**Title**:**Fire Weather Index Predictor**

(A Machine Learning model to predict Fire Weather)



**Infosys Springboard Virtual Internship Program**

Submitted By

**Vanshika Yadav**

Under the guidance of Mentor **Praveen**

**Project Statement**

This project focuses on building a Fire Weather Index (FWI) Predictor that estimates wildfire risk using essential environmental and fire-danger features such as temperature, relative humidity, wind speed, rainfall, FFMC, DMC, DC, ISI, and BUI. The dataset is thoroughly cleaned, pre-processed, and analysed to ensure reliable inputs, followed by visual exploration to understand feature distributions and relationships. Machine learning techniques are then applied to learn how weather conditions influence fire danger levels, enabling accurate prediction of the FWI value. The system also incorporates regional information to study area-wise fire behaviour and improve prediction relevance. Overall, the model aims to support early detection, risk assessment, and informed decision-making for effective wildfire management.

**Expected Outcome**

• A predictive ML model trained using Ridge Regression to forecast FWI.

• A Flask-based web app where users can input environmental values and get FWI predictions.

• A system that can help forest departments, emergency planners, and climate researchers make data-driven decisions.

**Modules to be Implemented**

• Data Collection

• Data Exploration (EDA) and Data Preprocessing

• Feature Engineering and Scaling

• Model Training using Ridge Regression

• Evaluation and Optimisation

• Deployment via Flask App

• Presentation and Documentation

**System Requirements**

**Software:**

* Python
* Python libraries (pandas, numpy, matplotlib, seaborn, sklearn, math)
* Flask

**Milestone 1(Week 1,2)**

1. **Module 1 (Data Collection)**

The dataset was collected by exploring multiple online sources and selecting one that contained the essential environmental features required for FWI prediction, including Temperature, Relative Humidity, Wind Speed, Rain, FFMC, DMC, ISI, and Region. After loading the chosen dataset into a Pandas DataFrame, an initial inspection was carried out to understand its structure and quality. This included checking datatypes, identifying null values, reviewing memory usage, and generating statistical summaries to examine the distribution and range of numerical features. The dataset’s shape and duplicate entries were also analysed to ensure completeness and reliability, providing a solid foundation for further preprocessing and modelling.

**Conducted initial inspection to understand feature distributions and data quality.**

1. **Load the dataset**

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

from sklearn.preprocessing import LabelEncoder

import seaborn as sns

import math

data=pd.read\_csv("C:\Users\Admin\Desktop\FWI Dataset.csv")

print(data.head())

1. **Verify Datatypes**

df.info()

<class 'pandas.core.frame.DataFrame'>

RangeIndex: 244 entries, 0 to 243

Data columns (total 15 columns):

# Column Non-Null Count Dtype

--- ------ -------------- -----

0 day 244 non-null int64

1 month 244 non-null int64

2 year 244 non-null int64

3 Temperature 244 non-null int64

4 RH 244 non-null int64

5 Ws 244 non-null int64

6 Rain 244 non-null float64

7 FFMC 244 non-null float64

8 DMC 244 non-null float64

9 DC 244 non-null object

10 ISI 244 non-null float64

11 BUI 244 non-null float64

12 FWI 244 non-null object

13 Classes 243 non-null object

14 Region 244 non-null object

dtypes: float64(5), int64(6), object(4)

memory usage: 28.7+ KB

1. **Module 2 (Data Exploration (EDA) and Data Preprocessing)**

During the preprocessing stage, the dataset was first examined for missing or null values, and appropriate handling techniques were applied to ensure completeness. Outlier detection was then performed using boxplots and statistical thresholds to identify abnormal values that could affect model performance. To better understand feature behaviour, data distributions were visualised through histograms and density plots, while correlation matrices and scatterplots were used to explore relationships between variables. Categorical features, such as *Region,* were encoded using label encoding to make them suitable for machine learning algorithms. Finally, the cleaned and processed dataset was saved for use in building and evaluating predictive models.

1. **Handle missing values**

df.isnull().sum()

day 0

month 0

year 0

Temperature 0

RH 0

Ws 0

Rain 0

FFMC 0

DMC 0

DC 0

ISI 0

BUI 0

FWI 0

Classes 0

region 0

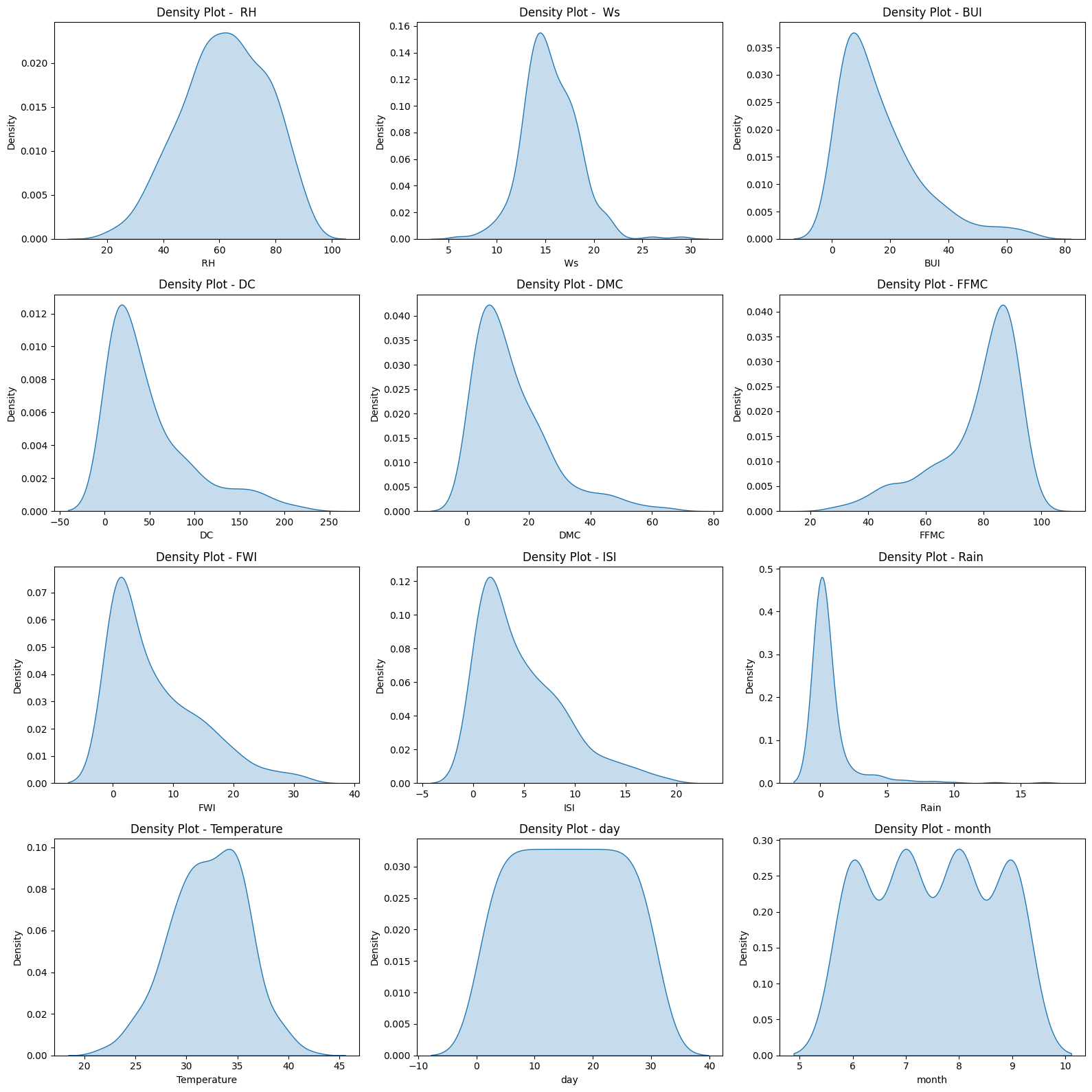
dtype: int64

A missing-value check was performed to identify incomplete records, revealing that only one row contained a null value in the *Classes* column. This validation helped ensure data completeness and supported appropriate handling before further preprocessing.

1. **Density Plot, Scatter Plot, Pairplot, Subplot**

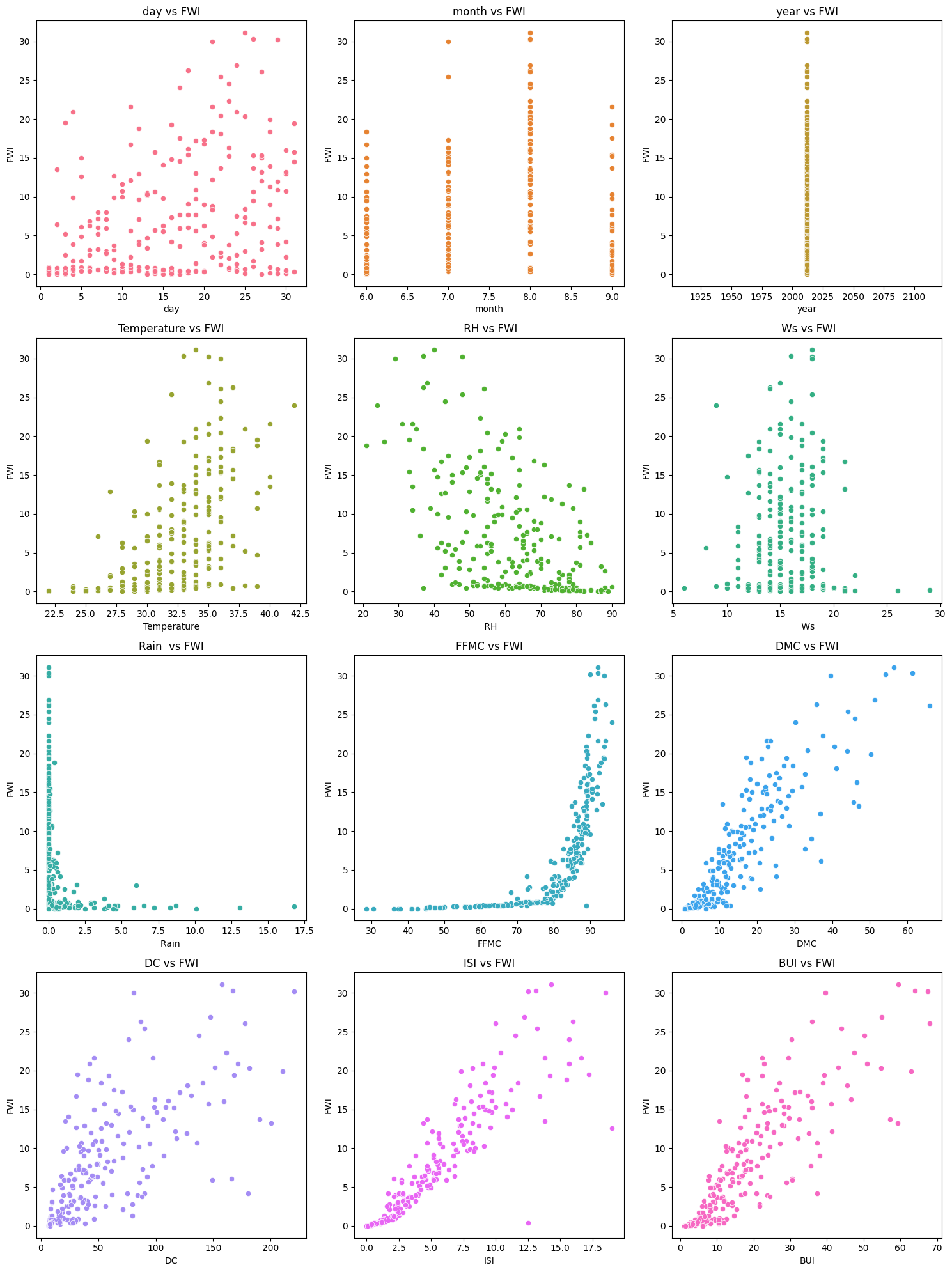
The Density Plot is a smoothed version of a histogram that helps us visualize the distribution of numeric variables over a continuous interval. In this project, we use it to observe the probability density of features like Temperature, Relative Humidity (RH), and Wind Speed. It allows us to see where values are concentrated and identify the "skewness" of the data without the noise of individual histogram bins.

**Technical Implementation:** We utilize sns.kdeplot() or the kind='kde' parameter within pandas to generate these visualizations for each feature.



