Imports

```
# Data Handling & Visualization
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt

# Preprocessing
from sklearn.preprocessing import PolynomialFeatures, StandardScaler
from sklearn.model_selection import train_test_split, GridSearchCV, RandomizedSearchCV, KFold
from sklearn.metrics import mean_squared_error, r2_score

# Models
from sklearn.ensemble import RandomForestRegressor, GradientBoostingRegressor
from xgboost import XGBRegressor

# Utilities
from scipy.stats import randint
import shap
import joblib
import gradio as gr
```

Data Handeling

```
# Step 1: Load the dataset
file_path = "data_500_cl_wm_rt_eq.csv"  # Update path if necessary
data = pd.read_csv(file_path)

# Show the first few rows of the dataset
print("First 5 rows of the dataset:")
print(data.head())
```

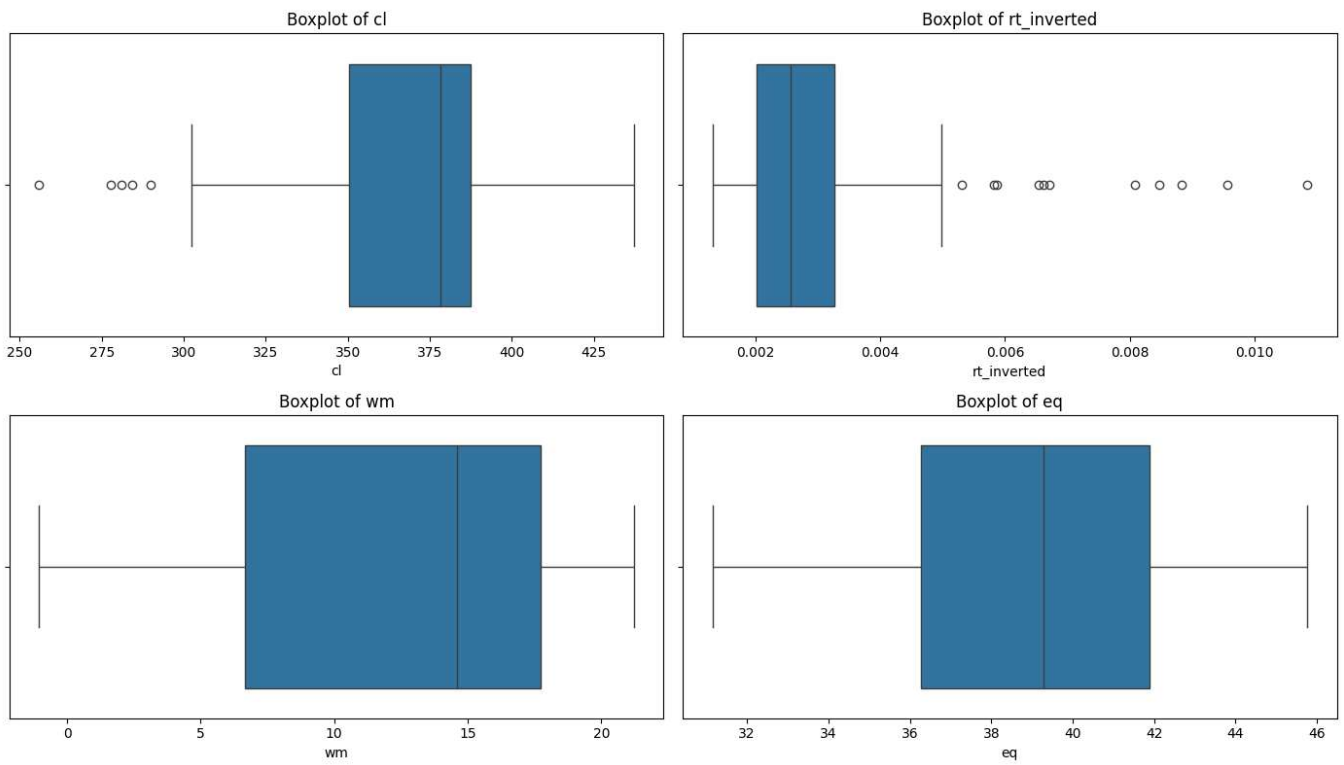
```
First 5 rows of the dataset:
           cl         rt         wm         eq
0  394.54790  340.60360   1.521294  39.220715
1  386.34042  278.84058   1.981495  39.195120
2  437.21225  520.28296   1.433642  39.446140
3  346.49445  392.56418  17.386406  32.108770
4  378.06738  586.46510  20.535595  42.982193
```

```
# Step 2.1: Invert Reaction Time (so higher = better)
data['rt_inverted'] = 1 / (data['rt'] + 1e-6)
data.drop(columns=['rt'], inplace=True)
print(data.head())
```

```
           cl         wm         eq  rt_inverted
0  394.54790   1.521294  39.220715     0.002936
1  386.34042   1.981495  39.195120     0.003586
2  437.21225   1.433642  39.446140     0.001922
3  346.49445  17.386406  32.108770     0.002547
4  378.06738  20.535595  42.982193     0.001705
```

```
# Select only the relevant numeric columns
features = ['cl', 'rt_inverted', 'wm', 'eq']

# Plot boxplots and calculate outliers
outlier_counts = {}

plt.figure(figsize=(14, 8))
for i, col in enumerate(features, 1):
    plt.subplot(2, 2, i)
    sns.boxplot(x=data[col], orient="h")
    plt.title(f'Boxplot of {col}')

    # IQR method
    Q1 = data[col].quantile(0.25)
    Q3 = data[col].quantile(0.75)
    IQR = Q3 - Q1
    lower_bound = Q1 - 1.5 * IQR
    upper_bound = Q3 + 1.5 * IQR
    outliers = data[(data[col] < lower_bound) | (data[col] > upper_bound)]
    outlier_counts[col] = len(outliers)

plt.tight_layout()
```

```
plt.show()

print("Outlier counts per feature:")
print(outlier_counts)
```




```
Outlier counts per feature:
{'cl': 5, 'rt_inverted': 11, 'wm': 0, 'eq': 0}
```

```
#Remove outliers

# Calculate IQR for 'cl'
Q1 = data['cl'].quantile(0.25)
Q3 = data['cl'].quantile(0.75)
IQR = Q3 - Q1

# Define bounds for outliers
lower_bound = Q1 - 1.5 * IQR
upper_bound = Q3 + 1.5 * IQR

# Print bounds (optional)
print(f"Lower Bound: {lower_bound}, Upper Bound: {upper_bound}")

# Filter out the outliers
data = data[(data['cl'] >= lower_bound) & (data['cl'] <= upper_bound)]

# Confirm removal
print("Updated shape of data:", data.shape)
```

```
Lower Bound: 295.1078462499999, Upper Bound: 442.75935625000005
Updated shape of data: (495, 4)
```

```
# Select features
X_base = data[['cl', 'wm', 'rt_inverted']]

# Generate polynomial and interaction features up to degree 2
poly = PolynomialFeatures(degree=2, include_bias=False)
X_poly = poly.fit_transform(X_base)

# Get feature names for reference
feature_names = poly.get_feature_names_out(['cl', 'wm', 'rt_inverted'])

# Create a DataFrame for better readability
X_poly_df = pd.DataFrame(X_poly, columns=feature_names)

# Check the new feature set
print("Polynomial Feature Names:")
print(feature_names)
```

