



## **INTERNSHIP REPORT**

**ADITI NADIGER**

**1BM22AI005**



**DEPARTMENT OF MACHINE LEARNING**

**B.M.S COLLEGE OF ENGINEERING**

*(An Autonomous Institution Affiliated to VTU, Belagavi)*

**Post Box No.: 1908, Bull Temple Road, Bengaluru – 560 019**



### ***DECLARATION***

I, Aditi Nadiger (1BM22AI005) students of 5<sup>th</sup> Semester, B.E, Department of Artificial Intelligence and Machine Learning Engineering, BMS College of Engineering, Bangalore, hereby declare that, this internship has been carried out during the academic semester August - November 2024. I also declare that to the best of my knowledge and belief, the report is not from part of any other report by any other students.

#### **Signature of the Candidate**

ADITI NADIGER (1BM22AI005)

## Table of Contents

Sl. No.	Title	Page No.
1	Ch 1: Introduction	4
2	Ch 2: Problem statement	5
2	Ch 3: Dataset Summary	6
3	Ch 3: Software Requirements	7
4	Ch 4: Architecture	8
5	Ch 5: Implementation	9
6	Ch 6: Results and discussion	32
7	Ch 7: UI Design	42

## **Chapter 1: Introduction**

Agriculture is a cornerstone of global food security and economic stability, with soil quality playing a vital role in determining crop yield and sustainability. Effective soil nutrient analysis is essential for optimizing agricultural productivity, reducing resource wastage, and ensuring environmental conservation. Traditional soil testing methods, while accurate, are often time-consuming, labour-intensive, and costly, making them inaccessible to many farmers and agricultural stakeholders.

This project aims to address these challenges by leveraging advanced machine learning techniques to predict soil nutrient levels based on spectral data. By utilizing a dataset comprising multiple predictors and target variables, the model provides a comprehensive analysis of critical soil nutrients such as pH, organic carbon, and micronutrient concentrations. This approach offers a rapid, scalable, and cost-effective solution to soil testing, empowering stakeholders with actionable insights to make informed decisions about crop selection, fertilization, and land management.

In addition to developing a robust predictive model, this project incorporates feature selection techniques to identify the most influential factors affecting soil quality. The model's performance is evaluated using state-of-the-art algorithms, including Random Forest, Gradient Boosting, and AdaBoost, ensuring reliability and accuracy. Furthermore, an intuitive user interface is designed to facilitate ease of use, making the tool accessible to a broader audience, from researchers to farmers.

By integrating technology into agriculture, this project contributes to sustainable farming practices, optimizing resource utilization while minimizing environmental impact. The outcome is a step forward in achieving precision agriculture, enabling smarter and more sustainable agricultural practices for the future.

## **Chapter 2: Problem Statement**

Develop a system that enables accurate determination of nutrient concentrations based on the light wavelengths, proving effective for real-time field analysis and to develop a user-friendly interface to simplify the application of this predictive model for soil analysis and support agricultural decision-making.

## Chapter 2: Dataset Summary

### Data Characteristics

- 100 Rows: Each row represents a soil sample with measured spectral data and corresponding nutrient levels.
- Correlations: Likely strong relationships between specific spectral bands and nutrient levels.
- Data Type: Continuous numerical values for both predictors and target variables.

### Potential Analysis Insights

- Exploratory Analysis: Investigating the relationships between spectral data (predictors) and nutrient levels (targets) will highlight key bands that influence nutrient concentration.
- Multi-target Regression: The nature of the data suggests using a model that can predict multiple nutrients simultaneously.

Predictors (18 total): Spectral measurements at various wavelengths (A(410), B(435), C(460), ..., T(730)) corresponding to specific bands in the electromagnetic spectrum.

Target Variables (13 total): Key soil nutrients and properties including pH, EC (dS/m), OC (%), P (kg/ha), K (kg/ha), Ca (meq/100g), Mg (meq/100g), S (ppm), Fe (ppm), Mn (ppm), Cu (ppm), Zn (ppm), and B (ppm).

	A(410)	B(435)	C(460)	D(485)	E(510)	F(535)	G(560)	H(585)	I(610)	I(645)	...
0	2429.58	1108.99	2091.93	737.33	835.14	1376.82	322.87	318.87	1305.33	163.26	...
1	1707.90	725.38	1974.49	670.81	835.93	1685.28	370.31	373.60	1335.77	185.56	...
2	1750.41	1102.02	2481.05	816.80	960.65	1741.43	430.51	437.88	1206.99	227.04	...
3	2546.68	1548.40	2791.56	1007.16	1128.00	1573.73	488.66	499.63	1338.11	261.84	...
4	3090.54	1266.42	2933.88	940.61	1109.84	2047.64	500.91	512.70	1482.11	267.19	...

