Fire Weather Index Predictor

(A Machine Learning Model to Predict Fire Weather Index)

Infosys Springboard

Infosys SpringBoard Virtual Internship Program

Submitted by

Gudipati Naga Venkata Sravanthi

Under the guidance of Mentor **Praveen**

PROBLEM STATEMENT:

We have developed a Fire Weather Index (FWI) Prediction System that will evaluate the potential risk of wildfires based on several key meteorological and fire behavior factors. The FWI Prediction System will utilize temperature, humidity, wind speed, precipitation levels, as well as several important fire danger indices (FFMC, DMC, DC, ISI). The data has been thoroughly pre- processed, cleaned and explored to validate its quality and visually identify the most relevant trends in the data.

These advanced models allow us to model the relationships between multiple variables, both environmental and fire risk-related, for the creation of accurate and reliable Fire Weather Index Predictions (FWI). In addition to this input, we include both localized and regional-level information to help assess patterns of wildfires to improve the FWI Prediction System's applicability to local fire patterns. The ultimate objective of the FWI Prediction System is to assist both government agencies as well as decision-makers in finding fire locations before a major wildfire outbreak, taking preventative action and effectively allocating resources toward those areas at greatest risk.

EXPECTED OUTCOMES:

Deliverables are expected, but this list does not include everything. Some examples of deliverables are: a Ridge Regression model that can give accurate Fire Weather Index (FWI) predictions in different weather conditions; a data-preparation process that uses StandardScaler to normalize the data so the model works better; a simple Flask web app where users can enter weather values and get FWI predictions instantly; and an analytical tool that helps forest departments, disaster management teams, and research groups plan for wildfires by using information from past data.

Modules to be Implemented:

- Data Collection
- Data Exploration (EDA) and Data Preprocessing
- Feature Engineering and Scaling
- Model Training using Ridge Regression
- Evaluation and Optimization
- Deployment via Flask App
- Presentation and Documentation

Requirements:

- pandas==2.3.3
- numpy==2.3.5
- matplotlib>=3.0
- seaborn>=0.11
- scikit-learn>=1.0

Milestone 1

Module 1: Data Collection

Initially, the data to be used in making accurate predictions of the FWI was collected from different sources on the internet. The final choice was based on the environmental and fire danger parameter factors that were necessary or relevant to make an accurate prediction. The chosen dataset contained the following parameters: Temperature, Humidity, Wind Speed, Rainfall, FFMC, DMC, ISI, Regional Identification. All of these parameters became part of the dataset. Once the dataset was acquired, it was placed in a Pandas DataFrame. The first thing that was done was to check that the data was ready for analysis by reviewing its integrity.

The integrity review included assessing the types of data, identifying which components contained missing or inconsistent values, assessing how much memory the dataset consumed, generating statistical summaries for all variables, understanding the distribution of each feature within the dataset, confirming the number or number of rows and columns, and identifying and removing any duplicated rows. After confirming that the dataset had met all of the needed requirements for quality, the dataset was validated to ensure that it was accurate enough for the future phases of development and modelling.

1. loading the dataset

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import LabelEncoder
import warnings
warnings.filterwarnings('ignore')
# Load the dataset
df= pd.read_csv("E:\FWI\myenv\FWI Dataset.csv")
df.head(4)
```

	day	month	year	Temperature	RH	Ws	Rain	FFMC	DMC	DC	ISI	BUI	FWI	Classes	Region
0	1	6	2012	29	57	18	0.0	65.7	3.4	7.6	1.3	3.4	0.5	not fire	Bejaia
1	2	6	2012	29	61	13	1.3	64.4	4.1	7.6	1.0	3.9	0.4	not fire	Bejaia
2	3	6	2012	26	82	22	13.1	47.1	2.5	7.1	0.3	2.7	0.1	not fire	Bejaia
3	4	6	2012	25	89	13	2.5	28.6	1.3	6.9	0.0	1.7	0	not fire	Bejaia

FIGURE 1

```
# Data types
df['DC'] = df['DC'].astype(str).str.replace(' ', '', regex=False)
df['FWI'] = df['FWI'].astype(str).str.replace(' ', '', regex=False)
df['DC'] = pd.to_numeric(df['DC'], errors='coerce')
df['FWI'] = pd.to_numeric(df['FWI'], errors='coerce')
df.dtypes
```

```
day              int64
month            int64
year             int64
Temperature      int64
 RH              int64
 Ws              int64
Rain           float64
FFMC           float64
DMC            float64
DC             float64
ISI            float64
BUI            float64
FWI            float64
Classes         object
Region          object
dtype: object
```

FIGURE 2: Data Types

```
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    float64
 10  ISI          244 non-null    float64
 11  BUI          244 non-null    float64
 12  FWI          243 non-null    float64
 13  Classes      243 non-null    object 
 14  Region       244 non-null    object 
dtypes: float64(7), int64(6), object(2)
memory usage: 28.7+ KB
```

FIGURE 3

```
#step 1: Check for missing values

print("Missing values in each column:")

df.isnull().sum()

rows_with_missing = df[df.isnull().any(axis=1)]

print("Rows with missing values: \n",rows_with_missing)
```

```
Rows with missing values:
      day  month  year  Temperature   RH   Ws  Rain   FFMC   DMC     DC   ISI  \
165   14       7  2012           37   37   18    0.2  88.9  12.9  14.69  12.5

      BUI  FWI Classes            Region
165  10.4  NaN       NaN  Sidi-Bel Abbes
```

FIGURE 4

MODULE 2: Data Exploration (EDA) and Data Preprocessing

During the preprocessing and exploratory analysis phase, the dataset was thoroughly examined to ensure its accuracy, completeness, and suitability for machine learning model development. This stage aimed to identify and resolve data quality issues while gaining a deeper understanding of the underlying patterns within the dataset.

