Homework 2

Set up

Installing packages

```
!pip install requests PyPDF2 gdown
!pip install 'markitdown[pdf]'
!pip install langchain_mcp_adapters langchain_google_genai langchain-openai
```

显示隐藏的输出项

Setup your API key

To run the following cell, your API key must be stored it in a Colab Secret named `VERTEX_API_KEY`.

1. Look for the key icon on the left panel of your colab.
2. Under `Name`, create `VERTEX_API_KEY`.
3. Copy your key to `Value`.

If you cannot use VERTEX_API_KEY, you can use deepseek models via `DEEPSEEK_API_KEY`. It does not affect your score.

```
from google.colab import userdata
DEEPSEEK_API_KEY = userdata.get('DEEPSEEK_API_KEY')
```

Download sample CVs

Downloading sample_cv.pdf

The codes below download the sample CV

```
import os
import gdown

folder_id = "1adYKq7gSSczFP3iikfA8Er-HSZP6VM7D"
folder_url = f"https://drive.google.com/drive/folders/{folder_id}"

output_dir = "downloaded_cvs"
os.makedirs(output_dir, exist_ok=True)

gdown.download_folder(
    url=folder_url,
    output=output_dir,
    quiet=False,
    use_cookies=False
)
```

显示隐藏的输出项

```
# ==================================================
#  Load and display all CV PDFs in order
# ==================================================
import os
from markitdown import MarkItDown

cv_dir = "downloaded_cvs"

# Initialize MarkItDown
md = MarkItDown(enable_plugins=False)

# Collect and sort PDFs numerically
pdf_files = sorted(
    [f for f in os.listdir(cv_dir) if f.lower().endswith(".pdf")],
    key=lambda x: int("".join(filter(str.isdigit, x)))  # CV_1.pdf → 1
)
```

```
all_cvs = []

for pdf_name in pdf_files:
    pdf_path = os.path.join(cv_dir, pdf_name)
    result = md.convert(pdf_path)

    all_cvs.append({
        "file": pdf_name,
        "text": result.text_content
    })

    print("=" * 80)
    print(f"📄 {pdf_name}")
    print("=" * 80)
    print(result.text_content)
    print("\n\n")
```

显示隐藏的输出项

Connect to our MCP server

Documentation about MCP: https://modelcontextprotocol.io/docs/getting-started/intro.

Using MCP servers in Langchain https://docs.langchain.com/oss/python/langchain/mcp.

Check which tools that the MCP server provide

```
import asyncio
import json
from langchain_mcp_adapters.client import MultiServerMCPClient

client = MultiServerMCPClient({
    "social_graph": {
        "transport": "http",
        "url": "https://ftec5660.ngrok.app/mcp",
        "headers": {"ngrok-skip-browser-warning": "true"}
    }
})

mcp_tools = await client.get_tools()
for tool in mcp_tools:
    print(tool.name)
    print(tool.description)
    print(tool.args)
    print("\n\n-----------------------------------------------------\n\n")
```

