```
!pip -q install "metaflow>=2.11.10" "transformers>=4.43.3" "datasets>=2.20.0" \
"accelerate>=0.33.0" "bitsandbytes>=0.43.1" "peft>=0.11.1" "evaluate>=0.4.2" \
"scikit-learn>=1.5.0" "pandas>=2.2.2" "tqdm>=4.66.4" spacy
!python -m spacy download en_core_web_sm
```

```
Collecting en-core-web-sm==3.8.0
  Downloading https://github.com/explosion/spacy-models/releases/download/en_core_web_sm-3.8.0/en_core_web_sm-3.8.0-py3-none
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 12.8/12.8 MB 81.1 MB/s eta 0:00:00
✓ Download and installation successful
You can now load the package via spacy.load('en_core_web_sm')
⚠ Restart to reload dependencies
If you are in a Jupyter or Colab notebook, you may need to restart Python in
order to load all the package's dependencies. You can do this by selecting the
'Restart kernel' or 'Restart runtime' option.
```

```
import os
os.environ["METAFLOW_USER"] = "colab"
os.environ["METAFLOW_DEFAULT_METADATA"] = "local"
os.environ["METAFLOW_DATASTORE_SYSROOT_LOCAL"] = "/content/metaflow_datastore"
os.makedirs(os.environ["METAFLOW_DATASTORE_SYSROOT_LOCAL"], exist_ok=True)

# REQUIRED for Llama 3.2 1B (accept license on HF first)
if "HF_TOKEN" not in os.environ or not os.environ["HF_TOKEN"]:
    from getpass import getpass
    os.environ["HF_TOKEN"] = getpass("Paste your HF token (hidden): ")
```

```
Paste your HF token (hidden): ··········
```

```
%%writefile med_helpers.py
import re, json
import numpy as np
from typing import List, Dict, Tuple

FIELDS = ["Examination","Clinical","Findings","Impression"]

ABBREV_MAP = {
    "wnl": "within normal limits",
    "nl": "normal",
    "no focal opacity": "no focal consolidation",  # simple example
}

def norm_txt(x: str) -> str:
    if not isinstance(x, str):
        return ""
    x = x.strip().lower()
    # spacing & punctuation
    x = re.sub(r"[ \t\r\n]+", " ", x)
    x = re.sub(r"[,:;]+", " ", x)
    x = re.sub(r"\s{2,}", " ", x).strip(" .")
    # expand a few common abbreviations
    for k,v in ABBREV_MAP.items():
        x = re.sub(rf"\b{k}\b", v, x)
    return x

def build_prompt(report_text: str, fields=FIELDS) -> str:
    # ——— Keep your format/content; just fixed f-string backslash issue previously ———
    max_chars = {"Examination":120,"Clinical":160,"Findings":600,"Impression":300}
    quoted_keys = ", ".join([f'"{k}"' for k in fields])

    role = ("You are a precise medical information extraction engine. "
            "Your only job is to extract specific sections from the given radiology report.")
    schema = (
        "Return a single JSON object with exactly these keys and string values:\n"
        f"{quoted_keys}.\n"
        "The JSON must be valid and parseable by a standard JSON parser.\n"
        "No extra keys, no comments, no explanations, no markdown."
    )
    constraints = (
        "Rules:\n"
        "1) Copy or lightly paraphrase from the report; do NOT invent clinical facts.\n"
        "2) If a section is missing or truly unspecified, use an empty string \"\".\n"
        "3) Remove patient identifiers/PHI. Keep clinical terminology intact.\n"
        "4) Keep each field concise and readable. Do not include section headers in values.\n"
        "5) Do not include JSON code fences. Output JSON only.\n"
        "6) Respect length limits (characters, not tokens):\n"
        f"   - Examination: <= {max_chars['Examination']}\n"
        f"   - Clinical:    <= {max_chars['Clinical']}\n"
        f"   - Findings:    <= {max_chars['Findings']}\n"
        f"   - Impression:  <= {max_chars['Impression']}\n"
        "7) If multiple mentions exist for a section, merge succinctly into one value.\n"
```

```
        "8) Normalize whitespace: single spaces, no trailing punctuation unless natural.\n"
        "9) Do not include quotes inside values unless they are part of the clinical text."
    )
    few_shot = (
        "Examples:\n"
        "Example 1 (well-structured input):\n"
        "Input:\n"
        "  Exam: Chest X-ray PA and lateral\n"
        "  Clinical indication: Dyspnea and cough for 3 days.\n"
        "  Findings: Cardiomediastinal silhouette normal. No focal consolidation or effusion.\n"
