```
# === Cell 1: Install dependencies ===
# Run this cell in Colab
!pip install openai beautifulsoup4 requests python-dotenv pydantic==1.10.12 tqdm python-Levenshtein

# Note: python-Levenshtein speeds fuzzy matching; if it fails to build, the fallback uses difflib.
# === Cell 2: Imports & config ===
import os
import re
import json
import csv
import time
from typing import List, Optional, Dict, Any
from dataclasses import dataclass
from urllib.parse import urlparse
import requests
from bs4 import BeautifulSoup
from pydantic import BaseModel, Field, ValidationError, validator
from tqdm import tqdm

# LLM
import openai

# fuzzy
try:
    import Levenshtein
    def similarity(a,b):
        if not a and not b: return 1.0
        return Levenshtein.ratio(a,b)
except Exception:
    import difflib
    def similarity(a,b):
        return difflib.SequenceMatcher(None, a, b).ratio()

# Configure OpenAI key (set env var in Colab or uncomment below)
# os.environ["OPENAI_API_KEY"] = "sk-..."
openai.api_key = os.getenv("OPENAI_API_KEY")
if not openai.api_key:
    raise RuntimeError("Set OPENAI_API_KEY in Colab environment variables before running.")
# === Cell 3: URLs to scrape (from screenshot) ===
URLS = [
 "https://en.wikipedia.org/wiki/Sustainable_agriculture",
 "https://www.nature.com/articles/d41586-025-03353-5",
 "https://www.sciencedirect.com/science/article/pii/S1043661820315152",
 "https://www.ncbi.nlm.nih.gov/pmc/articles/PMC10457221/",
 "https://www.fao.org/3/y4671e/y4671e06.htm",
 "https://www.medscape.com/viewarticle/time-reconsider-tramadol-chronic-pain-205a1000r1a",
 "https://www.sciencedirect.com/science/article/pii/S0378378202037088",
 "https://www.frontiersin.org/news/2025/09/01/rectangle-telescope-finding-habitable-planets",
 "https://www.medscape.com/viewarticle/second-dose-boosts-shingles-protection-adults-aged-65-years-2025a1000r7",
 "https://www.theguardian.com/global-development/2025/oct/13/astro-ambassadors-stargazers-himalayas-hanle-ladakh-india"
]
# === Cell 4: Pydantic schema for LLM output ===
class ExtractedEntity(BaseModel):
    entity: str = Field(..., description="Exact entity string (no truncation)")
    tag: str = Field(..., description="Semantic category, e.g., Crop, Process, Measurement, Concept")
    span: Optional[str] = Field(None, description="Optional short sentence fragment where it appears")
    confidence: float = Field(..., ge=0.0, le=1.0, description="LLM's confidence estimate 0..1")

    @validator("entity")
    def not_empty(cls, v):
        v2 = v.strip()
        if not v2:
            raise ValueError("entity must not be empty")
        return v2

class ExtractedOutput(BaseModel):
    link: str
    entities: List[ExtractedEntity]
# === Cell 5: Utilities - scraping & text extraction ===
def fetch_text_from_url(url, timeout=15):
    """Fetch and extract main text from a URL (very simple: grabs <p> text)."""
    headers = {"User-Agent": "Mozilla/5.0 (compatible; Colabbot/1.0)"}
    r = requests.get(url, headers=headers, timeout=timeout)
    r.raise_for_status()
    soup = BeautifulSoup(r.text, "html.parser")
    # Remove scripts/styles
    for s in soup(["script","style","noscript","iframe"]):
        s.decompose()
    # Heuristic: gather large <p> blocks and headlines
    texts = []
    for tag in soup.find_all(["h1","h2","h3","p"]):
        txt = tag.get_text(separator=" ", strip=True)
        if txt and len(txt) > 20:
            texts.append(txt)
    full = "\n\n".join(texts)
    # Trim very long pages to first 80k chars to avoid LLM context issues
    return full[:80000]

# quick sanitizer for node labels
def trim_label(s, max_chars=40):
    s = s.strip()
    if len(s) <= max_chars:
        return s
    else:
        return s[:max_chars-3].rstrip() + "..."