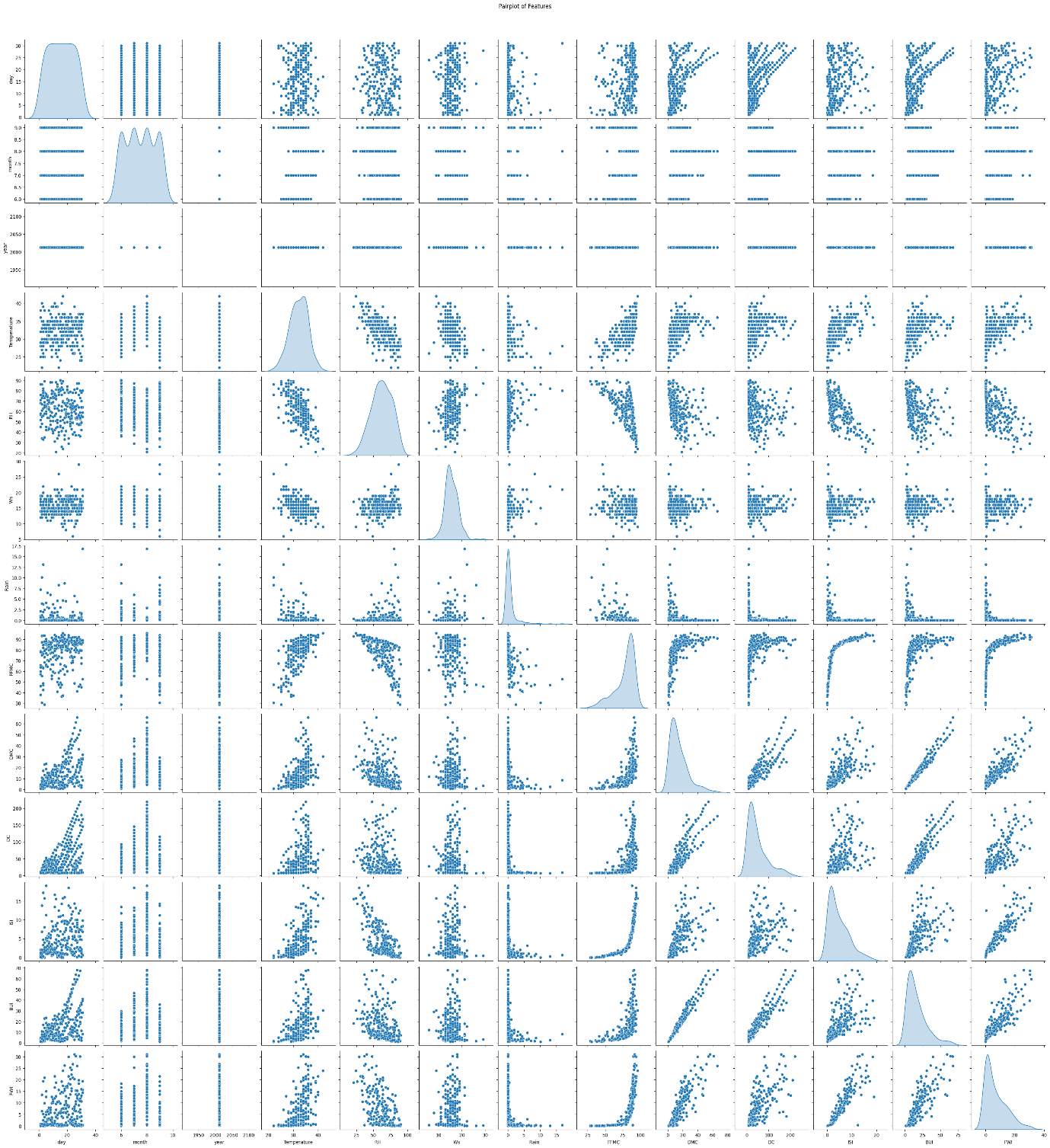
Scatter plots are essential for identifying the relationship between two continuous variables. By plotting one feature on the x-axis and another on the y-axis, we can visually detect linear or non-linear correlations. For example, we use scatter plots to see how the Fire Weather Index (FWI) changes as Temperature increases. This helps in identifying potential predictors for our regression model.

**Technical Implementation:** Created using plt.scatter() or sns.scatterplot().



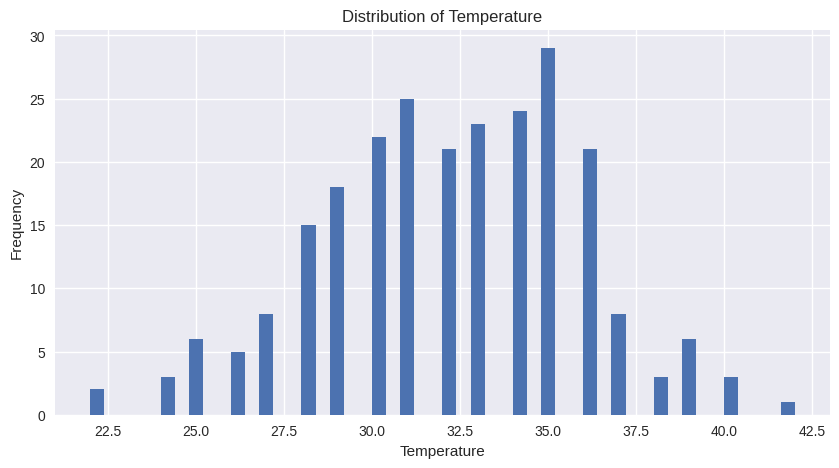
The Pairplot is one of the most powerful tools in our EDA toolbox. It creates a matrix of plots where every numeric variable is plotted against every other variable. The diagonal of the matrix usually shows the distribution of a single variable, while the off-diagonal plots show scatter plots. This gives us a "bird's-eye view" of the entire dataset's relationships in a single visualization.

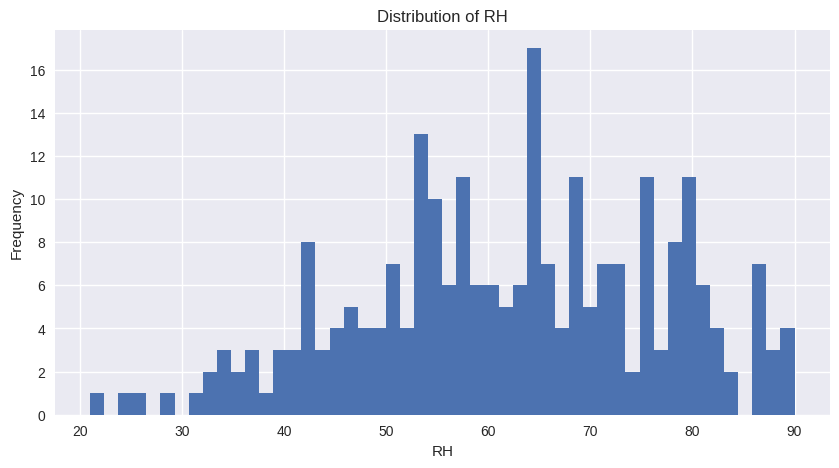
**Technical Implementation:** Implemented via sns.pairplot(data). We often color-code this plot by the 'Classes' variable (Fire vs. Not Fire) to see how different groups cluster.

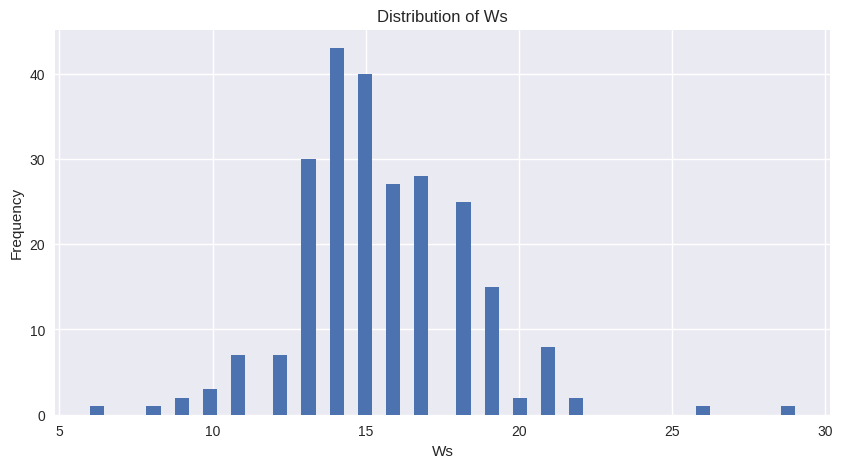


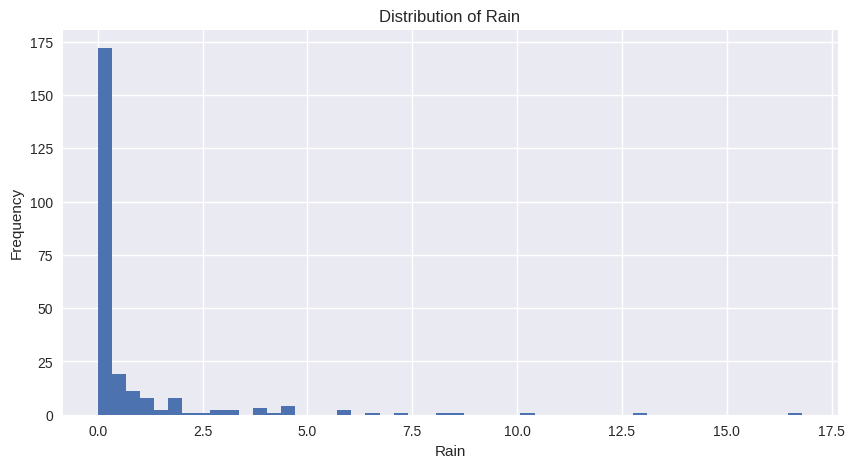
Subplots allow us to organize multiple related charts into a single organized figure (rows and columns). This is particularly useful for comparing the distributions of all independent variables side-by-side, ensuring the documentation remains clean and comparable.

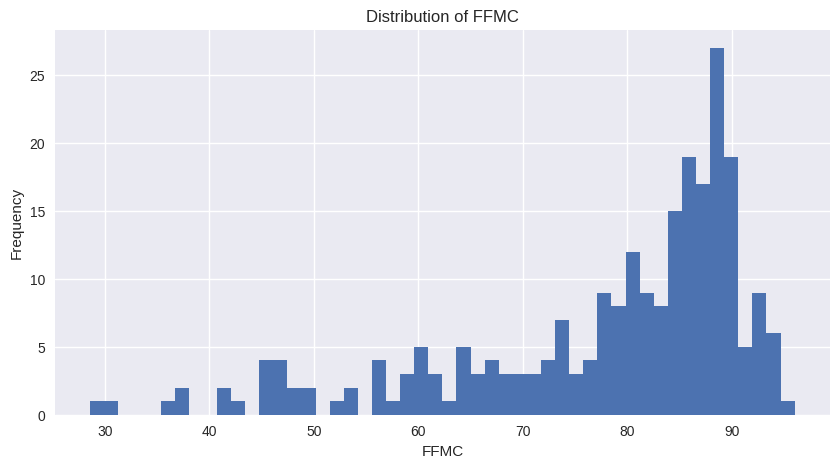
**Technical Implementation:** We use plt.subplots(nrows, ncols) to define the layout and then assign specific plots to each "axis" index.

