




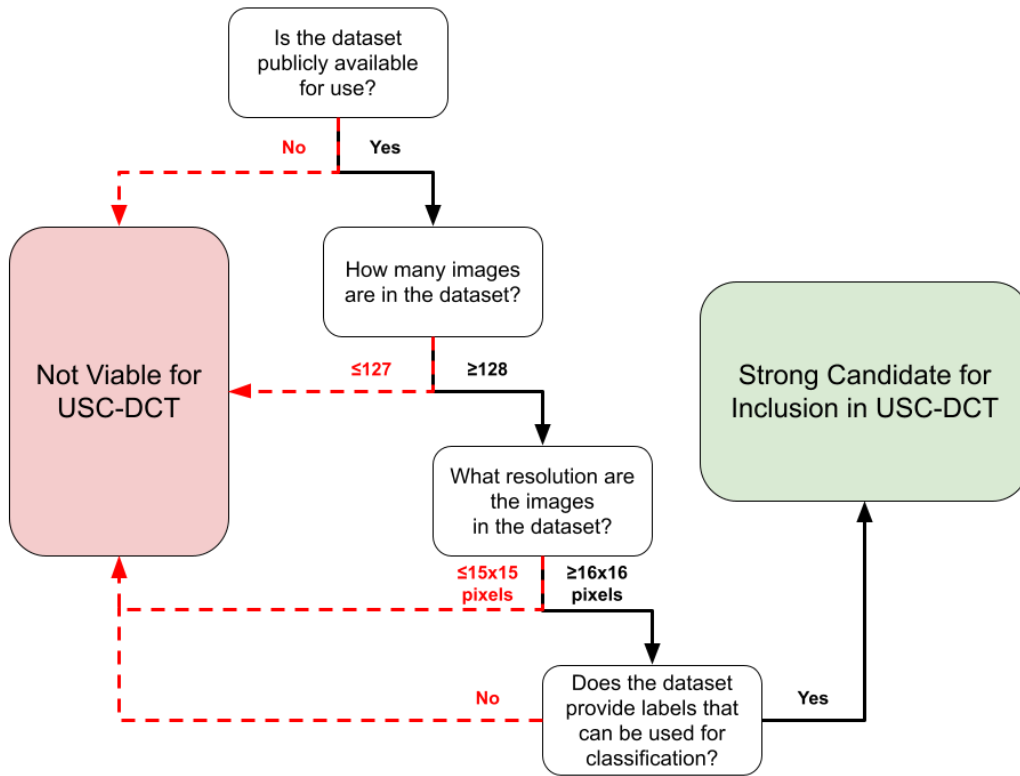


# Supplementary Materials: USC-DCT: A Collection of Diverse Classification Tasks

Adam M. Jones <sup>1,†</sup> , Gozde Sahin <sup>2,†</sup> , Zachary W. Murdock <sup>1,†</sup> , Yunhao Ge <sup>2</sup> , Ao Xu <sup>2</sup>, Yuecheng Li <sup>2</sup>, Di Wu <sup>2</sup>, Shuo Ni <sup>2</sup>, Po-Hsuan Huang <sup>1</sup>, Kiran Lekkala <sup>2</sup> and Laurent Itti <sup>1,2,\*</sup> 

## S.1. Dataset Inclusion Criteria Decision Tree



**Figure S1.** Visualization of the decision process used to determine if a computer vision dataset could be included in the development of USC-DCT, as mentioned in Section 3.1.1.

## S.2. Dataset Preparation Examples

As mentioned above, a *prepare\_dataset.py* script is created for each dataset during the inspection phase. Most of the work to "ingest" the dataset is done by a function called *parse\_dataset()*. In this function, the extracted files and any other knowledge are combined to create a list of every image file with its details and a dictionary of all classes. Many of the datasets have straightforward parsing. However, some are quite complex (not including the splitting up of the super-datasets: iNaturalist and Office-Home).