显示隐藏的输出项

A simple agent using tools from the MCP server

```
from google.colab import userdata
from langchain_openai import ChatOpenAI

DEEPSEEK_API_KEY = userdata.get("DEEPSEEK_API_KEY")

llm = ChatOpenAI(
    model="deepseek-chat",
    api_key=DEEPSEEK_API_KEY,
    base_url="https://api.deepseek.com/v1",
    temperature=0,
)
```

```
import json
import re

def _tool_to_obj(tool_result):

    if isinstance(tool_result, list) and tool_result and isinstance(tool_result[0], dict) and "text" in tool_result[0]:
        txt = tool_result[0]["text"]
        try:
            return json.loads(txt)
        except:
            return txt
    return tool_result
```

```python
def _pick_best_candidate(items, name_key="name"):
    """
    Select the most similar profile by giving priority to exact matches. If multiple candidates remain, choose the
    """
    if not items:
        return None
    return sorted(
        items,
        key=lambda x: (x.get("match_type") != "exact", -x.get("years_experience", 0))
    )[0]

async def fetch_linkedin_best(tools, name: str, location_hint: str | None):
    t_search = next(t for t in tools if t.name == "search_linkedin_people")
    t_get    = next(t for t in tools if t.name == "get_linkedin_profile")

    # 1) Start by searching with the location included
    r1 = await t_search.ainvoke({
        "q": name,
        "location": location_hint,
        "industry": None,
        "limit": 10,
        "fuzzy": True
    })
    people = _tool_to_obj(r1)

    # 2) If no result is found, remove the location and search again so that we return the most similar match
    if not people:
        r2 = await t_search.ainvoke({
            "q": name,
            "location": None,
            "industry": None,
            "limit": 10,
            "fuzzy": True
        })
        people = _tool_to_obj(r2)

    if not people:
        return None

    best = _pick_best_candidate(people)
    prof_raw = await t_get.ainvoke({"person_id": best["id"]})
    profile = _tool_to_obj(prof_raw)
    return {"candidate": best, "profile": profile}

async def fetch_facebook_best(tools, name: str):
    t_search = next(t for t in tools if t.name == "search_facebook_users")
    t_get    = next(t for t in tools if t.name == "get_facebook_profile")

    r1 = await t_search.ainvoke({"q": name, "limit": 10, "fuzzy": True})
    users = _tool_to_obj(r1)
    if not users:
        return None

    # For Facebook, prioritize exact matches because years of experience are unavailable.
    best = sorted(users, key=lambda u: (u.get("match_type") != "exact"))[0]
    prof_raw = await t_get.ainvoke({"user_id": best["id"]})
    profile = _tool_to_obj(prof_raw)
    return {"candidate": best, "profile": profile}
```

```python
import re
import json
from langchain_core.messages import HumanMessage


def extract_year_hints(cv_text: str, max_hints: int = 12) -> list[str]:
    t = cv_text or ""

    # capture "YYYY - YYYY" or "YYYY - Present"
    pattern = re.compile(r"\b((?:19|20)\d{2})\s*[--]\s*((?:19|20)\d{2}|present)\b", re.IGNORECASE)
    range_strs = []
    for m in pattern.finditer(t):
        a = m.group(1)
        b = m.group(2)
        range_strs.append(f"{a}-{b}")

    # also capture single years (YYYY)
    years = re.findall(r"\b(?:19|20)\d{2}\b", t)

    # combine + dedupe
    seen = set()
    out = []
```

```python
    for x in range_strs + years:
        x = x.strip()
        if x and x not in seen:
            seen.add(x)
            out.append(x)

    return out[:max_hints]

def extract_cv_json(llm, cv_text: str) -> dict:
    year_hints = extract_year_hints(cv_text)

    prompt = f"""
You are an information extraction system. Return STRICT JSON only (no markdown).
Your job is to extract structured fields from a CV.

IMPORTANT RULES:
1) For every experience item, include start_year and end_year if stated. If not stated, use null.
2) If the CV says "Present", set end_year to null and is_current=true.
3) Keep company/title/school strings as they appear (do not invent).

Return JSON with this exact schema:
{{
  "name": "",
  "city": "",
  "country": "",
  "education": [{{"school":"","degree":"","field":"","start_year":null,"end_year":null}}],
  "experience": [{{"company":"","title":"","start_year":null,"end_year":null,"is_current":false}}],
  "skills": []
}}

Year hints extracted from the CV (may help you locate dates): {year_hints}

CV TEXT:
{cv_text}
"""
    resp = llm.invoke([HumanMessage(content=prompt)])
    text = resp.content
    start = text.find("{")
    end = text.rfind("}")
    data = json.loads(text[start:end+1])
    return data
import pandas as pd
import re

def summarize_cv(cv: dict) -> dict:
    exp = cv.get("experience") or []
    edu = cv.get("education") or []
    # count years coverage
    exp_with_years = sum(1 for e in exp if isinstance(e.get("start_year"), int) or isinstance(e.get("end_year"), int))
    edu_with_years = sum(1 for e in edu if isinstance(e.get("start_year"), int) or isinstance(e.get("end_year"), int))

    return {
        "name": cv.get("name"),
        "city": cv.get("city"),
        "country": cv.get("country"),
        "n_exp": len(exp),
        "n_exp_with_years": exp_with_years,
        "n_edu": len(edu),
        "n_edu_with_years": edu_with_years,
        "n_skills": len(cv.get("skills") or []),
    }

def audit_extraction(all_cvs, llm):
    rows = []
    cv_objects = []
    for item in all_cvs:
        cv = extract_cv_json(llm, item["text"])
        cv_objects.append(cv)
