```python
"""
Script: log_conll2003_stats_to_wandb.py
Purpose:
 Load the HuggingFace CoNLL-2003 dataset (eriktks/conll2003),
 compute dataset statistics (samples + entity distribution),
 and log them to Weights & Biases project "Q1-weak-supervision-ner".
"""

# ============================
# Install dependencies first
# pip install datasets wandb
# ============================

import os
import json
import tempfile
from collections import Counter
from pathlib import Path

import matplotlib.pyplot as plt
from datasets import load_dataset
import wandb


# 🔐 Insert your Weights & Biases API Key here:
WANDB_API_KEY = ""
def wandb_login():
   """Login to W&B using provided API key."""
   if WANDB_API_KEY == "your_key_here":
       raise ValueError("Please replace WANDB_API_KEY with your actual W&B
key.")
   os.environ["WANDB_API_KEY"] = WANDB_API_KEY
   wandb.login(key=WANDB_API_KEY)





def compute_entity_counts(split_dataset):
   """
   Given a HF dataset split with 'ner_tags', return:
     - samples: number of examples
     - tokens: total tokens
     - entity_counts_by_tag: Counter of tag-name -> token count (e.g.,
'B-PER', 'I-PER', 'O', ...)
   """
   features = split_dataset.features
   if "ner_tags" not in features:
       raise ValueError("Split does not contain 'ner_tags' feature.")
   label_names = features["ner_tags"].feature.names  # maps id -> tag name

   entity_counts = Counter()
   total_tokens = 0
```

```python
    for example in split_dataset:
        tags = example["ner_tags"]
        total_tokens += len(tags)
        for tag_id in tags:
            entity_counts[label_names[tag_id]] += 1

    return {
        "samples": len(split_dataset),
        "tokens": total_tokens,
        "entity_counts_by_tag": entity_counts,
    }


def aggregate_coarse(entity_counts_by_tag):
    """
    Convert B-/I- style tags into coarse entity categories: PER, LOC, ORG,
MISC, and O.
    Returns Counter with keys: PER, LOC, ORG, MISC, O
    """
    coarse = Counter()
    for tag, count in entity_counts_by_tag.items():
        if tag == "O":
            coarse["O"] += count
        else:
            # tag like "B-PER" or "I-LOC"
            parts = tag.split("-", 1)
            ent = parts[-1] if len(parts) > 1 else tag
            if ent in {"PER", "LOC", "ORG", "MISC"}:
                coarse[ent] += count
            else:
                # map unknown to MISC
                coarse["MISC"] += count
    return coarse


def make_bar_chart(coarse_counts, title="Entity counts (coarse)"):
    """Return a matplotlib figure for the coarse entity counts
(PER/LOC/ORG/MISC)."""
    ents = ["PER", "LOC", "ORG", "MISC"]
    counts = [coarse_counts.get(e, 0) for e in ents]
    fig, ax = plt.subplots(figsize=(6, 4))
    ax.bar(ents, counts)
    ax.set_xlabel("Entity type")
    ax.set_ylabel("Token count")
    ax.set_title(title)
    for i, v in enumerate(counts):
        ax.text(i, v + max(counts) * 0.01, str(v), ha="center", va="bottom",
fontsize=9)
    plt.tight_layout()
    return fig


def main():
```

```python
    # 0) Setup W&B login (if API key present in env)
    wandb_login()

    # 1) Load official conll2003 dataset
    print("Loading dataset 'conll2003' from Hugging Face...")
    dataset = load_dataset("eriktks/conll2003", revision="convert/parquet")
    print("Loaded splits:", list(dataset.keys()))

    # 2) Compute per-split stats
    split_stats = {}
    overall_entity_by_tag = Counter()
    total_samples = 0
    total_tokens = 0

    for split_name, split_data in dataset.items():
        print(f"Computing stats for split: {split_name} ...")
        stats = compute_entity_counts(split_data)
        split_stats[split_name] = stats
        overall_entity_by_tag.update(stats["entity_counts_by_tag"])
        total_samples += stats["samples"]
        total_tokens += stats["tokens"]

    # 3) Aggregate coarse counts (overall) and per-split coarse counts
    coarse_overall = aggregate_coarse(overall_entity_by_tag)
    coarse_per_split = {
        s: aggregate_coarse(split_stats[s]["entity_counts_by_tag"]) for s in
split_stats
    }

    # 4) Prepare numeric summary dict
    summary = {
        "dataset": "conll2003",
        "total_samples": total_samples,
        "total_tokens": total_tokens,
    }
    for split in ["train", "validation", "test"]:
        summary[f"{split}_samples"] = split_stats.get(split,
{}).get("samples", 0)
        summary[f"{split}_tokens"] = split_stats.get(split, {}).get("tokens",
0)

    # add coarse counts and fractions overall
    for ent in ["PER", "LOC", "ORG", "MISC", "O"]:
        c = int(coarse_overall.get(ent, 0))
        summary[f"entity_count_{ent}"] = c
        summary[f"entity_frac_{ent}"] = c / total_tokens if total_tokens > 0
else 0.0

    # add per-split coarse counts into summary
    for split, coarse in coarse_per_split.items():
        for ent in ["PER", "LOC", "ORG", "MISC", "O"]:
            summary[f"{split}_entity_count_{ent}"] = int(coarse.get(ent, 0))
```

```
    # 5) Initialize W&B run (use finish_previous=True to avoid deprecation
warning)
    run = wandb.init(
        project="Q1-weak-supervision-ner",
        job_type="dataset-stats",
        reinit=True,
        config={"dataset_name": "conll2003"},
    )

    # 6) Update run summary with the numeric summary
    wandb.run.summary.update(summary)

    # 7) Log entity distribution dict (coarse overall) and per-split
distributions
    wandb.log({"entity_distribution_overall": dict(coarse_overall)})
    for split, coarse in coarse_per_split.items():