        "  Impression: No acute cardiopulmonary disease.\n"
        "JSON:\n"
        "{\n"
        "  \"Examination\": \"Chest X-ray PA and lateral\",\n"
        "  \"Clinical\": \"Dyspnea and cough for 3 days\",\n"
        "  \"Findings\": \"Cardiomediastinal silhouette normal. No focal consolidation or effusion\",\n"
        "  \"Impression\": \"No acute cardiopulmonary disease\"\n"
        "}\n"
        "\n"
        "Example 2 (messy input, missing Impression):\n"
        "Input:\n"
        "  ... portable ap chest performed at bedside ...\n"
        "  hx: fever; r/o pneumonia. lungs clear; no focal opacity; heart size wnl.\n"
        "JSON:\n"
        "{\n"
        "  \"Examination\": \"Portable AP chest radiograph\",\n"
        "  \"Clinical\": \"Fever; rule out pneumonia\",\n"
        "  \"Findings\": \"Lungs clear; no focal opacity; cardiac size within normal limits\",\n"
        "  \"Impression\": \"\"\n"
        "}\n"
    )
    task = f"Input report:\n{report_text}\n\nNow output ONLY the JSON object:"
    return "\n".join([role, "", schema, "", constraints, "", few_shot, task])

def safe_json(text: str) -> Dict[str,str]:
    if not isinstance(text, str):
        return {}
    try:
        obj = json.loads(text)
        if isinstance(obj, dict):
            return obj
    except Exception:
        pass
    blocks = re.findall(r"\{[\s\S]*\}", text)
    for cand in reversed(blocks):
        try:
            obj = json.loads(cand)
            if isinstance(obj, dict):
                return obj
        except Exception:
            continue
    # last resort: extract "Key":"value" pairs
    out = {}
    for k in FIELDS:
        mm = re.search(rf'"{re.escape(k)}"\s*:\s*"([^"]*)"', text)
        if mm:
            out[k] = mm.group(1)
    return out

# ---------- Metrics ----------
def token_f1(pred: str, gold: str) -> Tuple[float,float,float]:
    ptoks = [t for t in re.findall(r"\w+", norm_txt(pred))]
    gtoks = [t for t in re.findall(r"\w+", norm_txt(gold))]
    if len(ptoks) == 0 and len(gtoks) == 0:
        return 1.0, 1.0, 1.0
    if len(ptoks) == 0 or len(gtoks) == 0:
        return 0.0, 0.0, 0.0
    pset, gset = set(ptoks), set(gtoks)
    inter = len(pset & gset)
    prec = inter / max(1, len(pset))
    rec  = inter / max(1, len(gset))
    if prec+rec == 0:
        f1 = 0.0
    else:
        f1 = 2*prec*rec/(prec+rec)
    return prec, rec, f1
```

```
Writing med_helpers.py
```

```
%%writefile medical_extraction_flow.py
# -*- coding: utf-8 -*-
from metaflow import FlowSpec, step, Parameter
import os, json, re
```

```
import pandas as pd
import numpy as np
import spacy
import matplotlib.pyplot as plt
from tqdm import tqdm

# HF/Transformers
import torch
from datasets import Dataset
from transformers import (
    AutoTokenizer, AutoModelForCausalLM, BitsAndBytesConfig,
    DataCollatorForLanguageModeling, TrainingArguments, Trainer, set_seed
)
from peft import LoraConfig, get_peft_model, prepare_model_for_kbit_training, PeftModel

import evaluate as hf_eval

from med_helpers import (
    FIELDS, norm_txt, build_prompt, safe_json, token_f1
)

class MedicalExtractionFlow(FlowSpec):
    data_path = Parameter("data_path", default="open_ave_data.csv")

    # model
    model_name = "meta-llama/Llama-3.2-1B"

    # training cfg (T4-friendly)
    max_seq_len = 768
    num_epochs  = 2
    lr          = 2e-4
    grad_accum  = 8
    train_bs    = 1
    eval_bs     = 1

    output_dir  = "/content/llama1b_ie_qlora"
    adapter_dir = "/content/llama1b_ie_qlora_adapter"

    @step
    def start(self):
        set_seed(42)
        df = pd.read_csv(self.data_path)

        # harmonize to your schema
        df = df.rename(columns={
            "ReportText": "text",
            "ExamName": "Examination",
            "clinicaldata": "Clinical",
            "findings": "Findings",
            "impression": "Impression"
        })
        df = df.dropna(subset=["text"]).reset_index(drop=True)
        df["clean"] = (df["text"].astype(str)
                       .str.replace(r"\s+", " ", regex=True)
                       .str.strip())