# === Cell 6: LLM extraction prompt + wrapper ===
LLM_MODEL = "gpt-4o-mini"  # change to your available model; or "gpt-4" / "gpt-4o" etc.
# If you only have gpt-3.5, set model accordingly and maybe increase max tokens.
```

```
def build_extraction_prompt(url, text, max_entities=30):
    system = (
        "You are a precise information extraction assistant. "
        "Given a web page's text, extract notable entities relevant to knowledge graphs: "
        "things like crops, processes, chemicals, diseases, instruments, concepts, locations, measurements, organisms, people, organizations, etc."
        "Return EXACT JSON with keys: link, entities (list). Each entity object must contain: entity (exact string), tag (semantic category), span (short context snippet or sentence), confidence (0.0-1.0)."
        "Rules: do not invent facts. If unsure about the category, choose 'Concept'."
        "Return only the JSON -- nothing else."
    )
    user = {
        "url": url,
        "text": text,
        "instructions": f"Return at most {max_entities} entities. Entities should represent distinct concepts or things worthy of nodes. For repeated mentions, include once with best confidence estimate."
    }
    # We'll present text as-is (truncated earlier)
    return system, json.dumps(user)


def call_llm_for_extraction(system_prompt, user_content, max_tokens=1200, retries=3, backoff=2.0):
    """
    Calls the OpenAI chat completion endpoint and returns parsed JSON.
    We'll retry a couple times if the LLM returns non-JSON or invalid schema.
    """
    for attempt in range(1, retries+1):
        try:
            response = openai.ChatCompletion.create(
                model=LLM_MODEL,
                messages=[
                    {"role":"system", "content": system_prompt},
                    {"role":"user", "content": user_content}
                ],
                temperature=0.0,
                max_tokens=max_tokens,
            )
            text = response["choices"][0]["message"]["content"].strip()
            # Ensure it's JSON
            parsed = json.loads(text)
            return parsed
        except Exception as e:
            print(f"[LLM attempt {attempt}] parse/LLM error: {e}")
            if attempt < retries:
                time.sleep(backoff * attempt)
            else:
                raise
# === Cell 7: Parse + validate LLM output into our schema ===
def validate_extraction(parsed_json, url):
    """
    Ensure parsed JSON conforms to ExtractedOutput pydantic schema.
    If the LLM omits confidences, we'll set a default conservative value (0.7).
    """
```

```
# Normalize structure
if "link" not in parsed_json:
    parsed_json["link"] = url
if "entities" not in parsed_json:
    parsed_json["entities"] = []
# Ensure each entity has confidence
for ent in parsed_json["entities"]:
    if "confidence" not in ent:
        ent["confidence"] = 0.7
    else:
        # clamp
        try:
            ent["confidence"] = float(ent["confidence"])
        except:
            ent["confidence"] = 0.7
        if ent["confidence"] < 0: ent["confidence"] = 0.0
        if ent["confidence"] > 1: ent["confidence"] = 1.0
try:
    eo = ExtractedOutput(link=parsed_json["link"], entities=parsed_json["entities"])
    return eo
except ValidationError as e:
    # Try to be forgiving: attempt to coerce minimal fields
    final = {"link": url, "entities": []}
    for ent in parsed_json.get("entities", []):
        try:
                ee = ExtractedEntity(
                    entity=str(ent.get("entity","")).strip(),
                    tag=str(ent.get("tag","Concept")),
                    span=str(ent.get("span",""))[:300] if ent.get("span") else None,
                    confidence=float(ent.get("confidence", 0.7))
                )
                final["entities"].append(ee)
            except Exception:
                continue
        return ExtractedOutput(link=final["link"], entities=final["entities"])
# === Cell 8: Deduplication with confidence loop ===
def normalize_ent(s):
    s2 = s.lower()
    s2 = re.sub(r'[^a-z0-9\s\-]', '', s2)
    s2 = re.sub(r'\s+', ' ', s2).strip()
    return s2

def deduplicate_entities(entities: List[ExtractedEntity], similarity_threshold=0.85):
    """
    Aggressive dedup:
      - exact normalized match => merge, take max confidence
      - fuzzy similarity >= threshold => ask LLM to confirm merge or not (simulate via automatic merge)
    Returns deduped list.
