Overall Summary

This script defines a complete machine learning pipeline to predict solar panel efficiency. It reads raw data, performs extensive **feature engineering** by creating dozens of new, domain-specific variables, selects the most important features, tunes a **CatBoost Regressor** model, trains it, makes predictions, and saves the results. The pipeline is designed to be robust, including steps for handling outliers, post-processing predictions, and saving artifacts like the trained model and feature importance plots.

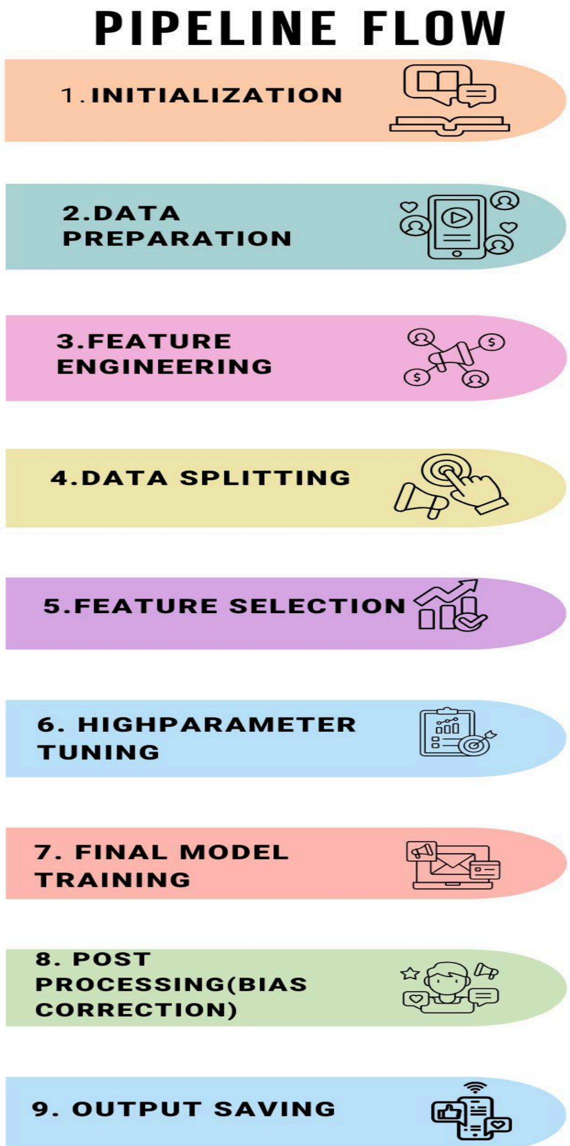
Directory Structure:

📁 Project Directory Overview

1. 📄 Text.pdf
   - Contains details of the approach, feature engineering steps, and tools used.
2. 📄 Main.ipynb
   - The main Jupyter Notebook containing the full source code for the project.
3. 📄 config.yaml
   - Configuration file specifying paths to:
     - train.csv
     - test.csv
     - Output directory for storing predictions
4. 📁 Submission/
   - Output directory containing:
     - prediction_adjusted_catboost_feature.csv – The final prediction file generated after adjustments using CatBoost features.
5. 📁 .ipynb_checkpoints/
   - Automatically created by Jupyter; stores notebook checkpoints.

6. 📁 catboost_info/
    - contains logs and metadata from CatBoost training.
7. 📄 test.csv and 📄 train.csv
    - Dataset files used for training and testing.

Pipeline Flow:



1. **Initialization**: Creates output directory.
2. **Data Preparation**: Loads and preprocesses training/test data.
3. **Feature Engineering**: Applies basic, advanced, and time-series transformations.
4. **Data Splitting**: Divides training data for validation.
5. **Feature Selection**: Retains only significant features.

6. **Hyperparameter Tuning**: Finds optimal CatBoost settings.
7. **Final Model Training**: Trains on the full dataset using the best parameters.
8. **Post-Processing (Bias Correction)**:
   - Trains `LinearRegression` on residuals.
   - Adjusts predictions using predicted errors.
   - Applies `np.clip()` to ensure outputs lie within the valid [0, 1] range.
9. **Output Saving**: Saves predictions, model, and visualizations.

Code Structure and Section-wise Explanation

The code is organized into a series of functions that handle specific tasks, orchestrated by a main execution block.

1. Preamble and Imports

```python
Python
#!/media/ayon1901/SERVER1/Zelestra/.venv/bin/python
# -*- coding: utf-8 -*-

"""
Main script that orchestrates the solar panel efficiency prediction pipeline.
"""

# Standard library imports
import logging
import time
import yaml
import os
import matplotlib.pyplot as plt

# Third-party imports
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
```

```python
from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error
from catboost import CatBoostRegressor
from sklearn.linear_model import LinearRegression
```

What it does

This section sets up the script's environment.

