Email Classification System

Project Description:

The Email Classifier is an intelligent full stack application designed to automatically categorize emails using machine learning techniques. The system integrates a React based user interface with a FastAPI backend and a trained classification model. It connects to Gmail through IMAP, retrieves email content, processes the text using TF IDF vectorization, and predicts one of four categories: Urgent, HR, Financial, or General.

The interface includes a unique interactive 3D solar system created with Three.js, offering an engaging environment for user navigation while maintaining full functionality for email management. This project demonstrates an end to end implementation of machine learning, backend API development, frontend design, and secure communication between system components.

Project Scenarios

Scenario 1: Office Communication Management

A working professional receives a large number of emails every day. When the system synchronizes with their Gmail inbox, it categorizes each email automatically. Urgent messages appear in one section, HR related messages in another, and so on. This helps the user focus on priority tasks without manually sorting emails.

Scenario 2: Academic Inbox Management

Students often receive emails related to classes, events, deadlines, and general updates. The system classifies these messages instantly, allowing them to find important emails such as exam notices or fee reminders without searching through a cluttered inbox.

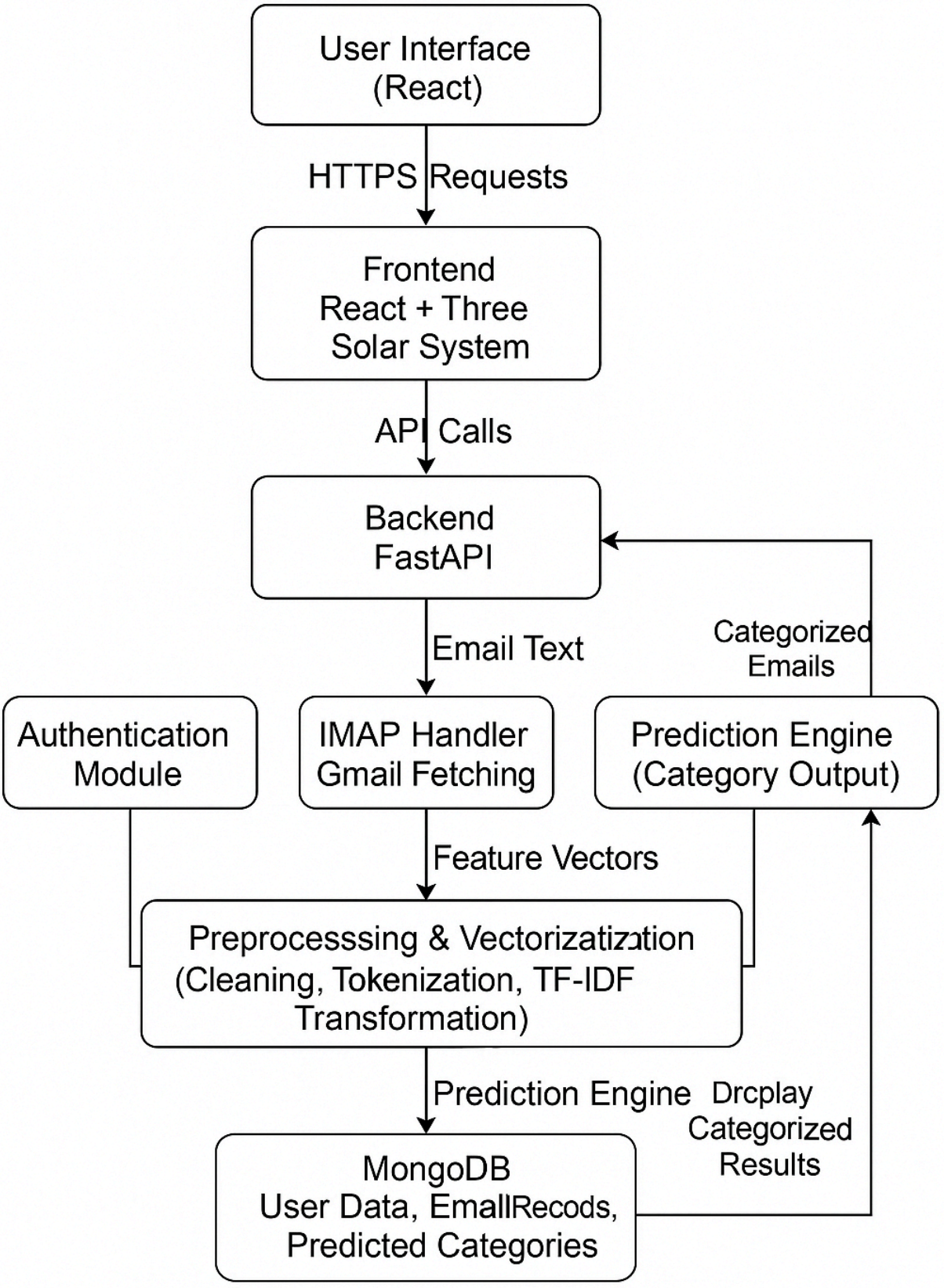
Scenario 3: Business Email Filtering

Small businesses receive financial updates, invoices, client communication, and internal notifications. By integrating this system, businesses can automatically categorize communications and improve workflow efficiency by identifying time sensitive emails.

Scenario 4: Enterprise Level Email Monitoring and Compliance

Large organizations often handle thousands of internal and external emails related to operations, finance, human resources, and urgent notifications. The Email Classifier system assists enterprises by automatically sorting incoming emails into predefined categories as soon as synchronization occurs. This helps compliance teams monitor financial or HR related communications, enables department heads to quickly identify urgent matters, and reduces manual effort in managing large volumes of emails. The system improves organizational efficiency by providing structured visibility into communication patterns and supporting faster decision making.

Email Classifier System Architecture



Prerequisites:

To complete this project, you must require the following software, concepts, and packages

- **Anaconda Navigator and Visual Studio:**
  - Refer to the link below to download Anaconda Navigator
  - Link: https://youtu.be/1ra4zH2G4o0
- **Python packages:**
  - Open anaconda prompt as administrator
  - Type "pip install numpy" and click enter.
  - Type "pip install pandas" and click enter.
  - Type "pip install scikit-learn" and click enter.
  - Type "pip install matplotlib" and click enter.
  - Type "pip install scipy" and click enter.
  - Type "pip install pickle-mixin" and click enter.
  - Type "pip install seaborn" and click enter.
  - Type "pip install Flask" and click enter.

Prior Knowledge:

You must have prior knowledge of the following topics to complete this project.

- **ML Concepts**
  - Supervised learning: https://www.javatpoint.com/supervised-machine-learning
  - Unsupervised learning: https://www.javatpoint.com/unsupervised-machine-learning
  - Logistic Regression: https://www.javatpoint.com/logistic-regression-in-machine-learning
  - Decision tree: https://www.geeksforgeeks.org/python-decision-tree-regression-using-sklearn/
  - Random forest: https://www.javatpoint.com/machine-learning-random-forest-algorithm
  - Evaluation metrics: https://www.analyticsvidhya.com/blog/2019/08/11-important-model-evaluation-error- metrics/
  - Navie Bayes: https://www.javatpoint.com/machine-learning-naive-bayes-classifier
- **Fast Api**: FastAPI for Beginners - Python Web Framework

Project Flow:

- User signs up and logs into the application.
- User sets up IMAP by entering Gmail address and app password.
- Backend connects to Gmail through IMAP and fetches email data.
- Email content is processed using the TF IDF vectorizer.
- Machine learning model predicts the category of each email.
- Emails along with their predicted categories are stored in MongoDB.
- User views categorized emails in the dashboard and filters them by category.
- User may sync new emails anytime through the interface.