...	P (kg/ha)	K (kg/ha)	Ca (meq/100g)	Mg (meq/100g)	S (ppm)	Fe (ppm)	Mn (ppm)
...	26.10	444.00	6.14	2.32	11.21	3.08	14.10
...	81.99	372.00	5.98	0.50	12.93	47.74	37.63
...	80.59	132.00	3.15	2.49	5.17	14.96	44.53
...	33.81	221.76	3.40	1.90	11.59	6.38	10.62
...	38.19	234.08	6.60	5.20	34.10	14.08	3.56

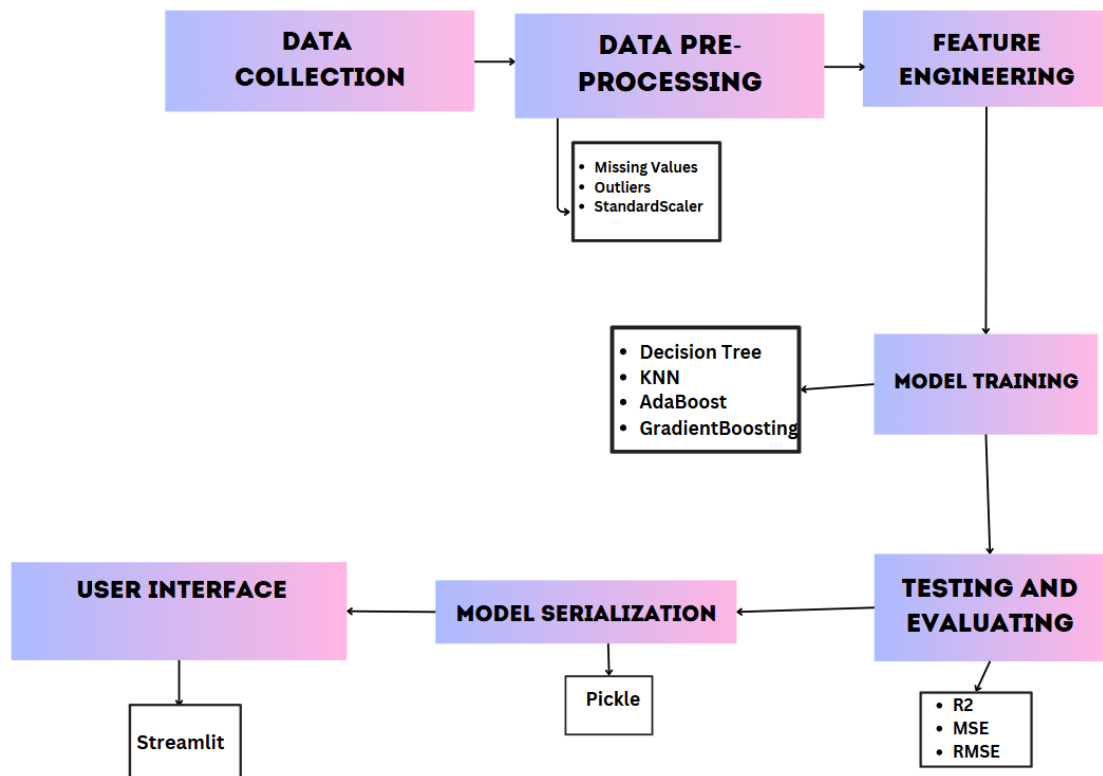
Fig 2.1

Fig 2.1 represents the dataset showing attributes from A(410) till I(645) and the target variables form P(kh/ha) to Mn(ppm).

## **Chapter 3: Software Requirements**

- Programming Language: Python
- Libraries and Frameworks:
  - Pandas, NumPy: Data manipulation and analysis
  - Scikit-learn: Machine learning algorithms and utilities
  - Matplotlib, Seaborn: Data visualization
  - Streamlit: User interface development
  - Pickle: Model serialization
- Development Environment:
  - Jupyter Notebook, VS Code

## Chapter 4: Architecture



The project architecture comprises the following components:

1. **Data Collection:** Input spectral measurements and corresponding nutrient data.
2. **Preprocessing:**
  - Handle missing values and outliers.
  - Standardize features using StandardScaler.
3. **Feature Engineering:**
  - Feature selection based on importance scores from Random Forest.
4. **Model Training and Evaluation:**
  - Train multi-output regression models like Random Forest, Elastic Net, Gradient Boosting, KNN, AdaBoost.
  - Evaluate using metrics such as  $R^2$ , MSE, and RMSE.
5. **Model Serialization:**
  - Save trained models using Pickle for deployment.
6. **User Interface:**
  - Develop an interactive UI using Streamlit for inputting data and visualizing predictions.



## Chapter 5: Implementation

Data preprocessing is a critical step in preparing the dataset for effective analysis and modelling. It involves cleaning, transforming, and organizing raw data to ensure it is suitable for machine learning algorithms. For this project, the preprocessing steps addressed issues such as missing values, outliers, and data standardization.

### 1. Handling Missing Values

- Since all the predictors and target variables are numerical, missing values were replaced using the **mean** of the respective columns. This approach ensures minimal impact on the overall data distribution while retaining sufficient variability for analysis.

```
#treating the null values
#Since all are numerical data, we can use mean as the measure the replace all the mising values
if df.isnull().values.any():
    # Calculate mean for numeric columns only
    numeric_cols = df.select_dtypes(include='number').columns
    df[numeric_cols] = df[numeric_cols].fillna(df[numeric_cols].mean())

# Change non-numeric values to numeric values
df = df.apply(pd.to_numeric, errors='coerce')
```

Fig 5.1

Fig 5.1 shows that since all are numerical data, we can use mean as the measure the replace all the missing values.

### 2. Outlier Detection and Treatment

- Outliers were identified using statistical measures such as interquartile ranges (IQR) and visualization tools like box plots.
- Fig 5.2 shows that, given the small number of outliers, no data points were removed. Instead, they were retained to preserve the integrity of the dataset and avoid potential bias introduced by manual removal.

