Download a the dataset sample from Supervisely

```
#@title Download a the dataset sample from Supervisely
!curl https://assets.supervisely.com/supervisely-supervisely-assets-public/teams_storage/W/6/pt/ANAGVgKaC62tTrDQWK5JhNP2dd8ynqaTKSM1QdVoA
```

```
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  548M  100  548M    0     0  8892k      0  0:01:03  0:01:03 --:--:-- 12.7M
```

Extract the tarball 'mvtec.tar' and look for test images for the hazelnut subset

```
#@title Extract the tarball 'mvtec.tar' and look for test images for the hazelnut subset
!tar -xf mvtec.tar
!ls test/img/hazelnut_*
```

```
tar: Removing leading `/' from member names
test/img/hazelnut_crack_002.png  test/img/hazelnut_print_004.png
test/img/hazelnut_cut_003.png    test/img/hazelnut_print_005.png
test/img/hazelnut_good_023.png   test/img/hazelnut_print_006.png
test/img/hazelnut_good_037.png   test/img/hazelnut_print_007.png
test/img/hazelnut_hole_005.png   test/img/hazelnut_print_009.png
test/img/hazelnut_hole_010.png   test/img/hazelnut_print_011.png
test/img/hazelnut_hole_013.png   test/img/hazelnut_print_012.png
test/img/hazelnut_hole_016.png   test/img/hazelnut_print_013.png
test/img/hazelnut_print_003.png  test/img/hazelnut_print_016.png
```

```
!git clone https://github.com/akridata-ai/ZS-CLIP-AC-naive.git
```

```
Cloning into 'ZS-CLIP-AC-naive'...
remote: Enumerating objects: 16, done.
remote: Counting objects: 100% (16/16), done.
remote: Compressing objects: 100% (14/14), done.
remote: Total 16 (delta 2), reused 12 (delta 0), pack-reused 0 (from 0)
Receiving objects: 100% (16/16), done.
Resolving deltas: 100% (2/2), done.
```

```
%cd ZS-CLIP-AC-naive
```

```
/content/ZS-CLIP-AC-naive/ZS-CLIP-AC-naive
```

```
!git branch -a
```

```
* main
  remotes/origin/HEAD -> origin/main
  remotes/origin/feature/template-code
  remotes/origin/main
```

```
!git checkout -b feature/template-code origin/feature/template-code
```

```
Branch 'feature/template-code' set up to track remote branch 'feature/template-code' from 'origin'.
Switched to a new branch 'feature/template-code'
```

```
!git pull
```

```
Already up to date.
```

```
!git config --global user.email "amalamahalakshmip@email.com"
!git config --global user.name "amalamahalakshmi"
```

```
%cd /content/ZS-CLIP-AC-naive
```

```
/content/ZS-CLIP-AC-naive
```

```
!ls
```

```
clip_ac.py  meta.json    README.md         test
data        mvtec.tar    requirements.txt  train
LICENSE.md  __pycache__  spec.py           ZS-CLIP-AC-naive
```

```
!cp /content/test/img/hazelnut_* data/
```

```
!pip install -qr requirements.txt
```

```
  Preparing metadata (setup.py) ... done
```

```
%%writefile spec.py
"""
Spec for zero-shot defect classification with CLIP.