```
#step 2: Outlier Detection using Boxplots
plt.figure(figsize=(12, 6))
df.boxplot(rot=45)
plt.title("Boxplot for Outlier Detection")
plt.show()
```

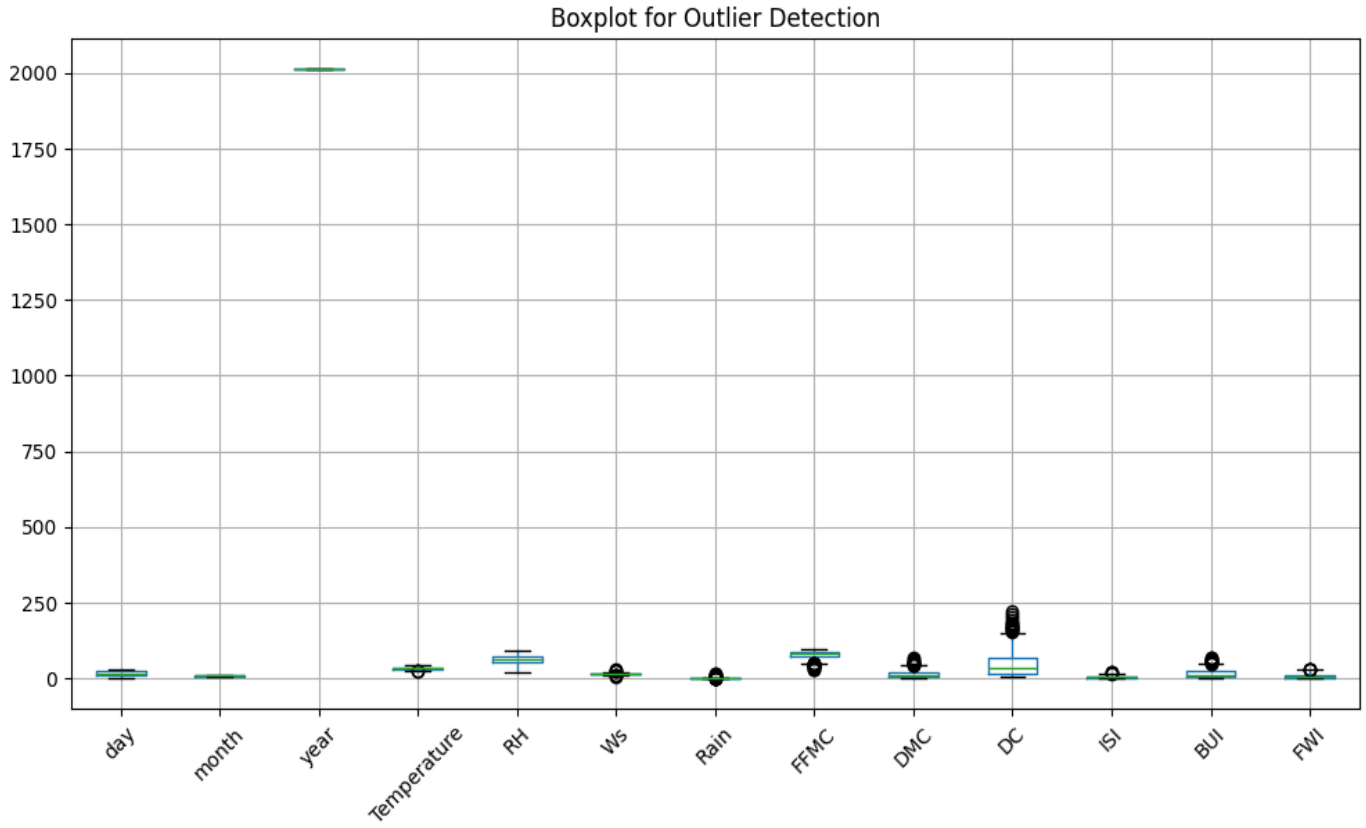

FIGURE 5 : Outlier Detection

Outliers in the dataset were analyzed using both visual methods (boxplots) and statistical techniques based on the Interquartile Range (IQR). This combined approach helped identify extreme values that could negatively impact model performance.

```
#Statistical Threshold
num_cols = df.select_dtypes(include=['float64', 'int64']).columns
def detect_outliers_iqr(data, column):
    Q1 = data[column].quantile(0.25)
    Q3 = data[column].quantile(0.75)
    IQR = Q3 - Q1
    lower_bound = Q1 - 1.5 * IQR
    upper_bound = Q3 + 1.5 * IQR
    outliers = data[(data[column] < lower_bound) | (data[column] >
upper_bound)]
    return outliers

# Check number of outliers in each numerical column
for col in num_cols:
    outliers = detect_outliers_iqr(df, col)
    print(f"Column: {col} → Number of outliers: {len(outliers)}")
```

```
Column: day → Number of outliers: 0
Column: month → Number of outliers: 0
Column: year → Number of outliers: 0
Column: Temperature → Number of outliers: 2
Column: RH → Number of outliers: 0
Column: Ws → Number of outliers: 8
Column: Rain → Number of outliers: 35
Column: FFMC → Number of outliers: 16
Column: DMC → Number of outliers: 12
Column: DC → Number of outliers: 15
Column: ISI → Number of outliers: 4
Column: BUI → Number of outliers: 12
Column: FWI → Number of outliers: 4
```

FIGURE 6: Statistical Threshold

To identify extreme or abnormal values, the **Interquartile Range (IQR) method** was applied to all numerical features in the dataset. This statistical technique detects outliers by measuring how far a value lies from the central spread of the data.

