

Project automation and pipelines

Got it 👍 Please share the project with me, and I'll explain:

1. **How the project works in detail** – step-by-step explanation of its flow, logic, and output.
2. **Why it automates the process** – what problem it solves, what's the benefit of automation.
3. **Difference between Automation and Pipelines** – with examples (so it's clear when you need each).

Quick Primer Before You Share the Project

♦ Automation

- Automation means reducing **manual work** by writing scripts, tools, or workflows that handle repetitive tasks.
- Example: Instead of manually uploading an Excel file into a database every day, you write a Python script that runs daily to do it.

♦ Pipelines

- A **pipeline** is a **structured sequence of automated tasks** connected together.
- Think of it as a chain of automation steps.
- Example: In a **CI/CD pipeline**, code goes through steps → Build → Test → Deploy automatically.

👉 Difference:

- **Automation = one task automated.**
- **Pipeline = multiple automation steps connected in sequence.**

Now, if you share your project (GitHub link or description), I'll explain:

- The working of the project.
- Where automation is happening.
- Whether it can also be structured into a pipeline.

Do you want me to start with one of your projects (like **Automated Deployment Failure Analysis Script** or **Mock Database Generator**) or will you paste a new one?

Perfect 🚀 Thanks for sharing the code. Let's break this down **step by step**:

🔍 How the Project (Data Ingestion Module) Works

1. Configuration Setup

```
python

@dataclass
class DataIngestionconfig:
    train_data_path=os.path.join('artifacts','train.csv')
```

```
test_data_path=os.path.join('artifacts','test.csv')
raw_data_path=os.path.join('artifacts','raw.csv')
```

- A `dataclass` is used to store file paths for train, test, and raw data.
 - All outputs will be saved inside the `artifacts/` folder (common in ML projects).
-

2. DataIngestion Class

```
python
```

```
class DataIngestion:
    def __init__(self):
        self.ingestion_config=DataIngestionconfig()
```

- When you create a `DataIngestion` object, it loads the config with paths.
-

3. Reading the Dataset

```
python
```

```
df=pd.read_csv(os.path.join('notebooks/data','gemstone.csv'))
```

- Reads the **input dataset** (`gemstone.csv`) into a Pandas DataFrame.
-

4. Saving the Raw Dataset

```
python
```

```
os.makedirs(os.path.dirname(self.ingestion_config.raw_data_path),exist_ok=True)
df.to_csv(self.ingestion_config.raw_data_path,index=False)
```

- Ensures that the `artifacts/` folder exists.
 - Saves the **entire raw dataset** as `raw.csv` for backup/reference.
-

5. Splitting into Train/Test

```
python
```

```
train_set,test_set=train_test_split(df,test_size=0.30,random_state=42)
```

- Splits dataset → **70% train** & **30% test**.
 - `random_state=42` ensures reproducibility (same split every time).
-

6. Saving Train & Test Data

```
python
```

```
train_set.to_csv(self.ingestion_config.train_data_path,index=False,header=True)
test_set.to_csv(self.ingestion_config.test_data_path,index=False,header=True)
```

- Saves **train.csv** and **test.csv** in the `artifacts/` folder.

7. Return File Paths

```
python

return(
    self.ingestion_config.train_data_path,
    self.ingestion_config.test_data_path
)
```

- Returns the file paths → these will be used by the next step (e.g., Data Transformation or Model Training).

✅ In short:

This script **automates the process of reading raw data → saving → splitting → storing train/test datasets** so that later ML stages can use them.

⚙️ Why Automates the Process?

Normally, you'd:

1. Manually open Excel/CSV.
2. Clean or copy into another file.
3. Manually split into train/test.
4. Save separately.

This is repetitive and error-prone.

By **automation**:

- You just call `initiate_data_ingestion()` once, and everything is handled.
- Reduces **human error** (like mixing test/train).
- Ensures **reproducibility** across experiments.
- Fits neatly into an **ML pipeline** (Data → Train → Evaluate → Deploy).

🔄 Automation vs Pipelines in this Context

- **Automation (this code):**
Automates one task → **splitting dataset into train/test automatically**.
- **Pipeline:**
A sequence of such automated steps connected together. For example:
 1. **Data Ingestion** (this step).
 2. **Data Transformation** (cleaning, encoding, scaling).