        row = {"file": item["file"], **summarize_cv(cv)}
        rows.append(row)
    df = pd.DataFrame(rows)
    return df, cv_objects

df_extract, extracted_cvs = audit_extraction(all_cvs, llm)
df_extract
def normalize_cv(cv: dict, raw_text: str) -> dict:
    """
    Postprocess to improve year coverage:
    - If experience has company/title but missing years, try to find a nearby year range in raw_text.
    This is heuristic but helps avoid CV~1 artifacts.
    """
    t = raw_text.replace("-", "-").replace("—", "-")
    year_range_pat = re.compile(r"\b(?:19|20)\d{2}\s*-\s*((?:19|20)\d{2}|present)\b", re.IGNORECASE)
    year_range_pat = re.compile(r"\b(?:19|20)\d{2}\b\s*-\s*\b((?:19|20)\d{2}|present)\b", re.IGNORECASE)

    # If many experience items missing years, try a coarse fallback:
    ranges = year_range_pat.findall(t)
    # ranges are tuples; reconstruct
    range_strs = re.findall(r"\b(?:19|20)\d{2}\s*-\s*((?:19|20)\d{2}|present)\b", t, flags=re.IGNORECASE)
    # easier: directly capture as strings
    range_strs = re.findall(r"\b((?:19|20)\d{2})\s*-\s*((?:19|20)\d{2}|present)\b", t, flags=re.IGNORECASE)

    exp = cv.get("experience") or []
    missing = [i for i,e in enumerate(exp) if (e.get("company") or e.get("title")) and e.get("start_year") is None]
    if missing and range_strs:
        # assign ranges in order to missing slots (best-effort)
        for idx, (sy, ey) in zip(missing, range_strs):
            try:
                exp[idx]["start_year"] = int(sy)
            except:
                pass
            if str(ey).lower() == "present":
                exp[idx]["end_year"] = None
                exp[idx]["is_current"] = True
            else:
                try:
                    exp[idx]["end_year"] = int(ey)
                except:
                    pass
    cv["experience"] = exp
    return cv

def internal_inconsistency_penalty(cv: dict) -> tuple[float, list[str]]:
    """
    TA requirement: internal contradictions should only result in a score deduction and must not trigger an early r
Here we apply a light penalty (up to 0.15) to avoid mistakenly rejecting a genuine CV.
    """
    issues = []
    penalty = 0.0

    # Time overlap: if multiple roles are marked as current or the dates clearly conflict.
    exps = cv.get("experience", []) or []
    current_count = sum(1 for e in exps if e.get("is_current") is True)
    if current_count >= 2:
        issues.append("Multiple current jobs in CV (possible timeline inconsistency).")
        penalty += 0.08

    # Location: city/country is missing, but multiple place names are detected.
    if (not cv.get("city") or not cv.get("country")) and len(re.findall(r"\b(Hong Kong|Singapore|Tokyo|London|Kowloon|Philip
        issues.append("Ambiguous/mixed location signals in CV (weak internal inconsistency).")
        penalty += 0.04

    # maximum
    penalty = min(penalty, 0.15)
    return penalty, issues

def compute_score(cv: dict, li: dict | None, fb: dict | None) -> tuple[float, list[str]]:
    import json

    issues = []
    major = 0
    minor = 0

    def _soft_mismatch(a, b):
        a = (a or "").lower().strip()
        b = (b or "").lower().strip()
        if not a or not b:
            return False
        return (a not in b) and (b not in a)

    def _latest_role(exps):
        exps = exps or []
        exps_sorted = sorted(
            exps,
            key=lambda e: (e.get("start_year") is None, -(e.get("start_year") or 0))
        )
        return exps_sorted[0] if exps_sorted else None

    # ==============================
    # LinkedIn (primary verification)
    # ==============================

    li_prof = None
    industry_conflict = False

    if not li:
```

```
        issues.append("No LinkedIn profile found after fallback search.")
        minor += 1
    else:
        li_prof = li["profile"]
        match_type = (li.get("candidate") or {}).get("match_type")
        is_exact = (match_type == "exact")

        cv_exps = cv.get("experience") or []
        li_exps = li_prof.get("experience") or []

        cv_role = next((e for e in cv_exps if e.get("is_current")), None) or _latest_role(cv_exps)
        li_role = next((e for e in li_exps if e.get("is_current")), None) or _latest_role(li_exps)

        company_mis = False
        title_mis = False

        if cv_role and li_role:
            if _soft_mismatch(cv_role.get("company"), li_role.get("company")):
                issues.append(
                    f"Current/Latest company mismatch: CV={cv_role.get('company')} vs LI={li_role.get('company')
                )
                company_mis = True

            if _soft_mismatch(cv_role.get("title"), li_role.get("title")):
                issues.append(
                    f"Current/Latest title mismatch: CV={cv_role.get('title')} vs LI={li_role.get('title')}"
                )
                title_mis = True

        # Title mismatch is weak signal
        if title_mis:
            minor += 1
            issues.append("Treated title mismatch as minor (title wording/noise or injected inconsistency).")

        # Education mismatch (weak)
        cv_schools = {
            e.get("school", "").lower().strip()
            for e in (cv.get("education") or [])
            if e.get("school")
        }

        li_schools = {
            e.get("school", "").lower().strip()
            for e in (li_prof.get("education") or [])
            if e.get("school")
        }

        edu_mis = bool(cv_schools and li_schools and cv_schools.isdisjoint(li_schools))