```
#step 3: Visualizations
#Histograms
df.hist(figsize=(14, 10), bins=20, edgecolor='black')
plt.suptitle("Histograms of Numerical Features")
plt.show()
```

```
# Density Plots
plt.figure(figsize=(12, 8))

for col in num_cols:
    sns.kdeplot(df[col], label=col, fill=True)

plt.title("Density Plots of Numerical Features")
plt.xlabel("Value")
plt.ylabel("Density")
plt.legend()
plt.show()
```

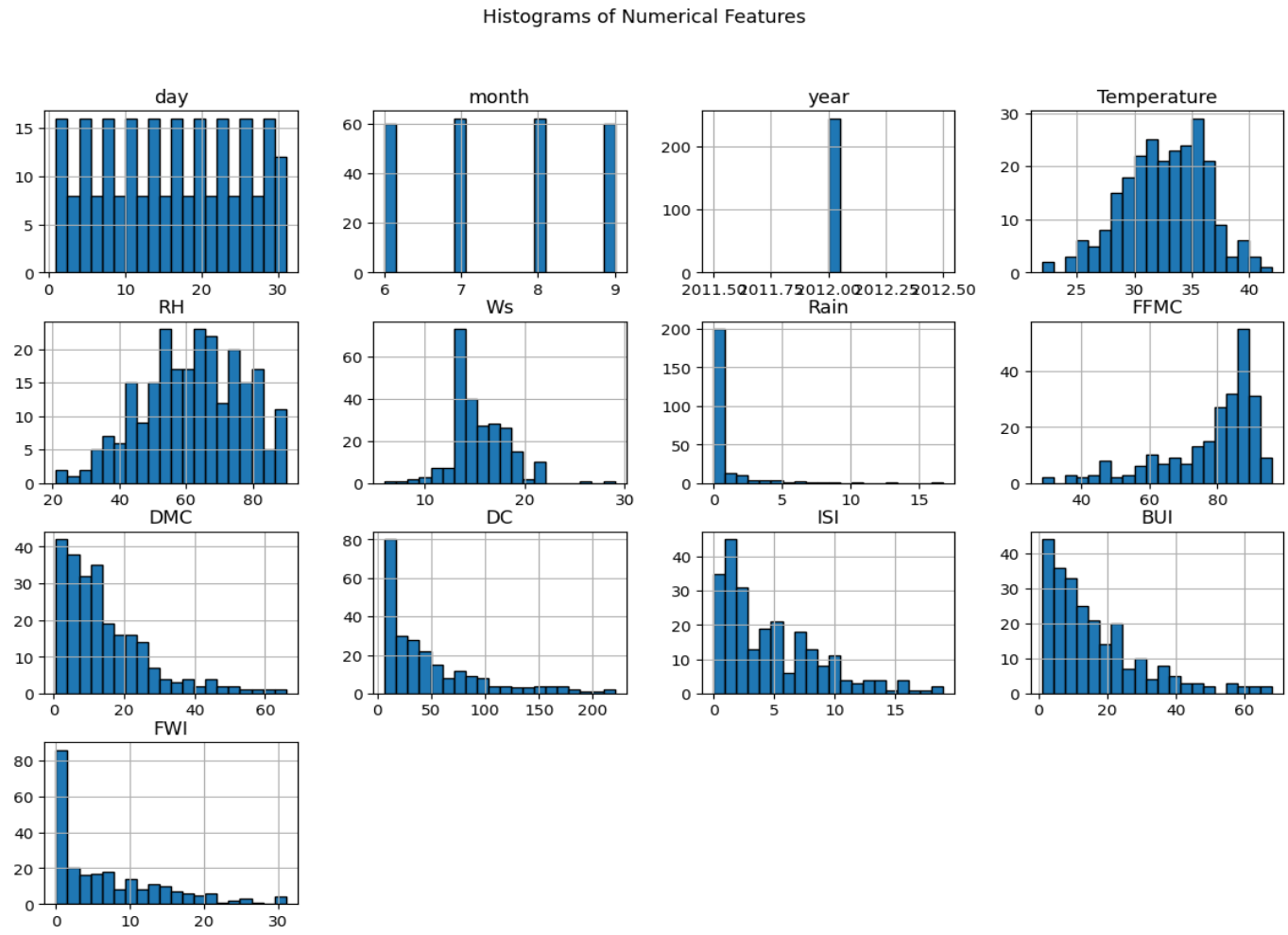

FIGURE 7: Histograms

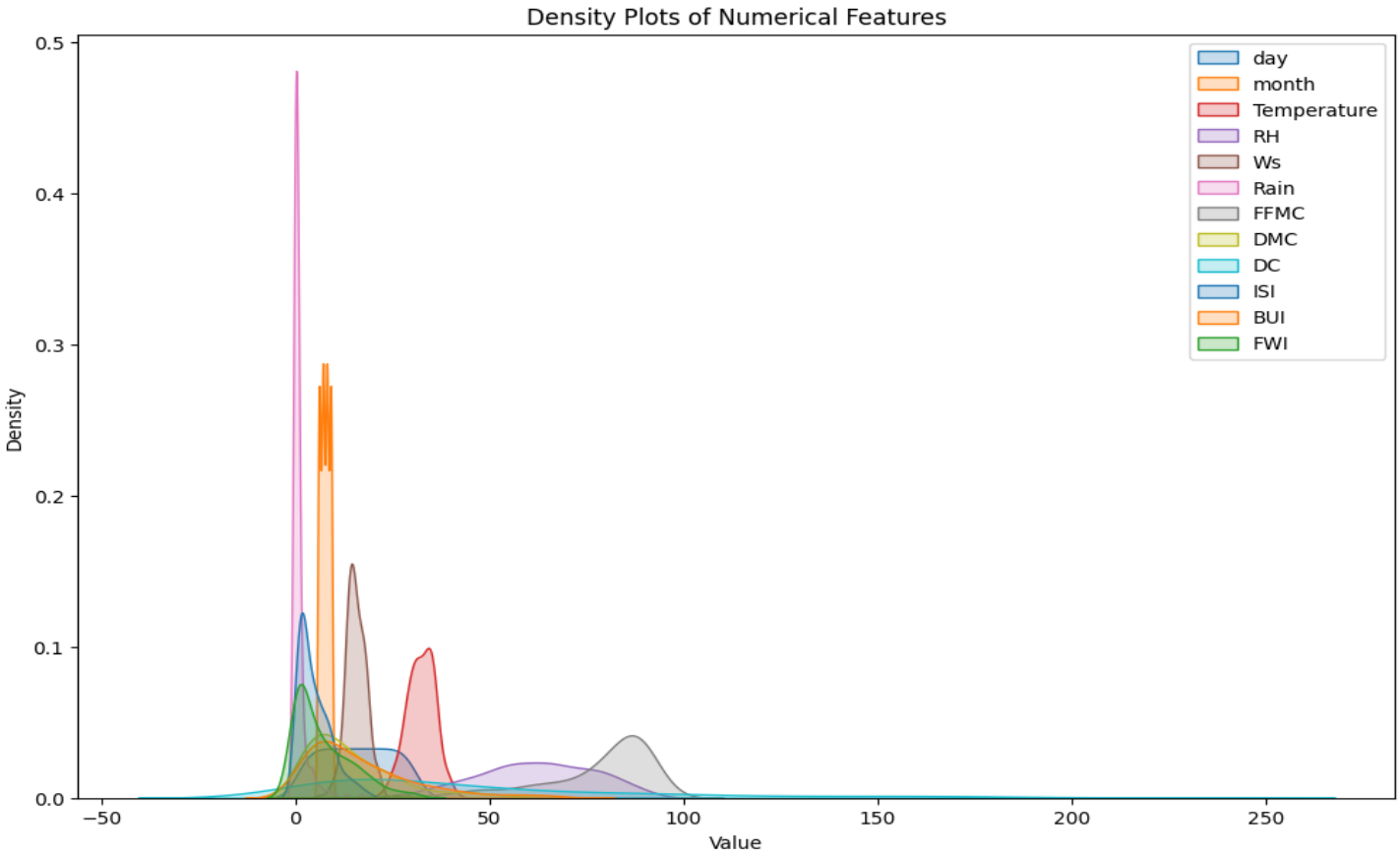

FIGURE 8: Density Plots

Histograms combined with KDE (Kernel Density Estimate) curves were generated for all numerical features to understand the overall distribution and behavior of the data. These visualizations help reveal important patterns that may influence the performance of the machine learning model.

Key Observations from the Graphs:

- **Temperature, Relative Humidity (RH), and Wind Speed** show smoother and more balanced distributions, indicating that these variables are spread fairly evenly across the dataset.
- **Rain, DMC, ISI, and BUI** exhibit strong right-skewness. This means most values are low, with only a few extreme high values present. Such skewness is common in environmental datasets where rainfall and fire indices vary significantly over time.

- Features like **FFMC and DC** display moderate variation with visible peaks, suggesting consistent seasonal or environmental patterns.
- The KDE curves helped identify whether each feature follows a normal distribution or deviates from it. Identifying non-normal distributions is crucial for selecting appropriate scaling methods and machine learning algorithms.