        wandb.log({f"entity_distribution_{split}": dict(coarse)})

    # 8) Create and log bar chart (matplotlib figure) for overall coarse
counts
    fig = make_bar_chart(coarse_overall, title="Coarse entity counts
(overall)")
    wandb.log({"entity_counts_bar": wandb.Image(fig)})
    plt.close(fig)

    # 9) Save summary JSON and log as artifact
    with tempfile.TemporaryDirectory() as td:
        summary_path = Path(td) / "conll2003_summary.json"
        with open(summary_path, "w", encoding="utf-8") as f:
            json.dump(summary, f, indent=2)
        artifact = wandb.Artifact("conll2003-dataset-stats", type="dataset")
        artifact.add_file(str(summary_path), name="conll2003_summary.json")
        run.log_artifact(artifact)
        print("Saved and logged artifact:", artifact.name)

    # 10) print summary to console and finish run
    print("\nSummary (sample):")
    keys_to_show = [
        "total_samples",
        "total_tokens",
        "entity_count_PER",
        "entity_count_LOC",
        "entity_count_ORG",
        "entity_count_MISC",
        "entity_count_O",
    ]
    for k in keys_to_show:
        print(f"  {k}: {summary.get(k)}")

    run.finish()
    print("\nW&B run finished. View the run at:", wandb.run.get_url() if
wandb.run else "(no run url)")
```

```python
if __name__ == "__main__":
    main()
```

```python
"""
snorkel_lfs_eval_q2.py

- Loads HuggingFace conll2003
- Implements two Snorkel labeling functions:
   a) lf_years: detects years 1900-2099 -> returns MISC
   b) lf_org_suffix: pattern matches organization suffixes -> returns ORG
- Computes coverage and accuracy per-LF and logs them to W&B via
wandb.log().
"""

import os
import re
from collections import Counter
from typing import List

from datasets import load_dataset
import wandb

# Optional: use snorkel decorator if present
try:
    from snorkel.labeling import labeling_function, ABSTAIN
    SNORKEL_AVAILABLE = True
except Exception:
    # If snorkel is not installed, just use a fallback ABSTAIN constant and
callables
    SNORKEL_AVAILABLE = False
    ABSTAIN = -1

# Label mapping (integers for snorkel-style labels)
LABEL_O = 0
LABEL_PER = 1
LABEL_LOC = 2
LABEL_ORG = 3
LABEL_MISC = 4
LABEL_NAMES = {0: "O", 1: "PER", 2: "LOC", 3: "ORG", 4: "MISC"}


# 🔐 Insert your Weights & Biases API Key here:
WANDB_API_KEY = ""
def wandb_login():
    """Login to W&B using provided API key."""
    if WANDB_API_KEY == "your_key_here":
        raise ValueError("Please replace WANDB_API_KEY with your actual W&B
key.")
    os.environ["WANDB_API_KEY"] = WANDB_API_KEY
    wandb.login(key=WANDB_API_KEY)
```

```python
def coarse_label_from_conll_tag(tag: str) -> int:
    """
    Map a tag like 'B-PER', 'I-ORG', 'O' -> coarse label integer.
    """
    if tag == "O":
        return LABEL_O
    parts = tag.split("-", 1)
    if len(parts) == 2:
        ent = parts[1]
    else:
        ent = tag
    if ent == "PER":
        return LABEL_PER
    if ent == "LOC":
        return LABEL_LOC
    if ent == "ORG":
        return LABEL_ORG
    # anything else -> MISC
    return LABEL_MISC


def build_token_level_dataset(max_samples=None):
    """
    Flatten sentence examples into token-level records.
    Returns list of dicts: {"token": str, "gold": int}
    max_samples: optional limit on number of sentences to process (for
speed).
    """
    ds = load_dataset("eriktks/conll2003", revision="convert/parquet")
    records = []
    # use train+validation+test or just train? we'll use all splits for
evaluation
    for split_name in ds.keys():
        for i, ex in enumerate(ds[split_name]):
            if max_samples is not None and i >= max_samples:
                break
            tokens = ex["tokens"]
            tags = ex["ner_tags"]  # integers -> label names from dataset
features
            label_names = ds[split_name].features["ner_tags"].feature.names
            # convert tag ids to label names then to coarse
            for tag_id, token in zip(tags, tokens):
                tag_name = label_names[tag_id]
                coarse = coarse_label_from_conll_tag(tag_name)
                records.append({"token": token, "gold": coarse, "split":
split_name})
    return records


# ------------------------
# Labeling functions
```

```python
# -------------------------

YEAR_REGEX = re.compile(r"^(19\d{2}|20\d{2})$")  # matches 1900-2099 strictly
# To avoid matching things like '2020,' or '(1999)' we strip punctuation boundaries when checking.

ORG_SUFFIXES = [r"\bInc\.?$", r"\bCorp\.?$", r"\bLtd\.?$", r"\bLLC\.?$", r"\bPLC\.?$"]
ORG_SUFFIX_REGEX = re.compile(r"(?i)(" + r"|".join(s[:-1] if s.endswith("$") else s for s in ORG_SUFFIXES) + r")\.?$")

# Snorkel decorator (if available) expects a function taking a single example arg.
if SNORKEL_AVAILABLE:
    @labeling_function()
    def lf_years(x):
        token = x["token"]
        t = token.strip().strip(".,;:()[]\"'")  # basic punctuation strip
        if YEAR_REGEX.match(t):
            return LABEL_MISC  # DATE/MISC as requested
        return ABSTAIN

    @labeling_function()
    def lf_org_suffix(x):
        token = x["token"]
        # check token endswith org suffix (case-insensitive)
        t = token.strip()
        # match whole token suffixes like 'Inc.' or 'Corp' etc.
        if re.search(r"(Inc\.?|Corp\.?|Ltd\.?|LLC\.?|PLC\.?)$", t, flags=re.IGNORECASE):
            return LABEL_ORG
        return ABSTAIN
else:
    # Fallback plain functions returning ints or ABSTAIN
    def lf_years(x):