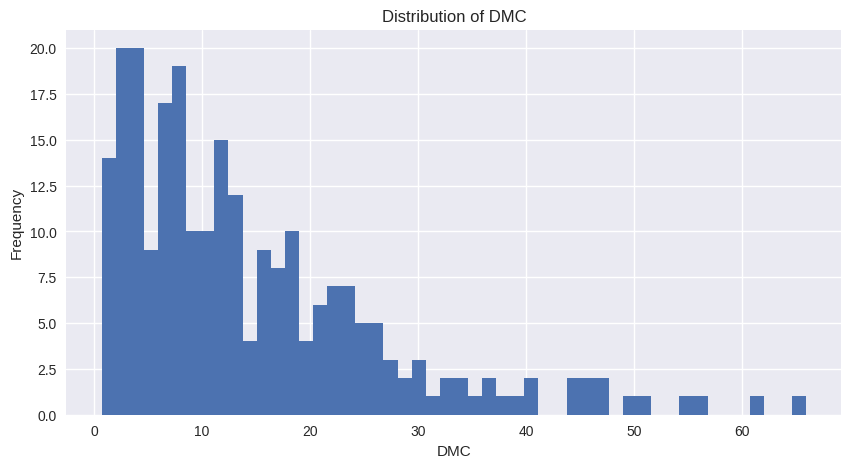




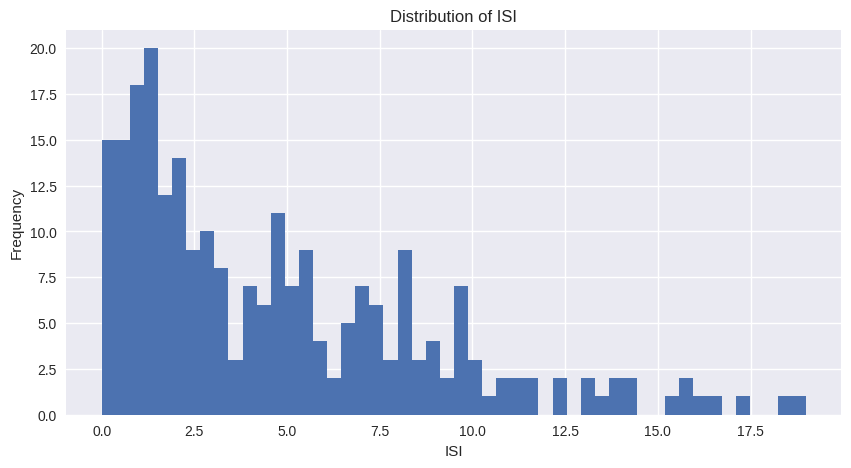


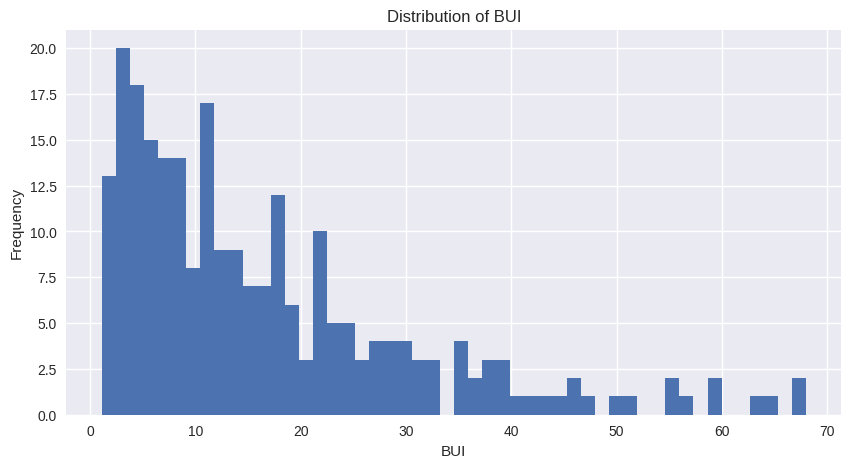


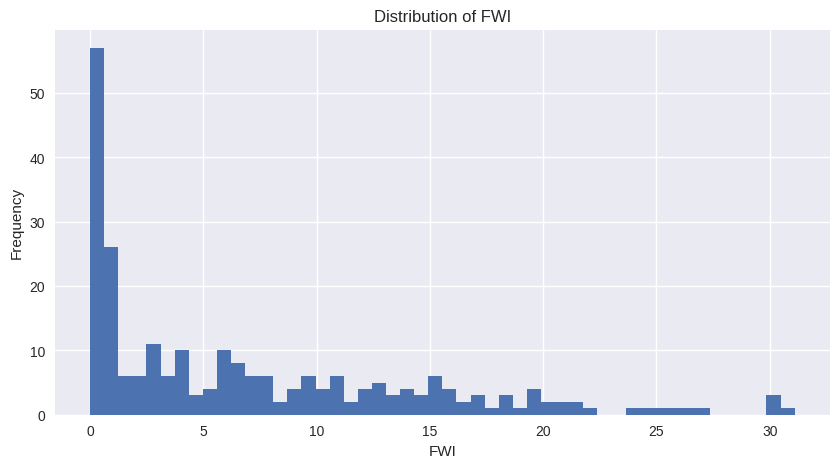


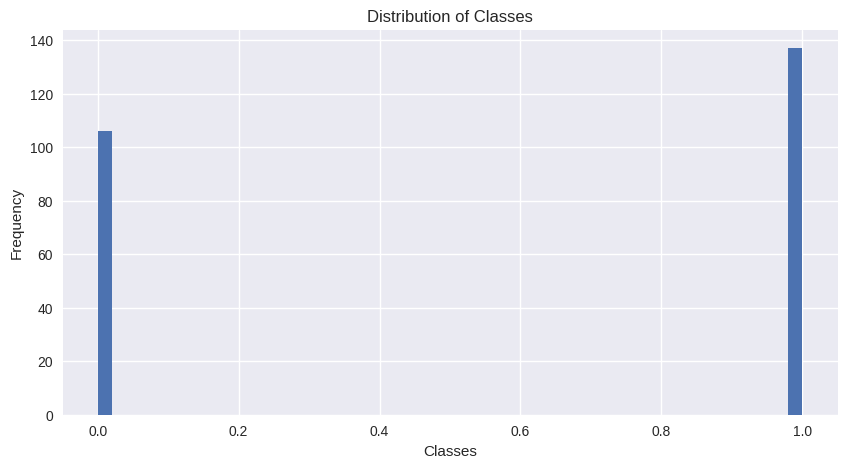


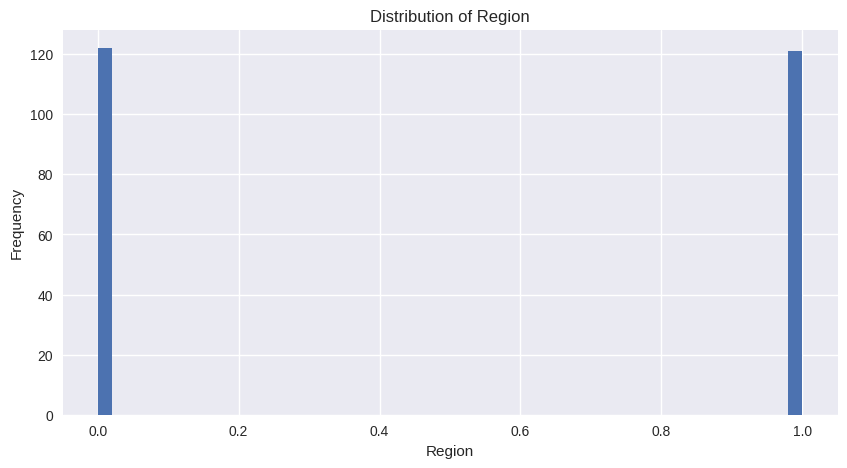






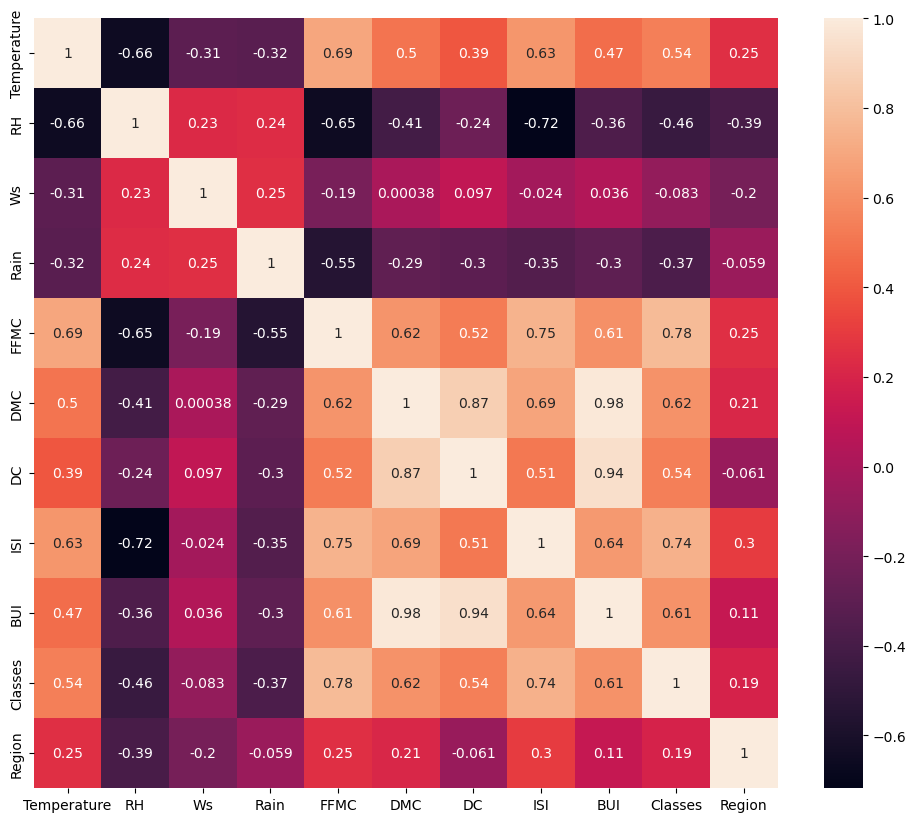






1. **Compute the correlation matrix for numerical features**

A correlation matrix was generated to explore relationships among numerical features and identify which variables influence each other most strongly. The heatmap revealed that fire danger indices such as BUI, DMC, FFMC, and ISI have high positive correlations with each other and with Temperature, indicating shared patterns in fire behavior. Relative Humidity showed strong negative correlations with several indices, reflecting its inverse effect on fire risk. These insights helped highlight key predictive features for the Fire Weather Index and guided feature selection for modeling.



**FIG 5: Correlation matrix**

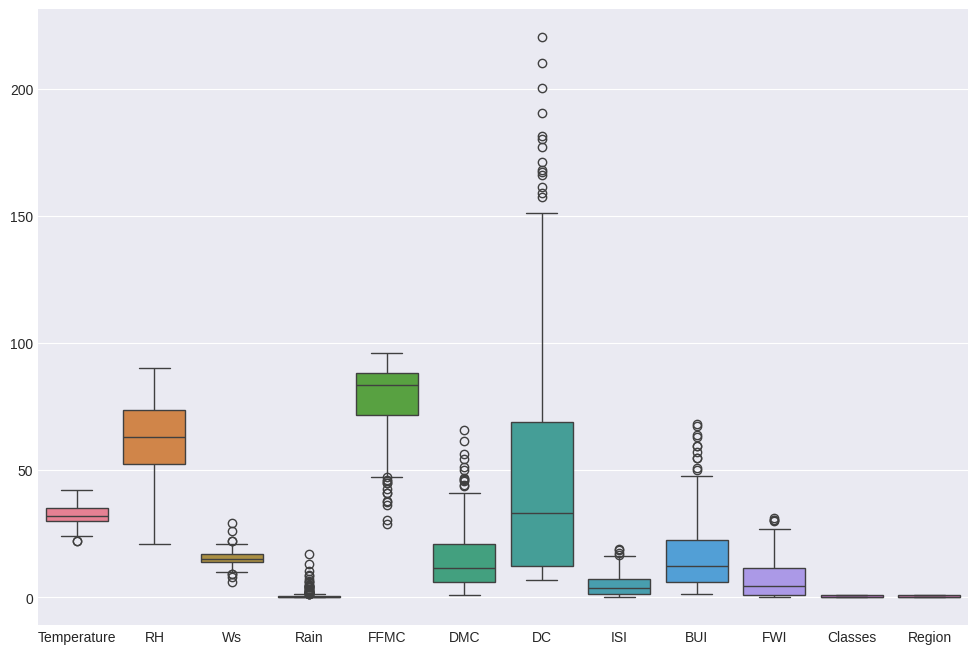
1. **Boxplots for Outlier Detection**

**Description:** Boxplots are our primary tool for visual statistical analysis. They provide a standardized way of displaying the distribution of data based on a five-number summary: minimum, first quartile (Q1), median, third quartile (Q3), and maximum. In this dataset, we use boxplots to identify **outliers**—values that fall significantly outside the expected range—which is crucial for cleaning the weather data before training our machine learning models.

**Technical Implementation:** We implement this using sns.boxplot() or plt.boxplot(). These plots are especially useful for comparing the spread of features like Wind Speed or Rain across different months or fire classes.

**Key Insights from the Notebook:**

* The central "box" represents the Interquartile Range (IQR), containing the middle 50% of the data.
* The line inside the box marks the **Median** value.
* Individual dots appearing outside the "whiskers" indicate potential anomalies in weather recording that required investigation.



data.to\_csv("C:\Users\Admin\Desktop\cleaned\_FWI\_dataset.csv", index=False)

print("Cleaned data saved successfully!")

