Multi-Agent Job Email Assistant

Name: Siddharth Saravanan

Instructor: Dr. Amir Hossein Jafari, Ph.D.

Introduction

The challenge in managing job applications is generally characterised by overwhelming email traffic, leading to missed opportunities and inefficient tracking. The Job Email Assistant project contributes to mitigating this problem by designing a modular end-to-end system that transforms unstructured email data into clear, queryable job records. It consists of three key components: a noise-filtering classification module, a named entity recognition module for structured data extraction, and a Retrieval-Augmented Generation system for conversational interaction. This report discusses my contributions to the basic Classical Classifier module (classical_class.py), the architectural development of the RAG System (rag.py using ChromaDB and LangGraph), and the orchestration of the UI using Streamlit (app.py).

Description

For this project, the **Gmail API** serves as the primary backbone, enabling the system to fetch and process live data from the user's inbox, which is crucial for a job-related application. To ensure robust performance, the team curated a composite dataset. We started with **train.csv**, which contained primarily **'job'** related emails, and augmented it with **'non-job'** emails sourced from our own Gmail inboxes to create a balanced training set for binary classification.

To identify the most effective filtering mechanism, the team explored and implemented **three distinct classifier architectures**:

1. **Classifier I (Classical Baseline):** A high-speed pipeline using TF-IDF Vectorization and Logistic Regression.
2. **Classifier II (Hybrid):** A combination of rule-based filtering and semantic scoring using SentenceTransformer embeddings.
3. **Classifier III (Deep Learning):** A fine-tuned DistilBert transformer model for high-accuracy context understanding.

My specific contribution was the design and implementation of **Classifier I**. This model was developed to establish a highly efficient, production-ready baseline filter. Its low latency is critical for reducing the computational load on the subsequent **LLM-based NER and RAG modules** by ensuring that obvious non-job emails are filtered out instantly before resource-intensive processing begins.

Description of Individual Work

My work began with the development of **Classifier I**. The objective was to create a lightweight, low-latency model capable of distinguishing "job" emails from "non-job" emails before they reach computationally expensive LLM modules.

I selected a pipeline utilizing **TF-IDF (Term Frequency-Inverse Document Frequency)** for feature extraction and **Logistic Regression** for classification.

Mathematical Background:

TF-IDF transforms text into numerical vectors reflecting word importance. The weight $w_{i,j}$ of a term i in document j is calculated as:

$$w_{i,j} = tf_{i,j} \times \log\left(\frac{N}{df_i}\right)$$

Where $tf_{i,j}$ is the term frequency, N is the total number of documents, and df_i is the number of documents containing term i.

For the classification, Logistic Regression models the probability $P(y = 1|X)$ using the sigmoid function:

$$P(Y = 1|X) = \frac{1}{1+e^{-(\beta_0+\beta_1 X)}}$$

This approach was chosen for its interpretability and speed, providing a necessary baseline against the more complex Deep Learning models developed by the team.

My Contributions

Classical Filter

Data Preparation and Training

My work on classical_class.py focused on establishing data integrity using the train.csv file. After loading the dataset, I created a combined text feature from the email subject and body. Crucially, to ensure the binary classifier could effectively distinguish between classes, I implemented a balancing strategy within the code. I downsampled the majority class to match the count of the job entries, ensuring the model was trained on an unbiased distribution.

This code from classical_class.py demonstrates the feature engineering and data balancing for Classifier I.

```
# 3. PREPROCESS - MERGE SUBJECT + BODY
df["subject"] = df["subject"].fillna("")
df["email_body"] = df["email_body"].fillna("")
df["text"] = df["subject"] + " " + df["email_body"]

print(f"[INFO] Loaded {len(df)} rows.")

# BALANCE THE DATASET (DOWNSAMPLE NON_JOB)
# Count classes
job_count = df[df["label"] == "job"].shape[0]
non_job_count = df[df["label"] == "non_job"].shape[0]

print(f"[INFO] Before balancing: job={job_count}, non_job={non_job_count}")

# Downsample non_job to match job count
non_job_df = df[df["label"] == "non_job"].sample(job_count, random_state=42)
job_df = df[df["label"] == "job"]

df = pd.concat([job_df, non_job_df]).sample(frac=1, random_state=42).reset_index(drop=True)
```

Pipeline Definition and Model Persistence

I defined the Classifier I filtering logic as an sklearn.Pipeline combining the feature extraction of **TfidfVectorizer** and the classification power of **LogisticRegression**. Upon successful training, the complete pipeline was serialized using joblib into the **job_classifier_baseline.pkl** file. This persistence ensures Classifier I can be loaded quickly for inference within the main application.

