC to make any sense!

Example

```
>>> # xdoctest: +REQUIRES(module:kwplot)
>>> from kwcoco.metrics import DetectionMetrics
>>> dmet = DetectionMetrics.demo(
>>>     nimgs=100, nboxes=(0, 30), n_fp=(0, 1), nclasses=3,
>>>     box_noise=0.00, cls_noise=.0, score_noise=1.0)
>>> dmet.true_detections(0).data
>>> cfsn_vecs = dmet.confusion_vectors(compat='mutex', prioritize='iou', bias=0)
>>> print(cfsn_vecs.data._pandas().sort_values('score'))
>>> classes = cfsn_vecs.classes
>>> roc_info = ub.peek(cfsn_vecs.binarize_ovr().measures()['perclass'].values())
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> draw_roc(roc_info)
>>> kwplot.show_if_requested()
```

kwcoco.metrics.drawing.**draw_perclass_roc**(*cx_to_rocinfo*, *classes=None*, *prefix=''*, *fnum=1*, *fp_axis='count'*, ***kw*)

fp_axis can be count or rate

cx_to_rocinfo = roc_perclass

kwcoco.metrics.drawing.**draw_perclass_prcurve**(*cx_to_peritem*, *classes=None*, *prefix=''*, *fnum=1*, ***kw*)

Example

```
>>> # xdoctest: +REQUIRES(module:kwplot)
>>> from kwcoco.metrics import DetectionMetrics
>>> dmet = DetectionMetrics.demo(
>>>     nimgs=10, nboxes=(0, 10), n_fp=(0, 1), nclasses=3)
>>> cfsn_vecs = dmet.confusion_vectors()
>>> classes = cfsn_vecs.classes
>>> cx_to_peritem = cfsn_vecs.binarize_ovr().measures()['perclass']
>>> import kwplot
>>> kwplot.autompl()
>>> draw_perclass_prcurve(cx_to_peritem, classes)
>>> # xdoctest: +REQUIRES(--show)
>>> kwplot.show_if_requested()
```

kwcoco.metrics.drawing.**draw_perclass_thresholds**(*cx_to_peritem*, *key='mcc'*, *classes=None*, *prefix=''*, *fnum=1*, ***kw*)

Notes

Each category is inspected independently of one another, there is no notion of confusion.

Example

```
>>> # xdoctest: +REQUIRES(module:kwplot)
>>> from kwcoco.metrics.drawing import *  # NOQA
>>> from kwcoco.metrics import ConfusionVectors
>>> cfsn_vecs = ConfusionVectors.demo()
>>> classes = cfsn_vecs.classes
>>> ovr_cfsn = cfsn_vecs.binarize_ovr(keyby='name')
>>> cx_to_peritem = ovr_cfsn.measures()['perclass']
>>> import kwplot
>>> kwplot.autompl()
>>> key = 'mcc'
>>> draw_perclass_thresholds(cx_to_peritem, key, classes)
>>> # xdoctest: +REQUIRES(--show)
>>> kwplot.show_if_requested()
```

kwcoco.metrics.drawing.**draw_prcurve**(*peritem*, *prefix=''*, *fnum=1*, ***kw*)

TODO: rename to draw prcurve. Just draws a single pr curve.

Example

```
>>> # xdoctest: +REQUIRES(module:kwplot)
>>> from kwcoco.metrics import DetectionMetrics
>>> dmet = DetectionMetrics.demo(
>>>     nimgs=10, nboxes=(0, 10), n_fp=(0, 1), nclasses=3)
>>> cfsn_vecs = dmet.confusion_vectors()
```

```
>>> classes = cfsn_vecs.classes
>>> peritem = cfsn_vecs.binarize_peritem().measures()
>>> import kwplot
>>> kwplot.autompl()
>>> draw_prcurve(peritem)
>>> # xdoctest: +REQUIRES(--show)
>>> kwplot.show_if_requested()
```

kwcoco.metrics.drawing.**draw_threshold_curves**(*info*, *keys=None*, *prefix=''*, *fnum=1*, ***kw*)

Example

```
>>> # xdoctest: +REQUIRES(module:kwplot)
>>> import sys, ubelt
>>> sys.path.append(ubelt.expandpath('~/code/kwcoco'))
>>> from kwcoco.metrics.drawing import *  # NOQA
>>> from kwcoco.metrics import DetectionMetrics
```

(continues on next page)

(continued from previous page)

```
>>> dmet = DetectionMetrics.demo(
>>>     nimgs=10, nboxes=(0, 10), n_fp=(0, 1), nclasses=3)
>>> cfsn_vecs = dmet.confusion_vectors()
>>> info = cfsn_vecs.binarize_peritem().measures()
>>> keys = None
>>> import kwplot
>>> kwplot.autompl()
>>> draw_threshold_curves(info, keys)
>>> # xdoctest: +REQUIRES(--show)
>>> kwplot.show_if_requested()
```

kwcoco.metrics.functional module

kwcoco.metrics.functional.**fast_confusion_matrix**(*y_true*, *y_pred*, *n_labels*, *sample_weight=None*)

faster version of sklearn confusion matrix that avoids the expensive checks and label rectification

Parameters

- **y_true** (*ndarray[int]*) – ground truth class label for each sample
- **y_pred** (*ndarray[int]*) – predicted class label for each sample
- **n_labels** (*int*) – number of labels
- **sample_weight** (*ndarray[int|float]*) – weight of each sample

Returns matrix where rows represent real and cols represent pred and the value at each cell is the total amount of weight

Return type ndarray[int64|float64, dim=2]

Example

```
>>> y_true = np.array([0, 0, 0, 0, 1, 1, 1, 0,  0, 1])
>>> y_pred = np.array([0, 0, 0, 0, 0, 0, 0, 1,  1, 1])
>>> fast_confusion_matrix(y_true, y_pred, 2)
array([[4, 2],
       [3, 1]])
>>> fast_confusion_matrix(y_true, y_pred, 2).ravel()
array([4, 2, 3, 1])
```

kwcoco.metrics.sklearn_alts module

Faster pure-python versions of sklearn functions that avoid expensive checks and label rectifications. It is assumed that all labels are consecutive non-negative integers.

kwcoco.metrics.sklearn_alts.**confusion_matrix**(*y_true*, *y_pred*, *n_labels=None*, *labels=None*, *sample_weight=None*)

faster version of sklearn confusion matrix that avoids the expensive checks and label rectification

Runs in about 0.7ms

Returns matrix where rows represent real and cols represent pred

Return type ndarray

Example

```
>>> y_true = np.array([0, 0, 0, 0, 1, 1, 1, 0,  0, 1])
>>> y_pred = np.array([0, 0, 0, 0, 0, 0, 0, 1,  1, 1])
>>> confusion_matrix(y_true, y_pred, 2)
array([[4, 2],
       [3, 1]])
>>> confusion_matrix(y_true, y_pred, 2).ravel()
array([4, 2, 3, 1])
```

Benchmarks: import ubelt as ub y_true = np.random.randint(0, 2, 10000) y_pred = np.random.randint(0, 2, 10000)

n = 1000 for timer in ub.Timerit(n, bestof=10, label='py-time'):

sample_weight = [1] * len(y_true) confusion_matrix(y_true, y_pred, 2, sample_weight=sample_weight)

for timer in ub.Timerit(n, bestof=10, label='np-time'): sample_weight = np.ones(len(y_true), dtype=np.int) confusion_matrix(y_true, y_pred, 2, sample_weight=sample_weight)

kwcoco.metrics.sklearn_alts.**global_accuracy_from_confusion**(*cfsn*)

kwcoco.metrics.sklearn_alts.**class_accuracy_from_confusion**(*cfsn*)

kwcoco.metrics.util module

class kwcoco.metrics.util.**DictProxy**

Bases: scriptconfig.dict_like.DictLike

Allows an object to proxy the behavior of a dict attribute

keys()

kwcoco.metrics.voc_metrics module

class kwcoco.metrics.voc_metrics.**VOC_Metrics**(*classes=None*)

Bases: ubelt.util_mixins.NiceRepr

API to compute object detection scores using Pascal VOC evaluation method.

To use, add true and predicted detections for each image and then run the *score* function.

Variables

- **recs** (*Dict[int, List[dict]]*) – true boxes for each image. maps image ids to a list of records within that image. Each record is a tlbr bbox, a difficult flag, and a class name.
- **cx_to_lines** (*Dict[int, List]*) – VOC formatted prediction preditions. mapping from class index to all predictions for that category. Each "line" is a list of [

[<imgid>, <score>, <tl_x>, <tl_y>, <br_x>, <br_y>]].

add_truth(*true_dets*, *gid*)

add_predictions(*pred_dets*, *gid*)

score(*ovthresh=0.5*, *bias=1*, *method='voc2012'*)

Compute VOC scores for every category

Example

```
>>> from kwcoco.metrics.detect_metrics import DetectionMetrics
>>> from kwcoco.metrics.voc_metrics import *  # NOQA
>>> dmet = DetectionMetrics.demo(
>>>     nimgs=1, nboxes=(0, 100), n_fp=(0, 30), n_fn=(0, 30), nclasses=2,␣
↪score_noise=0.9)
>>> self = VOC_Metrics(classes=dmet.classes)
>>> self.add_truth(dmet.true_detections(0), 0)
>>> self.add_predictions(dmet.pred_detections(0), 0)
>>> voc_scores = self.score()
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.figure(fnum=1, doclf=True)
>>> voc_scores['perclass'].draw()
```

kwplot.figure(fnum=2) dmet.true_detections(0).draw(color='green', labels=None) dmet.pred_detections(0).draw(color='blue', labels=None) kwplot.autoplt().gca().set_xlim(0, 100) kwplot.autoplt().gca().set_ylim(0, 100)

Module contents

mkinit kwcoco.metrics -w –relative

class kwcoco.metrics.**BinaryConfusionVectors**(*data*, *cx=None*, *classes=None*)

Bases: ubelt.util_mixins.NiceRepr

Stores information about a binary classification problem. This is always with respect to a specific class, which is given by *cx* and *classes*.

The *data* DataFrameArray must contain *is_true* - if the row is an instance of class *classes[cx] pred_score* - the predicted probability of class *classes[cx]*, and *weight* - sample weight of the example

Example

```
>>> self = BinaryConfusionVectors.demo(n=10)
>>> print('self = {!r}'.format(self))
>>> print('pr = {}'.format(ub.repr2(self.measures())))
>>> print('roc = {}'.format(ub.repr2(self.roc())))
```

```
>>> self = BinaryConfusionVectors.demo(n=0)
>>> print('pr = {}'.format(ub.repr2(self.measures())))
>>> print('roc = {}'.format(ub.repr2(self.roc())))
```

```
>>> self = BinaryConfusionVectors.demo(n=1)
>>> print('pr = {}'.format(ub.repr2(self.measures())))
>>> print('roc = {}'.format(ub.repr2(self.roc())))
```

```
>>> self = BinaryConfusionVectors.demo(n=2)
>>> print('self = {!r}'.format(self))
>>> print('pr = {}'.format(ub.repr2(self.measures())))
>>> print('roc = {}'.format(ub.repr2(self.roc())))
```

classmethod demo(*n=10*, *p_true=0.5*, *p_error=0.2*, *rng=None*)
Create random data for tests

Example

```
>>> from kwcoco.metrics.confusion_vectors import *  # NOQA
>>> cfsn = BinaryConfusionVectors.demo(n=1000, p_error=0.1)
>>> measures = cfsn.measures()
>>> print('measures = {}'.format(ub.repr2(measures, nl=1)))
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.figure(fnum=1, pnum=(1, 2, 1))
>>> measures.draw('pr')
>>> kwplot.figure(fnum=1, pnum=(1, 2, 2))
>>> measures.draw('roc')
```

catname

draw_distribution()

precision_recall(*stabalize_thresh=7*, *stabalize_pad=7*, *method='sklearn'*)
Deprecated, all information lives in measures now

roc(*fp_cutoff=None*, *stabalize_thresh=7*, *stabalize_pad=7*)
Deprecated, all information lives in measures now

measures
memoization decorator for a method that respects args and kwargs

References

http://code.activestate.com/recipes/577452-a-memoize-decorator-for-instance-methods/

Example

```
>>> import ubelt as ub
>>> closure = {'a': 'b', 'c': 'd'}
>>> incr = [0]
>>> class Foo(object):
>>>     @memoize_method
>>>     def foo_memo(self, key):
>>>         value = closure[key]
>>>         incr[0] += 1
>>>         return value
>>>     def foo(self, key):
>>>         value = closure[key]
>>>         incr[0] += 1
>>>         return value
>>> self = Foo()
>>> assert self.foo('a') == 'b' and self.foo('c') == 'd'
>>> assert incr[0] == 2
>>> print('Call memoized version')
>>> assert self.foo_memo('a') == 'b' and self.foo_memo('c') == 'd'
>>> assert incr[0] == 4
```

(continues on next page)

(continued from previous page)

```
>>> assert self.foo_memo('a') == 'b' and self.foo_memo('c') == 'd'
>>> print('Counter should no longer increase')
>>> assert incr[0] == 4
>>> print('Closure changes result without memoization')
>>> closure = {'a': 0, 'c': 1}
>>> assert self.foo('a') == 0 and self.foo('c') == 1
>>> assert incr[0] == 6
>>> assert self.foo_memo('a') == 'b' and self.foo_memo('c') == 'd'
>>> print('Constructing a new object should get a new cache')
>>> self2 = Foo()
>>> self2.foo_memo('a')
>>> assert incr[0] == 7
>>> self2.foo_memo('a')
>>> assert incr[0] == 7
```

class kwcoco.metrics.**ConfusionVectors**(*data*, *classes*, *probs=None*)

Bases: ubelt.util_mixins.NiceRepr

Stores information used to construct a confusion matrix. This includes corresponding vectors of predicted labels, true labels, sample weights, etc...

Variables

- **data** (*DataFrameArray*) – should at least have keys true, pred, weight
- **classes** (*Sequence | CategoryTree*) – list of category names or category graph
- **probs** (*ndarray, optional*) – probabilities for each class

Example

```
>>> # xdoctest: IGNORE_WANT
>>> from kwcoco.metrics import DetectionMetrics
>>> dmet = DetectionMetrics.demo(
>>>     nimgs=10, nboxes=(0, 10), n_fp=(0, 1), nclasses=3)
>>> cfsn_vecs = dmet.confusion_vectors()
>>> print(cfsn_vecs.data._pandas())
     pred  true   score  weight     iou  txs  pxs  gid
0       2     2 10.0000  1.0000  1.0000    0    4    0
1       2     2  7.5025  1.0000  1.0000    1    3    0
2       1     1  5.0050  1.0000  1.0000    2    2    0
3       3    -1  2.5075  1.0000 -1.0000   -1    1    0
4       2    -1  0.0100  1.0000 -1.0000   -1    0    0
5      -1     2  0.0000  1.0000 -1.0000    3   -1    0
6      -1     2  0.0000  1.0000 -1.0000    4   -1    0
7       2     2 10.0000  1.0000  1.0000    0    5    1
8       2     2  8.0020  1.0000  1.0000    1    4    1
9       1     1  6.0040  1.0000  1.0000    2    3    1
..    ...   ...     ...     ...     ...  ...  ...  ...
62     -1     2  0.0000  1.0000 -1.0000    7   -1    7
63     -1     3  0.0000  1.0000 -1.0000    8   -1    7
64     -1     1  0.0000  1.0000 -1.0000    9   -1    7
65      1    -1 10.0000  1.0000 -1.0000   -1    0    8
66      1     1  0.0100  1.0000  1.0000    0    1    8
67      3    -1 10.0000  1.0000 -1.0000   -1    3    9
68      2     2  6.6700  1.0000  1.0000    0    2    9
69      2     2  3.3400  1.0000  1.0000    1    1    9
```

(continues on next page)

(continued from previous page)

```
70      3    -1  0.0100  1.0000 -1.0000   -1    0    9
71     -1     2  0.0000  1.0000 -1.0000    2   -1    9
```

```
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> from kwcoco.metrics.confusion_vectors import ConfusionVectors
>>> cfsn_vecs = ConfusionVectors.demo(
>>>     nimgs=128, nboxes=(0, 10), n_fp=(0, 3), n_fn=(0, 3), nclasses=3)
>>> cx_to_binvecs = cfsn_vecs.binarize_ovr()
>>> measures = cx_to_binvecs.measures()['perclass']
>>> print('measures = {!r}'.format(measures))
measures = <PerClass_Measures({
    'cat_1': <Measures({'ap': 0.7501, 'auc': 0.7170, 'catname': cat_1, 'max_f1':␣
↪f1=0.77@0.41, 'max_mcc': mcc=0.71@0.44, 'nsupport': 787.0000, 'realneg_total':␣
↪594.0000, 'realpos_total': 193.0000})>,
    'cat_2': <Measures({'ap': 0.8288, 'auc': 0.8137, 'catname': cat_2, 'max_f1':␣
↪f1=0.83@0.40, 'max_mcc': mcc=0.78@0.40, 'nsupport': 787.0000, 'realneg_total':␣
↪589.0000, 'realpos_total': 198.0000})>,
    'cat_3': <Measures({'ap': 0.7536, 'auc': 0.7150, 'catname': cat_3, 'max_f1':␣
↪f1=0.77@0.40, 'max_mcc': mcc=0.71@0.42, 'nsupport': 787.0000, 'realneg_total':␣
↪578.0000, 'realpos_total': 209.0000})>,
}) at 0x7f1b9b0d6130>
```

```
>>> kwplot.figure(fnum=1, doclf=True)
>>> measures.draw(key='pr', fnum=1, pnum=(1, 3, 1))
>>> measures.draw(key='roc', fnum=1, pnum=(1, 3, 2))
>>> measures.draw(key='mcc', fnum=1, pnum=(1, 3, 3))
...
```

classmethod from_json(*state*)

classmethod demo(***kw*)

Example

```
>>> cfsn_vecs = ConfusionVectors.demo()
>>> print('cfsn_vecs = {!r}'.format(cfsn_vecs))
>>> cx_to_binvecs = cfsn_vecs.binarize_ovr()
>>> print('cx_to_binvecs = {!r}'.format(cx_to_binvecs))
```

classmethod from_arrays(*true*, *pred=None*, *score=None*, *weight=None*, *probs=None*, *classes=None*)

Construct confusion vector data structure from component arrays

Example

```
>>> import kwarray
>>> classes = ['person', 'vehicle', 'object']
>>> rng = kwarray.ensure_rng(0)
>>> true = (rng.rand(10) * len(classes)).astype(np.int)
>>> probs = rng.rand(len(true), len(classes))
>>> cfsn_vecs = ConfusionVectors.from_arrays(true=true, probs=probs,␣
↪classes=classes)
```

(continues on next page)

(continued from previous page)

```
>>> cfsn_vecs.confusion_matrix()
pred     person  vehicle  object
real
person        0        0       0
vehicle       2        4       1
object        2        1       0
```

confusion_matrix(*raw=False*, *compress=False*)

Builds a confusion matrix from the confusion vectors.