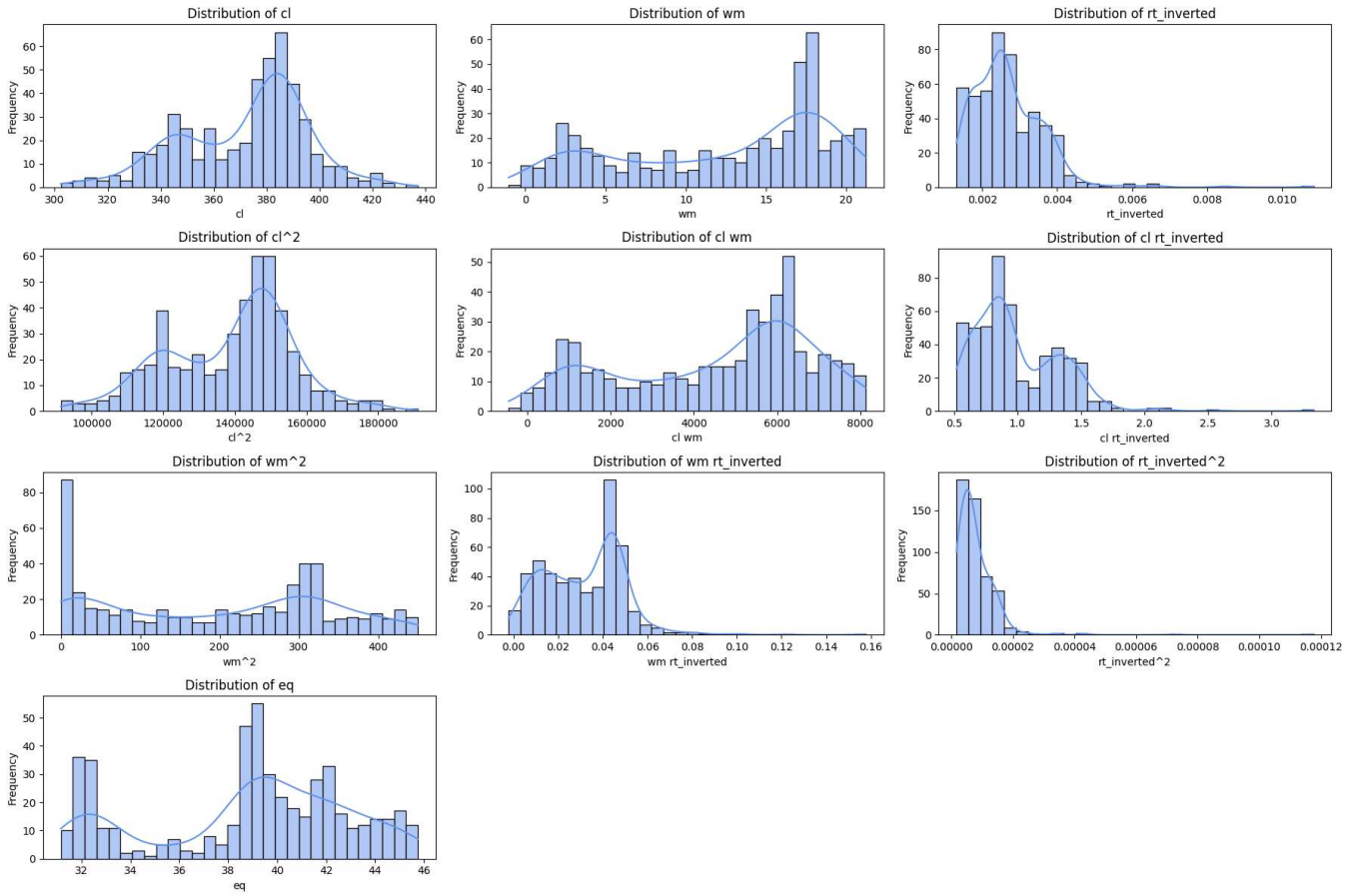
```
# Optionally, concatenate with the original target
X_poly_df['eq'] = data['eq'].values
```

```
Polynomial Feature Names:
['cl' 'wm' 'rt_inverted' 'cl^2' 'cl wm' 'cl rt_inverted' 'wm^2'
 'wm rt_inverted' 'rt_inverted^2']
```

```
print(X_poly_df.head())
```

```
           cl         wm  rt_inverted           cl^2        cl wm  \
0  394.54790   1.521294     0.002936  155668.045394   600.223156
1  386.34042   1.981495     0.003586  149258.920126   765.531495
2  437.21225   1.433642     0.001922  191154.551550   626.806063
3  346.49445  17.386406     0.002547  120058.403881  6024.293184
4  378.06738  20.535595     0.001705  142934.943820  7763.838598

   cl rt_inverted        wm^2  wm rt_inverted  rt_inverted^2         eq
0        1.158379    2.314334        0.004466       0.000009  39.220715
1        1.385524    3.926321        0.007106       0.000013  39.195120
2        0.840336    2.055331        0.002756       0.000004  39.446140
3        0.882644  302.287114        0.044289       0.000006  32.108770
4        0.644655  421.710662        0.035016       0.000003  42.982193
```

```
# Plot all 9 feature distributions
plt.figure(figsize=(18, 12))

for idx, feature in enumerate(X_poly_df.columns):
    plt.subplot(4, 3, idx + 1)
    sns.histplot(X_poly_df[feature], bins=30, kde=True, color='cornflowerblue')
    plt.title(f'Distribution of {feature}')
    plt.xlabel(feature)
    plt.ylabel('Frequency')

plt.tight_layout()
plt.show()
```



