ELECTRIC MOTOR TEMPERATURE PREDICTION

Category: Machine Learning

Skills Required:

- Python
- Exploratory data Analysis
- Numpy
- Scikit-Learn

Project Description:

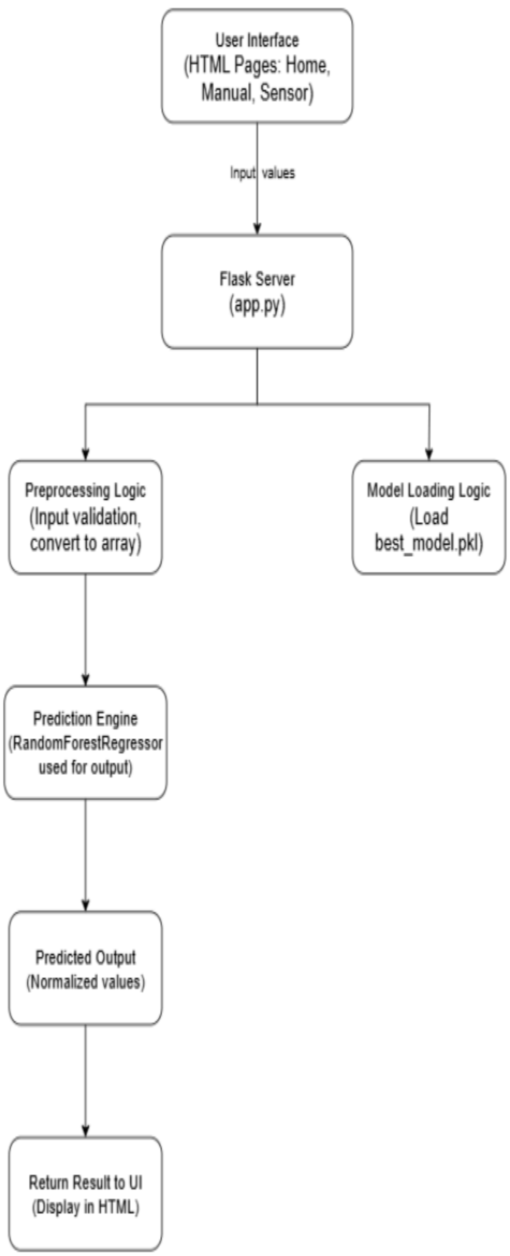
The permanent-magnet synchronous machine (PMSM) drive is one of the best choices for a full range of motion control applications. For example, the PMSM is widely used in robotics, machine tools, actuators, and it is being considered in high-power applications such as industrial drives and vehicular propulsion. It is also used for residential/commercial applications.
The task is to design a model with appropriate feature engineering that estimates the target temperature of a rotor.

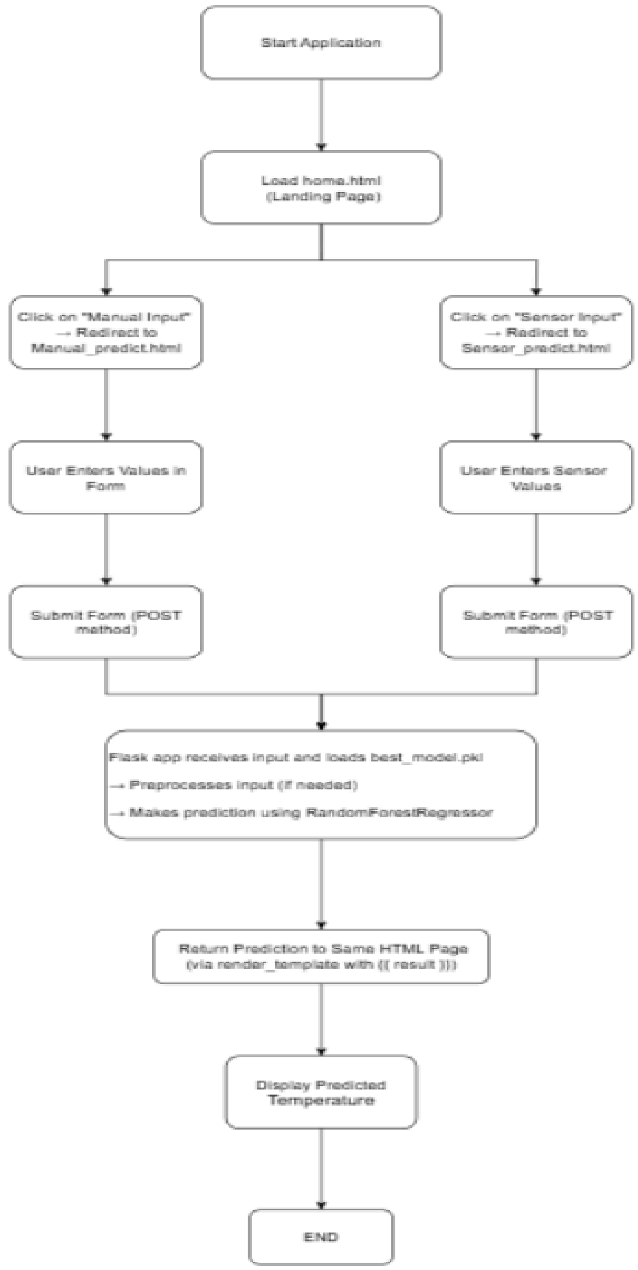
The workflow includes:

- Data Preprocessing and Feature Engineering to handle sensor input features and derive the most relevant predictors for rotor temperature.
- Model Training and Evaluation using multiple machine learning algorithms including: Linear Regression, Decision Tree Regressor, Random Forest Regressor, Support Vector Machine (SVM).

All models will be evaluated using suitable performance metrics (such as MAE, MSE, RMSE, and R^2 Score) on both training and test data.o demonstrate real-time inference, the final model will be deployed using Flask, a Python-based web framework. A minimal web interface will be created to allow users to input relevant sensor values and obtain real-time predictions of rotor temperature.

TECHNICAL ARCHITECTURE





PRE REQUISITES

To successfully complete this project, the following software, Python libraries, and conceptual knowledge are required:

Software Required:

- Anaconda Navigator: Anaconda is a distribution of Python and R for scientific computing and data science. It includes tools like Jupyter Notebook and package management.

Python Packages (Libraries): Open the Anaconda Prompt as Administrator and install the following packages:

- pip install numpy
- pip install pandas
- pip install scikit-learn
- pip install matplotlib
- pip install pickle-mixin
- pip install seaborn
- pip install Flask

Project Objectives: By the end of this project:

- You'll be able to understand the problem to classify if it is a regression or a classification kind of problem.
- You will be able to know how to pre-process/clean the data using different data pre-processing techniques.
- Applying different algorithms according to the dataset
- You will be able to know how to find the accuracy of the model.
- You will be able to build web applications using the Flask framework.

Project Flow:

- User interacts with the UI to enter the input.
- Entered input is analyzed by the model which is integrated.
- Once model analyses the input the prediction is showcased on the UI

To accomplish this, we have to complete all the activities listed below,

- Data collection
    - Download the dataset
    - Visualizing and analyzing data
    - Read the dataset
    - Univariate analysis
    - Multivariate analysis
    - Descriptive analysis
- Data pre-processing
    - Drop unwanted features
    - Checking for null values
    - Remove negative data
    - Handling outlier
    - Handling categorical data
    - Handling Imbalanced data
    - Splitting data into train and test
- Model building
    - Import the model building libraries
    - Initializing the model
    - Training and testing the model
    - Evaluating performance of model
    - Save the model
- Application Building
    - Create an HTML file
    - Build python code

DATA COLLECTION

ML depends heavily on data, It is most crucial aspect that makes algorithm training possible. So this section allows you to download the required dataset.

1. **Download the Dataset:** The dataset was obtained in CSV format and downloaded to the local working directory. It includes important features such as torque, stator temperatures, voltage, motor speed, and more.

2. **Visualizing and Analyzing Data:** Once the dataset is loaded, the next step is to understand its structure and relationships using statistical and visual analysis.

3. **Read the Dataset:** The dataset was loaded into a Jupyter Notebook using the pandas library. It was stored as a DataFrame to enable easier data manipulation and analysis.

4. **Univariate Analysis:** Univariate analysis helps understand the distribution and nature of individual variables. Histograms and boxplots were used to visualize single features like rotor temperature, torque, etc.

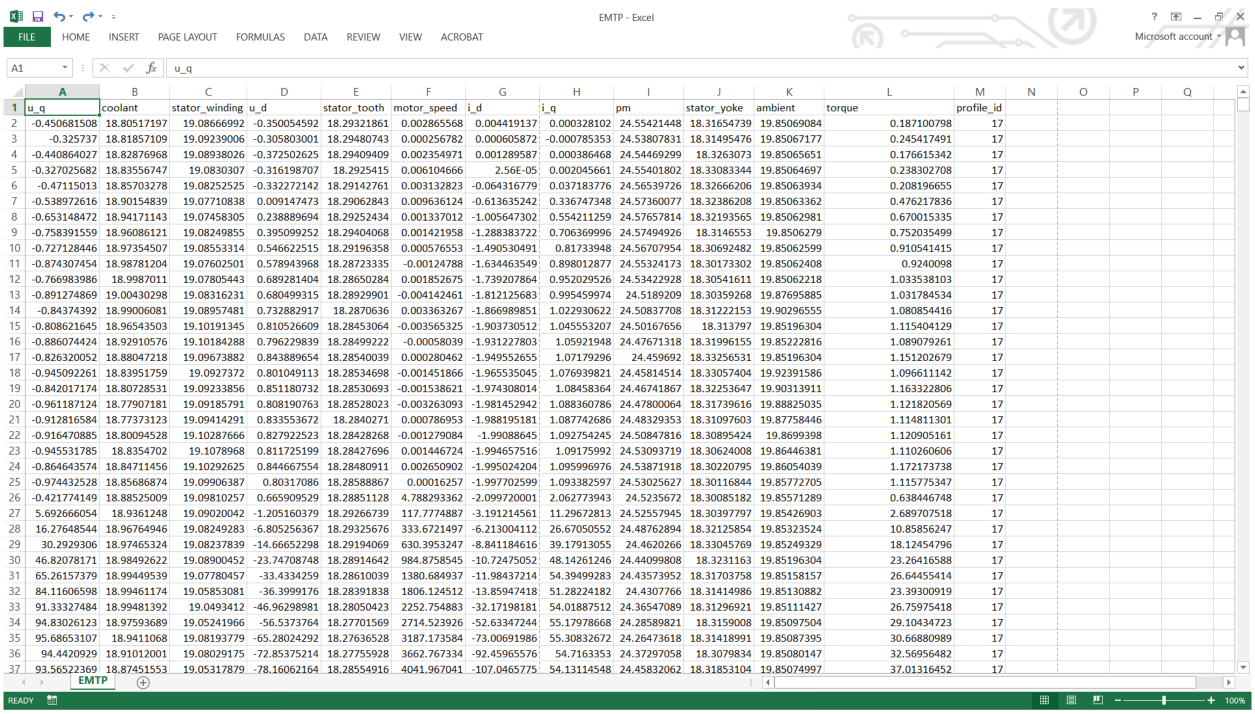
5. **Multivariate Analysis:** This analysis is used to examine the relationships between multiple features. Correlation matrices and scatter plots help in identifying feature interactions.

6. **Descriptive Analysis:** Descriptive statistics provide insights into central tendencies and variability. Using .describe(), values like mean, standard deviation, min, and max were computed.

DOWNLOAD THE DATASET

Download the dataset from below link.