Parameters **raw** (*bool*) – if True uses 'pred_raw' otherwise used 'pred'

Returns

cm [the labeled confusion matrix]

(Note: we should write a efficient replacement for this use case. #remove_pandas)

Return type pd.DataFrame

CommandLine: xdoctest -m ~/code/kwcoco/kwcoco/metrics/confusion_vectors.py ConfusionVectors.confusion_matrix

Example

```
>>> from kwcoco.metrics import DetectionMetrics
>>> dmet = DetectionMetrics.demo(
>>>     nimgs=10, nboxes=(0, 10), n_fp=(0, 1), n_fn=(0, 1), nclasses=3, cls_
↪noise=.2)
>>> cfsn_vecs = dmet.confusion_vectors()
>>> cm = cfsn_vecs.confusion_matrix()
...
>>> print(cm.to_string(float_format=lambda x: '%.2f' % x))
pred        background  cat_1  cat_2  cat_3
real
background        0.00   1.00   1.00   1.00
cat_1             2.00  12.00   0.00   1.00
cat_2             2.00   0.00  14.00   1.00
cat_3             1.00   0.00   1.00  17.00
```

coarsen(*cxs*)

Creates a coarsened set of vectors

binarize_peritem(*negative_classes=None*)

Creates a binary representation useful for measuring the performance of detectors. It is assumed that scores of "positive" classes should be high and "negative" clases should be low.

Parameters **negative_classes** (*List[str | int]*) – list of negative class names or idxs, by default chooses any class with a true class index of -1. These classes should ideally have low scores.

Returns BinaryConfusionVectors

Example

```
>>> from kwcoco.metrics import DetectionMetrics
>>> dmet = DetectionMetrics.demo(
>>>     nimgs=10, nboxes=(0, 10), n_fp=(0, 1), nclasses=3)
>>> cfsn_vecs = dmet.confusion_vectors()
>>> class_idxs = list(dmet.classes.node_to_idx.values())
>>> binvecs = cfsn_vecs.binarize_peritem()
```

binarize_ovr (*mode=1*, *keyby='name'*, *ignore_classes={'ignore'}*)

Transforms cfsn_vecs into one-vs-rest BinaryConfusionVectors for each category.

Parameters

- **mode** (*int, default=1*) – 0 for heirarchy aware or 1 for voc like. MODE 0 IS PROBABLY BROKEN
- **keyby** (*int | str*) – can be cx or name
- **ignore_classes** (*Set[str]*) – category names to ignore

Returns

which behaves like Dict[int, BinaryConfusionVectors]: cx_to_binvecs

Return type *OneVsRestConfusionVectors*

Example

```
>>> cfsn_vecs = ConfusionVectors.demo()
>>> print('cfsn_vecs = {!r}'.format(cfsn_vecs))
>>> catname_to_binvecs = cfsn_vecs.binarize_ovr(keyby='name')
>>> print('catname_to_binvecs = {!r}'.format(catname_to_binvecs))
```

Notes

Consider we want to measure how well we can classify beagles.

Given a multiclass confusion vector, we need to carefully select a subset. We ignore any truth that is coarser than our current label. We also ignore any background predictions on irrelevant classes

dog | dog <- ignore coarser truths dog | cat <- ignore coarser truths dog | beagle <- ignore coarser truths cat | dog cat | cat cat | background <- ignore failures to predict unrelated classes cat | maine-coon beagle | beagle beagle | dog beagle | background beagle | cat Snoopy | beagle Snoopy | cat maine-coon | background <- ignore failures to predict unrelated classes maine-coon | beagle maine-coon | cat

Anything not marked as ignore is counted. We count anything marked as beagle or a finer grained class (e.g. Snoopy) as a positive case. All other cases are negative. The scores come from the predicted probability of beagle, which must be remembered outside the dataframe.

classification_report (*verbose=0*)

Build a classification report with various metrics.

Example

```
>>> from kwcoco.metrics.confusion_vectors import *  # NOQA
>>> cfsn_vecs = ConfusionVectors.demo()
>>> report = cfsn_vecs.classification_report(verbose=1)
```

class kwcoco.metrics.**DetectionMetrics**(*classes=None*)

Bases: ubelt.util_mixins.NiceRepr

Variables

- **gid_to_true_dets** (*Dict*) – maps image ids to truth
- **gid_to_pred_dets** (*Dict*) – maps image ids to predictions
- **classes** (CategoryTree) – category coder

Example

```
>>> dmet = DetectionMetrics.demo(
>>>     nimgs=100, nboxes=(0, 3), n_fp=(0, 1), nclasses=8, score_noise=0.9,␣
↪hacked=False)
>>> print(dmet.score_kwcoco(bias=0, compat='mutex', prioritize='iou')['mAP'])
...
>>> # NOTE: IN GENERAL NETHARN AND VOC ARE NOT THE SAME
>>> print(dmet.score_voc(bias=0)['mAP'])
0.8582...
>>> #print(dmet.score_coco()['mAP'])
```

clear()

classmethod from_coco(*true_coco*, *pred_coco*, *gids=None*, *verbose=0*)

Create detection metrics from two coco files representing the truth and predictions.

Parameters

- **true_coco** (*kwcoco.CocoDataset*)
- **pred_coco** (*kwcoco.CocoDataset*)

Example

```
>>> import kwcoco
>>> true_coco = kwcoco.CocoDataset.demo('shapes')
>>> pred_coco = true_coco
>>> self = DetectionMetrics.from_coco(true_coco, pred_coco)
>>> self.score_voc()
```

add_predictions(*pred_dets*, *imgname=None*, *gid=None*)

Register/Add predicted detections for an image

Parameters

- **pred_dets** (*Detections*) – predicted detections
- **imgname** (*str*) – a unique string to identify the image
- **gid** (*int, optional*) – the integer image id if known

add_truth(*true_dets*, *imgname=None*, *gid=None*)

Register/Add groundtruth detections for an image

Parameters

- **true_dets** (*Detections*) – groundtruth
- **imgname** (*str*) – a unique string to identify the image

- **gid** (*int, optional*) – the integer image id if known

true_detections(*gid*)
gets Detections representation for groundtruth in an image

pred_detections(*gid*)
gets Detections representation for predictions in an image

confusion_vectors(*ovthresh=0.5*, *bias=0*, *gids=None*, *compat='all'*, *prioritize='iou'*, *ignore_classes='ignore'*, *background_class=NoParam*, *verbose='auto'*, *workers=0*, *track_probs='try'*)
Assigns predicted boxes to the true boxes so we can transform the detection problem into a classification problem for scoring.

Parameters

- **ovthresh** (*float, default=0.5*) – bounding box overlap iou threshold required for assignment
- **bias** (*float, default=0.0*) – for computing bounding box overlap, either 1 or 0
- **gids** (*List[int], default=None*) – which subset of images ids to compute confusion metrics on. If not specified all images are used.
- **compat** (*str, default='all'*) – can be ('ancestors' | 'mutex' | 'all'). determines which pred boxes are allowed to match which true boxes. If 'mutex', then pred boxes can only match true boxes of the same class. If 'ancestors', then pred boxes can match true boxes that match or have a coarser label. If 'all', then any pred can match any true, regardless of its category label.
- **prioritize** (*str, default='iou'*) – can be ('iou' | 'class' | 'correct') determines which box to assign to if mutiple true boxes overlap a predicted box. if prioritize is iou, then the true box with maximum iou (above ovthresh) will be chosen. If prioritize is class, then it will prefer matching a compatible class above a higher iou. If prioritize is correct, then ancestors of the true class are preferred over descendents of the true class, over unreleated classes.
- **ignore_classes** (*set, default={'ignore'}*) – class names indicating ignore regions
- **background_class** (*str, default=ub.NoParam*) – Name of the background class. If unspecified we try to determine it with heuristics. A value of None means there is no background class.
- **verbose** (*int, default='auto'*) – verbosity flag. In auto mode, verbose=1 if len(gids) > 1000.
- **workers** (*int, default=0*) – number of parallel assignment processes
- **track_probs** (*str, default='try'*) – can be 'try', 'force', or False. if truthy, we assume probabilities for multiple classes are available.

Ignore: globals().update(xdev.get_func_kwargs(dmet.confusion_vectors))

score_kwant(*ovthresh=0.5*)
Scores the detections using kwant

score_kwcoco(*ovthresh=0.5*, *bias=0*, *gids=None*, *compat='all'*, *prioritize='iou'*)
our scoring method

score_voc(*ovthresh=0.5*, *bias=1*, *method='voc2012'*, *gids=None*, *ignore_classes='ignore'*)
score using voc method

Example

```
>>> dmet = DetectionMetrics.demo(
>>>     nimgs=100, nboxes=(0, 3), n_fp=(0, 1), nclasses=8,
>>>     score_noise=.5)
>>> print(dmet.score_voc()['mAP'])
0.9399...
```

score_coco(*verbose=0*)
score using ms-coco method

Example

```
>>> # xdoctest: +REQUIRES(--pycocotools)
>>> dmet = DetectionMetrics.demo(
>>>     nimgs=100, nboxes=(0, 3), n_fp=(0, 1), nclasses=8)
>>> print(dmet.score_coco()['mAP'])
0.711016...
```

classmethod demo(***kwargs*)
Creates random true boxes and predicted boxes that have some noisy offset from the truth.

Kwargs: nclasses (int, default=1): number of foreground classes. nimgs (int, default=1): number of images in the coco datasts. nboxes (int, default=1): boxes per image. n_fp (int, default=0): number of false positives. n_fn (int, default=0): number of false negatives. box_noise (float, default=0): std of a normal distribution used to

perterb both box location and box size.

cls_noise (float, default=0): probability that a class label will change. Must be within 0 and 1.

anchors (ndarray, default=None): used to create random boxes null_pred (bool, default=0):

if True, predicted classes are returned as null, which means only localization scoring is suitable.

with_probs (bool, default=1): if True, includes per-class probabilities with predictions

Example

```
>>> kwargs = {}
>>> # Seed the RNG
>>> kwargs['rng'] = 0
>>> # Size parameters determine how big the data is
>>> kwargs['nimgs'] = 5
>>> kwargs['nboxes'] = 7
>>> kwargs['nclasses'] = 11
>>> # Noise parameters perterb predictions further from the truth
>>> kwargs['n_fp'] = 3
>>> kwargs['box_noise'] = 0.1
>>> kwargs['cls_noise'] = 0.5
>>> dmet = DetectionMetrics.demo(**kwargs)
>>> print('dmet.classes = {}'.format(dmet.classes))
dmet.classes = <CategoryTree(nNodes=12, maxDepth=3, maxBreadth=4...)>
```

(continues on next page)

(continued from previous page)

```
>>> # Can grab kwimage.Detection object for any image
>>> print(dmet.true_detections(gid=0))
<Detections(4)>
>>> print(dmet.pred_detections(gid=0))
<Detections(7)>
```

Example

```
>>> # Test case with null predicted categories
>>> dmet = DetectionMetrics.demo(nimgs=30, null_pred=1, nclasses=3,
>>>                              nboxes=10, n_fp=10, box_noise=0.3,
>>>                              with_probs=False)
>>> dmet.gid_to_pred_dets[0].data
>>> dmet.gid_to_true_dets[0].data
>>> cfsn_vecs = dmet.confusion_vectors()
>>> binvecs_ovr = cfsn_vecs.binarize_ovr()
>>> binvecs_per = cfsn_vecs.binarize_peritem()
>>> measures_per = binvecs_per.measures()
>>> measures_ovr = binvecs_ovr.measures()
>>> print('measures_per = {!r}'.format(measures_per))
>>> print('measures_ovr = {!r}'.format(measures_ovr))
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> pr_per.draw(fnum=1)
>>> measures_ovr['perclass'].draw(key='pr', fnum=2)
```

summarize(*out_dpath=None*, *plot=False*, *title=''*)

Example

```
>>> from kwcoco.metrics.confusion_vectors import *  # NOQA
>>> from kwcoco.metrics.detect_metrics import DetectionMetrics
>>> dmet = DetectionMetrics.demo(
>>>     n_fp=(0, 128), n_fn=(0, 4), nimgs=512, nboxes=(0, 32),
>>>     nclasses=3, rng=0)
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> dmet.summarize(plot=True, title='DetectionMetrics summary demo')
>>> kwplot.show_if_requested()
```

class kwcoco.metrics.**Measures**(*roc_info*)

Bases: ubelt.util_mixins.NiceRepr, *kwcoco.metrics.util.DictProxy*

Example

```
>>> from kwcoco.metrics.confusion_vectors import *  # NOQA
>>> binvecs = BinaryConfusionVectors.demo(n=100, p_error=0.5)
>>> self = binvecs.measures()
>>> print('self = {!r}'.format(self))
>>> # xdoctest: +REQUIRES(--show)
```

(continues on next page)

(continued from previous page)

```
>>> import kwplot
>>> kwplot.autompl()
>>> self.draw(doclf=True)
>>> self.draw(key='pr',  pnum=(1, 2, 1))
>>> self.draw(key='roc', pnum=(1, 2, 2))
>>> kwplot.show_if_requested()
```

catname

summary()

draw(*key=None*, *prefix=''*, ***kw*)

Example

```
>>> cfsn_vecs = ConfusionVectors.demo()
>>> ovr_cfsn = cfsn_vecs.binarize_ovr(keyby='name')
>>> self = ovr_cfsn.measures()['perclass']
>>> self.draw('mcc', doclf=True, fnum=1)
>>> self.draw('pr', doclf=1, fnum=2)
>>> self.draw('roc', doclf=1, fnum=3)
```

summary_plot(*fnum=1*, *title=''*)

Example

```
>>> from kwcoco.metrics.confusion_vectors import *  # NOQA
>>> cfsn_vecs = ConfusionVectors.demo(n=100, p_error=0.5)
>>> binvecs = cfsn_vecs.binarize_peritem()
>>> self = binvecs.measures()
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> self.summary_plot()
>>> kwplot.show_if_requested()
```

class kwcoco.metrics.**OneVsRestConfusionVectors**(*cx_to_binvecs*, *classes*)

Bases: ubelt.util_mixins.NiceRepr

Container for multiple one-vs-rest binary confusion vectors

Variables

- **cx_to_binvecs** –
- **classes** –

Example

```
>>> from kwcoco.metrics import DetectionMetrics
>>> dmet = DetectionMetrics.demo(
>>>     nimgs=10, nboxes=(0, 10), n_fp=(0, 1), nclasses=3)
>>> cfsn_vecs = dmet.confusion_vectors()
>>> self = cfsn_vecs.binarize_ovr(keyby='name')
>>> print('self = {!r}'.format(self))
```

classmethod demo()

keys()

measures(***kwargs*)

Example

```
>>> self = OneVsRestConfusionVectors.demo()
>>> thresh_result = self.measures()['perclass']
```

ovr_classification_report()

class kwcoco.metrics.**PerClass_Measures**(*cx_to_info*)

Bases: ubelt.util_mixins.NiceRepr, *kwcoco.metrics.util.DictProxy*

summary()

draw(*key='mcc'*, *prefix=''*, ***kw*)

Example

```
>>> cfsn_vecs = ConfusionVectors.demo()
>>> ovr_cfsn = cfsn_vecs.binarize_ovr(keyby='name')
>>> self = ovr_cfsn.measures()['perclass']
>>> self.draw('mcc', doclf=True, fnum=1)
>>> self.draw('pr', doclf=1, fnum=2)
>>> self.draw('roc', doclf=1, fnum=3)
```

draw_roc(*prefix=''*, ***kw*)

draw_pr(*prefix=''*, ***kw*)

summary_plot(*fnum=1*, *title=''*)

CommandLine: python ~/code/kwcoco/kwcoco/metrics/confusion_vectors.py PerClass_Measures.summary_plot –show

Example

```
>>> from kwcoco.metrics.confusion_vectors import *  # NOQA
>>> from kwcoco.metrics.detect_metrics import DetectionMetrics
>>> dmet = DetectionMetrics.demo(
>>>     n_fp=(0, 5), n_fn=(0, 5), nimgs=128, nboxes=(0, 10),
>>>     nclasses=3)
>>> cfsn_vecs = dmet.confusion_vectors()
>>> ovr_cfsn = cfsn_vecs.binarize_ovr(keyby='name')
>>> self = ovr_cfsn.measures()['perclass']
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> self.summary_plot(title='demo summary_plot ovr')
>>> kwplot.show_if_requested()
```

kwcoco.metrics.**eval_detections_cli**(***kw*)

CommandLine: xdoctest -m ~/code/kwcoco/kwcoco/metrics/detect_metrics.py eval_detections_cli

1.1.4 kwcoco.util package

Submodules

kwcoco.util.util_futures module

class kwcoco.util.util_futures.**SerialExecutor**
Bases: object

Implements the concurrent.futures API around a single-threaded backend

Example

```
>>> with SerialExecutor() as executor:
>>>     futures = []
>>>     for i in range(100):
>>>         f = executor.submit(lambda x: x + 1, i)
>>>         futures.append(f)
>>>     for f in concurrent.futures.as_completed(futures):
>>>         assert f.result() > 0
>>>     for i, f in enumerate(futures):
>>>         assert i + 1 == f.result()
```

submit(*func*, **args*, ***kw*)

shutdown()

class kwcoco.util.util_futures.**Executor**(*mode='thread'*, *max_workers=0*)
Bases: object

Wrapper around a specific executor.

Abstracts Serial, Thread, and Process Executor via arguments.

Parameters

- **mode** (*str, default='thread'*) – either thread, serial, or process
- **max_workers** (*int, default=0*) – number of workers. If 0, serial is forced.

submit(*func*, **args*, ***kw*)

shutdown()

kwcoco.util.util_json module

kwcoco.util.util_json.**ensure_json_serializable**(*dict_*, *normalize_containers=False*, *verbose=0*)

Attempt to convert common types (e.g. numpy) into something json complient

Convert numpy and tuples into lists

Parameters **normalize_containers** (*bool, default=False*) – if True, normalizes dict containers to be standard python structures.