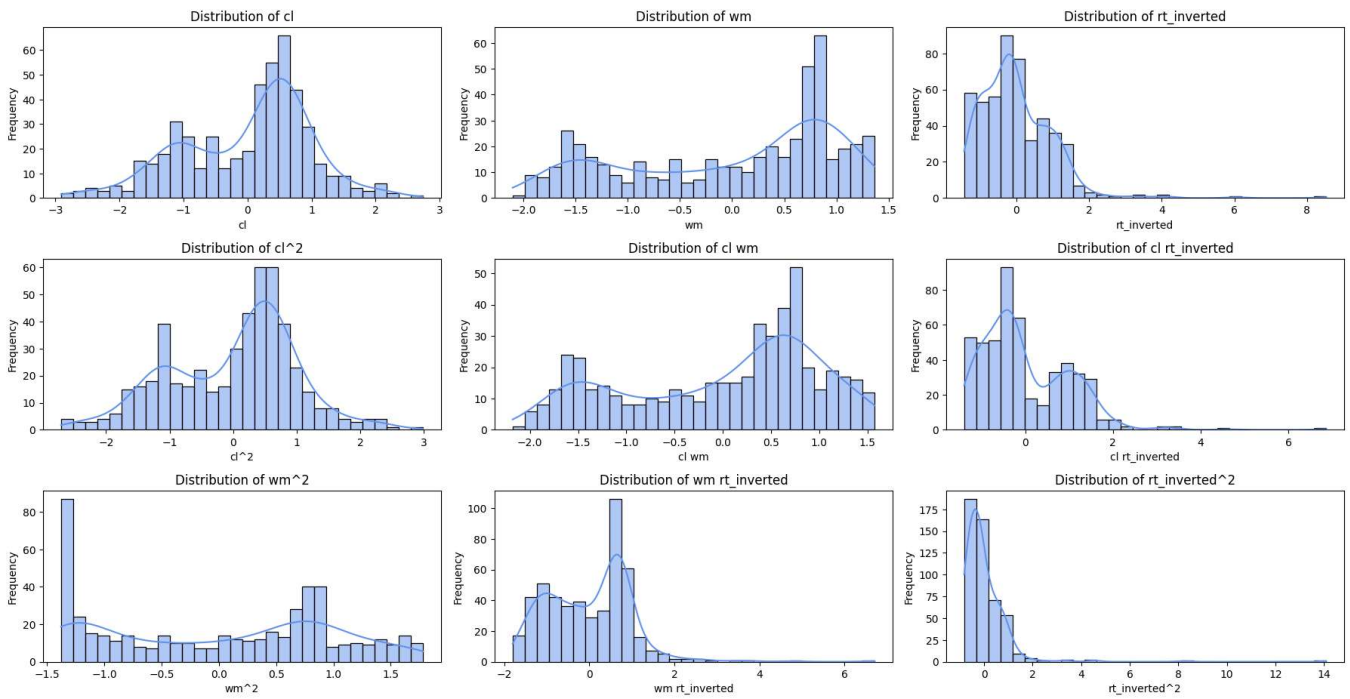

```
#Standerdization
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X_poly_df.drop(columns=['eq']))
```

```
# Convert back to DataFrame
```

```
X_scaled_df = pd.DataFrame(X_scaled, columns=X_poly_df.drop(columns=['eq']).columns)



# Preview scaled features
print("Scaled Features (first 5 rows):")
print(X_scaled_df.head())
Y=X_poly_df['eq']
print("\nTarget (first 5 rows):")
print(Y.head())
```