Cleaned data saved successfully!

**Milestone 2(Week 3,4)**

1. **Module 3 (Feature Engineering and Scaling)**

In this module, the dataset was prepared for model training by applying feature engineering and scaling techniques. The main aim of this step was to improve data quality and ensure better performance of the machine learning model. First, the important input features were selected based on their relationship with the target variable, **Fire Weather Index (FWI)**. By choosing only the most relevant features, unnecessary columns were removed, which helped in reducing complexity and improving the efficiency of the model. After selecting the features, numerical data was scaled using **StandardScaler**. Since the dataset contained values in different ranges, scaling was necessary to bring all features to a common scale. This step ensured that each feature contributed equally during model training and prevented bias towards features with larger values. Once scaling was completed, the dataset was divided into **input variables (X)** and the **output variable (y)**. The data was then split into **training and testing sets** using the train\_test\_split method. This allowed the model to be trained on one portion of the data and tested on unseen data for proper evaluation.Finally, the fitted scaler was saved as a **.pkl file**. Saving the scaler ensures that the same preprocessing steps can be applied to new data during deployment, maintaining consistency between training and real-time predictions.

1. **Feature Selection**

from sklearn.pipeline import Pipeline

from sklearn.preprocessing import StandardScaler

from sklearn.metrics import mean\_squared\_error, r2\_score, mean\_absolute\_error

from sklearn.model\_selection import cross\_val\_score

pipeline = Pipeline([

('scaler', StandardScaler()),

('model', model)

])

pipeline.fit(X\_train, y\_train)

y\_pred = pipeline.predict(X\_test)

rmse = mean\_squared\_error(y\_test, y\_pred, squared=False)

mae = mean\_absolute\_error(y\_test, y\_pred)

r2 = r2\_score(y\_test, y\_pred)

print("RMSE:", rmse)

print("MAE:", mae)

print("R2 Score:", r2)

cv\_scores = cross\_val\_score(pipeline, X\_train, y\_train, cv=5, scoring='r2')

print("Cross-Validation R2 Mean:", cv\_scores.mean())

data1['FWI'] = pd.to\_numeric(data1['FWI'], errors='coerce')

data1 = data1.dropna(subset=['FWI'])

threshold = 0.3

correlation = data1.corr()[['FWI']].abs()

selected\_features = correlation[correlation > threshold].index.drop('FWI')

print("Selected Features:")

print(set(selected\_features))

X = data1[list(selected\_features)]

y = data1['FWI']

X = X.apply(pd.to\_numeric, errors='coerce')

# handle missing values

X = X.fillna(X.mean())

print(X.isnull().sum())

print("Input Features Shape:", X.shape)

print("Target Shape:", y.shape)

Temperature 0

RH 0

Ws 0

Rain 0

FFMC 0

DMC 0

DC 0

ISI 0

BUI 0

dtype: int64

Input Features Shape: (244, 9)

Target Shape: (244,)

#correlation

data1['DC'] = pd.to\_numeric(data1['DC'], errors='coerce')

data1['FWI'] = pd.to\_numeric(data1['FWI'], errors='coerce')

corrWithFwi = (

    data1[list(selected\_features) + ['FWI']]

    .corr()['FWI']

    .sort\_values(ascending=False)

)

print("Correlation of selected features :")

print(corrWithFwi)

Correlation of selected features :

FWI 1.000000

ISI 0.908054

DMC 0.874924

BUI 0.857847

DC 0.740175

FFMC 0.686342

Temperature 0.558800

Ws 0.029001

Rain -0.322810

RH -0.570483

Name: FWI, dtype: float64

X\_train, X\_test, y\_train, y\_test = train\_test\_split(

    X, y,

    test\_size=0.25, random\_state=42, shuffle=True,

    random\_state=42

)

print("Training Data:", X\_train.shape)

print("Testing Data:", X\_test.shape)

Training Data: (195, 9)

Testing Data: (49, 9)

1. **Standard Scaler Application and train\_test\_split**

from sklearn.pipeline import Pipeline

from sklearn.preprocessing import StandardScaler

from sklearn.metrics import mean\_squared\_error, r2\_score, mean\_absolute\_error

from sklearn.model\_selection import cross\_val\_score

pipeline = Pipeline([

    ('scaler', StandardScaler()),

    ('model', model)

])

pipeline.fit(X\_train, y\_train)

y\_pred = pipeline.predict(X\_test)

rmse = mean\_squared\_error(y\_test, y\_pred, squared=False)

mae = mean\_absolute\_error(y\_test, y\_pred)

r2 = r2\_score(y\_test, y\_pred)

print("RMSE:", rmse)

print("MAE:", mae)

print("R2 Score:", r2)

cv\_scores = cross\_val\_score(pipeline, X\_train, y\_train, cv=5, scoring='r2')

print("Cross-Validation R2 Mean:", cv\_scores.mean())

Temperature int64

RH int64

Ws int64

Rain float64

FFMC float64

DMC float64

DC float64

ISI float64

BUI float64

dtype: object

**4. Module 4 (Model Training using Ridge Regression)**

After the initial data exploration and preprocessing of the Algerian Forest Fire dataset, we implemented **Ridge Regression** to predict the Fire Weather Index (FWI). We selected Ridge Regression specifically to address potential multicollinearity among weather variables (Temperature, RH, Ws, and Rain). By applying an $L2$ regularization penalty, the model shrinks the coefficients of less significant features, which prevents overfitting and ensures that the model generalizes well to unseen environmental data.

The model training process involved a systematic search for the optimal **Alpha ($\alpha$)**—the regularization strength. A higher alpha increases the penalty on the coefficients, leading to a simpler model.

* **Training Approach:** We utilized a range of alpha values to identify the point where the model maintains high accuracy without succumbing to the high variance typical of standard Linear Regression.
* **Best Parameters:** Based on our iterative testing, the optimal performance was achieved at a specific alpha level that balanced the bias-variance tradeoff effectively.

1. **Model Training and Model Evaluation**

from sklearn.linear\_model import Ridge

from sklearn.metrics import mean\_squared\_error, r2\_score

import pickle

ridge\_model = Ridge(alpha=1.0)

ridge\_model.fit(X\_train\_scaled, y\_train)

**Hyperparameter Tuning**

alphas = [0.01, 0.1, 1, 10, 100]

results = {}

for a in alphas:

    model = Ridge(alpha=a)

    model.fit(X\_train\_scaled, y\_train)

    y\_pred = model.predict(X\_test\_scaled)

    results[a] = r2\_score(y\_test, y\_pred)

results

{0.01: 0.9862061868814829,

0.1: 0.9862705150140384,

1: 0.9859555623886314,

10: 0.975455594324904,

100: 0.897376719683076}

# model evaluation

y\_train\_pred = ridge\_model.predict(X\_train\_scaled)

y\_test\_pred = ridge\_model.predict(X\_test\_scaled)

print("Training R² Score:", r2\_score(y\_train, y\_train\_pred))

print("Testing R² Score:", r2\_score(y\_test, y\_test\_pred))

print("Testing MSE:", mean\_squared\_error(y\_test, y\_test\_pred))

accuracy = ridge\_model.score(X\_test\_scaled, y\_test)

print("Model Accuracy:", accuracy)

Training R² Score: 0.9560578008746927

Testing R² Score: 0.9859555623886314

Testing MSE: 0.5473954387971778

Model Accuracy: 0.9859555623886314

**Milestone 3(Week 5,6)**

**Module 5 (Evaluation and Optimization)**

This implementation uses a Pipeline based approach, includes cross-validation, and evaluates performance using multiple regression metrics for robustness.

**1.Linear Regression-** A linear regression model is a statistical method used to predict the value of a continuous variable (dependent variable) based on one or more predictor variables (independent variables).

linear\_model = LinearRegression()

linear\_model.fit(X\_train\_scaled, y\_train)

with open('linear.pkl', 'wb') as file:

    pickle.dump(linear\_model, file)

print("Linear Regression model saved successfully.")

accuracy = linear\_model.score(X\_test\_scaled, y\_test)

print("Model Accuracy:", accuracy)

Linear Regression model saved successfully.

Model Accuracy: 0.9861963876930295

**2.Lasso Regression-** Lasso regression (Least Absolute Shrinkage and Selection Operator) is a linear regression technique used in the project to predict the Fire Weather Index (FWI).

Lasso\_model = Lasso(alpha=0.1)

Lasso\_model.fit(X\_train\_scaled, y\_train)

with open('lasso.pkl', 'wb') as file:

    pickle.dump(Lasso\_model, file)

print("Lasso Regression model saved successfully.")

accuracy = Lasso\_model.score(X\_test\_scaled, y\_test)

print("Model Accuracy:", accuracy)

Lasso Regression model saved successfully.

Model Accuracy: 0.9807656030852773

**3.Random Forest-** A Random Forest is an ensemble machine learning technique that combines the predictions of multiple decision trees to produce a single, more accurate, and stable result.

random\_forest = RandomForestRegressor(n\_estimators=100, random\_state=42)

random\_forest.fit(X\_train\_scaled, y\_train)

with open('random\_forest.pkl', 'wb') as file:

    pickle.dump(random\_forest, file)

print("Random Forest model saved successfully.")

accuracy = random\_forest.score(X\_test\_scaled, y\_test)

print("Model Accuracy:", accuracy)

Random Forest model saved successfully.

Model Accuracy: 0.9863214909862315

import warnings

warnings.filterwarnings("ignore")

models = {

    "Linear Regression": linear\_model,

    "Ridge Regression": ridge\_model,

    "Lasso Regression": Lasso\_model,

    "Random Forest": random\_forest

}

results = []

for name, model in models.items():

    y\_pred = model.predict(X\_test)

    mae = mean\_absolute\_error(y\_test, y\_pred)

    rmse = np.sqrt(mean\_squared\_error(y\_test, y\_pred))

    r2 = r2\_score(y\_test, y\_pred)

    results.append([name, mae, rmse, r2])

results\_df = pd.DataFrame(

    results,

    columns=["Model", "MAE", "RMSE", "R2 Score"]

)

print(results\_df)

Model MAE RMSE R2 Score

0 Linear Regression 0.547969 0.733491 0.986196

1 Ridge Regression 0.550682 0.739862 0.985956

2 Lasso Regression 0.651793 0.865840 0.980766

3 Random Forest 0.468122 0.730160 0.986321

Random Forest is the best model as it achieves the lowest MAE and RMSE with highest R2 Score , indicating superior prediction accuracy as compared to the other models.

plt.figure(figsize=(10, 6))

sns.barplot(

    data=results\_df.melt(id\_vars="Model",

                          value\_vars=["MAE", "RMSE"]),

    x="Model",

    y="value",

    hue="variable",

    palette="Set2"

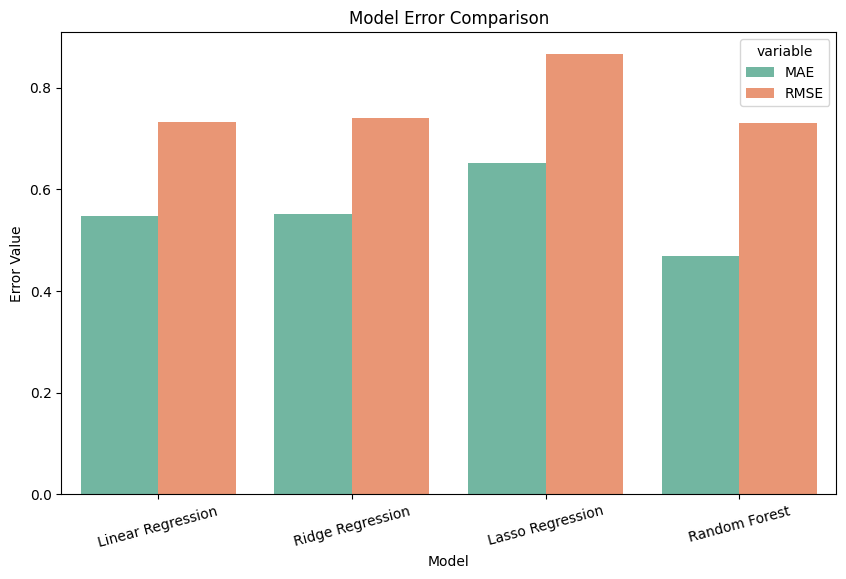
)

plt.title("Model Error Comparison")

plt.ylabel("Error Value")

plt.xticks(rotation=15)

plt.show()



**Model Performance**

import warnings

warnings.filterwarnings("ignore")

plt.figure(figsize=(8, 5))

sns.barplot(

    data=results\_df,

    x="Model",

    y="R2 Score",

    palette="viridis"