        for f in FIELDS:
            if f in df.columns:
                df[f] = df[f].astype(str).fillna("").apply(norm_txt)
            else:
                df[f] = ""

        # build target JSON (supervised pairs)
        def to_target_json(row):
            obj = {k: (row.get(k) or "") for k in FIELDS}
            return json.dumps(obj, ensure_ascii=False)
        df["target_json"] = df.apply(to_target_json, axis=1)

        # basic split
        n = len(df)
        val_frac = 0.15
        val_n = max(1, int(n*val_frac))
        self.train_df = df.iloc[:-val_n].reset_index(drop=True) if n>1 else df.copy()
        self.val_df   = df.iloc[-val_n:].reset_index(drop=True) if n>1 else df.copy()
        self.next(self.preprocess)

    @step
    def preprocess(self):
        # (kept) quick token stats
        nlp = spacy.blank("en")
        self.train_df["tokens"] = self.train_df["clean"].apply(lambda t: [tok.text for tok in nlp(t)])
        lengths = self.train_df["tokens"].str.len()
        plt.hist(lengths, bins=30)
        plt.xlabel("tokens"); plt.ylabel("reports")
        plt.savefig("lengths.png"); plt.close()
```

```
        self.next(self.train)

    @step
    def train(self):
        # Build SFT dataset: prompt -> target_json (teach exact format)
        def map_row(df):
            sources = []
            targets = []
            for t, y in zip(df["clean"].tolist(), df["target_json"].tolist()):
                sources.append(build_prompt(t))
                targets.append(y)
            return sources, targets

        tr_src, tr_tgt = map_row(self.train_df)
        va_src, va_tgt = map_row(self.val_df)

        train_ds = Dataset.from_dict({"prompt": tr_src, "text_target": tr_tgt})
        val_ds   = Dataset.from_dict({"prompt": va_src, "text_target": va_tgt})

        # tokenizer/model in 4-bit
        bnb = BitsAndBytesConfig(
            load_in_4bit=True,
            bnb_4bit_compute_dtype=torch.bfloat16,
            bnb_4bit_quant_type="nf4",
            bnb_4bit_use_double_quant=True
        )
        tok = AutoTokenizer.from_pretrained(self.model_name, use_auth_token=os.environ["HF_TOKEN"])
        if tok.pad_token is None:
            tok.pad_token = tok.eos_token

        model = AutoModelForCausalLM.from_pretrained(
            self.model_name, device_map="auto", quantization_config=bnb,
            use_auth_token=os.environ["HF_TOKEN"]
        )
        model = prepare_model_for_kbit_training(model)
        peft_cfg = LoraConfig(
            r=16, lora_alpha=32, lora_dropout=0.05,
            target_modules=["q_proj","k_proj","v_proj","o_proj","gate_proj","up_proj","down_proj"],
            bias="none", task_type="CAUSAL_LM"
        )
        model = get_peft_model(model, peft_cfg)

        # tokenization (prompt + target)
        def tok_fn(batch):
            texts = [p + " " + t + tok.eos_token for p,t in zip(batch["prompt"], batch["text_target"])]
            return tok(texts, max_length=self.max_seq_len, padding="longest", truncation=True)

        tok_tr = train_ds.map(tok_fn, batched=True, remove_columns=train_ds.column_names)
        tok_va = val_ds.map(tok_fn, batched=True, remove_columns=val_ds.column_names)

        collator = DataCollatorForLanguageModeling(tokenizer=tok, mlm=False)
        args = TrainingArguments(
            output_dir=self.output_dir,
            num_train_epochs=self.num_epochs,
            learning_rate=self.lr,
            per_device_train_batch_size=self.train_bs,
            per_device_eval_batch_size=self.eval_bs,
            gradient_accumulation_steps=self.grad_accum,
            logging_steps=20,
            evaluation_strategy="no",
            save_steps=300,
            save_total_limit=1,
            bf16=True,
            lr_scheduler_type="cosine",
            report_to="none"
        )

        trainer = Trainer(model=model, args=args, train_dataset=tok_tr,
                          eval_dataset=tok_va, data_collator=collator)
        trainer.train()

        os.makedirs(self.adapter_dir, exist_ok=True)
        trainer.save_model(self.adapter_dir)
        tok.save_pretrained(self.adapter_dir)

