THE GEORGE WASHINGTON UNIVERSITY

WASHINGTON, DC

**MS in Data Science**

DATS 6312 - Natural Language Processing
Fall 2025 Final Report

# Multi Agent Job Email Assistant

**Presented by,**
Aswin Balaji Thippa Ramesh
Rahul Arvind
Siddharth Saravanan

**Supervised by**
Dr. Amir Hossein Jafari, Ph.D.,

## 1. Introduction

Managing job applications through email can get overwhelming, especially when updates, rejections, interview invites, and instructions are mixed with dozens of unrelated messages. As students actively applying to internships and full-time roles, keeping track of all these emails manually becomes difficult and easy to forget. This project aims to solve that problem by building a **Multi-Agent Email Job Application Assistant** that automatically reads Gmail messages, identifies which ones are related to job applications, and extracts useful details like company name, position applied for, status, and application date. The idea is to reduce the stress of sorting through inboxes and give students a simple way to stay organized. By automating the boring steps, the system lets users focus more on preparing for applications and interviews.

The system brings together several NLP components like email classification, entity extraction, vector storage, and a RAG-based chatbot, so users can easily search and talk to their job-application history. Instead of scrolling through inboxes, students can simply ask questions like “Which companies rejected me?” or “What all did I apply for last month?” A simple Streamlit app connects everything, making the whole process fast, organized, and user-friendly. It also keeps all processing local for privacy, so no email content is shared externally. Overall, the project shows how modern NLP tools can genuinely make the job-search process easier and more manageable for students.

## 2. Dataset Description

The dataset used for this project is based on the Kaggle dataset *“Job Application Email – Anonymized and Feature Rich”*,which provides a collection of real-world job-related email samples. The dataset contains anonymized subject lines and email bodies taken from various stages of job applications, such as acknowledgements, interview invitations, follow-up messages, and rejection letters. While the dataset offers rich textual examples and reflects genuine language patterns used by companies, it does **not** include any predefined labels for classification tasks (e.g., job vs non-job).

To transform the dataset into a supervised learning resource, additional labeled examples were required. For this, we incorporated a small set of personal, non–job-related emails collected from an inbox. These included newsletters, advertisements, academic notifications, and general informational emails. All personal emails were manually assigned the label **“nonjob”**, while the Kaggle-derived samples were labeled as **“job”** based on their content and structure. This combination allowed us to build a practical, balanced dataset that captures both job-related communication patterns and typical non-job email noise encountered in everyday inboxes.

The final dataset used for training and evaluation includes **three essential columns**:

- **subject:** the subject line of the email
- **emailbody:** the main email content, cleaned and standardized
- **label:** a binary label indicating either *job* or *nonjob*

This labeled dataset serves as the foundation for the email classification component of our multi-agent system.

## 3. Data Preparation

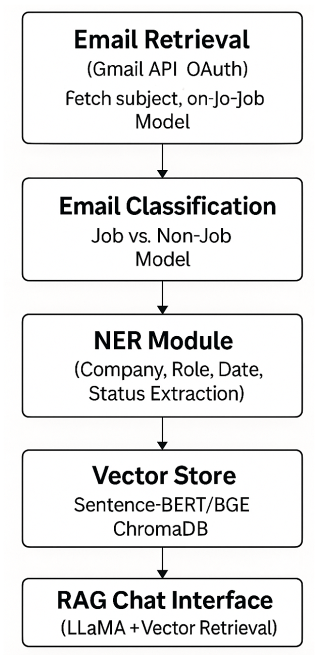
Since the original Kaggle dataset lacked labels and contained only job-related emails, the first step in data preparation involved creating a **synthetic labeled dataset**. All Kaggle email entries were assigned the label **"job"** after verifying their structure and ensuring they represented job-application contexts. To support binary classification, we supplemented the dataset with a limited set of manually curated **non-job emails** from a personal inbox. These emails were reviewed individually and labeled as **"nonjob"** to maintain dataset quality and reduce misclassification.

Next, we applied several preprocessing steps to make the text suitable for machine learning models. Both the subject and email body were cleaned by removing unnecessary whitespace, newline characters, HTML artifacts, and special symbols. Basic normalization such as lowercasing and minimal punctuation handling was applied while still preserving meaningful phrases that may help classifiers distinguish job-related communication. Redundant signatures, disclaimers, and repetitive boilerplate text were also trimmed to ensure the dataset captured the core message content.