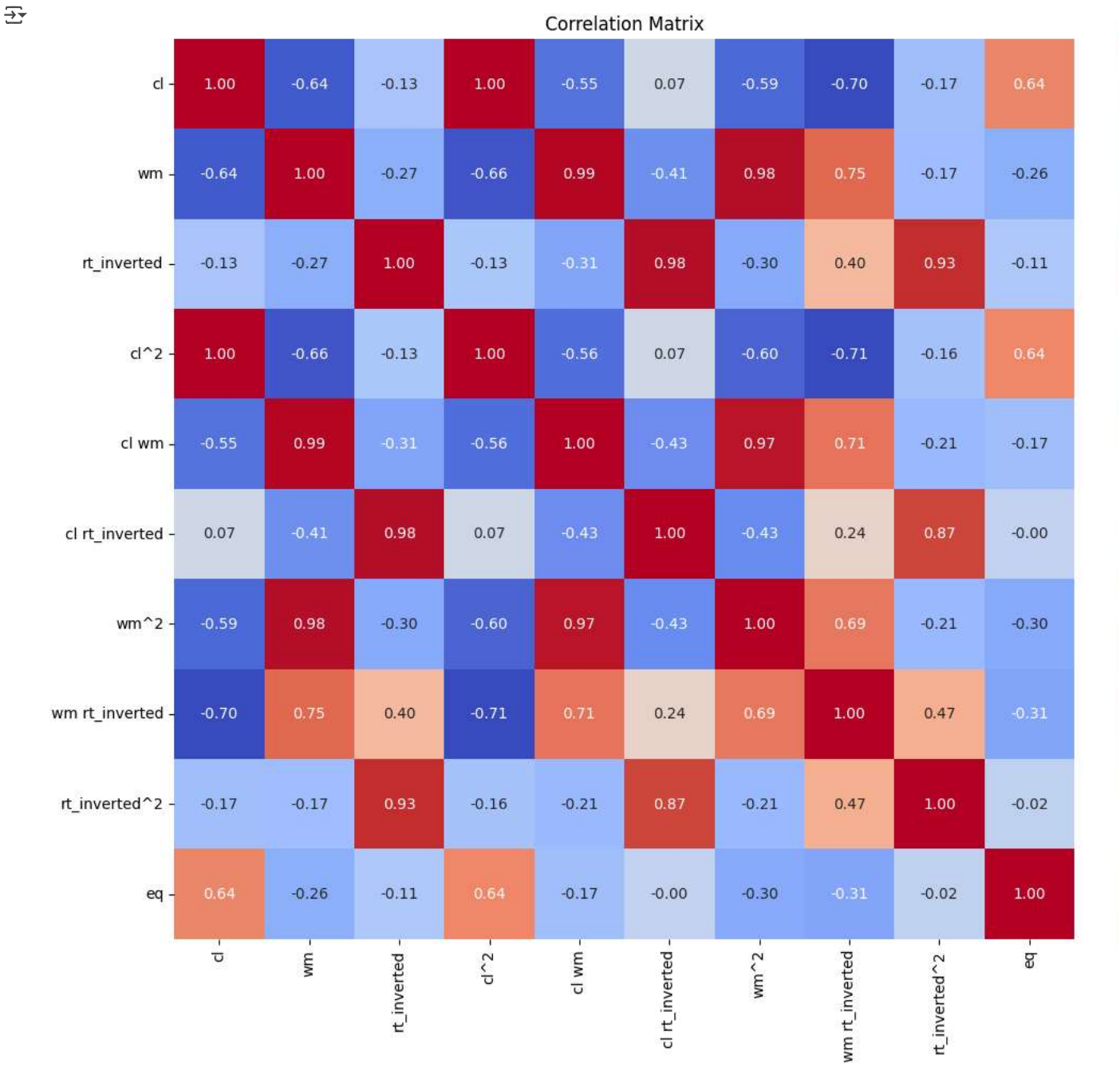
```
Scaled Features (first 5 rows):
         cl        wm  rt_inverted      cl^2     cl wm  cl rt_inverted  \
0  0.952348 -1.702139     0.266097  0.962044 -1.721757        0.483677
1  0.608487 -1.630475     0.945306  0.596133 -1.649351        1.151357
2  2.739816 -1.715788    -0.792886  2.988051 -1.710113       -0.451189
3 -1.060902  0.768436    -0.139779 -1.070992  0.654009       -0.326826
4  0.261879  1.258839    -1.019423  0.235082  1.415937       -1.026380

       wm^2  wm rt_inverted  rt_inverted^2
0 -1.362064       -1.449504       0.066200
1 -1.350745       -1.309004       0.612058
2 -1.363883       -1.540570      -0.567708
3  0.744353        0.670076      -0.208032
4  1.582949        0.176495      -0.668956

Target (first 5 rows):
0    39.220715
1    39.195120
2    39.446140
3    32.108770
4    42.982193
Name: eq, dtype: float64
```

```
# Plot all 9 feature distributions
plt.figure(figsize=(18, 12))

for idx, feature in enumerate(X_scaled_df.columns):
    plt.subplot(4, 3, idx + 1)
    sns.histplot(X_scaled_df[feature], bins=30, kde=True, color='cornflowerblue')
    plt.title(f'Distribution of {feature}')
    plt.xlabel(feature)
    plt.ylabel('Frequency')

plt.tight_layout()
plt.show()
```




```
# Compute correlation matrix
corr = X_poly_df.corr()

# Plot
plt.figure(figsize=(12, 10))
```

```
sns.heatmap(corr, annot=True, fmt='.2f', cmap='coolwarm', square=True)
plt.title("Correlation Matrix")
plt.tight_layout()
plt.show()
```




```
# X_poly_df already includes all engineered features
X = X_scaled_df  # Features
y = Y                # Target

X_train, X_temp, y_train, y_temp = train_test_split(X, Y, test_size=0.30, random_state=42)

# Step 2: Then split temp into validation and test (15% each)
X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.50, random_state=42)
```