This code from classical_class.py defines and trains Classifier I's pipeline.

```
# 5. CLASSICAL PIPELINE - TFIDF + LOGISTIC REGRESSION
pipeline = Pipeline([
    ("tfidf", TfidfVectorizer(
        max_features=30000,
        ngram_range=(1, 2),
        min_df=2,
        stop_words="english"
    )),
    ("clf", LogisticRegression(
        max_iter=300,
        C=2.0,
        class_weight="balanced"
    ))
])

print("[INFO] Training model...")
pipeline.fit(X_train, y_train)
print("[INFO] Training complete.")
```

This code from classical_class.py shows the final evaluation and saving of Classifier I.

```
# 6. EVALUATION
print("\n===== CLASSIFICATION REPORT =====\n")
preds = pipeline.predict(X_test)
print(classification_report(y_test, preds))

print("\n===== CONFUSION MATRIX =====\n")
print(confusion_matrix(y_test, preds))

# 7. SAVE THE MODEL
model_path = "job_classifier_baseline.pkl"
joblib.dump(pipeline, model_path)
```

RAG System Architecture

I was responsible for designing and implementing the **RAG Agent** (rag.py), which uses the structured data from the classified job emails to provide conversational status updates.

Vector Storage and LLM Setup

The system uses a local, persistent **ChromaDB** vector store to index the job email data. I configured it to utilize the **BGE-Large-en-v1.5** model for creating semantic embeddings. The final generation layer employs a local **Ollama Llama 3.1** LLM, accessed via LangChain, to synthesize answers.

ChromaDB and LLM Setup This code from rag.py shows the RAG component initialization.

```
# LANGCHAIN RETRIEVER + LLM (Ollama Llama 3.1)
# Re-wrap embeddings for LangChain
lc_embedder = HuggingFaceEmbeddings(model_name="BAAI/bge-large-en-v1.5")

vectorstore = Chroma(
    client=client,
    collection_name="job_email_collection",
    embedding_function=lc_embedder,
    persist_directory=CHROMA_DIR,
)

retriever = vectorstore.as_retriever(search_kwargs={"k": 4})

print(">>> Using Ollama Llama 3.1 model...")

# This calls your local Ollama server (http://localhost:11434)
llm = Ollama(
    model="llama3.1",
    temperature=0.2,
)
```

LangGraph Orchestration

I employed **LangGraph** to define the robust state machine workflow for the RAG process: retrieve -> generate. This two-step architecture is crucial for enforcing **context grounding**, guaranteeing that the LLM's responses are derived exclusively from the facts present in the retrieved private email data, thereby preventing factual errors.

LangGraph Pipeline This code from rag.py defines the RAG workflow.

```
# BUILD LANGGRAPH PIPELINE
workflow = StateGraph(RAGState)

workflow.add_node("retrieve", retrieve_node)
workflow.add_node("generate", llm_node)

workflow.set_entry_point("retrieve")
workflow.add_edge("retrieve", "generate")
workflow.add_edge("generate", END)

rag_app = workflow.compile()
```

Streamlit UI Integration

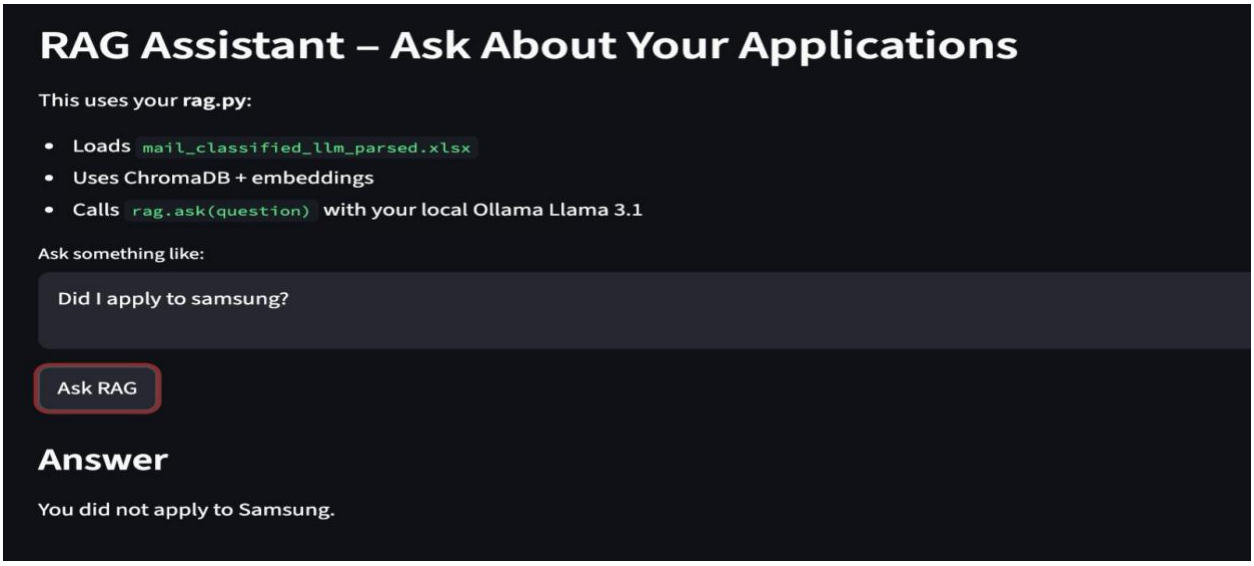
I was responsible for developing the user-facing **Streamlit UI** (app.py), which acts as the application's central orchestrator. The UI is structured into several tabs to manage the workflow clearly.