### S.2.1. Simple Example

One simple example of our task-specific scripts to integrate a task into USC-DCT is the Concrete Cracks dataset. Listing 1 includes the main processing function for this dataset, and shows a straightforward example of how a user would customize *parse\_dataset()* to add a task to USC-DCT. The script first creates a list of all the image files found in the *original\_files* folder where the downloaded archive for the dataset was extracted to. These images are processed using the method given in Section 3 to set status codes for each image (*problem\_value* and the set (train/val/test)). Images with non-zero status codes are excluded from the collection by setting their class and set values to -1.

```

1 def parse_dataset (
2     dataset_root_path: str,
3 ) -> tuple[list[DatasetImage], dict[str, int]]:
4     """This returns a list of all images within the dataset.
5     This includes **all images** in both ./original_files/ and,
6     if necessary, ./converted_files/
7     """
8
9     image_list: list[DatasetImage] = []
10    class_dict: dict[str, int] = {}
11
12    # get all file paths
13    files_list = get_all_files_in_directory (
14        dataset_root_path + "original_files/", dataset_root_path)
15
16    (image_path_list, _) = filter_file_list (files_list)
17
18    # create a class dict
19    class_list = os.listdir(dataset_root_path + "original_files/")
20    for k, class_name in enumerate(sorted(class_list, key=str.casefold)):
21        class_dict[class_name] = k
22
23    for image_path in image_path_list:
24        problem_value, set_value = 0, 0
25
26        # first address problem_value
27        if not is_image_valid(dataset_root_path + image_path):
28            problem_value = 1
29
30        # check non-dataset file
31
32        # once problem_value != 0, set all else as -1
33        if problem_value != 0:
34            class_int = -1
35            set_value = -1
36        else:
37            # finally address the class_int
38            class_from_path = image_path.split("/")[-2]
39            class_int = class_dict[class_from_path]
40
41            image_list.append (
42                DatasetImage (
43                    relative_path=image_path,
44                    class_id = class_int,
45                    set_id = set_value,
46                    problem=problem_value,
47                )
48            )
49
50    return (image_list, class_dict)

```

Code Snippet. 1: A simple example of a `prepare_dataset` function.

In addition to the sample `parse_dataset()`, we also provide a sample output that a human validator would obtain from the automated validation script for this dataset. Listing 2 shows this output, which notes important information like total images found in the original archives, number of images assigned to each set, and class statistics. The validation script also presents statistics on status codes across images and whether certain status codes can be independently confirmed (exact duplicate folders etc.). This information can be utilized by the human validator to make sure the customized script is running as expected.

```

1 verify md5sums
2 extract dataset
3 verify all archive files are used (this will be a manual check):
4 ##### archive.zip ##### #
5 #####
6 DATASET_FILE_ 1 = " archive . zip "
7 # #####
8 parse dataset
9 validate parsed dataset
10 validate classes dict
11 validate images list
12 get all remaining image details
13 get image details using 26 workers...
14 took 9 sec
15 # #####
16 statistics:
17 image statistics:

```

```

18 images total : 40000
19 images found independently : 40000
20 inoriginal_files: 40000
21 problem-free images: 40000
22 imagesintrainingset: 40000
23 imagesinvalidationset: 0
24 imagesintestingset: 0
25 problematic images : 0
26 class_id=-1 images: 0
27 set_id=-1 images: 0
28 all3 (problem/class_id/set_id) images: 0
29 classstatistics:
30 classes (>=0): 2
31 first 2 classes:
32   -id- , - name - , - count -
33     0 , Negative , 20000
34     1 , Positive , 20000
35 classnamesinalphabetical order :
36 True classids are contiguous : True
37 shortestclassname : Negative
38 longestclassname : Negative
39 allclasses are the same size:
40                               20000
41 #####
42 create database.sqlite
43 run additional checks on hashes
44 image count: 40000 folder count:2
45 there are no folders full of images that are duplicated elsewhere
46 there are 1519 ( problem =0) hashes that aren't unique
47 create examples

```

Code Snippet. 2: A simple example of what the output from the automated validation script looks like.

### S.2.2. Complex Example

A slightly more complex (but not too complex) example is from the Oxford Buildings dataset. Here, the folder structure was less straightforward, which necessitated some additional code. It bears mentioning that for some of the datasets, the `parse_dataset()` function runs into hundreds of lines, and sometimes requires the use of additional files that act as keys for the class structure.

```

1 def parse_dataset(
2     dataset_root_path: str,
3 ) -> tuple[list[DatasetImage], dict[str, int]]:
4     """ This returns a list of all images within the dataset .
5     This includes ** all images ** in both ./ original_files / and ,
6     if necessary, ./converted_files/
7     """
8
9     image_list: list[DatasetImage] = []
10    class_dict: dict[str, int] = {}
11
12
13    # get all file paths
14    files_list = get_all_files_in_directory(
15        dataset_root_path + "original_files/", dataset_root_path
16    )
17
18    # splitting the file list into images and non - images(
19    image_path_list, text_path_list) = filter_file_list(files_list)
20
21    # generate list of classes AND list of file labels
22    class_list = []
23    text2img_dict: dict[str, int] = {}
24    for filePath in text_path_list:
25        # remove all filepaths, break at counting number
26        # (pattern: .../.../LABEL_#_...)
27        className = re.split(r"\d+", filePath.replace("original_files/", "")[0][-1])
28        # Collect Unique Class Names
29        if className not in class_list:
30            class_list.append(className)
31
32    # creating class_dict, and text2img_dict
33    for id, classStr in enumerate(sorted(class_list, key=str.casefold)):
34        class_dict.update({classStr: id})
35
36    # Create list of files denoted for the given classStr
37    textLabels = []
38    for fileText in sorted(text_path_list):
39        # If class is in file name (and not a "Querv" file).