3. **Model Training.**
4. **Model Evaluation.**
5. **Model Deployment.**

👉 This script is **one automation step** that will later become part of the **entire ML pipeline**.

⚡ Example:

- **Automation** = "I wrote a script to split train/test data."
 - **Pipeline** = "When new data arrives → ingestion runs → transformation runs → model retrains → new model is deployed automatically."
-

Do you want me to also **draw a flow diagram (like UML activity or pipeline flow)** for this ingestion process so it's easier to visualize?

Nice 🍯 now you've moved to the **Data Transformation** stage of your ML project. Let's break this down in detail like before.

🔍 How This Data Transformation Code Works

1. Configuration

```
python

@dataclass
class DataTransformationconfig:
    preprocessor_obj_file_path=os.path.join('artifacts','preprocessor.pkl')
```

- This stores the path where the **preprocessor object (pickle file)** will be saved.
 - Purpose: Once you define the transformation pipeline, you don't want to rebuild it every time → you save it in `artifacts/preprocessor.pkl`.
-

2. Class Initialization

```
python

class DataTransformation:
    def __init__(self):
        self.data_transformation_config=DataTransformationconfig()
```

- Loads config when you create an object.
-

3. Defining the Transformation Pipelines

```
python
```

```
def get_data_transformation_object(self):
```

This function builds **separate pipelines for numerical and categorical features**.

a. Categorical Columns

```
python
```

```
categorical_cols = ['cut', 'color', 'clarity']
```

- These are **non-numeric columns** describing the diamonds.

b. Numerical Columns

```
python
```

```
numerical_cols = ['carat', 'depth', 'table', 'x', 'y', 'z']
```

- These are numeric features.

c. Custom Order for Ordinal Encoding

```
python
```

```
cut_categories = ['Fair', 'Good', 'Very Good', 'Premium', 'Ideal']  
color_categories = ['D', 'E', 'F', 'G', 'H', 'I', 'J']  
clarity_categories = ['I1', 'SI2', 'SI1', 'VS2', 'VS1', 'VVS2', 'VVS1', 'IF']
```

- Instead of random integer mapping, categories are encoded according to **domain knowledge (quality order)**.
- Example: `Fair < Good < Very Good < Premium < Ideal`.

4. Numerical Pipeline

```
python
```

```
num_pipeline=Pipeline(  
    steps=[  
        ('imputer', SimpleImputer(strategy='median')),  
        ('scaler', StandardScaler())  
    ]  
)
```

- **Imputer**: Replaces missing values with **median**.
- **Scaler**: Standardizes features → mean = 0, std = 1.

5. Categorical Pipeline

```
python
```

```
cat_pipeline=Pipeline(  
    steps=[  
        ('imputer', SimpleImputer(strategy='most_frequent')),
```

```
(('ordinalencoder',OrdinalEncoder(categories=[cut_categories,color_categories,clarity_categories])),
 ('scaler',StandardScaler()))
]
)
```

- **Imputer:** Fills missing categorical values with the most frequent one.
- **OrdinalEncoder:** Converts categories into integers based on ranking.
- **Scaler:** Ensures encoded values are on similar scale as numeric ones.

6. Combining Pipelines

```
python

preprocessor=ColumnTransformer([
    ('num_pipeline',num_pipeline,numerical_cols),
    ('cat_pipeline',cat_pipeline,categorical_cols)
])
```

- **ColumnTransformer** applies different transformations to numeric vs categorical columns in **one step**.
- Returns a **preprocessor object**.

7. Applying the Preprocessor

```
python

train_df = pd.read_csv(train_path)
test_df = pd.read_csv(test_path)
```

- Reads training & testing datasets generated from the Data Ingestion step.

```
python

input_feature_train_df = train_df.drop(columns=[target_column_name,'id'],axis=1)
target_feature_train_df=train_df[target_column_name]
```

- Drops **price (target)** and **id (useless column)** → keeps only features.

```
python

input_feature_train_arr=preprocessing_obj.fit_transform(input_feature_train_df)
input_feature_test_arr=preprocessing_obj.transform(input_feature_test_df)
```

- **fit_transform** on train (learns scaling/encoding).
- **transform** on test (applies same scaling/encoding → avoids data leakage).

8. Final Train & Test Arrays

```
python
```

```
train_arr = np.c_[input_feature_train_arr, np.array(target_feature_train_df)]
test_arr = np.c_[input_feature_test_arr, np.array(target_feature_test_df)]
```

- Concatenates transformed features with the target → final NumPy arrays ready for model training.