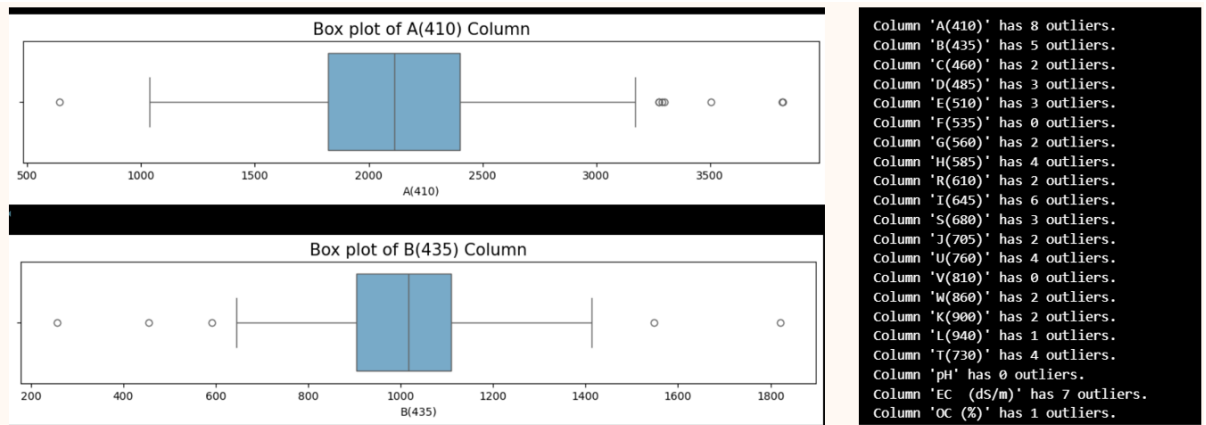


Fig 5.2

### 3. Feature Scaling

- Data scaling was performed using **StandardScaler**, which standardizes features by removing the mean and scaling them to unit variance.
- This ensures that all predictors are on the same scale, preventing features with larger numerical ranges from disproportionately influencing the model.

```
# Standardisation
# importing libraries for data transformation
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
xts = scaler.fit_transform(X) #X train and test scaled
```

Fig 5.3

### 4. Splitting the Dataset

- The dataset was divided into training (80%) and testing (20%) subsets using **train\_test\_split** with a fixed random state for reproducibility.
- This ensures that the model is trained on one portion of the data and evaluated on an unseen subset to assess generalization performance.

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(xts, y, test_size=0.2, random_state=42)
```

Fig 5.4

### 5. Data Augmentation

- To increase the robustness of the model, **noise addition** was applied to the feature set. This introduced subtle variations into the data, mimicking real-world scenarios and enhancing the model's ability to handle variability.

By systematically addressing these preprocessing steps, the dataset was prepared for further analysis and modeling, ensuring that the subsequent results are both reliable and accurate.

```
import numpy as np
import pandas as pd

# Function to add noise
def add_noise(data, noise_level=0.01):
    noise = np.random.normal(0, noise_level, data.shape)
    return data + noise

# Apply noise to feature columns
df_augmented = df.copy()
df_augmented[feature_columns] = add_noise(df[feature_columns])

# Combine original and augmented data
df_combined = pd.concat([df, df_augmented], ignore_index=True)
```

Fig 5.5

```
print("Samples in Training data ", len(X_train))
print("Samples in Testing data ", len(X_test))
```

```
Samples in Training data  160
Samples in Testing data   40
```

Fig 5.6

Fig 5.6 shows that the dataset now has 200 rows after data augmentation.

## Models Implemented

### 1. Random Forest Regressor for Multi-Target Regression

- RandomForestRegressor: A tree-based ensemble learning method that fits multiple decision trees and averages the results for regression tasks.
- MultiOutputRegressor: A wrapper that allows handling multiple target variables (multi-target regression) using individual regressors for each target.
- Metrics (MSE, MAE, RMSE, R<sup>2</sup>): Used to evaluate the model's performance.
- The model is trained using the training data (X\_train, y\_train), where X\_train contains the 18 predictors and y\_train contains the 13 target variables.

```
from sklearn.ensemble import RandomForestRegressor
from sklearn.multioutput import MultiOutputRegressor
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score

# Train Random Forest Regressor model for multiple target variables
rfr = RandomForestRegressor(random_state=42)

# Wrap Random Forest Regressor in MultiOutputRegressor to handle multiple outputs
multioutput_rfr = MultiOutputRegressor(rfr)

# Fit the Random Forest Regressor model
multioutput_rfr.fit(X_train, y_train)

# Predict for the test set
y_pred_rfr = multioutput_rfr.predict(X_test)
y_pred_train_rfr=multioutput_rfr.predict(X_train)
# Calculate performance metrics
mse_rfr = mean_squared_error(y_test, y_pred_rfr)
mae_rfr = mean_absolute_error(y_test, y_pred_rfr)
rmse_rfr = np.sqrt(mse_rfr)
r2_rfr = r2_score(y_test, y_pred_rfr)
r2_train_rfr = r2_score(y_train, y_pred_train_rfr)

print("Random Forest Regressor Performance Metrics:")
print(f"Mean Squared Error (MSE): {mse_rfr:.4f}")
print(f"Mean Absolute Error (MAE): {mae_rfr:.4f}")
print(f"Root Mean Squared Error (RMSE): {rmse_rfr:.4f}")
print(f"R2 Score: {r2_rfr:.4f}")
print(f"R2 Score on Training Data: {r2_train_rfr:.4f}")
```