Example

```
>>> data = ub.ddict(lambda: int)
>>> data['foo'] = ub.ddict(lambda: int)
>>> data['bar'] = np.array([1, 2, 3])
>>> data['foo']['a'] = 1
>>> result = ensure_json_serializable(data, normalize_containers=True)
>>> assert type(result) is dict
```

kwcoco.util.util_json.**find_json_unserializable**(*data*, *quickcheck=False*)

Recurse through json datastructure and find any component that causes a serialization error. Record the location of these errors in the datastructure as we recurse through the call tree.

Parameters

- **data** (*object*) – data that should be json serializable
- **quickcheck** (*bool*) – if True, check the entire datastructure assuming its ok before doing the python-based recursive logic.

Returns

list of "bad part" dictionaries containing items 'value' - the value that caused the serialization error 'loc' - which contains a list of key/indexes that can be used

to lookup the location of the unserializable value. If the "loc" is a list, then it indicates a rare case where a key in a dictionary is causing the serialization error.

Return type List[Dict]

Example

```
>>> from kwcoco.util.util_json import *  # NOQA
>>> part = ub.ddict(lambda: int)
>>> part['foo'] = ub.ddict(lambda: int)
>>> part['bar'] = np.array([1, 2, 3])
>>> part['foo']['a'] = 1
>>> # Create a dictionary with two unserializable parts
>>> data = [1, 2, {'nest1': [2, part]}, {frozenset({'badkey'}): 3, 2: 4}]
>>> parts = list(find_json_unserializable(data))
>>> print('parts = {}'.format(ub.repr2(parts, nl=1)))
>>> # Check expected structure of bad parts
>>> assert len(parts) == 2
>>> part = parts[0]
>>> assert list(part['loc']) == [2, 'nest1', 1, 'bar']
>>> # We can use the "loc" to find the bad value
>>> for part in parts:
>>>     # "loc" is a list of directions containing which keys/indexes
>>>     # to traverse at each descent into the data structure.
>>>     directions = part['loc']
>>>     curr = data
>>>     special_flag = False
>>>     for key in directions:
>>>         if isinstance(key, list):
>>>             # special case for bad keys
>>>             special_flag = True
>>>             break
>>>         else:
```

(continues on next page)

(continued from previous page)

```
>>>             # normal case for bad values
>>>             curr = curr[key]
>>>     if special_flag:
>>>         assert part['data'] in curr.keys()
>>>         assert part['data'] is key[1]
>>>     else:
>>>         assert part['data'] is curr
```

kwcoco.util.util_sklearn module

Extensions to sklearn constructs

class kwcoco.util.util_sklearn.**StratifiedGroupKFold**(*n_splits=3*, *shuffle=False*, *random_state=None*)

Bases: sklearn.model_selection._split._BaseKFold

Stratified K-Folds cross-validator with Grouping

Provides train/test indices to split data in train/test sets.

This cross-validation object is a variation of GroupKFold that returns stratified folds. The folds are made by preserving the percentage of samples for each class.

Read more in the User Guide.

Parameters **n_splits** (*int, default=3*) – Number of folds. Must be at least 2.

split(*X*, *y*, *groups=None*)

Generate indices to split data into training and test set.

kwcoco.util.util_slice module

kwcoco.util.util_slice.**padded_slice**(*data*, *in_slice*, *ndim=None*, *pad_slice=None*, *pad_mode='constant'*, ***padkw*)

Allows slices with out-of-bound coordinates. Any out of bounds coordinate will be sampled via padding.

Note: Negative slices have a different meaning here then they usually do. Normally, they indicate a wrap-around or a reversed stride, but here they index into out-of-bounds space (which depends on the pad mode). For example a slice of -2:1 literally samples two pixels to the left of the data and one pixel from the data, so you get two padded values and one data value.

Parameters

- **data** (*Sliceable[T]*) – data to slice into. Any channels must be the last dimension.
- **in_slice** (*Tuple[slice, ...]*) – slice for each dimensions
- **ndim** (*int*) – number of spatial dimensions
- **pad_slice** (*List[int|Tuple]*) – additional padding of the slice

Returns

data_sliced: subregion of the input data (possibly with padding, depending on if the original slice went out of bounds)

transform : information on how to return to the original coordinates

Currently a dict containing:

st_dims: a list indicating the low and high space-time coordinate values of the returned data slice.

Return type Tuple[Sliceable, Dict]

Example

```
>>> data = np.arange(5)
>>> in_slice = [slice(-2, 7)]
```

```
>>> data_sliced, transform = padded_slice(data, in_slice)
>>> print(ub.repr2(data_sliced, with_dtype=False))
np.array([0, 0, 0, 1, 2, 3, 4, 0, 0])
```

```
>>> data_sliced, transform = padded_slice(data, in_slice, pad_slice=(3, 3))
>>> print(ub.repr2(data_sliced, with_dtype=False))
np.array([0, 0, 0, 0, 0, 0, 1, 2, 3, 4, 0, 0, 0, 0, 0])
```

```
>>> data_sliced, transform = padded_slice(data, slice(3, 4), pad_slice=[(1, 0)])
>>> print(ub.repr2(data_sliced, with_dtype=False))
np.array([2, 3])
```

Module contents

mkinit ~/code/kwcoco/kwcoco/util/__init__.py -w

class kwcoco.util.**Executor**(*mode='thread'*, *max_workers=0*)

Bases: object

Wrapper around a specific executor.

Abstracts Serial, Thread, and Process Executor via arguments.

Parameters

- **mode** (*str, default='thread'*) – either thread, serial, or process
- **max_workers** (*int, default=0*) – number of workers. If 0, serial is forced.

submit(*func*, **args*, ***kw*)

shutdown()

class kwcoco.util.**SerialExecutor**

Bases: object

Implements the concurrent.futures API around a single-threaded backend

Example

```
>>> with SerialExecutor() as executor:
>>>     futures = []
>>>     for i in range(100):
>>>         f = executor.submit(lambda x: x + 1, i)
```

(continues on next page)

(continued from previous page)

```
>>>         futures.append(f)
>>>     for f in concurrent.futures.as_completed(futures):
>>>         assert f.result() > 0
>>>     for i, f in enumerate(futures):
>>>         assert i + 1 == f.result()
```

submit(*func*, **args*, ***kw*)

shutdown()

class kwcoco.util.**StratifiedGroupKFold**(*n_splits=3*, *shuffle=False*, *random_state=None*)

Bases: sklearn.model_selection._split._BaseKFold

Stratified K-Folds cross-validator with Grouping

Provides train/test indices to split data in train/test sets.

This cross-validation object is a variation of GroupKFold that returns stratified folds. The folds are made by preserving the percentage of samples for each class.

Read more in the User Guide.

Parameters **n_splits** (*int, default=3*) – Number of folds. Must be at least 2.

split(*X*, *y*, *groups=None*)

Generate indices to split data into training and test set.

1.2 Submodules

1.2.1 kwcoco.category_tree module

The `category_tree` module defines the *CategoryTree* class, which is used for maintaining flat or hierarchical category information. The kwcoco version of this class only contains the datastructure and does not contain any torch operations. See the ndsampler version for the extension with torch operations.

class kwcoco.category_tree.**CategoryTree**(*graph=None*)

Bases: ubelt.util_mixins.NiceRepr

Wrapper that maintains flat or hierarchical category information.

Helps compute softmaxes and probabilities for tree-based categories where a directed edge (A, B) represents that A is a superclass of B.

Notes

There are three basic properties that this object maintains:

name: Alphanumeric string names that should be generally descriptive. Using spaces and special characters in these names is discouraged, but can be done.

id: The integer id of a category should ideally remain consistent. These are often given by a dataset (e.g. a COCO dataset).

index: Contigous zero-based indices that indexes the list of categories. These should be used for the fastest access in backend computation tasks.

Variables

- **idx_to_node** (*List[str]*) – a list of class names. Implicitly maps from index to category name.
- **id_to_node** (*Dict[int, str]*) – maps integer ids to category names
- **node_to_id** (*Dict[str, int]*) – maps category names to ids
- **node_to_idx** (*Dict[str, int]*) – maps category names to indexes
- **graph** (*nx.Graph*) – a Graph that stores any hierarchy information. For standard mutually exclusive classes, this graph is edgeless. Nodes in this graph can maintain category attributes / properties.
- **idx_groups** (*List[List[int]]*) – groups of category indices that share the same parent category.

Example

```
>>> from kwcoco.category_tree import *
>>> graph = nx.from_dict_of_lists({
>>>     'background': [],
>>>     'foreground': ['animal'],
>>>     'animal': ['mammal', 'fish', 'insect', 'reptile'],
>>>     'mammal': ['dog', 'cat', 'human', 'zebra'],
>>>     'zebra': ['grevys', 'plains'],
>>>     'grevys': ['fred'],
>>>     'dog': ['boxer', 'beagle', 'golden'],
>>>     'cat': ['maine coon', 'persian', 'sphynx'],
>>>     'reptile': ['bearded dragon', 't-rex'],
>>> }, nx.DiGraph)
>>> self = CategoryTree(graph)
>>> print(self)
<CategoryTree(nNodes=22, maxDepth=6, maxBreadth=4...)>
```

Example

```
>>> # The coerce classmethod is the easiest way to create an instance
>>> import kwcoco
>>> kwcoco.CategoryTree.coerce(['a', 'b', 'c'])
<CategoryTree(nNodes=3, nodes=['a', 'b', 'c']) ...
>>> kwcoco.CategoryTree.coerce(4)
<CategoryTree(nNodes=4, nodes=['class_1', 'class_2', 'class_3', ...
>>> kwcoco.CategoryTree.coerce(4)
```

copy()

classmethod from_mutex(*nodes*, *bg_hack=True*)

Parameters **nodes** (*List[str]*) – or a list of class names (in which case they will all be assumed to be mutually exclusive)

Example

```
>>> print(CategoryTree.from_mutex(['a', 'b', 'c']))
<CategoryTree(nNodes=3, ...)>
```

classmethod from_json(*state*)

Parameters **state** (*Dict*) – see __getstate__ / __json__ for details

classmethod from_coco(*categories*)

Create a CategoryTree object from coco categories

Parameters **List[Dict]** – list of coco-style categories

classmethod coerce(*data*, ***kw*)

Attempt to coerce data as a CategoryTree object.

This is primarily useful for when the software stack depends on categories being represent

This will work if the input data is a specially formatted json dict, a list of mutually exclusive classes, or if it is already a CategoryTree. Otherwise an error will be thrown.

Parameters

- **data** (*object*) – a known representation of a category tree.
- ****kwargs** – input type specific arguments

Returns self

Return type *CategoryTree*

Raises

- TypeError - if the input format is unknown
- ValueError - if kwargs are not compatible with the input format

Example

```
>>> import kwcoco
>>> classes1 = kwcoco.CategoryTree.coerce(3)  # integer
>>> classes2 = kwcoco.CategoryTree.coerce(classes1.__json__())  # graph dict
>>> classes3 = kwcoco.CategoryTree.coerce(['class_1', 'class_2', 'class_3'])
↪# mutex list
>>> classes4 = kwcoco.CategoryTree.coerce(classes1.graph)  # nx Graph
>>> classes5 = kwcoco.CategoryTree.coerce(classes1)  # cls
>>> # xdoctest: +REQUIRES(module:ndsampler)
>>> import ndsampler
>>> classes6 = ndsampler.CategoryTree.coerce(3)
>>> classes7 = ndsampler.CategoryTree.coerce(classes1)
>>> classes8 = kwcoco.CategoryTree.coerce(classes6)
```

classmethod demo(*key='coco'*, ***kwargs*)

Parameters **key** (*str*) – specify which demo dataset to use. Can be 'coco' (which uses the default coco demo data). Can be 'btree' which creates a binary tree and accepts kwargs

'r' and 'h' for branching-factor and height.

CommandLine: xdoctest -m ~/code/kwcoco/kwcoco/category_tree.py CategoryTree.demo

Example

```
>>> from kwcoco.category_tree import *
>>> self = CategoryTree.demo()
>>> print('self = {}'.format(self))
self = <CategoryTree(nNodes=10, maxDepth=2, maxBreadth=4...)>
```

to_coco()
Converts to a coco-style data structure

Yields *Dict* – coco category dictionaries

id_to_idx

```
>>> import kwcoco
>>> self = kwcoco.CategoryTree.demo()
>>> self.id_to_idx[1]
```

Type Example

idx_to_id

```
>>> import kwcoco
>>> self = kwcoco.CategoryTree.demo()
>>> self.idx_to_id[0]
```

Type Example

idx_to_ancestor_idxs
memoization decorator for a method that respects args and kwargs

References

http://code.activestate.com/recipes/577452-a-memoize-decorator-for-instance-methods/

Example

```
>>> import ubelt as ub
>>> closure = {'a': 'b', 'c': 'd'}
>>> incr = [0]
>>> class Foo(object):
>>>     @memoize_method
>>>     def foo_memo(self, key):
>>>         value = closure[key]
>>>         incr[0] += 1
>>>         return value
>>>     def foo(self, key):
>>>         value = closure[key]
>>>         incr[0] += 1
>>>         return value
>>> self = Foo()
>>> assert self.foo('a') == 'b' and self.foo('c') == 'd'
>>> assert incr[0] == 2
```

(continues on next page)

(continued from previous page)

```
>>> print('Call memoized version')
>>> assert self.foo_memo('a') == 'b' and self.foo_memo('c') == 'd'
>>> assert incr[0] == 4
>>> assert self.foo_memo('a') == 'b' and self.foo_memo('c') == 'd'
>>> print('Counter should no longer increase')
>>> assert incr[0] == 4
>>> print('Closure changes result without memoization')
>>> closure = {'a': 0, 'c': 1}
>>> assert self.foo('a') == 0 and self.foo('c') == 1
>>> assert incr[0] == 6
>>> assert self.foo_memo('a') == 'b' and self.foo_memo('c') == 'd'
>>> print('Constructing a new object should get a new cache')
>>> self2 = Foo()
>>> self2.foo_memo('a')
>>> assert incr[0] == 7
>>> self2.foo_memo('a')
>>> assert incr[0] == 7
```

idx_to_descendants_idxs
memoization decorator for a method that respects args and kwargs

References

http://code.activestate.com/recipes/577452-a-memoize-decorator-for-instance-methods/

Example

```
>>> import ubelt as ub
>>> closure = {'a': 'b', 'c': 'd'}
>>> incr = [0]
>>> class Foo(object):
>>>     @memoize_method
>>>     def foo_memo(self, key):
>>>         value = closure[key]
>>>         incr[0] += 1
>>>         return value
>>>     def foo(self, key):
>>>         value = closure[key]
>>>         incr[0] += 1
>>>         return value
>>> self = Foo()
>>> assert self.foo('a') == 'b' and self.foo('c') == 'd'
>>> assert incr[0] == 2
>>> print('Call memoized version')
>>> assert self.foo_memo('a') == 'b' and self.foo_memo('c') == 'd'
>>> assert incr[0] == 4
>>> assert self.foo_memo('a') == 'b' and self.foo_memo('c') == 'd'
>>> print('Counter should no longer increase')
>>> assert incr[0] == 4
>>> print('Closure changes result without memoization')
>>> closure = {'a': 0, 'c': 1}
>>> assert self.foo('a') == 0 and self.foo('c') == 1
>>> assert incr[0] == 6
>>> assert self.foo_memo('a') == 'b' and self.foo_memo('c') == 'd'
```

(continues on next page)

(continued from previous page)

```
>>> print('Constructing a new object should get a new cache')
>>> self2 = Foo()
>>> self2.foo_memo('a')
>>> assert incr[0] == 7
>>> self2.foo_memo('a')
>>> assert incr[0] == 7
```

idx_pairwise_distance
memoization decorator for a method that respects args and kwargs

References

http://code.activestate.com/recipes/577452-a-memoize-decorator-for-instance-methods/

Example

```
>>> import ubelt as ub
>>> closure = {'a': 'b', 'c': 'd'}
>>> incr = [0]
>>> class Foo(object):
>>>     @memoize_method
>>>     def foo_memo(self, key):
>>>         value = closure[key]
>>>         incr[0] += 1
>>>         return value
>>>     def foo(self, key):
>>>         value = closure[key]
>>>         incr[0] += 1
>>>         return value
>>> self = Foo()
>>> assert self.foo('a') == 'b' and self.foo('c') == 'd'
>>> assert incr[0] == 2
>>> print('Call memoized version')
>>> assert self.foo_memo('a') == 'b' and self.foo_memo('c') == 'd'
>>> assert incr[0] == 4
>>> assert self.foo_memo('a') == 'b' and self.foo_memo('c') == 'd'
>>> print('Counter should no longer increase')
>>> assert incr[0] == 4
>>> print('Closure changes result without memoization')
>>> closure = {'a': 0, 'c': 1}
>>> assert self.foo('a') == 0 and self.foo('c') == 1
>>> assert incr[0] == 6
>>> assert self.foo_memo('a') == 'b' and self.foo_memo('c') == 'd'
>>> print('Constructing a new object should get a new cache')
>>> self2 = Foo()
>>> self2.foo_memo('a')
>>> assert incr[0] == 7
>>> self2.foo_memo('a')
>>> assert incr[0] == 7
```

is_mutex()
Returns True if all categories are mutually exclusive (i.e. flat)

If true, then the classes may be represented as a simple list of class names without any loss of information, otherwise the underlying category graph is necessary to preserve all knowledge.

Todo:

- [ ] what happens when we have a dummy root?

num_classes

class_names

category_names

cats

Returns a mapping from category names to category attributes.

If this category tree was constructed from a coco-dataset, then this will contain the coco category attributes.

Returns Dict[str, Dict[str, object]]

Example

```
>>> from kwcoco.category_tree import *
>>> self = CategoryTree.demo()
>>> print('self.cats = {!r}'.format(self.cats))
```

index(*node*)

Return the index that corresponds to the category name

show()

Ignore:

```
>>> import kwplot
>>> kwplot.autompl()
>>> from kwcoco import category_tree
>>> self = category_tree.CategoryTree.demo()
>>> self.show()
```

python -c "import kwplot, kwcoco, graphid; kwplot.autompl(); graphid.util.show_nx(kwcoco.category_tree.CategoryTree.demo().graph); kwplot.show_if_requested()" --show

1.2.2 kwcoco.coco_dataset module

An implementation and extension of the original MS-COCO API[1].