Project Activities:

- **Data Collection & Preparation**
  - Collecting Email Data
  - Importing Libraries
  - Reading Data
  - Data Preparation
- **Exploratory Data Analysis**
  - Descriptive Statistics
  - Visual Analysis
  - Univariate and Bivariate Analysis
  - Correlation Analysis
  - Feature Scaling
  - Train Test Split
- **Model Building**
  - Implementing TF IDF Vectorization
  - Model Training
  - Model Evaluation
  - Saving the Model
- **Backend API Development**
- User signup and login with JWT
  IMAP connection handling
  Email syncing
  Running classification on fetched emails
  Returning categorized email data to the frontend

Backend Workflow

1. Load model and TF IDF vectorizer at startup.
2. Receive IMAP credentials.
3. Fetch unread emails.
4. Clean and transform email text.
5. Predict category.
6. Store result in MongoDB.

• Frontend Development

- Intro Page
- Login Page
- Dashboard

Project Structure:

Create the Project folder which contains files as shown below

```
email_classifier/
├── backend/
│   ├── main.py           # FastAPI application & routes
│   ├── imap_handler.py   # Email fetching & IMAP logic
│   ├── model/
│   │   └── classifier.py  # ML inference engine for email classification
│   ├── requirements.txt   # Python dependencies
│   └── venv/             # Virtual environment
├── frontend/
│   ├── src/
│   │   ├── pages/
│   │   │   ├── IntroPage.js    # 3D Solar System landing page
│   │   │   ├── LoginPage.js    # Network animation login
│   │   │   ├── SignupPage.js   # User registration
│   │   │   └── Dashboard.js    # Main email dashboard
│   │   ├── App.js
│   │   └── index.js
│   ├── package.json
│   └── public/
├── model.pkl             # Trained classification model
├── tfidf.pkl             # TF-IDF vectorizer
└── README.md
```

Project Structure Explanation

backend

- Contains all server side code.
- `main.py`: Handles API routes, authentication, model loading, and prediction.
- `imap_handler.py`: Connects to Gmail, fetches emails, and extracts text.
- `model/`: Holds the classification logic and preprocessing.
- `requirements.txt`: Lists required Python packages.

frontend

- Includes all React UI files.
- `IntroPage.js`: 3D solar system intro screen.
- `LoginPage.js`: User login interface.
- `SignupPage.js`: Account registration page.
- `Dashboard.js`: Displays categorized emails and sync options.
- `App.js` and `index.js`: Control rendering and routing.

model.pkl

- Saved machine learning model used for email classification.

tfidf.pkl

- Vectorizer used to convert email text to numerical features.

public

- Holds static assets used by the frontend.

Milestone 1: Data Collection & Preparation

Data collection is an essential step in any machine learning based system. For the Email Classifier project, this stage involves gathering a large set of emails, cleaning text content, and preparing the dataset for training. The quality of the collected data directly impacts the accuracy and reliability of the classification model.

Activity 1.1: Collect the dataset

The dataset used for training the classification model consists of email subjects and bodies labeled into four categories: Urgent, HR, Financial, and General. Email samples were collected from publicly available datasets, synthetic examples, and manually created records. Each email was assigned a category based on its content, ensuring clear representation across all labels.
Link:
https://www.kaggle.com/datasets/wcukierski/enron-email-dataset

Dataset Source

https://www.kaggle.com/datasets/wcukierski/enron-email-dataset

The dataset used in this project contains email text samples collected for building and evaluating the classification model. Each record consists of an email subject, email body, and its assigned category label. The dataset includes the following features:

- **Subject**: The subject line of the email.
- **Body**: The main content of the email message.
- **Combined Text**: A merged version of subject and body used for text processing.
- **Category**: Target variable representing the class of the email. The four categories used in this project are:

    1. Urgent
    2. HR
    3. Financial
    4. General

The dataset file is stored in a structured format and contains a large number of email samples with their corresponding manually assigned categories. This dataset forms the foundation for training, validating, and testing the machine learning model used in the Email Classifier system.

Activity 1.2: Importing the Libraries

Essential Python libraries such as pandas, numpy, scikit-learn, and joblib were imported to support text preprocessing, feature extraction, and model training. These libraries handle string operations, dataset loading, vectorization, and machine learning algorithms.

These libraries serve the following purposes:

- **NumPy:** For numerical computations
- **Pandas:** For data manipulation and analysis
- **Matplotlib:** For creating visualizations
- **Seaborn:** For statistical data visualization

Import the necessary libraries as shown below

```
import pandas as pd
import numpy as np
import re
import pickle
import os
from collections import Counter

import nltk
nltk.download('punkt')
nltk.download('punkt_tab')
nltk.download('stopwords')
nltk.download('wordnet')

nltk.download('punkt_tab')
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
from nltk.stem import WordNetLemmatizer

from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix
import seaborn as sns
import matplotlib.pyplot as plt
```

Activity 1.3: Read the Dataset

The dataset was loaded using pandas to inspect its structure. This allowed viewing columns, understanding email lengths, and confirming the presence of both subject and body fields. The initial inspection helps identify inconsistencies that may affect preprocessing.

	file	message
0	allen-p/_sent_mail/1.	Message-ID: <18782981.1075855378110.JavaMail.e...
1	allen-p/_sent_mail/10.	Message-ID: <15464986.1075855378456.JavaMail.e...
2	allen-p/_sent_mail/100.	Message-ID: <24216240.1075855687451.JavaMail.e...
3	allen-p/_sent_mail/1000.	Message-ID: <13505866.1075863688222.JavaMail.e...
4	allen-p/_sent_mail/1001.	Message-ID: <30922949.1075863688243.JavaMail.e...

Activity 1.3: Data Cleaning and Preparation

Before the email dataset can be used to train the machine learning model, it must be cleaned and prepared. Since email text often contains noise, formatting issues, and inconsistent structures, preprocessing is essential. For this project, data cleaning includes:

- **Handling missing subjects or bodies**
  Emails with empty or incomplete fields are removed, as they do not provide meaningful text for training.
- **Removing duplicate email entries**
  Duplicate emails are eliminated to prevent the model from learning repetitive patterns that could bias its predictions.
- **Checking data types and formatting**
  All text fields are converted into a consistent string format to ensure smooth processing during vectorization.
- **Cleaning text content**
  Unnecessary characters, HTML tags, signatures, and special symbols are removed to focus on meaningful text.

Note: These preprocessing steps may vary based on the condition of the dataset. Some steps may be optional if the dataset is already clean or well-formatted.

Activity 1.4 : Check the size of the dataset:

```
Dataset Info:
  Shape: (517401, 2)
  Columns: ['file', 'message']
  ✓ Found email text in column: 'message'

✓ Created DataFrame with 517401 emails
```

This returns the number of rows and columns in the dataset.

Milestone 2: Exploratory Data Analysis

Exploratory Data Analysis (EDA) helps identify patterns in the email text, common keywords, and variations across different categories. It aids in selecting appropriate preprocessing techniques and understanding the separation between classes.