    """
    dedup = []
    seen = []
    for ent in entities:
        norm = normalize_ent(ent.entity)
        merged = False
        for i, existing in enumerate(dedup):
            norm2 = normalize_ent(existing.entity)
            sim = similarity(norm, norm2)
            if sim >= similarity_threshold:
                # merge: choose the longer label as canonical, max confidence, tags merged by priority
                chosen_entity = existing.entity if len(existing.entity) >= len(ent.entity) else ent.entity
                chosen_conf = max(existing.confidence, ent.confidence)
                # simple tag resolution: prefer non-Concept, or keep existing
                chosen_tag = existing.tag if existing.tag != "Concept" else ent.tag
                chosen_span = existing.span or ent.span
                dedup[i] = ExtractedEntity(entity=chosen_entity, tag=chosen_tag, span=chosen_span, confidence=chosen_conf)
                merged = True
                break
        if not merged:
            dedup.append(ent)
    return dedup


def deduplicate_with_confidence_loop(url, initial_output: ExtractedOutput, target_confidence=0.9, max_rounds=3):
    """
    Repeatedly ask LLM to re-check/clean/deduplicate until all entities >= target_confidence or max_rounds reached.
```

```
Implementation: for simplicity we call the LLM with current entity list and ask to:
  - remove duplicates
  - increase confidence only if justified (we trust LLM judgment)
"""
current = initial_output
for round_i in range(1, max_rounds+1):
    # quick local dedup first
    current.entities = deduplicate_entities(current.entities, similarity_threshold=0.9)
    if all(e.confidence >= target_confidence for e in current.entities):
        print(f"All entities reached target confidence after {round_i-1} rounds.")
        return current
    # build a small prompt describing current list and ask to re-evaluate/merge with confidences
    system = (
        "You are an expert data curator. Given a list of entities extracted from the URL, "
        "deduplicate aggressively, validate that each entity is real and present in the text, and "
        "return JSON with entities list where confidence is a justified 0..1 value. "
        "If an entity should be removed (not a real separate concept), drop it."
        "Return only JSON: {link:..., entities:[{entity,tag,span,confidence}, ...]}"
    )
    payload = {"link": url, "entities":[e.dict() for e in current.entities]}
    try:
        response = openai.ChatCompletion.create(
            model=LLM_MODEL,
            messages=[
                {"role":"system","content":system},
                    {"role":"user","content":json.dumps(payload)}
                ],
                temperature=0.0,
                max_tokens=800,
            )
            text = response["choices"][0]["message"]["content"].strip()
            parsed = json.loads(text)
            current = validate_extraction(parsed, url)
        except Exception as e:
            print(f"[round {round_i}] LLM re-eval failed, stopping: {e}")
            break
    return current
# === Cell 9: Mermaid generation ===
def generate_mermaid_from_entities(url, extracted: ExtractedOutput, max_edge_label_len=40):
    """
    Simple heuristic: create nodes for each entity and connect:
      - connect page root -> each entity
      - connect entities that share tag types in common (simple grouping edges)
    Produce a Mermaid "graph LR" string.