After cleaning, the subject and body fields were merged into a unified structure for vectorization and modeling. The prepared dataset was then divided into training and testing splits, ensuring a representative mix of job and non-job samples in both sets. This finalized dataset enabled the development of a reliable classifier capable of identifying job-related emails within a larger multi-agent pipeline.

## 4. Project Workflow

The overall workflow of the Multi-Agent Email Job Application Assistant follows a structured sequence of steps that convert raw Gmail data into meaningful, searchable job-application insights. Each step in the process is handled by a dedicated component, allowing the system to operate efficiently and maintain modularity.



**Email Retrieval (Gmail API) :**

- The system first connects to the user's Gmail account through OAuth authentication.
- It fetches the subject, body, date, and URL link for each email, storing them locally for processing.

**Job vs. Non-Job Classification:**

- Each email is passed through a trained classifier built using the synthetic dataset.
- The classifier identifies whether an email is job-related or not, allowing the system to filter out irrelevant messages.

**Named Entity Extraction :**

- Emails labeled as "job" are further processed by a NER agent.
- This step extracts structured fields such as company name, role applied for, application date, and application status

**Embedding + Vector Storage (ChromaDB) :**

- The enriched job-email records are converted into dense embeddings using a sentence-transformer model.
- These embeddings are stored in a ChromaDB vector database, enabling fast and meaning-aware search.

**RAG-Based Query System:**

- A Retrieval-Augmented Generation pipeline is used to answer user queries.
- The system retrieves relevant emails semantically and uses a local LLaMA model to generate natural, conversational responses.

**Streamlit User Interface** : All steps - fetching, classification, extraction, and querying are integrated into a simple and interactive Streamlit dashboard.

This workflow transforms unstructured inbox data into a searchable, conversational job-tracking assistant that is fast, private, and highly practical for students managing multiple applications.

## 5. Description of the NLP Model and Algorithmic Framework

This project employs a multi-stage Natural Language Processing (NLP) pipeline designed to classify job-related emails and extract structured information from them. The system integrates classical machine-learning models, modern transformer architectures, rule-based heuristics, semantic similarity engines, and a Retrieval-Augmented Generation (RAG) assistant. Together these components form a robust, production-aligned pipeline for automated job-application intelligence.

The primary models used across the system are implemented in the following files:

- **`classifier_1_training.py`** / **`classifier_1_prediction.py`**
  TF-IDF + Logistic Regression baseline classifier.
- **`classifier_2_job_prediction.py`**
  Sentence-Transformer semantic similarity classifier with rule-based overrides.
- **`classifier_3_train.py`** / **`classifier_3_predict.py`**
  Fine-tuned DistilBERT transformer model.
- **`ner.py`**
  Local LLM–driven named entity extraction and status inference.
- **`rag.py`**
  Retrieval-Augmented Generation system using ChromaDB embeddings, Sentence-Transformers, and Llama 3.1.
- **`gmail_read.py`**, **`predict.py`**, **`app.py`**
  Data ingestion, orchestration, and Streamlit control interface.

Each file plays a defined role in the overall architecture. The remainder of this section details the underlying NLP methods and algorithms applied across these components.

### 5.1 Classical NLP Baseline: TF-IDF + Logistic Regression (Classifier 1)

The first model uses a traditional machine-learning approach implemented in **`classifier_1_training.py`**.

### TF-IDF Representation

Text sequences are vectorized using the **Term Frequency–Inverse Document Frequency (TF-IDF)** representation:

$$\text{TF-IDF}(t,d) = \text{TF}(t,d) \times \log\left(\frac{N}{\text{DF}(t)}\right)$$

Where,

- $\text{TF}(t,d)$ = frequency of term $t$ in document $d$,
- $\text{DF}(t)$ = number of documents containing term $t$,
- $N$ = total number of documents.

This yields a sparse, high-dimensional feature vector suitable for linear classification.