Activity 2.1: Descriptive Statistics

Basic statistical summaries were generated, such as average text length and distribution of categories. This helps in understanding whether emails from certain categories tend to be longer or contain specific patterns.

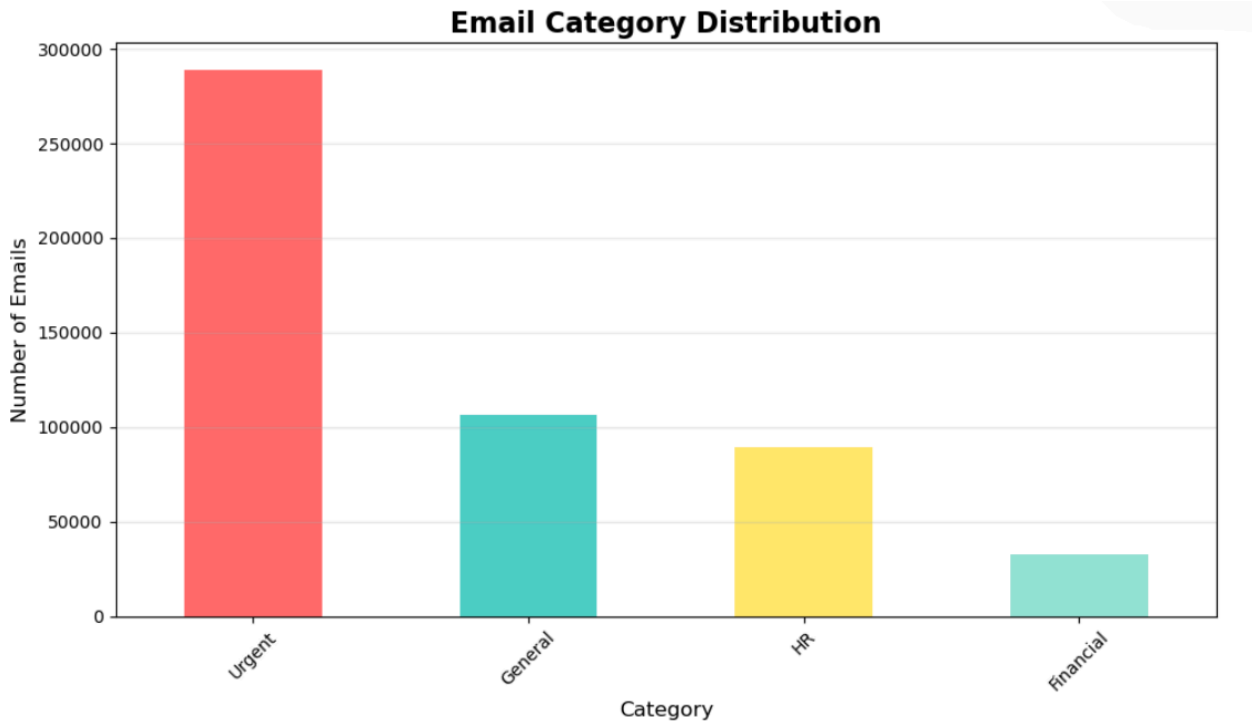
```
plt.figure(figsize=(10, 6))
category_counts.plot(kind='bar', color=['#FF6B6B', '#4ECDC4', '#FFE66D', '#95E1D3'])
plt.title('Email Category Distribution', fontsize=16, fontweight='bold')
plt.xlabel('Category', fontsize=12)
plt.ylabel('Number of Emails', fontsize=12)
plt.xticks(rotation=45)
plt.grid(axis='y', alpha=0.3)
plt.tight_layout()
plt.show()

# Show examples from each category
print("\n" + "=" * 70)
print("SAMPLE EMAILS BY CATEGORY:")
print("=" * 70)
for category in df_emails['category'].unique():
    sample = df_emails[df_emails['category'] == category]['email_text'].iloc[0]
    print(f"\n{category}:")
    print(f"  {sample[:150]}...")
```

```
======================================================================
[STEP 3] AUTO-LABELING EMAILS
======================================================================
Labeling emails...

✓ Labeled 517401 emails

======================================================================
CATEGORY DISTRIBUTION:
======================================================================
  Urgent      : 288698 emails (55.80%)
  General     : 106731 emails (20.63%)
  HR          :  89208 emails (17.24%)
  Financial   :  32764 emails ( 6.33%)
```

Activity 2.2: Visual Analysis

Word frequency plots were examined to observe commonly used terms within each category. This step highlights key phrases associated with urgent alerts, financial updates, HR instructions, and general communication.

Distribution of Numerical Features:



Histograms show the distribution of each numerical feature, helping identify:

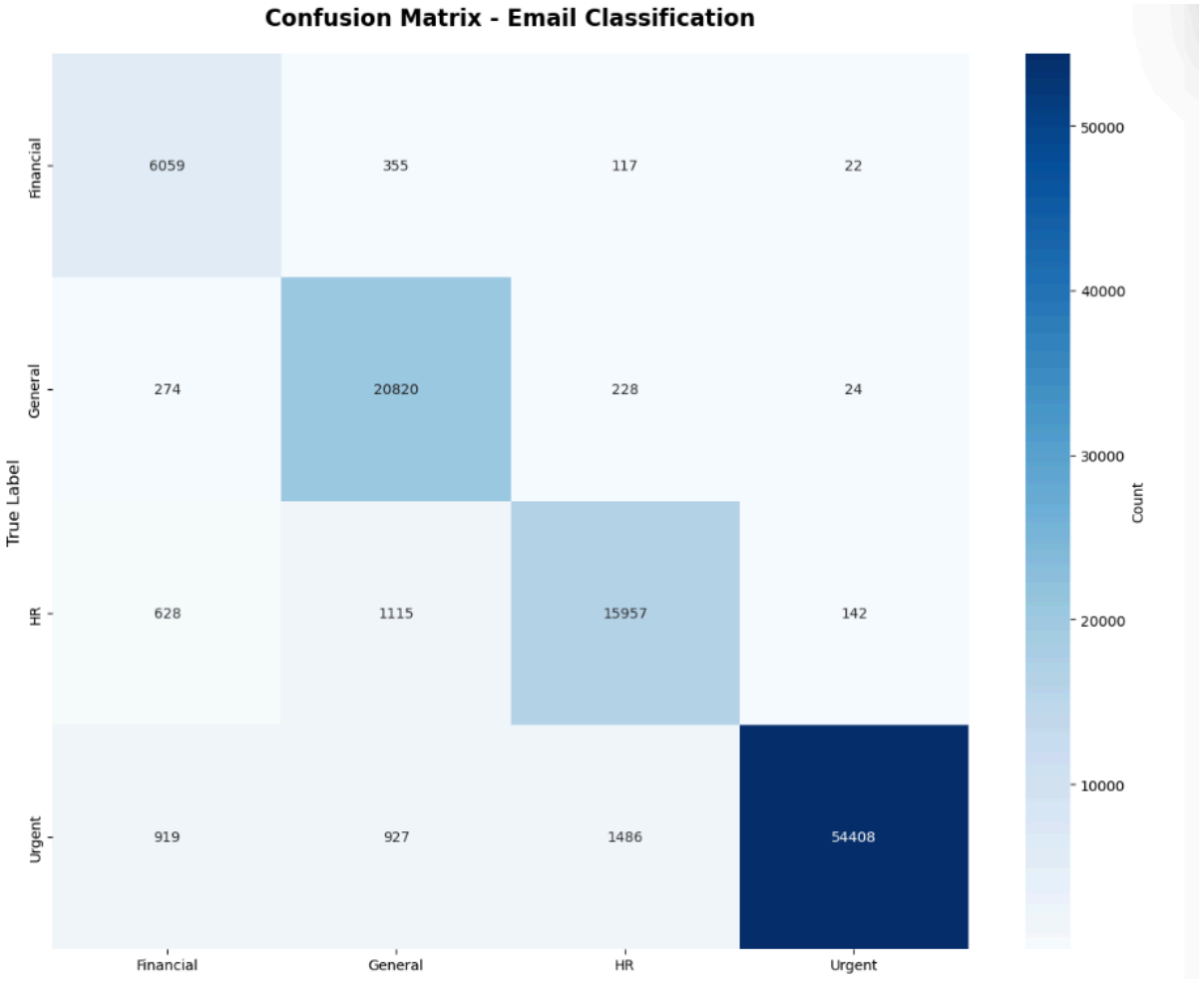
- Skewness in the data
- Most common value ranges
- Potential outliers

