Email Classification and PII Masking System: A Detailed Report

Author: VIRIKA OLIVIA SOANS

Date: June 2, 2025

1. Introduction to the Problem Statement

This project addresses the critical need for an automated email classification and Personally Identifiable Information (PII) masking system within a company's support team. The primary objective is to efficiently categorize incoming support emails into predefined categories while ensuring that personal information (PII) is masked before processing. After classification, the masked data should ideally be restored to its original form, though the API output here focuses on the masked version for security. The final solution is deployed as an API on Hugging Face Spaces, strictly adhering to specified input/output formats and router names for automated evaluation.

2. Approach Taken for PII Masking and Classification

The system is designed as a pipeline: emails are received, PII is masked, masked emails are classified, and the result is returned.

2.1 PII Masking (Without LLMs)

The PII masking component, implemented in pii_masker.py, explicitly avoids the use of Large Language Models (LLMs).It leverages a combination of Named Entity Recognition (NER) using spaCy and regular expressions (Regex) to detect and mask sensitive information.

The specific PII fields identified and masked include.

- Full Name (full_name)
- Email Address (email)
- Phone number (phone_number)
- Date of birth (dob)
- Aadhar card number (aadhar_num)
- Credit/Debit Card Number (credit_debit_no)
- CVV number (cvv_no)
- Card expiry number (expiry_no)

As an example, an input like "Hello, my name is John Doe, and my email is johndoe@example.com." is masked to "Hello, my name is [full_name], and my email is [email]. The original entities and their positions are stored for later demasking and inclusion in the API output.

2.2 Email Classification

For email classification, a deep learning approach using a fine-tuned BERT model is employed. This choice aligns with the assignment's allowance for deep learning models or Large Language Models (LLMs) for classification. The model is trained to categorize emails into four predefined support types: Incident, Request, Change, and Problem.

3. Model Selection and Training Details

3.1 Data Collection & Preprocessing

The system uses the provided dataset of emails containing different types of support requests. During preprocessing (train_bert_classifier.py), the emails are first masked for PII. A 10% sample of the data is taken for quicker training and iteration. Labels are encoded using LabelEncoder from sklearn.preprocessing.

3.2 Model Selection and Training (train_bert_classifier.py)

- **Model:** A BertForSequenceClassification model, pre-trained with bert-base-uncased, is fine-tuned for the classification task.
- **Tokenizer:** BertTokenizer.from_pretrained("bert-base-uncased") is used to tokenize the masked email text.
- **Dataset:** The raw emails are loaded from combined_emails_with_natural_pii.csv. After masking, the emails are tokenized and converted into a datasets.Dataset object, which is then split into training and testing sets (80% train, 20% test).
- **Training Arguments:** transformers.TrainingArguments are configured with a per_device_train_batch_size=4 and num_train_epochs=2 to manage resource usage and speed up training.
- **Trainer:** A transformers.Trainer is used to manage the training process, saving the fine-tuned model and tokenizer to a ./bert_model directory and the LabelEncoder to label_encoder.pkl upon completion.
    - *Note:* The vectorizer.pkl and classifier.pkl mentioned in your files list are not used in the BERT-based app.py and train_bert_classifier.py, indicating they are remnants of a different approach or not required for this BERT implementation.

4. API Development & Deployment (app.py)

The API is built using FastAPI, as recommended by the assignment. It exposes a single POST endpoint /classify, which accepts an email body as input in a specific JSON format.

4.1 API Functionality:

- **Input:** {"input_email_body": "string containing the email"}
- **Process:**
    1. Receives the input_email_body
    2. Calls mask_pii() to mask PII and extract entities
    3. Tokenizes the masked_email.
    4. Passes the tokenized input to the fine-tuned BERT model for classification.