RandomForestRegressor

```
# Step 1: Define model
rf = RandomForestRegressor(random_state=42)

# Step 2: Define hyperparameter grid
param_grid = {
    'n_estimators': [100, 200, 300],         # number of trees
    'max_depth': [None, 5, 10, 15],          # depth of tree
```

```
    'min_samples_split': [2, 5, 10],         # min samples to split
    'min_samples_leaf': [1, 2, 4],           # min samples per leaf
    'max_features': ['sqrt', 'log2']         # number of features to consider at each split
}

# Step 3: Set up K-Fold CV
kfold = KFold(n_splits=5, shuffle=True, random_state=42)

# Step 4: Grid Search with CV
rf_grid_search = GridSearchCV(estimator=rf,
                              param_grid=param_grid,
                              scoring='neg_mean_squared_error',  # since we want to reduce MSE
                              cv=kfold,
                              n_jobs=-1,
                              verbose=2)

# Step 5: Fit to your training data
rf_grid_search.fit(X_train, y_train)

# Step 6: Get best model and evaluate
best_rf = rf_grid_search.best_estimator_
y_val_pred = best_rf.predict(X_val)

# Step 7: Metrics
r2_val = r2_score(y_val, y_val_pred)
mse_val = mean_squared_error(y_val, y_val_pred)

print("✅ Best Parameters:", rf_grid_search.best_params_)
print(f"🎯 R² Score (Best RF): {r2_val}")
print(f"📉 MSE (Best RF): {mse_val}")
```

```
Fitting 5 folds for each of 216 candidates, totalling 1080 fits
✅ Best Parameters: {'max_depth': None, 'max_features': 'sqrt', 'min_samples_leaf': 1, 'min_samples_split': 2, 'n_estimators': 200}
🎯 R² Score (Best RF): 0.9541702437956228
📉 MSE (Best RF): 0.7542139077873043
```

```
# Define the model
rf = RandomForestRegressor(random_state=42)

# Extended search space
param_dist_rf = {
    'n_estimators': randint(100, 500),
    'max_depth': [None] + list(range(5, 30, 5)),
    'min_samples_split': randint(2, 10),
    'min_samples_leaf': randint(1, 7),
    'max_features': ['sqrt', 'log2', None]
}

kfold = KFold(n_splits=5, shuffle=True, random_state=42)

# Randomized Search
random_search_rf = RandomizedSearchCV(
    rf,
    param_distributions=param_dist_rf,
    n_iter=100,  # Try 100 combinations
    scoring='neg_mean_squared_error',
    cv=kfold,
    random_state=42,
    n_jobs=-1,
    verbose=2
)

random_search_rf.fit(X_train, y_train)

print("✅ Best RF Params:", random_search_rf.best_params_)
y_pred_rf = random_search_rf.best_estimator_.predict(X_val)
print("🎯 R² (RF):", r2_score(y_val, y_pred_rf))
print("📉 MSE (RF):", mean_squared_error(y_val, y_pred_rf))
```

```
Fitting 5 folds for each of 100 candidates, totalling 500 fits
✅ Best RF Params: {'max_depth': 10, 'max_features': None, 'min_samples_leaf': 1, 'min_samples_split': 2, 'n_estimators': 198}
🎯 R² (RF): 0.9463555880706945
📉 MSE (RF): 0.8828186074506977
```

GradientBoostingRegressor

```
# Step 1: Define the model
gbr = GradientBoostingRegressor(random_state=42)

# Step 2: Define hyperparameter grid
param_grid = {
    'n_estimators': [100, 200, 300],
    'learning_rate': [0.01, 0.05, 0.1],
    'max_depth': [3, 5, 7],
    'min_samples_split': [2, 5],
    'min_samples_leaf': [1, 2]
}

# Step 3: Define cross-validation strategy
kfold = KFold(n_splits=5, shuffle=True, random_state=42)

# Step 4: Grid search with cross-validation
gb_grid_search = GridSearchCV(
    estimator=gbr,
    param_grid=param_grid,
    scoring='neg_mean_squared_error',  # because we want to minimize MSE
    cv=kfold,
    n_jobs=-1,
    verbose=2
)

# Step 5: Fit on training data only
gb_grid_search.fit(X_train, y_train)

# Step 6: Evaluate on test set
best_gbr = gb_grid_search.best_estimator_
y_val_pred = best_gbr.predict(X_val)

# Step 7: Print metrics
r2 = r2_score(y_val, y_val_pred)
mse = mean_squared_error(y_val, y_val_pred)

print("✅ Best Parameters:", gb_grid_search.best_params_)
print(f"🎯 R² Score (Best GBR): {r2}")
print(f"📉 MSE (Best GBR): {mse}")
```

```
Fitting 5 folds for each of 108 candidates, totalling 540 fits
✅ Best Parameters: {'learning_rate': 0.1, 'max_depth': 3, 'min_samples_leaf': 2, 'min_samples_split': 5, 'n_estimators': 200}
🎯 R² Score (Best GBR): 0.9310371568556889
📉 MSE (Best GBR): 1.13491189409875
```

```
#Define the Model
gbr = GradientBoostingRegressor(random_state=42)

#Extended Search Space
param_dist_gbr = {
    'n_estimators': randint(100, 500),
    'max_depth': range(3, 10),
    'learning_rate': [0.01, 0.05, 0.1, 0.2],
    'min_samples_split': randint(2, 10),
    'min_samples_leaf': randint(1, 5)
}

#Randomized Search
random_search_gbr = RandomizedSearchCV(
    gbr,
    param_distributions=param_dist_gbr,
    n_iter=100,
    scoring='neg_mean_squared_error',
    cv=5,
    random_state=42,
    n_jobs=-1,
    verbose=2
)

random_search_gbr.fit(X_train, y_train)

print("✅ Best GBR Params:", random_search_gbr.best_params_)
y_pred_gbr = random_search_gbr.best_estimator_.predict(X_val)
print("🎯 R² (GBR):", r2_score(y_val, y_pred_gbr))
print("📉 MSE (GBR):", mean_squared_error(y_val, y_pred_gbr))
```

```
Fitting 5 folds for each of 100 candidates, totalling 500 fits
✅ Best GBR Params: {'learning_rate': 0.05, 'max_depth': 4, 'min_samples_leaf': 2, 'min_samples_split': 4, 'n_estimators': 307}
🎯 R² (GBR): 0.9156617665712601
📉 MSE (GBR): 1.3879425482104657
```