9. Saving the Preprocessor

```
python

save_object(
    file_path=self.data_transformation_config.preprocessor_obj_file_path,
    obj=preprocessing_obj
)
```

- Stores the preprocessor in `preprocessor.pkl`.
- Later, when predicting on **new/unseen data**, you'll load this same preprocessor to ensure consistent transformations.

✅ In short:

This module **automates preprocessing** (missing values, encoding, scaling) for both **train & test data**, then **saves the transformation pipeline**.

⚙️ Why Automates the Process?

- If you had to **manually clean & scale** features every time, it would:
 - Take hours.
 - Introduce inconsistency (maybe you scale train differently from test).
 - Cause human error.

Automation ensures:

- **Consistency** (same scaling & encoding always).
- **Reusability** (preprocessor.pkl can be reused on any new data).
- **Scalability** (works on 100 rows or 10M rows the same way).

🔄 Automation vs Pipelines (in this step)

- **Automation here:** Missing value imputation, encoding, scaling are automatically handled instead of manually coding each.
- **Pipeline here:** You are literally using **sklearn pipelines** to chain multiple preprocessing steps → one object handles them all.

👉 Difference:

- If you only automated **imputation** → that's automation.
- But since you chained **imputation** → **encoding** → **scaling** for both categorical & numerical columns, that's a **pipeline**.

⚡ Example in your project:

1. **Data Ingestion** = one automation step (splitting train/test).
2. **Data Transformation** = pipeline (multiple automated steps together).
3. **Model Training** (later) = another automation step.
4. **Connecting all three** = end-to-end ML pipeline.

Would you like me to **draw a full ML pipeline flow (Data Ingestion → Data Transformation → Model Training → Evaluation)** so you can visualize how these pieces connect together?

Awesome 🎯 now you've reached the **Model Training** step of your ML pipeline. Let's go through this code thoroughly.

🔍 How the Model Training Code Works

1. Configuration

```
python

@dataclass
class ModelTrainerConfig:
    trained_model_file_path = os.path.join('artifacts','model.pkl')
```

- Defines where the **trained ML model** will be saved.
- All models are stored inside the `artifacts/` folder.

2. Class Initialization

```
python

class ModelTrainer:
    def __init__(self):
        self.model_trainer_config = ModelTrainerConfig()
```

- Loads the config whenever `ModelTrainer()` is created.

3. Method: `initate_model_training`

```
python

def initate_model_training(self,train_array,test_array):
```

- Accepts the **transformed train & test arrays** from the Data Transformation step.
- These arrays already include features + target column.

4. Splitting Features & Target

```
python

X_train, y_train, X_test, y_test = (
    train_array[:, :-1],
    train_array[:, -1],
    test_array[:, :-1],
    test_array[:, -1]
)
```

- `train_array` and `test_array` are NumPy arrays where the **last column is the target** (price).
 - `X_train, X_test` = independent variables (features).
 - `y_train, y_test` = dependent variable (price).
-

5. Defining Models

```
python

models={
    'LinearRegression':LinearRegression(),
    'Lasso':Lasso(),
    'Ridge':Ridge(),
    'Elasticnet':ElasticNet(),
    'DecisionTree':DecisionTreeRegressor()
}
```

- A dictionary of models to try:
 - **LinearRegression** – basic regression.
 - **Lasso** – linear regression + L1 penalty (feature selection).
 - **Ridge** – linear regression + L2 penalty (reduces overfitting).
 - **ElasticNet** – combo of Lasso & Ridge.
 - **DecisionTreeRegressor** – non-linear model that splits data.
-

6. Evaluating Models

```
python

model_report=dict=evaluate_model(X_train,y_train,X_test,y_test,models)
```

- Calls a helper function `evaluate_model` (from `src.utils`).
- This function likely:
 1. Trains each model.
 2. Predicts on test data.
 3. Returns evaluation scores (probably **R² Score**).

Example of expected output:

```
python
```

```
{
  'LinearRegression': 0.89,
  'Lasso': 0.86,
  'Ridge': 0.88,
  'Elasticnet': 0.87,
  'DecisionTree': 0.92
}
```

7. Finding the Best Model

```
python

best_model_score = max(sorted(model_report.values()))

best_model_name = list(model_report.keys())[
    list(model_report.values()).index(best_model_score)
]

best_model = models[best_model_name]
```

- Finds the model with the **highest R² Score**.
- Gets both the **model name** and the actual **model object**.