Activity 2.3: Confusion Matrix

A confusion matrix is used to evaluate how well the email classification model performs across all four categories. It compares the actual category of an email with the predicted category generated by the machine learning model. Each row in the matrix represents the true label, and each column represents the predicted label.

For this project, the confusion matrix includes the following four classes:

1. Urgent
2. HR
3. Financial
4. General



Activity 2.4: Train-Test Split

Split the dataset into training and testing sets:

```
from sklearn.model_selection import train_test_split

x = df.drop('Result', axis=1)
y = df['Result']

X_train, X_test, y_train, y_test = train_test_split(
    x, y, test_size=0.2, random_state=42
)
```

- **Training set (80%)**: Used to train the email classification model by learning patterns from the text data.
- **Testing set (20%)**: Used to evaluate how well the model performs on unseen emails and to verify its generalization ability.
- **random_state = 42**: Ensures consistent and reproducible splitting of the dataset each time the code is executed.

Milestone 3: Model Building

This milestone focuses on selecting, training, and evaluating machine learning algorithms for email classification. Multiple models were tested to ensure optimal performance.

Activity 1: Import Required Libraries

```
import pandas as pd
import numpy as np
import re
import pickle
import os
from collections import Counter

import nltk
nltk.download('punkt')
nltk.download('punkt_tab')
nltk.download('stopwords')
nltk.download('wordnet')

nltk.download('punkt_tab')
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
from nltk.stem import WordNetLemmatizer

from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix
import seaborn as sns
import matplotlib.pyplot as plt

print("✓ All libraries imported successfully!")
print(f"Pandas version: {pd.__version__}")
print(f"Scikit-learn imported successfully")
```

Activity 2: Logistic Regression Model

```
print("=" * 70)
print("[STEP 6] TRAINING LOGISTIC REGRESSION MODEL")
print("=" * 70)

print("Initializing model...")
model = LogisticRegression(
    max_iter=1000,
    random_state=42,
    class_weight='balanced',
    C=1.0
)

print("Training model (this may take 1-2 minutes)...")
model.fit(X_train, y_train)

print("\n✓ Model Training Complete!")
print(f"  Algorithm: Logistic Regression")
print(f"  Classes: {model.classes_}")
print(f"  Iterations: {model.n_iter_[0]}")
```

Logistic Regression is a linear model for binary classification that predicts the probability of a sample belonging to a particular class.

Activity 3: Model Evaluation

```
[STEP 7] MODEL EVALUATION
================================================================================
Making predictions on test set...

================================================================================
ACCURACY: 0.9397 (93.97%)
================================================================================

================================================================================
CLASSIFICATION REPORT:
================================================================================
              precision    recall  f1-score   support

   Financial       0.77      0.92      0.84      6553
     General       0.90      0.98      0.93     21346
          HR       0.90      0.89      0.90     17842
      Urgent       1.00      0.94      0.97     57740

    accuracy                           0.94    103481
   macro avg       0.89      0.93      0.91    103481
weighted avg       0.94      0.94      0.94    103481


================================================================================
CONFUSION MATRIX:
================================================================================
✓ Confusion matrix saved as 'confusion_matrix.png'
```

Activity 4: Model Performance

Model Performance:

CATEGORY	PRECISION	RECALL	F1-SCORE
Urgent	1.00	0.94	0.97
General	0.90	0.98	0.93
PromotionHR	0.90	0.89	0.90
Financial	0.90	0.98	0.84

Overall Accuracy: ~94%

Milestone 4:Backend Development and API Integration

Activity 4.1: API Setup

FastAPI was used to create secure and efficient API endpoints. These include routes for signup, login, IMAP setup, email syncing, and classification requests.

Activity 4.2: Implementing Authentication

JSON Web Token (JWT) based authentication was implemented to secure user accounts and protect data. This ensures that only authenticated users can access their emails and sync new data.

Activity 4.3: Integrating the Machine Learning Model

The serialized TF IDF vectorizer and model were loaded into the backend. During email syncing, the backend preprocesses email text and generates the predicted category.

Activity 4.4: IMAP Email Fetching

The backend connects to Gmail using IMAP, retrieves email subjects and bodies, cleans them, and prepares them for classification. Errors such as authentication failures and connection issues are handled safely.

Milestone 5: Frontend Development

This milestone focuses on creating the user interface and integrating API responses into the dashboard.

Activity 5.1: Building the 3D Intro Page

Three.js and React Three Fiber were used to design an interactive solar system. Each planet serves as a symbolic representation of different email concepts.

Activity 5.2: Developing Login and Signup Pages

Clean and responsive interfaces were created for authentication. The Login page includes animated network effects for visual appeal.

Activity 5.3: Designing the Dashboard

The dashboard displays all categorized emails, provides filtering options, and shows email details. Users can sync new emails directly from this interface.

Activity 5.4: Connecting to Backend APIs

Axios was used to retrieve user data, fetch emails, update IMAP settings, and display classified emails in real time.

Milestone 6: Model Deployment

Activity 6.1: Save the Best Model

Save the trained model and scaler for future use:

```
============================================================
[STEP 8] SAVING MODEL AND VECTORIZER
============================================================
Saving model to 'model.pkl'...
✓ Model saved!

Saving TF-IDF vectorizer to 'tfidf.pkl'...
✓ Vectorizer saved!

✓ Files created successfully:
  model.pkl: 157.02 KB
  tfidf.pkl: 193.02 KB
```

Application Architecture

Create the Project folder which contains files as shown below

```
email_classifier/
├── backend/
│   ├── main.py          # FastAPI application & routes
│   ├── imap_handler.py  # Email fetching & IMAP logic
│   ├── model/
│   │   └── classifier.py  # ML inference engine for email classification
│   ├── requirements.txt  # Python dependencies
│   └── venv/            # Virtual environment
├── frontend/
│   ├── src/
│   │   ├── pages/
│   │   │   ├── IntroPage.js   # 3D Solar System landing page
│   │   │   ├── LoginPage.js   # Network animation login
│   │   │   ├── SignupPage.js  # User registration
│   │   │   └── Dashboard.js   # Main email dashboard
│   │   ├── App.js
│   │   └── index.js
│   ├── package.json
│   └── public/
├── model.pkl            # Trained classification model
├── tfidf.pkl            # TF-IDF vectorizer
└── README.md
```

Activity 6.2: Testing End to End Workflow

The complete workflow was tested by signing up, logging in, syncing emails, classifying them, and confirming accuracy and API responsiveness.

