**PII Masking and Email Classification System: Project Report**

**Date:** May 19, 2025

**1. Introduction**

In the modern digital communication landscape, emails are a primary medium for information exchange. However, they often contain Personally Identifiable Information (PII) such as names, email addresses, phone numbers, financial details, and other sensitive data. The inadvertent exposure or mishandling of PII can lead to significant privacy breaches, regulatory non-compliance (e.g., GDPR, CCPA, HIPAA), and reputational damage for organizations.

This project addresses the dual challenge of:

1. **PII Masking:** Automatically identifying and obscuring PII within email text to protect sensitive information.
2. **Email Classification:** Categorizing emails based on their content to streamline workflows, prioritize responses, and enable efficient data analysis.

The goal was to develop a robust system, accessible via an API, that ingests raw email text, masks PII entities, and classifies the email into one of several predefined categories: "Change," "Incident," "Problem," or "Request." This system aims to enhance data security and improve operational efficiency in handling email communications.

**2. Approach and Methodology**

A modular approach was adopted, separating the PII masking and email classification tasks into distinct components, later integrated into a cohesive FastAPI application.

**2.1. PII Masking**

A hybrid strategy combining Named Entity Recognition (NER) and regular expressions (regex) was implemented for comprehensive PII detection:

* **Named Entity Recognition (NER):**
  + A pre-trained transformer-based model (Davlan/bert-base-multilingual-cased-ner-hrl) was employed. This model is proficient in identifying entities like person names ("PER" or "Person") in multilingual text.
  + Detected names are replaced with a generic placeholder (e.g., [full\_name]).
* **Regular Expressions (Regex):**
  + Custom regex patterns were developed to identify and mask structured PII that typically follows predictable formats. This includes:
    - Email addresses (masked as [email])
    - Phone numbers (masked as [phone\_number])
    - Dates of Birth (masked as [dob])
    - Aadhar numbers (masked as [aadhar\_num])
    - Credit/Debit card numbers (masked as [credit\_debit\_no])
    - CVV numbers (masked as [cvv\_no])
    - Card expiry dates (masked as [expiry\_no])
  + The system iterates through these patterns, replacing found PII with corresponding placeholders.

The PII masking module outputs both the masked email text and a list of dictionaries, where each dictionary details the original PII entity, its classification (type of PII), and its start and end character positions in the original email.

**2.2. Email Classification**

A supervised machine learning pipeline was developed for classifying the emails:

* **Text Preprocessing & Embedding:**
  + The masked email text (output from the PII module) serves as input for classification. Using masked text ensures that PII does not unduly influence classification and aligns the training/inference conditions.
  + The paraphrase-multilingual-mpnet-base-v2 model from the SentenceTransformers library is used to convert the input text into dense vector embeddings. These embeddings capture the semantic meaning of the text, allowing the model to understand context beyond keyword matching.
* **Classification Model:**
  + A Multi-Layer Perceptron (MLP) classifier, implemented using PyTorch (via the models.py structure), takes these sentence embeddings as input.
  + The MLP is trained to predict one of the four predefined email categories.

**3. Model Selection, Training, and Evaluation**

**3.1. Model Selection Rationale**

* **NER Model (Davlan/bert-base-multilingual-cased-ner-hrl):** Chosen for its strong performance on multilingual name recognition, suitable for diverse email content.
* **Sentence Embedding Model (paraphrase-multilingual-mpnet-base-v2):** Selected for its excellent performance in generating semantically rich embeddings for various downstream tasks, including classification, across multiple languages.
* **Classifier (MLP):** An MLP provides a good balance between performance and complexity for text classification tasks when using high-quality embeddings. It can learn non-linear relationships in the data.

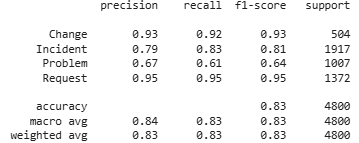
**3.2. Training Details (Assumed Generic Process)**

While specific details of the training dataset and hyperparameter tuning were not fully elaborated during the development iterations covered, a typical training process for the MLP classifier would involve:

1. **Dataset Collection:** A labeled dataset of emails, each assigned to one of the four categories.
2. **Data Splitting:** Division into training, validation, and test sets.
3. **Model Training:** The MLP is trained on the sentence embeddings of the training data, optimizing a loss function (e.g., cross-entropy loss) using an optimizer (e.g., Adam).
4. **Hyperparameter Tuning:** Potentially using the validation set to tune learning rate, number of layers/neurons in the MLP, dropout rates, etc.

**3.3. Evaluation and Performance**

The email classification model's performance was evaluated, yielding the following metrics:

****

**Discussion of Results:** The overall accuracy of the classification model is 83%, which is a good baseline.

* The model performs exceptionally well for "Change" and "Request" categories, with F1-scores of 0.93 and 0.95, respectively. This indicates high precision and recall for these classes.
* The "Incident" category shows reasonable performance with an F1-score of 0.81.
* The "Problem" category has the lowest F1-score (0.64), with precision at 0.67 and recall at 0.61. This suggests that the model finds it more challenging to accurately identify "Problem" emails, potentially due to overlapping features with other categories or insufficient distinct patterns in the training data for this class.