```
#step 4: Correlation Matrix
plt.figure(figsize=(12, 8))
correlation_matrix = df[num_cols].corr()
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', fmt=".2f")
plt.title("Correlation Matrix Heatmap")
plt.show()
```

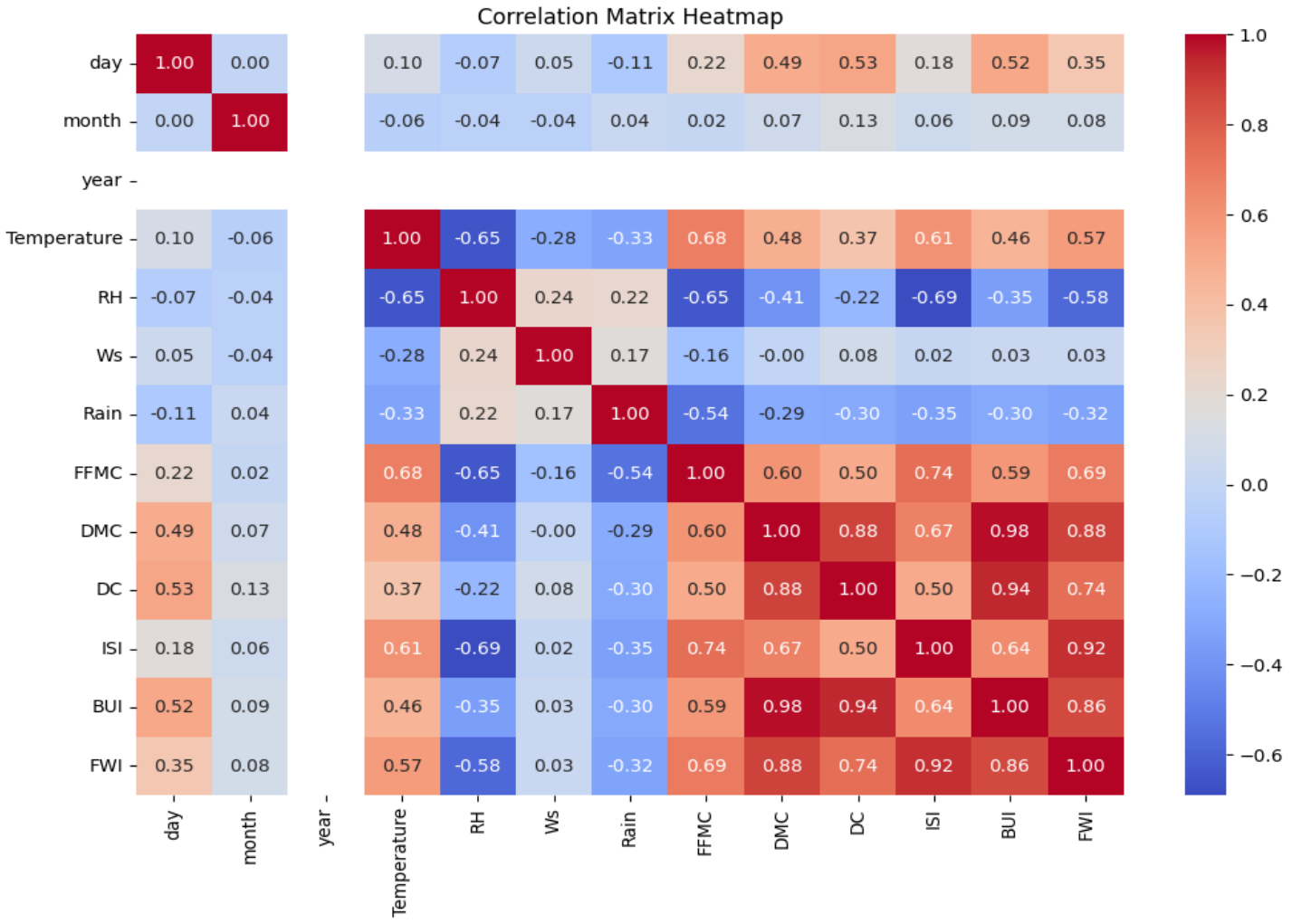

FIGURE 9: Correlation Matrix

A correlation heatmap was generated to study relationships among numerical features.

Key findings:

- Strong positive correlations were observed among *BUI, DMC, FFMC, ISI,* and *Temperature*, suggesting shared patterns in fire-danger conditions.
- *Relative Humidity (RH)* showed strong negative correlations with several fire indices, reflecting its natural dampening effect on fire risk.

