B”H

Email Classification GenAI Solution

Below is a concise overview addressing the first two deliverables.

The third delivery (Slide Presentation) can be found under the slides folder in the repository of this project as well as a simple grasp of the project’s code and structure: https://www.github.com/ShomoVolosky/email-classification

Further on this document, you will find a comprehensive, in-depth explanation of the selected solution and the methods employed to fulfill the task.

Deliverable 1: Output of the Required Development

What the Output Looks like:

1. Trained Model(s):
   - A multilingual model (English and Hebrew, for example bert-base-multilingual-cased or xlm-roberta-base) fine-tuned on the company's email dataset.

   - Model artifacts include:
     (1) pytorch_model.bin: model's weights file.
     (2) config.json
     (3) tokeniker file (for example vocab.txt)

2. Web-based (or CLI-based) Classifier Application:
   - A local Flask (or Streamlit) application that can:
     (1) Accept Subject and Body of an email (in English or Hebrew).
     (2) Output the Category (for example: HR, Finance, IT Support).
     (3) Output the Priority level (High, Medium, Low)
     (4) Optionally allow feedback from the user to confirm or correct the classification.

3. (Optional) RAG / Agentic AI Extension:
   - If LangChain / LangGraph or any other retrieval mechanism have been integrated, the user would have:
     (1) A basic retrieval system (using a Vector DB, for example FAISS or Pinecone) loaded with relevant domain docs.
     (2) A script or function (an agent) in the code to handle RAG-based lookups.

4. Evaluation Report:
   - Performance metrics (accuracy, precision, recall, F1) on a held-out test set.

In short, the main deliverable is a working pipeline and interface that shows how an email's subject and body get automatically classified and prioritized with optional user feedback.

Deliverable 2: Code for the Development, Organized and Versioned by Stages

Structure of the project's directory:

```
C:.
\---email-classification
    |   .gitignore
    |   Dockerfile
    |   LICENSE
    |   README.md
    |   requirements.txt
    |
    +---data
    |   +---processed
    |   \---raw
    +---images
    |       email_classification.jpg
    |
    +---model_output
    |       config.json
    |       encoder.json
    |       merges.txt
    |       pytorch_model.bin
    |       special_tokens_map.json
    |       tokenizer_config.json
    |       vocab.txt
    |
    +---notebooks
    |       01_data_preparation.ipynb
    |       02_model_finetuning.ipynb
    |       03_evaluation.ipynb
    |
    +---slides
    |       EmailClassification_GenAISolution.docx
    |       EmailClassification_Presentation.pptx
    |
    \---src
        |   agent.py
        |   app.py
        |   feedback.py
        |   utils.py
        |   __init__.py
        |
        \---templates
                index.html
```

<u>Stage 1: Data Collection and Preparation:</u>

Steps:

(1) Load raw email CSV / JSON datasets into a Pandas DataFrame,
(2) Clean the text (remove HTML, URLs, special characters),

(3) (Optional) Identify language (if for example both Hebrew and English are in the same dataset and the user wants to label them),
(4) Split into train/test sets in a balanced manner.

Stage 2: Model Training and Fine-Tunning:

Steps:

(1) Install and import Hugging Face transformers and datasets,
(2) Load the multilingual model and tokenizer,
(3) Prepare the dataset for tokenization and label alignment,
(4) Fine-tune for category classification; optionally multi-task for priority or train a separate model.
(5) Save the trained model artifacts in model_output/.

Stage 3: Evaluation:

Steps:

(1) Evaluate on the test set.
(2) Display accuracy, precision, recall, F1.
(3) (Optional) Show a confusion matrix.

Stage 4: UI/Chatbot and Feedback Loop:

Steps:

(1) Create src/app.py:
    - A Flask/Streamlit app that loads the saved model and provides a web form for email text.
    - Returns the predicted category and/or priority.
    - Incorporates a feedback function (thumbs up/down or correct labels).
    -
(2) Create src/feedback.py:
    - This is meant to handle storing user feedback in a local SQLite DB or a cloud-based database.
(3) (Optional) Create src/agent.py:
    - If implementing RAG or agentic AI via LangChain or LangGraph.

Why These Methods Were Chosen:

(1) Multilingual Transformer Models (mBERT / XLM-R): These models handle text from multiple languages with minimal separate engineering work. They also leverage massive pretrained datasets, reducing the labeled data requirement.

(2) NLP Preprocessing (NLTK / Regex / Potential Hebrew Tools): Basic text cleanup is essential for robust input to the model. NLTK is a mature library for tokenization, although advanced Hebrew tokenization might need specialized libraries (for example HebrewNLP – can be found at: https://discuss.huggingface.co/t/hebrew-nlp-introduction/4095) or rely on the transformer's build-in tokenizer.

(3) Feedback Loop / Human-in-the-Loop: Real-world data can differ from training data. Continual feedback ensures the model evolves and correct biases.