        if edu_mis:
            issues.append("Education schools mismatch between CV and LinkedIn.")
            if is_exact:
                minor += 1
            else:
                issues.append("Education mismatch not penalized because LinkedIn match is not exact.")

        # ------------------------------
        # Robust industry/domain mismatch
        # ------------------------------

        li_text = f"{li_prof.get('headline','')} {li_prof.get('industry','')}".lower()
        cv_text = json.dumps(cv).lower()

        # domain keyword sets
        legal_kw = ["legal", "law", "litigation", "compliance", "contract", "paralegal"]
        tech_kw  = ["engineer", "software", "developer", "data", "scientist", "machine learning", "ai", "cloud", "qu
        mkt_kw   = ["marketing", "seo", "social media", "content", "brand", "growth"]
        ops_kw   = ["logistics", "supply chain", "operations", "ops", "warehouse", "procurement"]

        li_is_legal = any(k in li_text for k in legal_kw)
        li_is_tech  = any(k in li_text for k in tech_kw)
        li_is_ops   = any(k in li_text for k in ops_kw)
        li_is_mkt   = any(k in li_text for k in mkt_kw)

        cv_is_legal = any(k in cv_text for k in legal_kw)
        cv_is_tech  = any(k in cv_text for k in tech_kw)
        cv_is_ops   = any(k in cv_text for k in ops_kw)
        cv_is_mkt   = any(k in cv_text for k in mkt_kw)

        industry_conflict = False

        # Strong conflicts -> MAJOR
        if (li_is_legal and cv_is_tech) or (li_is_tech and cv_is_legal):
```

```
                    issues.append("Strong domain conflict: Legal vs Tech between CV and LinkedIn.")
                    if is_exact:
                        industry_conflict = True
                        major += 1
                    else:
                        minor += 1
                        issues.append("Domain conflict treated as minor because LinkedIn match is not exact.")
                # Weak conflicts -> MINOR (do not reject)
                elif (li_is_ops and cv_is_mkt) or (li_is_mkt and cv_is_ops):
                    issues.append("Weak domain conflict: Marketing vs Logistics/Operations between CV and LinkedIn.")
                    minor += 1

                # ------------------------------
                # Company mismatch logic (conditional)
                # ------------------------------

                if company_mis:
                        if industry_conflict:
                                issues.append(
                                        "Escalation: company mismatch supported by domain conflict."
                                )
                                major += 1
                        else:
                                minor += 1
                                issues.append(
                                        "Treated company mismatch as minor (possible injected current-job inconsistency)."
                                )

        # ==============================
        # Facebook (very weak auxiliary)
        # ==============================

        if fb:
                fb_prof = fb["profile"]
                cv_city = (cv.get("city") or "").lower().strip()
                fb_city = (fb_prof.get("city") or "").lower().strip()

                if cv_city and fb_city and (cv_city not in fb_city) and (fb_city not in cv_city):
                        issues.append(
                                f"(Weak) City mismatch: CV={cv.get('city')} vs FB={fb_prof.get('city')}"
                        )


        # ==============================
        # Internal inconsistency penalty
        # ==============================

        pen, pen_issues = internal_inconsistency_penalty(cv)
        issues.extend(pen_issues)

        # ==============================
        # Final scoring
        # ==============================

        raw = 0.85 - 0.40 * major - 0.08 * minor - pen
        score = max(0.05, min(1.0, raw))

        return float(score), issues
```

```
import asyncio

async def run_all(all_cvs, tools, llm):
        scores = []
        debug = []

        for item in all_cvs:
                cv = extract_cv_json(llm, item["text"])
                cv = normalize_cv(cv, item["text"])
                name = (cv.get("name") or "").strip()

                # LinkedIn location hint：可选，有就用
                loc = None
                if cv.get("city") and cv.get("country"):
                        loc = f"{cv['city']}, {cv['country']}"
                elif cv.get("country"):
                        loc = cv["country"]

                li = await fetch_linkedin_best(tools, name, loc) if name else None
                fb = await fetch_facebook_best(tools, name) if name else None


                score, issues = compute_score(cv, li, fb)
```

```
                scores.append(score)
                debug.append({
                        "file": item["file"],
                        "name": name,
                        "score": score,
                        "issues": issues,
                        "li_match_type": (li.get("candidate") or {}).get("match_type") if li else None,
                        "li_headline": (li.get("profile") or {}).get("headline") if li else None,
                        "li_industry": (li.get("profile") or {}).get("industry") if li else None,
                })

        return scores, debug

scores, debug = await run_all(all_cvs, mcp_tools, llm)
scores
```

```
[0.53, 0.61, 0.61, 0.05, 0.05]
```