How it works & Details

- `#!/.../python`: This is a "shebang." It tells the operating system to execute this file using the specified Python interpreter.
- `# -*- coding: utf-8 -*-`: This declaration ensures the script can handle Unicode characters (like comments or data in various languages).
- **Docstring**: The `"""..."""` block provides a high-level description of the script's purpose.
- **Standard library imports**:
  - `logging`: To log information, warnings, and errors during execution.
  - `time`: To timestamp output files and measure execution time.
  - `yaml`: To load configuration settings from an external file (`config.yaml`).
  - `os`: To interact with the operating system, mainly for creating directories and building file paths.
  - `matplotlib.pyplot`: To create and save plots (specifically for feature importance).
- **Third-party imports**:
  - `numpy as np`: A fundamental library for numerical operations, especially array manipulations.
  - `pandas as pd`: The primary library for data manipulation using DataFrames.
  - `sklearn`: A comprehensive machine learning library.
    - `train_test_split`: To split data into training and validation sets.
    - `metrics`: To evaluate the model's performance (MSE, R^2, MAE).
    - `LinearRegression`: Used for a clever post-processing step to correct model bias.
  - `catboost.CatBoostRegressor`: The core machine learning model used for prediction. It's a gradient boosting algorithm known for its performance and handling of categorical features.

2. Configuration and Setup Functions

These are helper functions for setting up the pipeline's environment.

`load_config()`

- **What:** Loads settings from `config.yaml`. This is good practice as it separates configuration (like file paths or model parameters) from the code.

- **How:** It opens the config.yaml file and uses yaml.safe_load() to parse the YAML content into a Python dictionary.

create_output_dir(base_dir)

- **What:** Creates a unique directory to store the output of a single pipeline run (predictions, model files, plots).
- **How:** It generates a timestamp string (e.g., 20250607-133000) using time.strftime(). It then joins this timestamp with the base directory path from the config to create a unique folder name. os.makedirs(exist_ok=True) creates the directory and doesn't raise an error if it already exists.

3. Feature Engineering Functions

This is the most complex part of the script, where raw data is transformed into meaningful features for the model.

add_basic_features(df)

- **What:** Creates a first set of domain-specific features related to solar panel physics and environmental conditions.
- **How & Details:**
    1. **Data Cleaning:** It first ensures that key columns are numeric using pd.to_numeric(errors='coerce') (which turns invalid values into NaN). It then fills any missing numerical values with the mean of their respective columns.
    2. **Irradiance & Soiling:** It calculates effective_irradiance (sunlight reaching the panel after being blocked by dirt) and adds squared/square root terms to help the model capture non-linear relationships.
    3. **Aging:** It models the panel's degradation over time. maintenance_frequency and maintenance_effectiveness attempt to quantify the impact of maintenance relative to the panel's age.
    4. **Power Quality:** It calculates power (P=V×I). It then derives features like power_efficiency (P/Irradiance) to normalize the power output by the available sunlight.
    5. **Temperature:** It creates a temp_efficiency_factor using a standard industry coefficient (-0.4% efficiency loss per °C above 25°C). It also calculates the difference between the module and ambient temperature (temp_difference), which is a key indicator of panel health.
    6. **Environment:** It combines several weather metrics (humidity, cloud_coverage, etc.) into a single weather_impact score. The weights (-0.3, -0.5, etc.) are chosen based on domain knowledge about how each factor affects efficiency.
    7. **Panel Health:** It creates an overall_health score by combining factors like temperature difference, soiling, age, and maintenance frequency into a single, comprehensive metric.

8. **Time-based:** It extracts time components like `hour` and `day_of_year`. Crucially, it calculates `solar_elevation` and `solar_azimuth` using trigonometric functions to model the sun's position in the sky, which directly impacts the amount of incident light.

`add_advanced_features(df)`

- **What:** Builds upon the basic features by adding more complex mathematical transformations.
- **How & Details:**
  - **Polynomial Features:** It adds squared (`**2`), cubed (`**3`), and square root (`sqrt`) versions of key columns. This allows the linear-based components of the model to capture highly curved, non-linear relationships.
  - **Interaction Terms:** It multiplies key features together (e.g., `irradiance * temperature`). This helps the model learn how the effect of one feature changes based on the value of another.
  - **Log Transformations:** It applies `np.log1p` (which calculates log(1+x)) to skewed features. This can help normalize their distribution and reduce the influence of very high values. `log1p` is used instead of `log` to gracefully handle values of 0.