Extends the format to also include line annotations.

Dataset Spec:

- Note: a formal spec has been defined in

```
category = {
    'id': int,
    'name': str,
    'supercategory': Optional[str],
    'keypoints': Optional(List[str]),
```

(continues on next page)

[1] http://cocodataset.org/#format-data

(continued from previous page)

```
    'skeleton': Optional(List[Tuple[Int, Int]]),
}

image = {
    'id': int,
    'file_name': str
}

dataset = {
    # these are object level categories
    'categories': [category],
    'images': [image]
        ...
    ],
    'annotations': [
        {
            'id': Int,
            'image_id': Int,
            'category_id': Int,
            'track_id': Optional[Int],

            'bbox': [tl_x, tl_y, w, h],  # optional (xywh format)
            "score" : float,  # optional
            "prob" : List[float],  # optional
            "weight" : float,  # optional

            "caption": str,  # an optional text caption for this annotation
            "iscrowd" : <0 or 1>,  # denotes if the annotation covers a single object␣
↪(0) or multiple objects (1)
            "keypoints" : [x1,y1,v1,...,xk,yk,vk], # or new dict-based format
            'segmentation': <RunLengthEncoding | Polygon>,  # formats are defined␣
↪bellow
        },
        ...
    ],
    'licenses': [],
    'info': [],
}

Polygon:
    A flattned list of xy coordinates.
    [x1, y1, x2, y2, ..., xn, yn]

    or a list of flattned list of xy coordinates if the CCs are disjoint
    [[x1, y1, x2, y2, ..., xn, yn], [x1, y1, ..., xm, ym],]

    Note: the original coco spec does not allow for holes in polygons.

    We also allow a non-standard dictionary encoding of polygons
        {'exterior': [(x1, y1)...],
         'interiors': [[(x1, y1), ...], ...]}

RunLengthEncoding:
    The RLE can be in a special bytes encoding or in a binary array
    encoding. We reuse the original C functions are in [2]_ in
    ``kwimage.structs.Mask`` to provide a convinient way to abstract this
    rather esoteric bytes encoding.
```

(continues on next page)

(continued from previous page)

```
    For pure python implementations see kwimage:
        Converting from an image to RLE can be done via kwimage.run_length_encoding
        Converting from RLE back to an image can be done via:
            kwimage.decode_run_length

        For compatibility with the COCO specs ensure the binary flags
        for these functions are set to true.

Keypoints:
    Annotation keypoints may also be specified in this non-standard (but
    ultimately more general) way:

    'annotations': [
        {
            'keypoints': [
                {
                    'xy': <x1, y1>,
                    'visible': <0 or 1 or 2>,
                    'keypoint_category_id': <kp_cid>,
                    'keypoint_category': <kp_name, optional>,  # this can be␣
↪specified instead of an id
                }, ...
            ]
        }, ...
    ],
    'keypoint_categories': [{
        'name': <str>,
        'id': <int>,  # an id for this keypoint category
        'supercategory': <kp_name>  # name of coarser parent keypoint class (for␣
↪hierarchical keypoints)
        'reflection_id': <kp_cid>  # specify only if the keypoint id would be swapped␣
↪with another keypoint type
    },...
    ]

    In this scheme the "keypoints" property of each annotation (which used
    to be a list of floats) is now specified as a list of dictionaries that
    specify each keypoints location, id, and visibility explicitly. This
    allows for things like non-unique keypoints, partial keypoint
    annotations. This also removes the ordering requirement, which makes it
    simpler to keep track of each keypoints class type.

    We also have a new top-level dictionary to specify all the possible
    keypoint categories.

Auxillary Channels:
    For multimodal or multispectral images it is possible to specify
    auxillary channels in an image dictionary as follows:

    {
        'id': int, 'file_name': str
        'channels': <spec>,  # a spec code that indicates the layout of these␣
↪channels.
        'auxillary': [  # information about auxillary channels
            {
                'file_name':
```

(continues on next page)

(continued from previous page)

```
                'channels': <spec>
            }, ... # can have many auxillary channels with unique specs
        ]
    }

Video Sequences:
    For video sequences, we add the following video level index:

    "videos": [
        { "id": <int>, "name": <video_name:str> },
    ]

    Note that the videos might be given as encoded mp4/avi/etc.. files (in
    which case the name should correspond to a path) or as a series of
    frames in which case the images should be used to index the extracted
    frames and information in them.

    Then image dictionaries are augmented as follows:

    {
        'video_id': str  # optional, if this image is a frame in a video sequence,␣
↪this id is shared by all frames in that sequence.
        'timestamp': int  # optional, timestamp (ideally in flicks), used to identify␣
↪the timestamp of the frame. Only applicable video inputs.
        'frame_index': int  # optional, ordinal frame index which can be used if␣
↪timestamp is unknown.
    }

    And annotations are augmented as follows:

    {
        "track_id": <int | str | uuid>  # optional, indicates association between␣
↪annotations across frames
    }
```

Notes

The main object in this file is class:*CocoDataset*, which is composed of several mixin classes. See the class and method documentation for more details.

Todo:

- [ ] Use ijson to lazilly load pieces of the dataset in the background or on demand. This will give us faster access to categories / images, whereas we will always have to wait for annotations etc...
- [ ] Should img_root be changed to data root?
- [ ] Read video data, return numpy arrays (requires API for images)
- [ ] Spec for video URI, and convert to frames @ framerate function.
- [ ] remove videos

References

class kwcoco.coco_dataset.**ObjectList1D**(*ids*, *dset*, *key*)
Bases: ubelt.util_mixins.NiceRepr

Vectorized access to lists of dictionary objects

Lightweight reference to a set of object (e.g. annotations, images) that allows for convenient property access.

Parameters

- **ids** (*List[int]*) – list of ids
- **dset** (*CocoDataset*) – parent dataset
- **key** (*str*) – main object name (e.g. 'images', 'annotations')

Types: ObjT = Ann | Img | Cat # can be one of these types ObjectList1D gives us access to a List[ObjT]

Example

```
>>> import kwcoco
>>> dset = kwcoco.CocoDataset.demo()
>>> # Both annots and images are object lists
>>> self = dset.annots()
>>> self = dset.images()
>>> # can call with a list of ids or not, for everything
>>> self = dset.annots([1, 2, 11])
>>> self = dset.images([1, 2, 3])
>>> self.lookup('id')
>>> self.lookup(['id'])
```

objs
all object dictionaries

Type Returns

Type List

take(*idxs*)
Take a subset by index

Example

```
>>> self = CocoDataset.demo().annots()
>>> assert len(self.take([0, 2, 3])) == 3
```

compress(*flags*)
Take a subset by flags

Example

```
>>> self = CocoDataset.demo().images()
>>> assert len(self.compress([True, False, True])) == 2
```

peek()
Return the first object dictionary

lookup(*key*, *default=NoParam*, *keepid=False*)
Lookup a list of object attributes

Parameters

- **key** (*str | Iterable*) – name of the property you want to lookup can also be a list of names, in which case we return a dict
- **default** – if specified, uses this value if it doesn't exist in an ObjT.
- **keepid** – if True, return a mapping from ids to the property

Returns a list of whatever type the object is Dict[str, ObjT]

Return type List[ObjT]

Example

```
>>> import kwcoco
>>> dset = kwcoco.CocoDataset.demo()
>>> self = dset.annots()
>>> self.lookup('id')
>>> key = ['id']
>>> default = None
>>> self.lookup(key=['id', 'image_id'])
>>> self.lookup(key=['id', 'image_id'])
>>> self.lookup(key='foo', default=None, keepid=True)
>>> self.lookup(key=['foo'], default=None, keepid=True)
>>> self.lookup(key=['id', 'image_id'], keepid=True)
```

get(*key*, *default=NoParam*, *keepid=False*)
Lookup a list of object attributes

Parameters

- **key** (*str*) – name of the property you want to lookup
- **default** – if specified, uses this value if it doesn't exist in an ObjT.
- **keepid** – if True, return a mapping from ids to the property

Returns a list of whatever type the object is Dict[str, ObjT]

Return type List[ObjT]

Example

```
>>> import kwcoco
>>> dset = kwcoco.CocoDataset.demo()
>>> self = dset.annots()
>>> self.get('id')
>>> self.get(key='foo', default=None, keepid=True)
```

set(*key*, *values*)
Assign a value to each annotation

Parameters

- **key** (*str*) – the annotation property to modify

- **values** (*Iterable | scalar*) – an iterable of values to set for each annot in the dataset. If the item is not iterable, it is assigned to all objects.

Example

```
>>> dset = CocoDataset.demo()
>>> self = dset.annots()
>>> self.set('my-key1', 'my-scalar-value')
>>> self.set('my-key2', np.random.rand(len(self)))
>>> print('dset.imgs = {}'.format(ub.repr2(dset.imgs, nl=1)))
>>> self.get('my-key2')
```

class kwcoco.coco_dataset.**ObjectGroups**(*groups*, *dset*)

Bases: ubelt.util_mixins.NiceRepr

An object for holding a groups of *ObjectList1D* objects

lookup(*key*, *default=NoParam*)

class kwcoco.coco_dataset.**Categories**(*ids*, *dset*)

Bases: *kwcoco.coco_dataset.ObjectList1D*

Vectorized access to category attributes

Example

```
>>> from kwcoco.coco_dataset import Categories  # NOQA
>>> import kwcoco
>>> dset = kwcoco.CocoDataset.demo()
>>> ids = list(dset.cats.keys())
>>> self = Categories(ids, dset)
>>> print('self.name = {!r}'.format(self.name))
>>> print('self.supercategory = {!r}'.format(self.supercategory))
```

cids

name

supercategory

class kwcoco.coco_dataset.**Videos**(*ids*, *dset*)

Bases: *kwcoco.coco_dataset.ObjectList1D*

Vectorized access to video attributes

Example

```
>>> from kwcoco.coco_dataset import Videos  # NOQA
>>> import kwcoco
>>> dset = kwcoco.CocoDataset.demo('vidshapes5')
>>> ids = list(dset.index.videos.keys())
>>> self = Videos(ids, dset)
>>> print('self = {!r}'.format(self))
```

class kwcoco.coco_dataset.**Images**(*ids*, *dset*)

Bases: *kwcoco.coco_dataset.ObjectList1D*

Vectorized access to image attributes

gids

gname

gpath

width

height

size

```
>>> from kwcoco.coco_dataset import *
>>> self = CocoDataset.demo().images()
>>> self._dset._ensure_imgsize()
>>> print(self.size)
[(512, 512), (300, 250), (256, 256)]
```

Type Example

area

```
>>> from kwcoco.coco_dataset import *
>>> self = CocoDataset.demo().images()
>>> self._dset._ensure_imgsize()
>>> print(self.area)
[262144, 75000, 65536]
```

Type Example

n_annots

```
>>> self = CocoDataset.demo().images()
>>> print(ub.repr2(self.n_annots, nl=0))
[9, 2, 0]
```

Type Example

aids

```
>>> self = CocoDataset.demo().images()
>>> print(ub.repr2(list(map(list, self.aids)), nl=0))
[[1, 2, 3, 4, 5, 6, 7, 8, 9], [10, 11], []]
```

Type Example

annots

```
>>> self = CocoDataset.demo().images()
>>> print(self.annots)
<AnnotGroups(n=3, m=3.7, s=3.9)>
```

Type Example

class kwcoco.coco_dataset.**Annots**(*ids*, *dset*)

Bases: *kwcoco.coco_dataset.ObjectList1D*

Vectorized access to annotation attributes

aids

The annotation ids of this column of annotations

images

Get the column of images

Returns Images

image_id

category_id

gids

Get the column of image-ids

Returns list of image ids

Return type List[int]

cids

Get the column of category-ids

Returns List[int]

cnames

Get the column of category names

Returns List[int]

detections

Get the kwimage-style detection objects

Returns kwimage.Detections

Example

```
>>> # xdoctest: +REQUIRES(module:kwimage)
>>> from kwcoco.coco_dataset import *  # NOQA
>>> self = CocoDataset.demo('shapes32').annots([1, 2, 11])
>>> dets = self.detections
>>> print('dets.data = {!r}'.format(dets.data))
>>> print('dets.meta = {!r}'.format(dets.meta))
```

boxes

Get the column of kwimage-style bounding boxes

Example

```
>>> self = CocoDataset.demo().annots([1, 2, 11])
>>> print(self.boxes)
<Boxes(xywh,
    array([[ 10,  10, 360, 490],
           [350,   5, 130, 290],
           [124,  96,  45,  18]]))>
```

xywh
Returns raw boxes

Example

```
>>> self = CocoDataset.demo().annots([1, 2, 11])
>>> print(self.xywh)
```

class kwcoco.coco_dataset.**AnnotGroups**(*groups*, *dset*)
Bases: *kwcoco.coco_dataset.ObjectGroups*

cids

class kwcoco.coco_dataset.**ImageGroups**(*groups*, *dset*)
Bases: *kwcoco.coco_dataset.ObjectGroups*

class kwcoco.coco_dataset.**MixinCocoDepricate**
Bases: object

These functions are marked for deprication and may be removed at any time

lookup_imgs(*filename=None*)
Linear search for an images with specific attributes

DEPRICATE

Ignore: filename = '201503.20150525.101841191.573975.png' list(self.lookup_imgs(filename)) gid = 64940 img = self.imgs[gid] img['file_name'] = filename

lookup_anns(*has=None*)
Linear search for an annotations with specific attributes

DEPRICATE

Ignore: list(self.lookup_anns(has='radius')) gid = 112888 img = self.imgs[gid] img['file_name'] = filename

class kwcoco.coco_dataset.**MixinCocoExtras**
Bases: object

Misc functions for coco

load_image(*gid_or_img*)
Reads an image from disk and

Parameters **gid_or_img** (*int or dict*) – image id or image dict

Returns the image

Return type np.ndarray

load_image_fpath(*gid_or_img*)

get_image_fpath(*gid_or_img*)
Returns the full path to the image

Parameters **gid_or_img** (*int or dict*) – image id or image dict

Returns full path to the image

Return type PathLike

get_auxillary_fpath(*gid_or_img*, *channels*)
Returns the full path to auxillary data for an image

Parameters

- **gid_or_img** (*int | dict*) – an image or its id
- **channels** (*str*) – the auxillary channel to load (e.g. disparity)

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo('shapes8', aux=True)
>>> self.get_auxillary_fpath(1, 'disparity')
```

load_annot_sample(*aid_or_ann*, *image=None*, *pad=None*)
Reads the chip of an annotation. Note this is much less efficient than using a sampler, but it doesn't require disk cache.

Parameters

- **aid_or_int** (*int or dict*) – annot id or dict
- **image** (*ArrayLike, default=None*) – preloaded image (note: this process is inefficient unless image is specified)

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo()
>>> sample = self.load_annot_sample(2, pad=100)
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(sample['im'])
>>> kwplot.show_if_requested()
```

classmethod coerce(*key*, ***kw*)

classmethod demo(*key='photos'*, ***kw*)
Create a toy coco dataset for testing and demo puposes

Parameters

- **key** (*str*) – either photos or shapes
- ****kw** – if key is shapes, these arguments are passed to toydata generation

Example

```
>>> print(CocoDataset.demo('photos'))
>>> print(CocoDataset.demo('shapes', verbose=0))
>>> print(CocoDataset.demo('shapes256', verbose=0))
>>> print(CocoDataset.demo('shapes8', verbose=0))
```

Example

```
>>> import kwcoco
>>> dset = kwcoco.CocoDataset.demo('vidshapes5', num_frames=5, verbose=0,␣
↪rng=None)
>>> dset = kwcoco.CocoDataset.demo('vidshapes5', num_frames=5, num_tracks=4,␣
↪verbose=0, rng=44)
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> pnums = kwplot.PlotNums(nSubplots=len(dset.imgs))
>>> fnum = 1
>>> for gx, gid in enumerate(dset.imgs.keys()):
>>>     canvas = dset.draw_image(gid=gid)
>>>     kwplot.imshow(canvas, pnum=pnums[gx], fnum=fnum)
>>>     #dset.show_image(gid=gid, pnum=pnums[gx])
>>> kwplot.show_if_requested()
```

category_graph()

Construct a networkx category hierarchy

Returns

graph: a directed graph where category names are the nodes, supercategories define edges, and items in each category dict (e.g. category id) are added as node properties.

Return type network.DiGraph

Example

```
>>> self = CocoDataset.demo()
>>> graph = self.category_graph()
>>> assert 'astronaut' in graph.nodes()
>>> assert 'keypoints' in graph.nodes['human']
```

import graphid graphid.util.show_nx(graph)

object_categories()

Construct a consistent CategoryTree representation of object classes

Returns category data structure

Return type *kwcoco.CategoryTree*

Example

```
>>> self = CocoDataset.demo()
>>> classes = self.object_categories()
>>> print('classes = {}'.format(classes))
```

keypoint_categories()
Construct a consistent CategoryTree representation of keypoint classes

Returns category data structure

Return type *kwcoco.CategoryTree*

Example

```
>>> self = CocoDataset.demo()
>>> classes = self.keypoint_categories()
>>> print('classes = {}'.format(classes))
```

missing_images(*check_aux=False*, *verbose=0*)
Check for images that don't exist

Parameters **check_aux** (*bool, default=Fasle*) – if specified also checks auxillary images

Returns bad indexes and paths

Return type List[Tuple[int, str]]

corrupted_images(*verbose=0*)
Check for images that don't exist or can't be opened

Returns bad indexes and paths

Return type List[Tuple[int, str]]

rename_categories(*mapper*, *strict=False*, *preserve=False*, *rebuild=True*, *simple=True*, *merge_policy='ignore'*)
Create a coarser categorization

Note: this function has been unstable in the past, and has not yet been properly stabalized. Either avoid or use with care. Ensuring `simple=True` should result in newer saner behavior that will likely be backwards compatible.

Todo:

- [X] Simple case where we relabel names with no conflicts
- **[ ] Case where annotation labels need to change to be coarser**
  - dev note: see internal libraries for work on this
- [ ] Other cases

Parameters

- **mapper** (*dict or Function*) – maps old names to new names.
- **strict** (*bool*) – DEPRICATED IGNORE. if True, fails if mapper doesnt map all classes
- **preserve** (*bool*) – DEPRICATED IGNORE. if True, preserve old categories as supercategories. Broken.