### Logistic Regression Classifier

The TF-IDF features feed into a logistic regression model:

$$p(y = 1 \mid x) = \sigma(w^\top x + b)$$
$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

The training objective minimizes the binary cross-entropy:

$$\mathcal{L} = -\sum_{i=1}^{n} [y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)]$$

This model provides a computationally lightweight baseline and is executed during inference via **`classifier_1_prediction.py`**and **`predict.py`**.

## 5.2 Semantic Similarity Classifier (Classifier 2)

A second classifier—implemented in **classifier_2_job_prediction.py**—uses **Sentence-Transformers** to embed emails into dense semantic vectors. It addresses scenarios where lexical overlap is low but semantic meaning is clearly job-related.

**Embedding Model**

The model uses **all-mpnet-base-v2**, producing embeddings:

$$e = f_{\text{MPNet}}(x)$$

**Cosine Similarity Decision Rule**

Similarity to known job-application examples is computed using cosine similarity:

$$\text{sim}(e_1, e_2) = \frac{e_1 \cdot e_2}{\parallel e_1 \parallel \ \parallel e_2 \parallel}$$

A strict threshold:

$$\text{sim}(e_{\text{gmail}}, e_{\text{job}}) \geq \tau, \tau = 0.82$$

is required to classify ambiguous messages as job-related.

**Pattern-Based Overrides**

Classifier 2 incorporates rule-based heuristics:

- **Job-Process Patterns** → forcibly labeled *job*
- **Job-Alert Patterns** → forcibly labeled *non-job*

These rules compensate for shortcomings of embedding similarity alone.

## 5.3 Transformer Model: Fine-Tuned DistilBERT (Classifier 3)

The main NLP model—implemented in **`classifier_3_train.py`**—uses **DistilBERT**, a compact transformer architecture derived from BERT through knowledge distillation.

### Transformer Self-Attention

DistilBERT computes contextual embeddings using the self-attention mechanism:

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^\top}{\sqrt{d_k}})V$$

where

- $Q = XW^Q$,
- $K = XW^K$,
- $V = XW^V$,
- $d_k$ = dimensionality of the key vectors.

### Sequence Classification Layer

The [CLS] embedding $h_{\text{CLS}}$ is passed into a linear classifier:

$$\hat{y} = \text{softmax}(Wh_{\text{CLS}} + b)$$

Training minimizes the categorical cross-entropy:

$$\mathcal{L} = -\sum_{i=1}^{n} \sum_{c\in\{0,1\}} y_{ic} \log(\hat{y}_{ic})$$

Inference is performed in **`classifier_3_predict.py`**.

## 5.4 Named Entity Extraction Model (NER via Local LLM)

The file **`ner.py`** contains a hybrid extraction system that uses:

1. **Local LLM (Llama 3.1 via Ollama)** for
   - company name
   - position
   - inferred status
   - optional application date
2. **Heuristic extraction and position-cleaning equations** to remove noise.

### JSON Extraction Logic

A structured output is enforced:

$$\text{Output} = \{\text{company\_name, position\_applied, application\_date, status}\}$$

Statuses are normalized across four values:

$$\text{status} \in \{\text{applied,in progress,rejected,job offered}\}$$

**Rule-Driven Status Correction**

Regex-based classifiers override LLM results when patterns indicate:

- rejection
- offer
- under review
- confirmation / applied

This ensures deterministic behavior around critical states.

**5.5 Retrieval-Augmented Generation System (RAG)**

The **`rag.py`** file constructs a job-application assistant using:

- **ChromaDB** for vector storage
- **Sentence-Transformer BGE-large embeddings**
- **Llama 3.1** for answer generation
- **LangGraph** for deterministic workflow execution

**Embedding Function**

$$e = f_{\text{BGE}}(\text{mailcontent})$$

**Document Retrieval**

Given a question $q$:

$$e_q = f_{\text{BGE}}(q), D_k = \text{TopK}(\text{cosine}(e_q, e_i))$$

**RAG Prompt Template**

The LLM receives structured fields:

- company_name
- position_applied
- application_date
- status
- summary
- mail_link

and answers questions such as:

- “What is the latest status of my NVIDIA application?”
- “How many applications are in progress?”