XGBRegressor

```
# Step 1: Define the model
xgb = XGBRegressor(objective='reg:squarederror', random_state=42)

# Step 2: Define hyperparameter grid
param_grid = {
    'n_estimators': [100, 200, 300],
    'learning_rate': [0.01, 0.05, 0.1],
    'max_depth': [3, 5, 7],
    'subsample': [0.7, 1.0],
    'colsample_bytree': [0.7, 1.0]
}
# Step 3: Define cross-validation strategy
cv_strategy = KFold(n_splits=5, shuffle=True, random_state=42)

# Step 4: Grid search with cross-validation
xg_grid_search = GridSearchCV(
    estimator=xgb,
    param_grid=param_grid,
    scoring='neg_mean_squared_error',
    cv=cv_strategy,
    n_jobs=-1,
    verbose=2
)

# Step 5: Fit on training data only
xg_grid_search.fit(X_train, y_train)

# Step 5: Fit on training data only
best_xgb = xg_grid_search.best_estimator_
y_val_pred = best_xgb.predict(X_val)

# Step 7: Print metrics
r2 = r2_score(y_val, y_val_pred)
mse = mean_squared_error(y_val, y_val_pred)

print("✅ Best Parameters:", xg_grid_search.best_params_)
print(f"🎯 R² Score (Best XGB): {r2}")
print(f"📉 MSE (Best XGB): {mse}")
```

```
Fitting 5 folds for each of 108 candidates, totalling 540 fits
✅ Best Parameters: {'colsample_bytree': 0.7, 'learning_rate': 0.05, 'max_depth': 7, 'n_estimators': 200, 'subsample': 0.7}
🎯 R² Score (Best XGB): 0.9432189724504375
📉 MSE (Best XGB): 0.934437453373226
```

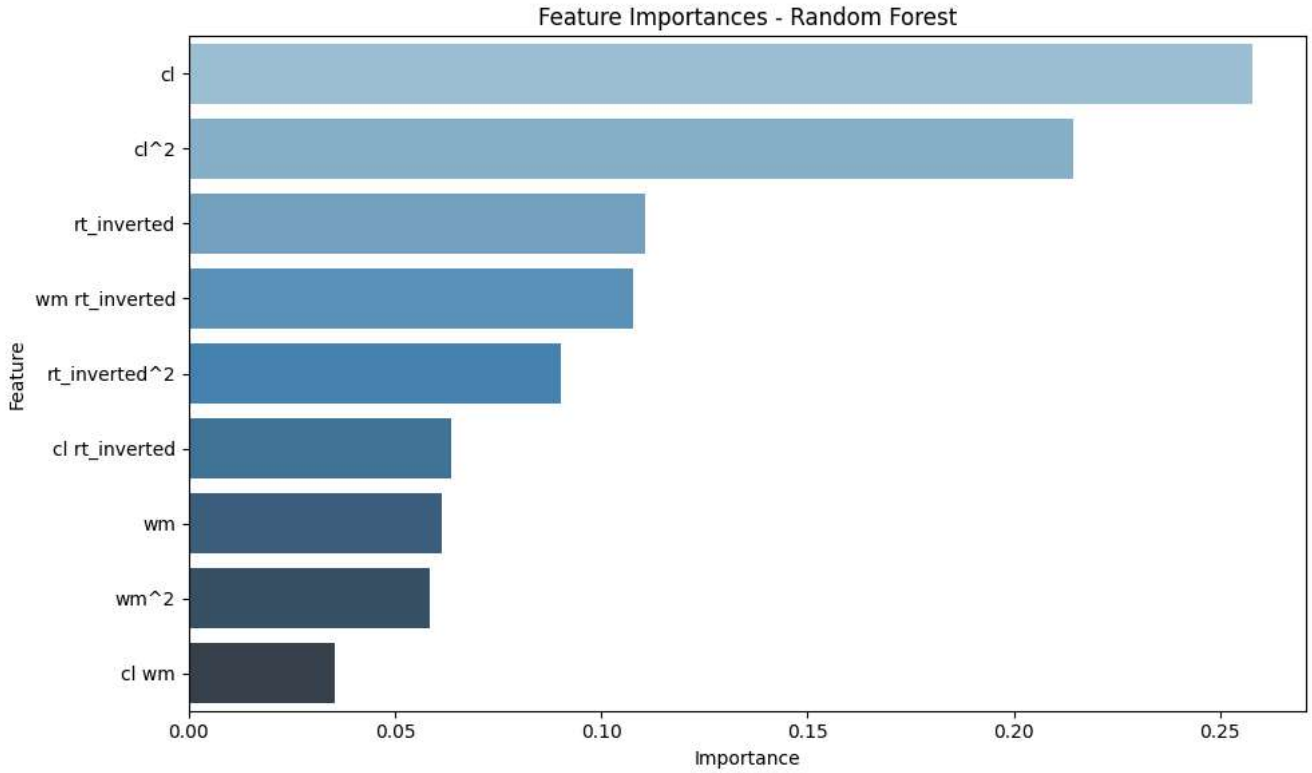
```
#define model
xgb = XGBRegressor(random_state=42)

#Extended Search Space
param_dist_xgb = {
    'n_estimators': randint(100, 500),
    'max_depth': range(3, 15),
    'learning_rate': [0.01, 0.05, 0.1, 0.2],
    'subsample': [0.6, 0.7, 0.8, 1.0],
    'colsample_bytree': [0.6, 0.8, 1.0]
}

#Randomized Search
random_search_xgb = RandomizedSearchCV(
    xgb,
    param_distributions=param_dist_xgb,
    n_iter=100,
    scoring='neg_mean_squared_error',
    cv=5,
    random_state=42,
    n_jobs=-1,
    verbose=2
)

random_search_xgb.fit(X_train, y_train)

print("✅ Best XGB Params:", random_search_xgb.best_params_)
y_pred_xgb = random_search_xgb.best_estimator_.predict(X_val)
print("🎯 R² (XGB):", r2_score(y_val, y_pred_xgb))
print("📉 MSE (XGB):", mean_squared_error(y_val, y_pred_xgb))
```

```
Fitting 5 folds for each of 100 candidates, totalling 500 fits
✅ Best XGB Params: {'colsample_bytree': 1.0, 'learning_rate': 0.1, 'max_depth': 8, 'n_estimators': 474, 'subsample': 0.7}
🎯 R² (XGB): 0.9325640372710 28
📉 MSE (XGB): 1.1097842359972916
```