    """
    lines = ["graph LR"]
    page_node = f"page_{abs(hash(url)) % (10**8)}"
    page_label = trim_label(url, 60)
    lines.append(f'{page_node}["{page_label}"]')
    # create node ids
```

````
    nodes = []
    for i, ent in enumerate(extracted.entities):
        node_id = f"n{i}_{abs(hash(ent.entity)) % 10000}"
        label = trim_label(ent.entity, max_edge_label_len)
        lines.append(f'{node_id}["{label}"]')
        lines.append(f"{page_node} --> {node_id}")
        nodes.append((node_id, ent))
    # add some intra-entity edges: if tags match and not exactly same entity
    for i in range(len(nodes)):
        for j in range(i+1, len(nodes)):
            ent_i = nodes[i][1]
            ent_j = nodes[j][1]
            if ent_i.tag == ent_j.tag and ent_i.tag != "Concept" and similarity(normalize_ent(ent_i.entity),
normalize_ent(ent_j.entity)) < 0.95:
                lines.append(f"{nodes[i][0]} ---|{ent_i.tag}| {nodes[j][0]}")
    return "\n".join(lines)


def save_mermaid_file(url, mermaid_str, out_dir="/content/mermaids"):
    os.makedirs(out_dir, exist_ok=True)
    safe = re.sub(r'[^a-zA-Z0-9\-_.]', '_', url)[:120]
    fname = os.path.join(out_dir, f"mermaid_{safe}.md")
    with open(fname, "w", encoding="utf-8") as f:
        f.write("```mermaid\n")
        f.write(mermaid_str)
        f.write("\n```\n")
    return fname
````

```
# === Cell 10: CSV write helper ===
def write_tags_csv(all_extracted: List[ExtractedOutput], csv_path="/content/tags.csv"):
    """
    CSV columns: link, tag, entity, tag_type
    tag_type in assignment appears to be same as 'tag' semantic category.
    We'll write: link, entity, tag, tag_type (same as tag) for clarity.
    """
    with open(csv_path, "w", newline="", encoding="utf-8") as csvfile:
        writer = csv.writer(csvfile)
        writer.writerow(["link","entity","tag","tag_type"])
        for eo in all_extracted:
            for e in eo.entities:
                writer.writerow([eo.link, e.entity, e.tag, e.tag])
    return csv_path
# === Cell 11: End-to-end pipeline ===
OUTPUTS = []
mermaid_files = []

for url in tqdm(URLS, desc="Processing URLs"):
    try:
        print(f"\n---\nFetching text for: {url}")
        text = fetch_text_from_url(url)
        if not text or len(text) < 100:
            print("Warning: page text very short or empty; skipping.")
            continue
        system, user_content = build_extraction_prompt(url, text)
        parsed = call_llm_for_extraction(system, user_content, max_tokens=1600, retries=3)
        extracted = validate_extraction(parsed, url)
        # deduplicate loop
        extracted = deduplicate_with_confidence_loop(url, extracted, target_confidence=0.9, max_rounds=2)
        # final local dedup safety
        extracted.entities = deduplicate_entities(extracted.entities, similarity_threshold=0.9)
        # save
        OUTPUTS.append(extracted)
        mer = generate_mermaid_from_entities(url, extracted)
        mfile = save_mermaid_file(url, mer)
        mermaid_files.append(mfile)
        print(f"Saved mermaid to {mfile}; {len(extracted.entities)} entities.")
    except Exception as e:
        print(f"Error processing {url}: {e}")
# === Cell 12: Save CSV and show summary ===
csv_path = write_tags_csv(OUTPUTS, "/content/tags.csv")
print("Wrote CSV:", csv_path)
print("Mermaid files:", mermaid_files[:5], " ... total", len(mermaid_files))
# Print short sample of CSV
with open(csv_path, "r", encoding="utf-8") as f:
    lines = f.readlines()
print("--- CSV sample ---")
print("".join(lines[:20]))
# === Cell 13: Optional: create a Colab-friendly ZIP to download ===
import zipfile
zipf = "/content/assignment_outputs.zip"
with zipfile.ZipFile(zipf, "w") as z:
    z.write(csv_path, arcname="tags.csv")
    for mf in mermaid_files:
        z.write(mf, arcname=os.path.basename(mf))
print("Created zip:", zipf)
```