Activity 6.3: Final Optimization

API responses were optimized, loading animations were added, and error handling was refined to create a smooth user experience.

The application follows a standard Client-Server architecture for machine learning deployment:

- Client (Frontend): index.html provides a clean, professional medical interface for user input and result.html displays the prediction.
- Backend (Server): app.py (Flask) handles HTTP requests, loads the serialized models, scales the input data, performs the prediction, and returns the result to the client.
- Persistence Layer: LogisticRegression_model.pkl and scaler.pkl store the trained model parameters and scaling statistics.

Backend Implementation (app.py)

The Flask backend is responsible for the core ML workflow:

- Model Loading: Loads LogisticRegression_model.pkl and scaler.pkl on application startup to minimize prediction latency.
- Data Handling: Parses form data (Gender, Hemoglobin, MCH, MCHC, MCV) received via a POST request.
- Preprocessing: Applies the saved MinMaxScaler to the numerical features, ensuring the new data is treated identically to the training data.
- Prediction: Executes the logreg.predict() and logreg.predict_proba() methods on the scaled input.
- Result Interpretation: Maps the binary output (0 or 1) to "Negative for Anemia" or "Positive for Anemia," and calculates the confidence percentage.

```python
from fastapi import FastAPI, Depends, HTTPException, status
from fastapi.middleware.cors import CORSMiddleware
from fastapi.security import OAuth2PasswordBearer, OAuth2PasswordRequestForm
from motor.motor_asyncio import AsyncIOMotorClient
from jose import JWTError, jwt
from datetime import datetime, timedelta
from pydantic import BaseModel, EmailStr
from typing import Optional, List
import os
import hashlib
from dotenv import load_dotenv

load_dotenv()

# Configuration
MONGODB_URL = os.getenv("MONGODB_URL", "mongodb://localhost:27017")
DATABASE_NAME = os.getenv("DATABASE_NAME", "email_classifier")
SECRET_KEY = os.getenv("SECRET_KEY", "your-secret-key-change-in-production")
ALGORITHM = os.getenv("ALGORITHM", "HS256")
ACCESS_TOKEN_EXPIRE_MINUTES = int(os.getenv("ACCESS_TOKEN_EXPIRE_MINUTES", "30"))

# Initialize FastAPI
app = FastAPI(title="Email Classifier API", version="1.0.0")

# CORS Configuration
app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],
    allow_credentials=True,
    allow_methods=["*"],
```

```python
)

# MongoDB Client
client = AsyncIOMotorClient(MONGODB_URL)
db = client[DATABASE_NAME]

# OAuth2
oauth2_scheme = OAuth2PasswordBearer(tokenUrl="api/auth/token")

# Pydantic Models
class UserSignup(BaseModel):
    username: str
    email: EmailStr
    password: str

class UserLogin(BaseModel):
    email: EmailStr
    password: str

class Token(BaseModel):
    access_token: str
    token_type: str
    user: dict

class IMAPConfig(BaseModel):
    email: EmailStr
    password: str

class EmailResponse(BaseModel):
    id: str
    subject: str
    from_: str
    date: datetime
    body: str
    category: str
    confidence: float

# Utility Functions
def verify_password(plain_password, hashed_password):
    """Verify a password against its hash using SHA256"""
    salt = hashed_password[:32]
    stored_hash = hashed_password[32:]
    pwd_hash = hashlib.pbkdf2_hmac('sha256', plain_password.encode('utf-8'), salt, 100000)
    return pwd_hash == stored_hash

def get_password_hash(password):
    """Hash a password using SHA256 with salt"""
    salt = os.urandom(32)
    pwd_hash = hashlib.pbkdf2_hmac('sha256', password.encode('utf-8'), salt, 100000)
    return salt + pwd_hash

def create_access_token(data: dict, expires_delta: Optional[timedelta] = None):
    to_encode = data.copy()
    if expires_delta:
        expire = datetime.utcnow() + expires_delta
    else:
        expire = datetime.utcnow() + timedelta(minutes=15)
    to_encode.update({"exp": expire})
    encoded_jwt = jwt.encode(to_encode, SECRET_KEY, algorithm=ALGORITHM)
    return encoded_jwt
```

```python
    to_encode = data.copy()
    if expires_delta:
        expire = datetime.utcnow() + expires_delta
    else:
        expire = datetime.utcnow() + timedelta(minutes=15)
    to_encode.update({"exp": expire})
    encoded_jwt = jwt.encode(to_encode, SECRET_KEY, algorithm=ALGORITHM)
    return encoded_jwt

async def get_current_user(token: str = Depends(oauth2_scheme)):
    credentials_exception = HTTPException(
        status_code=status.HTTP_401_UNAUTHORIZED,
        detail="Could not validate credentials",
        headers={"WWW-Authenticate": "Bearer"},
    )
    try:
        payload = jwt.decode(token, SECRET_KEY, algorithms=[ALGORITHM])
        email: str = payload.get("sub")
        if email is None:
            raise credentials_exception
    except JWTError:
        raise credentials_exception

    user = await db.users.find_one({"email": email})
    if user is None:
        raise credentials_exception
    return user

# Routes
@app.get("/")
async def root():
    return {"message": "Email Classifier API", "version": "1.0.0"}

@app.get("/api/user/me")
async def get_current_user_info(current_user: dict = Depends(get_current_user)):
    return {
        "username": current_user.get("username"),
        "email": current_user.get("email"),
        "imap_email": current_user.get("imap_email"),
        "imap_configured": current_user.get("imap_configured", False),
        "created_at": current_user.get("created_at")
    }

@app.post("/api/auth/signup")
async def signup(user: UserSignup):
    # Check if user exists
    existing_user = await db.users.find_one({"email": user.email})
    if existing_user:
        raise HTTPException(status_code=400, detail="Email already registered")

    existing_username = await db.users.find_one({"username": user.username})
    if existing_username:
        raise HTTPException(status_code=400, detail="Username already taken")

    # Create new user
    hashed_password = get_password_hash(user.password)
    user_dict = {
        "username": user.username,
        "email": user.email,
        "password": hashed_password,
        "created_at": datetime.utcnow(),
        "imap_configured": False
```

```python

    result = await db.users.insert_one(user_dict)

    return {"message": "User created successfully", "user_id": str(result.inserted_id)}

@app.post("/api/auth/login", response_model=Token)
async def login(user: UserLogin):
    # Find user
    db_user = await db.users.find_one({"email": user.email})
    if not db_user or not verify_password(user.password, db_user["password"]):
        raise HTTPException(
            status_code=status.HTTP_401_UNAUTHORIZED,
            detail="Incorrect email or password"
        )

    # Create access token
    access_token_expires = timedelta(minutes=ACCESS_TOKEN_EXPIRE_MINUTES)
    access_token = create_access_token(
        data={"sub": user.email}, expires_delta=access_token_expires
    )

    return {
        "access_token": access_token,
        "token_type": "bearer",
        "user": {
            "username": db_user["username"],
            "email": db_user["email"]
        }
    }

@app.post("/api/imap/setup")
async def setup_imap(config: IMAPConfig, current_user: dict = Depends(get_current_user)):
    # Update user's IMAP credentials (encrypted in production)
    await db.users.update_one(
        {"_id": current_user["_id"]},
        {
            "$set": {
                "imap_email": config.email,
                "imap_password": config.password,  # Should be encrypted
                "imap_configured": True
            }
        }
    )

    return {"message": "IMAP configuration saved successfully"}

@app.post("/api/imap/sync")
async def sync_emails(current_user: dict = Depends(get_current_user)):
    if not current_user.get("imap_configured"):
        raise HTTPException(status_code=400, detail="IMAP not configured")

    from services.email_service import sync_user_emails

    try:
        synced_count = await sync_user_emails(
            current_user["_id"],
            current_user["imap_email"],
            current_user["imap_password"]
        )

        return {"message": "Emails synced successfully", "synced_count": synced_count}
    except Exception as e:
```

```python

            return {"message": "Emails synced successfully", "synced_count": synced_count}
        except Exception as e:
            raise HTTPException(status_code=500, detail=str(e))

    @app.get("/api/emails")
    async def get_emails(current_user: dict = Depends(get_current_user)):
        emails = await db.emails.find(
            {"user_id": current_user["_id"]}
        ).sort("date", -1).to_list(length=100)

        # Convert ObjectId to string
        for email in emails:
            email["_id"] = str(email["_id"])
            email["user_id"] = str(email["user_id"])

        return emails

    @app.get("/api/notifications")
    async def get_notifications(current_user: dict = Depends(get_current_user)):
        notifications = await db.notifications.find(
            {"user_id": current_user["_id"]}
        ).sort("timestamp", -1).to_list(length=50)

        for notification in notifications:
            notification["_id"] = str(notification["_id"])
            notification["user_id"] = str(notification["user_id"])

        return notifications

    if __name__ == "__main__":
        import uvicorn
        uvicorn.run(app, host="0.0.0.0", port=8000)
```