- **simple** (*bool, default=True*) – defaults to the new way of doing this. The old way is depricated.
- **merge_policy** (*str*) – How to handle multiple categories that map to the same name. Can be update or ignore.

Example

```
>>> self = CocoDataset.demo()
>>> self.rename_categories({'astronomer': 'person',
>>>                         'astronaut': 'person',
>>>                         'mouth': 'person',
>>>                         'helmet': 'hat'}, preserve=0)
>>> assert 'hat' in self.name_to_cat
>>> assert 'helmet' not in self.name_to_cat
>>> # Test merge case
>>> self = CocoDataset.demo()
>>> mapper = {
>>>     'helmet': 'rocket',
>>>     'astronomer': 'rocket',
>>>     'human': 'rocket',
>>>     'mouth': 'helmet',
>>>     'star': 'gas'
>>> }
>>> self.rename_categories(mapper)
```

rebase (**args*, ***kw*)
Deprecated use reroot instead

reroot (*new_root=None*, *old_root=None*, *absolute=False*, *check=True*, *safe=True*, *smart=False*)
Rebase image/data paths onto a new image/data root.

Parameters

- **new_root** (*str, default=None*) – New image root. If unspecified the current `self.img_root` is used.
- **old_root** (*str, default=None*) – If specified, removes the root from file names. If unspecified, then the existing paths MUST be relative to `new_root`.
- **absolute** (*bool, default=False*) – if True, file names are stored as absolute paths, otherwise they are relative to the new image root.
- **check** (*bool, default=True*) – if True, checks that the images all exist.
- **safe** (*bool, default=True*) – if True, does not overwrite values until all checks pass
- **smart** (*bool, default=False*) – If True, we can try different reroot strategies and choose the one that works. Note, always be wary when algorithms try to be smart.

CommandLine: xdoctest -m /home/joncrall/code/kwcoco/kwcoco/coco_dataset.py MixinCocoExtras.reroot

---

Todo:

- [ ] Incorporate maximum ordered subtree embedding once completed?

Ignore:

```
>>> # There might not be a way to easily handle the cases that I
>>> # want to here. Might need to discuss this.
>>> import kwcoco
>>> import os
>>> gname = 'images/foo.png'
>>> remote = '/remote/path'
>>> host = ub.ensure_app_cache_dir('kwcoco/tests/reroot')
>>> fpath = join(host, gname)
>>> ub.ensuredir(dirname(fpath))
>>> # In this test the image exists on the host path
>>> import kwimage
>>> kwimage.imwrite(fpath, np.random.rand(8, 8))
>>> #
>>> cases = {}
>>> # * given absolute paths on current machine
>>> cases['abs_curr'] = kwcoco.CocoDataset.from_image_paths([join(host,␣
↪gname)])
>>> # * given "remote" rooted relative paths on current machine
>>> cases['rel_remoterooted_curr'] = kwcoco.CocoDataset.from_image_
↪paths([gname], img_root=remote)
>>> # * given "host" rooted relative paths on current machine
>>> cases['rel_hostrooted_curr'] = kwcoco.CocoDataset.from_image_
↪paths([gname], img_root=host)
>>> # * given unrooted relative paths on current machine
>>> cases['rel_unrooted_curr'] = kwcoco.CocoDataset.from_image_
↪paths([gname])
>>> # * given absolute paths on another machine
>>> cases['abs_remote'] = kwcoco.CocoDataset.from_image_
↪paths([join(remote, gname)])
>>> def report(dset, name):
>>>     gid = 1
>>>     rel_fpath = dset.imgs[gid]['file_name']
>>>     abs_fpath = dset.get_image_fpath(gid)
>>>     color = 'green' if exists(abs_fpath) else 'red'
>>>     print('    * strategy_name = {!r}'.format(name))
>>>     print('        * rel_fpath = {!r}'.format(rel_fpath))
>>>     print('        * ' + ub.color_text('abs_fpath = {!r}'.format(abs_
↪fpath), color))
>>> for key, dset in cases.items():
>>>     print('----')
>>>     print('case key = {!r}'.format(key))
>>>     print('ORIG = {!r}'.format(dset.imgs[1]['file_name']))
>>>     print('dset.img_root = {!r}'.format(dset.img_root))
>>>     print('missing_gids = {!r}'.format(dset.missing_images()))
>>>     print('cwd = {!r}'.format(os.getcwd()))
>>>     print('host = {!r}'.format(host))
>>>     print('remote = {!r}'.format(remote))
>>>     #
>>>     dset_None_rel = dset.copy().reroot(absolute=False, check=0)
>>>     report(dset_None_rel, 'dset_None_rel')
>>>     #
>>>     dset_None_abs = dset.copy().reroot(absolute=True, check=0)
>>>     report(dset_None_abs, 'dset_None_abs')
>>>     #
>>>     dset_host_rel = dset.copy().reroot(host, absolute=False, check=0)
>>>     report(dset_host_rel, 'dset_host_rel')
```

(continues on next page)

(continued from previous page)

```
>>>     #
>>>     dset_host_abs = dset.copy().reroot(host, absolute=True, check=0)
>>>     report(dset_host_abs, 'dset_host_abs')
>>>     #
>>>     dset_remote_rel = dset.copy().reroot(host, old_root=remote,␣
↪absolute=False, check=0)
>>>     report(dset_remote_rel, 'dset_remote_rel')
>>>     #
>>>     dset_remote_abs = dset.copy().reroot(host, old_root=remote,␣
↪absolute=True, check=0)
>>>     report(dset_remote_abs, 'dset_remote_abs')
```

Example

```
>>> import kwcoco
>>> def report(dset, name):
>>>     gid = 1
>>>     abs_fpath = dset.get_image_fpath(gid)
>>>     rel_fpath = dset.imgs[gid]['file_name']
>>>     color = 'green' if exists(abs_fpath) else 'red'
>>>     print('strategy_name = {!r}'.format(name))
>>>     print(ub.color_text('abs_fpath = {!r}'.format(abs_fpath), color))
>>>     print('rel_fpath = {!r}'.format(rel_fpath))
>>> dset = self = kwcoco.CocoDataset.demo()
>>> # Change base relative directory
>>> img_root = ub.expandpath('~')
>>> print('ORIG self.imgs = {!r}'.format(self.imgs))
>>> print('ORIG dset.img_root = {!r}'.format(dset.img_root))
>>> print('NEW img_root       = {!r}'.format(img_root))
>>> self.reroot(img_root)
>>> report(self, 'self')
>>> print('NEW self.imgs = {!r}'.format(self.imgs))
>>> assert self.imgs[1]['file_name'].startswith('.cache')
```

```
>>> # Use absolute paths
>>> self.reroot(absolute=True)
>>> assert self.imgs[1]['file_name'].startswith(img_root)
```

```
>>> # Switch back to relative paths
>>> self.reroot()
>>> assert self.imgs[1]['file_name'].startswith('.cache')
```

Example

```
>>> # demo with auxillary data
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo('shapes8', aux=True)
>>> img_root = ub.expandpath('~')
>>> print(self.imgs[1]['file_name'])
>>> print(self.imgs[1]['auxillary'][0]['file_name'])
>>> self.reroot(img_root)
>>> print(self.imgs[1]['file_name'])
```

(continues on next page)

(continued from previous page)

```
>>> print(self.imgs[1]['auxillary'][0]['file_name'])
>>> assert self.imgs[1]['file_name'].startswith('.cache')
>>> assert self.imgs[1]['auxillary'][0]['file_name'].startswith('.cache')
```

data_root
In the future we may deprecate img_root for data_root

find_representative_images(*gids=None*)
Find images that have a wide array of categories. Attempt to find the fewest images that cover all categories using images that contain both a large and small number of annotations.

Parameters **gids** (*None | List*) – Subset of image ids to consider when finding representative images. Uses all images if unspecified.

Returns list of image ids determined to be representative

Return type List

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo()
>>> gids = self.find_representative_images()
>>> print('gids = {!r}'.format(gids))
>>> gids = self.find_representative_images([3])
>>> print('gids = {!r}'.format(gids))
```

```
>>> self = kwcoco.CocoDataset.demo('shapes8')
>>> gids = self.find_representative_images()
>>> print('gids = {!r}'.format(gids))
>>> valid = {7, 1}
>>> gids = self.find_representative_images(valid)
>>> assert valid.issuperset(gids)
>>> print('gids = {!r}'.format(gids))
```

class kwcoco.coco_dataset.**MixinCocoAttrs**
Bases: object

Expose methods to construct object lists / groups

annots(*aids=None*, *gid=None*)
Return vectorized annotation objects

Parameters

- **aids** (*List[int]*) – annotation ids to reference, if unspecified all annotations are returned.
- **gid** (*int*) – return all annotations that belong to this image id. mutually exclusive with *aids* arg.

Returns vectorized annotation object

Return type *Annots*

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo()
>>> annots = self.annots()
>>> print(annots)
<Annots(num=11)>
>>> sub_annots = annots.take([1, 2, 3])
>>> print(sub_annots)
<Annots(num=3)>
>>> print(ub.repr2(sub_annots.get('bbox', None)))
[
    [350, 5, 130, 290],
    None,
    None,
]
```

images (*gids=None*)
Return vectorized image objects

Parameters **gids** (*List[int]*) – image ids to reference, if unspecified all images are returned.

Returns vectorized images object

Return type *Images*

Example

```
>>> self = CocoDataset.demo()
>>> images = self.images()
>>> print(images)
<Images(num=3)>
```

categories (*cids=None*)
Return vectorized category objects

Example

```
>>> self = CocoDataset.demo()
>>> categories = self.categories()
>>> print(categories)
<Categories(num=8)>
```

videos (*vidids=None*)
Return vectorized video objects

Example

```
>>> self = CocoDataset.demo('vidshapes2')
>>> videos = self.videos()
>>> print(videos)
>>> videos.lookup('name')
>>> videos.lookup('id')
>>> print('videos.objs = {}'.format(ub.repr2(videos.objs[0:2], nl=1)))
```

class kwcoco.coco_dataset.**MixinCocoStats**
Bases: object

Methods for getting stats about the dataset

n_annots

n_images

n_cats

n_videos

keypoint_annotation_frequency()

Example

```
>>> from kwcoco.coco_dataset import *
>>> self = CocoDataset.demo('shapes', rng=0)
>>> hist = self.keypoint_annotation_frequency()
>>> hist = ub.odict(sorted(hist.items()))
>>> # FIXME: for whatever reason demodata generation is not determenistic␣
↪when seeded
>>> print(ub.repr2(hist))  # xdoc: +IGNORE_WANT
{
    'bot_tip': 6,
    'left_eye': 14,
    'mid_tip': 6,
    'right_eye': 14,
    'top_tip': 6,
}
```

category_annotation_frequency()
Reports the number of annotations of each category

Example

```
>>> from kwcoco.coco_dataset import *
>>> self = CocoDataset.demo()
>>> hist = self.category_annotation_frequency()
>>> print(ub.repr2(hist))
{
    'astroturf': 0,
    'human': 0,
    'astronaut': 1,
    'astronomer': 1,
    'helmet': 1,
    'rocket': 1,
    'mouth': 2,
    'star': 5,
}
```

category_annotation_type_frequency()
Reports the number of annotations of each type for each category

Example

```
>>> self = CocoDataset.demo()
>>> hist = self.category_annotation_frequency()
>>> print(ub.repr2(hist))
```

basic_stats()
Reports number of images, annotations, and categories.

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo()
>>> print(ub.repr2(self.basic_stats()))
{
    'n_anns': 11,
    'n_imgs': 3,
    'n_videos': 0,
    'n_cats': 8,
}
```

```
>>> from kwcoco.demo.toydata import *  # NOQA
>>> dset = random_video_dset(render=True, num_frames=2, num_tracks=10, rng=0)
>>> print(ub.repr2(dset.basic_stats()))
{
    'n_anns': 20,
    'n_imgs': 2,
    'n_videos': 1,
    'n_cats': 3,
}
```

extended_stats()
Reports number of images, annotations, and categories.

Example

```
>>> self = CocoDataset.demo()
>>> print(ub.repr2(self.extended_stats()))
```

boxsize_stats(*anchors=None*, *perclass=True*, *gids=None*, *aids=None*, *verbose=0*, *clusterkw={}*, *statskw={}*)
Compute statistics about bounding box sizes.

Also computes anchor boxes using kmeans if `anchors` is specified.

Parameters

- **anchors** (*int*) – if specified also computes box anchors
- **perclass** (*bool*) – if True also computes stats for each category
- **gids** (*List[int], default=None*) – if specified only compute stats for these image ids.
- **aids** (*List[int], default=None*) – if specified only compute stats for these annotation ids.
- **verbose** (*int*) – verbosity level

- **clusterkw** (*dict*) – kwargs for `sklearn.cluster.KMeans` used if computing anchors.
- **statskw** (*dict*) – kwargs for `kwarray.stats_dict()`

Returns Dict[str, Dict[str, Dict | ndarray]

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo('shapes32')
>>> infos = self.boxsize_stats(anchors=4, perclass=False)
>>> print(ub.repr2(infos, nl=-1, precision=2))
```

```
>>> infos = self.boxsize_stats(gids=[1], statskw=dict(median=True))
>>> print(ub.repr2(infos, nl=-1, precision=2))
```

class kwcoco.coco_dataset.**MixinCocoDraw**

Bases: object

Matplotlib / display functionality

imread(*gid*)

Loads a particular image

draw_image(*gid*)

Use kwimage to draw all annotations on an image and return the pixels as a numpy array.

Returns canvas

Return type ndarray

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo('shapes8')
>>> self.draw_image(1)
>>> # Now you can dump the annotated image to disk / whatever
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(canvas)
```

show_image(*gid=None*, *aids=None*, *aid=None*, ***kwargs*)

Use matplotlib to show an image with annotations overlaid

Parameters

- **gid** (*int*) – image to show
- **aids** (*list*) – aids to highlight within the image
- **aid** (*int*) – a specific aid to focus on. If gid is not give, look up gid based on this aid.
- ****kwargs** – show_annots, show_aid, show_catname, show_kpname, show_segmentation, title, show_gid, show_filename, show_boxes,

Ignore: # Programatically collect the kwargs for docs generation import xinspect import kwcoco kwargs = xinspect.get_kwargs(kwcoco.CocoDataset.show_image) print(ub.repr2(list(kwargs.keys()), nl=1, si=1))

class kwcoco.coco_dataset.**MixinCocoAddRemove**

Bases: object

Mixin functions to dynamically add / remove annotations images and categories while maintaining lookup indexes.

add_video(*name*, *id=None*, ***kw*)

Add a video to the dataset (dynamically updates the index)

Parameters

- **name** (*str*) – Unique name for this video.
- **id** (*None or int*) – ADVANCED. Force using this image id.
- ****kw** – stores arbitrary key/value pairs in this new video

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset()
>>> print('self.index.videos = {}'.format(ub.repr2(self.index.videos, nl=1)))
>>> print('self.index.imgs = {}'.format(ub.repr2(self.index.imgs, nl=1)))
>>> print('self.index.vidid_to_gids = {!r}'.format(self.index.vidid_to_gids))
```

```
>>> vidid1 = self.add_video('foo', id=3)
>>> vidid2 = self.add_video('bar')
>>> vidid3 = self.add_video('baz')
>>> print('self.index.videos = {}'.format(ub.repr2(self.index.videos, nl=1)))
>>> print('self.index.imgs = {}'.format(ub.repr2(self.index.imgs, nl=1)))
>>> print('self.index.vidid_to_gids = {!r}'.format(self.index.vidid_to_gids))
```

```
>>> gid1 = self.add_image('foo1.jpg', video_id=vidid1)
>>> gid2 = self.add_image('foo2.jpg', video_id=vidid1)
>>> gid3 = self.add_image('foo3.jpg', video_id=vidid1)
>>> self.add_image('bar1.jpg', video_id=vidid2)
>>> print('self.index.videos = {}'.format(ub.repr2(self.index.videos, nl=1)))
>>> print('self.index.imgs = {}'.format(ub.repr2(self.index.imgs, nl=1)))
>>> print('self.index.vidid_to_gids = {!r}'.format(self.index.vidid_to_gids))
```

```
>>> self.remove_images([gid2])
>>> print('self.index.vidid_to_gids = {!r}'.format(self.index.vidid_to_gids))
```

add_image(*file_name*, *id=None*, ***kw*)

Add an image to the dataset (dynamically updates the index)

Parameters

- **file_name** (*str*) – relative or absolute path to image
- **id** (*None or int*) – ADVANCED. Force using this image id.
- ****kw** – stores arbitrary key/value pairs in this new image

Example

```
>>> self = CocoDataset.demo()
>>> import kwimage
>>> gname = kwimage.grab_test_image_fpath('paraview')
>>> gid = self.add_image(gname)
>>> assert self.imgs[gid]['file_name'] == gname
```

add_annotation(*image_id*, *category_id=None*, *bbox=None*, *id=None*, ***kw*)

Add an annotation to the dataset (dynamically updates the index)

Parameters

- **image_id** (*int*) – image_id to add to
- **category_id** (*int*) – category_id to add to
- **bbox** (*list or kwimage.Boxes*) – bounding box in xywh format
- **id** (*None or int*) – ADVANCED. Force using this annotation id.
- ****kw** – stores arbitrary key/value pairs in this new image

Example

```
>>> self = CocoDataset.demo()
>>> image_id = 1
>>> cid = 1
>>> bbox = [10, 10, 20, 20]
>>> aid = self.add_annotation(image_id, cid, bbox)
>>> assert self.anns[aid]['bbox'] == bbox
```

Example

```
>>> # Attempt to annot without a category or bbox
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo()
>>> image_id = 1
>>> aid = self.add_annotation(image_id)
>>> assert None in self.index.cid_to_aids
```

add_category(*name*, *supercategory=None*, *id=None*, ***kw*)

Adds a category

Parameters

- **name** (*str*) – name of the new category
- **supercategory** (*str, optional*) – parent of this category
- **id** (*int, optional*) – use this category id, if it was not taken
- ****kw** – stores arbitrary key/value pairs in this new image

Example

```
>>> self = CocoDataset.demo()
>>> prev_n_cats = self.n_cats
>>> cid = self.add_category('dog', supercategory='object')
>>> assert self.cats[cid]['name'] == 'dog'
>>> assert self.n_cats == prev_n_cats + 1
>>> import pytest
>>> with pytest.raises(ValueError):
>>>     self.add_category('dog', supercategory='object')
```

ensure_image (*file_name*, *id=None*, ***kw*)

Like add_image, but returns the existing image id if it already exists instead of failing. In this case all metadata is ignored.