8. Logging Best Model

```
python

print(f'Best Model Found , Model Name : {best_model_name} , R2 Score : {best_model_score}')
logging.info(f'Best Model Found , Model Name : {best_model_name} , R2 Score : {best_model_score}')
```

- Prints and logs the result.
- Example:

```
yaml
```

```
Best Model Found , Model Name : DecisionTree , R2 Score : 0.92
```

9. Saving the Best Model

```
python

save_object(
    file_path=self.model_trainer_config.trained_model_file_path,
    obj=best_model
)
```

- Saves the **best model** as `model.pkl`.
- Later you can load this file to make predictions on new data.

✅ In short:

This module **automates model selection by training multiple models, evaluating them, picking the best, and saving it for future use.**

⚙️ Why Automates the Process?

If you trained models manually, you would:

1. Train each model separately.
2. Manually calculate evaluation metrics.
3. Compare results.
4. Save the model yourself.

This would take a lot of time & introduce errors.

By automating:

- One function call → trains & evaluates all models.
 - Best model is auto-selected.
 - Saved model ensures reproducibility.
 - Saves **time** + **effort** for experimentation.
-

🔄 Automation vs Pipeline (Here)

- **Automation:**
 - Training & evaluating each model without manual coding.
 - Saving the best model automatically.
- **Pipeline:**
 - When you connect **Data Ingestion** → **Transformation** → **Model Training** → **Evaluation** → **Deployment** in one continuous flow.
 - Example: A new dataset arrives → automatically ingested → transformed → best model retrained → deployed.

So here, ModelTrainer is **one automation step** in the **ML pipeline**.

👉 Up to now, you've built:

1. **Data Ingestion (automation)**
2. **Data Transformation (pipeline automation)**
3. **Model Training (automation with model selection)**

Together, they are forming an **end-to-end ML pipeline**.

Do you want me to now **connect all three steps (Ingestion → Transformation → Training) in a single orchestrator pipeline script** so you see how they work together as one flow?

Perfect 🎯 now you've built the **final orchestrator script** that connects all the pieces of your ML project together. Let's break it down step by step:

🔍 How This Orchestrator Script Works

1. Imports

```
python

from src.components.data_ingestion import DataIngestion
from src.components.data_transformation import DataTransformation
from src.components.model_trainer import ModelTrainer
```

- Brings in the **three major modules** you wrote earlier:
 - **DataIngestion** → loads and splits raw data.
 - **DataTransformation** → cleans, encodes, and scales features.
 - **ModelTrainer** → trains, evaluates, and saves the best ML model.

2. Main Execution

```
python

if __name__ == '__main__':
```

- Ensures the script only runs when executed directly (not when imported).

3. Step 1: Data Ingestion

```
python

obj=DataIngestion()
train_data_path,test_data_path=obj.initiate_data_ingestion()
```

- Creates a `DataIngestion` object.
- Runs `initiate_data_ingestion()` which:
 - Reads `gemstone.csv`.
 - Saves `raw.csv`, `train.csv`, `test.csv` inside `artifacts/`.
 - Returns the file paths of `train.csv` and `test.csv`.

📌 Output example:

```
bash

artifacts/train.csv artifacts/test.csv
```

4. Step 2: Data Transformation

python

```
data_transformation=DataTransformation()
train_arr,test_arr,_=data_transformation.initiate_data_transformation(train_data_path,test_data_path)
```

- Creates a `DataTransformation` object.
- Runs `initiate_data_transformation()` which:
 - Reads the train/test CSVs.
 - Drops unwanted columns (`price`, `id`).
 - Applies the preprocessing pipeline (imputation, encoding, scaling).
 - Returns:
 - `train_arr` → transformed training data.
 - `test_arr` → transformed testing data.
 - Path to saved preprocessor object (`preprocessor.pkl`).

🔗 Example:

- `train_arr` shape: `(rows, features+1)`
- Last column = target (`price`).