Frontend Implementation (HTML Templates)

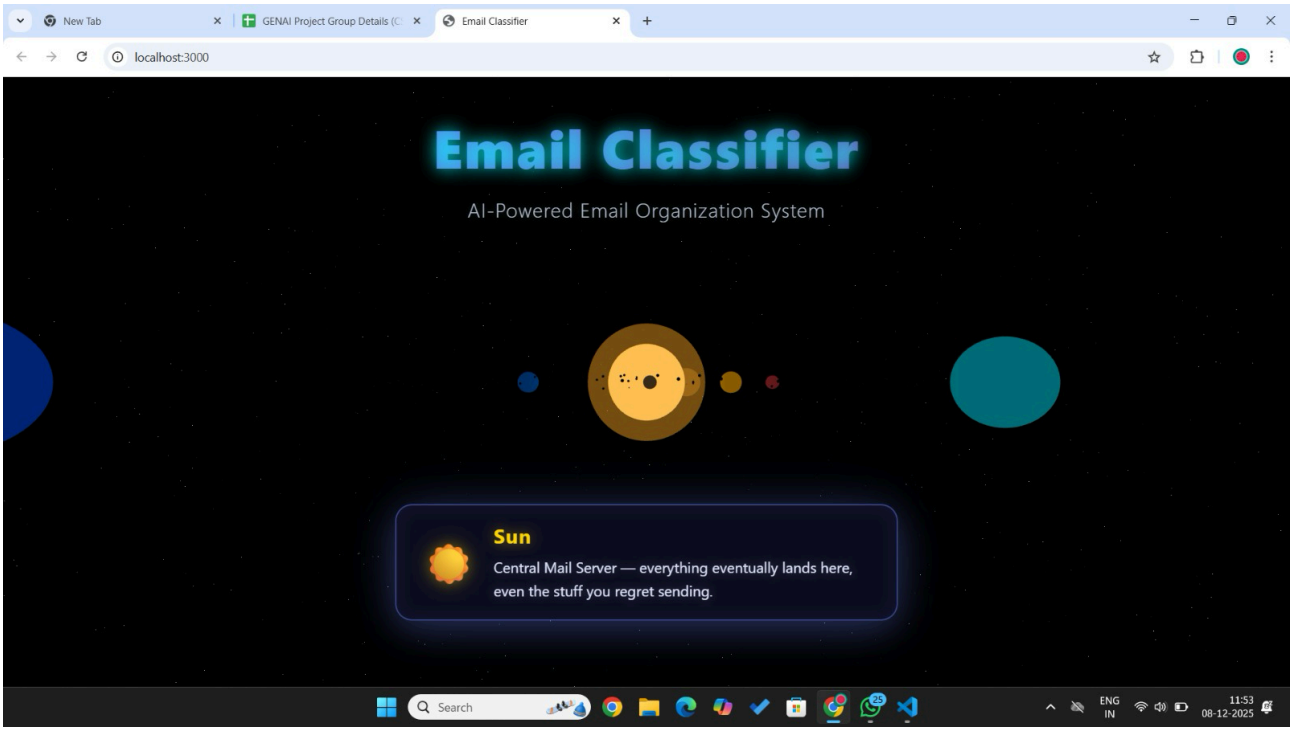
- The frontend of the Email Classifier system is built using React and provides a simple and interactive interface for users. The Intro page displays a 3D solar system using Three.js, while the Login and Signup pages offer clean forms for authentication. The Dashboard is the main working area where users can view emails, filter them by category, and sync new messages.
- The frontend communicates with the FastAPI backend through Axios. API calls are used for login, registration, IMAP setup, and retrieving classified emails. All pages are designed to be responsive, ensuring smooth usage across devices. The interface focuses on clarity, easy navigation, and quick access to categorized emails.

```jsx
import React from 'react';
import { BrowserRouter as Router, Routes, Route, Navigate } from 'react-router-dom';
import { ToastContainer } from 'react-toastify';
import 'react-toastify/dist/ReactToastify.css';
import IntroPage from './pages/IntroPage';
import LoginPage from './pages/LoginPage';
import Dashboard from './pages/Dashboard';
import './App.css';

function App() {
  return (
    <Router>
      <div className="App">
        <Routes>
          <Route path="/" element={<IntroPage />} />
          <Route path="/login" element={<LoginPage />} />
          <Route path="/dashboard" element={<Dashboard />} />
          <Route path="*" element={<Navigate to="/" replace />} />
        </Routes>
        <ToastContainer
          position="top-right"
          autoClose={3000}
          hideProgressBar={false}
          newestOnTop={false}
          closeOnClick
          rtl={false}
          pauseOnFocusLoss
          draggable
          pauseOnHover
          theme="dark"
        />
      </div>
    </Router>
```

Application Screenshots and Workflow

The workflow moves sequentially from data input to clinical risk assessment.