```
Random Forest Regressor Performance Metrics:  
Mean Squared Error (MSE): 4283.1648  
Mean Absolute Error (MAE): 18.5869  
Root Mean Squared Error (RMSE): 65.4459  
R2 Score: -0.2098  
R2 Score on Training Data: 0.8218
```

This is for the dataset with 100 datapoints. It indicates that the model is overfitted

## 2. Hyperparameter Tuning with GridSearchCV

- Parameter Grid: Defines the set of hyperparameters to be optimized during model training:
- `n_estimators`: Number of trees in the forest.
- `max_depth`: Maximum depth of the trees.
- `min_samples_split`: Minimum number of samples required to split a node.
- `min_samples_leaf`: Minimum number of samples required to be a leaf node.
- GridSearchCV: Performs cross-validation (`cv=3`) across different hyperparameter combinations to find the best model.
- Scoring: The model is scored based on the  $R^2$  metric.

Hyperparameter tuning can be done after using a Random Forest Regressor. In fact, it's a common practice to optimize the performance of the model. While Random Forests are generally robust and can perform well with their default settings, fine-tuning the hyperparameters can lead to better model performance, especially for complex datasets.

```

# Define the parameter grid for GridSearchCV
param_grid = {
    'estimator__n_estimators': [50, 100, 200],
    'estimator__max_depth': [None, 10, 20, 30],
    'estimator__min_samples_split': [2, 5, 10],
    'estimator__min_samples_leaf': [1, 2, 4]
}

# Initialize GridSearchCV
grid_search = GridSearchCV(multioutput_rfr, param_grid, cv=3, scoring='r2', verbose=2, n_jobs=-1)

# Fit GridSearchCV
grid_search.fit(X_train, y_train)

# Best parameters from GridSearchCV
print("Best parameters found: ", grid_search.best_params_)

# Use the best estimator to predict
best_model = grid_search.best_estimator_
y_pred_rfr = best_model.predict(X_test)
y_pred_train_rfr = best_model.predict(X_train)

# Calculate R2 score for the training set
r2_train_rfr = r2_score(y_train, y_pred_train_rfr)

# Calculate R2 score for the test set
r2_rfr = r2_score(y_test, y_pred_rfr)

```

```

Fitting 3 folds for each of 108 candidates, totalling 324 fits
Best parameters found: {'estimator__max_depth': None, 'estimator__min_s
Random Forest Regressor Performance Metrics with Tuned Hyperparameters:
R2 Score on Training Data: 0.5068
R2 Score on Test Data: -0.1026
Mean Squared Error (MSE) on Test Data: 3809.4877
Mean Absolute Error (MAE) on Test Data: 17.6747
Root Mean Squared Error (RMSE) on Test Data: 61.7210

```

- Training Data: The model's fit has slightly improved in terms of  $R^2$ .
- Test Data: There's a slight increase in error metrics (MSE, MAE, RMSE), which suggests that the model might be overfitting the training data, and the new hyperparameters might have made the model more complex or sensitive to the data.

### 3. Feature Importance with Random Forest

- `feature_importances_per_target` = `[est.feature_importances_ for est in multioutput_rfr.estimators_]`: Extracts feature importance from each individual `RandomForestRegressor` model (one for each target variable). Each estimator corresponds to one target (e.g., pH, EC, etc.).
- Feature Importance: Provides insight into which features (predictors) are most relevant for each specific target variable.
- Visualization: Separate visualizations for each target variable allow deeper analysis into how different features impact the prediction of soil nutrients.

```
# Get feature importances (average across all outputs)
feature_importances = np.mean([tree.feature_importances_ for tree in multioutput_rfr.estimators_], axis=0)

# Create a DataFrame for better visualization
feature_importance_df = pd.DataFrame({
    'Feature': X.columns,
    'Importance': feature_importances
})

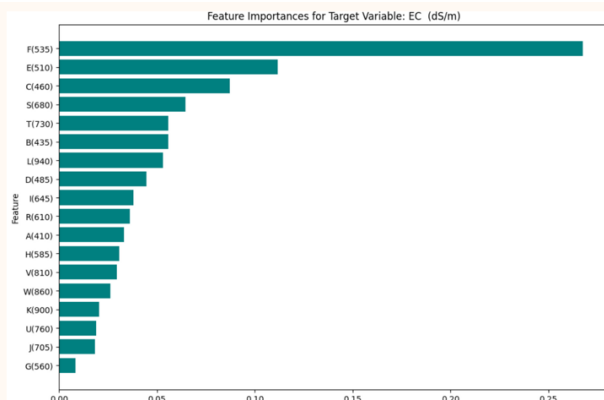
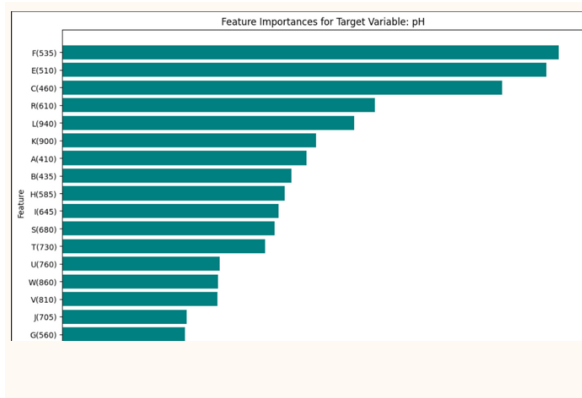
# Sort features by importance
feature_importance_df = feature_importance_df.sort_values(by='Importance', ascending=False)