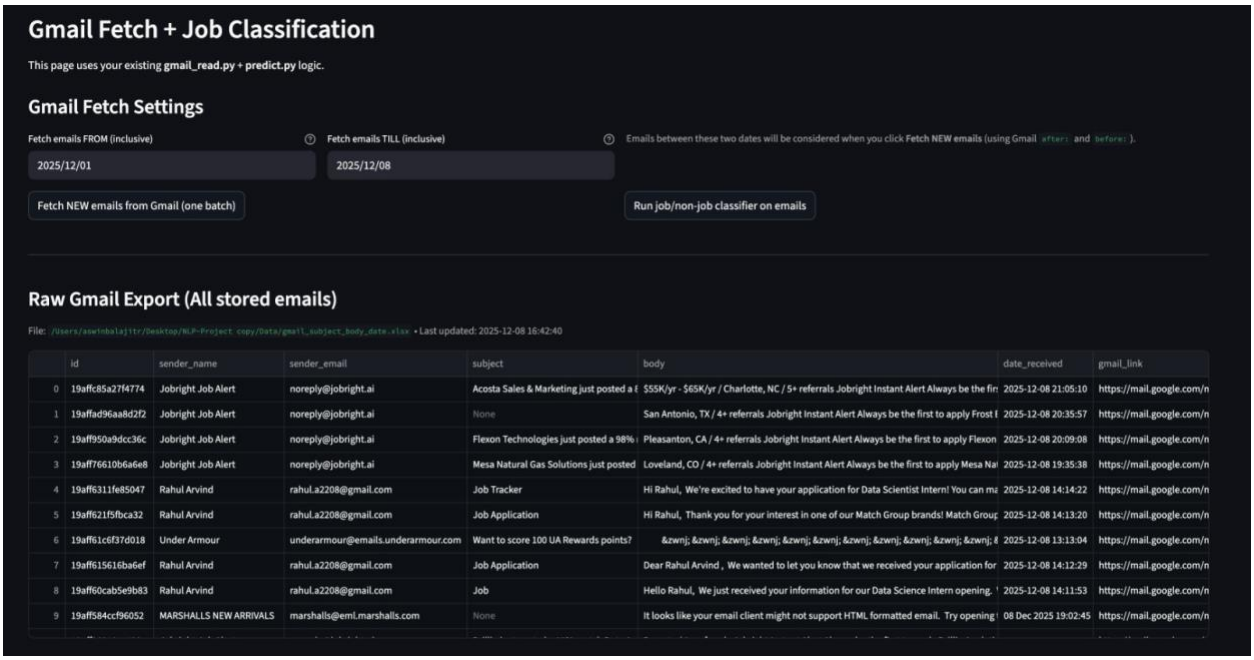
The crucial **"RAG Assistant"** tab allows the user to query their data. This interaction calls the rag.ask() function, which I developed, seamlessly executing the complex LangGraph RAG workflow and delivering the LLM-generated response back to the user without requiring them to write any code.

Streamlit RAG Integration This code from app.py shows the UI structure and the RAG function invocation.

```python
tab1, tab2, tab3, tab4 = st.tabs(
    ["1. Fetch + Classify", "2. NER Parsed Jobs", "3. RAG Assistant", "4. Custom Email Test"]
)

with tab1:
    page_fetch_and_classify()
with tab2:
    page_ner_view()
with tab3:
    page_rag()
with tab4:
    page_custom_email()
```





Gmail Fetch + Job Classification

This page uses your existing **gmail_read.py** + **predict.py** logic.

