Multi Agent Job Email Assistant

Name: Aswin Balaji Thippa Ramesh
Instructor: Dr. Amir Hossein Jafari, Ph.D.

Introduction

Managing job applications through email can quickly become overwhelming, especially when important updates like interview invites, rejections, and follow-ups are mixed with unrelated messages. To make this process easier, we built a Multi-Agent Email Job Application Assistant that automatically reads Gmail messages, identifies job-related emails, and extracts key information such as company name, position, status, and application date. The system combines email classification, entity extraction, vector storage, and a RAG-based chatbot, allowing users to search or chat with their job-application history instead of manually digging through their inbox. A simple Streamlit interface ties everything together, and all processing runs locally for privacy. Overall, the project reduces the stress of managing applications and helps students stay organized throughout their job search.

Description

The dataset for this project is based on the Kaggle dataset *"Job Application Email – Anonymized and Feature Rich,"* which contains real-world job-related email subjects and bodies but no labels. To create a usable supervised dataset, we combined these job-related samples with a small set of personal, non-job emails collected from an inbox, including newsletters, promotions, and academic messages. All Kaggle emails were labeled **job**, while the manually collected emails were labeled **nonjob**, resulting in a balanced dataset with three columns: **subject**, **emailbody**, and **label**. After labeling, the text was cleaned by removing unnecessary characters, standardizing formatting, and normalizing content. The final dataset was then split into training and testing sets, forming the foundation for the email classification component of the multi-agent assistant.

My Contributions

The following points summarize my major technical contributions across dataset creation, email processing, model development, and the Streamlit-based interface for the Multi-Agent Email Job

Application Assistant:

- **Created a synthetic labeled dataset** by assigning job labels to the Kaggle job-email samples and collecting additional non-job emails from my personal inbox.

- **Integrated Gmail API using OAuth credentials** and used BeautifulSoup to cleanly extract subjects, bodies, timestamps, and mail links directly from Gmail.

- **Developed Classifier 2 using sentence-transformer embeddings**, where I generated vector representations for job emails and compared them to embeddings of incoming emails using cosine similarity.

- **Built a dedicated NER agent** using a locally downloaded LLM model to extract entities such as company name, position applied, application status, and date from job-related emails.

- **Enhanced the Streamlit application** by adding features such as:
  - a **flush/reset option** to clear stored data and ChromaDB for a fresh tracking session
  - a **custom email input tab** allowing users to test the classifier and NER on manually typed text

These contributions collectively strengthened the accuracy, usability, and modularity of the system, making it a practical tool for students managing multiple job applications.

Synthetic Labeled Dataset for Job Classification

Code files: Data/jobs.csv

- I constructed a **synthetic labeled dataset** for the job vs. non-job classification task.
- All job-related examples were sourced from the Kaggle dataset *"Job Application Email – Anonymized and Feature Rich"* and manually labeled as job.
- I then collected a small set of my own emails (newsletters, promos, university announcements, etc.) and manually labeled them as nonjob.
- The final training data was stored with three columns: subject, emailbody, and label, and was later used to train and evaluate different classifiers in the project.

Live Gmail Fetcher with OAuth + Parsing

Code file: Code/gmail_read.py

- I implemented the **GmailLiveReader** class that handles end-to-end Gmail integration using the Gmail API, OAuth tokens, and a locally stored credentials.json file.
- The module authenticates the user, queries Gmail using a date-based filter (e.g., QUERY = "after:2025/10/28"), and iteratively fetches all matching message IDs.
- For each message, I decode the raw email, parse headers and body, and clean HTML content using **BeautifulSoup** to obtain a readable subject, body, and date_received.
- I also added logic to build a direct gmail_link URL and to **append only new emails** to gmail_subject_body_date.xlsx, maintaining a growing dataset of inbox messages.
- The clean_for_excel() utility ensures all text is safe to write into Excel (removing illegal XML characters and long strings).

```
class GmailLiveReader:
    def __init__(
        self,
        credentials_path="credentials.json",
        token_path="token.json",
        gmail_account_index=0,
    ):
        self.credentials_path = os.path.abspath(credentials_path)
        self.token_path = os.path.abspath(token_path)
        self.gmail_account_index = gmail_account_index
        self.gmail_web_base =
f"https://mail.google.com/mail/u/{gmail_account_index}/#all/"
        self.scopes = DEFAULT_SCOPES
        self.service = self._authenticate()

    # ---------------- AUTH ----------------
    def _authenticate(self):
        creds = None
        if os.path.exists(self.token_path):
            creds = Credentials.from_authorized_user_file(self.token_path,
self.scopes)

        if not creds or not creds.valid:
            if creds and creds.expired and creds.refresh_token:
                creds.refresh(Request())
            else:
                flow = InstalledAppFlow.from_client_secrets_file(
                    self.credentials_path, self.scopes
                )
                creds = flow.run_local_server(port=0)

            with open(self.token_path, "w") as f:
                f.write(creds.to_json())

        return build("gmail", "v1", credentials=creds)

    # ------------- UTILITIES -------------
    @staticmethod
    def html_to_text(html):
        if not html:
            return ""
        soup = BeautifulSoup(html, "html.parser")
        for tag in soup(["script", "style"]):
            tag.decompose()
        return " ".join(soup.get_text(separator=" ", strip=True).split())
```