        self.next(self.infer)

    @step
    def infer(self):
        # Load base + adapter and generate JSON for validation set
        bnb = BitsAndBytesConfig(
            load_in_4bit=True,
            bnb_4bit_compute_dtype=torch.bfloat16,
            bnb_4bit_quant_type="nf4",
            bnb_4bit_use_double_quant=True
```

```
        )
        tok = AutoTokenizer.from_pretrained(self.model_name, use_auth_token=os.environ["HF_TOKEN"])
        if tok.pad_token is None:
            tok.pad_token = tok.eos_token
        base = AutoModelForCausalLM.from_pretrained(
            self.model_name, device_map="auto", quantization_config=bnb,
            use_auth_token=os.environ["HF_TOKEN"]
        )
        model = PeftModel.from_pretrained(base, self.adapter_dir)
        model.eval(); model.config.use_cache = True

        preds = {f"pred_{f}":[] for f in FIELDS}
        max_new = 256
        outs = []
        for report in tqdm(self.val_df["clean"].tolist(), desc="Generating"):
            prompt = build_prompt(report)
            inp = tok(prompt, return_tensors="pt").to(model.device)
            with torch.no_grad():
                out_ids = model.generate(
                    **inp, max_new_tokens=max_new,
                    do_sample=False, temperature=0.0,
                    pad_token_id=tok.pad_token_id, eos_token_id=tok.eos_token_id
                )
            gen = tok.decode(out_ids[0][inp["input_ids"].shape[-1]:], skip_special_tokens=True)
            outs.append(gen)
            js = safe_json(gen)
            for f in FIELDS:
                preds[f"pred_{f}"].append(norm_txt(js.get(f, "")))

        self.val_df = self.val_df.copy()
        for k,v in preds.items():
            self.val_df[k] = v
        self.gen_raw = outs
        self.next(self.evaluate)

    @step
    def evaluate(self):
        # Exact match + token P/R/F1 + ROUGE-L per field
        rouge = hf_eval.load("rouge")
        per_field = {}
        for f in FIELDS:
            gold = self.val_df[f].fillna("").tolist()
            pred = self.val_df[f"pred_{f}"].fillna("").tolist()

            # exact
            exact = np.mean([norm_txt(p)==norm_txt(g) for p,g in zip(pred,gold)])

            # token PRF1
            prfs = [token_f1(p,g) for p,g in zip(pred,gold)]
            prec = float(np.mean([x[0] for x in prfs]))
            rec  = float(np.mean([x[1] for x in prfs]))
            f1   = float(np.mean([x[2] for x in prfs]))

            # ROUGE-L F1
            # (compute per pair, average F1)
            rfs = []
            for p,g in zip(pred,gold):
                sc = rouge.compute(predictions=[p], references=[g])  # dict with rougeL etc.
                rfs.append(sc.get("rougeL", 0.0))
            rougeL = float(np.mean(rfs))

            per_field[f] = {
                "exact_match": float(exact),
                "token_precision": prec,
                "token_recall": rec,
                "token_f1": f1,
                "rougeL_f1": rougeL
            }

        # macro averages
        macro = {
            "exact_match": float(np.mean([per_field[f]["exact_match"] for f in FIELDS])),
            "token_precision": float(np.mean([per_field[f]["token_precision"] for f in FIELDS])),
            "token_recall": float(np.mean([per_field[f]["token_recall"] for f in FIELDS])),
            "token_f1": float(np.mean([per_field[f]["token_f1"] for f in FIELDS])),
            "rougeL_f1": float(np.mean([per_field[f]["rougeL_f1"] for f in FIELDS])),
        }

        metrics = {"per_field": per_field, "macro": macro}
        print(json.dumps(metrics, indent=2))

        # save artifacts
        out = self.val_df.copy()
        out["gen_raw"] = self.gen_raw
        out.to_csv("val_predictions.csv", index=False)
```

```
        with open("metrics.json","w") as f:
            json.dump(metrics, f, indent=2)
        self.next(self.end)

    @step
    def end(self):
        print("Flow complete. Artifacts: lengths.png, val_predictions.csv, metrics.json")
        print("Adapter dir:", self.adapter_dir)

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

