Sentiment Analysis Pipeline API & Training

This project provides a sentiment analysis pipeline designed with separate components for model training and serving via a Flask API. It demonstrates a basic approach to deploying a machine learning model, focusing on modularity, configuration, and model persistence.

Overview

This project implements a sentiment classifier capable of categorizing text into 'Positive', 'Negative', or 'Neutral'. It is structured into two main parts: a `training_code` module responsible for data generation (synthetic), preprocessing, model training, and saving; and a `sys` module containing a lightweight API to load the trained model and provide predictions for new text inputs.

- **`Sentiment_analysis_api/`**: Contains the code necessary to run the sentiment analysis as a service (API). `app.py` is the entry point for the web application.
- **`training_Script/`**: Contains all scripts related to preparing data, training the machine learning model, evaluating it, and saving the trained artifacts. `train.py` is the main execution script for training. `utils.py` likely contains shared functions like text preprocessing and the `SentimentPredictor` class used by both training and API.
- **`models/`**: This directory serves as the bridge between the training and API components. The training script saves the vectorizer, label encoder, and model here, and the API script loads them from here.
- **`logs/`**: Intended for storing application and training logs.

Machine Learning Pipeline Details

The underlying machine learning pipeline performs the following steps:

1. **Data Handling:** Uses `pandas` DataFrames. Currently generates synthetic data, but structured to handle external data sources.
2. **Text Preprocessing:** Standard NLP cleaning (lowercasing, punctuation removal, tokenization, stop word removal, lemmatization) using NLTK. This logic should ideally be encapsulated in a function within `training_code/utils.py` and used consistently by both `train.py` and `app.py`.

3. **Feature Extraction:** Converts preprocessed text into numerical vectors using TF-IDF (Term Frequency-Inverse Document Frequency) from `sklearn.feature_extraction.text`. Includes support for n-grams.
4. **Model:** Employs an `xgboost.XGBClassifier` for classification.
5. **Training & Tuning:** Uses `sklearn.model_selection.GridSearchCV` to find optimal XGBoost hyperparameters on the TF-IDF features.
6. **Evaluation:** Assesses performance using `sklearn.metrics` (Accuracy, Classification Report).
7. **Persistence:** Saves the trained `TfidfVectorizer`, `LabelEncoder`, and `XGBClassifier` objects using `joblib` to the `models/` directory.

For a detailed justification of the chosen model and features, as well as assumptions and limitations of this pipeline, please refer to the dedicated sections below.

Prerequisites

- Python
- Libraries listed in `requirement.txt`
- NLTK data (`stopwords`, `punkt`, `wordnet`).

How to Use

Using the sentiment analysis pipeline involves two main steps: training the model and then running the API to make predictions.

Step 1: Training the Model

Navigate to the `training_Script/` directory and run the train.py training script. This script will generate synthetic data, preprocess it, train the model (including tuning), evaluate it, and save the necessary artifacts to the `models/` directory.

Step 2: Running the API

Navigate to the `Sentiment_analysis_api/` directory and run the Flask application script. This will start a local web server hosting the prediction endpoint.

ed differently in `app.py`). It will print a message indicating the server is running.

Step 3: Making Predictions via API

Once the API is running, you can send HTTP POST requests to the `/predict` endpoint with the text you want to analyze.

You can use tools like `curl`, Postman, or any programming language's HTTP library. Here's an example using `curl`:

Bash

curl -X POST http://localhost:5000/predict \

-H "Content-Type: application/json" \

-d '{"text": "I hate this service, it was terrible!"}'

Example Expected Output:

JSON

{

"predicted_sentiment": "Negative",

"confidence_score": 0.8649

}

The exact `confidence_score` will depend on the model's output probabilities and how it's calculated in `app.py`.

API Endpoint Description

- **Endpoint:** `/predict`
- **Method:** `POST`
- **Request Body:**
  - Content Type: `application/json`

  Structure: A JSON object with a single key `text` containing the string to analyze.
  JSON
  {

  "text": "Your input text goes here."

  }

  -
- **Response Body:**
  - Content Type: `application/json`

  Structure: A JSON object containing the predicted sentiment and a confidence score.
  JSON
  {

"predicted_sentiment": "Positive" | "Negative" | "Neutral",

"confidence_score": float # Probability or score associated with the prediction

}

- **Error Responses:** The API should ideally include error handling for invalid input formats (e.g., non-JSON body, missing `text` key). Currently, basic error handling might be present but could be expanded.

Classification Report

	precision	recall	f1-score	support
Negative	1.00	0.99	0.99	175
Neutral	0.98	1.00	0.99	172
Positive	1.00	0.99	1.00	153
accuracy			0.99	500
macro avg	0.99	0.99	0.99	500
weighted avg	0.99	0.99	0.99	500

Model and Feature Justification

- **Model (XGBoost):** Chosen for its balance of performance, robustness, regularization capabilities, and relative speed compared to more complex models. It's a strong model for structured data like TF-IDF vectors.
- **Features (TF-IDF with N-grams):** A standard and effective text vectorization method that captures word importance based on frequency and rarity. Including n-grams helps capture context and phrases important for sentiment.

For a more detailed explanation of the choice of XGBoost and TF-IDF, their benefits, and alternatives, please refer to the relevant section in the full documentation (or the previous README content).

Assumptions and Limitations

- The project assumes the input text is primarily English.
- It relies on synthetic data generation in `train.py` for demonstration. Real-world performance requires training on a diverse, real-world dataset.
- The text preprocessing is basic and may not handle complex real-world text noise effectively (emojis, URLs, slang, etc.).
- The model's vocabulary is limited by the training data.
- The design does not inherently support multi-language sentiment analysis.

Future Enhancements

Potential areas for improvement include:

- Integrating with real-world data loading instead of synthetic generation.
- Exploring transformer models (BERT, etc.) for potentially higher accuracy on large datasets.
- Adding more sophisticated API features (e.g., batch processing, request validation, rate limiting).
- Containerizing the application using Docker for easier deployment.
- Setting up continuous integration/continuous deployment (CI/CD) pipelines for training and deployment.
- Implementing model versioning and experiment tracking.