        token = x["token"]
        t = token.strip().strip(".,;:()[]\"'")
        if YEAR_REGEX.match(t):
            return LABEL_MISC
        return ABSTAIN

    def lf_org_suffix(x):
        token = x["token"]
        t = token.strip()
        if re.search(r"(Inc\.?|Corp\.?|Ltd\.?|LLC\.?|PLC\.?)$", t, flags=re.IGNORECASE):
            return LABEL_ORG
        return ABSTAIN


# -------------------------
# Evaluation helpers
# -------------------------
```

```python
def evaluate_labeling_function(records: List[dict], lf_callable, lf_name:
str):
    """
    Apply lf_callable to every token record.
    Compute coverage = fraction of tokens where LF != ABSTAIN.
    Compute accuracy = fraction of labeled tokens where LF_label == gold
label.
    Returns dict of metrics.
    """
    n = len(records)
    labeled = 0
    correct = 0
    for rec in records:
        lab = lf_callable(rec)
        if lab != ABSTAIN:
            labeled += 1
            if lab == rec["gold"]:
                correct += 1
    coverage = labeled / n if n > 0 else 0.0
    accuracy = (correct / labeled) if labeled > 0 else 0.0
    return {
        "lf_name": lf_name,
        "n_tokens": n,
        "n_labeled": int(labeled),
        "coverage": float(coverage),
        "accuracy": float(accuracy),
    }


def main():
    wandb_login()
    # Build token-level dataset (use all sentences; if large, you can pass
max_samples to limit)
    print("Building token-level dataset (this may take a moment)...")
    records = build_token_level_dataset(max_samples=None)  # None -> use all
    print(f"Total tokens: {len(records)}")

    # Evaluate both LFs
    metrics_years = evaluate_labeling_function(records, lf_years, "lf_years")
    metrics_org = evaluate_labeling_function(records, lf_org_suffix,
"lf_org_suffix")

    # Print results locally
    print("\nLabeling function metrics:")
    for m in (metrics_years, metrics_org):
        print(f"- {m['lf_name']}: labeled {m['n_labeled']}/{m['n_tokens']} "
              f"({m['coverage']:.4f}), accuracy={m['accuracy']:.4f}")

    # Log to W&B
    run = wandb.init(
        project="Q1-weak-supervision-ner",
        job_type="lf-eval",
```

```python
        reinit=True,
        config={"note": "LF coverage and accuracy for Q2"}
    )
    # Log each LF metrics
    wandb.log({
        "lf_years/coverage": metrics_years["coverage"],
        "lf_years/accuracy": metrics_years["accuracy"],
        "lf_years/n_labeled": metrics_years["n_labeled"],
        "lf_years/n_tokens": metrics_years["n_tokens"],

        "lf_org_suffix/coverage": metrics_org["coverage"],
        "lf_org_suffix/accuracy": metrics_org["accuracy"],
        "lf_org_suffix/n_labeled": metrics_org["n_labeled"],
        "lf_org_suffix/n_tokens": metrics_org["n_tokens"],
    })

    # Also store the raw metrics in run.summary for quick top-level view
    wandb.run.summary.update({
        "lf_years/coverage": metrics_years["coverage"],
        "lf_years/accuracy": metrics_years["accuracy"],
        "lf_org_suffix/coverage": metrics_org["coverage"],
        "lf_org_suffix/accuracy": metrics_org["accuracy"],
    })

    print("\nLogged LF metrics to W&B. Run URL:", wandb.run.get_url() if
wandb.run else "(no run url)")
    run.finish()


if __name__ == "__main__":
    main()
```

```python
"""
majority_label_voter_q3.py

Implements Snorkel-style majority label aggregation (MajorityLabelVoter).
Uses two LFs:
- lf_years -> LABEL_MISC for 1900-2099 tokens
- lf_org_suffix -> LABEL_ORG for tokens ending with org suffixes

If snorkel is installed, uses
snorkel.labeling.majority_label_model.MajorityLabelVoter (or the
LabelModel alternative). Otherwise uses local majority-vote implementation.

Logs coverage/accuracy to W&B using wandb.log().
"""

import os
import re
import numpy as np
from collections import Counter, defaultdict
```

```python
from datasets import load_dataset
import wandb

# Try to import snorkel's MajorityLabelVoter if available
try:
    from snorkel.labeling import MajorityLabelVoter, ABSTAIN
    SNORKEL_AVAILABLE = True
except Exception:
    SNORKEL_AVAILABLE = False
    ABSTAIN = -1  # we'll use -1 for abstain

# Label mapping (same as Q2)
LABEL_O = 0
LABEL_PER = 1
LABEL_LOC = 2
LABEL_ORG = 3
LABEL_MISC = 4
LABEL_NAMES = {0: "O", 1: "PER", 2: "LOC", 3: "ORG", 4: "MISC"}


# 🔐 Insert your Weights & Biases API Key here:
WANDB_API_KEY = ""
def wandb_login():
    """Login to W&B using provided API key."""
    if WANDB_API_KEY == "your_key_here":
        raise ValueError("Please replace WANDB_API_KEY with your actual W&B
key.")
    os.environ["WANDB_API_KEY"] = WANDB_API_KEY
    wandb.login(key=WANDB_API_KEY)

# ----- Build token-level records (flatten dataset) -----
def coarse_label_from_conll_tag(tag: str) -> int:
    if tag == "O":
        return LABEL_O
    parts = tag.split("-", 1)
    ent = parts[1] if len(parts) == 2 else tag
    if ent == "PER":
        return LABEL_PER
    if ent == "LOC":
        return LABEL_LOC
    if ent == "ORG":
        return LABEL_ORG
    return LABEL_MISC


def build_token_records(max_sentences_per_split=None):
    ds = load_dataset("eriktks/conll2003", revision="convert/parquet")
    records = []
    for split in ds.keys():
        for i, ex in enumerate(ds[split]):
            if max_sentences_per_split is not None and i >=
max_sentences_per_split:
                break
```

```
            tokens = ex["tokens"]
            tags = ex["ner_tags"]
            label_names = ds[split].features["ner_tags"].feature.names
