Hyperspectral Imaging DON Prediction Pipeline

This project implements a machine learning pipeline for predicting deoxynivalenol (DON) concentration in corn samples using hyperspectral imaging data. The primary focus of this project was on structuring and modularizing the pipeline to ensure clear separation of concerns, maintainability, and extensibility, rather than on maximizing predictive performance. The pipeline supports multiple models – including XGBoost, Fully Connected Network (FCN), Convolutional Neural Network (CNN), and an experimental Transformer – with facilities for data preprocessing, model training, evaluation, and deployment via API.

Directory Structure

```
hyperspectral_regression/
├── src/
│   ├── data/
│   │   ├── preprocess.py         # Data preprocessing functions,
imputation, scaling, and optional PCA-based feature selection.
│   │   ├── dataset_loader.py     # Functions to load and
(optionally) normalize the hyperspectral dataset.
│   │
│   ├── models/
│   │   ├── xgboost_model.py      # XGBoost regressor
implementation.
│   │   ├── fcn_model.py          # Fully Connected Network (FCN)
implementation using PyTorch.
│   │   ├── cnn_model.py          # 1D Convolutional Neural Network
(CNN) implementation using PyTorch.
│   │   ├── transformer_model.py  # Experimental Transformer-based
model for regression (debugging in progress).
│   │   ├── base_model.py         # Abstract base class to enforce a
standard interface for all models.
│   │
│   ├── training/
│   │   ├── train.py              # Training pipeline, orchestrates
training of all models and saving of best performers.
│   │   ├── optimizer.py          # (Not yet integrated)
Hyperparameter optimization ideas using Optuna.
│   │
│   ├── evaluation/
```

```
|   |   ├── evaluate.py           # Evaluation metrics (MAE, RMSE,
R²) and residual plotting functions.
|   |
|   ├── deployment/
|   |   ├── predictor.py          # Model inference; abstracts
prediction across different model types.
|   |   ├── api.py                # FastAPI-based API to expose
predictions.
|   |
|   ├── utils/
|   |   ├── logger.py             # Logger configuration and setup.
|   |   ├── helper.py             # Helper functions (e.g., choosing
and saving the best model based on R² score).
|   |   ├── load_config.py        # Loads project configuration from
config.yaml.
|
|── notebooks/
|   ├── exploratory_analysis.ipynb          # Initial data
exploration.
|   ├── feature_selection_experiments.ipynb   # Experiments with
feature selection techniques.
|   ├── model_benchmarking.ipynb              # Comparative analysis
of the various models.
|
|── tests/
|   ├── test_data.py            # Unit tests for the data pipeline.
|   ├── test_models.py          # Unit tests for the machine
learning models.
|   ├── test_api.py             # Tests for the API endpoints.
|
|── requirements.txt            # Project dependencies.
|── config.yaml                 # Global configuration settings for
data, model parameters, preprocessing, training, and deployment.
|── docker-compose.yaml         # Docker Compose configuration for
containerizing training and API services.
|── Dockerfile.train            # Dockerfile for the training
pipeline.
|── Dockerfile.api              # Dockerfile for the API service.
|── run.py                      # Main entry point to execute the
pipeline.
```

```
|—— README.md                    # Overview and instructions on
running the project.
```

Pipeline Components and Workflow

1. Data Handling and Preprocessing

- **Dataset Loading:**
  `src/data/dataset_loader.py` handles reading the hyperspectral CSV file, setting the appropriate index, and splitting the data into features (`X`) and target (`y`).
- **Preprocessing:**
  `src/data/preprocess.py` applies missing value imputation (using a configurable strategy), optional standard scaling, and (optionally) PCA-based feature selection. These steps ensure the data is in a suitable format for model training.

2. Model Implementations

The project implements multiple models to provide a comparative study:

- **XGBoost:**
  Implemented in `src/models/xgboost_model.py`, this model uses the XGBoost library to perform regression.
- **FCN (Fully Connected Network):**
  Defined in `src/models/fcn_model.py` with an underlying MLP architecture, the FCN is built with PyTorch and provides a modular approach to regression on spectral data.
- **CNN (Convolutional Neural Network):**
  Implemented in `src/models/cnn_model.py`, the 1D CNN applies convolutional layers on the spectral data after reshaping it accordingly.
- **Transformer (Experimental):**
  In `src/models/transformer_model.py`, an experimental Transformer-based regressor is implemented. However, due to architectural debugging challenges, its integration into the main training pipeline was not completed.

Each model class inherits from the abstract `BaseModel` class (`src/models/base_model.py`), ensuring consistency in the interface (methods for training, prediction, saving, and loading).