Feature Importance (Random Forest)

```
# Get the best model from RandomizedSearchCV
best_rf = rf_grid_search.best_estimator_

# Get feature importances
importances = best_rf.feature_importances_
feature_names = X_train.columns  # or use poly.get_feature_names_out([...]) if using PolynomialFeatures

# Create DataFrame for easier plotting
feat_imp_df = pd.DataFrame({
    'Feature': feature_names,
    'Importance': importances
}).sort_values(by='Importance', ascending=False)

# Plot
plt.figure(figsize=(10, 6))
sns.barplot(x='Importance', y='Feature', data=feat_imp_df, palette='Blues_d')
plt.title('Feature Importances - Random Forest')
plt.tight_layout()
plt.show()
```

```
<ipython-input-136-330479700>:16: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be removed in v0.14.0. Assign the `y` variable to `hue` and set `le

  sns.barplot(x='Importance', y='Feature', data=feat_imp_df, palette='Blues_d')
```

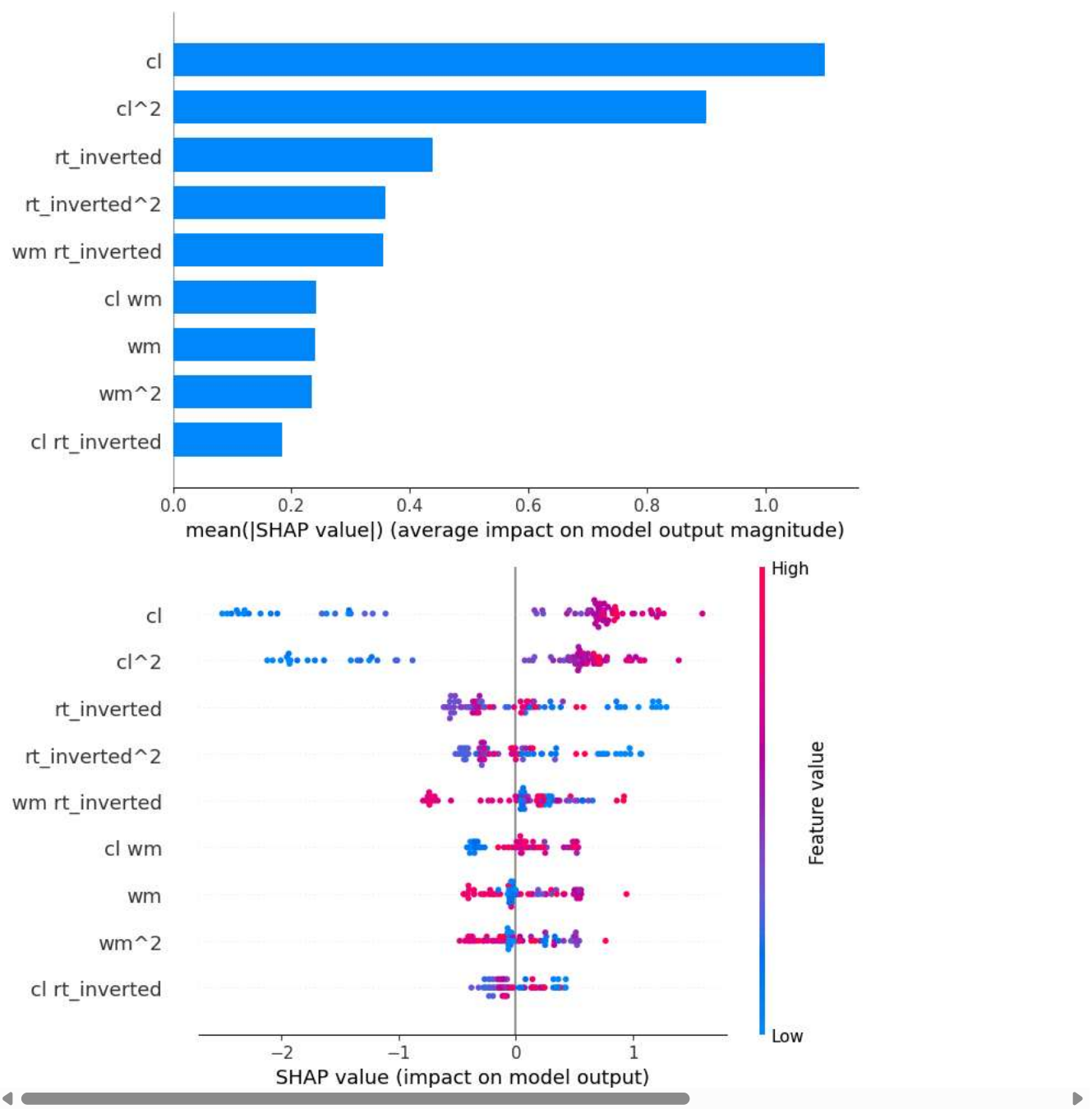


SHAP and Explainer

```
# Use TreeExplainer for tree-based models
explainer = shap.TreeExplainer(best_rf)  # Replace `best_rf` with your best model
```

```
# You can use a subset of your data for faster results
shap_values = explainer.shap_values(X_val)
```

```
shap.summary_plot(shap_values, X_val, plot_type="bar")  # Bar plot of average importance
shap.summary_plot(shap_values, X_val)  # Beeswarm plot for distribution of impacts
```