Parameters

- **file_name** (*str*) – relative or absolute path to image
- **id** (*None or int*) – ADVANCED. Force using this image id.
- ****kw** – stores arbitrary key/value pairs in this new image

Returns the existing or new image id

Return type int

ensure_category (*name*, *supercategory=None*, *id=None*, ***kw*)

Like add_category, but returns the existing category id if it already exists instead of failing. In this case all metadata is ignored.

Returns the existing or new category id

Return type int

add_annotations (*anns*)

Faster less-safe multi-item alternative

Parameters **anns** (*List[Dict]*) – list of annotation dictionaries

Example

```
>>> self = CocoDataset.demo()
>>> anns = [self.anns[aid] for aid in [2, 3, 5, 7]]
>>> self.remove_annotations(anns)
>>> assert self.n_annots == 7 and self._check_index()
>>> self.add_annotations(anns)
>>> assert self.n_annots == 11 and self._check_index()
```

add_images (*imgs*)

Faster less-safe multi-item alternative

Note: THIS FUNCTION WAS DESIGNED FOR SPEED, AS SUCH IT DOES NOT CHECK IF THE IMAGE-IDs or FILE_NAMES ARE DUPLICATED AND WILL BLINDLY ADD DATA EVEN IF IT IS BAD. THE SINGLE IMAGE VERSION IS SLOWER BUT SAFER.

Parameters **imgs** (*List[Dict]*) – list of image dictionaries

Example

```
>>> imgs = CocoDataset.demo().dataset['images']
>>> self = CocoDataset()
>>> self.add_images(imgs)
>>> assert self.n_images == 3 and self._check_index()
```

clear_images()
Removes all images and annotations (but not categories)

Example

```
>>> self = CocoDataset.demo()
>>> self.clear_images()
>>> print(ub.repr2(self.basic_stats(), nobr=1, nl=0, si=1))
n_anns: 0, n_imgs: 0, n_videos: 0, n_cats: 8
```

clear_annotations()
Removes all annotations (but not images and categories)

Example

```
>>> self = CocoDataset.demo()
>>> self.clear_annotations()
>>> print(ub.repr2(self.basic_stats(), nobr=1, nl=0, si=1))
n_anns: 0, n_imgs: 3, n_videos: 0, n_cats: 8
```

remove_all_images()
Removes all images and annotations (but not categories)

Example

```
>>> self = CocoDataset.demo()
>>> self.clear_images()
>>> print(ub.repr2(self.basic_stats(), nobr=1, nl=0, si=1))
n_anns: 0, n_imgs: 0, n_videos: 0, n_cats: 8
```

remove_all_annotations()
Removes all annotations (but not images and categories)

Example

```
>>> self = CocoDataset.demo()
>>> self.clear_annotations()
>>> print(ub.repr2(self.basic_stats(), nobr=1, nl=0, si=1))
n_anns: 0, n_imgs: 3, n_videos: 0, n_cats: 8
```

remove_annotation(*aid_or_ann*)
Remove a single annotation from the dataset

If you have multiple annotations to remove its more efficient to remove them in batch with `self.remove_annotations`

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo()
>>> aids_or_anns = [self.anns[2], 3, 4, self.anns[1]]
>>> self.remove_annotations(aids_or_anns)
>>> assert len(self.dataset['annotations']) == 7
>>> self._check_index()
```

remove_annotations(*aids_or_anns*, *verbose=0*, *safe=True*)

Remove multiple annotations from the dataset.

Parameters

- **anns_or_aids** (*List*) – list of annotation dicts or ids
- **safe** (*bool, default=True*) – if True, we perform checks to remove duplicates and non-existing identifiers.

Returns num_removed: information on the number of items removed

Return type Dict

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo()
>>> prev_n_annots = self.n_annots
>>> aids_or_anns = [self.anns[2], 3, 4, self.anns[1]]
>>> self.remove_annotations(aids_or_anns)  # xdoc: +IGNORE_WANT
{'annotations': 4}
>>> assert len(self.dataset['annotations']) == prev_n_annots - 4
>>> self._check_index()
```

remove_categories(*cat_identifiers*, *keep_annots=False*, *verbose=0*, *safe=True*)

Remove categories and all annotations in those categories. Currently does not change any hierarchy information

Parameters

- **cat_identifiers** (*List*) – list of category dicts, names, or ids
- **keep_annots** (*bool, default=False*) – if True, keeps annotations, but removes category labels.
- **safe** (*bool, default=True*) – if True, we perform checks to remove duplicates and non-existing identifiers.

Returns num_removed: information on the number of items removed

Return type Dict

Example

```
>>> self = CocoDataset.demo()
>>> cat_identifiers = [self.cats[1], 'rocket', 3]
>>> self.remove_categories(cat_identifiers)
```

(continues on next page)

(continued from previous page)

```
>>> assert len(self.dataset['categories']) == 5
>>> self._check_index()
```

remove_images (*gids_or_imgs*, *verbose=0*, *safe=True*)

Parameters

- **gids_or_imgs** (*List*) – list of image dicts, names, or ids
- **safe** (*bool, default=True*) – if True, we perform checks to remove duplicates and non-existing identifiers.

Returns num_removed: information on the number of items removed

Return type Dict

Example

```
>>> from kwcoco.coco_dataset import *
>>> self = CocoDataset.demo()
>>> assert len(self.dataset['images']) == 3
>>> gids_or_imgs = [self.imgs[2], 'astro.png']
>>> self.remove_images(gids_or_imgs)  # xdoc: +IGNORE_WANT
{'annotations': 11, 'images': 2}
>>> assert len(self.dataset['images']) == 1
>>> self._check_index()
>>> gids_or_imgs = [3]
>>> self.remove_images(gids_or_imgs)
>>> assert len(self.dataset['images']) == 0
>>> self._check_index()
```

remove_annotation_keypoints (*kp_identifiers*)

Removes all keypoints with a particular category

Parameters **kp_identifiers** (*List*) – list of keypoint category dicts, names, or ids

Returns num_removed: information on the number of items removed

Return type Dict

remove_keypoint_categories (*kp_identifiers*)

Removes all keypoints of a particular category as well as all annotation keypoints with those ids.

Parameters **kp_identifiers** (*List*) – list of keypoint category dicts, names, or ids

Returns num_removed: information on the number of items removed

Return type Dict

Example

```
>>> self = CocoDataset.demo('shapes', rng=0)
>>> kp_identifiers = ['left_eye', 'mid_tip']
>>> remove_info = self.remove_keypoint_categories(kp_identifiers)
>>> print('remove_info = {!r}'.format(remove_info))
>>> # FIXME: for whatever reason demodata generation is not determenistic␣
↪when seeded
```

(continues on next page)

(continued from previous page)

```
>>> # assert remove_info == {'keypoint_categories': 2, 'annotation_keypoints
↪': 16, 'reflection_ids': 1}
>>> assert self._resolve_to_kpcat('right_eye')['reflection_id'] is None
```

set_annotation_category(*aid_or_ann*, *cid_or_cat*)

Sets the category of a single annotation

Parameters

- **aid_or_ann** (*dict | int*) – annotation dict or id
- **cid_or_cat** (*dict | int*) – category dict or id

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo()
>>> old_freq = self.category_annotation_frequency()
>>> aid_or_ann = aid = 2
>>> cid_or_cat = new_cid = self.ensure_category('kitten')
>>> self.set_annotation_category(aid, new_cid)
>>> new_freq = self.category_annotation_frequency()
>>> print('new_freq = {}'.format(ub.repr2(new_freq, nl=1)))
>>> print('old_freq = {}'.format(ub.repr2(old_freq, nl=1)))
>>> assert sum(new_freq.values()) == sum(old_freq.values())
>>> assert new_freq['kitten'] == 1
```

class kwcoco.coco_dataset.**CocoIndex**

Bases: object

Fast lookup index for the COCO dataset with dynamic modification

Variables

- ***imgs*** (*Dict[int, dict]*) – mapping between image ids and the image dictionaries
- ***anns*** (*Dict[int, dict]*) – mapping between annotation ids and the annotation dictionaries
- ***cats*** (*Dict[int, dict]*) – mapping between category ids and the category dictionaries

cid_to_gids

```
>>> import kwcoco
>>> self = dset = kwcoco.CocoDataset()
>>> self.index.cid_to_gids
```

Type Example

clear()

build(*parent*)

Build all id-to-obj reverse indexes from scratch.

Parameters **parent** (*CocoDataset*) – the dataset to index

Notation: aid - Annotation ID gid - imaGe ID cid - Category ID vidid - Video ID

Example

```
>>> from kwcoco.demo.toydata import *  # NOQA
>>> parent = CocoDataset.demo('vidshapes1', num_frames=4, rng=1)
>>> index = parent.index
>>> index.build(parent)
```

class kwcoco.coco_dataset.**MixinCocoIndex**

Bases: object

Give the dataset top level access to index attributes

anns

imgs

cats

videos

gid_to_aids

cid_to_aids

name_to_cat

class kwcoco.coco_dataset.**CocoDataset**(*data=None*, *tag=None*, *img_root=None*, *autobuild=True*)

Bases: ubelt.util_mixins.NiceRepr, *kwcoco.coco_dataset.MixinCocoAddRemove*, *kwcoco.coco_dataset.MixinCocoStats*, *kwcoco.coco_dataset.MixinCocoAttrs*, *kwcoco.coco_dataset.MixinCocoDraw*, *kwcoco.coco_dataset.MixinCocoExtras*, *kwcoco.coco_dataset.MixinCocoIndex*, *kwcoco.coco_dataset.MixinCocoDepricate*

Notes

A keypoint annotation

{ “image_id” : int, “category_id” : int, “keypoints” : [x1,y1,v1,. . . ,xk,yk,vk], “score” : float,

} Note that `v[i]` is a visibility flag, where v=0: not labeled,

v=1: labeled but not visible, and v=2: labeled and visible.

A bounding box annotation

{ “image_id” : int, “category_id” : int, “bbox” : [x,y,width,height], “score” : float,

}

We also define a non-standard “line” annotation (which our fixup scripts will interpret as the diameter of a circle to convert into a bounding box)

A line* annotation (note this is a non-standard field)

{ “image_id” : int, “category_id” : int, “line” : [x1,y1,x2,y2], “score” : float,

}

Lastly, note that our datasets will sometimes specify multiple bbox, line, and/or, keypoints fields. In this case we may also specify a field roi_shape, which denotes which field is the “main” annotation type.

Variables

- **dataset** (*Dict*) – raw json data structure. This is the base dictionary that contains {'annotations': List, 'images': List, 'categories': List}
- ***index*** (CocoIndex) – an efficient lookup index into the coco data structure. The index defines its own attributes like anns, cats, imgs, etc. See *CocoIndex* for more details on which attributes are available.
- **fpath** (*PathLike | None*) – if known, this stores the filepath the dataset was loaded from
- **tag** (*str*) – A tag indicating the name of the dataset.
- **img_root** (*PathLike | None*) – If known, this is the root path that all image file names are relative to. This can also be manually overwritten by the user.
- **hashid** (*str | None*) – If computed, this will be a hash uniquely identifing the dataset. To ensure this is computed see _build_hashid().

References

http://cocodataset.org/#format http://cocodataset.org/#download

CommandLine: python -m kwcoco.coco_dataset CocoDataset –show

Example

```
>>> dataset = demo_coco_data()
>>> self = CocoDataset(dataset, tag='demo')
>>> # xdoctest: +REQUIRES(--show)
>>> self.show_image(gid=2)
>>> from matplotlib import pyplot as plt
>>> plt.show()
```

classmethod from_data(*data*, *img_root=None*)
Constructor from a json dictionary

classmethod from_image_paths(*gpaths*, *img_root=None*)
Constructor from a list of images paths

Example

```
>>> coco_dset = CocoDataset.from_image_paths(['a.png', 'b.png'])
>>> assert coco_dset.n_images == 2
```

classmethod from_coco_paths(*fpaths*, *max_workers=0*, *verbose=1*, *mode='thread'*, *union='try'*)
Constructor from multiple coco file paths.

Loads multiple coco datasets and unions the result

Notes

if the union operation fails, the list of individually loaded files is returned instead.

Parameters

- **fpaths** (*List[str]*) – list of paths to multiple coco files to be loaded and unioned.
- **max_workers** (*int, default=0*) – number of worker threads / processes
- **verbose** (*int*) – verbosity level
- **mode** (*str*) – thread, process, or serial
- **union** (*str | bool, default='try'*) – If True, unions the result datasets after loading. If False, just returns the result list. If 'try', then try to preform the union, but return the result list if it fails.

copy()

Deep copies this object

Example

```
>>> from kwcoco.coco_dataset import *
>>> self = CocoDataset.demo()
>>> new = self.copy()
>>> assert new.imgs[1] is new.dataset['images'][0]
>>> assert new.imgs[1] == self.dataset['images'][0]
>>> assert new.imgs[1] is not self.dataset['images'][0]
```

dumps(*indent=None*, *newlines=False*)

Writes the dataset out to the json format

Parameters **newlines** (*bool*) – if True, each annotation, image, category gets its own line

Notes

Using newlines=True is similar to: print(ub.repr2(dset.dataset, nl=2, trailsep=False)) However, the above may not output valid json if it contains ndarrays.

Example

```
>>> from kwcoco.coco_dataset import *
>>> import json
>>> self = CocoDataset.demo()
>>> text = self.dumps(newlines=True)
>>> print(text)
>>> self2 = CocoDataset(json.loads(text), tag='demo2')
>>> assert self2.dataset == self.dataset
>>> assert self2.dataset is not self.dataset
```

```
>>> text = self.dumps(newlines=True)
>>> print(text)
>>> self2 = CocoDataset(json.loads(text), tag='demo2')
>>> assert self2.dataset == self.dataset
>>> assert self2.dataset is not self.dataset
```

Ignore:

for k in self2.dataset:

if self.dataset[k] == self2.dataset[k]: print('YES: k = {!r}'.format(k))

else: print('NO: k = {!r}'.format(k))

self2.dataset['categories'] self.dataset['categories']

dump (*file*, *indent=None*, *newlines=False*)

Writes the dataset out to the json format

Parameters

- **file** (*PathLike | FileLike*) – Where to write the data. Can either be a path to a file or an open file pointer / stream.
- **newlines** (*bool*) – if True, each annotation, image, category gets its own line.

Example

```
>>> import tempfile
>>> from kwcoco.coco_dataset import *
>>> self = CocoDataset.demo()
>>> file = tempfile.NamedTemporaryFile('w')
>>> self.dump(file)
>>> file.seek(0)
>>> text = open(file.name, 'r').read()
>>> print(text)
>>> file.seek(0)
>>> dataset = json.load(open(file.name, 'r'))
>>> self2 = CocoDataset(dataset, tag='demo2')
>>> assert self2.dataset == self.dataset
>>> assert self2.dataset is not self.dataset
```

```
>>> file = tempfile.NamedTemporaryFile('w')
>>> self.dump(file, newlines=True)
>>> file.seek(0)
>>> text = open(file.name, 'r').read()
>>> print(text)
>>> file.seek(0)
>>> dataset = json.load(open(file.name, 'r'))
>>> self2 = CocoDataset(dataset, tag='demo2')
>>> assert self2.dataset == self.dataset
>>> assert self2.dataset is not self.dataset
```

union (**others*, ***kwargs*)

Merges multiple *CocoDataset* items into one. Names and associations are retained, but ids may be different.

Parameters

- **self** – note that *union()* can be called as an instance method or a class method. If it is a class method, then this is the class type, otherwise the instance will also be unioned with `others`.
- ***others** – a series of CocoDatasets that we will merge
- ****kwargs** – constructor options for the new merged CocoDataset

Returns a new merged coco dataset

Return type *CocoDataset*

Example

```
>>> # Test union works with different keypoint categories
>>> dset1 = CocoDataset.demo('shapes1')
>>> dset2 = CocoDataset.demo('shapes2')
>>> dset1.remove_keypoint_categories(['bot_tip', 'mid_tip', 'right_eye'])
>>> dset2.remove_keypoint_categories(['top_tip', 'left_eye'])
>>> dset_12a = CocoDataset.union(dset1, dset2)
>>> dset_12b = dset1.union(dset2)
>>> dset_21 = dset2.union(dset1)
>>> def add_hist(h1, h2):
>>>     return {k: h1.get(k, 0) + h2.get(k, 0) for k in set(h1) | set(h2)}
>>> kpfreq1 = dset1.keypoint_annotation_frequency()
>>> kpfreq2 = dset2.keypoint_annotation_frequency()
>>> kpfreq_want = add_hist(kpfreq1, kpfreq2)
>>> kpfreq_got1 = dset_12a.keypoint_annotation_frequency()
>>> kpfreq_got2 = dset_12b.keypoint_annotation_frequency()
>>> assert kpfreq_want == kpfreq_got1
>>> assert kpfreq_want == kpfreq_got2
```

```
>>> # Test disjoint gid datasets
>>> import kwcoco
>>> dset1 = kwcoco.CocoDataset.demo('shapes3')
>>> for new_gid, img in enumerate(dset1.dataset['images'], start=10):
>>>     for aid in dset1.gid_to_aids[img['id']]:
>>>         dset1.anns[aid]['image_id'] = new_gid
>>>     img['id'] = new_gid
>>> dset1.index.clear()
>>> dset1._build_index()
>>> # ------
>>> dset2 = kwcoco.CocoDataset.demo('shapes2')
>>> for new_gid, img in enumerate(dset2.dataset['images'], start=100):
>>>     for aid in dset2.gid_to_aids[img['id']]:
>>>         dset2.anns[aid]['image_id'] = new_gid
>>>     img['id'] = new_gid
>>> dset1.index.clear()
>>> dset2._build_index()
>>> others = [dset1, dset2]
>>> merged = kwcoco.CocoDataset.union(*others)
>>> print('merged = {!r}'.format(merged))
>>> print('merged.imgs = {}'.format(ub.repr2(merged.imgs, nl=1)))
>>> assert set(merged.imgs) & set([10, 11, 12, 100, 101]) == set(merged.imgs)
```