5. Obtains the predicted category using the label_encoder.

- **Output:** {"input_email_body": "original string", "list_of_masked_entities": [...], "masked_email": "masked string", "category_of_the_email": "predicted class"}

4.2 Deployment on Hugging Face Spaces:

The solution is deployed on Hugging Face Spaces using a Docker SDK. The Dockerfile handles the environment setup and dependency installation, including spacy model download. The app.py is configured to run as the main application. Crucially, the BertForSequenceClassification.from_pretrained("./bert_model", local_files_only=True) and BertTokenizer.from_pretrained("./bert_model", local_files_only=True) calls ensure that the model and tokenizer are loaded directly from the local bert_model directory within the deployed Space, preventing attempts to download from the Hugging Face Hub. The label_encoder.pkl is also loaded locally. The deployment should not have a frontend app

5. Challenges Faced and Solutions Implemented

1. **"Not a Git Repository" Error:** Initial attempts to push local changes to GitHub failed due to the project directory not being recognized as a Git repository.
   - **Solution:** The local project was either correctly initialized as a new Git repository (git init) and linked to the remote, or the GitHub repository was cloned (git clone) and subsequent work/commits were performed within the cloned directory. This ensured proper Git tracking.
2. **transformers Library Searching Hugging Face Hub for Local Model:** The transformers library's from_pretrained() method initially tried to validate the local path ("./bert_model") as a Hugging Face Hub repo_id, leading to RepositoryNotFoundError or HFValidationError.
   - **Solution:** Explicitly setting local_files_only=True in both model.from_pretrained() and tokenizer.from_pretrained() calls in app.py resolved this, forcing the library to load exclusively from the local file system.
3. **Corrupted/Missing config.json:** An OSError: ...not a valid JSON file indicated that config.json was empty or corrupted within the bert_model directory.
   - **Solution:** The existing bert_model directory and label_encoder.pkl were deleted locally. train_bert_classifier.py was then rerun to regenerate all model artifacts, ensuring config.json and other files were complete and valid. These regenerated files were then committed and pushed to GitHub.
4. **Hugging Face Spaces Folder Upload Misconception:** Initially, there was confusion regarding how to "upload" entire folders like bert_model/ to Hugging Face Spaces.
   - **Solution:** Clarification was provided that for a Git-backed Space, "uploading" means committing the folder and its contents to the linked GitHub repository using git add ., git commit, and git push. The COPY . . command in the Dockerfile then handles copying these files into the Docker container during the build process.

6. Final Output

- **API Endpoint Details for Testing:**

- **Base URL Example:** https://your-username-your-space-name.hf.space (replace with your actual Space URL)
- **Router:** /classify
- **Full Endpoint Example:** https://your-username-your-space-name.hf.space/classify
- **Method:** POST

- **Example Input JSON Structure:**
- {
- "input_email_body": "Hi, I’m John. My number is 9876543210. I need help with billing."
- }

- **Example Output JSON Structure:**
- {
- "input_email_body": "Hi, I’m John. My number is 9876543210. I need help with billing.",
- "list_of_masked_entities": [
- {
- "position": [9, 13],
- "classification": "full_name",
- "entity": "John"
- },
- {
- "position": [25, 35],
- "classification": "phone_number",
- "entity": "9876543210"
- }
-],
- "masked_email": "Hi, I’m [full_name]. My number is [phone_number]. I need help with billing.",
- "category_of_the_email": "Request" // Example category
- }

7. Code Implementation Details

The project adheres to a modular structure for clarity and maintainability:

- **app.py**: The entry point for the FastAPI application. It defines the /classify endpoint, handles input/output, orchestrates PII masking, and calls the classification model.
- **pii_masker.py**: Contains the logic for PII detection and masking using spaCy and regular expressions. It returns the masked text and a list of identified entities with their original values and positions.
- **train_bert_classifier.py**: Responsible for data loading, preprocessing (including PII masking during training), label encoding, BERT model initialization, tokenizer loading, and the fine-tuning process using the transformers.Trainer API. It saves the trained model, tokenizer, and LabelEncoder for use in the API.
- **requirements.txt**: Lists all necessary Python dependencies for the project, ensuring a reproducible environment in the Docker container.
- **README.md**: Provides a clear overview of the project, API features, categories, PII types, and usage instructions, as specified in the assignment guidelines.

Github Repository Link: https://github.com/V-virika/Email_classifier_BERT

Hugging face: https://huggingface.co/spaces/Virika/Email-Classifier-virika/tree/main