            for token, tag_id in zip(tokens, tags):
                tag_name = label_names[tag_id]
                gold = coarse_label_from_conll_tag(tag_name)
                records.append({"token": token, "gold": gold, "split": split})
    return records


# ----- Labeling functions (same as Q2) -----
YEAR_REGEX = re.compile(r"^(19\d{2}|20\d{2})$")
def lf_years(rec):
    t = rec["token"].strip().strip(".,;:()[]\"'")
    if YEAR_REGEX.match(t):
        return LABEL_MISC
    return ABSTAIN

def lf_org_suffix(rec):
    t = rec["token"].strip()
    if re.search(r"(Inc\.?|Corp\.?|Ltd\.?|LLC\.?|PLC\.?)$", t,
flags=re.IGNORECASE):
        return LABEL_ORG
    return ABSTAIN

LABELING_FUNCTIONS = [lf_years, lf_org_suffix]


# ----- Build label matrix L (n_tokens x n_lfs) -----
def build_label_matrix(records, lfs):
    n = len(records)
    m = len(lfs)
    L = np.full((n, m), ABSTAIN, dtype=int)
    for i, rec in enumerate(records):
        for j, lf in enumerate(lfs):
            try:
                L[i, j] = lf(rec)
            except Exception:
                L[i, j] = ABSTAIN
    return L


# ----- Majority voting aggregator (fallback) -----
def majority_vote_row(row):
    """
    row: 1D array of LF outputs for a single token (values are label ints or
ABSTAIN)
    Return: aggregated label int or ABSTAIN
    Tie-breaking strategy: if tie between two or more labels for max count,
return ABSTAIN.
    """
    counts = Counter([int(x) for x in row if int(x) != ABSTAIN])
    if not counts:
```

```
        return ABSTAIN
    most_common = counts.most_common()
    if len(most_common) == 0:
        return ABSTAIN
    # Check tie: compare top two counts
    top_label, top_count = most_common[0]
    if len(most_common) > 1 and most_common[1][1] == top_count:
        # tie -> abstain
        return ABSTAIN
    return top_label


def aggregate_majority_local(L):
    # Apply majority_vote_row for each row
    n = L.shape[0]
    aggregated = np.full(n, ABSTAIN, dtype=int)
    for i in range(n):
        aggregated[i] = majority_vote_row(L[i, :])
    return aggregated


# ----- Evaluation -----
def evaluate_aggregated_labels(aggr_labels, gold_labels):
    n = len(gold_labels)
    assert len(aggr_labels) == n
    labeled_mask = (aggr_labels != ABSTAIN)
    n_labeled = int(np.sum(labeled_mask))
    coverage = n_labeled / n if n > 0 else 0.0
    correct = int(np.sum((aggr_labels == gold_labels) & labeled_mask))
    accuracy = (correct / n_labeled) if n_labeled > 0 else 0.0
    return {"n_tokens": n, "n_labeled": n_labeled, "coverage": coverage,
"accuracy": accuracy}


# ----- Main flow -----
def main():
    wandb_login()
    print("Building token-level records (this may take a moment)...")
    records = build_token_records(max_sentences_per_split=None)  # use all
    n_tokens = len(records)
    print(f"Total tokens: {n_tokens}")

    # Build label matrix L
    L = build_label_matrix(records, LABELING_FUNCTIONS)
    print("Label matrix shape:", L.shape)

    # If snorkel is available, use its MajorityLabelVoter
    if SNORKEL_AVAILABLE:
        print("Using Snorkel's MajorityLabelVoter for aggregation...")
        mv = MajorityLabelVoter()
        # Snorkel expects L as numpy array with shape (n, m)
        aggregated = mv.predict(L)  # returns a 1D array of aggregated labels
(or ABSTAIN)
```

```python
    else:
        print("Snorkel not available — using local majority-vote aggregator.")
        aggregated = aggregate_majority_local(L)

    # Evaluate aggregated labels vs gold
    gold = np.array([rec["gold"] for rec in records], dtype=int)
    metrics = evaluate_aggregated_labels(aggregated, gold)

    print("\nMajority-voter metrics:")
    print(f"  tokens labeled: {metrics['n_labeled']}/{metrics['n_tokens']}")
    print(f"  coverage: {metrics['coverage']:.6f}")
    print(f"  accuracy (on labeled tokens): {metrics['accuracy']:.6f}")

    # Log to W&B
    run = wandb.init(
        project="Q1-weak-supervision-ner",
        job_type="majority-aggregation",
        reinit=True,
        config={"n_lfs": len(LABELING_FUNCTIONS)}
    )

    wandb.log({
        "majority/n_tokens": metrics["n_tokens"],
        "majority/n_labeled": metrics["n_labeled"],
        "majority/coverage": metrics["coverage"],
        "majority/accuracy": metrics["accuracy"],
    })
    # Also store a small confusion-like breakdown: counts for aggregated
labels vs gold (only labeled tokens)
    aggr_counts = Counter(int(x) for x in aggregated if int(x) != ABSTAIN)
    wandb.log({"majority/aggregated_label_counts": dict(aggr_counts)})

    # Optionally: log a tiny table of examples where aggregator labeled
(token, aggr, gold)
    # We'll log up to 100 examples to avoid huge uploads
    table_rows = []
    max_examples = 100
    added = 0
    for i, rec in enumerate(records):
        if added >= max_examples:
            break
        if aggregated[i] != ABSTAIN:
            table_rows.append([rec["token"], int(aggregated[i]),
LABEL_NAMES.get(int(aggregated[i]), str(aggregated[i])), rec["gold"],
LABEL_NAMES.get(rec["gold"], str(rec["gold"])), rec["split"]])
            added += 1
    # create a W&B table if there are rows
    if table_rows:
        tb = wandb.Table(columns=["token", "aggr_label_id", "aggr_label_name",
"gold_id", "gold_name", "split"], data=table_rows)
        wandb.log({"majority/labeled_examples_table": tb})

    wandb.run.summary.update({
```

```python
        "majority/coverage": metrics["coverage"],
        "majority/accuracy": metrics["accuracy"],
    })

    run.finish()