"""
from pydantic import BaseModel
from typing import List, Dict

class DefectClassificationSpec(BaseModel):
    # final labels you want in the confusion matrix (order is fixed)
    class_names: List[str] = ["good", "print", "hole", "cut", "crack"]

    # base CLIP model (try ViT-B/16 first; switch to ViT-L/14 if GPU has room)
    model_name: str = "ViT-B/16"

    # two-stage: first 'good' vs 'defect', then if defect -> {print, hole, cut, crack}
    use_two_stage: bool = True

    # simple test-time augmentation (original + horizontal flip)
    use_tta: bool = True

    # temperature (logit scaling); 50–100 works well for CLIP
    temperature: float = 100.0

    # if two-stage, confidence margin to call something 'good'
    # larger = stricter 'good' (reduces false goods)
    good_margin: float = 0.0

    # natural-language phrases per class (richer semantics than single words)
    phrases: Dict[str, List[str]] = {
        "good": [
            "a clean, defect-free hazelnut",
            "an undamaged hazelnut",
            "a perfect normal hazelnut without any defects",
            "an intact hazelnut with smooth surface",
            "a pristine hazelnut"
        ],
        "print": [
            "a hazelnut with a print defect",
            "a hazelnut with printed text on the shell",
            "a hazelnut with a stamped marking",
            "a hazelnut with surface printing marks",
            "a hazelnut with an ink print on the shell"
        ],
        "hole": [
            "a hazelnut with a hole defect",
            "a hazelnut with a drilled hole",
            "a hazelnut with a punctured hole in the shell",
            "a hazelnut with a perforated shell",
            "a hazelnut with a hole on the surface"
        ],
        "cut": [
            "a hazelnut with a cut defect",
            "a sliced hazelnut",
            "a hazelnut with a cut surface",
            "a hazelnut cut in half",
            "a hazelnut showing a clean cut"
        ],
        "crack": [
            "a hazelnut with a crack defect",
            "a cracked hazelnut shell",
            "a broken hazelnut with a visible crack",
            "a hazelnut shell split open",
            "a hazelnut showing a fracture"
        ],
    }

    # extra templates (from CLIP/ImageNet style) — multiplied with phrases
    templates: List[str] = [
        "a photo of {}",
        "a close-up photo of {}",
        "a cropped photo of {}",
        "a bright photo of {}",
        "a low-light photo of {}",
        "a high-resolution photo of {}",
        "a clean product photo of {}",
        "an image of {}",
        "a centered photo of {}",
        "a detailed photo of {}",
        "a studio photo of {}",
        "a DSLR photo of {}",
    ]
    # prompts for stage-1 binary decision
    stage1_good_phrases: List[str] = [
        "a clean, defect-free hazelnut",
        "a normal undamaged hazelnut",
        "a pristine hazelnut with no defects",
    ]
    stage1_defect_phrases: List[str] = [
        "a defective or damaged hazelnut",
        "a hazelnut with some kind of defect",
        "a faulty hazelnut with visible damage",
    ]
```

```
Overwriting spec.py
```

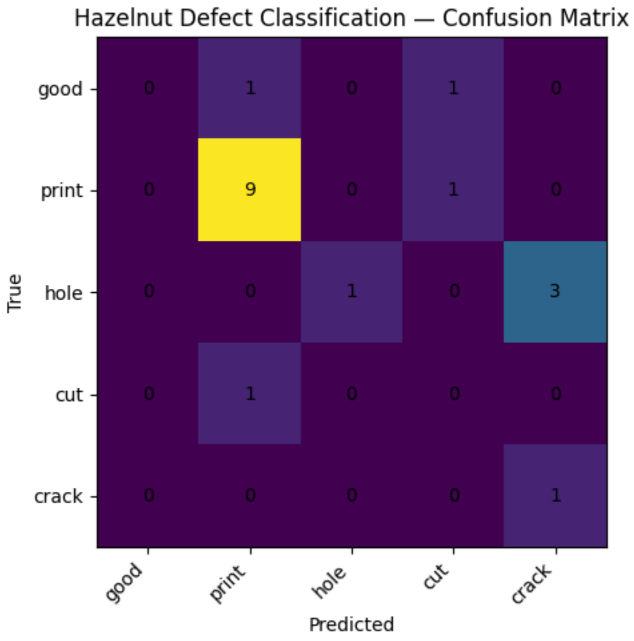
```
%%writefile clip_ac.py
"""
Zero-shot defect classification using CLIP with:
- prompt ensembling over templates & phrases
- optional two-stage head (good vs defect, then defect type)
- light TTA (horizontal flip)
"""
from pathlib import Path
from typing import List, Tuple, Dict

import clip
import torch
from PIL import Image, ImageOps

from spec import DefectClassificationSpec

def _build_prompts(phrases: List[str], templates: List[str]) -> List[str]:
    # e.g., template="a photo of {}", phrase="a hazelnut with a crack defect"
    return [tpl.format(ph) for tpl in templates for ph in phrases]

@torch.no_grad()
def _encode_prompt_set(model, device, prompts: List[str]) -> torch.Tensor:
    # tokenize, encode, L2-normalize, then average across prompts
    tokens = clip.tokenize(prompts).to(device)
    feats = model.encode_text(tokens)
    feats = feats / feats.norm(dim=-1, keepdim=True)
    mean = feats.mean(dim=0, keepdim=True)
    mean = mean / mean.norm(dim=-1, keepdim=True)
    return mean  # [1, d]

@torch.no_grad()
def _encode_classes(model, device, class_names: List[str],
                    phrases_map: Dict[str, List[str]],
                    templates: List[str]) -> torch.Tensor:
    # returns [C, d] class text embeddings
    cls_feats = []
    for cls in class_names:
        phrases = phrases_map[cls]
        prompts = _build_prompts(phrases, templates)
        mean = _encode_prompt_set(model, device, prompts)
        cls_feats.append(mean)
    return torch.cat(cls_feats, dim=0)

def _parse_label(fname: str) -> str:
    # hazelnut_good_023.png -> good; hazelnut_crack_002.png -> crack
    name = fname.lower()
    if "good" in name:
        return "good"
    parts = name.split("_")
    if len(parts) >= 2:
        return parts[1].replace(".png", "")
    return "good"

@torch.no_grad()
def classify_defects(spec: DefectClassificationSpec, test_dir: Path) -> Tuple[List[str], List[str]]:
    device = "cuda" if torch.cuda.is_available() else "cpu"

    # load model with gentle fallback
    try:
        model, preprocess = clip.load(spec.model_name, device=device)
    except RuntimeError:
        try:
            model, preprocess = clip.load("ViT-B/16", device=device)
        except RuntimeError:
            model, preprocess = clip.load("ViT-B/32", device=device)
    model.eval()
    # ---------- text features ----------
    # stage 1: good vs defect (binary)
    if spec.use_two_stage:
        stage1_names = ["good", "defect"]
        stage1_phrases = {
            "good": spec.stage1_good_phrases,
            "defect": spec.stage1_defect_phrases,
        }
        stage1_text = _encode_classes(model, device, stage1_names, stage1_phrases, spec.templates)  # [2, d]

    # stage 2: per-defect classes
    text_features = _encode_classes(model, device, spec.class_names, spec.phrases, spec.templates)  # [C, d]

    y_true, y_pred = [], []

    for img_path in sorted(test_dir.glob("*.png")):
        true_label = _parse_label(img_path.name)
        y_true.append(true_label)

        # --------- image features with TTA ---------
        img = Image.open(img_path).convert("RGB")
        images = [img]
        if spec.use_tta:
            images.append(ImageOps.mirror(img))

        img_feats = []
        for im in images:
            image = preprocess(im).unsqueeze(0).to(device)
            feat = model.encode_image(image)
            feat = feat / feat.norm(dim=-1, keepdim=True)
            img_feats.append(feat)
        image_feat = torch.stack(img_feats, dim=0).mean(dim=0)  # [1, d]
        image_feat = image_feat / image_feat.norm(dim=-1, keepdim=True)

        # --------- two-stage decision (optional) ---------
        if spec.use_two_stage:
            logits_bin = spec.temperature * (image_feat @ stage1_text.T)  # [1, 2]
            probs_bin = logits_bin.softmax(dim=-1).squeeze(0)
            prob_good = float(probs_bin[0].item())
            prob_defect = float(probs_bin[1].item())

            # margin gate: call good only if clearly more probable than defect
            if (prob_good - prob_defect) >= spec.good_margin:
                y_pred.append("good")
                continue  # no need stage-2 if good