Evaluation code

In the test phase, you will be given 5 CV files with fixed names:

```
CV_1.pdf, CV_2.pdf, CV_3.pdf, CV_4.pdf, CV_5.pdf
```

Your system must process these CVs and output a list of 5 scores, one score per CV, in the same order:

```
scores = [s1, s2, s3, s4, s5]
```

Each score must be a float in the range [0, 1], representing the reliability or confidence that the CV is valid (or meets the task criteria).

The ground-truth labels are binary:

```
groundtruth = [0 or 1, ..., 0 or 1]
```

Each CV is evaluated independently using a threshold of 0.5:

- If score > 0.5 and groundtruth == 1 → Full credit
- If score ⩽ 0.5 and groundtruth == 0 → Full credit
- Otherwise → No credit

In other words, 0.5 is the decision threshold.

- Each CV contributes equally.
- Final score = (number of correct decisions) / 5

```
# ==================================================
#   Evaluation code
# ==================================================

def evaluate(scores, groundtruth, threshold=0.5):
        """
        scores: list of floats in [0, 1], length = 5
        groundtruth: list of ints (0 or 1), length = 5
        """
        assert len(scores) == 5
        assert len(groundtruth) == 5

        correct = 0
        decisions = []

        for s, gt in zip(scores, groundtruth):
                pred = 1 if s > threshold else 0
                decisions.append(pred)
                if pred == gt:
                        correct += 1

        final_score = correct / len(scores)

        return {
                "decisions": decisions,
                "correct": correct,
                "total": len(scores),
                "final_score": final_score
        }
```

```
groundtruth = [1, 1, 1, 0, 0] # Do not modify

result = evaluate(scores, groundtruth)
print(result)
```

```
{'decisions': [1, 1, 1, 0, 0], 'correct': 5, 'total': 5, 'final_score': 1.0}
```