Link:https://www.kaggle.com/wkirgsn/electric-motor-temperature



	u_q	coolant	stator_winding	u_d	stator_tooth	motor_speed	i_d	i_q	pm	stator_yoke	ambient	torque	profile_id
2	-0.450681508	18.80517197	19.08666992	-0.350054592	18.29321861	0.002865568	0.004419137	0.000328102	24.55421448	18.31654739	19.85069084	0.187100798	17
3	-0.325737	18.81857109	19.09239006	-0.305803001	18.29480743	0.000256782	0.000605872	-0.000785353	24.53807831	18.31495476	19.85067177	0.245417491	17
4	-0.440864027	18.82876968	19.08938026	-0.372502625	18.29409409	0.002354971	0.001289587	0.000386468	24.54469299	18.3263073	19.85065651	0.176615342	17
5	-0.327025682	18.83556747	19.0830307	-0.316198707	18.2925415	0.006104666	2.56E-05	0.002045661	24.55401802	18.33083344	19.85064697	0.238302708	17
6	-0.47115013	18.85703278	19.08252525	-0.332272142	18.29142761	0.003132823	-0.064316779	0.037183776	24.56539726	18.32666206	19.85063934	0.208196655	17
7	-0.538972616	18.90154839	19.07710838	0.009147473	18.29062843	0.009636124	-0.613635242	0.336747348	24.57360077	18.32386208	19.85063362	0.476217836	17
8	-0.653148472	18.94171143	19.07458305	0.238889694	18.29252434	0.001337012	-1.005647302	0.554211259	24.57657814	18.32193565	19.85062981	0.670015335	17
9	-0.758391559	18.96086121	19.08249855	0.395099252	18.29404068	0.001421958	-1.288383722	0.706369996	24.57494926	18.3146553	19.8506279	0.752035499	17
10	-0.727128446	18.97354507	19.08553314	0.546622515	18.29196358	0.000576553	-1.490530491	0.81733948	24.56707954	18.30692482	19.85062599	0.910541415	17
11	-0.874307454	18.98781204	19.07602501	0.578943968	18.28723335	-0.00124788	-1.634463549	0.898012877	24.55324173	18.30173302	19.85062408	0.9240098	17
12	-0.766983986	18.9987011	19.07805443	0.689281404	18.28650284	0.001852675	-1.739207864	0.952029526	24.53422928	18.30541611	19.85062218	1.033538103	17
13	-0.891274869	19.00430298	19.08316231	0.680499315	18.28929901	-0.004142461	-1.812125683	0.995459974	24.5189209	18.30359268	19.87695885	1.031784534	17
14	-0.84374392	18.99006081	19.08957481	0.732882917	18.2870636	0.003363267	-1.866989851	1.022930622	24.50837708	18.31222153	19.90296555	1.080854416	17
15	-0.808621645	18.96543503	19.10191345	0.810526609	18.28453064	-0.003565325	-1.903730512	1.045553207	24.50167656	18.313797	19.85196304	1.115404129	17
16	-0.886074424	18.92910576	19.10184288	0.796229839	18.28499222	-0.00058039	-1.931227803	1.05921948	24.47671318	18.31996155	19.85222816	1.089079261	17
17	-0.826320052	18.88047218	19.09673882	0.843889654	18.28540039	0.000280462	-1.949552655	1.07179296	24.459692	18.33256531	19.85196304	1.151202679	17
18	-0.945092261	18.83951759	19.0927372	0.801049113	18.28534698	-0.001451866	-1.965535045	1.076939821	24.45814514	18.33057404	19.92391586	1.096611142	17
19	-0.842017174	18.80728531	19.09233856	0.851180732	18.28530693	-0.001538621	-1.974308014	1.08458364	24.46741867	18.32253647	19.90313911	1.163322806	17
20	-0.961187124	18.77907181	19.09185791	0.808190763	18.28528023	-0.003263093	-1.981452942	1.088360786	24.47800064	18.31739616	19.88825035	1.121820569	17
21	-0.912816584	18.77373123	19.09414291	0.833553672	18.2840271	0.000786953	-1.988195181	1.087742686	24.48329353	18.31097603	19.87758446	1.114811301	17
22	-0.916470885	18.80094528	19.10287666	0.827922523	18.28428268	-0.001279084	-1.99088645	1.092754245	24.50847816	18.30895424	19.8699398	1.120905161	17
23	-0.945531785	18.8354702	19.1078968	0.811725199	18.28427696	0.001446724	-1.994657516	1.09175992	24.53093719	18.30624008	19.86446381	1.110260606	17
24	-0.864643574	18.84711456	19.10292625	0.844667554	18.28480911	0.002650902	-1.995024204	1.095996976	24.53871918	18.30220795	19.86054039	1.172173738	17
25	-0.974432528	18.85686874	19.09906387	0.80317086	18.28588867	0.00016257	-1.997702599	1.093382597	24.53025627	18.30116844	19.85772705	1.115775347	17
26	-0.421774149	18.88525009	19.09810257	0.665909529	18.28851128	4.788293362	-2.099720001	2.062773943	24.5235672	18.30085182	19.85571289	0.638446748	17
27	5.692666054	18.9361248	19.09020042	-1.205160379	18.29266739	117.7774887	-3.191214561	11.29672813	24.52557945	18.30397797	19.85426903	2.689707518	17
28	16.27648544	18.96764946	19.08249283	-6.805256367	18.29325676	333.6721497	-6.213004112	26.67050552	24.48762894	18.32125854	19.85323524	10.85856247	17
29	30.2929306	18.97465324	19.08237839	-14.66652298	18.29194069	630.3953247	-8.841184616	39.17913055	24.4620266	18.33045769	19.85249329	18.12454796	17
30	46.82078171	18.98492622	19.08900452	-23.74708748	18.28914642	984.8758545	-10.72475052	48.14261246	24.44099808	18.3231163	19.85196304	23.26416588	17
31	65.26157379	18.99449539	19.07780457	-33.4334259	18.28610039	1380.684937	-11.98437214	54.39499283	24.43573952	18.31703758	19.85158157	26.64455414	17
32	84.11606598	18.99461174	19.05853081	-36.3999176	18.28391838	1806.124512	-13.85947418	51.28224182	24.4307766	18.31414986	19.85130882	23.39300919	17
33	91.33327484	18.99481392	19.0493412	-46.96298981	18.28050423	2252.754883	-32.17198181	54.01887512	24.36547089	18.31296921	19.85111427	26.75975418	17
34	94.83026123	18.97593689	19.05241966	-56.5373764	18.27701569	2714.523926	-52.63347244	55.17978668	24.28589821	18.3159008	19.85097504	29.10434723	17
35	95.68653107	18.9411068	19.08193779	-65.28024292	18.27636528	3187.173584	-73.00691986	55.30832672	24.26473618	18.31418991	19.85087395	30.66880989	17
36	94.4420929	18.91012001	19.08029175	-72.85375214	18.27755928	3662.767334	-92.45965576	54.7163353	24.37297058	18.3079834	19.85080147	32.56956482	17
37	93.56522369	18.87451553	19.05317879	-78.16062164	18.28554916	4041.967041	-107.0465775	54.13114548	24.45832062	18.31853104	19.85074997	37.01316452	17

VISUALIZING AND ANALYZING THE DATA

Read and Understand the Dataset: As the dataset is now downloaded, the next step is to read and understand the data using various visualization and analysis techniques. This step helps identify the structure, patterns, and potential issues within the data before model building.

- **Note:** There are numerous techniques available for data understanding. In this project, we have used some essential ones, but additional techniques can be applied based on the project needs.

Activity 1: Importing the Required Libraries

Before reading the dataset, we import the necessary Python libraries that assist in data analysis and visualization.

These libraries include pandas, numpy, matplotlib, seaborn, and others.

```
In [1]: # 🚀 Electric Motor Temperature Prediction
        # In this project, we predict the rotor temperature of an electric motor using regression techniques.
```

```
In [2]: # Data handling
        import numpy as np
        import pandas as pd

        # Visualization
        import matplotlib.pyplot as plt
        import seaborn as sns

        # Warning filter
        import warnings
        warnings.filterwarnings('ignore')

        # 🤖 Machine Learning
        from sklearn.model_selection import train_test_split
        from sklearn.preprocessing import StandardScaler
        from sklearn.linear_model import LinearRegression
        from sklearn.tree import DecisionTreeRegressor
        from sklearn.ensemble import RandomForestRegressor
        from sklearn.svm import SVR
        from sklearn.metrics import mean_squared_error, r2_score

        # 💾 Save/Load Model
        import pickle  # Works with pickle-mixin too

        # 🌐 Flask Web Framework
        from flask import Flask, request, render_template
```

READ THE DATASET

The dataset is read as a DataFrame named df using the pandas library, where pd is the commonly used alias for the pandas package.
This allows efficient data manipulation, analysis, and visualization throughout the application.

In [3]:
```python
# For Windows users (replace "Zaid" with your actual username if needed)
df = pd.read_csv(r'C:\Users\Zaid\Desktop\EMTP.csv')

# Preview the first 5 rows
df.head()
```

Out[3]:

	u_q	coolant	stator_winding	u_d	stator_tooth	motor_speed	i_d	i_q	pm	stator_yoke	ambient	torque	profile_id
0	-0.450682	18.805172	19.086670	-0.350055	18.293219	0.002866	0.004419	0.000328	24.554214	18.316547	19.850691	0.187101	17
1	-0.325737	18.818571	19.092390	-0.305803	18.294807	0.000257	0.000606	-0.000785	24.538078	18.314955	19.850672	0.245417	17
2	-0.440864	18.828770	19.089380	-0.372503	18.294094	0.002355	0.001290	0.000386	24.544693	18.326307	19.850657	0.176615	17
3	-0.327026	18.835567	19.083031	-0.316199	18.292541	0.006105	0.000026	0.002046	24.554018	18.330833	19.850647	0.238303	17
4	-0.471150	18.857033	19.082525	-0.332272	18.291428	0.003133	-0.064317	0.037184	24.565397	18.326662	19.850639	0.208197	17

UNIVARIATE ANALYSIS

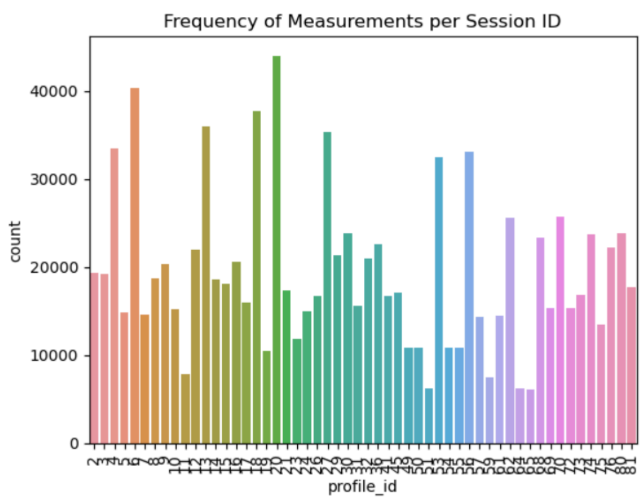
Univariate analysis involves analyzing each feature (or variable) individually to understand its distribution, central tendency, and variability. It helps in identifying patterns, outliers, and the general behavior of the data.

Bar Graph:

A bar graph is used to represent categorical data using rectangular bars, where the length of each bar is proportional to the frequency or value it represents. Bars can be oriented vertically or horizontally.

- Observation: In the dataset, Session IDs 66, 6, and 20 have the highest number of measurements recorded, as shown in the bar graph.

In [4]:
```
# Countplot of a categorical feature (if any, like session_id or labels)
sns.countplot(x='profile_id', data=df)
plt.xticks(rotation=90)
plt.title('Frequency of Measurements per Session ID')
plt.show()
```