(4) Agentic AI / RAG: For advanced capabilities like referencing internal documents or providing explainable answers. Future-proofs the system so that it can reference additional context on the fly.

(5) Cloud Deployment (GCP, Azure, AWS): Offers scalable, reliable infrastructure with integrated CI/CD, managed services (compute, data storage, APIs), and ease of real-time integration (Pub/Sub, Event Hubs, etc.).

(6) UI / Chatbot: Provides an intuitive interface for non-technical users. Enhancing adoption and makes the classification results easily actionable.

In-depth Implementation:

1. Data Collection and Preparation:

(1.1) Data Sources:

- Historical emails (in Hebrew and English) exported from Gmail / Outlook or internal email servers.

- Labeled with categories (HR, Finance, etc.) and priority levels (High, Medium, Low).

(1.2) Data Cleaning (NLP Preprocessing).

(1.3) Tokenization and Multi-lingual Handling: Possibly store language tags in the dataset (English / Hebrew) to let the model handle domain adaptation.

(1.4) Train /Test Split and Balancing: stratify can ensure a balanced representation in train/test.

2. Model Training and Fine-Tuning:

The code examples provided can be done locally on a beefy machine with GPU, or in the cloud (using for example, Vertex AI, Azure ML, or some local server with GPU).

3. RAG / Agentic AI Integration:

For advanced use cases, we could add a Retrieval Augmented Generation pipeline or an agent:

(3.1) To store domain-specific documents (for example, department guidelines, FAQ, priority rules) in a vector store (for example, FAISS, Pinecone).

(3.2) LangChain / LangGraph can be used to build a pipeline:

- An agent queries the vector store for relevant documents based on the email's content (for example, if the user asks, "Why was this classified as HR?").
- The language model can incorporate these documents in the reasoning process.
- This approach can provide explanations or allow for more interactive Q&A about the classification logic.

4. UI / Chatbot Development:

(4.1) Local Developer Testing:

- A simple console or Flask / Streamlit app that runs on localhost.

- Allows the data scientist/engineer to quickly test classification on sample emails.

(4.2) User Acceptance Testing (UAT) UI:

- Include feedback functionality. For example, after classification, the UI can ask: "Did I predict correctly?"

- This feedback is logged and used later for retraining.

(4.3) Production UI / Chatbot:

- Could be more polished with a React/Angular front-end, or integrated with Microsoft Teams, Slack, or an internal chat platform.

- Additional features could include advanced search across classified emails, role-based access control (for overriding classifications), integration with enterprise SSO (Azure AD, Google Workspace).

5. Human-in-the-Loop and Active Learning:

(5.1) Store User Feedback: Each time a user corrects a classification, log (email_id, user_label, user_priority, model_prediction, timestamp), then store in a central DB (for example, PostgreSQL, MongoDB).

(5.2) Retraining Pipeline:

- On a scheduled basis (daily, weekly) or on-demand, incorporating new labeled data into the training set.

- Use Active Learning strategies to specifically ask humans about emails that the model found uncertain.

(5.3) Versioning:

- Maintain versioned checkpoints of the model.

- Compare performance metrics across versions.

6. Integration with Enterprise Email Systems:

(6.1) Gmail:

- Use the Gmail API to fetch incoming messages in real-time or via periodic polling.

- For real-time events, push notifications can be set to a webhook endpoint (for example will need a GCP Pub/Sub subscription).

(6.2) Outlook (Microsoft 365):

- Use the Microsoft Graph API or another IMAP approach.

- Also needs to subscribe to incoming email events.

(6.3) Workflow example:

- Kafka receives an event with the email's metadata.

- A microservice calls the classification model endpoint to get the predicted category and priority.

- The microservice updates metadata in the email system or writes it to a DB or user interface.

7. Deployment and Cloud Integration:

(7.1) Local Docker:

- Build a Docker image containing the model, code and environment.

- Test locally (Flask/Streamlit, model interface, Postman requests).

(7.2) Deploy Docker Image to Cloud:

- GCP (Cloud Run, GKE Kubernetes cluster or Vertex AI Endpoint).

- Azure (Azure Container Instances, Azure Kubernetes Service, Azure App Service).

- Leverage large-scale interface with GPU/TPU.

(7.3) Real-Time Pipeline with Kafka:

- Producers: Email ingestion service or an Outlook/Gmail connector.

- Consumers: A microservice that classifies emails by calling the model.

- Write results to a database or pass them downstream (for example, to a user UI, notifications).

(7.4) Real-Time Pipeline with Apache Flink:

- Streaming classification at scale.

- Inference calls embeddings or another advanced pipeline inside a Flink job.

- Flink's stateful stream processing can handle large throughput with sub-second latency.

(7.5) Scheduling and Workflow Orchestration with Airflow:

- Airflow DAG runs daily, pulling new labeled feedback data, and re-tunes the model if certain thresholds are met.

- After training, it can automatically push the new model artifact to a registry (for example, an S3 bucket, GCS bucker, Azure Blon) and trigger a CI/CD pipeline to roll out updates.