`add_time_series_features(df)`

- **What:** Creates features based on past data points for each individual solar panel string. This helps the model understand trends and recent conditions.
- **How & Details:**
  1. **Grouping:** It first groups the data by `string_id` to ensure that time-series features are calculated independently for each panel.
  2. **Lag Features:** It uses `.shift(n)` to create features representing the value of a variable from `n` time steps ago (e.g., `temperature_lag_1` is the temperature from the previous measurement).
  3. **Rolling Window Features:** It uses `.rolling(window=n)` to calculate statistics (like mean or standard deviation) over the last `n` time steps (e.g., `irradiance_roll_mean_3` is the average irradiance over the last 3 measurements). This smooths out noise and captures recent trends.
  4. **Fill NA:** It uses `bfill()` (backfill) to fill `NaN` values that appear at the beginning of each group after shifting.

4. Modeling and Pipeline Functions

These functions orchestrate the model training, evaluation, and prediction process.

`select_features(...)`

- **What:** Automatically selects the most important features from the vast number created in the previous steps. This reduces model complexity, training time, and the risk of overfitting.
- **How:** It trains a quick, temporary CatBoostRegressor model. It then extracts the feature importance scores from this model. Any feature with an importance score below a threshold (e.g., 1%) is discarded.

handle_outliers(...)

- **What:** Manages extreme or anomalous values in the target variable (efficiency).
- **How:** It uses the **Z-score method**. The Z-score measures how many standard deviations a data point is from the mean. If the absolute Z-score is above a z_threshold (typically 3), the point is considered an outlier. Instead of removing it, the script **caps** the value, replacing it with the maximum or minimum allowed value (e.g., mean + 3 * std_dev). This preserves the data point while reducing its extreme influence.

tune_catboost_model(...)

- **What:** Finds the best set of hyperparameters for the CatBoost model to maximize its predictive accuracy.
- **How:** It performs a grid search. It iterates through a predefined param_grid of different hyperparameter combinations (e.g., learning_rate, depth). For each combination, it trains a model and evaluates it on a validation set using Mean Absolute Error (MAE). It keeps track of the parameters that resulted in the lowest MAE. early_stopping_rounds is used to stop training a given model if its performance on the validation set doesn't improve for 100 consecutive iterations, saving time.

save_feature_importance(...)

- **What:** Saves a chart and a CSV file showing which features were most important to the final model's predictions.
- **How:** It gets the importance scores from the trained model, puts them in a pandas DataFrame, saves the DataFrame to a CSV, and uses matplotlib to generate and save a bar plot.

run_pipeline(config)

- **What:** The main orchestrator function that calls all other functions in the correct order to execute the entire pipeline from start to finish.
- **How & Details:**
    1. **Setup:** Initializes the output directory.
    2. **Data Loading & Preprocessing:** Loads train/test data, handles outliers, and calls the feature engineering functions (add_basic_features, add_advanced_features).
    3. **Data Splitting:** Splits the training data into a training set (for model learning) and a validation set (for tuning and evaluation) using train_test_split.

4. **Feature Selection:** Calls select_features to get the most valuable features.
5. **Hyperparameter Tuning:** Calls tune_catboost_model to find the best settings.
6. **Final Model Training:** Trains a new CatBoostRegressor on the *full* training dataset using the best-found parameters. use_best_model=True ensures that the model state with the lowest error on the validation set is saved.
7. **Post-processing (Residual Correction):** This is an advanced and clever step.
   - It calculates the errors (residuals) of the model on the validation set: error = actual_value - predicted_value.
   - It trains a simple LinearRegression model to predict these errors based on the initial prediction.
   - It then corrects the final test predictions by adding the predicted error: final_prediction = initial_prediction + predicted_error. This helps correct any systematic bias the main model might have (e.g., consistently under-predicting in a certain range).
   - Finally, np.clip(..., 0, 1) ensures all efficiency predictions are physically possible (between 0% and 100%).
8. **Saving:** Saves the final predictions, the trained CatBoost model, and the feature importance plot to the output directory.

5. Main Execution Block

```python
if __name__ == '__main__':
    try:
        # Load configuration
        config = load_config()
        logger.info("Configuration loaded successfully")

        # Run the pipeline
        predictions = run_pipeline(config)

    except Exception as e:
        logger.error(f"Error in main: {str(e)}")
        raise
```

- **What:** This is the entry point of the script. When you run python your_script_name.py, the code inside this block is executed.
- **How:**
  - The if __name__ == '__main__': check prevents this code from running if the script is imported as a module into another Python file.
  - It's wrapped in a try...except block for error handling. If any part of the pipeline fails, it will log the error and stop, rather than crashing silently.

- It first calls `load_config()` to get the settings and then passes these settings to the `run_pipeline()` function to kick off the entire process.