```
# Scatterplots of Selected Features
sns.pairplot(df[num_cols].dropna())
plt.suptitle("Scatterplots of Selected Features", y=1.02)
plt.show()
```

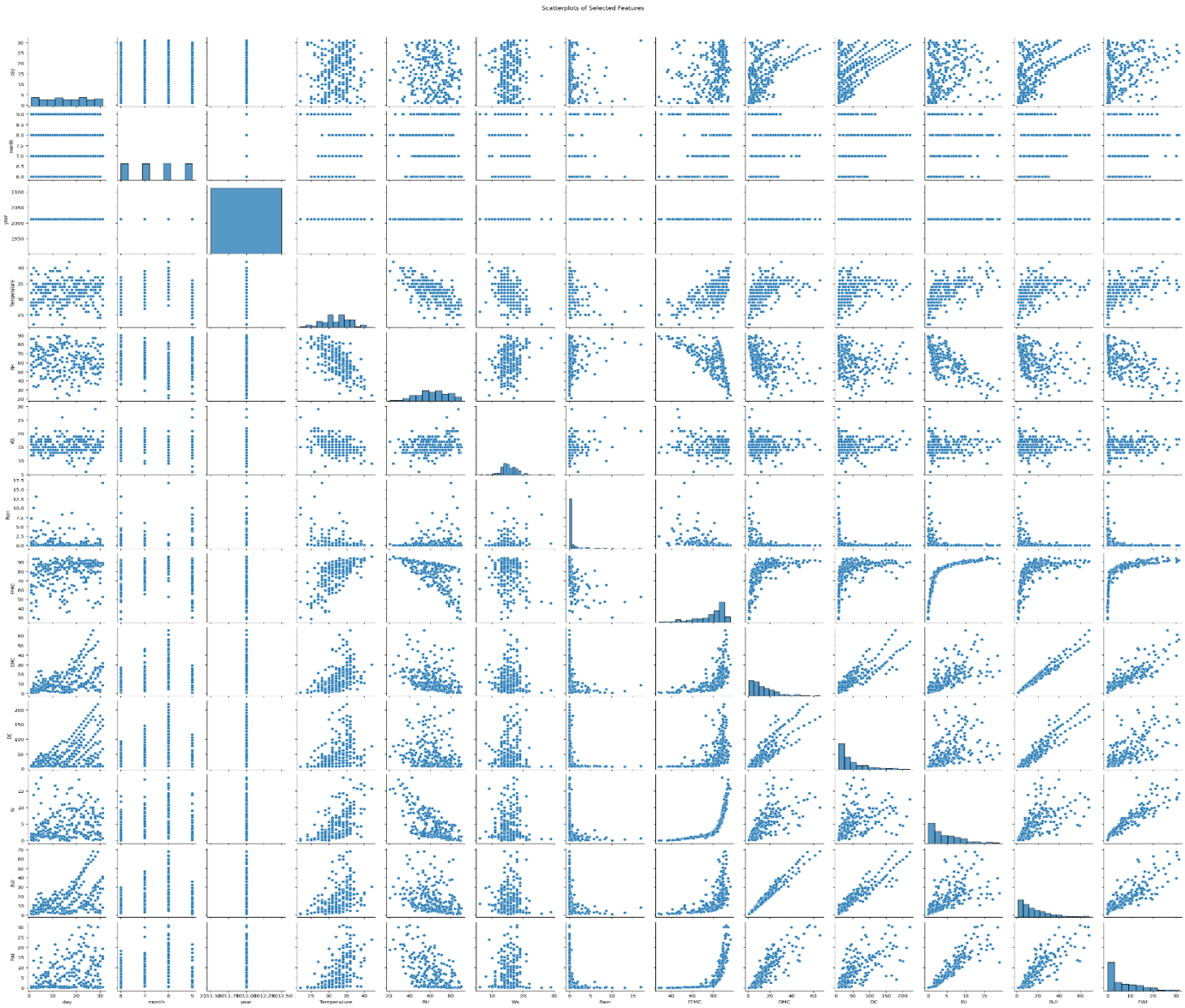

FIGURE 10: ScatterPlots

Pair plots were generated to visually analyze the relationships between multiple numerical features in the dataset. These plots display scatterplots for every pair of variables along with individual feature distributions on the diagonal. This makes it easier to observe how different environmental and fire-danger features interact with each other.

- Variables such as **FFMC, DMC, ISI, and Temperature** show noticeable upward trends when paired with one another, indicating strong positive relationships linked to fire-prone conditions.
- **Relative Humidity (RH)** demonstrates an inverse pattern with several fire indices, reflecting the natural decrease in fire risk when humidity levels are higher.
- Some features, such as **Rain and Wind Speed**, show more scattered relationships, suggesting weaker correlations with the Fire Weather Index (FWI).
- The diagonal plots reveal each feature's distribution, helping identify skewness, spread, and potential outliers.