```
Writing medical_extraction_flow.py
```

```
%%writefile medical_extraction_flow.py
# -*- coding: utf-8 -*-
from metaflow import FlowSpec, step, Parameter
import os, json, re
import pandas as pd
import numpy as np
import spacy
import matplotlib.pyplot as plt
from tqdm import tqdm

# HF/Transformers
import torch
from datasets import Dataset
from transformers import (
    AutoTokenizer, AutoModelForCausalLM, BitsAndBytesConfig,
    DataCollatorForLanguageModeling, TrainingArguments, Trainer, set_seed
)
from peft import LoraConfig, get_peft_model, prepare_model_for_kbit_training, PeftModel

import evaluate as hf_eval

from med_helpers import (
    FIELDS, norm_txt, build_prompt, safe_json, token_f1
)

class MedicalExtractionFlow(FlowSpec):
    data_path = Parameter("data_path", default="open_ave_data.csv")

    # model
    model_name = "meta-llama/Llama-3.2-1B"

    # training cfg (T4-friendly)
    max_seq_len = 768
    num_epochs  = 2
    lr          = 2e-4
    grad_accum  = 8
    train_bs    = 1
    eval_bs     = 1

    output_dir  = "/content/llama1b_ie_qlora"
    adapter_dir = "/content/llama1b_ie_qlora_adapter"

    @step
    def start(self):
        set_seed(42)
        df = pd.read_csv(self.data_path)

        # harmonize to your schema
        df = df.rename(columns={
            "ReportText": "text",
            "ExamName": "Examination",
            "clinicaldata": "Clinical",
            "findings": "Findings",
            "impression": "Impression"
        })
        df = df.dropna(subset=["text"]).reset_index(drop=True)
        df["clean"] = (df["text"].astype(str)
                       .str.replace(r"\s+", " ", regex=True)
                       .str.strip())

        for f in FIELDS:
            if f in df.columns:
                df[f] = df[f].astype(str).fillna("").apply(norm_txt)
            else:
                df[f] = ""

        # build target JSON (supervised pairs)
        def to_target_json(row):
            obj = {k: (row.get(k) or "") for k in FIELDS}
```

```
            return json.dumps(obj, ensure_ascii=False)
        df["target_json"] = df.apply(to_target_json, axis=1)

        # basic split
        n = len(df)
        val_frac = 0.15
        val_n = max(1, int(n*val_frac))
        self.train_df = df.iloc[:-val_n].reset_index(drop=True) if n>1 else df.copy()
        self.val_df   = df.iloc[-val_n:].reset_index(drop=True) if n>1 else df.copy()
        self.next(self.preprocess)

    @step
    def preprocess(self):
        # (kept) quick token stats
        nlp = spacy.blank("en")
        self.train_df["tokens"] = self.train_df["clean"].apply(lambda t: [tok.text for tok in nlp(t)])
        lengths = self.train_df["tokens"].str.len()
        plt.hist(lengths, bins=30)
        plt.xlabel("tokens"); plt.ylabel("reports")
        plt.savefig("lengths.png"); plt.close()
        self.next(self.train)

    @step
    def train(self):
        # Build SFT dataset: prompt -> target_json (teach exact format)
        def map_row(df):
            sources = []
            targets = []
            for t, y in zip(df["clean"].tolist(), df["target_json"].tolist()):
                sources.append(build_prompt(t))
                targets.append(y)
            return sources, targets

        tr_src, tr_tgt = map_row(self.train_df)
        va_src, va_tgt = map_row(self.val_df)

        train_ds = Dataset.from_dict({"prompt": tr_src, "text_target": tr_tgt})
        val_ds   = Dataset.from_dict({"prompt": va_src, "text_target": va_tgt})