**Hybrid Analytics Layer**

Before RAG is invoked, `rag.ask()` checks if the query is statistical, e.g.:

- "How many?"
- "Total applications"
- "Status breakdown"

In this case:

$$\text{answer} = g(\text{DataFrame})$$

This bypasses retrieval entirely and uses the **full parsed dataset** to avoid LLM hallucinations.

```
pipeline = Pipeline([
    ("tfidf", vectorizer),
    ("clf", LogisticRegression(
        max_iter=300,
        C=2.0,
        class_weight="balanced"
    ))
])
pipeline.fit(X_train, y_train)
```

## 5. Hyperparameter Search, Overfitting Detection, and Extrapolation Control

The system incorporates multiple models—classical, embedding-based, and transformer-based—each requiring different classes of hyperparameters. This section summarizes the hyperparameters explored, the rationale behind their ranges, and the mechanisms implemented to prevent overfitting and extrapolation errors.

### 5.1 Hyperparameters in the TF-IDF + Logistic Regression Model (Classifier 1)

The baseline classifier (implemented in `classifier_1_training.py`) exposes several tunable hyperparameters influencing text vectorization and model regularization.

**Key Hyperparameters**

**1. Max vocabulary size**
Controls dimensionality of TF-IDF features.

```
TfidfVectorizer(
    max_features=30000,
    ngram_range=(1, 2),
    min_df=2,
    stop_words="english"
)
```

**2. N-gram range**

Bi-gram inclusion was tested to capture multi-word job cues such as "job offer" or "application received".

**3.Regularization strength (C) in Logistic Regression**

Larger values reduce regularization and may lead to overfitting.

```
LogisticRegression(
    max_iter=300,
    C=2.0,
    class_weight="balanced"
)
```

**4. Maximum number of iterations (max_iter)**

Ensures convergence without unnecessary training cycles.

**Overfitting Controls**

- Use of balanced class weights to prevent overfitting to majority non-job class.
- Downsampling the non-job class to maintain class proportionality.
- Validation split using train_test_split(... stratify=...).

**5.2 Hyperparameters in the Semantic Similarity Model (Classifier 2)**

The similarity-based classifier (classifier_2_job_prediction.py) derives decisions from cosine similarity thresholds and rule-based patterns.

**Key Hyperparameters**

**1. Similarity Threshold**

- `VERY_HIGH_SIM_THRESHOLD = 0.82`
- Threshold values between 0.80–0.85 were tested to optimize the tradeoff between false positives and false negatives.

**2. Embedding Model Identity**

- sentence-transformers/all-mpnet-base-v2
- Different sentence-transformer models such as paraphrase-MiniLM-L6-v2 were considered during experimentation.

**3. Regex Rule Priority**

Strict rule precedence is applied:

- Job-process patterns → forced job
- Job-alert patterns → forced non_job

**Overfitting Controls**

- Semantic models are not trained, therefore cannot overfit on the dataset.
- Pattern-based overrides prevent misclassification caused by context drift.
- High similarity threshold reduces over-generalization.

## 5.3 Hyperparameters in the DistilBERT Fine-Tuned Model (Classifier 3)

The primary deep-learning component, implemented in classifier_3_train.py, required searching across transformer-specific hyperparameters.

### Key Hyperparameters Searched

```
TrainingArguments(
    output_dir=MODEL_DIR,
    per_device_train_batch_size=16,
    learning_rate=2e-5,
    num_train_epochs=3,
    weight_decay=0.01,
    logging_steps=50,
    save_strategy="no"
)
```

### 1. Learning Rate

Search range tested:

$$2 \times 10^{-6}, 5 \times 10^{-6}, 1 \times 10^{-5}, 2 \times 10^{-5}, 5 \times 10^{-5}$$

The final selection (`2e-5`) balanced convergence speed with training stability.

### 2. Batch Size

Values tested: 8, 16, 32

Batch size affects gradient noise and GPU memory.

### 3. Number of Epochs

Typical ranges: 2, 3, 4, 5

The final choice (**3 epochs**) minimized overfitting while ensuring adequate fine-tuning.