```
#step 5: Encoding Categorical Variables
df.columns = df.columns.str.strip()

print("Unique values in Region:", df['Region'].unique())
le = LabelEncoder()

df['Region_encoded'] = le.fit_transform(df['Region'])
df[['Region', 'Region_encoded']].head()
```

	Region	Region_encoded
0	Bejaia	0
1	Bejaia	0
2	Bejaia	0
3	Bejaia	0
4	Bejaia	0

FIGURE 11: Categorical Variables

Milestone 2

Module 3: Feature Engineering and Scaling

Feature Selection

- Correlation analysis was used to identify input features that have the strongest relationship with the FWI target.
- Highly correlated features improve model accuracy and reduce noise.

```
#Module 3:Feature Engineering and Scaling
#Step 1:Select numerical columns
df = df.dropna()
corr = df.corr(numeric_only=True)['FWI'].abs()
selected_features = corr[corr > 0.3].index.tolist()
selected_features.remove('FWI')

if 'day' in selected_features:
    selected_features.remove('day')

if 'Ws' not in selected_features:
    selected_features.append('Ws')

X = df[selected_features]
y = df['FWI']

print("selected features:")
print(X.columns.tolist())
```

```
selected features:
['Temperature', 'RH', 'Rain', 'FFMC', 'DMC', 'DC', 'ISI', 'BUI', 'Ws']
```

Figure 12

Splitting Input and Output

- Input features were stored in X
- Target variable (FWI) was stored in y

Train-Test Split

- The dataset was divided into training and testing sets using train_test_split.
- This ensures unbiased evaluation of model performance.

```
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

X_scaled_df = pd.DataFrame(X_train_scaled,
columns=selected_features)
print("\nScaled Feature Summary:\n")
print(X_scaled_df.describe())
```

```
Scaled Feature Summary:

        Temperature            RH          Rain          FFMC            DMC  \
count  1.940000e+02  1.940000e+02  1.940000e+02  1.940000e+02  1.940000e+02
mean  -3.021638e-16  1.922860e-16  5.379431e-17 -1.327689e-16 -7.325183e-17
std    1.002587e+00  1.002587e+00  1.002587e+00  1.002587e+00  1.002587e+00
min   -2.796647e+00 -2.722316e+00 -3.801557e-01 -3.514355e+00 -1.095782e+00
25%   -6.012013e-01 -6.540988e-01 -3.801557e-01 -4.968853e-01 -7.204398e-01
50%    2.220908e-01  7.978492e-02 -3.801557e-01  3.835060e-01 -3.067969e-01
75%    7.709523e-01  7.469519e-01 -1.520870e-01  7.669022e-01  4.841035e-01
max    2.691967e+00  1.881136e+00  7.686274e+00  1.270997e+00  3.883252e+00

                DC           ISI           BUI            Ws
count  1.940000e+02  1.940000e+02  1.940000e+02  1.940000e+02
mean   9.614302e-17 -4.921607e-17 -9.156479e-17  1.671057e-16
std    1.002587e+00  1.002587e+00  1.002587e+00  1.002587e+00
min   -8.894062e-01 -1.143565e+00 -1.060395e+00 -2.896903e+00
25%   -8.216428e-01 -8.138422e-01 -7.529531e-01 -6.154450e-01
50%   -3.633611e-01 -2.990111e-01 -3.391536e-01 -2.352019e-01
75%    5.150539e-01  6.091740e-01  4.352665e-01  5.252842e-01
max    3.397259e+00  3.252745e+00  3.366762e+00  3.947472e+00
```

Figure 13

Saving the Scaler

The trained scaler was saved as scaler.pkl to ensure consistent preprocessing during deployment.

```
with open("scaler.pkl", "wb") as f:
    pickle.dump(scaler, f)
print("Training data shape:", X_train_scaled.shape)
print("Testing data shape:", X_test_scaled.shape)
```

```
Training data shape: (194, 9)
Testing data shape: (49, 9)
```

Figure 14

Module 4: Model Training using Ridge Regression

Ridge Regression is a regularized linear regression technique used to reduce overfitting and effectively handle multicollinearity in the input data. It is particularly suitable for weather-related datasets, where input features such as temperature, humidity, wind speed, and rainfall are often highly correlated with each other. Ridge Regression addresses this issue by adding an L2 penalty term to the loss function, which restricts the magnitude of model coefficients and improves generalization. The regularization strength is controlled by the alpha parameter, and multiple alpha values were tested to achieve an optimal balance between bias and variance. Model performance was evaluated using Mean Squared Error (MSE) and R^2 score, with both training and validation results monitored to ensure consistent learning and avoid overfitting. After selecting the best-performing model, the trained Ridge Regression model was saved as ridge.pkl using the pickle library. Additionally, all hyperparameters, evaluation metrics, and preprocessing steps were carefully documented to ensure reproducibility and support future deployment.