    print("\nLogged majority aggregation metrics to W&B:", wandb.run.get_url()
if wandb.run else "(no run url)")


if __name__ == "__main__":
    main()
```

```python
#%%
# Cell 1 — Install dependencies
!pip install torchvision torchaudio wandb
import torchvision
import torchaudio
#%%
import wandb
wandb.login()
#%%
import os
import random
import time
from pathlib import Path
from typing import Tuple, Dict

import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
from torch.utils.data import DataLoader, Subset
from torchvision.models import resnet18

import wandb

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print("Using device:", device)

def set_seed(seed=42):
    random.seed(seed)
    torch.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)
set_seed(42)

#%%
# Cell 4 — Dataloaders factory
def get_cifar_loaders(name: str, batch_size=256, num_workers=2,
augment=True):
```

```python
    assert name in ("CIFAR10", "CIFAR100")
    if name == "CIFAR10":
        dataset_class = torchvision.datasets.CIFAR10
    else:
        dataset_class = torchvision.datasets.CIFAR100

    mean = (0.4914, 0.4822, 0.4465)
    std  = (0.2470, 0.2435, 0.2616)

    train_transforms = [
        transforms.RandomCrop(32, padding=4),
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(),
        transforms.Normalize(mean, std)
    ] if augment else [transforms.ToTensor(), transforms.Normalize(mean,std)]

    test_transforms = [transforms.ToTensor(), transforms.Normalize(mean,std)]

    trainset = dataset_class(root='./data', train=True, download=True,
transform=transforms.Compose(train_transforms))
    testset  = dataset_class(root='./data', train=False, download=True,
transform=transforms.Compose(test_transforms))

    train_loader = DataLoader(trainset, batch_size=batch_size, shuffle=True,
num_workers=num_workers, pin_memory=True)
    test_loader  = DataLoader(testset, batch_size=batch_size, shuffle=False,
num_workers=num_workers, pin_memory=True)
    return train_loader, test_loader, trainset, testset

#%%
# Cell 5 — Model builder (ResNet18 adapted to num_classes)
def build_model(num_classes: int, pretrained=False):
    model = resnet18(pretrained=pretrained)
    # adapt first conv for CIFAR (32x32): change kernel_size=3, stride=1,
padding=1
    model.conv1 = torch.nn.Conv2d(3, 64, kernel_size=3, stride=1, padding=1,
bias=False)
    model.maxpool = torch.nn.Identity()
    model.fc = torch.nn.Linear(model.fc.in_features, num_classes)
    return model

#%%
# Cell 6 — Training/validation utilities + logging helpers
from sklearn.metrics import confusion_matrix

def evaluate(model, dataloader, device):
    model.eval()
    correct = 0
    total = 0
    losses = []
    criterion = nn.CrossEntropyLoss()
    preds = []
    targets = []
```

```
    with torch.no_grad():
        for x, y in dataloader:
            x = x.to(device)
            y = y.to(device)
            out = model(x)
            loss = criterion(out, y)
            losses.append(loss.item())
            _, p = out.max(1)
            correct += (p == y).sum().item()
            total += y.size(0)
            preds.append(p.cpu().numpy())
            targets.append(y.cpu().numpy())
    avg_loss = float(np.mean(losses))
    acc = correct / total
    preds = np.concatenate(preds)
    targets = np.concatenate(targets)
    return avg_loss, acc, preds, targets

def log_confusion_matrix(targets, preds, class_labels, step=None, prefix=""):
    # simple confusion matrix logging to W&B (as a table)
    cm = confusion_matrix(targets, preds)
    # Normalize for readability
    cm_norm = cm.astype(float)
    with np.errstate(divide='ignore', invalid='ignore'):
        row_sums = cm_norm.sum(axis=1, keepdims=True)
        cm_norm = np.divide(cm_norm, row_sums, where=row_sums!=0)
    # Log as an image via wandb.plot.confusion_matrix if available
    try:
        wandb.log({f"{prefix}confusion_matrix":
wandb.plot.confusion_matrix(probs=None,

y_true=targets.tolist(),

preds=preds.tolist(),

class_names=class_labels)},
                 step=step)
    except Exception:
        # fallback to logging the raw matrix as artifact/table
        wandb.log({f"{prefix}confusion_matrix_array": cm.tolist()}, step=step)

#%%
# Cell 7 — Training loop that supports sequential datasets, and forgetting
measurement
def run_sequential_experiment(seq, base_seed=42, epochs_per_task=100,
batch_size=256, lr=0.01, weight_decay=5e-4):
    """
    seq: list like ["CIFAR100", "CIFAR10"] in the order to train
    """
    set_seed(base_seed)
    # Common initialization: same seed -> same initial weights for both
experiments
```

```
    initial_model = build_model(num_classes=100)  # create model with largest
class count
    init_state = initial_model.state_dict()

    # Initialize a W&B run for this experiment
    run_name = "_then_".join(seq)
    wandb.init(project="cifar-sequential-wandb", name=run_name, config={
        "sequence": seq,
        "epochs_per_task": epochs_per_task,
        "batch_size": batch_size,
        "lr": lr,
        "weight_decay": weight_decay,
        "seed": base_seed,
        "model": "resnet18_cifar"
    })

    model = build_model(num_classes=100).to(device)
    model.load_state_dict(init_state)  # ensure same start across experiments

    optimizer = optim.SGD(model.parameters(), lr=lr, momentum=0.9,
weight_decay=weight_decay)
    scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=50, gamma=0.1)
    criterion = nn.CrossEntropyLoss()

    # Keep holdouts for both datasets so we can measure forgetting