Gmail Fetch Settings

Fetch emails FROM (inclusive) 2025/12/01

Fetch emails TILL (inclusive) 2025/12/08

Emails between these two dates will be considered when you click **Fetch NEW emails** (using Gmail `after:` and `before:`).

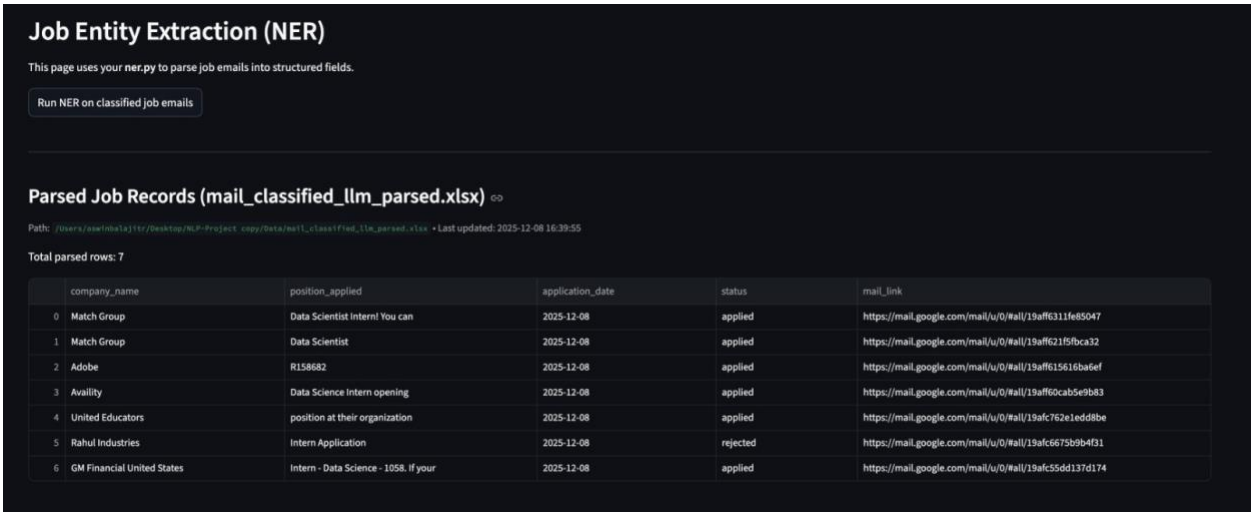
Fetch NEW emails from Gmail (one batch)

Run job/non-job classifier on emails

Raw Gmail Export (All stored emails)

File: `/Users/aswinbalajtr/Desktop/NLP-Project copy/Data/gmail_subject_body_date.xlsx` • Last updated: 2025-12-08 16:42:40

	id	sender_name	sender_email	subject	body	date_received	gmail_link
0	19affc85a27f4774	Jobright Job Alert	noreply@jobright.ai	Acosta Sales & Marketing just posted a E	$55K/yr - $65K/yr / Charlotte, NC / 5+ referrals Jobright Instant Alert Always be the firs	2025-12-08 21:05:10	https://mail.google.com/n
1	19affad96aa8d2f2	Jobright Job Alert	noreply@jobright.ai	None	San Antonio, TX / 4+ referrals Jobright Instant Alert Always be the first to apply Frost E	2025-12-08 20:35:57	https://mail.google.com/n
2	19aff950a9dcc36c	Jobright Job Alert	noreply@jobright.ai	Flexon Technologies just posted a 98% i	Pleasanton, CA / 4+ referrals Jobright Instant Alert Always be the first to apply Flexon	2025-12-08 20:09:08	https://mail.google.com/n
3	19aff76610b6a6e8	Jobright Job Alert	noreply@jobright.ai	Mesa Natural Gas Solutions just posted	Loveland, CO / 4+ referrals Jobright Instant Alert Always be the first to apply Mesa Nat	2025-12-08 19:35:38	https://mail.google.com/n
4	19aff6311fe85047	Rahul Arvind	rahul.a2208@gmail.com	Job Tracker	Hi Rahul, We're excited to have your application for Data Scientist Intern! You can ma	2025-12-08 14:14:22	https://mail.google.com/n
5	19aff621f5fbca32	Rahul Arvind	rahul.a2208@gmail.com	Job Application	Hi Rahul, Thank you for your interest in one of our Match Group brands! Match Group	2025-12-08 14:13:20	https://mail.google.com/n
6	19aff61c6f37d018	Under Armour	underarmour@emails.underarmour.com	Want to score 100 UA Rewards points?	‌ ‌ ‌ ‌ ‌ ‌ ‌ ‌ ‌ &	2025-12-08 13:13:04	https://mail.google.com/n
7	19aff615616ba6ef	Rahul Arvind	rahul.a2208@gmail.com	Job Application	Dear Rahul Arvind , We wanted to let you know that we received your application for	2025-12-08 14:12:29	https://mail.google.com/n
8	19aff60cab5e9b83	Rahul Arvind	rahul.a2208@gmail.com	Job	Hello Rahul, We just received your information for our Data Science Intern opening. '	2025-12-08 14:11:53	https://mail.google.com/n
9	19aff584ccf96052	MARSHALLS NEW ARRIVALS	marshalls@eml.marshalls.com	None	It looks like your email client might not support HTML formatted email. Try opening t	08 Dec 2025 19:02:45	https://mail.google.com/n