```
#Module 4: Model Training using Ridge Regression
ridge = Ridge(alpha=1.0)
ridge.fit(X_train_scaled, y_train)

# Make predictions
y_pred = ridge.predict(X_test_scaled)

# Evaluate the model
mae = mean_absolute_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)

print(f"Ridge Regression MAE: {mae:.4f}")
print(f"Ridge Regression R2 Score: {r2:.4f}")
# Save the tuned model
with open("ridge.pkl", "wb") as f:
    pickle.dump(ridge, f)
print("Tuned Ridge model saved as ridge.pkl")
```

```
Ridge Regression MAE: 0.4769
Ridge Regression R2 Score: 0.9814
Tuned Ridge model saved as ridge.pkl
```

Figure 15

```
# Model Comparison
from sklearn.linear_model import LinearRegression, Lasso,
ElasticNet
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import RandomForestRegressor
models = {
    "Linear Regression": LinearRegression(),
    "Lasso Regression": Lasso(alpha=0.1),
```

```
"ElasticNet": ElasticNet(alpha=0.1),
    "Decision Tree": DecisionTreeRegressor(random_state=42),
    "Random Forest": RandomForestRegressor(random_state=42)
}

results = []

for name, model in models.items():
    model.fit(X_train_scaled, y_train)
    y_pred = model.predict(X_test_scaled)

    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
results_df = pd.DataFrame(results, columns=['Model', 'MAE',
'RMSE', 'R2'])
print("Model Comparison Results:")
print(results_df)
```

```
Model Comparison Results:
              Model       MAE      RMSE        R2
0  Linear Regression  0.424018  0.596185  0.988273
1   Lasso Regression  0.548091  0.733245  0.982261
2         ElasticNet  0.719575  1.014471  0.966044
3      Decision Tree  0.912245  1.635137  0.911784
4      Random Forest  0.546714  0.833029  0.977104
```

Figure 16

Milestone 3

Module 5: Evaluation and Optimization

Mean Absolute Error (MAE) is a regression evaluation metric that measures the average absolute difference between the actual values and the predicted values produced by a model.

```
# Module 5
# Training and Testing Accuracy of Ridge Regression
y_train_pred = ridge.predict(X_train_scaled)
y_test_pred = ridge.predict(X_test_scaled)
train_accuracy = r2_score(y_train, y_train_pred)
test_accuracy = r2_score(y_test, y_test_pred)
print(f"Training Accuracy (R² Score): {train_accuracy:.4f}")
print(f"Testing Accuracy  (R² Score): {test_accuracy:.4f}")
```

Output:

```
Training Accuracy (R² Score): 0.9729
Testing Accuracy  (R² Score): 0.9814
```

Figure 17

- Training $R^2 = 0.9729$

  The model explains 97.29% of the variance in the training data.
- Testing $R^2 = 0.9814$

  The model explains 98.14% of the variance in unseen test data.
- Testing accuracy is slightly higher than training

  Indicates excellent generalization and no overfitting.

Root Mean Squared Error (RMSE) is a regression evaluation metric that measures the square root of the average of the squared differences between actual values and predicted values. It indicates how far, on average, the predictions are from the true values, with larger errors being penalized more heavily.

```
  from sklearn.metrics import mean_absolute_error, mean_squared_error,
r2_score
  import numpy as np

  # Training metrics
  train_mae = mean_absolute_error(y_train, y_train_pred)
  train_rmse = np.sqrt(mean_squared_error(y_train, y_train_pred))
  train_r2 = r2_score(y_train, y_train_pred)

  # Testing metrics
  test_mae = mean_absolute_error(y_test, y_test_pred)
  test_rmse = np.sqrt(mean_squared_error(y_test, y_test_pred))
  test_r2 = r2_score(y_test, y_test_pred)

  print("Training Metrics")
  print("MAE :", train_mae)
  print("RMSE:", train_rmse)
  print("R²  :", train_r2)

  print("\nTesting Metrics")
  print("MAE :", test_mae)
  print("RMSE:", test_rmse)
  print("R²  :", test_r2)
```

Output:

```
Training Metrics
MAE : 0.6793297885894147
RMSE: 1.2815126950299722
R²  : 0.972931326425548

Testing Metrics
MAE : 0.4769024031723181
RMSE: 0.7513475825725479
R²  : 0.9813740484635797
```

Figure 18

- The high R^2 scores for both training and testing indicate that the Ridge Regression model captures the underlying patterns effectively.
- Lower testing errors (MAE and RMSE) show that the model generalizes well and is not overfitting.
- The results validate the choice of Ridge Regression for handling multicollinearity in FWI-related features.

R^2 score measures the proportion of variance in the dependent variable that is predictable from the independent variables. Its value ranges from 0 to 1, where a higher value indicates better model performance.

```
models = {
    "Linear Regression": LinearRegression(),
    "Lasso Regression": Lasso(alpha=0.1),
    "ElasticNet": ElasticNet(alpha=0.1),
    "Decision Tree": DecisionTreeRegressor(random_state=42),
    "Random Forest": RandomForestRegressor(random_state=42)
}

results = []

for name, model in models.items():

    model.fit(X_train, y_train)
    y_train_pred = model.predict(X_train)
    y_test_pred = model.predict(X_test)

    # Metrics
    train_r2 = r2_score(y_train, y_train_pred)
    test_r2 = r2_score(y_test, y_test_pred)
    mae = mean_absolute_error(y_test, y_test_pred)
    rmse = np.sqrt(mean_squared_error(y_test, y_test_pred))
    results.append([name, train_r2, test_r2, mae, rmse])
```

```
# Results DataFrame
results_df = pd.DataFrame(
    results,
    columns=['Model', 'Train R2', 'Test R2', 'MAE', 'RMSE']
)

print("Model Comparison Results:")
results_df
```

Output:

```
Model Comparison Results:
```

	Model	Train R2	Test R2	MAE	RMSE
0	Linear Regression	0.973084	0.988273	0.424018	0.596185
1	Lasso Regression	0.972916	0.986094	0.446964	0.649201
2	ElasticNet	0.972959	0.985592	0.453632	0.660809
3	Decision Tree	1.000000	0.911784	0.912245	1.635137
4	Random Forest	0.996727	0.977309	0.539816	0.829291

Figure 19

predicted vs actual values

To visualize model performance, predicted values were plotted against actual values of the target variable (FWI).

This plot helps in understanding how closely the model's predictions match the real values. Ideally, if the model performs well, the points should lie close to the diagonal reference line, indicating minimal prediction error.