    holdouts = {}  # name -> (dataloader, num_classes, class_labels)
    dataloaders = {}  # current dataloaders
    datasets_full = {}

    for name in ("CIFAR10","CIFAR100"):
        train_loader, test_loader, trainset, testset = get_cifar_loaders(name,
batch_size=batch_size, augment=True)
        dataloaders[name] = (train_loader, test_loader)
        datasets_full[name] = (trainset, testset)
        # class labels
        if name=="CIFAR10":
            labels = [str(i) for i in range(10)]
        else:
            labels = [str(i) for i in range(100)]
        holdouts[name] = (test_loader, len(labels), labels)

    # Track checkpoint before any training (to measure initial performance)
    performance = {}

    # Evaluate initial model on both testsets (after adapting final layer to
dataset num_classes)
    def eval_on_dataset(model, name):
        # Create copy of model and adapt final layer (since we used
num_classes=100)
        n_classes = 10 if name=="CIFAR10" else 100
        m = build_model(num_classes=n_classes).to(device)
        # copy all weights except final fc (if shapes differ)
        sd = model.state_dict()
```

```
        m_sd = m.state_dict()
        # copy compatible keys
        for k in m_sd:
            if k in sd and sd[k].shape == m_sd[k].shape:
                m_sd[k] = sd[k]
        m.load_state_dict(m_sd)
        return evaluate(m, holdouts[name][0], device)

    # Initial eval
    for name in seq:
        loss, acc, _, _ = eval_on_dataset(model, name)
        wandb.log({f"initial/{name}_loss": loss, f"initial/{name}_acc": acc},
step=0)
        performance[f"initial_{name}"] = (loss, acc)

    global_step = 0
    # Now train sequentially
    for task_idx, task in enumerate(seq):
        train_loader, test_loader = dataloaders[task]
        n_classes = 10 if task=="CIFAR10" else 100

        # If model.fc size != n_classes, replace final layer (fine-tune last
layer)
        if model.fc.out_features != n_classes:
            in_features = model.fc.in_features
            model.fc = nn.Linear(in_features, n_classes).to(device)
            # note: reinit new layer's params (keeps other weights)
        wandb.log({f"task_started": task, "task_index": task_idx},
step=global_step)
        for epoch in range(1, epochs_per_task+1):
            model.train()
            epoch_losses = []
            correct = 0
            total = 0
            for xb, yb in train_loader:
                xb = xb.to(device)
                yb = yb.to(device)
                optimizer.zero_grad()
                out = model(xb)
                loss = criterion(out, yb)
                loss.backward()
                optimizer.step()
                epoch_losses.append(loss.item())
                _, p = out.max(1)
                correct += (p==yb).sum().item()
                total += yb.size(0)
                global_step += 1
            scheduler.step()

            train_loss = float(np.mean(epoch_losses))
            train_acc = correct/total
            val_loss, val_acc, val_preds, val_targets = evaluate(model,
test_loader, device)
```

```
            # Log to W&B
            wandb.log({
                f"{task}/train_loss": train_loss,
                f"{task}/train_acc": train_acc,
                f"{task}/val_loss": val_loss,
                f"{task}/val_acc": val_acc,
                "epoch": epoch,
            }, step=global_step)

            # Periodic confusion matrix (less frequent to save bandwidth)
            if epoch % 25 == 0 or epoch == epochs_per_task:
                class_labels = [str(i) for i in range(n_classes)]
                log_confusion_matrix(val_targets, val_preds, class_labels,
step=global_step, prefix=f"{task}/")

        # After finishing this task, evaluate performance on all tasks
(measure forgetting)
        for other in seq:
            loss_o, acc_o, _, _ = eval_on_dataset(model, other)
            wandb.log({f"after_{task}/{other}_loss": loss_o,
f"after_{task}/{other}_acc": acc_o}, step=global_step)
            performance[f"after_{task}_{other}"] = (loss_o, acc_o)

        # Save model artifact snapshot
        artifact = wandb.Artifact(f"{run_name}_after_{task}", type="model")
        model_file = f"model_{run_name}_after_{task}.pth"
        torch.save(model.state_dict(), model_file)
        artifact.add_file(model_file)
        wandb.log_artifact(artifact)

    # Final evaluations and summary
    for name in seq:
        loss, acc, _, _ = eval_on_dataset(model, name)
        wandb.log({f"final/{name}_loss": loss, f"final/{name}_acc": acc},
step=global_step)
        performance[f"final_{name}"] = (loss, acc)

    wandb.finish()
    return performance

#%%
# Cell 8 — Run both experiments sequentially in the notebook (this will take
time — 200 epochs per experiment total)
# If you want to run them in separate Colab sessions / sequentially,
comment/uncomment as needed.

# EXPERIMENT A: CIFAR100 -> CIFAR10
perf_A = run_sequential_experiment(["CIFAR100", "CIFAR10"], base_seed=42,
epochs_per_task=100, batch_size=256, lr=0.05)

# EXPERIMENT B: CIFAR10 -> CIFAR100
```

```python
perf_B = run_sequential_experiment(["CIFAR10", "CIFAR100"], base_seed=42,
epochs_per_task=100, batch_size=256, lr=0.05)

# Save performance summaries to disk for quick inspection
import json
with open("perf_A.json","w") as f:
   json.dump(perf_A, f)
with open("perf_B.json","w") as f:
   json.dump(perf_B, f)

print("Experiments finished. Check W&B project: cifar-sequential-wandb")

#%%
```