        # --------- stage-2: multi-class among spec.class_names ---------
        logits = spec.temperature * (image_feat @ text_features.T)  # [1, C]
        probs = logits.softmax(dim=-1).squeeze(0)
        pred_idx = int(probs.argmax().item())
        y_pred.append(spec.class_names[pred_idx])

    return y_true, y_pred
```

```
Overwriting clip_ac.py
```

```
from pathlib import Path
from spec import DefectClassificationSpec
from clip_ac import classify_defects

# 3.1 run classification
spec = DefectClassificationSpec(
    # try a stronger backbone if you have GPU memory:
    # model_name="ViT-L/14",
    use_two_stage=True,
    use_tta=True,
    temperature=100.0,
    good_margin=0.0,  # you can try 0.05–0.15 to be stricter for 'good'
)
y_true, y_pred = classify_defects(spec, Path("data"))

# 3.2 evaluation
from sklearn.metrics import confusion_matrix, classification_report
import numpy as np
import matplotlib.pyplot as plt

labels = spec.class_names
cm = confusion_matrix(y_true, y_pred, labels=labels)
acc = (np.trace(cm)/cm.sum()) if cm.sum() > 0 else 0.0
print("Labels order:", labels)
print(f"Accuracy: {acc*100:.2f}%\n")
print(classification_report(y_true, y_pred, labels=labels, zero_division=0))
```

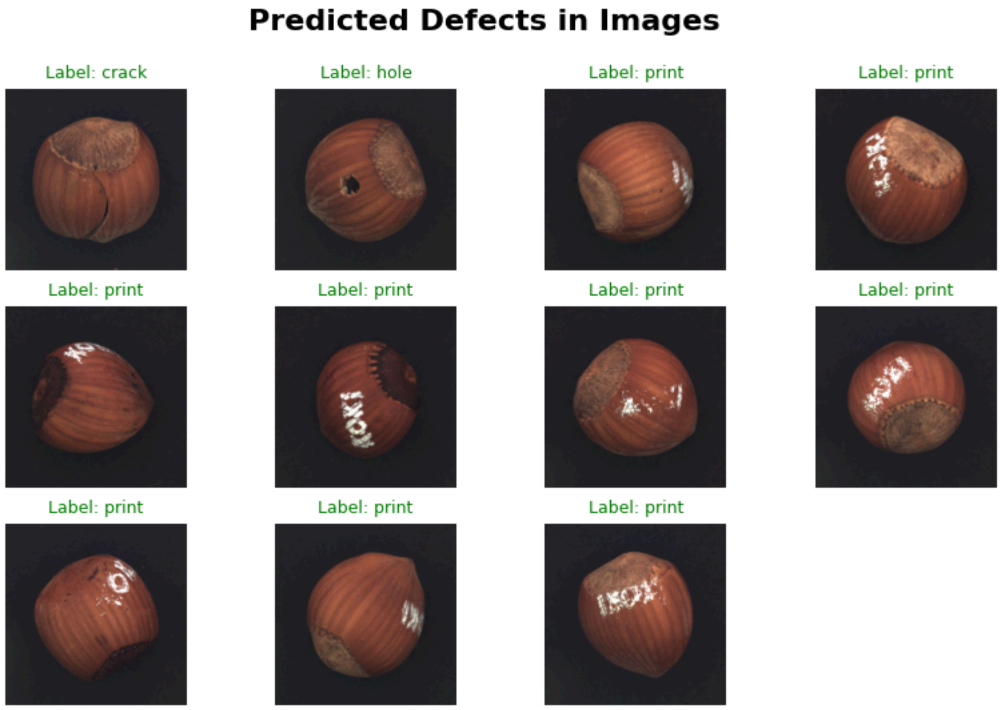
```
# 3.3 plot confusion matrix
fig = plt.figure(figsize=(6,5))
plt.imshow(cm, interpolation='nearest')
plt.title("Hazelnut Defect Classification — Confusion Matrix")
plt.xlabel("Predicted")
plt.ylabel("True")
plt.xticks(range(len(labels)), labels, rotation=45, ha="right")
plt.yticks(range(len(labels)), labels)
for i in range(len(labels)):
    for j in range(len(labels)):
        plt.text(j, i, cm[i, j], ha="center", va="center")
plt.tight_layout()
plt.show()

# 3.4 peek a few predictions
for f, t, p in list(zip(sorted(Path("data").glob("*.png")), y_true, y_pred))[:10]:
    print(f"{f.name:30}  true={t:5}  pred={p:5}")
```

```
Labels order: ['good', 'print', 'hole', 'cut', 'crack']
Accuracy: 61.11%

              precision    recall  f1-score   support

        good       0.00      0.00      0.00         2
       print       0.82      0.90      0.86        10
        hole       1.00      0.25      0.40         4
         cut       0.00      0.00      0.00         1
       crack       0.25      1.00      0.40         1

    accuracy                           0.61        18
   macro avg       0.41      0.43      0.33        18
weighted avg       0.69      0.61      0.59        18
```



```
hazelnut_crack_002.png          true=crack  pred=crack
hazelnut_cut_003.png            true=cut    pred=print
hazelnut_good_023.png           true=good   pred=cut
hazelnut_good_037.png           true=good   pred=print
hazelnut_hole_005.png           true=hole   pred=hole
hazelnut_hole_010.png           true=hole   pred=crack
hazelnut_hole_013.png           true=hole   pred=crack
hazelnut_hole_016.png           true=hole   pred=crack
hazelnut_print_003.png          true=print  pred=print
hazelnut_print_004.png          true=print  pred=print
```

```
import matplotlib.pyplot as plt
from PIL import Image
from pathlib import Path

# filter only correct predictions
correct = [(f, t, p) for f, t, p in zip(sorted(Path("data").glob("*.png")), y_true, y_pred) if t == p]

def show_correct(samples):
    plt.figure(figsize=(10, 6))
    plt.suptitle("Predicted Defects in Images", fontsize=16, fontweight="bold")
    for i, (f, t, p) in enumerate(samples[:12]):  # show max 12
        img = Image.open(f)
        plt.subplot(3, 4, i+1)
        plt.imshow(img)
        plt.axis("off")
        plt.title(f"Label: {t}", fontsize=9, color="green")
    plt.show()

# Show only correct predictions
show_correct(correct)
```



Start coding or generate with AI.