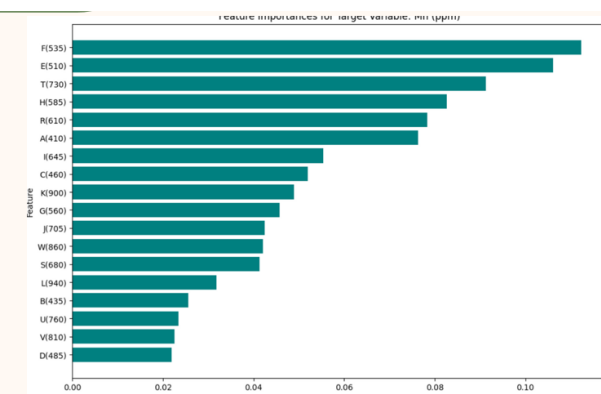
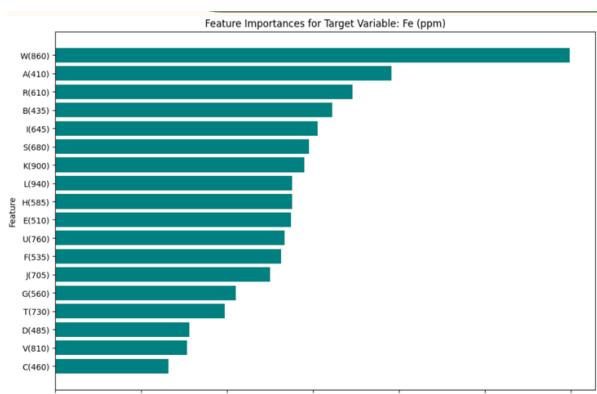
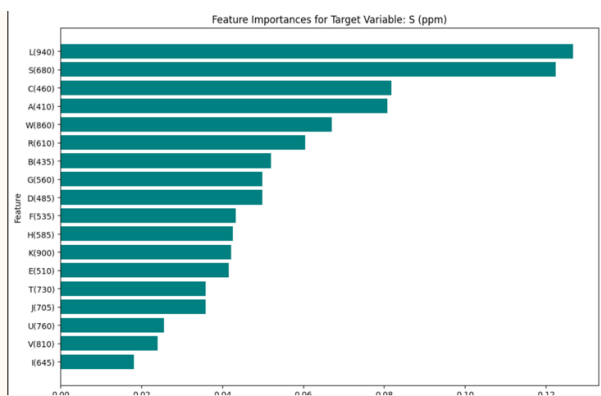
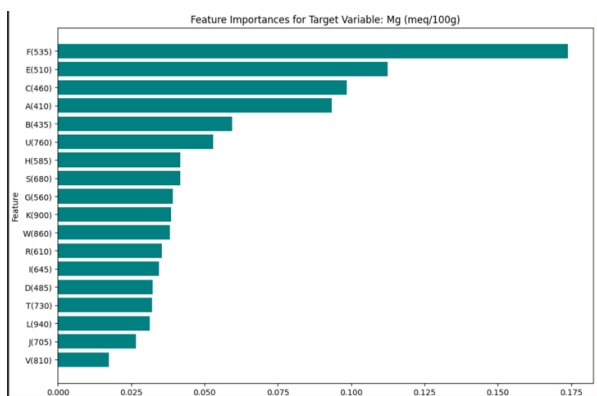
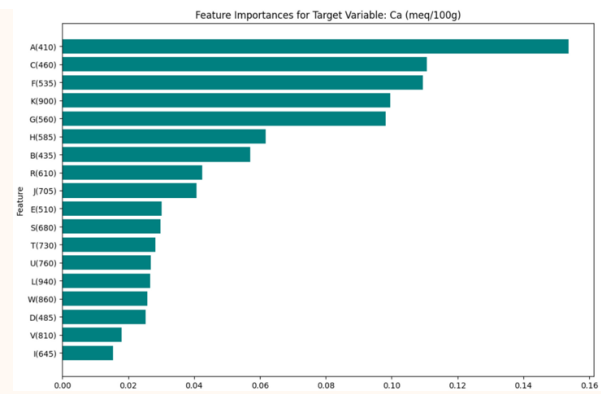
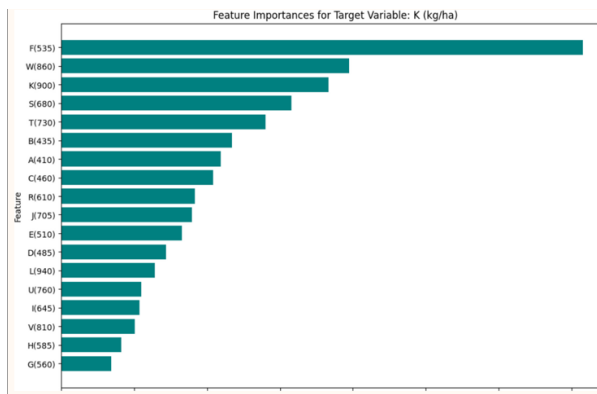
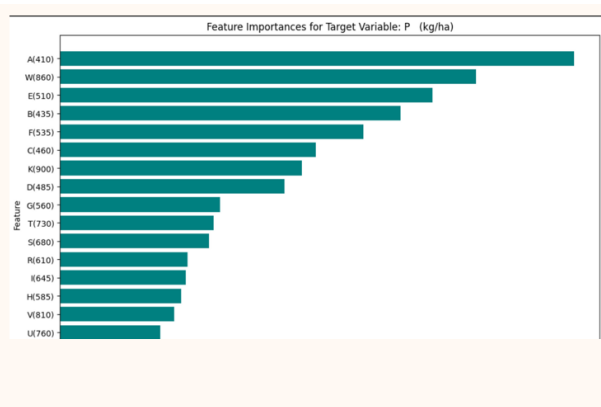
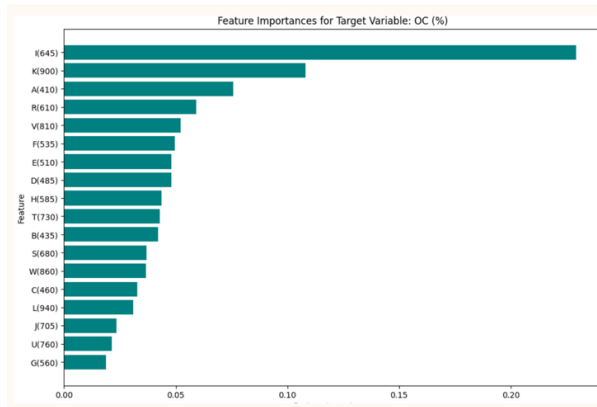
# Print feature importances
print("Feature importances:")
print(feature_importance_df)