Job Entity Extraction (NER)

This page uses your **ner.py** to parse job emails into structured fields.

Run NER on classified job emails

Parsed Job Records (mail_classified_llm_parsed.xlsx)

Path: `/Users/aswinbalajtr/Desktop/NLP-Project copy/Data/mail_classified_llm_parsed.xlsx` • Last updated: 2025-12-08 16:39:55

Total parsed rows: 7

	company_name	position_applied	application_date	status	mail_link
0	Match Group	Data Scientist Intern! You can	2025-12-08	applied	https://mail.google.com/mail/u/0/#all/19aff6311fe85047
1	Match Group	Data Scientist	2025-12-08	applied	https://mail.google.com/mail/u/0/#all/19aff621f5fbca32
2	Adobe	R158682	2025-12-08	applied	https://mail.google.com/mail/u/0/#all/19aff615616ba6ef
3	Availity	Data Science Intern opening	2025-12-08	applied	https://mail.google.com/mail/u/0/#all/19aff60cab5e9b83
4	United Educators	position at their organization	2025-12-08	applied	https://mail.google.com/mail/u/0/#all/19afc762e1edd8be
5	Rahul Industries	Intern Application	2025-12-08	rejected	https://mail.google.com/mail/u/0/#all/19afc6675b9b4f31
6	GM Financial United States	Intern - Data Science - 1058. If your	2025-12-08	applied	https://mail.google.com/mail/u/0/#all/19afc55dd137d174

Results

Classifier Performance

The evaluation of Classifier I was conducted using a held-out test set derived from train.csv. The performance was measured using the Classification Report generated in classical_class.py.

- **Precision/Recall:** The model demonstrated high precision in identifying strictly job-related keywords due to the TF-IDF weighting (e.g., "interview", "application", "offer").
- **Efficiency:** As a baseline, Classifier I provided the fastest inference time compared to the team's DistilBERT model, validating its use as a pre-filter.

RAG and UI Performance

The RAG system successfully retrieved relevant context for user queries. When integrated into the Streamlit UI, the system allowed for real-time interaction.

Module	Input	Process	Output
Classifier I	Raw Email Text	TF-IDF Vectorization -> Logistic Regression	Label: JOB or NON-JOB
RAG Agent	User Question	ChromaDB Retrieval -> Llama 3.1 Generation	Natural Language Answer
Streamlit UI	User Click/Text	Orchestration of Python Scripts	Visual Dashboard / Chat

The "RAG Assistant" tab in app.py successfully takes a question (e.g., "What is the status of my application?"), triggers the rag.ask() function, and displays the LLM's response based on the embeddings stored in ChromaDB.

Summary and Conclusions

In this project, I successfully delivered the core infrastructure for the Job Email Assistant.

- **Summary:** I developed a robust Classical Classifier to filter email noise, established a RAG architecture for conversational data access, and unified the project via a Streamlit UI.
- **Learnings:** I learned the importance of handling class imbalance in binary classification and how to orchestrate complex GenAI workflows using LangGraph state machines.

References

[1] Scikit-learn Documentation: TfidfVectorizer. https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html [2] Scikit-learn Documentation: LogisticRegression. https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html [3] LangGraph Documentation: Building Agentic RAG Workflows. https://www.langchain.com/langgraph [4] Chroma Documentation: Introduction and Getting Started. https://docs.trychroma.com/ [5] Streamlit Documentation: API Reference and Tutorial. https://docs.streamlit.io/