**4. Challenges Faced and Solutions Implemented**

Several challenges were encountered during the development process:

* **Environment and Dependency Management:**
  + **Challenge:** A ModuleNotFoundError arose related to an incompatible PyTorch version (torch==2.6.0 initially specified was problematic).
  + **Solution:** The PyTorch version was updated to a more stable/compatible version (torch==2.1.0, later torch==2.5.0 was settled on as per requirements.txt) in the requirements.txt file. Guidance was provided on recreating the virtual environment to ensure correct dependency resolution.
* **API Response Formatting for PII Positions:**
  + **Challenge:** The API needed to return a JSON response where PII entity position arrays (e.g., [start\_index, end\_index]) were formatted on a single line (e.g., "position": [10, 25]), while the rest of the JSON structure was pretty-printed with standard indentation for readability. Standard JSON serializers do not easily support such mixed formatting.
  + **Solution:** This required an iterative approach:
    1. Initial attempts using Pydantic models and default FastAPI JSON encoding were insufficient.
    2. A custom solution was developed involving a CompactListWrapper class to hold the position lists.
    3. A CustomJsonEncoder (subclass of json.JSONEncoder) was implemented. This encoder would identify CompactListWrapper instances and serialize them into unique string placeholders containing the compact list representation (e.g., "\_\_COMPACT\_LIST\_PLACEHOLDER\_\_[10,25]\_\_END\_PLACEHOLDER\_\_").
    4. The main JSON dump used indent=2 for overall pretty-printing.
    5. A regular expression (re.sub) was then used to find these quoted placeholders in the generated JSON string and replace them with their unquoted, compact list content.
    6. This logic was encapsulated by creating a CustomFormattedJSONResponse class (inheriting from fastapi.responses.JSONResponse), which overrode the render method to apply this custom serialization and placeholder replacement strategy. This provided a clean integration with FastAPI.
* **Model Deployment and Portability (Dockerization):**
  + **Challenge:** Ensuring the application, along with its models and dependencies, could be reliably built, deployed, and run in various environments.
  + **Solution:** A Dockerfile was created to containerize the FastAPI application. Key aspects of the Dockerfile include:
    1. Using python:3.10-slim as the base image.
    2. Setting environment variables (PYTHONDONTWRITEBYTECODE, PYTHONUNBUFFERED).
    3. Installing system dependencies like git via apt-get (necessary for fetching some Hugging Face models if not pre-cached or if models are specified by repo).
    4. Copying requirements.txt and installing Python dependencies using pip install --no-cache-dir -r requirements.txt.
    5. **Pre-downloading Models:** To ensure models are available within the image and reduce startup time/reliance on live downloads during container run, steps were added to pre-download and save the NER model (Davlan/bert-base-multilingual-cased-ner-hrl) and the SentenceTransformer model (paraphrase-multilingual-mpnet-base-v2) into specified directories (./model and ./sbert\_model respectively) within the Docker image during the build process. This was achieved using RUN python -c "..." commands.
    6. Copying the application source code (main.py, utils.py, models.py, etc.) into the image.
    7. Exposing the application port (e.g., 7860 as per the final Dockerfile configuration).
    8. Defining the CMD to run the application using Uvicorn: CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "7860"].
* **PII Position Calculation Accuracy:**
  + **Challenge:** An ongoing issue reported was that the start and end positions for masked PII entities were sometimes being calculated incorrectly by the utils.py logic. (This was noted as an active issue at the time of this report).
  + **Solution:** This was identified as an area requiring further debugging and refinement in the PII masking logic within utils.py. The logic for calculating indices, especially when multiple PII types are processed sequentially and text length changes, needs careful review.

**5. Conclusion and Future Work**

The project successfully developed a FastAPI-based system capable of masking Personally Identifiable Information from email text and subsequently classifying these emails into predefined categories. The PII masking utilizes a hybrid NER and regex approach, while classification leverages advanced sentence embeddings and an MLP model, achieving an overall accuracy of 83%. Significant effort was invested in ensuring correct API response formatting and creating a portable Dockerized deployment.

**Future Work and Recommendations:**

* **Improve "Problem" Category Classification:**
  + Investigate techniques to improve precision and recall for the "Problem" category. This could involve collecting more specific training examples for this class, data augmentation, exploring different classification model architectures, or fine-tuning hyperparameters.
* **Resolve PII Position Calculation:**
  + Prioritize debugging and correcting the logic for calculating PII entity positions in utils.py to ensure accurate localization information is provided in the API response. This is crucial for downstream applications that might rely on these indices.
* **Expand PII Coverage:**
  + Extend the range of PII types detected by adding more NER categories (if applicable models are found or fine-tuned) and more sophisticated regex patterns.
* **Enhanced Model Management:**
  + Implement more robust model versioning and management, especially if models are frequently retrained or updated.
* **Scalability and Performance Optimization:**
  + For higher throughput, investigate optimizations such as batch processing for model inference, asynchronous task handling for long-running operations, and potentially deploying multiple instances of the service behind a load balancer.
* **Comprehensive Testing:**
  + Conduct more extensive User Acceptance Testing (UAT) with real-world email data to gather feedback and identify further areas for improvement.
* **Security Hardening:**
  + Implement additional security best practices for the API, such as input validation beyond Pydantic, rate limiting, and authentication/authorization if required for the deployment context.

This project provides a solid foundation for an intelligent email processing system. Addressing the outlined future work will further enhance its capabilities, accuracy, and robustness.