# Select top N features (e.g., top 10 features)
top_n_features = feature_importance_df['Feature'].values[:10]

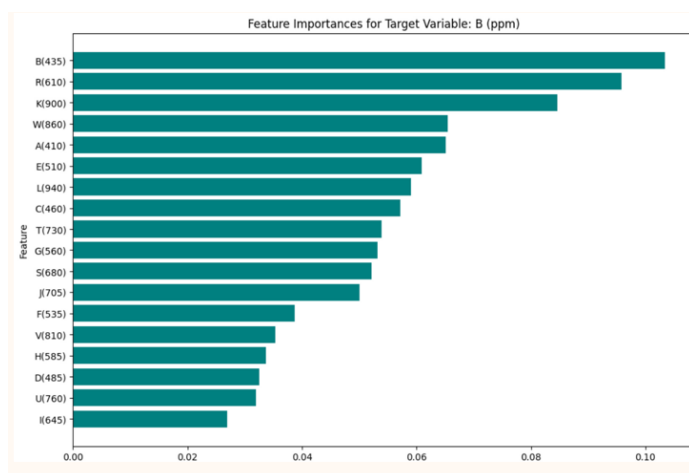
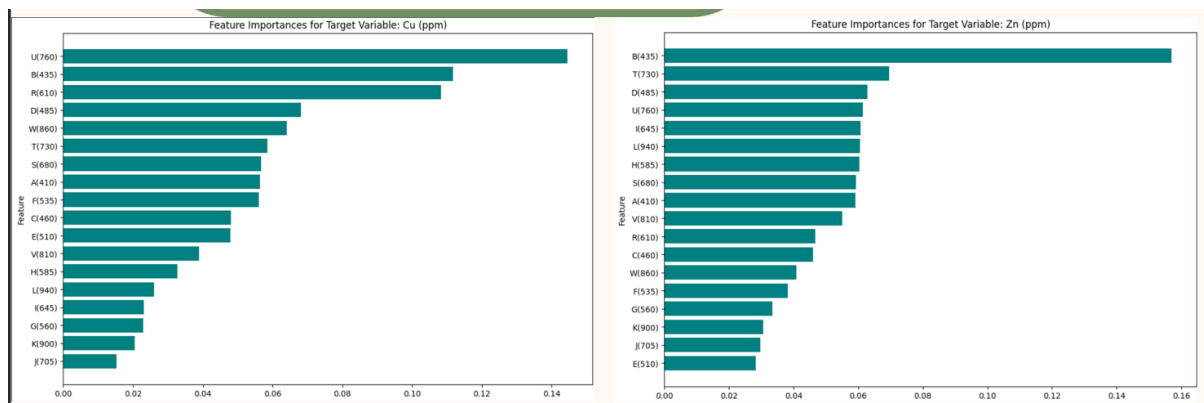
# Create a new dataset with only the top N features
X_train_selected = X_train[top_n_features]
X_test_selected = X_test[top_n_features]

# Hyperparameter tuning with GridSearchCV
param_grid = {
    'estimator__n_estimators': [50, 100, 200],
    'estimator__max_depth': [None, 10, 20, 30],
    'estimator__min_samples_split': [2, 5, 10],
    'estimator__min_samples_leaf': [1, 2, 4]
```









### Performance Metrics of Feature Importance:

```
Best parameters found: {'estimator_max_depth': 10, 'estimator_min_s
Random Forest Regressor Performance Metrics with Selected Features:
R² Score on Training Data: 0.9297
R² Score on Test Data: 0.6995
Mean Squared Error (MSE) on Test Data: 823.0928
Mean Absolute Error (MAE) on Test Data: 8.3535
Root Mean Squared Error (RMSE) on Test Data: 28.6896
```

Applied feature selection and further hyperparameter tuning to improve the Random Forest Regressor's performance.

### Key Observations:

- **Improvement in Test Performance:** After tuning the hyperparameters and selecting the most important features, your model shows improved performance on the test data, both in terms of  $R^2$  and error metrics. This indicates that feature selection helped improve generalization and reduce overfitting.

- **Hyperparameter Tuning:** By adjusting the depth of the trees and using a higher number of estimators, you achieved a better balance between model complexity and overfitting.
- **Feature Selection:** Focusing on the most important features, as indicated by the feature importance scores, likely reduced noise and improved the model's predictive ability.

### **Feature Selection Limitations:**

- **Limited Impact of Removed Features:**
  - Some features, though removed, may not have contributed significantly to model performance.
  - Feature importance showed that top features (e.g., F(535), A(410), B(435)) were more influential than others, but the overall dataset may still have unrecognized relationships.
- **Correlated Features:**
  - Removing highly correlated features might overlook important interactions between them.
  - Multicollinearity can distort the importance of some features.
- **Complex Data Relationships:**
  - The relationships between features and target variables may not be captured fully by the current feature selection method.
  - Non-linear or higher-order interactions may require more sophisticated approaches.

#### 4. Random Forest (RF) after increasing the dataset to 200

An ensemble learning method that builds multiple decision trees and merges them to improve accuracy and control overfitting.

## 2. Key Features:

- **Bagging Technique:** Randomly samples subsets of the training data to create diverse trees.
- **Feature Randomness:** At each split, a random subset of features is considered, enhancing model diversity.

### 3. Advantages:

- **Handles Non-Linearity:** Effective for datasets with complex relationships, such as spectral measurements for soil nutrients.
- **Robustness:** Less prone to overfitting compared to single decision trees.

#### 4. Disadvantages:

- **Computationally Intensive:** Requires more resources for training due to multiple trees.
- **Interpretability:** Harder to interpret compared to linear models.

```
Random Forest Regressor Performance Metrics:  
Mean Squared Error (MSE): 888.3761  
Mean Absolute Error (MAE): 8.7485  
Root Mean Squared Error (RMSE): 29.8056  
R2 Score: 0.6745  
R2 Score on Training Data: 0.9281
```

The model has improved after increasing the dataset

### Performance Metrics of RF with hyperparameter tuning after increasing the dataset:

```
Fitting 3 folds for each of 108 candidates, totalling 324 fits
Best parameters found: {'estimator__max_depth': None, 'estimator__min_samples': 10}
Random Forest Regressor Performance Metrics with Tuned Hyperparameters:
R2 Score on Training Data: 0.9305
R2 Score on Test Data: 0.6801
Mean Squared Error (MSE) on Test Data: 931.6294
Mean Absolute Error (MAE) on Test Data: 8.8005
Root Mean Squared Error (RMSE) on Test Data: 30.5226
```

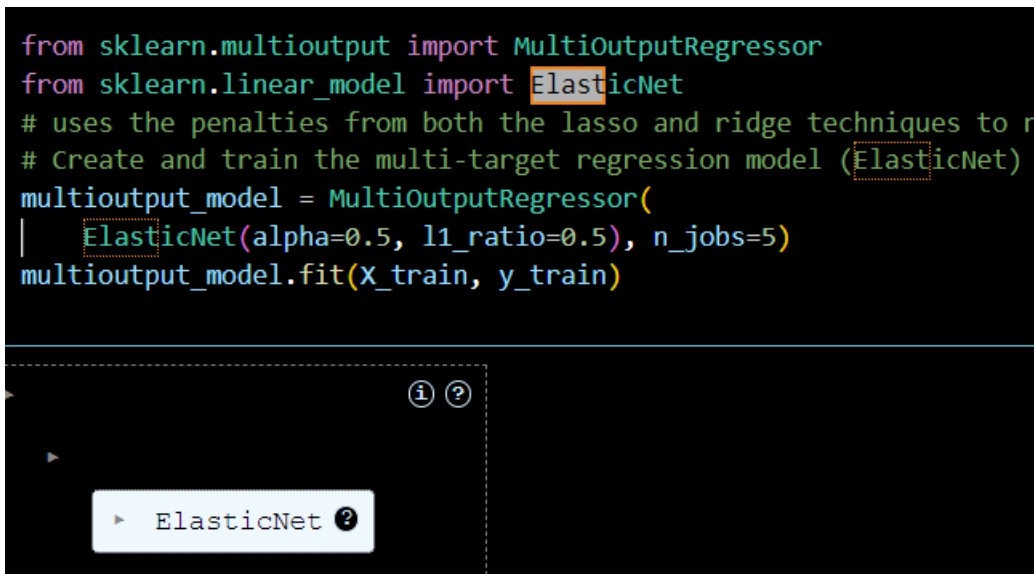
It looks like the Random Forest Regressor's performance has improved slightly on the training data after hyperparameter tuning, with the  $R^2$  score increasing from 0.9281 to

0.9305. However, on the test data, there is a slight decrease in performance, as the  $R^2$  score dropped from 0.6745 to 0.6801, and the error metrics (MSE, MAE, RMSE) increased slightly.

## 5. Elastic Net

A linear regression model that combines L1 (Lasso) and L2 (Ridge) regularization to enhance feature selection and reduce multicollinearity.