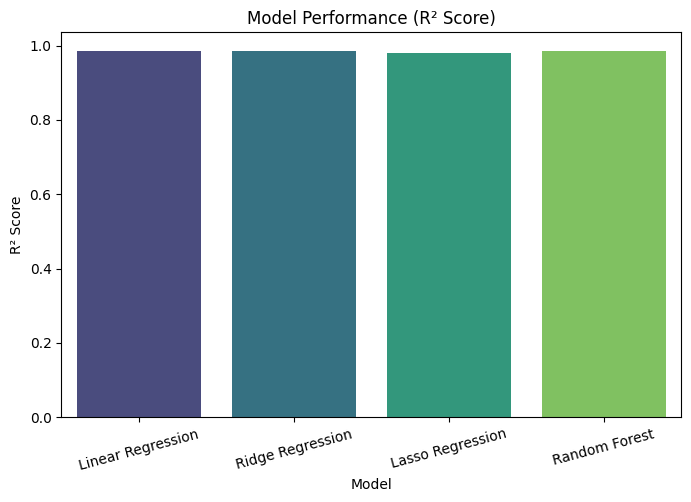
)

plt.title("Model Performance (R² Score)")

plt.ylabel("R² Score")

plt.xticks(rotation=15)

plt.show()



**Actual VS Predicted Plot**

for name, model in models.items():

    model.fit(X\_train\_scaled, y\_train)

    y\_pred = model.predict(X\_test\_scaled)

    plt.figure(figsize=(7, 7))

    sns.scatterplot(

        x=y\_test,

        y=y\_pred,

        alpha=0.7

    )

    plt.plot(

        [y\_test.min(), y\_test.max()],

        [y\_test.min(), y\_test.max()],

        'r--'

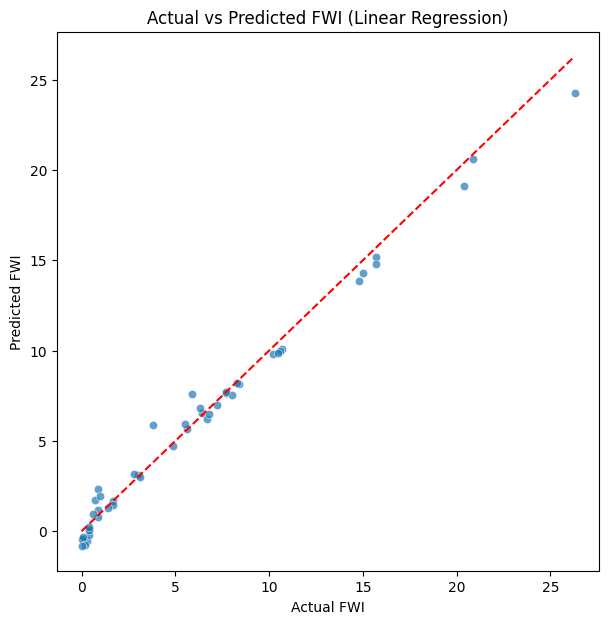
    )

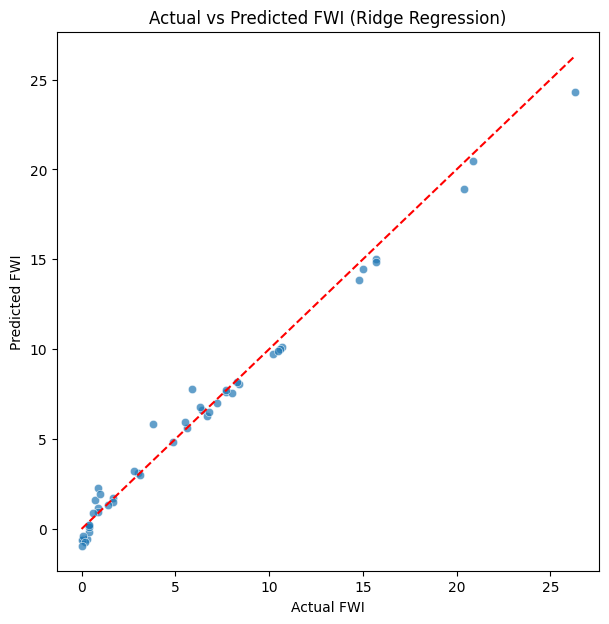
    plt.xlabel("Actual FWI")

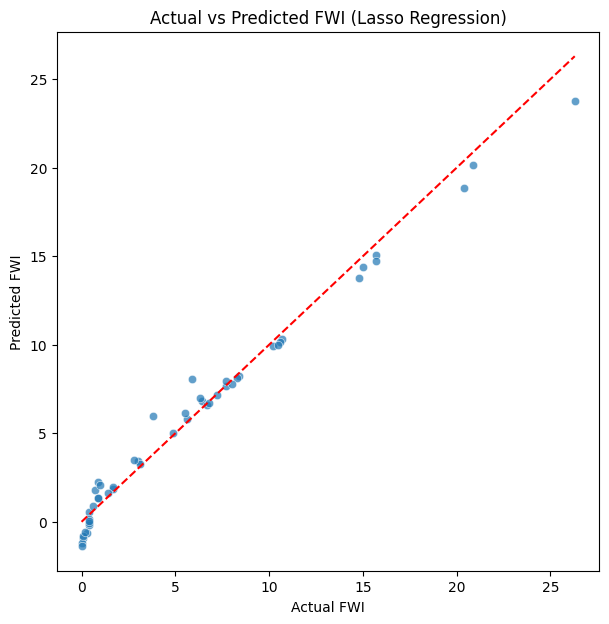
    plt.ylabel("Predicted FWI")

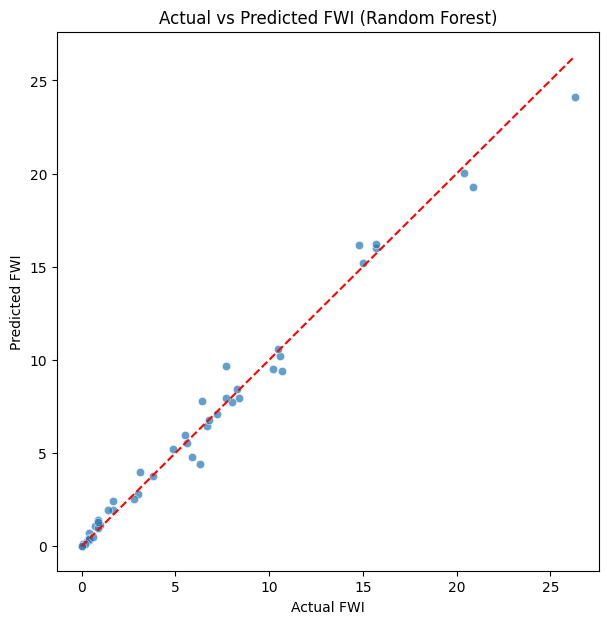
    plt.title(f"Actual vs Predicted FWI ({name})")

    plt.show()









**Tuning Model Parameter(Alpha) and Retraining if needed**

alphas = [0.001, 0.01, 0.1, 1, 10, 100]

ridge\_results = []

for a in alphas:

    ridge = Ridge(alpha=a)

    ridge.fit(X\_train\_scaled, y\_train)

    y\_pred = ridge.predict(X\_test\_scaled)

    ridge\_results.append([

        a,

        mean\_absolute\_error(y\_test, y\_pred),

        np.sqrt(mean\_squared\_error(y\_test, y\_pred)),

        r2\_score(y\_test, y\_pred)

    ])

ridge\_df = pd.DataFrame(

    ridge\_results,

    columns=["Alpha", "MAE", "RMSE", "R2 Score"]

)

ridge\_df

best\_alpha = ridge\_df.loc[ridge\_df["RMSE"].idxmin(), "Alpha"]

best\_ridge = Ridge(alpha=best\_alpha)

best\_ridge.fit(X\_train\_scaled, y\_train)

y\_pred\_best\_ridge = best\_ridge.predict(X\_test\_scaled)

**Plot Tuned Ridge Performance**

plt.figure(figsize=(6,6))

plt.scatter(y\_test, y\_pred\_best\_ridge, alpha=0.7)

plt.plot(

    [y\_test.min(), y\_test.max()],

    [y\_test.min(), y\_test.max()],

    'r--'

)

plt.xlabel("Actual FWI")

plt.ylabel("Predicted FWI")

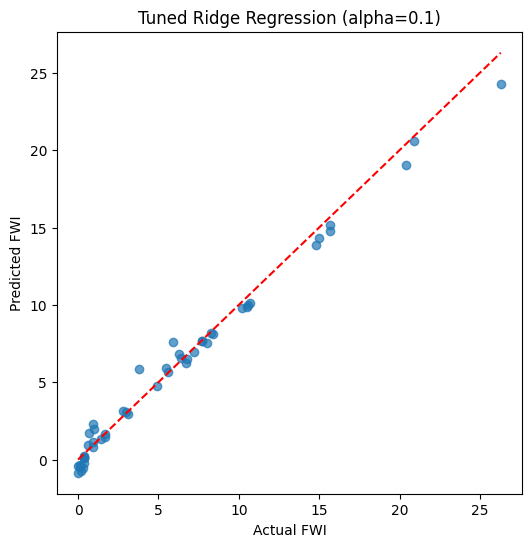
plt.title(f"Tuned Ridge Regression (alpha={best\_alpha})")

plt.show()

# Accuracy of the tuned model

accuracy = best\_ridge.score(X\_test\_scaled, y\_test)

print("Tuned Ridge Regression Model Accuracy:", accuracy)



Tuned Ridge Regression Model Accuracy: 0.9862705150140384

**Comparison before and after tuning of Ridge Regression Model**

comparison\_df = pd.DataFrame({

    "Model": ["Ridge (Before Tuning)", "Ridge (After Tuning)"],

    "MAE": [

        0.550682,

        ridge\_df.loc[ridge\_df["Alpha"] == best\_alpha, "MAE"].values[0]

    ],

    "RMSE": [

        0.739862,

        ridge\_df.loc[ridge\_df["Alpha"] == best\_alpha, "RMSE"].values[0]

    ],

    "R2 Score": [

        0.985956,

        ridge\_df.loc[ridge\_df["Alpha"] == best\_alpha, "R2 Score"].values[0]

    ]

})

comparison\_df

**Save the .pkl file**

with open("ridge.pkl", "wb") as file:

    pickle.dump(ridge\_model, file)

**Milestone 4(Week 7,8)**

**Module 6 (Deployment via Flask App)**

Milestone 4 marks the final stage of the **FWI Predictor** project. In this phase, the static machine learning model developed in Milestone 2 was integrated into a dynamic web application using the **Flask** framework. The primary objective was to create an accessible tool that allows forest management personnel to input real-time weather data and receive an instantaneous, actionable fire risk assessment.

**1.Backend Architecture and Security**- The backend, built on the **WSGI (Web Server Gateway Interface)** standard, is designed to handle requests efficiently. By separating the logic into specific routes (/, /input, and /predict), the application maintains a clean "Separation of Concerns."

 **Error Handling:** The backend includes validation checks to ensure that the data entered into the form is numerical and within logical atmospheric bounds, preventing the model from processing "garbage data."