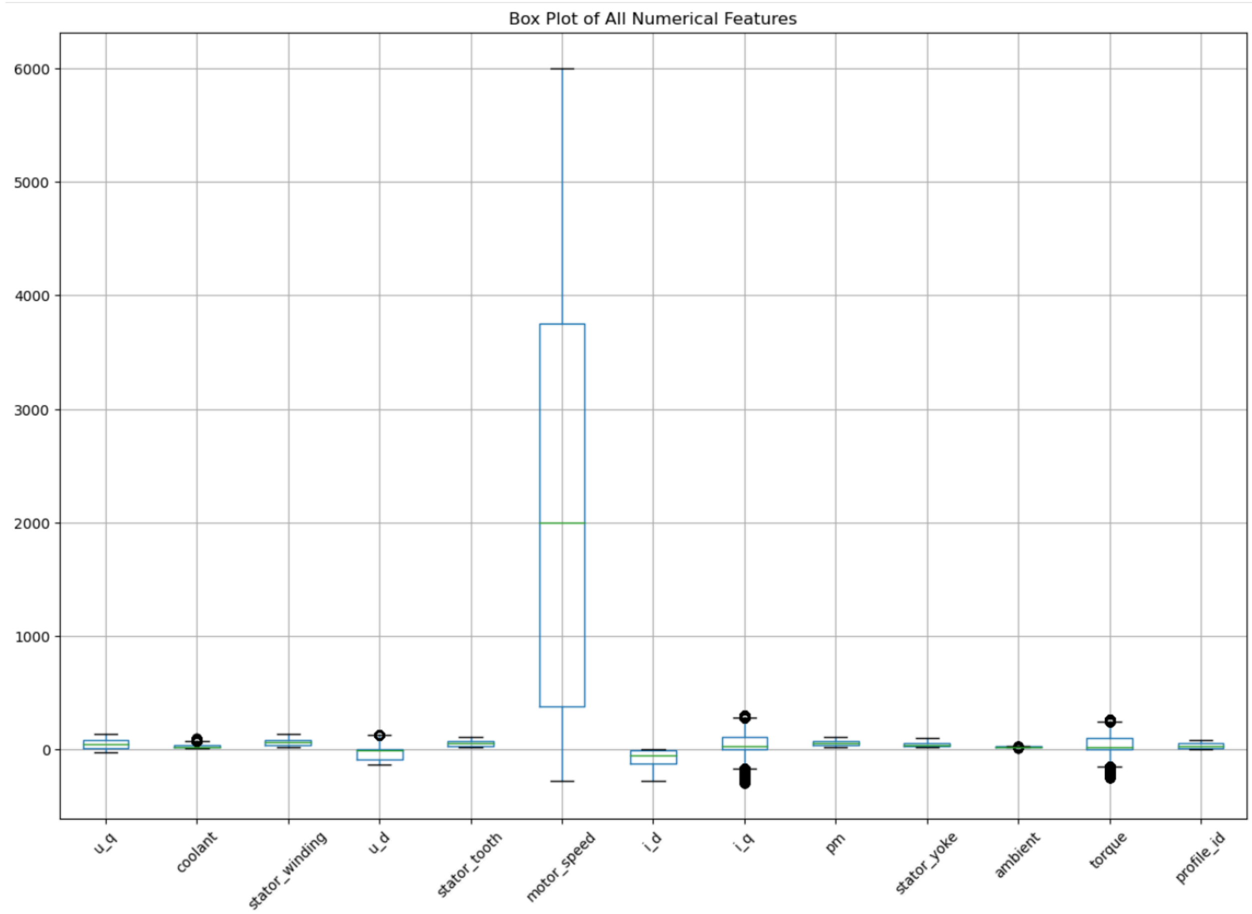


Box Plot:
A box plot is a graphical representation of data based on a five-number summary: minimum, first quartile (Q1), median, third quartile (Q3), and maximum.
It is especially useful for identifying outliers and understanding the spread and symmetry of the data.

- Observation: All features were plotted using boxplots. Most variables show that their mean and median are close, indicating low skewness and a fairly symmetric distribution.

In [5]:
```
# Box plots for all numerical columns

plt.figure(figsize=(15, 10))
df.boxplot(rot=90)
plt.title('Box Plot of All Numerical Features')
plt.xticks(rotation=45)
plt.show()
```

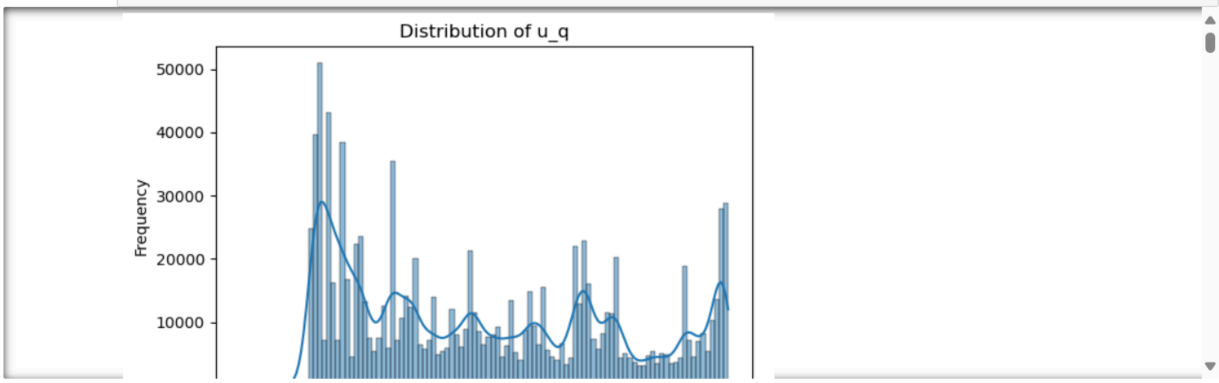


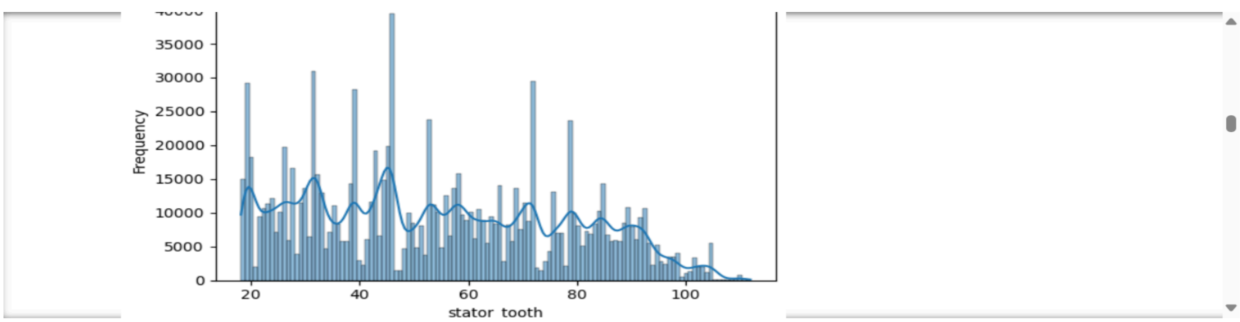
Distribution Plot:
A distribution plot (or distplot) shows the distribution of a numerical variable over a continuous interval.
It is useful for identifying the shape of the data (normal, skewed, etc.) and comparing it across multiple variables.

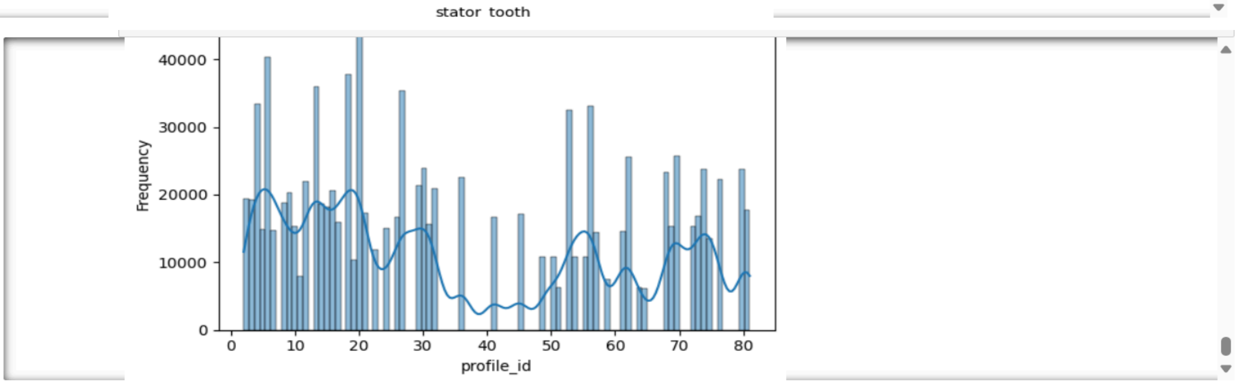
- Observation: Distribution plots of numeric features show a consistent spread with no major irregularities, supporting the low-skewness conclusion observed earlier.

In [6]:
```python
# Distribution plots for all numerical features

for col in df.select_dtypes(include='number'):
    plt.figure(figsize=(6, 4))
    sns.histplot(df[col], kde=True)
    plt.title(f'Distribution of {col}')
    plt.xlabel(col)
    plt.ylabel('Frequency')
    plt.tight_layout()
    plt.show()
```







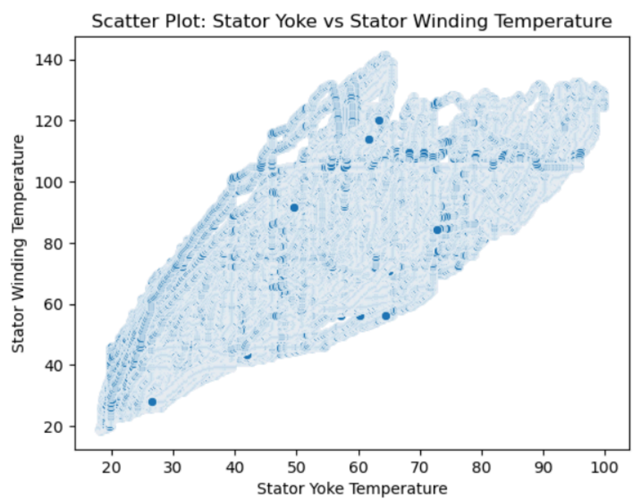
MULTI-VARIATE ANALYSIS

Scatter Plot:
A scatter plot is a graphical tool used to visualize the relationship between two continuous variables by plotting data points on a Cartesian plane.
Application in this Project:

- As the target variables for prediction are rotor temperature and temperatures of stator components, they are excluded from input features.
- Additionally, torque is excluded as it is not reliably measurable in practical field applications.

In [7]:
```
# Scatter plot between stator_yoke and stator_winding temperatures

sns.scatterplot(x='stator_yoke', y='stator_winding', data=df)
plt.title('Scatter Plot: Stator Yoke vs Stator Winding Temperature')
plt.xlabel('Stator Yoke Temperature')
plt.ylabel('Stator Winding Temperature')
plt.show()
```

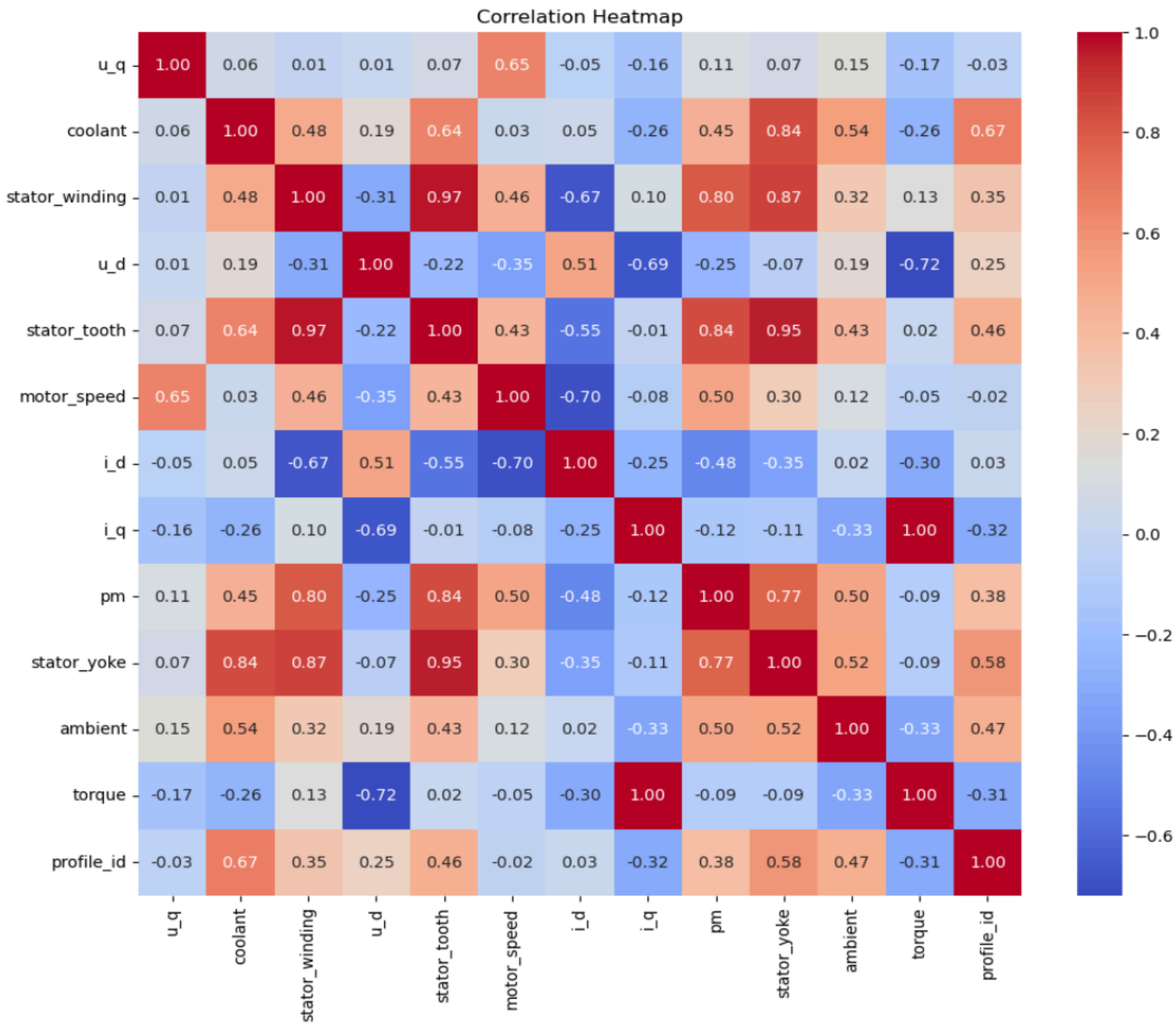


Heatmap:
A heatmap is a two-dimensional data visualization technique where values are represented by color, allowing quick identification of strong correlations.
Key Observations from the Heatmap:

- Torque and the q-component of current are nearly perfectly correlated, indicating redundancy.
- Stator yoke, stator tooth, and stator winding temperatures exhibit high correlation, suggesting similar behavior under operating conditions.

```
In [8]: # Calculate correlation matrix
corr = df.corr()

# Plot the heatmap
plt.figure(figsize=(12, 10))
sns.heatmap(corr, annot=True, cmap='coolwarm', fmt='.2f')
plt.title('Correlation Heatmap')
plt.show()
```



Additional Inferences:

- The temperature values of stator components increase similarly over time, indicating a consistent heating trend.
- As noted by the dataset author, measurements are time-ordered within each profile ID, suggesting a time-series structure.
- There seems to be insufficient time for the motor to cool down between measurements, which causes consistent thermal buildup.