        # tokenizer/model in 4-bit
        bnb = BitsAndBytesConfig(
            load_in_4bit=True,
            bnb_4bit_compute_dtype=torch.bfloat16,
            bnb_4bit_quant_type="nf4",
            bnb_4bit_use_double_quant=True
        )
        tok = AutoTokenizer.from_pretrained(self.model_name, use_auth_token=os.environ["HF_TOKEN"])
        if tok.pad_token is None:
            tok.pad_token = tok.eos_token

        model = AutoModelForCausalLM.from_pretrained(
            self.model_name, device_map="auto", quantization_config=bnb,
            use_auth_token=os.environ["HF_TOKEN"]
        )
        model = prepare_model_for_kbit_training(model)
        peft_cfg = LoraConfig(
            r=16, lora_alpha=32, lora_dropout=0.05,
            target_modules=["q_proj","k_proj","v_proj","o_proj","gate_proj","up_proj","down_proj"],
            bias="none", task_type="CAUSAL_LM"
        )
        model = get_peft_model(model, peft_cfg)

        # tokenization (prompt + target)
        def tok_fn(batch):
            texts = [p + " " + t + tok.eos_token for p,t in zip(batch["prompt"], batch["text_target"])]
            return tok(texts, max_length=self.max_seq_len, padding="longest", truncation=True)

        tok_tr = train_ds.map(tok_fn, batched=True, remove_columns=train_ds.column_names)
        tok_va = val_ds.map(tok_fn, batched=True, remove_columns=val_ds.column_names)

        collator = DataCollatorForLanguageModeling(tokenizer=tok, mlm=False)
        args = TrainingArguments(
            output_dir=self.output_dir,
            num_train_epochs=self.num_epochs,
            learning_rate=self.lr,
            per_device_train_batch_size=self.train_bs,
            per_device_eval_batch_size=self.eval_bs,
            gradient_accumulation_steps=self.grad_accum,
            logging_steps=20,
            eval_strategy="no",
            save_steps=300,
            save_total_limit=1,
            bf16=True,
            lr_scheduler_type="cosine",
            report_to="none"
```

```
        )

        trainer = Trainer(model=model, args=args, train_dataset=tok_tr,
                          eval_dataset=tok_va, data_collator=collator)
        trainer.train()

        os.makedirs(self.adapter_dir, exist_ok=True)
        trainer.save_model(self.adapter_dir)
        tok.save_pretrained(self.adapter_dir)

        self.next(self.infer)

    @step
    def infer(self):
        # Load base + adapter and generate JSON for validation set
        bnb = BitsAndBytesConfig(
            load_in_4bit=True,
            bnb_4bit_compute_dtype=torch.bfloat16,
            bnb_4bit_quant_type="nf4",
            bnb_4bit_use_double_quant=True
        )
        tok = AutoTokenizer.from_pretrained(self.model_name, use_auth_token=os.environ["HF_TOKEN"])
        if tok.pad_token is None:
            tok.pad_token = tok.eos_token
        base = AutoModelForCausalLM.from_pretrained(
            self.model_name, device_map="auto", quantization_config=bnb,
            use_auth_token=os.environ["HF_TOKEN"]
        )
        model = PeftModel.from_pretrained(base, self.adapter_dir)
        model.eval(); model.config.use_cache = True

        preds = {f"pred_{f}":[] for f in FIELDS}
        max_new = 256
        outs = []
        for report in tqdm(self.val_df["clean"].tolist(), desc="Generating"):
            prompt = build_prompt(report)
            inp = tok(prompt, return_tensors="pt").to(model.device)
            with torch.no_grad():
                out_ids = model.generate(
                    **inp, max_new_tokens=max_new,
                    do_sample=False, temperature=0.0,
                    pad_token_id=tok.pad_token_id, eos_token_id=tok.eos_token_id
                )
            gen = tok.decode(out_ids[0][inp["input_ids"].shape[-1]:], skip_special_tokens=True)
            outs.append(gen)
            js = safe_json(gen)
            for f in FIELDS:
                preds[f"pred_{f}"].append(norm_txt(js.get(f, "")))

        self.val_df = self.val_df.copy()
        for k,v in preds.items():
            self.val_df[k] = v
        self.gen_raw = outs
        self.next(self.evaluate)

    @step
    def evaluate(self):
        # Exact match + token P/R/F1 + ROUGE-L per field
        rouge = hf_eval.load("rouge")
        per_field = {}
        for f in FIELDS:
            gold = self.val_df[f].fillna("").tolist()
            pred = self.val_df[f"pred_{f}"].fillna("").tolist()

            # exact
            exact = np.mean([norm_txt(p)==norm_txt(g) for p,g in zip(pred,gold)])

            # token PRF1
            prfs = [token_f1(p,g) for p,g in zip(pred,gold)]
            prec = float(np.mean([x[0] for x in prfs]))
            rec  = float(np.mean([x[1] for x in prfs]))
            f1   = float(np.mean([x[2] for x in prfs]))