Embedding-Based Job / Non-Job Classifier (Classifier 2)

Code file: Code/Classifier 2/job_classifier.py

- I designed **Classifier 2**, a lightweight classifier that uses **SentenceTransformer embeddings** instead of a traditional ML model with explicit training labels.
- First, I generate embeddings for the Kaggle-based jobs.csv examples using all-mpnet-base-v2.
- For each Gmail email in gmail_subject_body_date.xlsx, I combine subject + body into full_text, embed it, and compute **cosine similarity** against all job examples.
- I then apply a combination of rules:
    - Regex patterns (JOB_PROCESS_PATTERNS) → force label job.
    - Job-alert patterns (JOB_ALERT_PATTERNS) → force non_job.
    - If the maximum similarity is ≥ **0.82** *and* the text contains core job phrases, the email is labeled job; otherwise, non_job.
- The final labels are written to mail_classified2.xlsx with a new column job_label, and this file feeds downstream modules.

```
def combine(subject, body) -> str:
    s = safe_str(subject)
    b = safe_str(body)
    return normalize(s + "\n" + b)

df_jobs["full_text"] = df_jobs.apply(
    lambda r: combine(r[JOBS_SUBJECT_COL], r[JOBS_BODY_COL]),
    axis=1,
)
df_gmail["full_text"] = df_gmail.apply(
    lambda r: combine(r[GMAIL_SUBJECT_COL], r[GMAIL_BODY_COL]),
    axis=1,
)

model = SentenceTransformer(MODEL_NAME)

job_emb = model.encode(
    df_jobs["full_text"].tolist(),
    convert_to_numpy=True,
    normalize_embeddings=True,
)
gmail_emb = model.encode(
    df_gmail["full_text"].tolist(),
    convert_to_numpy=True,
    normalize_embeddings=True,
)

sim_matrix = cosine_similarity(gmail_emb, job_emb)
max_sims = sim_matrix.max(axis=1)
```

Local LLM-Based NER Agent

Code file: Code/ner.py

- I built a **NER agent** that uses a **locally hosted LLaMA 3.1 model via Ollama** to extract structured job information from emails.
- The script ner.py loads classified emails (e.g., mail_classified.xlsx), filters rows where job_label == "job", and processes only those.
- For each email, I:
    - Generate a concise summary (summarize_email) for stable extraction.
    - Call the LLM with a strict **JSON-only system prompt** to extract: company_name, position_applied, application_date, and status.
    - Apply heuristic cleanup functions (heuristic_position, clean_final_position, infer_status) so that the final status is one of: applied, in progress, rejected, or job offered.
    - Normalize application_date from the date_received field where possible.
- Parsed results are stored in mail_classified_llm_parsed.xlsx with columns: mailcontent, company_name, position_applied, application_date, status, mail_link.

Output of NER:

mailcontent	company_name	position_applied	application_date	status
"Thank you for applying to Microsoft. We received your application and our team will review your profile soon."	Microsoft	Data Scientist Intern	2025-11-02	applied

mailcontent	company_name	position_applied	application_date	status
"We are pleased to invite you for a first round interview. Please select a time slot for the virtual interview."	Amazon	ML Engineer Intern	2025-11-05	in progress
"After reviewing your application, we will not be moving forward with your candidacy at this time."	Adobe	Research Intern	2025-11-01	rejected
"Your application has moved to the hiring manager review stage. We will contact you with next steps soon."	IBM	AI Engineer Intern	2025-11-03	in progress
"Congratulations! We are excited to extend you an offer for the position. More details are attached."	Meta	Applied Scientist Intern	2025-11-07	job offered

LLM Call and JSON Extraction

```python
def call_llm_extract(subject: str, mailcontent: str, date_received: str) -> dict:
    text_for_extraction = subject + "\n" + (mailcontent or "")

    user_prompt = f"""
{SYSTEM_PROMPT}

Here is the email content you must extract from:

\"\"\"{text_for_extraction}\"\"\"

Email received date:
\"\"\"{date_received or ""}\"\"\"

Return ONLY a JSON object with keys:
company_name, position_applied, application_date, status.
"""

    raw_text = call_ollama(user_prompt)
    if not raw_text:
        return {
            "company_name": "",
            "position_applied": "",
            "application_date": "",
            "status": "applied",
        }

    text = extract_json_object(raw_text)

    for attempt in range(2):
        try:
            data = json.loads(text)
            break
        except json.JSONDecodeError:
            if attempt == 0:
                text = re.sub(r",\s*([}\]])", r"\1", text)
            else:
                data = {
                    "company_name": "",
                    "position_applied": "",
                    "application_date": "",
                    "status": "applied",
                }

    return {
        "company_name": str(data.get("company_name", "")).strip(),
        "position_applied": str(data.get("position_applied", "")).strip(),
        "application_date": str(data.get("application_date", "")).strip(),
        "status": str(data.get("status", "")).strip().lower(),
    }
```