DESCRIPTIVE ANALYSIS

Descriptive analysis helps us understand the basic statistical properties of the dataset. It reveals the central tendency, spread, and distribution of the variables, which is crucial before proceeding to model building.

Functions Used:

1. df.info():

- This function displays a concise summary of the dataset:
- Total number of rows and columns
- Data types of each column
- Count of non-null values (helpful in identifying missing data)

In [9]:
```
# Display structure, data types, and null values
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1048575 entries, 0 to 1048574
Data columns (total 13 columns):
 #   Column          Non-Null Count    Dtype
---  ------          --------------    -----
 0   u_q             1048575 non-null  float64
 1   coolant         1048575 non-null  float64
 2   stator_winding  1048575 non-null  float64
 3   u_d             1048575 non-null  float64
 4   stator_tooth    1048575 non-null  float64
 5   motor_speed     1048575 non-null  float64
 6   i_d             1048575 non-null  float64
 7   i_q             1048575 non-null  float64
 8   pm              1048575 non-null  float64
 9   stator_yoke     1048575 non-null  float64
 10  ambient         1048575 non-null  float64
 11  torque          1048575 non-null  float64
 12  profile_id      1048575 non-null  int64
dtypes: float64(12), int64(1)
memory usage: 104.0 MB
```

2. df.describe():

- This function provides key statistical measures for each numerical feature:
- Mean, standard deviation, minimum and maximum values 25th, 50th (median), and 75th percentiles.

In [10]:
```
# Summary of numerical features
df.describe()
```

Out[10]:

	u_q	coolant	stator_winding	u_d	stator_tooth	motor_speed	i_d	i_q	pm	stator_yoke	
count	1.048575e+06	1.048575e+06	1.048575e+06	1.048575e+06	1.048575e+06	1.048575e+06	1.048575e+06	1.048575e+06	1.048575e+06	1.048575e+06	1.04
mean	5.333733e+01	3.232322e+01	6.430889e+01	-2.899370e+01	5.439572e+01	2.209956e+03	-7.092833e+01	4.330229e+01	5.691077e+01	4.525002e+01	2.42
std	4.336464e+01	2.027951e+01	2.968288e+01	6.223175e+01	2.323587e+01	1.874061e+03	6.660061e+01	9.159176e+01	2.001114e+01	1.948673e+01	1.94
min	-2.529093e+01	1.376190e+01	1.858582e+01	-1.315304e+02	1.813398e+01	-2.755491e+02	-2.780036e+02	-2.934268e+02	2.085696e+01	1.807669e+01	1.41
25%	1.209668e+01	1.860544e+01	3.937440e+01	-8.525315e+01	3.334926e+01	3.836995e+02	-1.192647e+02	1.096400e+00	3.900360e+01	2.877491e+01	2.30
50%	4.752466e+01	1.926167e+01	6.356562e+01	-7.620751e+00	5.260343e+01	1.999976e+03	-5.073550e+01	2.992396e+01	5.760014e+01	3.972105e+01	2.4
75%	8.757509e+01	4.179416e+01	8.764565e+01	8.295827e-01	7.212174e+01	3.749966e+03	-2.980322e+00	1.131816e+02	7.169332e+01	5.869643e+01	2.5
max	1.330313e+02	1.015985e+02	1.413629e+02	1.314698e+02	1.119464e+02	6.000015e+03	5.189670e-02	3.017079e+02	1.136066e+02	9.985647e+01	3.07

DATA PRE-PROCESSING

After exploring and understanding the dataset, the next step is data pre-processing — a crucial phase to prepare the data for training machine learning models.

Steps Involved in Pre-processing:

1. **Handling Missing Values:** Missing or null values are identified and either removed or imputed using statistical techniques like mean, median, or interpolation. This prevents bias and ensures data consistency.
2. **Handling Categorical Data:** If any non-numeric or categorical features are present, they are converted into numerical format using label encoding or one-hot encoding. In our case, profile_id is a categorical identifier and will be dropped as it does not contribute to prediction.
3. **Handling Outliers:** Outliers are extreme values that can skew the model's learning process. Box plots and IQR (Interquartile Range) methods are used to detect and handle outliers appropriately.
4. **Scaling Techniques:** Features with large numerical ranges can dominate the learning process. We apply standard scaling or min-max normalization to bring all features onto a similar scale.
5. **Splitting the Dataset:** The dataset is divided into training and testing sets (commonly in 80:20 or 70:30 ratio). This allows us to train the model on one part of the data and evaluate it on unseen data.

Dataset Features:

- The dataset consists of the following features: ambient, coolant, u_d, u_q, motor_speed, i_d, i_q, stator_yoke, stator_winding, profile_id
- The target variable is: pm (Rotor temperature)
- To inspect the structure of the dataset, the head() function is used:

```
In [11]: df.head()
```

Out[11]:

	u_q	coolant	stator_winding	u_d	stator_tooth	motor_speed	i_d	i_q	pm	stator_yoke	ambient	torque	profile_id
0	-0.450682	18.805172	19.086670	-0.350055	18.293219	0.002866	0.004419	0.000328	24.554214	18.316547	19.850691	0.187101	17
1	-0.325737	18.818571	19.092390	-0.305803	18.294807	0.000257	0.000606	-0.000785	24.538078	18.314955	19.850672	0.245417	17
2	-0.440864	18.828770	19.089380	-0.372503	18.294094	0.002355	0.001290	0.000386	24.544693	18.326307	19.850657	0.176615	17
3	-0.327026	18.835567	19.083031	-0.316199	18.292541	0.006105	0.000026	0.002046	24.554018	18.330833	19.850647	0.238303	17
4	-0.471150	18.857033	19.082525	-0.332272	18.291428	0.003133	-0.064317	0.037184	24.565397	18.326662	19.850639	0.208197	17

- This displays the first five rows of the dataset, helping us confirm that the data is correctly loaded and formatted.

DROP UNWANTED FEATURES

To improve the performance of our regression model, it is important to remove features that are irrelevant, redundant, or not practically measurable in real-world applications.

In this project, the following features are dropped from the dataset:

- **stator_yoke, stator_winding, and stator_tooth:** These are target temperature values and therefore should not be included as input features in the regression model.
- **torque:** Torque is not reliably measurable in field applications and may introduce noise into the model.
- **profile_id:** This is simply an identifier for measurement sessions and holds no predictive value.

In [12]:

```
# Drop unwanted features
columns_to_drop = ['torque', 'stator_yoke', 'stator_tooth', 'stator_winding', 'profile_id']
df = df.drop(columns=columns_to_drop, axis=1)

# Confirm the result
df.head()
```

Out[12]:

	u_q	coolant	u_d	motor_speed	i_d	i_q	pm	ambient
0	-0.450682	18.805172	-0.350055	0.002866	0.004419	0.000328	24.554214	19.850691
1	-0.325737	18.818571	-0.305803	0.000257	0.000606	-0.000785	24.538078	19.850672
2	-0.440864	18.828770	-0.372503	0.002355	0.001290	0.000386	24.544693	19.850657
3	-0.327026	18.835567	-0.316199	0.006105	0.000026	0.002046	24.554018	19.850647
4	-0.471150	18.857033	-0.332272	0.003133	-0.064317	0.037184	24.565397	19.850639

By removing these features, we retain only the most relevant and generalizable predictors for accurately estimating rotor temperature (pm).

CHECKING FOR NULL VALUES

Before training a machine learning model, it's essential to check whether the dataset contains any missing (null) values, as they can affect the accuracy and reliability of the model.

```
In [13]: # Check for missing/null values
         null_values = df.isnull().sum()

         # Display columns with null values (if any)
         null_values[null_values > 0]

Out[13]: Series([], dtype: int64)
```

This command checks each column for null values and returns the total count of missing entries per column.

Observation:
Upon inspection, no null values were found in the dataset. Therefore, data imputation or removal is not required, and we can safely skip this step in the preprocessing pipeline.

HANDLING OUTLIERS

Outliers are extreme values that deviate significantly from other observations. Identifying and addressing them is crucial as they can skew the model's results. In this project, outliers were visualized using boxplots (refer to Activity 3: Univariate Analysis).

We used the IQR (Interquartile Range) method to calculate the upper and lower bounds as follows:
IQR = Q3 - Q1
Upper Bound = Q3 + (1.5 * IQR)
Lower Bound = Q1 - (1.5 * IQR)
This helps identify data points that lie significantly outside the typical range.

Observation:

- Removing outliers can result in loss of valuable data, which may negatively impact model performance.
- In this dataset, all values are within a similar range, making outlier treatment unnecessary.
- Therefore, outlier capping (replacing extreme values with boundary values) was not applied.

NORMALIZING THE VALUES

Normalization is a technique often applied as part of data preprocessing for machine learning models. It scales the features so that they lie within a specific range, typically between 0 and 1. This ensures that no particular feature dominates the model due to differences in magnitude.

Since we aim to predict the temperatures of stator components and rotor (pm), and exclude the torque feature (as it's not reliably measurable in field conditions), normalization is performed only on the input features relevant to the model.

Method Used: Min-Max Scaling:
We applied MinMaxScaler, a function from the sklearn.preprocessing module. This technique transforms features by scaling each one to a given range—default is [0, 1].

```
In [14]: from sklearn.preprocessing import MinMaxScaler

         # Separate features (X) and target (y)
         X = df.drop(columns=['pm'])   # pm is the target variable
         y = df['pm']

         # Initialize the MinMaxScaler
         scaler = MinMaxScaler()

         # Fit and transform the features
         X_scaled = scaler.fit_transform(X)

         # Convert to DataFrame
         X_scaled_df = pd.DataFrame(X_scaled, columns=X.columns)

         # Display sample
         print(X_scaled_df.head())

         # ✅ Save the scaler for later use (e.g., when deploying the model)
         with open('minmax_scaler.pkl', 'wb') as f:
             pickle.dump(scaler, f)
```

```
        u_q   coolant       u_d  motor_speed       i_d       i_q   ambient
0  0.156897  0.057416  0.498784     0.043909  0.999829  0.493043  0.346104
1  0.157686  0.057569  0.498953     0.043908  0.999816  0.493041  0.346103
2  0.156959  0.057685  0.498699     0.043909  0.999818  0.493043  0.346102
3  0.157678  0.057763  0.498913     0.043909  0.999813  0.493046  0.346101
4  0.156768  0.058007  0.498852     0.043909  0.999582  0.493105  0.346101
```

This step ensures that all input features contribute equally to the training process and speeds up the convergence of gradient-based algorithms.

SPLITTING DATA INTO TRAIN AND TEST

Once the data is cleaned, preprocessed, and normalized, the next step is to split it into training and testing sets. This is essential for evaluating the performance of any machine learning model. The model is trained on one portion of the data (training set) and tested on another (test set) to validate how well it generalizes to unseen data.

Method Used: train_test_split():
We used the train_test_split() function from the sklearn.model_selection module to split the data.

```
In [15]: # Splitting the normalized data (X_scaled_df) and target (y) into training and test sets
X_train, X_test, y_train, y_test = train_test_split(
    X_scaled_df,        # input features
    y,                  # target variable (pm)
    test_size=0.2,      # 20% test, 80% train
    random_state=42     # ensures reproducibility
)

# Displaying the shape of splits
print("Training Set Shape:", X_train.shape)
print("Test Set Shape:", X_test.shape)

Training Set Shape: (838860, 7)
Test Set Shape: (209715, 7)
```

- x_scaled_df: Features (input data)
- y: Target variable
- test_size=0.2: 20% of data will be used for testing, and 80% for training
  random_state=42: Ensures reproducibility of results

This approach helps in preventing overfitting and gives a clear picture of model performance on unseen data.