5. Step 3: Model Training

python

```
model_trainer=ModelTrainer()
model_trainer.initate_model_training(train_arr,test_arr)
```

- Creates a `ModelTrainer` object.
- Runs `initate_model_training()` which:
 - Splits features/target from arrays.
 - Trains multiple models (LinearRegression, Ridge, Lasso, ElasticNet, DecisionTree).
 - Evaluates them (probably using R^2 Score).
 - Selects the best model.
 - Saves it as `model.pkl` inside `artifacts/`.

🔗 Example Output:

yaml

```
{'LinearRegression': 0.89, 'Lasso': 0.86, 'Ridge': 0.88, 'Elasticnet': 0.87, 'DecisionTree': 0.92}
```

```
Best Model Found , Model Name : DecisionTree , R2 Score : 0.92
```

✅ In short

This script is your **end-to-end ML pipeline**:

1. **Data Ingestion** → Collects and splits dataset.
2. **Data Transformation** → Preprocesses features consistently.
3. **Model Training** → Trains, evaluates, and saves the best model.

All results (`train.csv` , `test.csv` , `preprocessor.pkl` , `model.pkl`) are stored in **artifacts/** for reproducibility.

⚙️ Why Automates the Process?

- Without this script, you'd have to **manually run each step separately**.
 - With automation:
 - One run → everything executes in the correct sequence.
 - Ensures **consistency** across experiments.
 - Saves **time** → faster ML development.
 - Reduces **human error** (e.g., forgetting to scale test data).
-

🔄 Automation vs Pipeline Here

- **Automation:** Each module (ingestion, transformation, training) automates its own task.
- **Pipeline:** This orchestrator connects them together into a **seamless flow** → now you have an **ML pipeline**.

👉 Example:

- If you only ran **Data Ingestion** → that's automation.
 - If you connect **Ingestion** → **Transformation** → **Training** → **Deployment** → that's a pipeline.
-

⚡ Final Note:

This orchestrator is basically your **training pipeline script**. In real-world projects, you'd later add:

- **Model Evaluation & Metrics Dashboard**
 - **Deployment (Flask/FastAPI/Streamlit)**
 - **CI/CD automation**
-

Do you want me to **draw a pipeline diagram** (showing Data Ingestion → Transformation → Training → Saved Model) so it's crystal clear how the flow works?

Perfect 👍 You've shared the **Prediction Pipeline** part of your **Diamond Price Prediction** project. Let's break this down in detail, explain **how it works**, **why automation is used**, and **how it differs from pipelines**:

🔍 Step-by-Step Explanation of the Code

1. `PredictPipeline` class

This class is responsible for **loading the trained model & preprocessor** and then making predictions on new incoming data.

◆ Key Points:

- `preprocessor.pkl`:

This file (saved earlier during training) contains the preprocessing steps (scaling, encoding, etc.).

Without applying the same preprocessing, the model cannot understand raw input data.

- `model.pkl`:

This is the best trained model (Linear Regression, Ridge, etc.) saved in `artifacts`.

- `predict()` method:

```
python

preprocessor = load_object(preprocessor_path)
model = load_object(model_path)

data_scaled = preprocessor.transform(features)
pred = model.predict(data_scaled)
```

- First loads the **preprocessor & model**.
- Applies preprocessing transformations to new input features.
- Uses the trained model to predict the **diamond price**.

2. `CustomData` class

This acts as a **data collector** for new input values (from UI, API, or CLI).

◆ Example:

```
python

custom_data = CustomData(
    carat=1.0,
    depth=61.5,
    table=55,
    x=5.7,
    y=5.6,
    z=3.5,
    cut="Ideal",
    color="E",
    clarity="VS2"
)
```

- Internally, it converts the input values into a **DataFrame** format because scikit-learn works best with DataFrames.
- `get_data_as_dataframe()` method ensures input is ready for preprocessing.

⚡ Why Automate the Process?

Manually doing this would look like:

1. Load preprocessor & model.
2. Convert input into DataFrame.

3. Apply transformations.
4. Make prediction.

If every time you had to write all of this code manually, it would be error-prone and time-consuming.

Automation (via `PredictPipeline` & `CustomData`) makes it:

- **Reusable** → same code works for UI, API, CLI.
- **Reliable** → ensures preprocessing & model steps are consistent.
- **Maintainable** → if you update preprocessing, you don't need to change prediction logic everywhere.