```
>>> # Test data is not preserved
>>> dset2 = kwcoco.CocoDataset.demo('shapes2')
>>> dset1 = kwcoco.CocoDataset.demo('shapes3')
>>> others = (dset1, dset2)
>>> cls = self = kwcoco.CocoDataset
>>> merged = cls.union(*others)
>>> print('merged = {!r}'.format(merged))
>>> print('merged.imgs = {}'.format(ub.repr2(merged.imgs, nl=1)))
>>> assert set(merged.imgs) & set([1, 2, 3, 4, 5]) == set(merged.imgs)
```

Todo:

- [ ] are supercategories broken?

- [ ] reuse image ids where possible
- [ ] reuse annotation / category ids where possible
- [ ] disambiguate track-ids
- [x] disambiguate video-ids

---

subset(*gids*, *copy=False*, *autobuild=True*)

Return a subset of the larger coco dataset by specifying which images to port. All annotations in those images will be taken.

Parameters

- **gids** (*List[int]*) – image-ids to copy into a new dataset
- **copy** (*bool, default=False*) – if True, makes a deep copy of all nested attributes, otherwise makes a shallow copy.
- **autobuild** (*bool, default=True*) – if True will automatically build the fast lookup index.

Example

```
>>> self = CocoDataset.demo()
>>> gids = [1, 3]
>>> sub_dset = self.subset(gids)
>>> assert len(self.gid_to_aids) == 3
>>> assert len(sub_dset.gid_to_aids) == 2
```

Example

```
>>> self = CocoDataset.demo()
>>> sub1 = self.subset([1])
>>> sub2 = self.subset([2])
>>> sub3 = self.subset([3])
>>> others = [sub1, sub2, sub3]
>>> rejoined = CocoDataset.union(*others)
>>> assert len(sub1.anns) == 9
>>> assert len(sub2.anns) == 2
>>> assert len(sub3.anns) == 0
>>> assert rejoined.basic_stats() == self.basic_stats()
```

kwcoco.coco_dataset.**demo_coco_data**()

Simple data for testing

Ignore: # code for getting a segmentation polygon kwimage.grab_test_image_fpath('astro') labelme /home/joncrall/.cache/kwimage/demodata/astro.png cat /home/joncrall/.cache/kwimage/demodata/astro.json

Example

```
>>> # xdoctest: +REQUIRES(--show)
>>> from kwcoco.coco_dataset import demo_coco_data, CocoDataset
>>> dataset = demo_coco_data()
>>> self = CocoDataset(dataset, tag='demo')
>>> import kwplot
```

(continues on next page)

(continued from previous page)

```
>>> kwplot.autompl()
>>> self.show_image(gid=1)
>>> kwplot.show_if_requested()
```

1.2.3 kwcoco.coco_evaluator module

Evaluates a predicted coco dataset against a truth coco dataset.

The components in this module work programatically or as a command line script.

class kwcoco.coco_evaluator.**CocoEvalConfig**(*data=None*, *default=None*, *cmdline=False*)

Bases: scriptconfig.config.Config

Evaluate and score predicted versus truth detections / classifications in a COCO dataset

default = {'classes_of_interest': <Value(<class 'list'>: None)>, 'draw': <Value(Non

class kwcoco.coco_evaluator.**CocoEvaluator**(*config*)

Bases: object

Abstracts the evaluation process to execute on two coco datasets.

This can be run as a standalone script where the user specifies the paths to the true and predited dataset explicitly, or this can be used by a higher level script that produces the predictions and then sends them to this evaluator.

Ignore:

```
>>> pred_fpath1 = ub.expandpath("$HOME/remote/viame/work/bioharn/fit/nice/
↪bioharn-det-mc-cascade-rgb-fine-coi-v43/eval/may_priority_habcam_cfarm_v7_
↪test.mscoc/bioharn-det-mc-cascade-rgb-fine-coi-v43__epoch_00000007/c=0.1,
↪i=window,n=0.8,window_d=512,512,window_o=0.0/all_pred.mscoco.json")
>>> pred_fpath2 = ub.expandpath('$HOME/tmp/cached_clf_out_cli/reclassified.
↪mscoco.json')
>>> true_fpath = ub.expandpath('$HOME/remote/namek/data/noaa_habcam/combos/
↪habcam_cfarm_v8_test.mscoco.json')
>>> config = {
>>>     'true_dataset': true_fpath,
>>>     'pred_dataset': pred_fpath2,
>>>     'out_dpath': ub.expandpath('$HOME/remote/namek/tmp/reclassified_eval
↪'),
>>>     'classes_of_interest': [],
>>> }
>>> coco_eval = CocoEvaluator(config)
>>> config = coco_eval.config
>>> coco_eval._init()
>>> coco_eval.evaluate()
```

Example

```
>>> from kwcoco.coco_evaluator import CocoEvaluator
>>> import kwcoco
>>> dpath = ub.ensure_app_cache_dir('kwcoco/tests/test_out_dpath')
>>> true_dset = kwcoco.CocoDataset.demo('shapes8')
>>> from kwcoco.demo.perterb import perterb_coco
>>> kwargs = {
>>>     'box_noise': 0.5,
```

(continues on next page)

(continued from previous page)

```
>>>     'n_fp': (0, 10),
>>>     'n_fn': (0, 10),
>>>     'with_probs': True,
>>> }
>>> pred_dset = perterb_coco(true_dset, **kwargs)
>>> config = {
>>>     'true_dataset': true_dset,
>>>     'pred_dataset': pred_dset,
>>>     'out_dpath': dpath,
>>>     'classes_of_interest': [],
>>> }
>>> coco_eval = CocoEvaluator(config)
>>> results = coco_eval.evaluate()
```

log(*msg*, *level='INFO'*)

evaluate()

Example

```
>>> from kwcoco.coco_evaluator import *  # NOQA
>>> from kwcoco.coco_evaluator import CocoEvaluator
>>> import kwcoco
>>> dpath = ub.ensure_app_cache_dir('kwcoco/tests/test_out_dpath')
>>> true_dset = kwcoco.CocoDataset.demo('shapes8')
>>> from kwcoco.demo.perterb import perterb_coco
>>> kwargs = {
>>>     'box_noise': 0.5,
>>>     'n_fp': (0, 10),
>>>     'n_fn': (0, 10),
>>>     'with_probs': True,
>>> }
>>> pred_dset = perterb_coco(true_dset, **kwargs)
>>> config = {
>>>     'true_dataset': true_dset,
>>>     'pred_dataset': pred_dset,
>>>     'out_dpath': dpath,
>>> }
>>> coco_eval = CocoEvaluator(config)
>>> results = coco_eval.evaluate()
```

class kwcoco.coco_evaluator.**CocoResults**(*measures*, *ovr_measures*, *cfsn_vecs*, *meta=None*)

Bases: ubelt.util_mixins.NiceRepr

Container class to store, draw, summarize, and serialize results from CocoEvaluator.

dump(*file*, *indent=' '*)

Serialize to json file

dump_figures(*out_dpath*, *expt_title=None*)

class kwcoco.coco_evaluator.**CocoEvalCLIConfig**(*data=None*, *default=None*, *cmdline=False*)

Bases: scriptconfig.config.Config

default = {'classes_of_interest': <Value(<class 'list'>: None)>, 'draw': <Value(Non

kwcoco.coco_evaluator.**main**(*cmdline=True*, ***kw*)

1.2.4 kwcoco.compat_dataset module

A wrapper around the basic kwcoco dataset with a pycocotools API.

We do not recommend using this API because it has some idiosyncrasies, where names can be missleading and APIs are not always clear / efficient: e.g.

(1) catToImgs returns integer image ids but imgToAnns returns annotation dictionaries.

(2) showAnns takes a dictionary list as an argument instead of an integer list

The cool thing is that this extends the kwcoco API so you can drop this for compatibility with the old API, but you still get access to all of the kwcoco API including dynamic addition / removal of categories / annotations / images.

class kwcoco.compat_dataset.**COCO**(*annotation_file=None*, ***kw*)

Bases: *kwcoco.coco_dataset.CocoDataset*

A wrapper around the basic kwcoco dataset with a pycocotools API.

Example

```
>>> from kwcoco.compat_dataset import *  # NOQA
>>> import kwcoco
>>> basic = kwcoco.CocoDataset.demo('shapes8')
>>> self = COCO(basic.dataset)
>>> self.info()
>>> print('self.imgToAnns = {!r}'.format(self.imgToAnns[1]))
>>> print('self.catToImgs = {!r}'.format(self.catToImgs))
```

createIndex()

info()

Print information about the annotation file. :return:

imgToAnns

catToImgs

unlike the name implies, this actually goes from category to image ids Name retained for backward compatibility

getAnnIds(*imgIds=[]*, *catIds=[]*, *areaRng=[]*, *iscrowd=None*)

Get ann ids that satisfy given filter conditions. default skips that filter :param imgIds (int array) : get anns for given imgs

catIds (int array) : get anns for given cats areaRng (float array) : get anns for given area range (e.g. [0 inf]) iscrowd (boolean) : get anns for given crowd label (False or True)

Returns ids (int array) : integer array of ann ids

Example

```
>>> from kwcoco.compat_dataset import *  # NOQA
>>> import kwcoco
>>> self = COCO(kwcoco.CocoDataset.demo('shapes8').dataset)
>>> self.getAnnIds()
>>> self.getAnnIds(imgIds=1)
>>> self.getAnnIds(imgIds=[1])
>>> self.getAnnIds(catIds=[3])
```

getCatIds(*catNms=[]*, *supNms=[]*, *catIds=[]*)

filtering parameters. default skips that filter. :param catNms (str array) : get cats for given cat names :param supNms (str array) : get cats for given supercategory names :param catIds (int array) : get cats for given cat ids :return: ids (int array) : integer array of cat ids

Example

```
>>> from kwcoco.compat_dataset import *  # NOQA
>>> import kwcoco
>>> self = COCO(kwcoco.CocoDataset.demo('shapes8').dataset)
>>> self.getCatIds()
>>> self.getCatIds(catNms=['superstar'])
>>> self.getCatIds(supNms=['raster'])
>>> self.getCatIds(catIds=[3])
```

getImgIds(*imgIds=[]*, *catIds=[]*)

Get img ids that satisfy given filter conditions. :param imgIds (int array) : get imgs for given ids :param catIds (int array) : get imgs with all given cats :return: ids (int array) : integer array of img ids

Example

```
>>> from kwcoco.compat_dataset import *  # NOQA
>>> import kwcoco
>>> self = COCO(kwcoco.CocoDataset.demo('shapes8').dataset)
>>> self.getImgIds(imgIds=[1, 2])
>>> self.getImgIds(catIds=[3, 6, 7])
>>> self.getImgIds(catIds=[3, 6, 7], imgIds=[1, 2])
```

loadAnns(*ids=[]*)

Load anns with the specified ids. :param ids (int array) : integer ids specifying anns :return: anns (object array) : loaded ann objects

loadCats(*ids=[]*)

Load cats with the specified ids. :param ids (int array) : integer ids specifying cats :return: cats (object array) : loaded cat objects

loadImgs(*ids=[]*)

Load anns with the specified ids. :param ids (int array) : integer ids specifying img :return: imgs (object array) : loaded img objects

showAnns(*anns*, *draw_bbox=False*)

Display the specified annotations. :param anns (array of object): annotations to display :return: None

loadRes(*resFile*)

Load result file and return a result api object. :param resFile (str) : file name of result file :return: res (obj) : result api object

download(*tarDir=None*, *imgIds=[]*)

Download COCO images from mscoco.org server. :param tarDir (str): COCO results directory name

imgIds (list): images to be downloaded

Returns

loadNumpyAnnotations(*data*)
Convert result data from a numpy array [Nx7] where each row contains {imageID,x1,y1,w,h,score,class} :param data (numpy.ndarray) :return: annotations (python nested list)

annToRLE(*ann*)
Convert annotation which can be polygons, uncompressed RLE to RLE. :return: binary mask (numpy 2D array)

Example

```
>>> from kwcoco.compat_dataset import *  # NOQA
>>> import kwcoco
>>> self = COCO(kwcoco.CocoDataset.demo('shapes8').dataset)
>>> ann = {'id': 1}
>>> self.annToRLE(ann)
```

annToMask(*ann*)
Convert annotation which can be polygons, uncompressed RLE, or RLE to binary mask. :return: binary mask (numpy 2D array)

TODO: fixme

Ignore:

```
>>> from kwcoco.compat_dataset import *  # NOQA
>>> import kwcoco
>>> self = COCO(kwcoco.CocoDataset.demo('shapes8').dataset)
>>> ann = {'id': 1}
>>> self.annToMask(ann)
```

1.2.5 kwcoco.kpf module

WIP:

Conversions to and from KPF format.

kwcoco.kpf.**coco_to_kpf**(*coco_dset*)
import kwcoco coco_dset = kwcoco.CocoDataset.demo('shapes8')

kwcoco.kpf.**demo**()

1.2.6 kwcoco.kw18 module

A helper for converting COCO to / from KW18 format.

class kwcoco.kw18.**KW18**(*data*)
Bases: kwarray.dataframe_light.DataFrameArray

A DataFrame like object that stores KW18 column data

Example

```
>>> import kwcoco
>>> from kwcoco.kw18 import KW18
>>> coco_dset = kwcoco.CocoDataset.demo('shapes')
>>> kw18_dset = KW18.from_coco(coco_dset)
>>> print(kw18_dset.pandas())
```

DEFAULT_COLUMNS = ['track_id', 'track_length', 'frame_number', 'tracking_plane_loc_x',

classmethod demo()

classmethod from_coco(*coco_dset*)

to_coco()

Translates a kw18 files to a CocoDataset.

Note: kw18 does not contain complete information, and as such the returned coco dataset may need to be augmented.

Todo:

- [ ] allow kwargs to specify path to frames / videos

Example

```
>>> from kwcoco.kw18 import KW18
>>> self = KW18.demo()
>>> self.to_coco()
```

classmethod load(*file*)

classmethod loads(*text*)

Example

```
>>> self = KW18.demo()
>>> text = self.dumps()
>>> self2 = KW18.loads(text)
>>> empty = KW18.loads('')
```

dump(*file*)

dumps()

Example

```
>>> self = KW18.demo()
>>> text = self.dumps()
>>> print(text)
```

1.2.7 kwcoco.spec module

1.2.8 kwcoco.toydata module

1.2.9 kwcoco.toypatterns module

1.3 Module contents

The Kitware COCO module defines a variant of the Microsoft COCO format, originally developed for the "collected images in context" object detection challenge. We are backwards compatible with the original module, but we also have improved implementations in several places, including segmentations and keypoints.

The *kwcoco.CocoDataset* class is capable of dynamic addition and removal of categories, images, and annotations. Has better support for keypoints and segmentation formats than the original COCO format. Despite being written in Python, this data structure is reasonably efficient.

class kwcoco.**CocoDataset**(*data=None*, *tag=None*, *img_root=None*, *autobuild=True*)

Bases: ubelt.util_mixins.NiceRepr, *kwcoco.coco_dataset.MixinCocoAddRemove*, *kwcoco.coco_dataset.MixinCocoStats*, *kwcoco.coco_dataset.MixinCocoAttrs*, *kwcoco.coco_dataset.MixinCocoDraw*, *kwcoco.coco_dataset.MixinCocoExtras*, *kwcoco.coco_dataset.MixinCocoIndex*, *kwcoco.coco_dataset.MixinCocoDepricate*

Notes

A keypoint annotation

{ "image_id" : int, "category_id" : int, "keypoints" : [x1,y1,v1,...,xk,yk,vk], "score" : float,

} Note that `v[i]` is a visibility flag, where v=0: not labeled,

v=1: labeled but not visible, and v=2: labeled and visible.

A bounding box annotation

{ "image_id" : int, "category_id" : int, "bbox" : [x,y,width,height], "score" : float,

}

We also define a non-standard "line" annotation (which our fixup scripts will interpret as the diameter of a circle to convert into a bounding box)

A line* annotation (note this is a non-standard field)

{ "image_id" : int, "category_id" : int, "line" : [x1,y1,x2,y2], "score" : float,

}

Lastly, note that our datasets will sometimes specify multiple bbox, line, and/or, keypoints fields. In this case we may also specify a field roi_shape, which denotes which field is the "main" annotation type.

Variables

- **dataset** (*Dict*) – raw json data structure. This is the base dictionary that contains {'annotations': List, 'images': List, 'categories': List}
- ***index*** (CocoIndex) – an efficient lookup index into the coco data structure. The index defines its own attributes like `anns`, `cats`, `imgs`, etc. See `CocoIndex` for more details on which attributes are available.

- **fpath** (*PathLike | None*) – if known, this stores the filepath the dataset was loaded from
- **tag** (*str*) – A tag indicating the name of the dataset.
- **img_root** (*PathLike | None*) – If known, this is the root path that all image file names are relative to. This can also be manually overwritten by the user.
- **hashid** (*str | None*) – If computed, this will be a hash uniquely identifing the dataset. To ensure this is computed see `_build_hashid()`.

References

http://cocodataset.org/#format http://cocodataset.org/#download

CommandLine: python -m kwcoco.coco_dataset CocoDataset –show

Example

```
>>> dataset = demo_coco_data()
>>> self = CocoDataset(dataset, tag='demo')
>>> # xdoctest: +REQUIRES(--show)
>>> self.show_image(gid=2)
>>> from matplotlib import pyplot as plt
>>> plt.show()
```

classmethod from_data(*data*, *img_root=None*)
Constructor from a json dictionary

classmethod from_image_paths(*gpaths*, *img_root=None*)
Constructor from a list of images paths

Example

```
>>> coco_dset = CocoDataset.from_image_paths(['a.png', 'b.png'])
>>> assert coco_dset.n_images == 2
```

classmethod from_coco_paths(*fpaths*, *max_workers=0*, *verbose=1*, *mode='thread'*, *union='try'*)
Constructor from multiple coco file paths.

Loads multiple coco datasets and unions the result

Notes

if the union operation fails, the list of individually loaded files is returned instead.

Parameters

- **fpaths** (*List[str]*) – list of paths to multiple coco files to be loaded and unioned.
- **max_workers** (*int, default=0*) – number of worker threads / processes
- **verbose** (*int*) – verbosity level
- **mode** (*str*) – thread, process, or serial

- **union** (*str | bool, default='try'*) – If True, unions the result datasets after loading. If False, just returns the result list. If 'try', then try to preform the union, but return the result list if it fails.

copy()
Deep copies this object

Example