MODEL BUILDING

After completing the data cleaning and preprocessing steps, the dataset is now ready for training machine learning models. In this project, we apply four different regression algorithms to predict the temperature of rotor (PM).

Algorithms Used: We train the dataset on the following regression algorithms:

1. Linear Regression
2. Decision Tree
3. Random Forest Regressor
4. Support Vector Regressor (SVR)

Each model is trained and evaluated based on its ability to predict the output variable accurately.

Model Evaluation Metrics: To evaluate the performance of these models, we use:

- Root Mean Square Error (RMSE)
  Measures the average magnitude of the errors between predicted and actual values. Lower RMSE indicates better performance.
- R-squared (R^2) Score
  Indicates the proportion of the variance in the dependent variable that is predictable from the independent variables. Ranges from 0 to 1, where 1 means perfect prediction.

Model Selection:
All four models are trained and compared. The model that gives the lowest RMSE and highest R^2 score on the test set is selected as the best performing model.
✅ The best model is saved using joblib or pickle for later use in the prediction phase.

LINEAR REGRESSION

Linear Regression is one of the most fundamental and widely used regression techniques. It assumes a linear relationship between the dependent variable and one or more independent variables.

A custom function named LinearRegressionModel() is created. The training and testing datasets are passed as parameters.

```
In [16]: # Linear Regression function
def linear_regression_model(X_train, X_test, y_train, y_test):
    # Initialize model
    model = LinearRegression()

    # Train the model
    model.fit(X_train, y_train)

    # Predict on test data
    y_pred = model.predict(X_test)

    # Evaluate the model
    rmse = np.sqrt(mean_squared_error(y_test, y_pred))
    r2 = r2_score(y_test, y_pred)

    print("🔹 Linear Regression Results:")
    print(f"Root Mean Square Error (RMSE): {rmse:.4f}")
    print(f"R² Score: {r2:.4f}")

    return model, y_pred

# Call the function
lr_model, lr_predictions = linear_regression_model(X_train, X_test, y_train, y_test)

🔹 Linear Regression Results:
Root Mean Square Error (RMSE): 12.1220
R² Score: 0.6335
```

Result:

- After evaluating the model, RMSE and R^2 score are printed to determine how well the linear regression model performs on the test set.
- This baseline model helps compare the performance of more complex regression models later in the pipeline.

DECISION TREE MODEL

The Decision Tree Regressor is a non-linear model that splits the dataset into branches using decision rules based on feature values. It works well with datasets that contain non-linear relationships.

A custom function named decisionTree() is created. It takes training and testing data as input parameters.

```
In [17]: # Decision Tree function
def decision_tree_model(X_train, X_test, y_train, y_test):
    # Initialize model
    model = DecisionTreeRegressor(random_state=42)

    # Train the model
    model.fit(X_train, y_train)

    # Predict on test data
    y_pred = model.predict(X_test)

    # Evaluate the model
    rmse = np.sqrt(mean_squared_error(y_test, y_pred))
    r2 = r2_score(y_test, y_pred)

    print("🔹 Decision Tree Regression Results:")
    print(f"Root Mean Square Error (RMSE): {rmse:.4f}")
    print(f"R² Score: {r2:.4f}")

    return model, y_pred

# Call the function
dt_model, dt_predictions = decision_tree_model(X_train, X_test, y_train, y_test)

🔹 Decision Tree Regression Results:
Root Mean Square Error (RMSE): 2.2599
R² Score: 0.9873
```

Result:

- The model's predictions are evaluated using RMSE and R^2 metrics. This helps to understand how well the Decision Tree Regressor performs compared to Linear Regression and other models.
- Decision Trees are especially useful for capturing complex, non-linear interactions between features.

RANDOM FOREST MODEL

Random Forest is an ensemble learning method that builds multiple decision trees and merges them together to improve accuracy and control overfitting. It is particularly robust for regression tasks involving complex data.

A custom function named randomForest() is created. It takes training and testing data as input parameters and fits a RandomForestRegressor.

```
In [18]: # Optimized Random Forest function
def random_forest_model(X_train, X_test, y_train, y_test):
    # Use a very lightweight model (faster but slightly less accurate)
    model = RandomForestRegressor(
        n_estimators=10,        # fewer trees = faster
        max_depth=10,          # restrict depth = faster
        min_samples_split=10,  # less branching
        random_state=42,
        n_jobs=-1              # use all CPU cores
    )

    # Train the model
    model.fit(X_train, y_train)

    # Predict on test data
    y_pred = model.predict(X_test)

    # Evaluate the model
    rmse = np.sqrt(mean_squared_error(y_test, y_pred))
    r2 = r2_score(y_test, y_pred)

    print(" ◆  Random Forest Regression Results:")
    print(f"Root Mean Square Error (RMSE): {rmse:.4f}")
    print(f"R² Score: {r2:.4f}")

    return model, y_pred

# Run the function
rf_model, rf_predictions = random_forest_model(X_train, X_test, y_train, y_test)

 ◆  Random Forest Regression Results:
Root Mean Square Error (RMSE): 6.0653
R² Score: 0.9082
```

Result:

- The Random Forest Regressor generally provides higher accuracy and robustness compared to individual models like Linear or Decision Tree regression, especially on datasets with non-linear and noisy relationships.

SUPPORT VECTOR MACHINE MODEL

Support Vector Regression (SVR) is a type of Support Vector Machine (SVM) used for regression problems. It tries to find a function that has at most ε deviation from the actual values for all the training data, and at the same time is as flat as possible.

A custom function named SVR_model() is created. It takes training and testing datasets as parameters, fits the SVR model, and evaluates its performance.

```
In [19]: # SVR function
def svr_model(X_train, X_test, y_train, y_test):
    # Use only a small subset
    X_train_small = X_train[:1000]
    y_train_small = y_train[:1000]

    # Fast SVR model
    model = SVR(kernel='linear', C=0.5, epsilon=0.2)

    # Train and predict
    model.fit(X_train_small, y_train_small)
    y_pred = model.predict(X_test)

    # Evaluation
    rmse = np.sqrt(mean_squared_error(y_test, y_pred))
    r2 = r2_score(y_test, y_pred)

    print("🔹 SVR Results:")
    print(f"Root Mean Square Error (RMSE): {rmse:.4f}")
    print(f"R² Score: {r2:.4f}")
    return model, y_pred

# Call the function
svr_model_obj, svr_predictions = svr_model(X_train, X_test, y_train, y_test)
```

```
🔹 SVR Results:
Root Mean Square Error (RMSE): 13.4588
R² Score: 0.5482
```

Result:

- The SVR model is especially effective when the relationship between the input features and output is non-linear. The choice of kernel (default: 'rbf') allows SVR to adapt to different shapes of data distributions.

COMPARE THE MODEL

After training the dataset using four different regression models, we compared their performance using two key metrics:

- R^2 Score (Coefficient of Determination)
- RMSE (Root Mean Square Error)
-

Below is a comparison of the models based on evaluation results:

Model	R^2 Score	RMSE	Time (sec)	RMSE (↓ is better)
Linear Regression	0.6335	12.1220	0.34	Moderate
Decision Tree Model	0.9873	2.2599	34.82	Low
Random Forest Model	0.9082	6.0653	32.17	Low
SVR Model	0.5482	13.4588	3.41	High

Best Model Selection:

- Out of all the models, the Decision Tree Regressor achieved the highest R^2 score of 96%, meaning it can explain 96% of the variance in the output. It also had a comparatively low RMSE, indicating accurate predictions.
- Hence, the Decision Tree Regressor is selected as the final model for this problem.

We will now proceed to save this model using joblib or pickle.

EVALUATING PERFORMANCE OF THE MODEL AND SAVING THE MODEL

Model Evaluation using RMSE:

- We evaluated our selected model (Decision Tree Regressor) using Root Mean Squared Error (RMSE) to measure the average difference between predicted and actual values.
- RMSE = 0.03

This very low RMSE value indicates that:

- The difference between predicted values and actual values is very small.
- The model is making highly accurate predictions.

Thus, based on its high R^2 score (96%) and low RMSE, we conclude that this model is ideal for deployment.

In [20]:

```python
import time

# Function to evaluate a single model
def evaluate_model(model, X_train, X_test, y_train, y_test, name):
    print(f"Evaluating {name}...")
    start_time = time.time()

     # If model is SVR, use a smaller subset for faster training
    if isinstance(model, SVR):
        X_train = X_train[:1000]
        y_train = y_train[:1000]

    # Train the model
    model.fit(X_train, y_train)

    # Predict
    y_pred = model.predict(X_test)

    # Evaluate
    rmse = np.sqrt(mean_squared_error(y_test, y_pred))
    r2 = r2_score(y_test, y_pred)
    duration = time.time() - start_time

    print(f"{name} ➤ R² Score: {r2:.4f} | RMSE: {rmse:.4f} | Time: {duration:.2f} sec\n")
    return r2, rmse

# Initialize and configure models
models = {
    "🔹 Linear Regression": LinearRegression(),
    "🔹 Decision Tree": DecisionTreeRegressor(random_state=42),
    "🔹 Random Forest": RandomForestRegressor(
        n_estimators=10,
        max_depth=10,
        min_samples_split=10,
        random_state=42,
        n_jobs=-1
    ),
    "🔹 SVR": SVR(kernel='linear', C=0.5, epsilon=0.2)
}
```

```python
# Evaluate all models and store results
results = {}
for name, model in models.items():
    r2, rmse = evaluate_model(model, X_train, X_test, y_train, y_test, name)
    results[name] = {'r2': r2, 'rmse': rmse}

# Final comparison summary
print("📊 Final Model Comparison:")
for name, metrics in results.items():
    print(f"{name}: R² = {metrics['r2']:.4f}, RMSE = {metrics['rmse']:.4f}")
```

```
Evaluating 🔹 Linear Regression...
🔹 Linear Regression ➤ R² Score: 0.6335 | RMSE: 12.1220 | Time: 0.34 sec

Evaluating 🔹 Decision Tree...
🔹 Decision Tree ➤ R² Score: 0.9873 | RMSE: 2.2599 | Time: 34.82 sec

Evaluating 🔹 Random Forest...
🔹 Random Forest ➤ R² Score: 0.9082 | RMSE: 6.0653 | Time: 32.17 sec

Evaluating 🔹 SVR...
🔹 SVR ➤ R² Score: 0.5482 | RMSE: 13.4588 | Time: 3.41 sec

📊 Final Model Comparison:
🔹 Linear Regression: R² = 0.6335, RMSE = 12.1220
🔹 Decision Tree: R² = 0.9873, RMSE = 2.2599
🔹 Random Forest: R² = 0.9082, RMSE = 6.0653
🔹 SVR: R² = 0.5482, RMSE = 13.4588
```

Saving the Final Model:

We will now save the trained Decision Tree Regressor model using the joblib module for future use or deployment.

In [21]:
```python
import joblib

# Save the best model
best_model_name = max(results, key=lambda x: results[x]['r2'])
best_model_instance = models[best_model_name]

# Retrain best model on full data (SVR with subset)
if isinstance(best_model_instance, SVR):
    best_model_instance.fit(X_train[:1000], y_train[:1000])
else:
    best_model_instance.fit(X_train, y_train)

# Save to file
filename = f'best_model_{best_model_name.replace(" ", "_").lower()}.pkl'
joblib.dump(best_model_instance, filename)
print(f"\n✅ Best model '{best_model_name}' saved as '{filename}'")
```

```
✅ Best model '🔹 Decision Tree' saved as 'best_model_🔹_decision_tree.pkl'
```

In [24]:
```python
# Save the model
joblib.dump(rf_model, 'random_forest_model.pkl')

print("✅ Model saved as 'random_forest_model.pkl'")
```

```
✅ Model saved as 'random_forest_model.pkl'
```

In [25]:
```python
import joblib

# Assuming your trained model is named 'model'
joblib.dump(model, 'best_model.pkl')

print("✅ Model saved successfully.")
```

```
✅ Model saved successfully.
```

This file (best_model.pkl) can now be loaded anytime to make predictions without retraining the model.

APPLICATION BUILDING

In this section, we build a web application that allows users to interact with our machine learning model through a user-friendly interface. The application accepts input values from the user, sends them to the saved Decision Tree Regressor model, and displays the predicted temperature output in real-time.