wandb.ai/142502015-indian-institute-of-technology/Q1-weak-supervision-ner/runs/4szjqaqc/...

For quick access, place your bookmarks here on the bookmarks bar. Import bookmarks now...

Summary


```
{
  "_runtime": 0,
  "_step": 0,
  "_timestamp": 1760277578.8445635,
  "_wandb.runtime": 0,
  "dataset": "eriktks/conll2003",
  "entity_count_LOC": 12316,
  "entity_count_MISC": 6779,
  "entity_count_O": 250660,
  "entity_count_ORG": 14613,
  "entity_count_PER": 17050,
  "entity_distribution.LOC": 12316,
  "entity_distribution.MISC": 6779,
  "entity_distribution.O": 250660,
  "entity_distribution.ORG": 14613,
  "entity_distribution.PER": 17050,
  "entity_frac_LOC": 0.04086020078429291,
  "entity_frac_MISC": 0.022490362221234295,
  "entity_frac_O": 0.8316026249261822,
  "entity_frac_ORG": 0.04848084719558885,
```


Close Copy

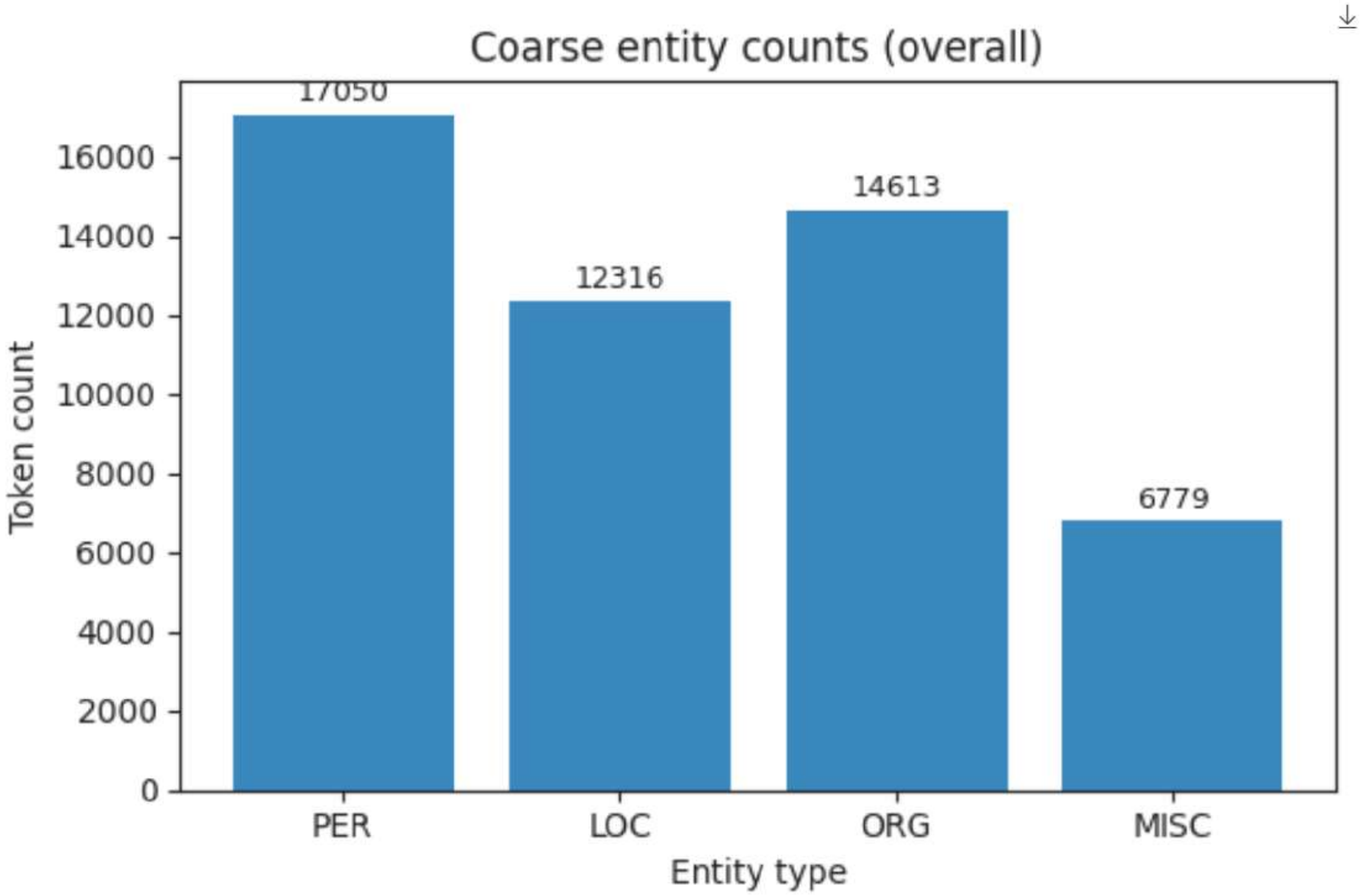
wandb.ai/142502015-indian-institute-of-technology/Q1-weak-supervision-ner/runs/k7973hji/...

For quick access, place your bookmarks here on the bookmarks bar. Import bookmarks now...