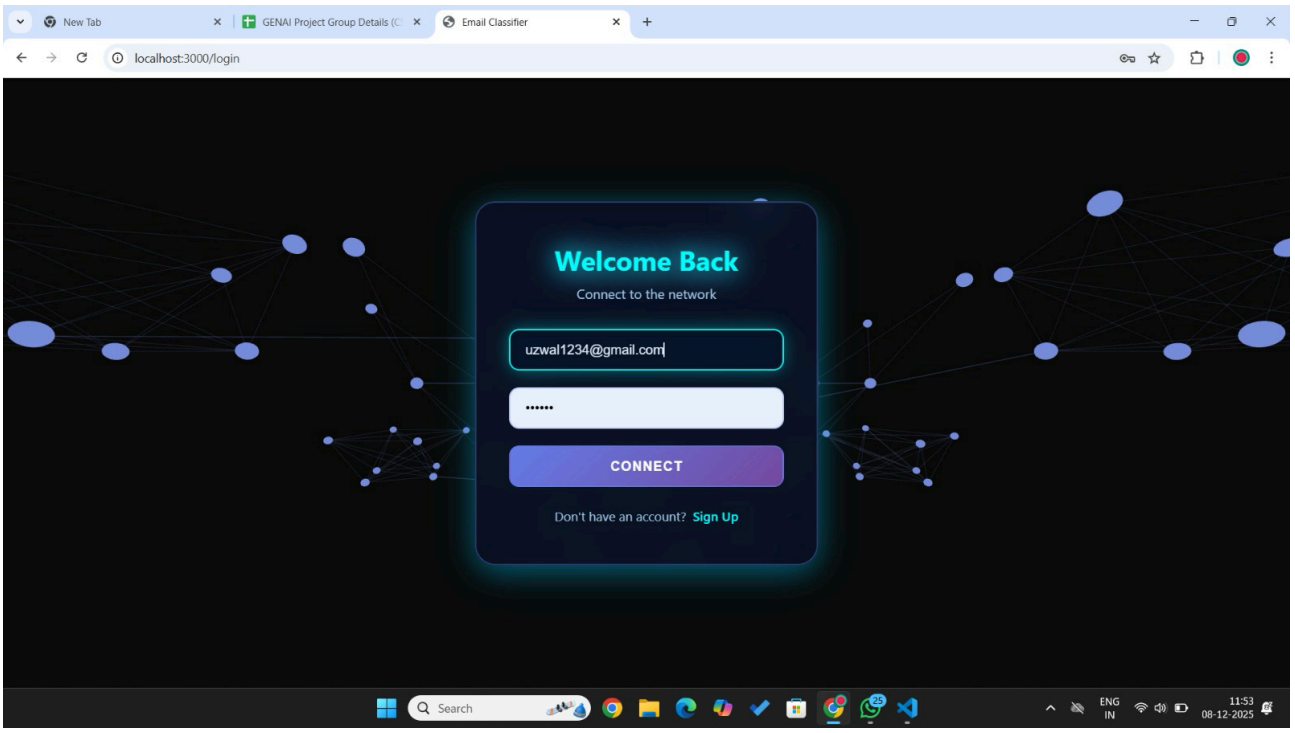
1. **Landing / Intro Page**
    - Caption: Intro page with 3D solar system
    - Purpose: Provides the initial visual entry point and branding for the application. The interactive solar system helps users navigate to the login page.
    - Screenshot should show: Full page view of the 3D solar system scene, navigation controls, and a visible call to action to proceed to login.
    - Suggested filename: intropage_solar_system.png



2. Signup Page/ Login Page

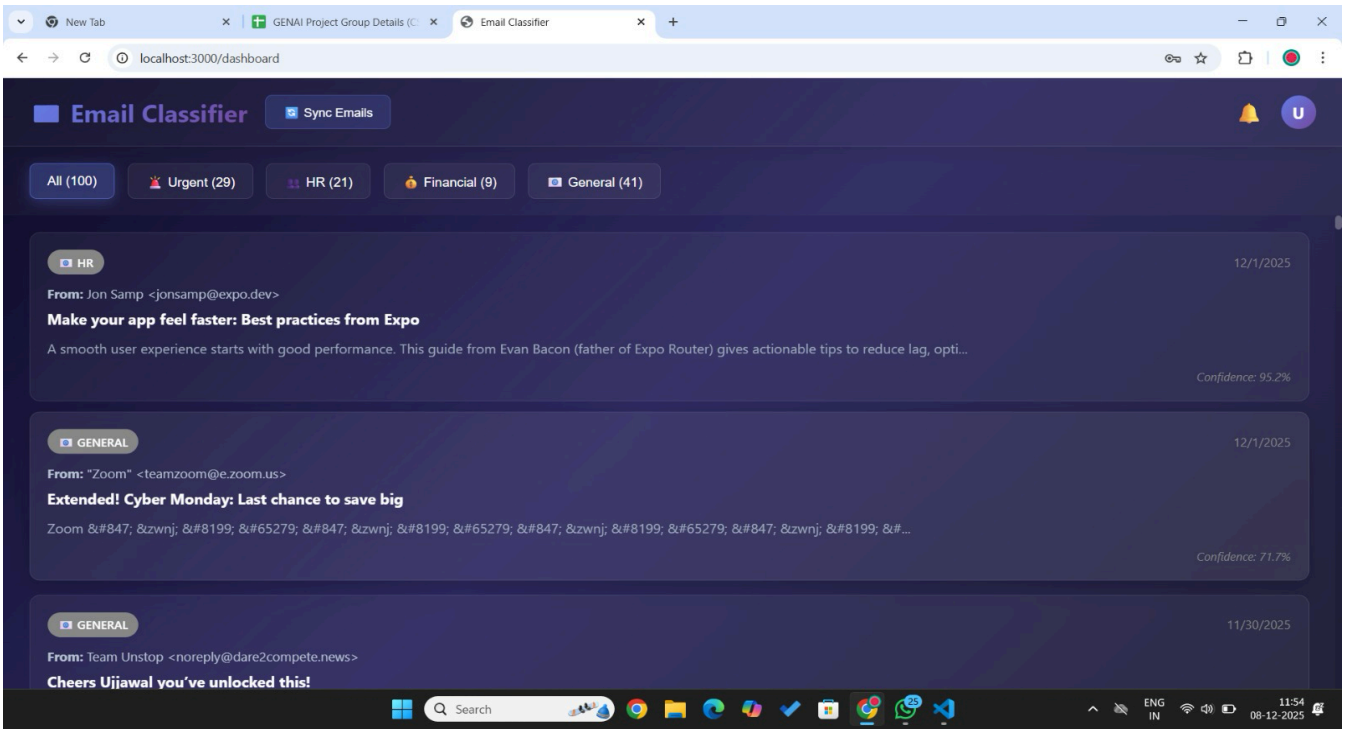
Caption: User registration form
Purpose: Allows a new user to create an account. This screen collects basic user information required for authentication.
Screenshot should show: The signup form fields (name, email, password), submit button, and any inline validation messages.
Suggested filename: signup_page.png

Caption: Login screen with animated background
Purpose: Enables existing users to authenticate and access their dashboard. The page also links to account recovery and signup.
Screenshot should show: Email and password fields, login button, and visual animation background.
Suggested filename: login_page.png



3. Dashboard - Overview

- Caption: Main dashboard showing category counts and quick actions
  Purpose: Central workspace that summarizes categorized email counts and provides quick actions such as sync and filter.
  Screenshot should show: Category cards with counts (Urgent, HR, Financial, General), Sync Emails button, and navigation menu.
  Suggested filename: dashboard_overview.png