            # ROUGE-L F1
            # (compute per pair, average F1)
            rfs = []
            for p,g in zip(pred,gold):
                sc = rouge.compute(predictions=[p], references=[g])  # dict with rougeL etc.
                rfs.append(sc.get("rougeL", 0.0))
            rougeL = float(np.mean(rfs))

            per_field[f] = {
                "exact_match": float(exact),
                "token_precision": prec,
                "token_recall": rec,
```

```
                "token_f1": f1,
                "rougeL_f1": rougeL
            }

        # macro averages
        macro = {
            "exact_match": float(np.mean([per_field[f]["exact_match"] for f in FIELDS])),
            "token_precision": float(np.mean([per_field[f]["token_precision"] for f in FIELDS])),
            "token_recall": float(np.mean([per_field[f]["token_recall"] for f in FIELDS])),
            "token_f1": float(np.mean([per_field[f]["token_f1"] for f in FIELDS])),
            "rougeL_f1": float(np.mean([per_field[f]["rougeL_f1"] for f in FIELDS])),
        }

        metrics = {"per_field": per_field, "macro": macro}
        print(json.dumps(metrics, indent=2))

        # save artifacts
        out = self.val_df.copy()
        out["gen_raw"] = self.gen_raw
        out.to_csv("val_predictions.csv", index=False)
        with open("metrics.json","w") as f:
            json.dump(metrics, f, indent=2)
        self.next(self.end)

    @step
    def end(self):
        print("Flow complete. Artifacts: lengths.png, val_predictions.csv, metrics.json")
        print("Adapter dir:", self.adapter_dir)