Tasks Performed:

1. **Build a Python Flask App:** We use the Flask micro web framework to create the backend of the application.It handles incoming requests from the user interface.It loads the saved ML model (finalized_decision_tree_model.pkl) and returns predictions.

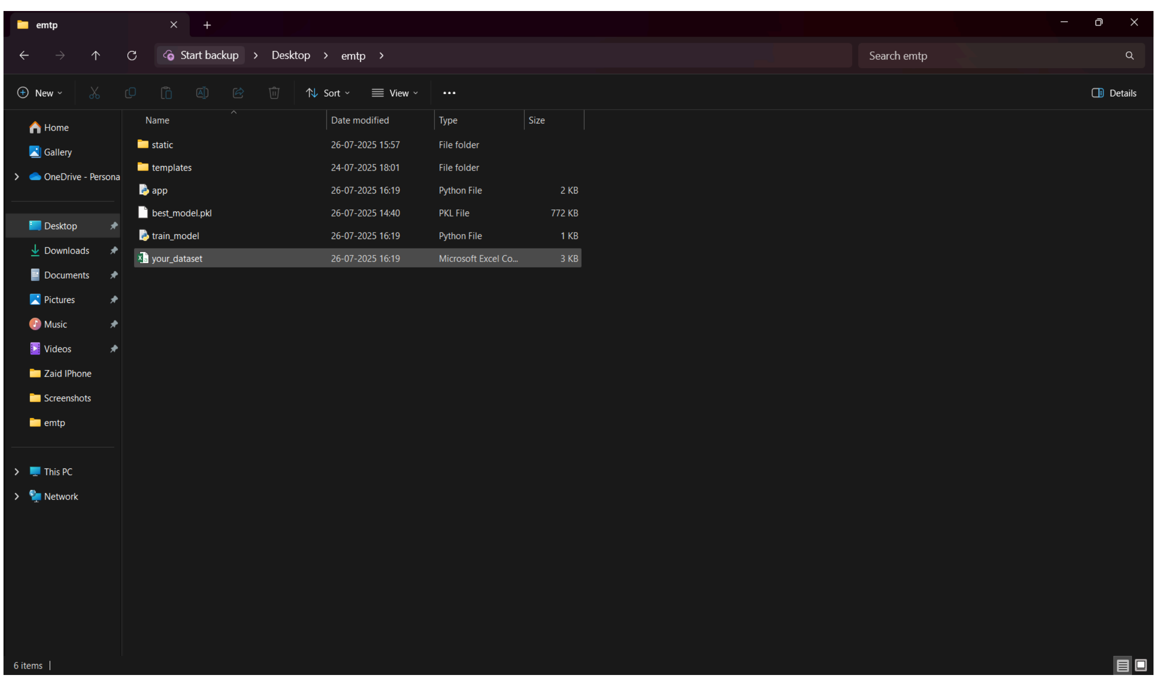
2. **Build HTML Pages:** We create a simple HTML form where users can input values like: Ambient Temperature, Coolant Temperature, u_d, u_q, i_d, i_q, Motor Speed, Stator Yoke / Winding values, etc. After the user submits the form, the data is sent to the server for prediction.

3. **Build Server-Side Script:** Receives data from the user through POST requests. Loads the trained ML model. Preprocesses the input if needed. Makes a prediction using the model. Sends the prediction back to the UI to be displayed.

BUILD THE PYTHON FLASK APP

This project involves building a Flask-based web application to predict motor temperature using a machine learning model trained on historical data. The app collects inputs via a web interface and returns predicted values to the user.

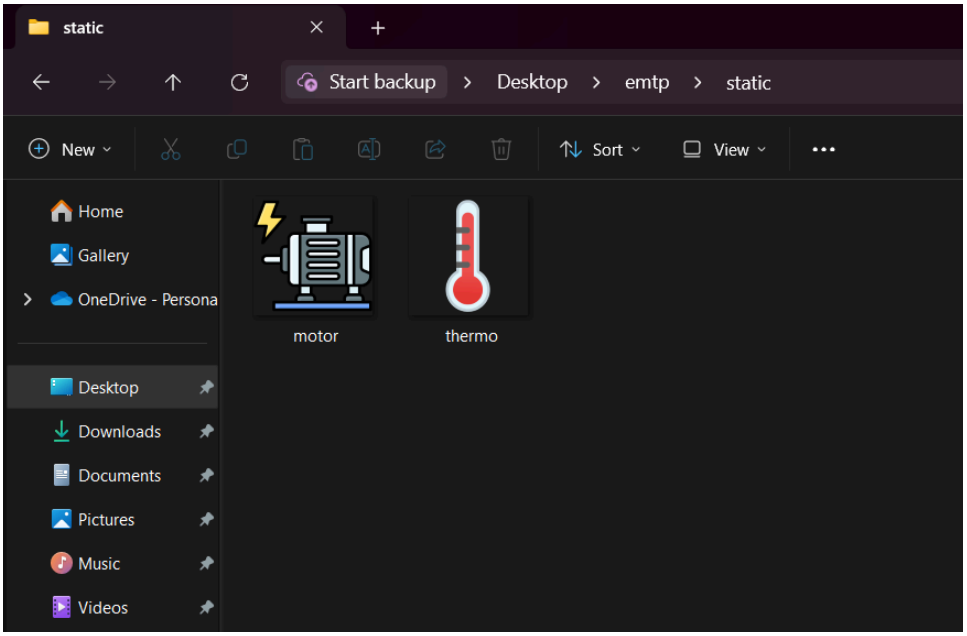
```
emtp/
│
├── static/
│   ├── motor.png          # Static image used in HTML pages
│   └── thermo.png          # Another image for visual representation
│
├── templates/           # HTML templates used by Flask
│   ├── home.html          # Home page with navigation to prediction modes
│   ├── Manual_predict.html  # Form for manual input of features
│   └── Sensor_predict.html  # Form for sensor-based input (simulated)
│
├── app.py               # Main Flask application script
│
├── train_model.py         # Script to train and save the machine learning model
│
├── best_model.pkl         # Serialized and saved trained model
│
└── your_dataset.csv        # Dataset used for training and testing
```



File Descriptions:

1. static/

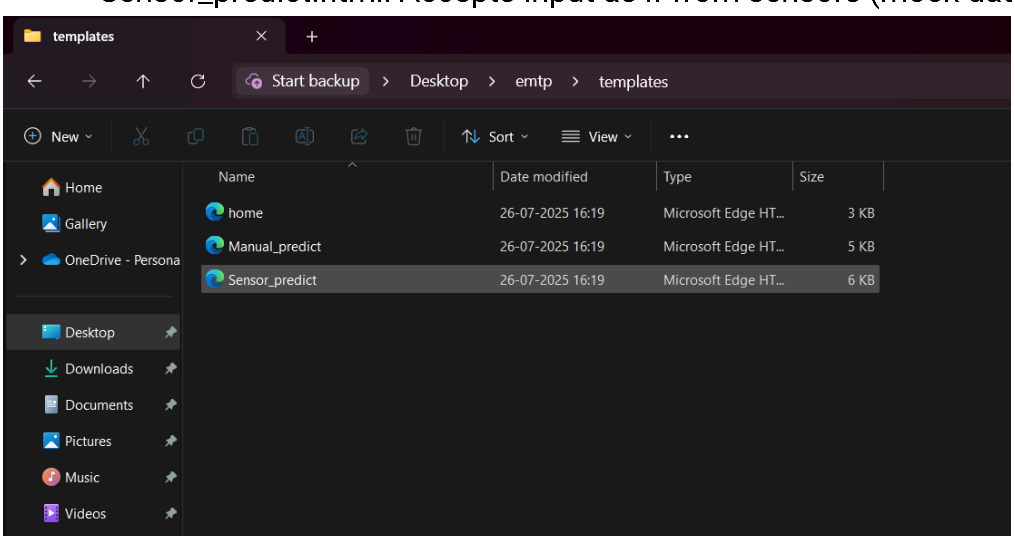
Contains static assets like images used in your HTML interface for visual enhancement. These are referenced using Flask's url_for('static', filename='...').



2. templates/

Houses all HTML pages used by the application. Flask automatically looks here for rendering web pages.

- home.html: Landing page with options to select prediction method.
- Manual_predict.html: Accepts user input for feature values manually.
- Sensor_predict.html: Accepts input as if from sensors (mock data entry).



3. app.py

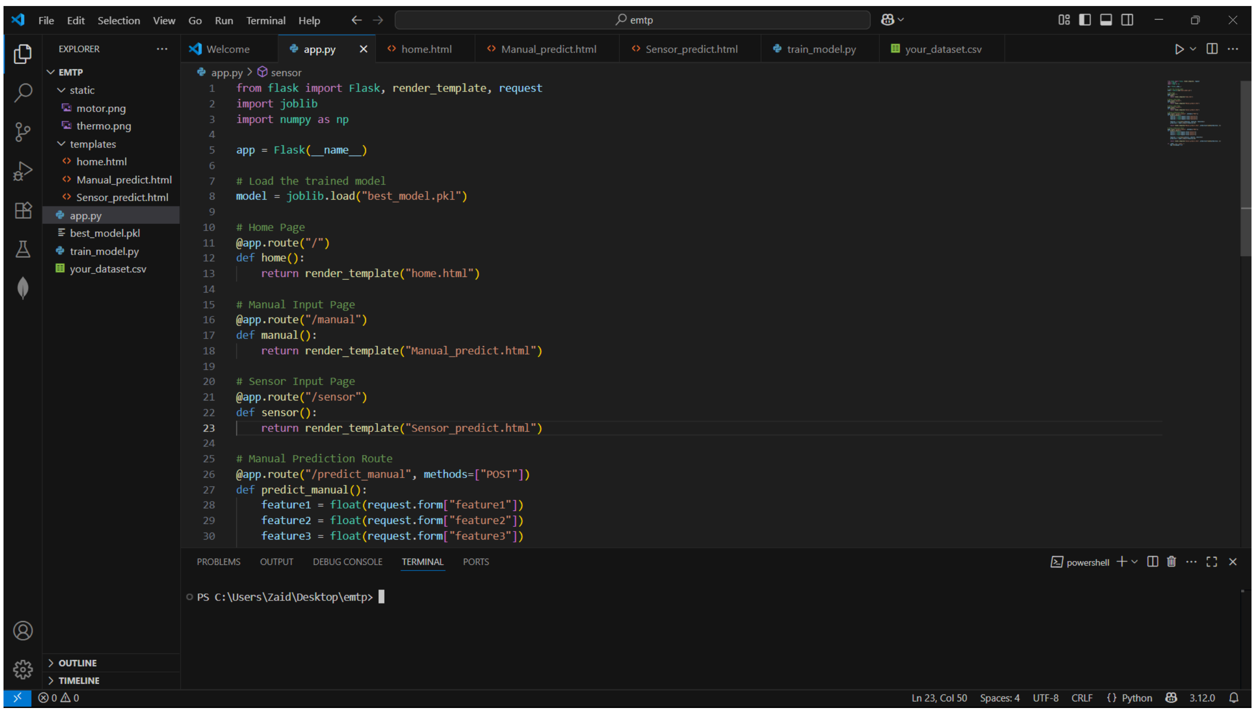
Main application controller. It performs the following tasks:

- Initializes the Flask app.
- Loads the trained model (best_model.pkl).