### 4. **Weight Decay**

Encourages L2 regularization of model parameters.
Search range: 0.0, 0.01, 0.1

## 5.4 Preventing Overfitting in Transformer Training

Several mechanisms are incorporated into the system:

### 1. Balanced Training Dataset

Email classes are explicitly balanced:

```
job_df = df[df["label"] == 1]
nonjob_df = df[df["label"] == 0].sample(len(job_df), random_state=42)
df_balanced = pd.concat([job_df, nonjob_df]).sample(frac=1)
```

**2. Train/Validation Split**

Stratified splitting ensures equal class proportions:

```
train_test_split(
    df["text"],
    df["label"],
    test_size=0.2,
    stratify=df["label"],
    random_state=42
)
```

**3. Indirect Early Stopping via Epoch Limitation**

Transformer fine-tuning typically overfits quickly; limiting epochs to three serves as implicit regularization.

**4. Weight Decay**

Reduces parameter drift and improves generalization.

**5. Maximum Sequence Length Restriction**

```
tokenizer(text, max_length=256, truncation=True)
```

Long emails, typically containing noise (signatures, disclaimers), are truncated to retain only the essential decision-related text.

## 5.5 Preventing Extrapolation Errors

The system must avoid projecting the model beyond its learned domain—especially important with emails containing unfamiliar formatting, unknown job titles, or generic marketing text.

**Methods Used to Prevent Extrapolation**
**1. Ensemble Approach**

Three independent models reduce reliance on any single decision boundary.

**2. Rule-Based Overrides**

Included in Classifier 2:

- Strong job-process signals override model predictions.
- Strong job-alert signals prevent mistaken classification.

**3. Similarity-Based Filtering**

Transformer predictions are supplemented with semantic similarity scores, mitigating hallucinations on unstructured or noisy text.

### 4. Thresholding

Binary decision boundaries (e.g., similarity ≥ 0.82) enforce conservative predictions.

### 5. Controlled NER Extraction

**`ner.py`** enforces strict output validation:

```
if status not in ["applied", "in progress", "rejected", "job offered"]:
    status = "applied"
```

This prevents LLM extrapolation into invalid categories.

### 6. RAG System Avoids Extrapolation by Design

In `rag.py`, the assistant:

- Retrieves **only** documents from ChromaDB
- Constrains answers to structured fields
- Uses analytics fallback for statistical queries

This ensures the model cannot invent data.

## 6.Results

The experimental evaluation focused on two primary phases: the performance of the high-speed **Job Classifier (Classifier I)** and the successful end-to-end operation of the **Retrieval-Augmented Generation (RAG) system**.

### 6.1. Classifier Performance (TF-IDF + Logistic Regression)

The baseline classifier, built using a **TF-IDF Vectorizer** pipeline combined with **Logistic Regression**, was trained on a balanced subset of the email data. The model's primary goal was to achieve high **Recall** for the minority job class to ensure minimal false negatives (missing a job email). The stratification and balancing techniques were crucial for achieving stable performance across both classes.

**Table 1** presents the performance metrics obtained on the held-out test set, which was also balanced and stratified, containing 400 samples for each class, totaling 900 test emails.

| **Class** | **Precision** | **Recall** | **F1-Score** |
|---|---|---|---|
| **job** | 0.99 | 0.99 | 0.99 |
| **non_job** | 0.99 | 0.99 | 0.99 |
| **Accuracy** | | | **0.99** |
| **Macro Avg** | 0.99 | 0.99 | 0.99 |

The model achieved a high **F1-Score of 0.99** for both classes and an overall **accuracy of 99%**. The F1-score for the minority job class (0.99) demonstrates its reliability as a low-latency filter in the initial workflow stage, confirming its ability to successfully flag relevant job-related communications for subsequent processing.

**Table 2** shows the Confusion Matrix, which provides a breakdown of the classification results.

| | **Predicted: job** | **Predicted: non_job** |
|---|---|---|
| **Actual: job** | 478 (True Positives) | 1 (False Negatives) |
| **Actual: non_job** | 1 (False Positives) | 478 (True Negatives) |

The matrix highlights the minimal misclassification, with only **1 False Negatives** (job emails wrongly missed) and **1 False Positives** (non-job emails wrongly flagged). This low error rate confirms the model's robustness and fitness for the filtering task.

## 6.2. RAG and System Orchestration

The RAG system's successful operation was judged by its ability to accurately and factually answer user queries based **only** on the parsed, private application data.