3. Training Pipeline

- **Training Execution:**
  The primary training orchestration is done in `src/training/train.py`, which:
  - Splits the dataset into training and validation sets.
  - Trains the FCN, XGBoost, and CNN models sequentially (the Transformer model code is present but commented out).

    - Saves the trained models to designated paths in the configuration.
- **Hyperparameter Optimization (Not Integrated):**
  The file `src/training/optimizer.py` contains ideas for using Optuna for hyperparameter tuning. Due to time constraints, this component was not integrated into the main pipeline.

4. Model Evaluation and Best Model Selection

- **Evaluation Metrics and Plots:**
  In `src/evaluation/evaluate.py`, functions calculate key regression metrics—Mean Absolute Error (MAE), Root Mean Squared Error (RMSE), and R^2 score—and generate scatter and residual plots to visualize performance.

- **Best Model Selection:**
  After evaluation, the `run.py` file calls a helper function (`choose_and_save_best_model` in `src/utils/helper.py`) to select the best-performing model based on the R^2 score.
    - **Why R^2?**
      The R^2 score measures the proportion of variance in the dependent variable that is predictable from the independent variables. A higher R^2 indicates a model that better explains the variability of the response data, making it a robust metric for comparing regression models.
    - **Process:**
      The evaluation results from each model are collected, sorted by the R^2 score, and the model with the highest R^2 is deemed the best. The chosen model is then copied to a standardized path (e.g., `models/best_model.pkl` or `models/best_model.pt`) for deployment, ensuring a consistent inference workflow.

5. Deployment and API

- **Inference Wrapper:**
  `src/deployment/predictor.py` abstracts the model inference process across different model types. It loads the saved model and provides a unified interface for predictions.
- **API Endpoint:**
  The API in `src/deployment/api.py` uses FastAPI to create a web service that accepts new samples and returns predicted DON concentration. The API loads the “best model” based on evaluation results and exposes endpoints for status checking and prediction.

6. Utilities

- **Configuration Management:**
  `src/utils/load_config.py` loads project settings from `config.yaml`.

- **Logging:**
  `src/utils/logger.py` provides a standardized logger across modules.
- **Helper Functions:**
  `src/utils/helper.py` includes functionality to compare models based on evaluation metrics, choose the best model using the R^2 score, and copy the best-performing model for deployment.

7. Containerization and Running the Pipeline

- **Docker Integration:**
  Two Dockerfiles are provided:
  - `Dockerfile.train` for setting up the training environment.
  - `Dockerfile.api` for deploying the API.
- **Docker Compose:**
  `docker-compose.yaml` defines services for both model training and API deployment, ensuring reproducibility and scalability.
- **Entry Point:**
  `run.py` orchestrates the complete pipeline—from data loading and preprocessing to training, evaluation, best model selection, and model deployment.

Design Decisions and Focus

Emphasis on Structure Over Performance

The project was developed with a primary focus on creating a well-organized, modular pipeline rather than on squeezing out the best possible model performance. This decision allowed for easier extension and maintenance while laying the groundwork for a production-ready system. The clear separation into data handling, model training, evaluation, and deployment modules ensures that future improvements can be implemented with minimal disruption.

Future Improvements and Ideas

1. **Enhanced API Architecture:**
   - **Proposed Improvement:**
     Instead of a single API entry point, a more robust architecture would involve a cluster of APIs. For instance, an internal `train_api` could manage model training and selection, and a separate internal prediction API could be called by the public API endpoint. This separation would improve security, scalability, and maintainability.
   - **Benefits:**
     Such a design would allow for specialized endpoints that can be secured internally, better handling of load, and isolated updates to training or inference services without affecting the other.

2. **Integration of Hyperparameter Optimization:**
    - **Proposed Improvement:**
      While the current implementation sets hyperparameters via configuration files, integrating an automated optimizer (e.g., using Optuna as outlined in `src/training/optimizer.py`) could help fine-tune the models. This would likely enhance model performance by automatically searching for the optimal hyperparameter configurations.
    - **Benefits:**
      An integrated optimization module would streamline experiments and improve the accuracy of predictions with minimal manual intervention.
3. **Development of the Transformer Model:**
    - **Proposed Improvement:**
      An experimental Transformer model was attempted to capture complex patterns in spectral data. However, due to architectural debugging challenges and time constraints, full integration was not achieved.
    - **Benefits:**
      Once debugged and optimized, the Transformer could potentially capture nonlinear relationships in the data more effectively, offering another avenue for performance improvement in complex scenarios.