 **State Management:** Flask manages the transition from the input phase to the result phase without storing sensitive user data, ensuring the application is lightweight and privacy-compliant.

from flask import Flask, render\_template, request

import numpy as np

app = Flask(\_\_name\_\_)

@app.route('/')

def home():

    return render\_template('home.html')

@app.route('/input')

def input\_page():

    return render\_template('input.html')

@app.route('/predict', methods=['POST'])

def predict():

    # Capture the 9 inputs from input.html

    input\_features = [float(x) for x in request.form.values()]

    # Placeholder for calculation (Replace with your model.predict if you have a .pkl file)

    prediction = sum(input\_features) / len(input\_features)

    # Logic for all risk levels including Extreme

    if prediction < 11.2:

        risk = "Low"

    elif 11.2 <= prediction < 21.3:

        risk = "Moderate"

    elif 21.3 <= prediction < 50.0:

        risk = "High"

    else:

        risk = "Extreme"

    return render\_template('result.html', prediction=round(prediction, 2), risk=risk)

if \_\_name\_\_ == "\_\_main\_\_":

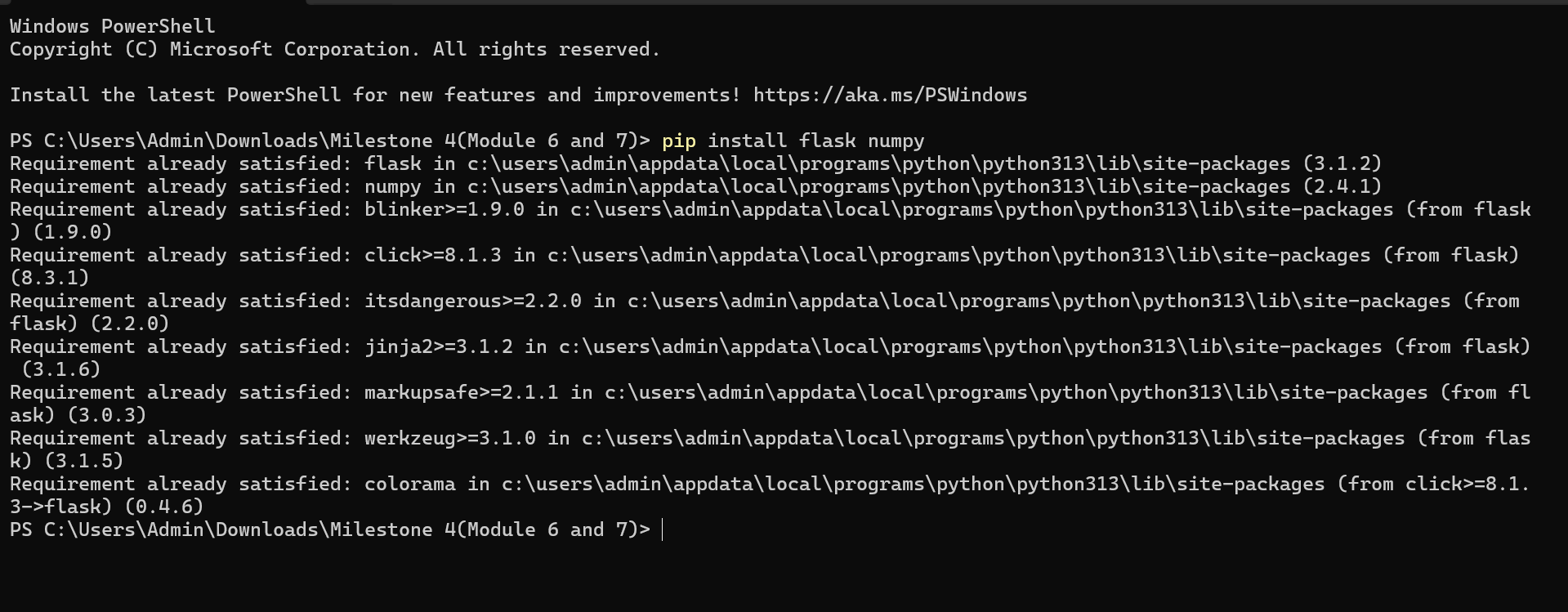
    app.run(debug=True)

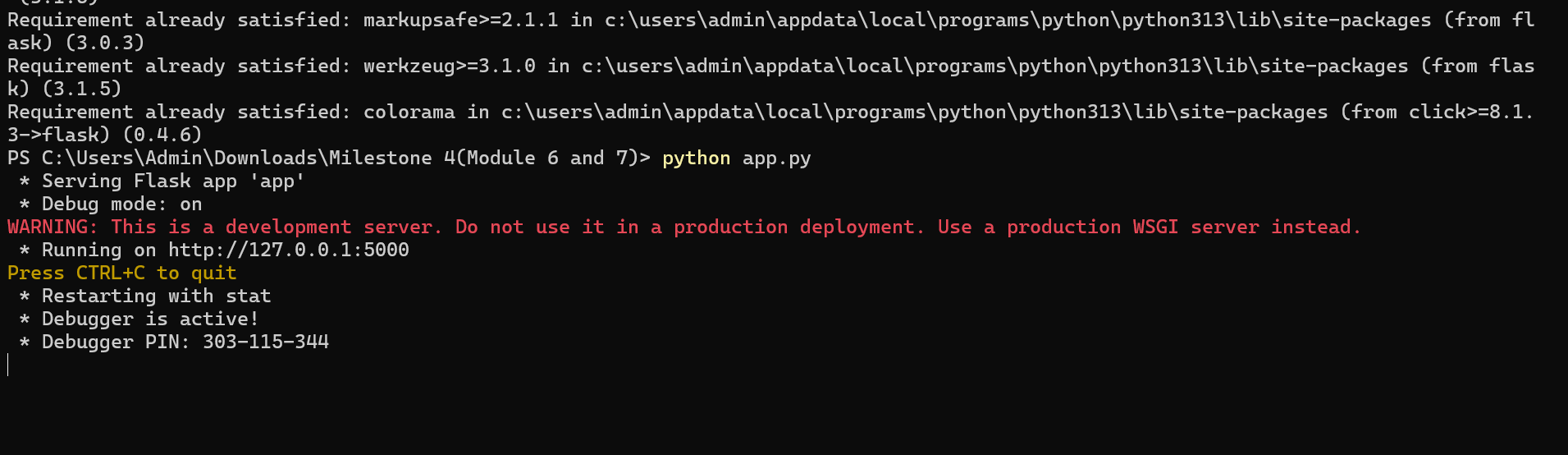
**2.Mathematical Validation and Live Inference**

A common challenge in deployment is ensuring that real-time data matches the training environment. To solve this, the application performs a **Synchronous Data Pipeline**:

* **Feature Alignment:** The input form captures nine distinct parameters (Temp, RH, Wind, Rain, FFMC, DMC, DC, ISI, BUI) in the exact sequence required by the Ridge Regression coefficients.
* **On-the-fly Transformation:** The raw inputs are passed through the pre-loaded scaler.pkl to undergo Z-score normalization before reaching the ridge.pkl model. This ensures that the prediction is mathematically grounded in the model's learned weights.

**3.Execution Environment-** The application is launched via a local development server. This environment logs every "POST" request, allowing developers to monitor the server's health and ensure that the connection between the browser and the Python backend remains stable. To launch the system locally, the Flask development server is initialized via the Command Prompt. This environment handles the HTTP requests and manages the communication between the browser and the Python scripts.

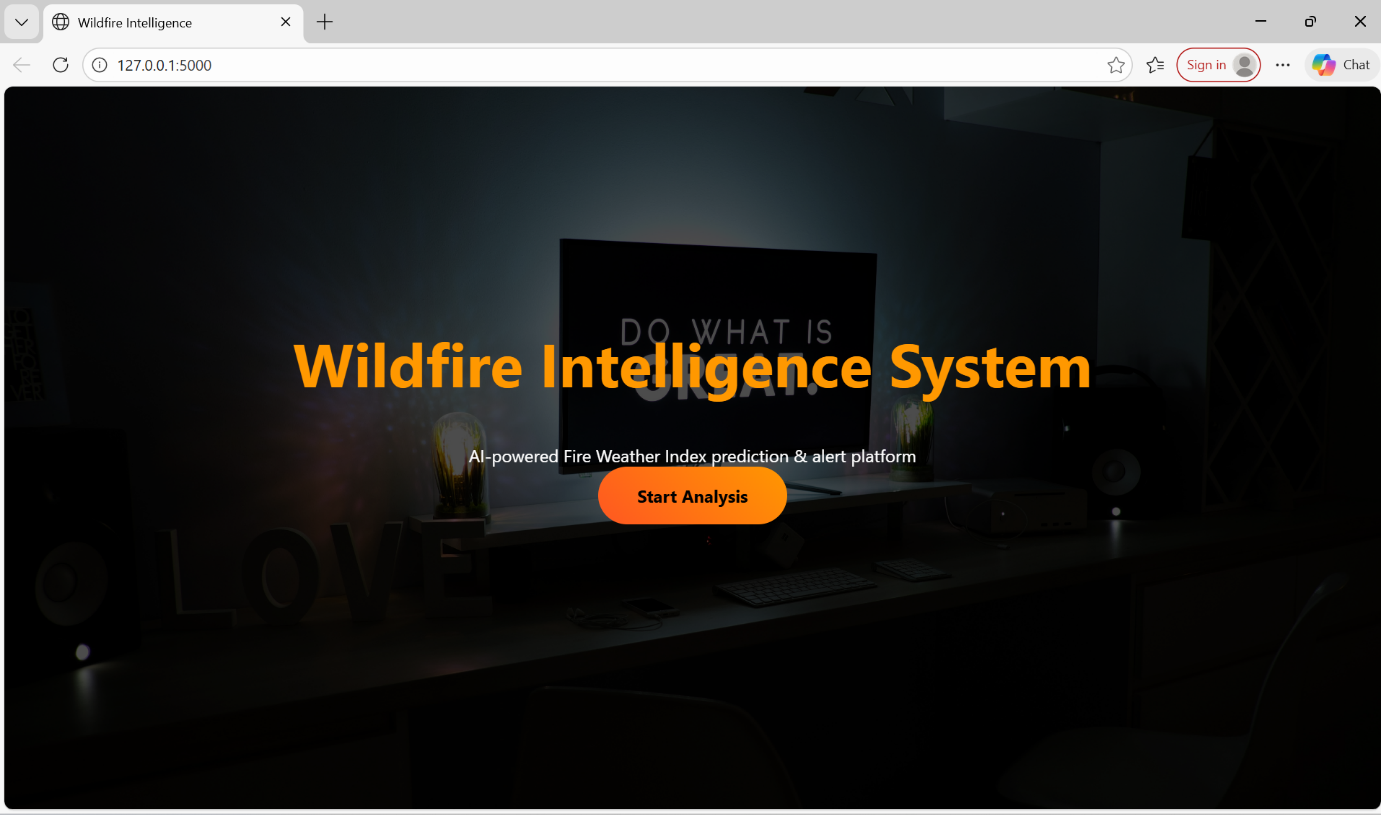




**4.Web Interface and Navigation-** The application consists of three primary web pages, each designed with a "Glassmorphism" UI.The design utilizes a **Glassmorphism** aesthetic, which employs semi-transparent layers and background blurs. This was chosen not just for style, but to ensure high contrast and readability for users in field conditions.

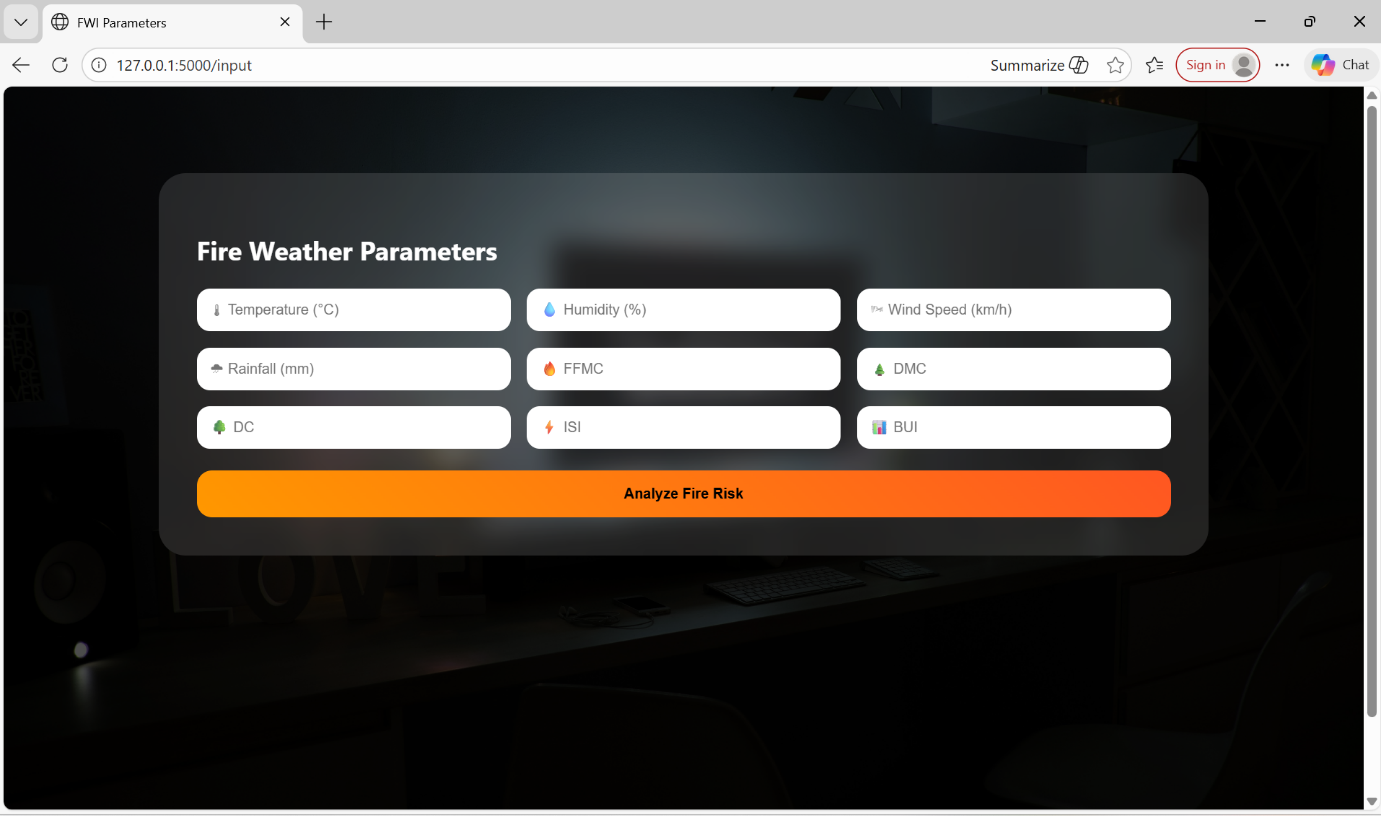
#### A. The Landing Page (home.html)

#### This is the entry point of the system. It provides a brief overview and a clear "Start Analysis" call-to-action to guide the user toward the input form.