```

























































```

39         # add to collection, else ignore
40         if classStr in fileText and "query" not in fileText:
41             for x in (
42                 open(str(dataset_root_path + fileText), "r").read().split("\n")[:-1]
43             ):
44                 textLabels.append(x)
45         else:
46             pass
47         # Generate dict with the image name: id#
48         n = 0
49         for txtImg in textLabels:
50             n += 1
51             text2img_dict.update({"original_files/" + txtImg + ".jpg": id})
52
53     # go through each image, ensure valid, flag any problems, else find label
54     for image_path in sorted(image_path_list):
55         problem_value, set_value = 0, 0
56
57         # first address problem_value
58         if not is_image_valid (dataset_root_path + image_path):
59             problem_value = 1
60             class_int = -1
61
62         # If not in listed named files
63         elif image_path not in text2img_dict.keys():
64             class_int = -1
65             problem_value = 4
66         else:
67             class_int = text2img_dict[image_path]
68
69         # once problem_value != 0, set all else as -1
70         if problem_value != 0:
71             class_int = -1
72             set_value = -1
73
74         assert isinstance(class_int, int)
75
76         image_list.append(
77             DatasetImage(
78                 relative_path=image_path,
79                 class_id=class_int,
80                 set_id=set_value,
81                 problem=problem_value,
82             )
83         )
84
85     return (image_list, class_dict)

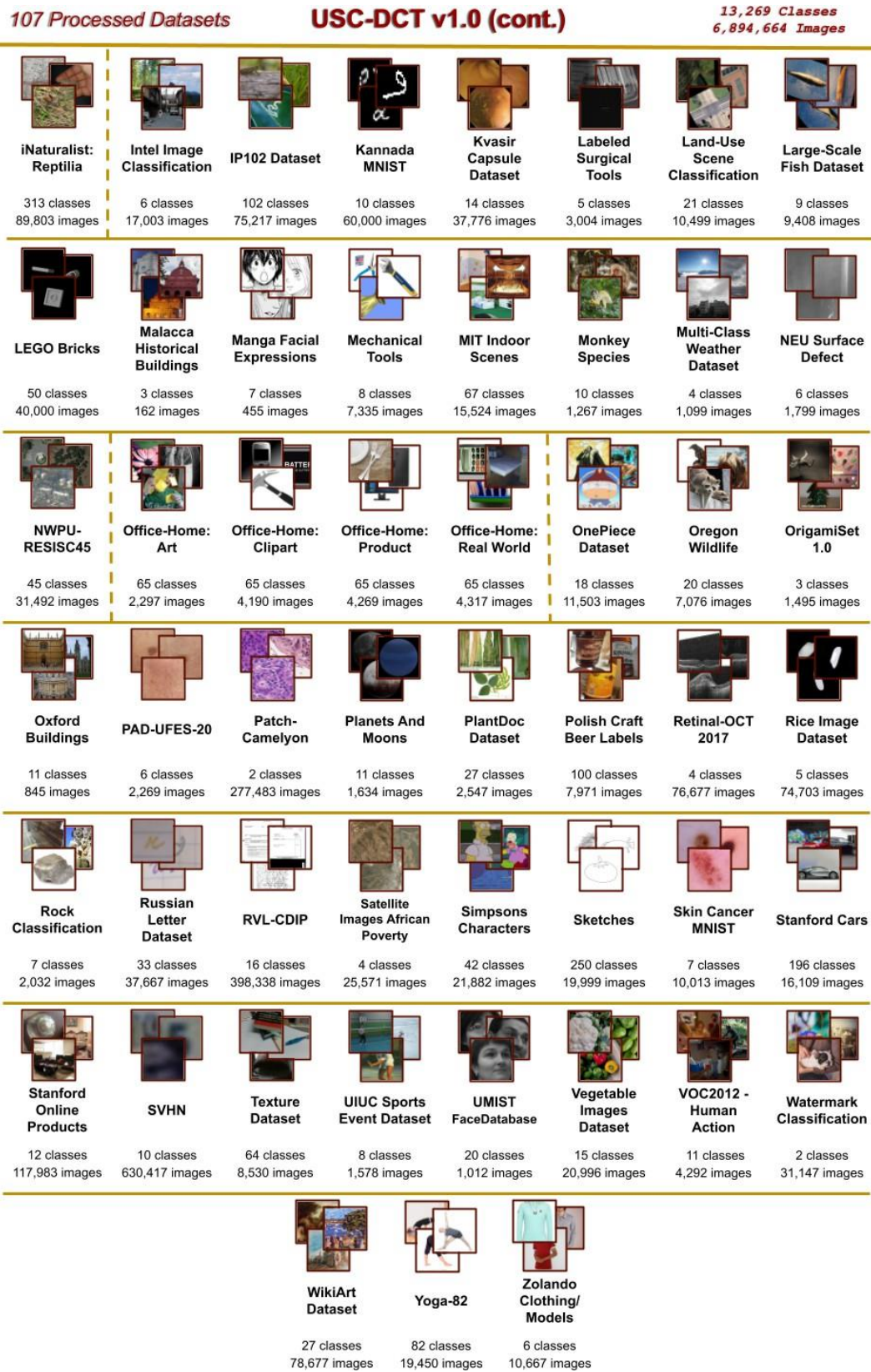
```

Code Snippet. 3: A more-complex example example of a prepare\_dataset function.