Summary


```
{
  "_runtime": 0,
  "_step": 0,
  "_timestamp": 1760278615.8519857,
  "_wandb.runtime": 0,
  "lf_org_suffix/accuracy": 0.9465020576131687,
  "lf_org_suffix/coverage": 0.0008061894113822001,
  "lf_org_suffix/n_labeled": 243,
  "lf_org_suffix/n_tokens": 301418,
  "lf_years/accuracy": 0.0149812734082397,
  "lf_years/coverage": 0.0026574391708524374,
  "lf_years/n_labeled": 801,
  "lf_years/n_tokens": 301418
}
```


Close Copy


Coarse entity counts (overall)
17050
14613
12316
6779
0
2000
4000
6000
8000
10000
12000
14000
16000
Token count
PER
LOC
ORG
MISC
Entity type

Summary metrics: {} 8 keys

lf_org_suffix/accuracy: 0.9465020576131687

lf_org_suffix/coverage: 0.0008061894113822001

lf_org_suffix/n_labeled: 243

lf_org_suffix/n_tokens: 301,418

lf_years/accuracy: 0.0149812734082397

lf_years/coverage: 0.0026574391708524374

lf_years/n_labeled: 801

lf_years/n_tokens: 301,418

Config parameters: {} 1 key

note: "LF coverage and accuracy for Q2"

Summary metrics: {} 8 keys

lf_org_suffix/accuracy: 0.9465020576131687

lf_org_suffix/coverage: 0.0008061894113822001

lf_org_suffix/n_labeled: 243

lf_org_suffix/n_tokens: 301,418

lf_years/accuracy: 0.0149812734082397

lf_years/coverage: 0.0026574391708524374

lf_years/n_labeled: 801

lf_years/n_tokens: 301,418

Summary metrics: {} 7 keys

majority/accuracy: 0.23180076628352492

majority/aggregated_label_counts.3: 243

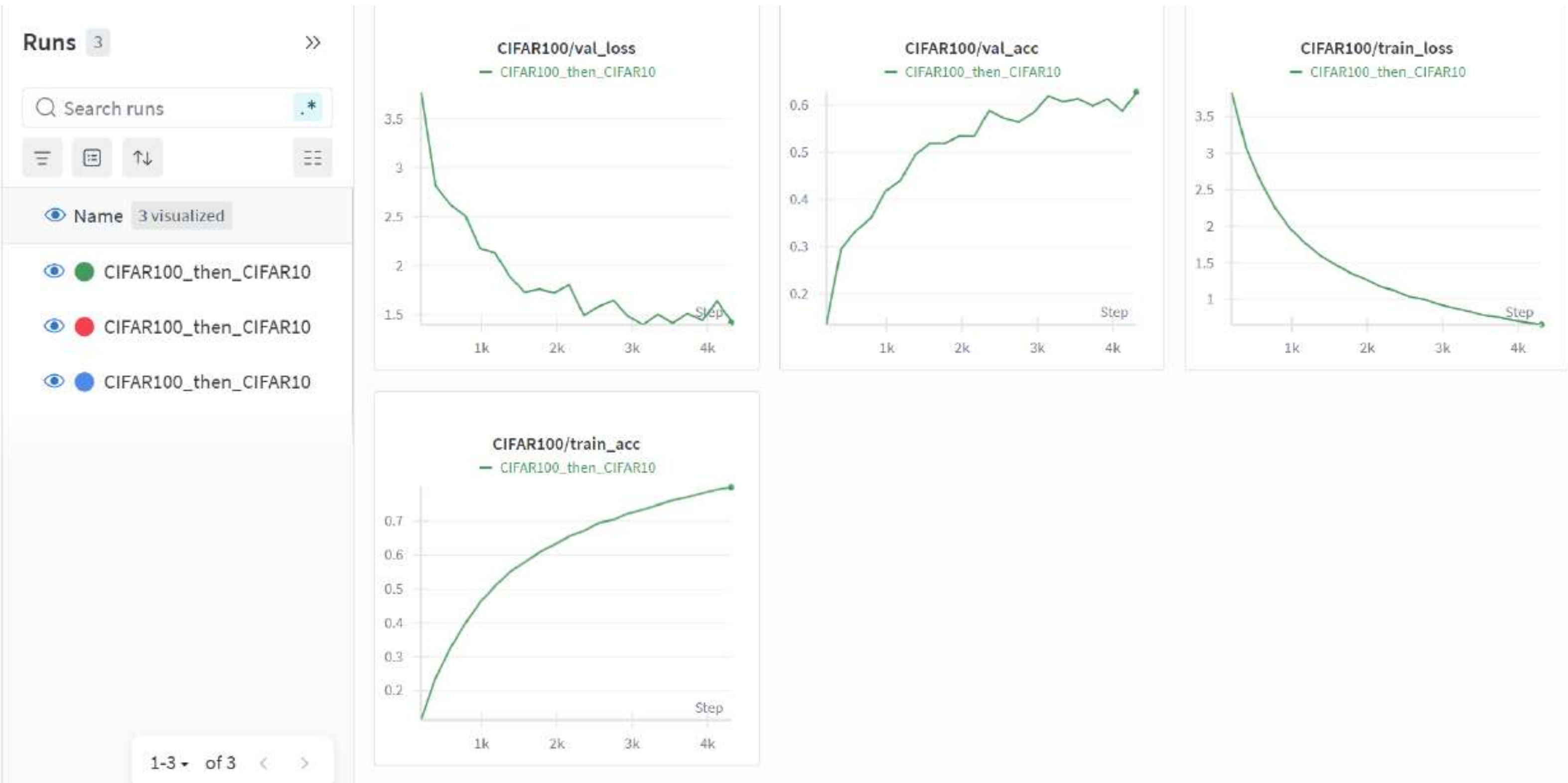
majority/aggregated_label_counts.4: 801

majority/coverage: 0.0034636285822346375

majority/labeled_examples_table: "table-file"

majority/n_labeled: 1,044

majority/n_tokens: 301,418


Runs 3
Search runs
Name 3 visualized
CIFAR100_then_CIFAR10
CIFAR100_then_CIFAR10
CIFAR100_then_CIFAR10
1-3 of 3
CIFAR100/val_loss
CIFAR100_then_CIFAR10
Step
CIFAR100/val_acc
CIFAR100_then_CIFAR10
Step
CIFAR100/train_loss
CIFAR100_then_CIFAR10
Step
CIFAR100/train_acc
CIFAR100_then_CIFAR10
Step