Heuristic + Rule-Based Status Refinement

```
def infer_status(full_text: str, llm_status: str) -> str:
    text = full_text.lower()

    if matches_any(offer_patterns, text):
        return "job offered"
    if matches_any(reject_patterns, text):
        return "rejected"
    if matches_any(in_progress_patterns, text):
        return "in progress"
    if matches_any(applied_patterns, text):
        return "applied"

    if llm_status in {"applied", "in progress", "rejected", "job offered"}:
        return llm_status
    return "applied"
```

Streamlit UI for Custom Testing Features

Code file: Code/app.py

- I implemented the **Streamlit front-end** that orchestrates all core modules: gmail_read.py, predict.py, ner.py, and rag.py.
- The app provides four main tabs:
    1. **Fetch + Classify** – lets users upload their own credentials.json, choose a date range, fetch new Gmail emails, and run the classifier.
    2. **NER Parsed Jobs** – runs ner.main() and displays parsed job records (company, role, date, status, mail link).
    3. **RAG Assistant** – connects to rag.ask(question) so users can ask natural-language questions about their applications.
    4. **Custom Email Test** – a feature I added that allows users to type any subject/body and run **only classification** or **classification + NER** without using Gmail.
- I also added:
    - A **global flush/reset** function that deletes gmail_subject_body_date.xlsx, mail_classified.xlsx, mail_classified_llm_parsed.xlsx, and the entire chroma_store/ folder, giving users a clean state.
    - Dynamic date-based Gmail queries (after: / before:) directly from the UI.
    - Visual summaries (tables and bar charts) of job vs non-job counts

Flush Data

```python
def _flush_all() -> tuple[int, bool]:
    files = [
        os.path.join(DATA_DIR, "gmail_subject_body_date.xlsx"),
        os.path.join(DATA_DIR, "mail_classified.xlsx"),
        os.path.join(DATA_DIR, "mail_classified_llm_parsed.xlsx"),
    ]

    deleted_count = 0
    for f in files:
        if os.path.exists(f):
            os.remove(f)
            deleted_count += 1

    chroma_deleted = False
    if os.path.exists(CHROMA_DIR):
        shutil.rmtree(CHROMA_DIR)
        chroma_deleted = True
        os.makedirs(CHROMA_DIR, exist_ok=True)

    return deleted_count, chroma_deleted
```

Custom Email Classify and NER

```python
def _run_custom_email_through_pipeline(subject: str, body: str, run_ner: bool = True):
    gmail_file = OUTPUT_EXCEL
    classified_path = os.path.join(DATA_DIR, "mail_classified.xlsx")
    parsed_path = os.path.join(DATA_DIR, "mail_classified_llm_parsed.xlsx")

    timestamp_str = datetime.now().strftime("%Y%m%d%H%M%S%f")
    custom_id = f"custom_{timestamp_str}"
    custom_link = f"custom://{timestamp_str}"

    custom_df = pd.DataFrame([{
        "id": custom_id,
        "sender_name": "Custom User",
        "sender_email": "custom@example.com",
        "subject": subject or "(no subject)",
        "body": body or "",
        "date_received": datetime.now(),
        "gmail_link": custom_link,
    }])

    custom_df.to_excel(gmail_file, index=False)

    # run classifier
    predict.main()

    # force job_label = 'job' if NER is requested
    if run_ner and os.path.exists(classified_path):
        cdf = pd.read_excel(classified_path)
        cdf.loc[cdf["id"] == custom_id, "job_label"] = "job"
        cdf.to_excel(classified_path, index=False)
        ner.main()
```

Conclusion :

In my view, this project was a practical and meaningful way to apply everything I learned about NLP. Managing job applications is something almost every student struggles with, and building a system that can automatically fetch emails, classify them, extract useful details, and store everything in a searchable format felt genuinely useful. Working with tools like the Gmail API, embedding-based classification, local LLMs for NER, and a RAG assistant helped me understand how real-world applications combine multiple NLP techniques to solve everyday problems. Overall, this project not only improved my technical skills but also showed me how NLP can make the job-search process more organized, efficient, and less stressful.

References :

1) HuggingFace. *SentenceTransformers – Semantic Search & Cosine Similarity Guide*. https://www.sbert.net/examples/applications/semantic-search/README.html

2) Google Developers. *Gmail API Overview*. https://developers.google.com/gmail/api/guides