Testing

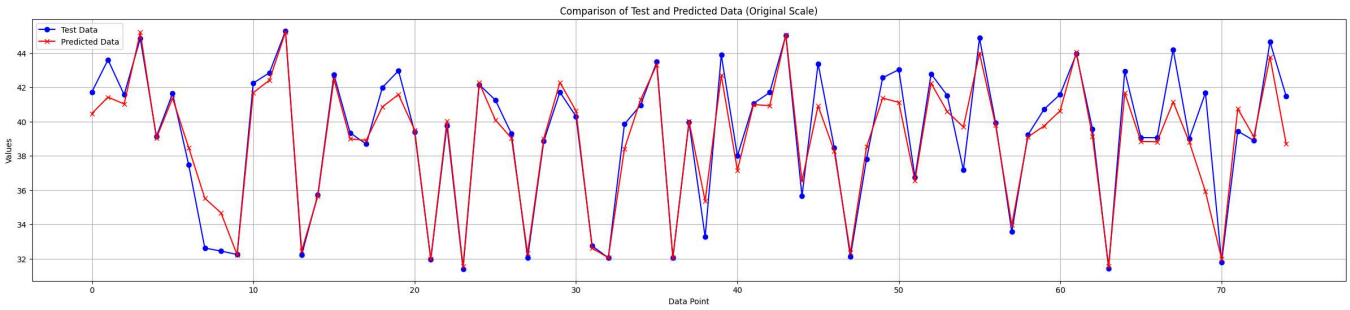
```
# Predict on test set using the best model (e.g., best_rf from RandomizedSearchCV)
y_test_pred = rf_grid_search.best_estimator_.predict(X_test)

# Metrics
test_r2 = r2_score(y_test, y_test_pred)
test_mse = mean_squared_error(y_test, y_test_pred)

print("🎯 R² Score on Test Set:", test_r2)
print("📉 MSE on Test Set:", test_mse)
```

```
🎯 R² Score on Test Set: 0.9082126659998242
📉 MSE on Test Set: 1.52419405269637
```

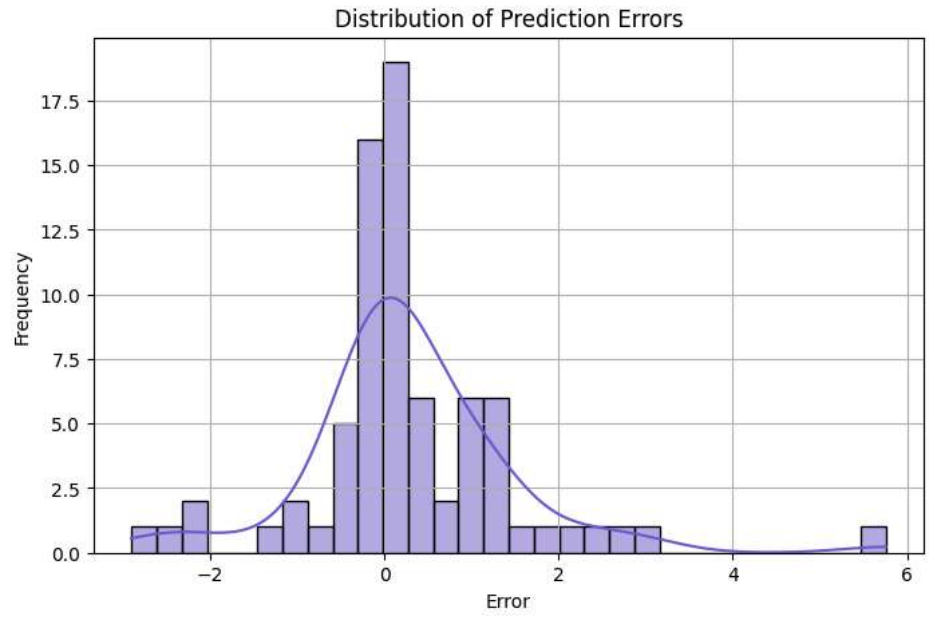
PLOTS

```
# Line Plot
plt.figure(figsize=(30, 6))
plt.plot(range(len(y_test)), y_test, marker='o', color='blue', label='Test Data')
```

```
plt.plot(range(len(y_test)), y_test, marker='o', color='blue', label='Test Data')
plt.plot(range(len(y_test_pred)), y_test_pred, marker='x', color='red', label='Predicted Data')
plt.xlabel('Data Point')
plt.ylabel('Values')
plt.title('Comparison of Test and Predicted Data (Original Scale)')
plt.legend()
plt.grid(True)
plt.show()
```




```
#Distribution of Prediction Errors
errors = y_test - y_test_pred
plt.figure(figsize=(8, 5))
sns.histplot(errors, bins=30, kde=True, color='slateblue')
plt.title('Distribution of Prediction Errors')
plt.xlabel('Error')
plt.ylabel('Frequency')
plt.grid(True)
plt.show()
```



DEMO

```
# Save the model and preprocessing steps
best_rf = rf_grid_search.best_estimator_
joblib.dump(best_rf, "best_rf_model.joblib")
joblib.dump(scaler, "scaler.joblib")
joblib.dump(poly, "poly.joblib")
```

```
['poly.joblib']
```

```
# Load the saved components
model = joblib.load("best_rf_model.joblib")
scaler = joblib.load("scaler.joblib")
poly = joblib.load("poly.joblib")  # this is your PolynomialFeatures

# Define the prediction function
def predict_eq(cl, wm, rt):
    # Invert RT as done in preprocessing
    rt_inverted = 1 / (rt + 1e-6)
```

```
    # Create input array
    input_features = np.array([[cl, wm, rt_inverted]])

    # Polynomial transformation
    input_poly = poly.transform(input_features)

    # Standardize
    input_scaled = scaler.transform(input_poly)

    # Predict
    prediction = model.predict(input_scaled)[0]
    return round(prediction, 3)

# Set up Gradio interface
interface = gr.Interface(
    fn=predict_eq,
    inputs=[
        gr.Number(label="Cognitive Load (cl)"),
        gr.Number(label="Working Memory (wm)"),
        gr.Number(label="Reaction Time (rt)"),
    ],
    outputs=gr.Number(label="Predicted Emotional Intelligence (eq)"),
    title="EI Predictor",
    description="Predict Emotional Intelligence based on Cognitive Load, Working Memory, and Reaction Time"
)

# Launch the app
interface.launch()
```

```
It looks like you are running Gradio on a hosted a Jupyter notebook. For the Gradio app to work, sharing must be enabled. Automatica

Colab notebook detected. To show errors in colab notebook, set debug=True in launch()
* Running on public URL: https://dd91faf98dec06b01f.gradio.live

This share link expires in 1 week. For free permanent hosting and GPU upgrades, run `gradio deploy` from the terminal in the working
```