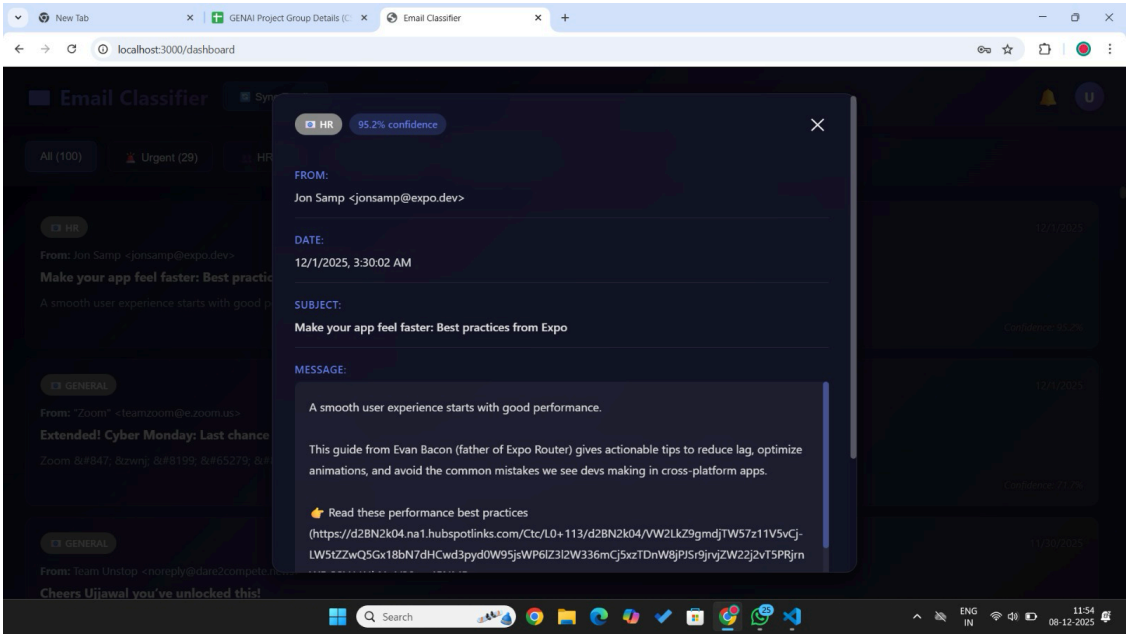


4. Gmail Content

The Gmail content module is responsible for retrieving and processing emails directly from the user's Gmail account. The application connects to Gmail through the IMAP protocol, which allows secure access to messages without modifying the mailbox. Once connected, the system extracts essential components of each email such as:

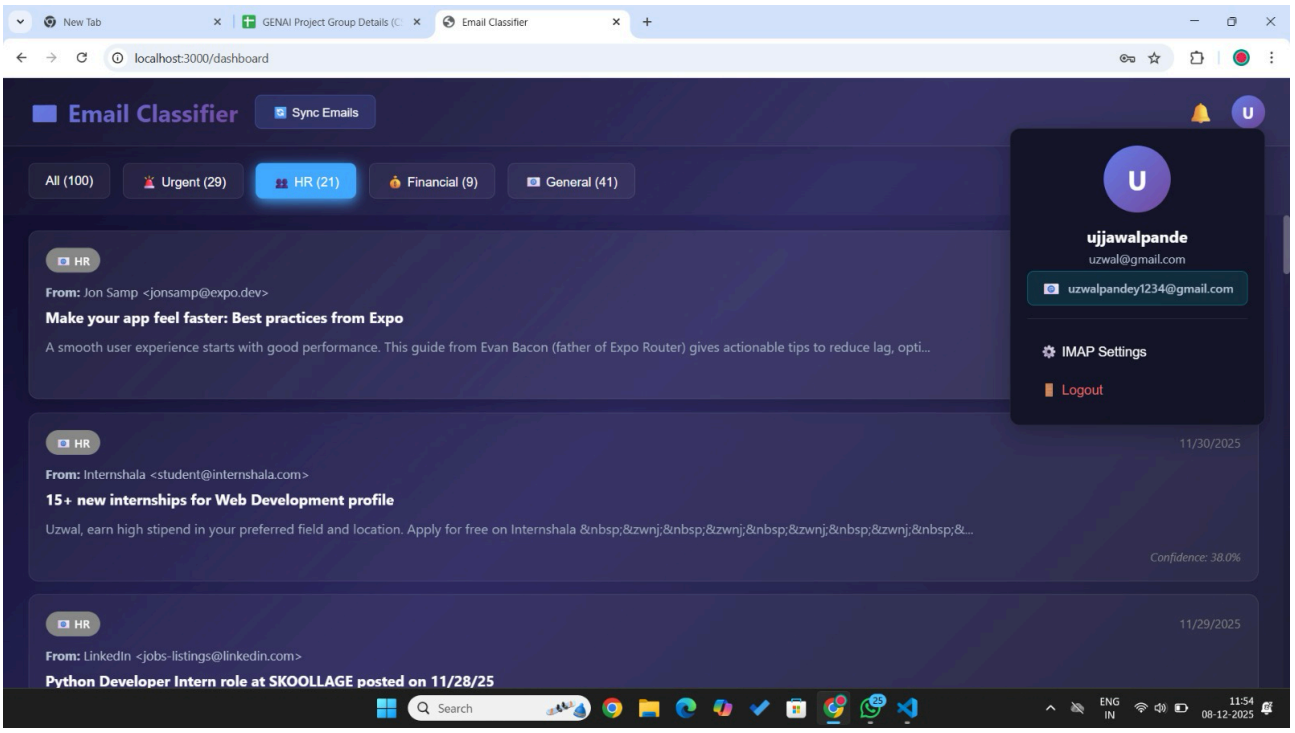
- Sender address
- Recipient address
- Subject line
- Email body text
- Date and time of the message

The extracted content is cleaned and formatted before being sent to the machine learning pipeline. Only the subject and body are used for classification, while metadata such as sender and timestamp are stored for display on the dashboard. This section ensures that all emails required for analysis are collected accurately and converted into a readable format for further processing.



5. Account Setting

This module connects to Gmail using IMAP and retrieves email data such as sender details, subject, body, and date. The text content is cleaned and passed to the machine learning model for classification, while metadata is stored for user display.



Future Implementations

Future enhancements for the Email Classifier system focus on improving accuracy, user experience, and real world usability. Possible extensions include:

- **Gmail OAuth Integration:** Replace app passwords with OAuth2 for more secure and seamless email access.
- **Advanced Classification Models:** Implement deep learning based text classifiers to improve prediction accuracy across categories.
- **Real Time Email Monitoring:** Automatically classify emails as soon as they arrive in the inbox without manual syncing.
- **Email Summarization:** Generate short summaries to help users quickly understand long emails.
- **Spam and Phishing Detection:** Add an additional classifier to identify harmful or suspicious emails.
- **Mobile Application:** Develop a mobile app version for easier access and notification based updates.
- **Auto Reply Suggestions:** Use natural language techniques to offer reply recommendations for certain email types.

Conclusion

The Email Classifier project successfully demonstrates the integration of machine learning with a modern full stack application. By combining a trained text classification model, a FastAPI backend, and a React based frontend, the system is able to automatically categorize emails into meaningful groups such as Urgent, HR, Financial, and General. The addition of an interactive 3D interface enhances user engagement while maintaining functional clarity.

The project achieves its objective of reducing manual effort in email management by providing accurate predictions, secure authentication, and seamless communication with Gmail through IMAP. With its structured architecture and reliable performance, the system serves as a strong foundation for future enhancements, including advanced classification methods, mobile deployment, and real time monitoring. Overall, the project demonstrates an effective end to end solution for intelligent email organization.