```
>>> from kwcoco.coco_dataset import *
>>> self = CocoDataset.demo()
>>> new = self.copy()
>>> assert new.imgs[1] is new.dataset['images'][0]
>>> assert new.imgs[1] == self.dataset['images'][0]
>>> assert new.imgs[1] is not self.dataset['images'][0]
```

dumps(*indent=None*, *newlines=False*)
Writes the dataset out to the json format

Parameters **newlines** (*bool*) – if True, each annotation, image, category gets its own line

Notes

Using newlines=True is similar to: print(ub.repr2(dset.dataset, nl=2, trailsep=False)) However, the above may not output valid json if it contains ndarrays.

Example

```
>>> from kwcoco.coco_dataset import *
>>> import json
>>> self = CocoDataset.demo()
>>> text = self.dumps(newlines=True)
>>> print(text)
>>> self2 = CocoDataset(json.loads(text), tag='demo2')
>>> assert self2.dataset == self.dataset
>>> assert self2.dataset is not self.dataset
```

```
>>> text = self.dumps(newlines=True)
>>> print(text)
>>> self2 = CocoDataset(json.loads(text), tag='demo2')
>>> assert self2.dataset == self.dataset
>>> assert self2.dataset is not self.dataset
```

Ignore:

for k in self2.dataset:

if self.dataset[k] == self2.dataset[k]: print('YES: k = {!r}'.format(k))

else: print('NO: k = {!r}'.format(k))

self2.dataset['categories'] self.dataset['categories']

dump(*file*, *indent=None*, *newlines=False*)
Writes the dataset out to the json format

Parameters

- **file** (*PathLike | FileLike*) – Where to write the data. Can either be a path to a file or an open file pointer / stream.
- **newlines** (*bool*) – if True, each annotation, image, category gets its own line.

Example

```
>>> import tempfile
>>> from kwcoco.coco_dataset import *
>>> self = CocoDataset.demo()
>>> file = tempfile.NamedTemporaryFile('w')
>>> self.dump(file)
>>> file.seek(0)
>>> text = open(file.name, 'r').read()
>>> print(text)
>>> file.seek(0)
>>> dataset = json.load(open(file.name, 'r'))
>>> self2 = CocoDataset(dataset, tag='demo2')
>>> assert self2.dataset == self.dataset
>>> assert self2.dataset is not self.dataset
```

```
>>> file = tempfile.NamedTemporaryFile('w')
>>> self.dump(file, newlines=True)
>>> file.seek(0)
>>> text = open(file.name, 'r').read()
>>> print(text)
>>> file.seek(0)
>>> dataset = json.load(open(file.name, 'r'))
>>> self2 = CocoDataset(dataset, tag='demo2')
>>> assert self2.dataset == self.dataset
>>> assert self2.dataset is not self.dataset
```

union(**others*, ***kwargs*)

Merges multiple *CocoDataset* items into one. Names and associations are retained, but ids may be different.

Parameters

- **self** – note that *union()* can be called as an instance method or a class method. If it is a class method, then this is the class type, otherwise the instance will also be unioned with `others`.
- ***others** – a series of CocoDatasets that we will merge
- ****kwargs** – constructor options for the new merged CocoDataset

Returns a new merged coco dataset

Return type *CocoDataset*

Example

```
>>> # Test union works with different keypoint categories
>>> dset1 = CocoDataset.demo('shapes1')
>>> dset2 = CocoDataset.demo('shapes2')
```

(continues on next page)

(continued from previous page)

```
>>> dset1.remove_keypoint_categories(['bot_tip', 'mid_tip', 'right_eye'])
>>> dset2.remove_keypoint_categories(['top_tip', 'left_eye'])
>>> dset_12a = CocoDataset.union(dset1, dset2)
>>> dset_12b = dset1.union(dset2)
>>> dset_21 = dset2.union(dset1)
>>> def add_hist(h1, h2):
>>>     return {k: h1.get(k, 0) + h2.get(k, 0) for k in set(h1) | set(h2)}
>>> kpfreq1 = dset1.keypoint_annotation_frequency()
>>> kpfreq2 = dset2.keypoint_annotation_frequency()
>>> kpfreq_want = add_hist(kpfreq1, kpfreq2)
>>> kpfreq_got1 = dset_12a.keypoint_annotation_frequency()
>>> kpfreq_got2 = dset_12b.keypoint_annotation_frequency()
>>> assert kpfreq_want == kpfreq_got1
>>> assert kpfreq_want == kpfreq_got2
```

```
>>> # Test disjoint gid datasets
>>> import kwcoco
>>> dset1 = kwcoco.CocoDataset.demo('shapes3')
>>> for new_gid, img in enumerate(dset1.dataset['images'], start=10):
>>>     for aid in dset1.gid_to_aids[img['id']]:
>>>         dset1.anns[aid]['image_id'] = new_gid
>>>     img['id'] = new_gid
>>> dset1.index.clear()
>>> dset1._build_index()
>>> # ------
>>> dset2 = kwcoco.CocoDataset.demo('shapes2')
>>> for new_gid, img in enumerate(dset2.dataset['images'], start=100):
>>>     for aid in dset2.gid_to_aids[img['id']]:
>>>         dset2.anns[aid]['image_id'] = new_gid
>>>     img['id'] = new_gid
>>> dset1.index.clear()
>>> dset2._build_index()
>>> others = [dset1, dset2]
>>> merged = kwcoco.CocoDataset.union(*others)
>>> print('merged = {!r}'.format(merged))
>>> print('merged.imgs = {}'.format(ub.repr2(merged.imgs, nl=1)))
>>> assert set(merged.imgs) & set([10, 11, 12, 100, 101]) == set(merged.imgs)
```

```
>>> # Test data is not preserved
>>> dset2 = kwcoco.CocoDataset.demo('shapes2')
>>> dset1 = kwcoco.CocoDataset.demo('shapes3')
>>> others = (dset1, dset2)
>>> cls = self = kwcoco.CocoDataset
>>> merged = cls.union(*others)
>>> print('merged = {!r}'.format(merged))
>>> print('merged.imgs = {}'.format(ub.repr2(merged.imgs, nl=1)))
>>> assert set(merged.imgs) & set([1, 2, 3, 4, 5]) == set(merged.imgs)
```

Todo:

- [ ] are supercategories broken?
- [ ] reuse image ids where possible
- [ ] reuse annotation / category ids where possible
- [ ] disambiguate track-ids

- [x] disambiguate video-ids

---

subset (*gids*, *copy=False*, *autobuild=True*)

Return a subset of the larger coco dataset by specifying which images to port. All annotations in those images will be taken.

Parameters

- **gids** (*List[int]*) – image-ids to copy into a new dataset
- **copy** (*bool, default=False*) – if True, makes a deep copy of all nested attributes, otherwise makes a shallow copy.
- **autobuild** (*bool, default=True*) – if True will automatically build the fast lookup index.

Example

```
>>> self = CocoDataset.demo()
>>> gids = [1, 3]
>>> sub_dset = self.subset(gids)
>>> assert len(self.gid_to_aids) == 3
>>> assert len(sub_dset.gid_to_aids) == 2
```

Example

```
>>> self = CocoDataset.demo()
>>> sub1 = self.subset([1])
>>> sub2 = self.subset([2])
>>> sub3 = self.subset([3])
>>> others = [sub1, sub2, sub3]
>>> rejoined = CocoDataset.union(*others)
>>> assert len(sub1.anns) == 9
>>> assert len(sub2.anns) == 2
>>> assert len(sub3.anns) == 0
>>> assert rejoined.basic_stats() == self.basic_stats()
```

class kwcoco.**CategoryTree**(*graph=None*)

Bases: ubelt.util_mixins.NiceRepr

Wrapper that maintains flat or hierarchical category information.

Helps compute softmaxes and probabilities for tree-based categories where a directed edge (A, B) represents that A is a superclass of B.

Notes

There are three basic properties that this object maintains:

name: Alphanumeric string names that should be generally descriptive. Using spaces and special characters in these names is discouraged, but can be done.

id: The integer id of a category should ideally remain consistent. These are often given by a dataset (e.g. a COCO dataset).

index: Contigous zero-based indices that indexes the list of categories. These should be used for the fastest access in backend computation tasks.

Variables

- **idx_to_node** (*List[str]*) – a list of class names. Implicitly maps from index to category name.
- **id_to_node** (*Dict[int, str]*) – maps integer ids to category names
- **node_to_id** (*Dict[str, int]*) – maps category names to ids
- **node_to_idx** (*Dict[str, int]*) – maps category names to indexes
- **graph** (*nx.Graph*) – a Graph that stores any hierarchy information. For standard mutually exclusive classes, this graph is edgeless. Nodes in this graph can maintain category attributes / properties.
- **idx_groups** (*List[List[int]]*) – groups of category indices that share the same parent category.

Example

```
>>> from kwcoco.category_tree import *
>>> graph = nx.from_dict_of_lists({
>>>     'background': [],
>>>     'foreground': ['animal'],
>>>     'animal': ['mammal', 'fish', 'insect', 'reptile'],
>>>     'mammal': ['dog', 'cat', 'human', 'zebra'],
>>>     'zebra': ['grevys', 'plains'],
>>>     'grevys': ['fred'],
>>>     'dog': ['boxer', 'beagle', 'golden'],
>>>     'cat': ['maine coon', 'persian', 'sphynx'],
>>>     'reptile': ['bearded dragon', 't-rex'],
>>> }, nx.DiGraph)
>>> self = CategoryTree(graph)
>>> print(self)
<CategoryTree(nNodes=22, maxDepth=6, maxBreadth=4...)>
```

Example

```
>>> # The coerce classmethod is the easiest way to create an instance
>>> import kwcoco
>>> kwcoco.CategoryTree.coerce(['a', 'b', 'c'])
<CategoryTree(nNodes=3, nodes=['a', 'b', 'c']) ...
>>> kwcoco.CategoryTree.coerce(4)
<CategoryTree(nNodes=4, nodes=['class_1', 'class_2', 'class_3', ...
>>> kwcoco.CategoryTree.coerce(4)
```

copy()

classmethod from_mutex(*nodes*, *bg_hack=True*)

Parameters **nodes** (*List[str]*) – or a list of class names (in which case they will all be assumed to be mutually exclusive)

Example

```
>>> print(CategoryTree.from_mutex(['a', 'b', 'c']))
<CategoryTree(nNodes=3, ...)>
```

classmethod from_json(*state*)

Parameters **state** (*Dict*) – see __getstate__ / __json__ for details

classmethod from_coco(*categories*)
Create a CategoryTree object from coco categories

Parameters **List[Dict]** – list of coco-style categories

classmethod coerce(*data*, ***kw*)
Attempt to coerce data as a CategoryTree object.

This is primarily useful for when the software stack depends on categories being represent

This will work if the input data is a specially formatted json dict, a list of mutually exclusive classes, or if it is already a CategoryTree. Otherwise an error will be thrown.

Parameters

- **data** (*object*) – a known representation of a category tree.
- ****kwargs** – input type specific arguments

Returns self

Return type *CategoryTree*

Raises

- TypeError - if the input format is unknown
- ValueError - if kwargs are not compatible with the input format

Example

```
>>> import kwcoco
>>> classes1 = kwcoco.CategoryTree.coerce(3)  # integer
>>> classes2 = kwcoco.CategoryTree.coerce(classes1.__json__())  # graph dict
>>> classes3 = kwcoco.CategoryTree.coerce(['class_1', 'class_2', 'class_3'])
↪# mutex list
>>> classes4 = kwcoco.CategoryTree.coerce(classes1.graph)  # nx Graph
>>> classes5 = kwcoco.CategoryTree.coerce(classes1)  # cls
>>> # xdoctest: +REQUIRES(module:ndsampler)
>>> import ndsampler
>>> classes6 = ndsampler.CategoryTree.coerce(3)
>>> classes7 = ndsampler.CategoryTree.coerce(classes1)
>>> classes8 = kwcoco.CategoryTree.coerce(classes6)
```

classmethod demo(*key='coco'*, ***kwargs*)

Parameters **key** (*str*) – specify which demo dataset to use. Can be 'coco' (which uses the default coco demo data). Can be 'btree' which creates a binary tree and accepts kwargs

'r' and 'h' for branching-factor and height.

CommandLine: xdoctest -m ~/code/kwcoco/kwcoco/category_tree.py CategoryTree.demo

Example

```
>>> from kwcoco.category_tree import *
>>> self = CategoryTree.demo()
>>> print('self = {}'.format(self))
self = <CategoryTree(nNodes=10, maxDepth=2, maxBreadth=4...)>
```

to_coco()
Converts to a coco-style data structure

Yields *Dict* – coco category dictionaries

id_to_idx

```
>>> import kwcoco
>>> self = kwcoco.CategoryTree.demo()
>>> self.id_to_idx[1]
```

Type Example

idx_to_id

```
>>> import kwcoco
>>> self = kwcoco.CategoryTree.demo()
>>> self.idx_to_id[0]
```

Type Example

idx_to_ancestor_idxs
memoization decorator for a method that respects args and kwargs

References

http://code.activestate.com/recipes/577452-a-memoize-decorator-for-instance-methods/

Example

```
>>> import ubelt as ub
>>> closure = {'a': 'b', 'c': 'd'}
>>> incr = [0]
>>> class Foo(object):
>>>     @memoize_method
>>>     def foo_memo(self, key):
>>>         value = closure[key]
>>>         incr[0] += 1
>>>         return value
>>>     def foo(self, key):
>>>         value = closure[key]
>>>         incr[0] += 1
>>>         return value
>>> self = Foo()
>>> assert self.foo('a') == 'b' and self.foo('c') == 'd'
```

(continues on next page)

(continued from previous page)

```
>>> assert incr[0] == 2
>>> print('Call memoized version')
>>> assert self.foo_memo('a') == 'b' and self.foo_memo('c') == 'd'
>>> assert incr[0] == 4
>>> assert self.foo_memo('a') == 'b' and self.foo_memo('c') == 'd'
>>> print('Counter should no longer increase')
>>> assert incr[0] == 4
>>> print('Closure changes result without memoization')
>>> closure = {'a': 0, 'c': 1}
>>> assert self.foo('a') == 0 and self.foo('c') == 1
>>> assert incr[0] == 6
>>> assert self.foo_memo('a') == 'b' and self.foo_memo('c') == 'd'
>>> print('Constructing a new object should get a new cache')
>>> self2 = Foo()
>>> self2.foo_memo('a')
>>> assert incr[0] == 7
>>> self2.foo_memo('a')
>>> assert incr[0] == 7
```

idx_to_descendants_idxs
memoization decorator for a method that respects args and kwargs

References

http://code.activestate.com/recipes/577452-a-memoize-decorator-for-instance-methods/

Example

```
>>> import ubelt as ub
>>> closure = {'a': 'b', 'c': 'd'}
>>> incr = [0]
>>> class Foo(object):
>>>     @memoize_method
>>>     def foo_memo(self, key):
>>>         value = closure[key]
>>>         incr[0] += 1
>>>         return value
>>>     def foo(self, key):
>>>         value = closure[key]
>>>         incr[0] += 1
>>>         return value
>>> self = Foo()
>>> assert self.foo('a') == 'b' and self.foo('c') == 'd'
>>> assert incr[0] == 2
>>> print('Call memoized version')
>>> assert self.foo_memo('a') == 'b' and self.foo_memo('c') == 'd'
>>> assert incr[0] == 4
>>> assert self.foo_memo('a') == 'b' and self.foo_memo('c') == 'd'
>>> print('Counter should no longer increase')
>>> assert incr[0] == 4
>>> print('Closure changes result without memoization')
>>> closure = {'a': 0, 'c': 1}
>>> assert self.foo('a') == 0 and self.foo('c') == 1
>>> assert incr[0] == 6
```

(continues on next page)

(continued from previous page)

```
>>> assert self.foo_memo('a') == 'b' and self.foo_memo('c') == 'd'
>>> print('Constructing a new object should get a new cache')
>>> self2 = Foo()
>>> self2.foo_memo('a')
>>> assert incr[0] == 7
>>> self2.foo_memo('a')
>>> assert incr[0] == 7
```

idx_pairwise_distance
memoization decorator for a method that respects args and kwargs

References

http://code.activestate.com/recipes/577452-a-memoize-decorator-for-instance-methods/

Example

```
>>> import ubelt as ub
>>> closure = {'a': 'b', 'c': 'd'}
>>> incr = [0]
>>> class Foo(object):
>>>     @memoize_method
>>>     def foo_memo(self, key):
>>>         value = closure[key]
>>>         incr[0] += 1
>>>         return value
>>>     def foo(self, key):
>>>         value = closure[key]
>>>         incr[0] += 1
>>>         return value
>>> self = Foo()
>>> assert self.foo('a') == 'b' and self.foo('c') == 'd'
>>> assert incr[0] == 2
>>> print('Call memoized version')
>>> assert self.foo_memo('a') == 'b' and self.foo_memo('c') == 'd'
>>> assert incr[0] == 4
>>> assert self.foo_memo('a') == 'b' and self.foo_memo('c') == 'd'
>>> print('Counter should no longer increase')
>>> assert incr[0] == 4
>>> print('Closure changes result without memoization')
>>> closure = {'a': 0, 'c': 1}
>>> assert self.foo('a') == 0 and self.foo('c') == 1
>>> assert incr[0] == 6
>>> assert self.foo_memo('a') == 'b' and self.foo_memo('c') == 'd'
>>> print('Constructing a new object should get a new cache')
>>> self2 = Foo()
>>> self2.foo_memo('a')
>>> assert incr[0] == 7
>>> self2.foo_memo('a')
>>> assert incr[0] == 7
```

is_mutex()
Returns True if all categories are mutually exclusive (i.e. flat)

If true, then the classes may be represented as a simple list of class names without any loss of information, otherwise the underlying category graph is necessary to preserve all knowledge.

Todo:

- [ ] what happens when we have a dummy root?

---

num_classes

class_names

category_names

cats

Returns a mapping from category names to category attributes.

If this category tree was constructed from a coco-dataset, then this will contain the coco category attributes.

Returns Dict[str, Dict[str, object]]

Example

```
>>> from kwcoco.category_tree import *
>>> self = CategoryTree.demo()
>>> print('self.cats = {!r}'.format(self.cats))
```

index(*node*)

Return the index that corresponds to the category name

show()

Ignore:

```
>>> import kwplot
>>> kwplot.autompl()
>>> from kwcoco import category_tree
>>> self = category_tree.CategoryTree.demo()
>>> self.show()
```

python -c "import kwplot, kwcoco, graphid; kwplot.autompl(); graphid.util.show_nx(kwcoco.category_tree.CategoryTree.demo().graph); kwplot.show_if_requested()" –show

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

k

Index

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

X