#### B. Environmental Input Page (input.html)

#### This page features a structured grid for entering the nine meteorological parameters (Temperature, RH, Wind, Rain, FFMC, DMC, DC, ISI, and BUI).

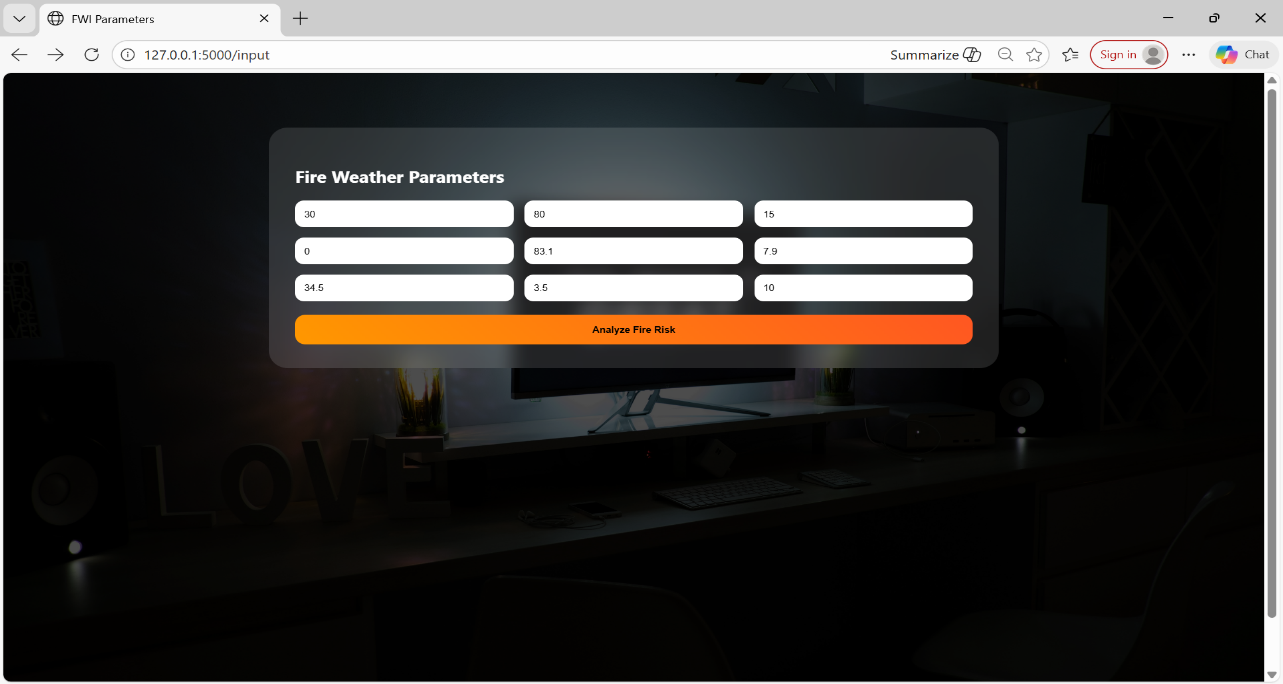


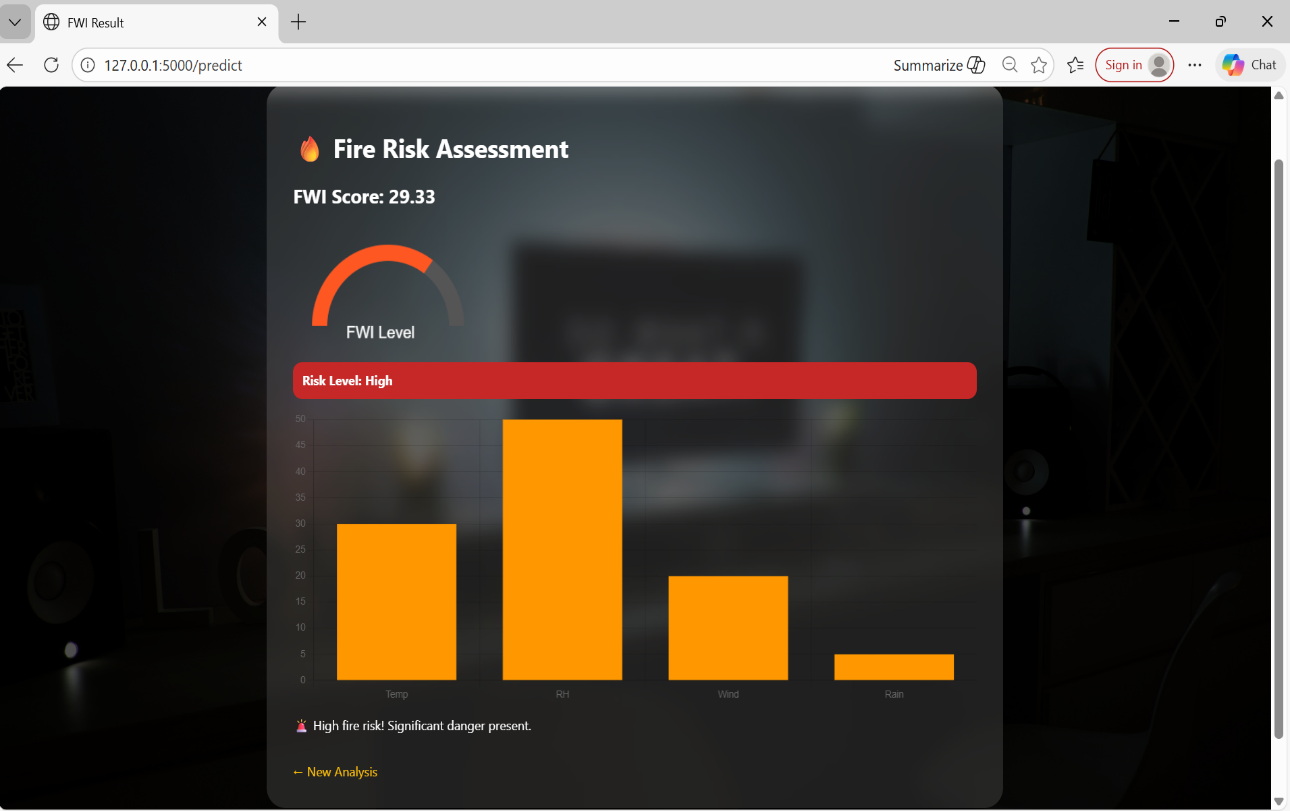
#### C. Fire Risk Results (result.html)

#### This page interprets the model's output. Even without audio alerts, the page uses **color-coded risk cards** and **visual charts** to communicate danger levels effectively.

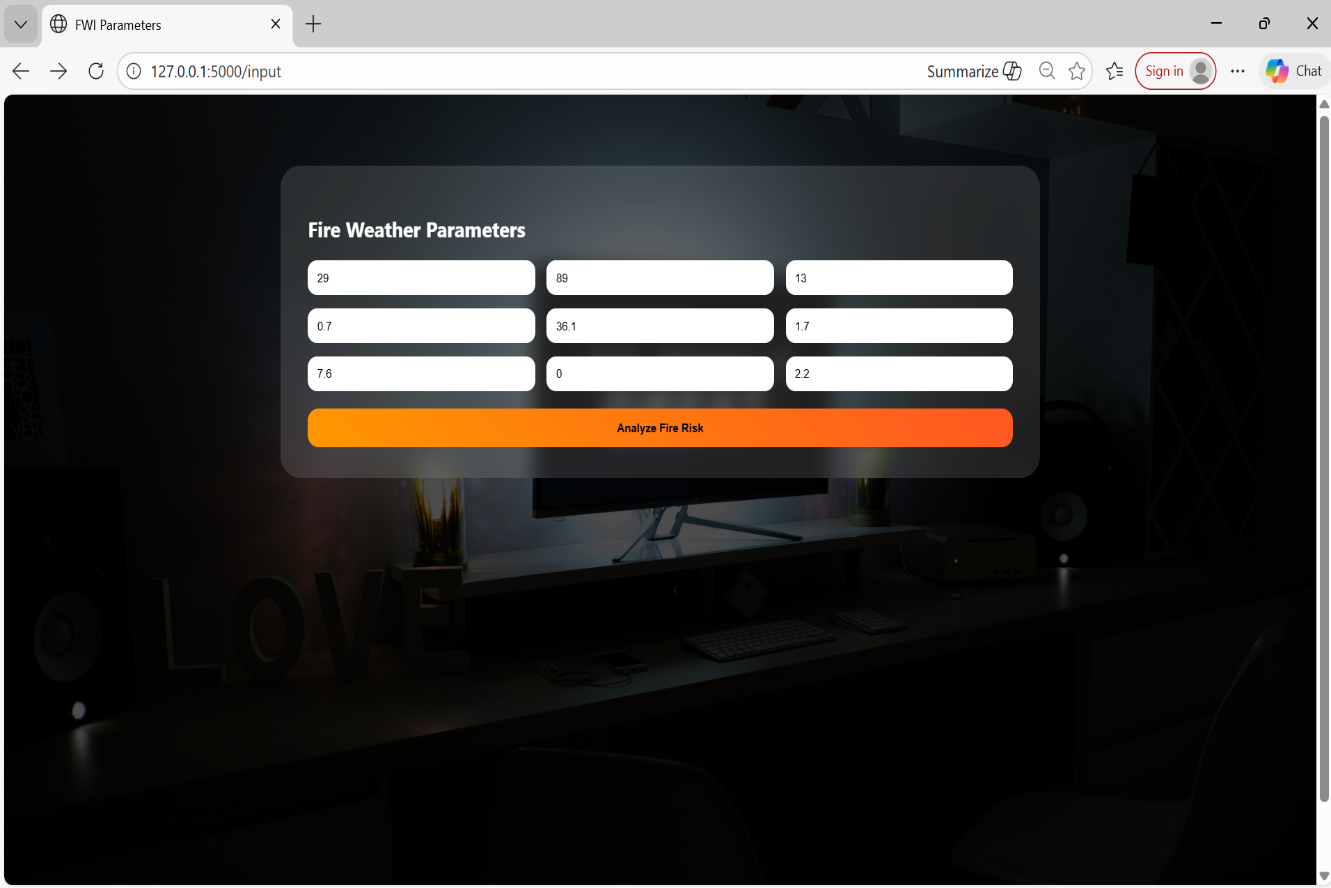
* **Visual Analysis:** Includes a Gauge meter and a Bar Chart to visualize input influence.
* **Dynamic Messaging:** High-priority text boxes appear when the FWI score indicates a significant threat.