### S.3. USC-DCT Visual Overview

107 Processed Datasets				USC-DCT v1.0				13,269 Classes 6,894,664 Images			
											
<b>100 Sports</b>	<b>7000 Pokemon</b>	<b>Apparel Images</b>	<b>APTOS 2019</b>	<b>Art Images Type</b>	<b>ASL Alphabet</b>	<b>Blood Cell Images</b>	<b>Boat Types</b>				
100 classes 14,558 images	150 classes 6,803 images	24 classes 11,372 images	196 classes 16185 images	5 classes 7,110 images	29 classes 87,000 images	4 classes 12,513 images	9 classes 1,460 images				
											
<b>Book Covers 30</b>	<b>Brain Tumor Dataset</b>	<b>Brazilian Coins</b>	<b>Breast Ultrasound</b>	<b>Cataract Dataset</b>	<b>CelebA</b>	<b>Chars74k</b>	<b>Chest X-Ray</b>				
30 classes 56,975 images	4 classes 2,870 images	5 classes 3,059 images	3 classes 778 images	4 classes 601 images	5 classes 124,803 images	62 classes 11,202 images	2 classes 5,824 images				
											
<b>CLEVR v1.0</b>	<b>Colorectal Histology MNIST</b>	<b>Concrete Cracks</b>	<b>CORe50</b>	<b>CUB-200</b>	<b>DeepVP-1M</b>	<b>DeepWeedsX</b>	<b>DermNet Dataset</b>				
8 classes 85,000 images	8 classes 5,000 images	2 classes 38,402 images	10 classes 163,006 images	200 classes 11,787 images	9 classes 74,288 images	9 classes 17,508 images	23 classes 17,826 images				
											
<b>Describable Textures</b>	<b>Diabetic Retinopathy</b>	<b>Dragon Ball Super Saiyan</b>	<b>Electronic Components</b>	<b>EuroSAT</b>	<b>FaceMask Dataset</b>	<b>Facial Expression 2013</b>	<b>Fashion Product Images</b>				
47 classes 5,623 images	5 classes 35,126 images	6 classes 148 images	36 classes 10,153 images	10 classes 27,000 images	3 classes 13,755 images	7 classes 33,977 images	43 classes 43,653 images				
											
<b>Fine-Grained Aircraft</b>	<b>Flowers</b>	<b>Food-101</b>	<b>Freiburg Groceries</b>	<b>Galaxy10</b>	<b>Garbage Classification</b>	<b>GTSRB</b>	<b>HistAerial</b>				
70 classes 10,000 images	102 classes 8,185 images	101 classes 100,938 images	25 classes 4,933 images	10 classes 17,615 images	12 classes 15,493 images	43 classes 51,831 images	7 classes 137,427 images				
											
<b>House Room Images</b>	<b>Hurricane Damage</b>	<b>iFood2019</b>	<b>iLab 80m</b>	<b>iLab Atari</b>	<b>iMaterialist Fashion 2019</b>	<b>iNaturalist: Actinopterygii</b>	<b>iNaturalist: Amphibia</b>				
5 classes 5,174 images	2 classes 21,050 images	251 classes 130,468 images	15 classes 15,000 images	67 classes 368,870 images	46 classes 45,191 images	183 classes 46,641 images	170 classes 47,850 images				
											
<b>iNaturalist: Animalia</b>	<b>iNaturalist: Arachnida</b>	<b>iNaturalist: Aves</b>	<b>iNaturalist: Fungi</b>	<b>iNaturalist: Insecta</b>	<b>iNaturalist: Mammalia</b>	<b>iNaturalist: Mollusca</b>	<b>iNaturalist: Plantae</b>				
142 classes 38,349 images	153 classes 42,164 images	1,486 classes 428,775 images	341 classes 93,359 images	2,526 classes 687,823 images	246 classes 71,276 images	169 classes 46,239 images	4,271 classes 1,189,798 images				





**Figure S2.** Visual Overview of the Datasets used within USC-DCT. Dashed-Lines represent data subsets created by splitting one larger dataset (e.g. iNaturalist). See also: Table 4 in the main paper.