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

```
Overwriting medical_extraction_flow.py
```

```
# Use your real CSV path here
!python medical_extraction_flow.py run --data_path /content/open_ave_data.csv
```

```
2025-08-12 19:20:59.509945: E external/local_xla/xla/stream_executor/cuda/cuda_fft.cc:467] Unable to register cuFFT factory:
WARNING: All log messages before absl::InitializeLog() is called are written to STDERR
E0000 00:00:1755026459.542006    7181 cuda_dnn.cc:8579] Unable to register cuDNN factory: Attempting to register factory for
E0000 00:00:1755026459.552147    7181 cuda_blas.cc:1407] Unable to register cuBLAS factory: Attempting to register factory f
W0000 00:00:1755026459.576375    7181 computation_placer.cc:177] computation placer already registered. Please check linkage
W0000 00:00:1755026459.576433    7181 computation_placer.cc:177] computation placer already registered. Please check linkage
W0000 00:00:1755026459.576441    7181 computation_placer.cc:177] computation placer already registered. Please check linkage
W0000 00:00:1755026459.576448    7181 computation_placer.cc:177] computation placer already registered. Please check linkage
2025-08-12 19:20:59.584802: I tensorflow/core/platform/cpu_feature_guard.cc:210] This TensorFlow binary is optimized to use
To enable the following instructions: AVX2 AVX512F FMA, in other operations, rebuild TensorFlow with the appropriate compile
Metaflow 2.17.1 executing MedicalExtractionFlow for user:colab
Validating your flow...
    The graph looks good!
Running pylint...
    Pylint not found, so extra checks are disabled.
2025-08-12 19:21:07.293 Workflow starting (run-id 1755026467292063):
2025-08-12 19:21:07.322 [1755026467292063/start/1 (pid 7302)] Task is starting.
2025-08-12 19:21:18.315 [1755026467292063/start/1 (pid 7302)] 2025-08-12 19:21:18.315632: E external/local_xla/xla/stream_ex
2025-08-12 19:21:18.336 [1755026467292063/start/1 (pid 7302)] WARNING: All log messages before absl::InitializeLog() is call
2025-08-12 19:21:18.343 [1755026467292063/start/1 (pid 7302)] E0000 00:00:1755026478.335964    7302 cuda_dnn.cc:8579] Unable
2025-08-12 19:21:18.343 [1755026467292063/start/1 (pid 7302)] E0000 00:00:1755026478.343012    7302 cuda_blas.cc:1407] Unabl
2025-08-12 19:21:18.358 [1755026467292063/start/1 (pid 7302)] W0000 00:00:1755026478.358892    7302 computation_placer.cc:17
2025-08-12 19:21:23.554 [1755026467292063/start/1 (pid 7302)] W0000 00:00:1755026478.358916    7302 computation_placer.cc:17
2025-08-12 19:21:23.554 [1755026467292063/start/1 (pid 7302)] W0000 00:00:1755026478.358919    7302 computation_placer.cc:17
2025-08-12 19:21:23.555 [1755026467292063/start/1 (pid 7302)] W0000 00:00:1755026478.358921    7302 computation_placer.cc:17
2025-08-12 19:21:23.555 [1755026467292063/start/1 (pid 7302)] Task finished successfully.
2025-08-12 19:21:23.560 [1755026467292063/preprocess/2 (pid 7379)] Task is starting.
2025-08-12 19:21:31.383 [1755026467292063/preprocess/2 (pid 7379)] 2025-08-12 19:21:31.383381: E external/local_xla/xla/stre
2025-08-12 19:21:31.402 [1755026467292063/preprocess/2 (pid 7379)] WARNING: All log messages before absl::InitializeLog() is
2025-08-12 19:21:31.408 [1755026467292063/preprocess/2 (pid 7379)] E0000 00:00:1755026491.402739    7379 cuda_dnn.cc:8579] U
2025-08-12 19:21:31.409 [1755026467292063/preprocess/2 (pid 7379)] E0000 00:00:1755026491.408761    7379 cuda_blas.cc:1407]
2025-08-12 19:21:31.423 [1755026467292063/preprocess/2 (pid 7379)] W0000 00:00:1755026491.423492    7379 computation_placer.
2025-08-12 19:21:40.185 [1755026467292063/preprocess/2 (pid 7379)] W0000 00:00:1755026491.423515    7379 computation_placer.
2025-08-12 19:21:40.185 [1755026467292063/preprocess/2 (pid 7379)] W0000 00:00:1755026491.423518    7379 computation_placer.
2025-08-12 19:21:40.185 [1755026467292063/preprocess/2 (pid 7379)] W0000 00:00:1755026491.423520    7379 computation_placer.
2025-08-12 19:21:40.186 [1755026467292063/preprocess/2 (pid 7379)] Task finished successfully.
2025-08-12 19:21:40.191 [1755026467292063/train/3 (pid 7460)] Task is starting.
2025-08-12 19:21:47.451 [1755026467292063/train/3 (pid 7460)] 2025-08-12 19:21:47.451777: E external/local_xla/xla/stream_ex
2025-08-12 19:21:47.480 [1755026467292063/train/3 (pid 7460)] WARNING: All log messages before absl::InitializeLog() is call
2025-08-12 19:21:47.486 [1755026467292063/train/3 (pid 7460)] E0000 00:00:1755026507.480302    7460 cuda_dnn.cc:8579] Unable
2025-08-12 19:21:47.486 [1755026467292063/train/3 (pid 7460)] E0000 00:00:1755026507.486408    7460 cuda_blas.cc:1407] Unabl
2025-08-12 19:21:47.501 [1755026467292063/train/3 (pid 7460)] W0000 00:00:1755026507.501805    7460 computation_placer.cc:17
2025-08-12 19:22:00.065 [1755026467292063/train/3 (pid 7460)] W0000 00:00:1755026507.501827    7460 computation_placer.cc:17
2025-08-12 19:22:00.065 [1755026467292063/train/3 (pid 7460)] W0000 00:00:1755026507.501830    7460 computation_placer.cc:17
2025-08-12 19:22:00.065 [1755026467292063/train/3 (pid 7460)] W0000 00:00:1755026507.501833    7460 computation_placer.cc:17
2025-08-12 19:22:00.065 [1755026467292063/train/3 (pid 7460)] Parameter 'function'=<function MedicalExtractionFlow.train.<lo
2025-08-12 19:22:00.066 [1755026467292063/train/3 (pid 7460)] WARNING:datasets.fingerprint:Parameter 'function'=<function Me
Map: 100%|██████████| 811/811 [00:01<00:00, 633.07 examples/s]
Map: 100%|██████████| 143/143 [00:00<00:00, 692.88 examples/s]
```

```
2025-08-12 19:22:02.265 [1755026467292063/train/3 (pid 7460)] 0%|          | 0/204 [00:00<?, ?it/s]`use_cache=True` is incom
```

Start coding or generate with AI.

Start coding or generate with AI.

Start coding or generate with AI.

Start coding or generate with AI.