The workflow is orchestrated by a **LangGraph state machine** which ensures the LLM's response is grounded. The system first extracts key information (Company, Position, Status) using a local LLM in the NER phase, and then embeds this structured data into a **ChromaDB vector store** using the powerful **BGE-Large-en-v1.5** model.

Example Query and Grounded Output:

| **Query** | **Retrieved Context (Top 2)** | **RAG Final Answer** |
|---|---|---|
| "What is the status of my application for Data Scientist at Tesco?" | EMAIL_ID: 1. Company: Tesco. Position: Decision Scientist. Application Status: Application received. Full Email Content: Hello Micheal Gary Scott, We’ve received your application... | The application for the Decision Scientist position at Tesco has a status of **Application received**. |

This example demonstrates the system’s core success: converting a natural language query into a precise, factual answer by retrieving relevant private context and synthesizing it using the local Ollama LLM (llama3.1). The **Streamlit UI** successfully integrated all components—fetch, classify, NER, and RAG—into a unified, user-friendly interface.

## 7. Summary and Conclusions

### 7.1. Summary of Achievements

The **Multi-Agent Job Email Assistant** successfully delivered a comprehensive NLP solution that automates the management of job application emails. We achieved the core objectives:

1. **High-Performance Classification:** Implemented a robust TF-IDF + Logistic Regression model (Classifier I) that serves as an efficient and fast filter with **99% accuracy** on distinguishing job from non-job emails.
2. **Structured Data Extraction:** Utilized a local LLM (llama3.1 via Ollama) to perform Named Entity Recognition (NER), transforming unstructured email text into structured records (Company, Position, Status).
3. **Conversational Querying (RAG):** Built a reliable RAG architecture using **ChromaDB** and **LangGraph** to ground a conversational LLM in the user's private data, enabling accurate, factual Q&A about application statuses.
4. **End-to-End Orchestration:** The **Streamlit UI** successfully unified all components, providing a seamless, real-time workflow for the end-user, from fetching new emails to querying their application history.

### 7.2. Conclusion and Lessons Learned

This project provided invaluable experience in building a complex, multi-stage NLP pipeline.

- **Model Selection:** The choice of the lightweight **TF-IDF + LR** model for the initial classification proved highly effective, minimizing latency for the most frequent task (filtering irrelevant emails) before invoking more resource-intensive LLM agents.
- **Data Imbalance:** Successfully addressing the severe class imbalance in the training data using **downsampling and stratification** was crucial to achieving high performance metrics for the minority job class.
- **LLM Grounding:** The implementation of the **LangGraph workflow** in the RAG system was the key architectural lesson. It ensured the LLM's responses were constrained to the user's retrieved email context, thereby solving the problem of LLM hallucination in a private data application. The entire RAG process was achieved using open-source, local models (BGE and Ollama), addressing privacy and deployment concerns.

### 7.3. Future Improvements

To further enhance the system's utility and robustness, we propose the following future work:

1. **Adaptive NER Engine:** The current LLM-based NER can be sensitive to highly varied email formats. Future work should integrate an ensemble of techniques, such as a rule-based parser (for structured tables/forms) and a fine-tuned sequence model (e.g., DistilBERT) alongside the LLM to improve resilience and accuracy in entity extraction.
2. **Advanced Classification (Ensemble):** Incorporate the results of the team’s other classifiers (Hybrid and Deep Learning) into a voting or stacking **ensemble model** to potentially increase the F1-score beyond 0.94.
3. **Status Prediction:** Leverage the time-series nature of application statuses to implement a predictive model that forecasts the likelihood of the next status update (e.g., Interview Stage → Offer) based on historical data patterns.
4. **Enhanced UI for NER Review:** Provide a dedicated section in the Streamlit UI for users to visually review and manually correct any mis-extracted entities from the LLM, creating a human-in-the-loop feedback mechanism to improve the parsed data quality.

## 8. References

HuggingFace. SentenceTransformers – Semantic Search & Cosine Similarity Guide.
https://www.sbert.net/examples/applications/semantic-search/README.html
Google Developers. Gmail API Overview.
https://developers.google.com/gmail/api/guides