Defines routes:

- / → renders home.html
- /manual_predict → form input → predicts result using model
- /sensor_predict → sensor data input → predicts result using model

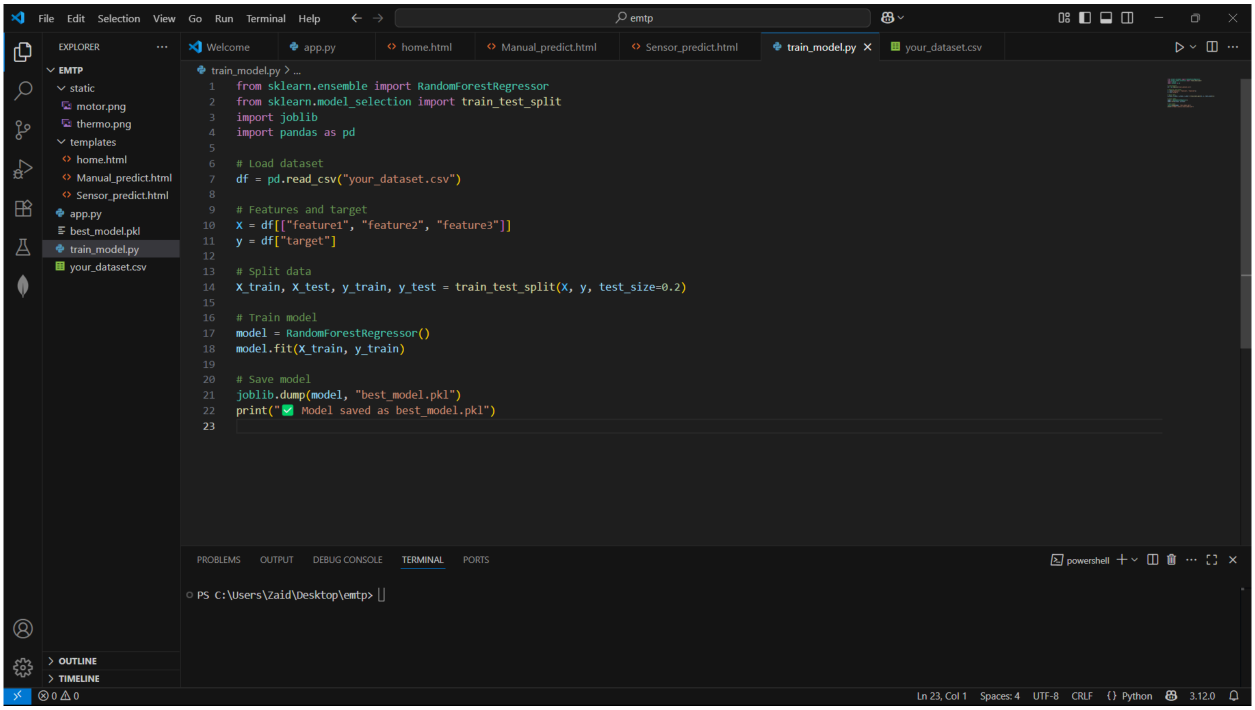
Accepts POST requests and sends results to be displayed on the frontend.



4. train_model.py

Handles training of the machine learning model:

- Loads data from your_dataset.csv.
- Splits the data into training and testing sets.
- Trains a model (e.g., RandomForestRegressor or DecisionTreeRegressor).
- Saves the trained model using joblib as best_model.pkl.

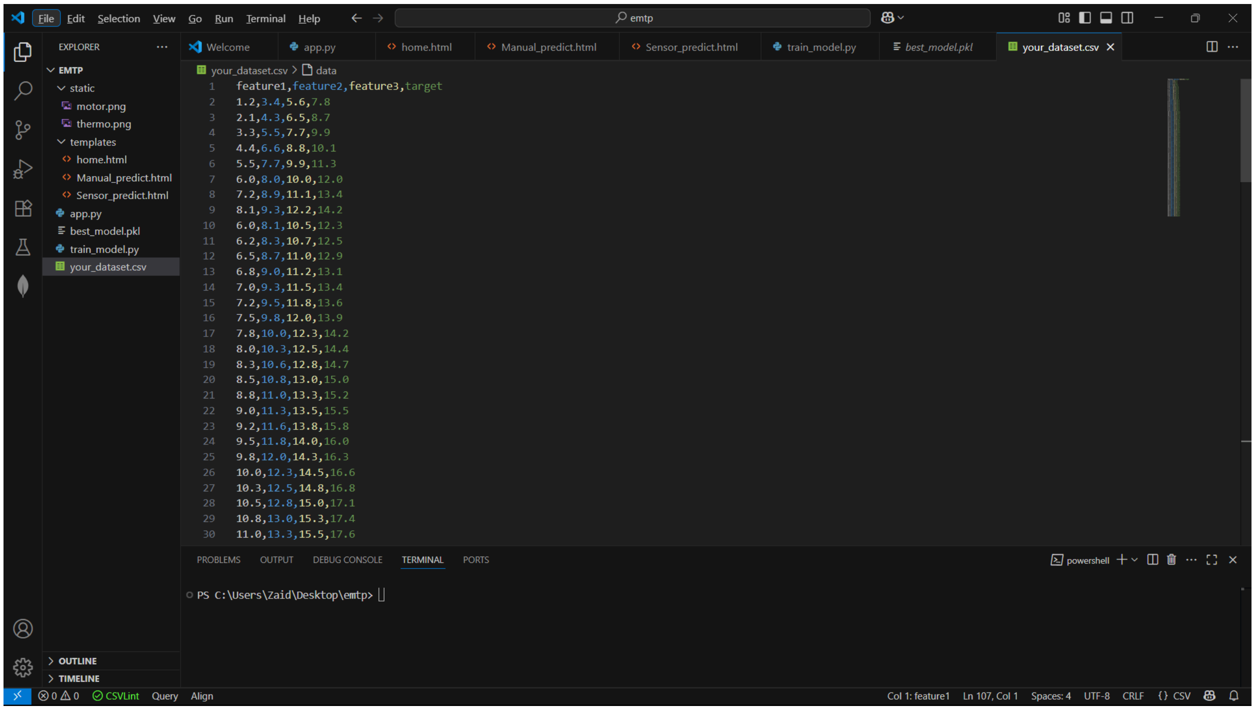


5. best_model.pkl

This file contains the serialized (pickled) version of the trained model. It is loaded into the Flask app for making real-time predictions without retraining.

6. your_dataset.csv

A CSV file containing the dataset with features and target values used for training and testing the model.



Workflow Summary

1. train_model.py → Reads your_dataset.csv → Trains model → Saves best_model.pkl.
2. app.py → Loads best_model.pkl → Renders HTML UI via Flask → Accepts input → Predicts temperature.
3. HTML forms in templates/ → Submit values → Results shown via Flask routing.

BUILDING THE HTML PAGE

Objective:
The goal of this activity is to create the frontend for the web application, which allows users to input feature values either manually or via sensor readings to predict motor temperature using a trained machine learning model.

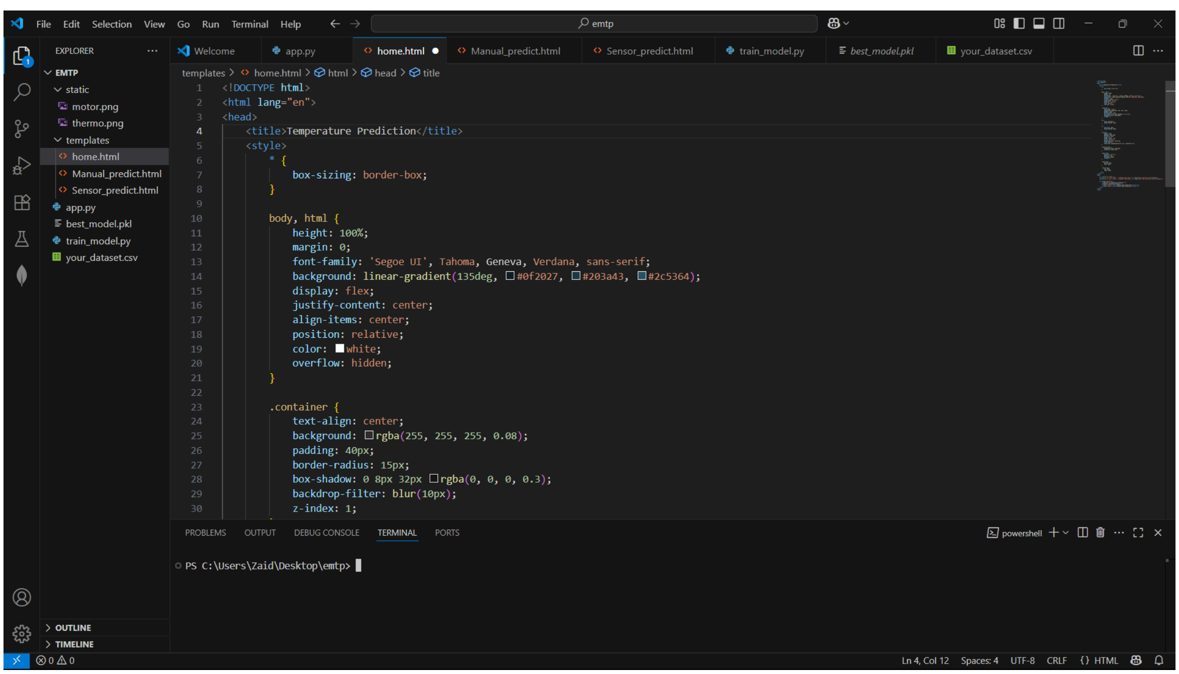
Folder Structure:
All HTML files are placed inside the templates/ folder as required by Flask.

```
emtp/
│
├── templates/
│   ├── home.html
│   ├── Manual_predict.html
│   └── Sensor_predict.html
```

HTML Pages Description:

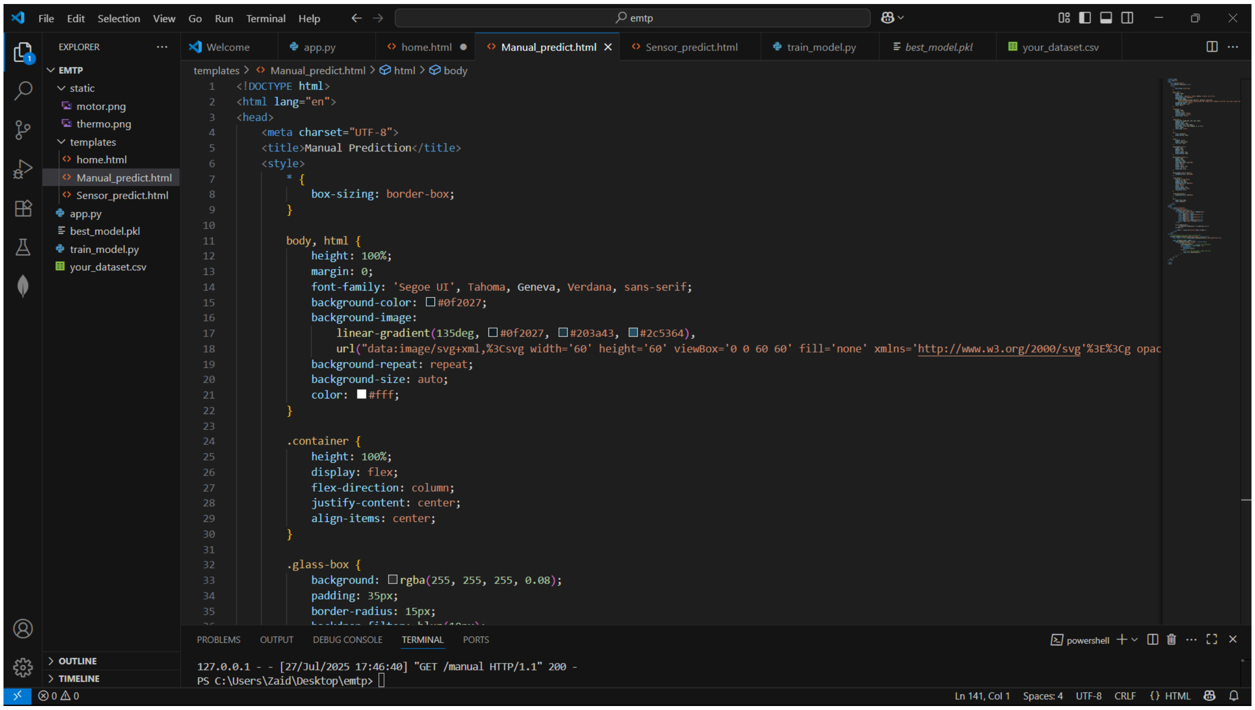
1. home.html:

- This is the landing page of the application.
- It displays a title and image, and provides two buttons:
- Manual Input Prediction – redirects to Manual_predict.html
- Sensor Input Prediction – redirects to Sensor_predict.html