Automation vs Pipelines

Aspect	Automation	Pipeline
Definition	Using code/scripts to perform repetitive tasks without manual effort.	A structured sequence of data transformation + model training steps.
Focus	Reducing manual effort (ex: auto-load model, auto-transform).	Ensuring ordered execution of ML steps (clean → transform → train → evaluate).
Example in your project	Loading model & preprocessor automatically in <code>PredictPipeline</code> .	<code>sklearn.pipeline.Pipeline</code> (not used here, but could combine preprocessing + model into one pipeline).
When used	During deployment (prediction stage).	During training (preprocessing + training workflow).

✅ In your project:

- **Automation** → Code automatically handles preprocessing + prediction via `PredictPipeline`.
- **Pipeline (conceptually)** → Your project workflow: ingestion → transformation → training → saving model → prediction.

👉 Would you like me to also **draw a flow diagram** (step by step pipeline of your whole project from ingestion to prediction) so it's crystal clear?

Perfect 🎯 now you've shared the **Flask app** part — this is where your whole project comes together into a **deployable web application**.

Let's break it down step by step:

How this Flask App Works

1. Setup

```
python

from flask import Flask, request, render_template
from src.pipelines.prediction_pipeline import CustomData, PredictPipeline
```

- Imports **Flask** (to run the web server).
- Imports your **prediction pipeline** (`CustomData` & `PredictPipeline`) to handle model input & prediction.

```
python

application = Flask(__name__)
app = application
```

- Initializes the Flask app.

2. Route Handling (/)

```
python

@app.route('/', methods=['GET', 'POST'])
def predict():
```

- The `/` route is both **GET** (load page) and **POST** (submit form).
- Handles two cases:
 1. Show form (empty fields + 0 result).
 2. Submit form → predict diamond price.

3. Form Data Handling

```
python

form_data = {
    'carat': "", 'depth': "", 'table': "", 'x': "",
    'y': "", 'z': "", 'cut': "", 'color': "", 'clarity': ""
}
```

- A dictionary to **store user inputs** (so form values don't vanish after refresh).

4. POST Request Handling

```
python

if request.method == 'POST':
    if 'reset' in request.form:
        final_result = 0
```

- **Reset button** clears the form and resets prediction to 0.

```
python

elif 'predict' in request.form:
    # Gather inputs
    form_data = {key: request.form.get(key, "") for key in form_data}

    data = CustomData(
        carat=float(form_data['carat']),
        depth=float(form_data['depth']),
        table=float(form_data['table']),
        x=float(form_data['x']),
        y=float(form_data['y']),
        z=float(form_data['z']),
        cut=form_data['cut'],
        color=form_data['color'],
        clarity=form_data['clarity']
    )
```

- Collects form values, converts numeric fields into floats, keeps categorical as strings.
- Creates a **CustomData** object.

5. Prediction

```
python

final_new_data = data.get_data_as_dataframe()
predict_pipeline = PredictPipeline()
pred = predict_pipeline.predict(final_new_data)
final_result = round(pred[0], 2)
```

- Converts input → DataFrame.
- Loads preprocessor + trained model.
- Applies transformation + prediction.
- Rounds predicted diamond price to 2 decimal places.

6. Error Handling

```
python

except Exception as e:
    final_result = f"Error: {str(e)}"
```

- If preprocessing/model fails, error is displayed instead of crashing app.

7. Render Template

```
python
```

```
return render_template('form.html', final_result=final_result, form_data=form_data)
```

- Passes the prediction result + form values to `form.html` (your frontend template).
- `form.html` likely contains input fields and shows `final_result`.

⚡ Why Automate Here?

Without automation:

- You would manually preprocess inputs every time.
- You would manually load model files.
- You would risk mismatched transformations.

With `PredictPipeline` + `CustomData`, Flask just has to **collect form values** → **call pipeline** → **display result**.

🔄 Pipeline vs Automation in Flask

- **Automation:**
Flask automatically handles converting inputs → DataFrame → preprocessor → prediction → result.
- **Pipeline:**
The workflow from raw form inputs → cleaned data → scaled → predicted price is your ML pipeline.

✅ End-to-End Flow:

1. **User** enters diamond details in `form.html`.
2. **Flask** receives input → `CustomData` → DataFrame.
3. **PredictPipeline** loads preprocessor + model → transforms data → predicts price.
4. **Flask** sends result back → displayed in `form.html`.

👉 Do you want me to also write a **sample** `form.html` **template** (with all input fields and reset/predict buttons) so your Flask app will run without errors?

2/2