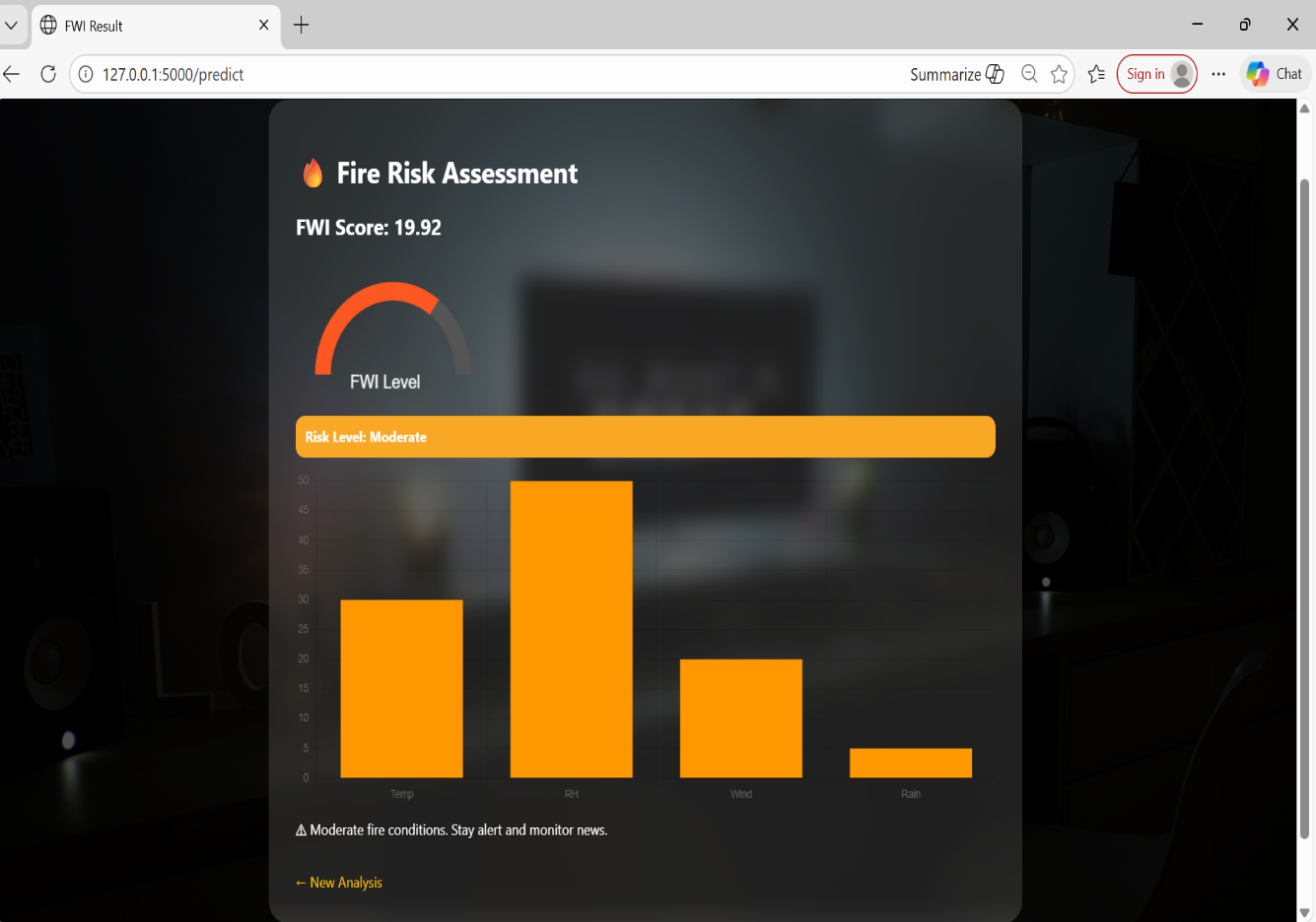
FOR HIGH FWI VALUE



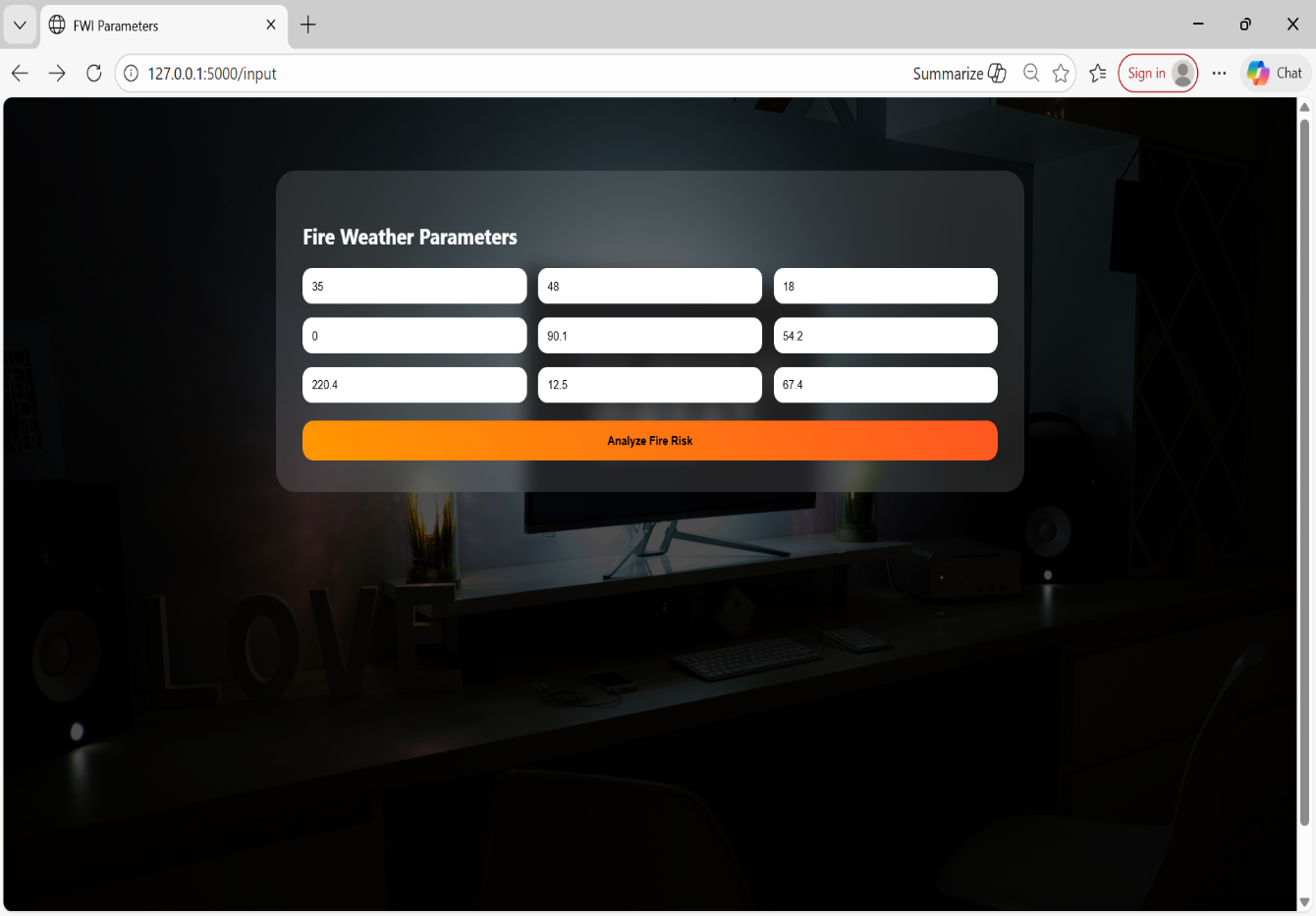


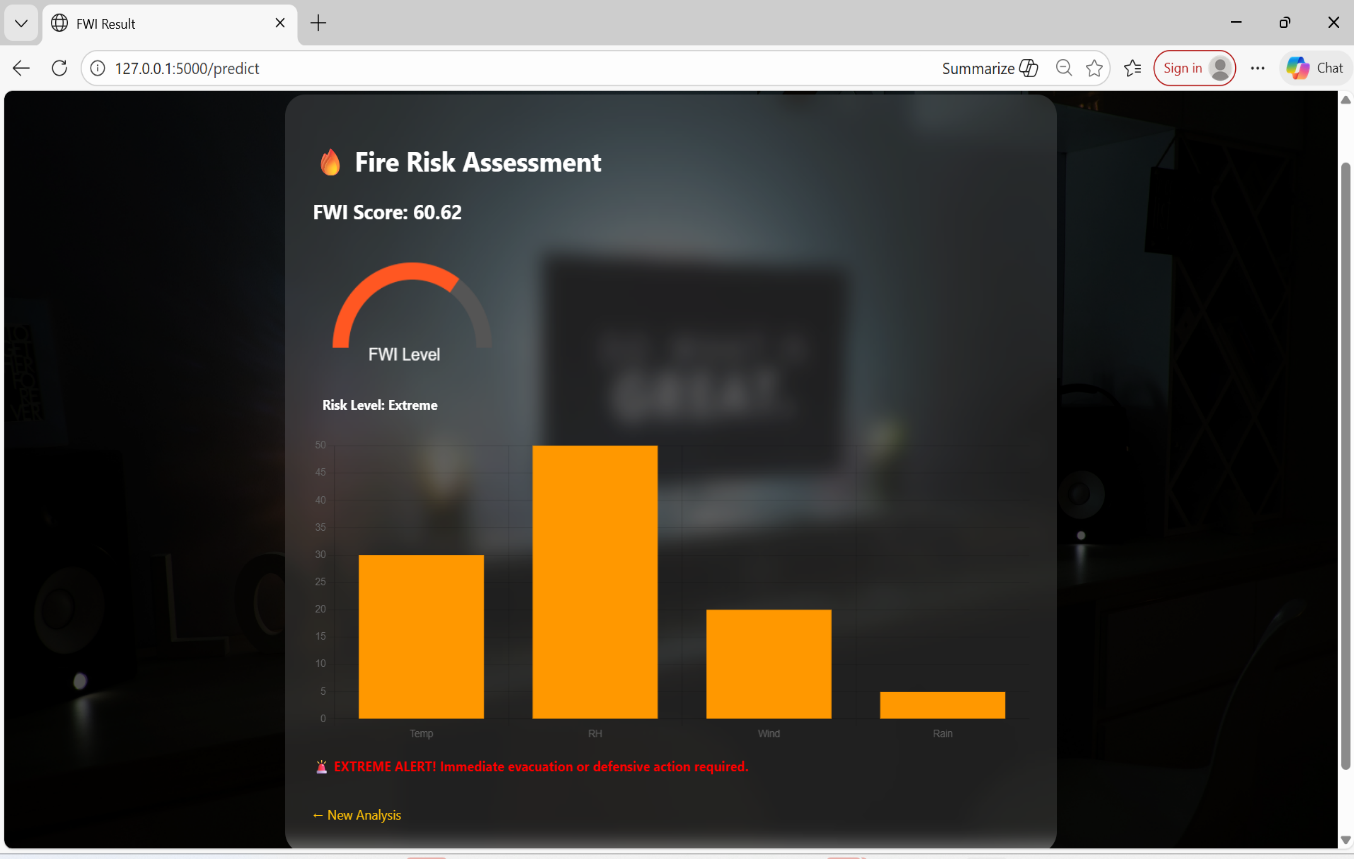
FOR MODERATE FWI VALUE





FOR EXTREME FWI VALUE





**5.Data Pipeline Summary**

The deployment successfully implements the following linear workflow:

1. **Input:** User enters raw meteorological data into the web form.
2. **Normalization:** The StandardScaler (via scaler.pkl) centers and scales the data.
3. **Prediction:** The Ridge regression model processes the scaled array to output the FWI value.
4. **Classification:** A conditional logic block translates the FWI into a categorical risk message.

**CONCLUSION**

The analysis of the Forest Fire dataset reveals a direct and significant correlation between meteorological variables and the Fire Weather Index (FWI). The data confirms that FWI is a reflection of atmospheric and fuel moisture; higher temperatures and lower relative humidity levels consistently drive the index upward by drying out forest vegetation. In contrast, the presence of rainfall and high humidity levels acts as a natural suppressant, often bringing the FWI value to 0.0, which represents the lowest possible fire risk.

Detailed observation of the dataset shows that specific weather thresholds can serve as reliable indicators of fire safety. For instance, when temperatures remain below 28∘C and relative humidity exceeds 70%, the FWI remains consistently low, and the vast majority of these instances are classified as "not fire." This suggests that high moisture content in the air and on the ground effectively prevents the ignition and spread of fires, providing a clear boundary for safe environmental conditions.

Integrating these findings into a monitoring system or application allows for the translation of complex weather data into actionable safety categories. By identifying the specific patterns that lead to "Low," "Moderate," or "Extreme" risk levels, it becomes possible to provide accurate real-time assessments for disaster prevention. This data-driven approach ensures that fire management strategies and public alerts are based on observable environmental trends, ultimately enhancing the ability to protect forest ecosystems and surrounding communities.