```
# Actual vs Predicted Plot
plt.figure(figsize=(7,5))
plt.scatter(y_test, y_test_pred,alpha=0.7)
plt.plot([y_test.min(), y_test.max()],
         [y_test.min(), y_test.max()],
         linestyle='--')
plt.xlabel("Actual FWI")
```

```
plt.ylabel("Predicted FWI")
plt.title("Actual vs Predicted FWI Values")
plt.show()
```

Output:

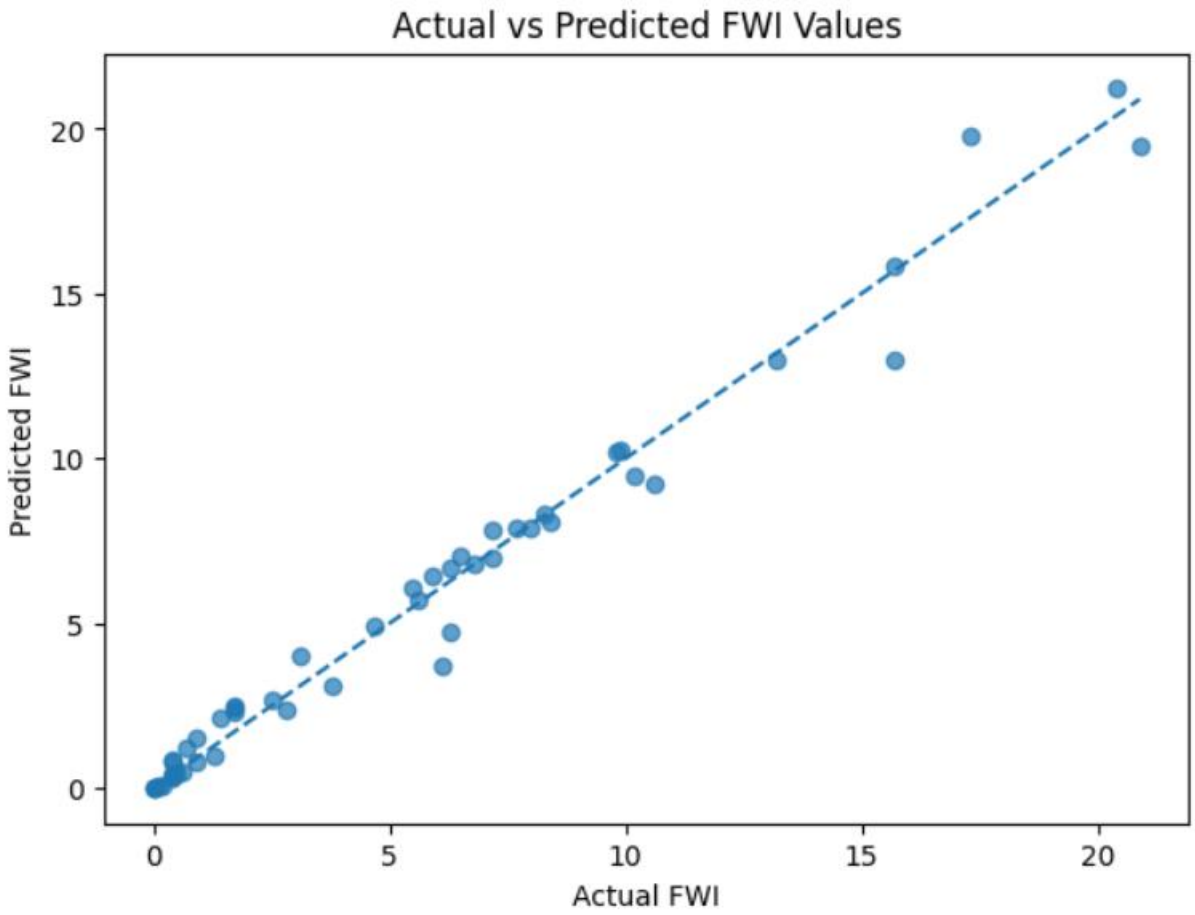

Figure 20: Actual Vs Predicted FWI Values

```
# Hyperparameter tuning for Ridge Regression alpha

from sklearn.model_selection import GridSearchCV
param_grid = {'alpha': [0.01, 0.1, 1, 10, 100]}
grid_search = GridSearchCV(Ridge(), param_grid, cv=5, scoring='r2')
grid_search.fit(X_train_scaled, y_train)

best_alpha = grid_search.best_params_['alpha']
```

```
print("Best Alpha:", best_alpha)

# Retrain model with best alpha
ridge_model = Ridge(alpha=best_alpha)
ridge_model.fit(X_train_scaled, y_train)

y_pred = ridge_model.predict(X_test_scaled)

mae = mean_absolute_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)

print(f"Tuned Ridge Regression MAE: {mae:.4f}")
print(f"Tuned Ridge Regression R2 Score: {r2:.4f}")

# Save the tuned model
with open("ridge_tuned.pkl", "wb") as f:
    pickle.dump(ridge_model, f)
print("Tuned Ridge model saved as ridge_tuned.pkl")
```

Output:

```
Best Alpha: 10
Tuned Ridge Regression MAE: 0.6771
Tuned Ridge Regression R2 Score: 0.9677
Tuned Ridge model saved as ridge_tuned.pkl
```

Figure 21