CO2 Prediction Pipeline and ML Analysis

A journey from local training to a reusable analysis package.

Table of Contents

1. **Local Pipeline Execution**: Running an end-to-end ML workflow locally.
2. **Model Evaluation**: Understanding model performance.
3. **Inference**: Using the trained model for predictions.
4. **Bias & Explainability**: Analyzing model fairness and behavior.
5. **Data Drift**: Detecting changes in production data.
6. **Code Packaging**: Creating a reusable analysis library.

1. Local Pipeline Execution

This section simulates a cloud-based workflow on a local machine, encapsulating the entire process in a single script.

- **Script Creation**: A `local_pipeline.py` script is created to define the end-to-end process.
- **Execution**: The script is run from the command line, automating all subsequent steps.

Inside `local_pipeline.py`

The script performs a sequence of automated tasks:

1. **Load Data**: Reads `co2_data.csv`.
2. **Feature Engineering**: Creates lag, rolling window, and time-based features.
3. **Data Split**: Divides data into train, validation, and test sets chronologically.
4. **Hyperparameter Tuning**: Uses `RandomizedSearchCV` with XGBoost to find the best model parameters.
5. **Train & Evaluate**: Trains the best model and calculates RMSE and R^2 on the test set.
6. **Save Artifacts**: Saves the final model (`model.joblib`) and performance metrics (`evaluation.json`).

2. Model Evaluation

After the local pipeline runs, we assess the model's performance using the generated `evaluation.json` report.

- **RMSE (Root Mean Squared Error)**:
  - Measures the average magnitude of the prediction errors.
  - *Lower is better.*
- **R^2 Score (Coefficient of Determination)**:
  - Represents the proportion of variance in the CO_2 level that is predictable from the features.
  - *Closer to 1 is better.*

3. Predicting with the Trained Model

This section demonstrates how to use the saved model for inference on new data.

1. **Load Model**: The `model.joblib` artifact is loaded into the environment.
2. **Prepare New Data**: A sample of new data is loaded.
3. **Apply Feature Engineering**: The *exact same* feature engineering steps from training are applied to the new data. This is a critical step for consistency.
4. **Predict**: The model's `.predict()` method is called on the processed new data to generate CO2 predictions.

4. Bias and Explainability Analysis

We use open-source libraries to ensure our model is fair and interpretable.

- **Bias Analysis**:
  - **Goal**: Check if the model performs differently for various subgroups.
  - **Method**: Group data by a sensitive attribute (e.g., `Occupancy`) and compare the average prediction error across groups.
- **Explainability Analysis (SHAP)**:
  - **Goal**: Understand *why* the model makes its predictions.
  - **Method**: Use the SHAP library to calculate feature contributions for each prediction.

SHAP: Visualizing Model Explanations

- **SHAP Summary Plot**:
  - Provides a high-level view of global feature importance.
  - Shows which features have the most impact on predictions across the entire dataset.
- **SHAP Dependence Plot**:
  - Illustrates how a single feature's value affects the model's output.
  - Helps uncover complex relationships (e.g., non-linear effects).

5. Data Drift Analysis

This section focuses on detecting if the live data your model sees in production has changed compared to the data it was trained on.

- **Baseline vs. Current**: A baseline dataset (training data) is compared against a current dataset (live production data).
- **Statistical Tests**:
  - **Numerical Features**: The **Kolmogorov-Smirnov (KS) test** compares the distributions.
  - **Categorical Features**: The **Chi-squared test** compares the frequency of categories.
- **Drift Detection**: A low p-value (e.g., < 0.05) from these tests indicates significant drift.

Visualizing Data Drift

When drift is detected for a feature, it's crucial to visualize it.

- **Histograms**: For numerical features, a histogram overlay shows how the distribution has shifted between the baseline and current data.
- **Bar Plots**: For categorical features, a bar plot comparison shows changes in the proportions of each category.

These plots provide clear, actionable evidence of data drift.

6. Packaging the Analysis Code

To promote reusability and maintainability, the analysis code is packaged into a standard Python library.

- **Goal**: Move from ad-hoc notebook cells to a structured, installable package.
- **Benefits**:
  - **Reusability**: Easily run the same analysis on different models or datasets.
  - **Collaboration**: Share the package with team members.
  - **Automation**: Integrate the analysis into automated CI/CD or MLOps pipelines.

The Packaging Process

1. **Refactor into Functions**: The logic for bias, explainability, and drift analysis is organized into clean, well-documented functions.
2. **Create Modules**: These functions are saved into Python files (e.g., `analysis.py`, `reporting.py`) inside a package directory.
3. **Define `setup.py`**: A `setup.py` file is created to define the package's metadata, such as its name, version, and dependencies.
4. **Build the Package**: The `setup.py` script is used to build distributable files (`.tar.gz` and `.whl`).
5. **Install and Test**: The package is installed locally using `pip`, and its functions are imported and tested to ensure everything works correctly.

Conclusion

This workflow demonstrates a complete machine learning lifecycle:

- **Local Development**: Rapidly prototype and train a model locally.
- **Rigorous Analysis**: Evaluate the model for performance, bias, and explainability.
- **Production Readiness**: Monitor for data drift and package code for reusability.

This structured approach ensures that models are not only accurate but also robust, fair, and maintainable over time.