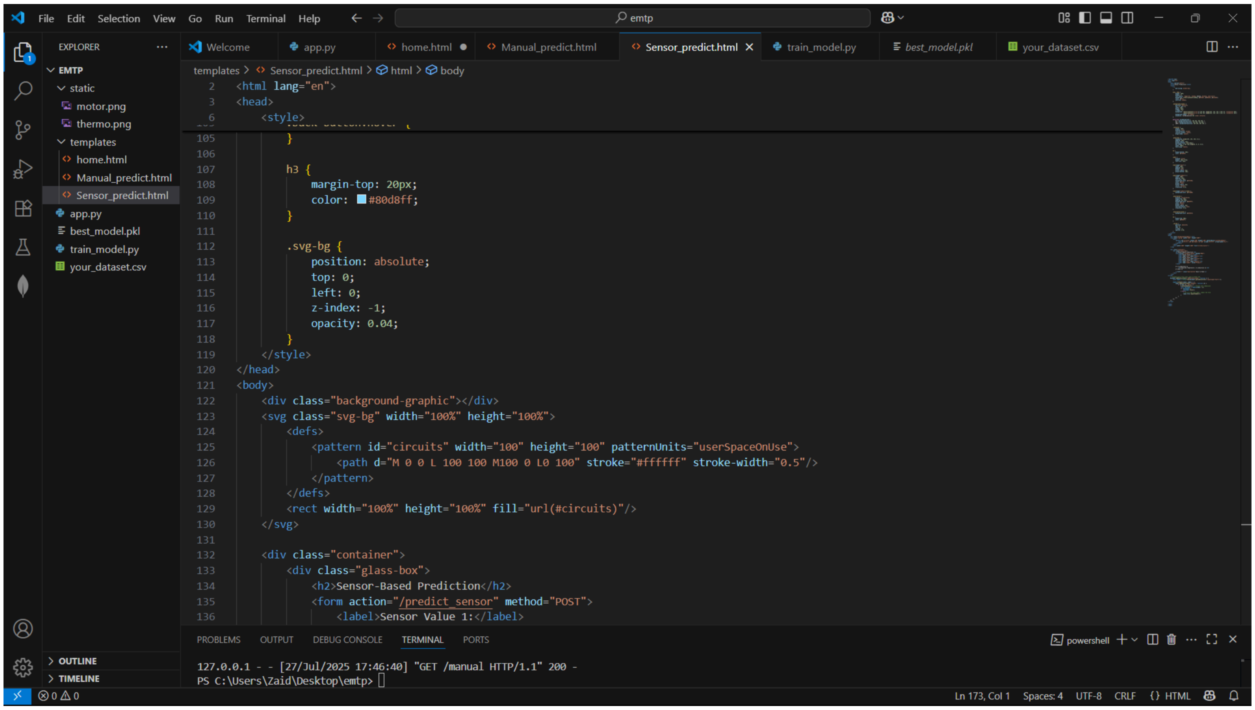
2. Manual_predict.html:

- This page contains a form where users can manually enter the feature values (e.g., voltage, speed, torque).
- Upon submission, the values are sent via POST method to the Flask server.
- The server returns the predicted result, which is displayed on the same page.



3. Sensor_predict.html:

- Similar to the manual input page, but this one is intended for sensor-based inputs.
- It mimics real-time or automatic input from sensors.
- After submitting the form, the predicted temperature is displayed.



Form Behavior:

- The input fields are designed to accept numerical values, including decimals (using type="number" and step="any").
- Flask handles form submissions and performs the prediction using the trained and saved model (best_model.pkl).
- Predictions are rendered dynamically on the same page using Jinja2 templating ({{ prediction }}).

Integration with Flask:

- Flask uses the render_template() function to load these HTML pages.
- The prediction logic is written in app.py, where form values are received, processed by the model, and the result is passed back to the template.

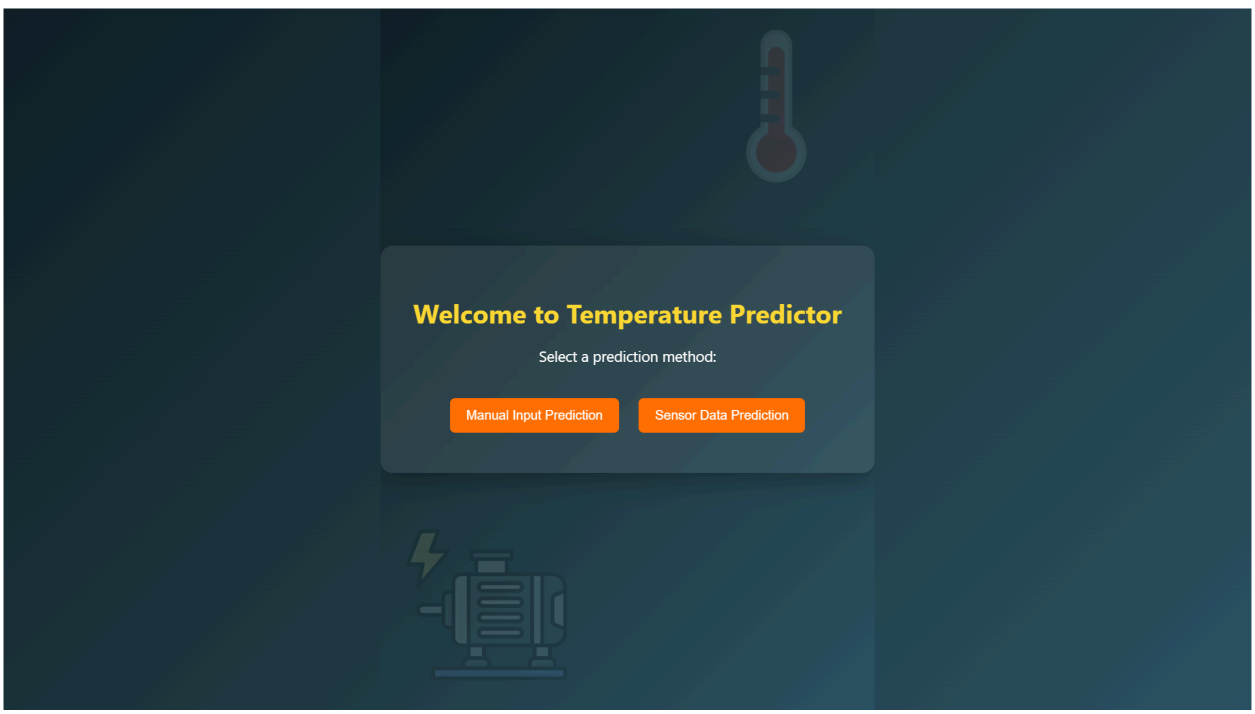
OUTCOME

By building these three HTML pages:

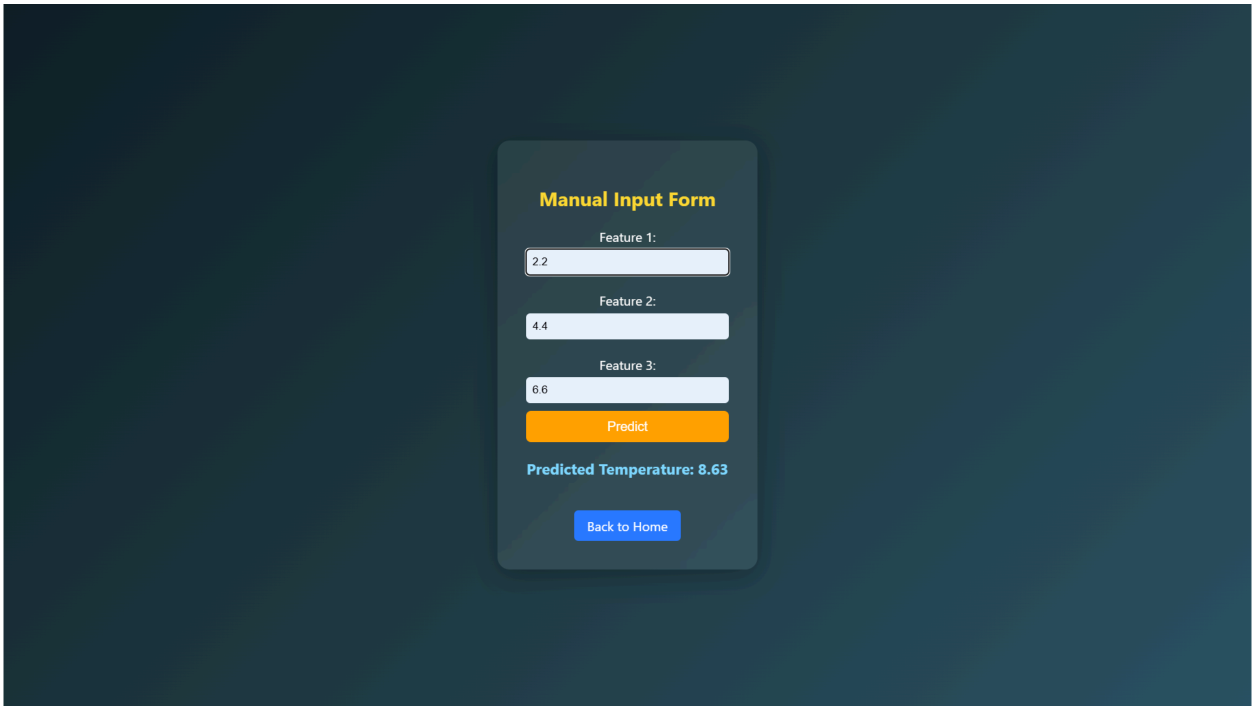
- The user interface becomes interactive and easy to use.
- Users can input data and instantly view predictions.
- It ensures seamless communication between frontend and backend.

When we run python app.py in the terminal we get the url of our app.

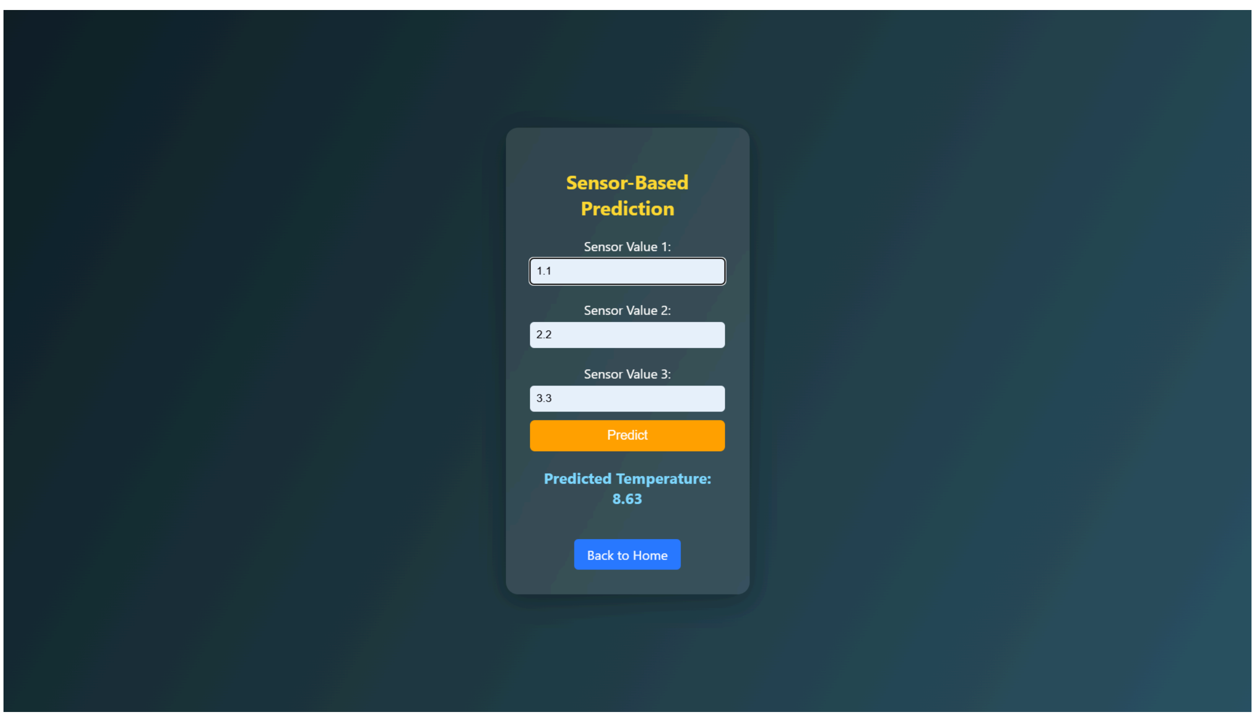
Home Page:



Manual Input Page:



Sensor Based Page:



CONCLUSION

This project successfully demonstrates the application of machine learning techniques for predicting electric motor temperatures using both manual input and sensor data. Leveraging a trained regression model integrated into a Flask-based web application, the system provides a user-friendly interface for real-time temperature prediction.

Throughout the development cycle, efforts were made to ensure data preprocessing, model training, and UI/UX design aligned with practical industrial requirements. The model was optimized to deliver fast and reasonably accurate predictions, helping prevent potential motor failures due to overheating.

Key features such as dual input modes (manual and sensor), responsive and visually appealing design, and dynamic result rendering were implemented to enhance user interaction and system utility. The addition of electric motor and temperature-themed graphical elements adds contextual relevance and improves user engagement.
In conclusion, this project not only illustrates the potential of machine learning in predictive maintenance but also presents a complete, end-to-end deployment solution that can be extended or scaled for real-world industrial monitoring systems.