```
from sklearn.multioutput import MultiOutputRegressor
from sklearn.linear_model import ElasticNet
# uses the penalties from both the lasso and ridge techniques to r
# Create and train the multi-target regression model (ElasticNet)
multioutput_model = MultiOutputRegressor(
| ElasticNet(alpha=0.5, l1_ratio=0.5), n_jobs=5)
multioutput_model.fit(X_train, y_train)
```



Key Features:

- Regularization: Controls overfitting by penalizing large coefficients. The balance between L1 and L2 penalties is adjustable.

Advantages:

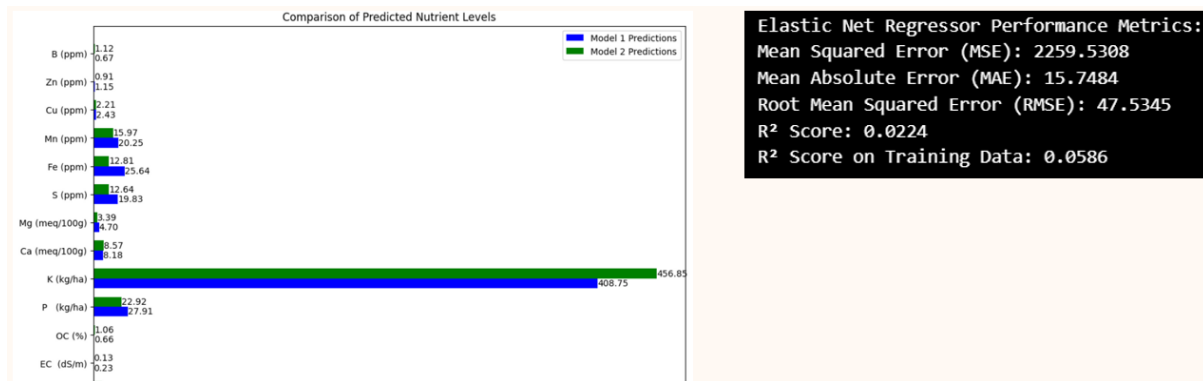
- Feature Selection: Effectively selects a subset of relevant features, which is valuable when dealing with many predictors (e.g., spectral measurements).
- Flexibility: Performs well in situations where predictors are correlated.

Disadvantages:

- Assumption of Linearity: Assumes a linear relationship between predictors and targets, which may not capture complex patterns in soil

nutrient data.

- Sensitive to Scaling: Requires features to be standardized for optimal performance.



The reason why Elastic Net and Random Forest (RF) are giving similar outputs, but Elastic Net's accuracy is much lower compared to Random Forest, likely stems from the fundamental differences in how these two models work and their ability to capture patterns in the data.

Model Complexity and Flexibility:

- Elastic Net:

Elastic Net is a linear model that combines both Lasso (L1) and Ridge (L2) regularization. It works by finding a linear relationship between the features and the target, which means it is limited in its ability to model complex, non-linear relationships. Elastic Net may not be able to capture interactions or non-linearities between features, especially if the data is inherently complex or contains non-linear patterns.

- Random Forest (RF):

Random Forest is an ensemble model that builds multiple decision trees, where each tree is fit to a random subset of the data and features. It can capture non-linear relationships and interactions between features because decision trees do not rely on linear assumptions. RF is much more flexible and capable of modeling complex patterns, making it a better choice when the relationships in

the data are non-linear or when there are interactions between features.

Elastic Net is a linear model, limiting its ability to capture complex, non-linear relationships, which may explain its lower accuracy. Random Forest is more flexible and can model non-linear relationships, making it a better fit for data with complex interactions between features. Elastic Net's performance can be heavily impacted by regularization, underfitting, and feature scaling issues. Random Forest tends to be more robust and can handle the complexity of real-world datasets more effectively, which is why its performance metrics are significantly better in this case. If Elastic Net's accuracy is low while RF performs well, it suggests that the relationships in the data are non-linear and complex, making Random Forest a more suitable model choice in this scenario.

## **6. Lasso Regression**

Lasso Regression is a type of linear regression that uses L1 regularization to enforce sparsity in the model. It penalizes the absolute size of the coefficients, effectively shrinking some of them to zero, which can help with feature selection and reduce overfitting.

```

from sklearn.linear_model import Lasso

# Train Lasso Regression model
lasso = Lasso()

# Fit the model
lasso.fit(X_train, y_train)

# Predict for the test set
y_pred_lasso = lasso.predict(X_test)
y_pred_train_lasso = lasso.predict(X_train)

# Calculate performance metrics
mse_lasso = mean_squared_error(y_test, y_pred_lasso)
mae_lasso = mean_absolute_error(y_test, y_pred_lasso)
rmse_lasso = np.sqrt(mse_lasso)
r2_lasso = r2_score(y_test, y_pred_lasso)
r2_train_lasso = r2_score(y_train, y_pred_train_lasso)

print("Lasso Regression Performance Metrics:")
print(f"Mean Squared Error (MSE): {mse_lasso:.4f}")
print(f"Mean Absolute Error (MAE): {mae_lasso:.4f}")
print(f"Root Mean Squared Error (RMSE): {rmse_lasso:.4f}")
print(f"R2 Score: {r2_lasso:.4f}")
print(f"R2 Score on Training Data: {r2_train_lasso:.4f}")

```

```

Lasso Regression Performance Metrics:
Mean Squared Error (MSE): 2203.1764
Mean Absolute Error (MAE): 15.8557
Root Mean Squared Error (RMSE): 46.9380
R2 Score: 0.0020
R2 Score on Training Data: 0.0366

```

Since R<sup>2</sup> score is very low, let us not consider this model to be efficient.

## 7. Decision Tree

### 1. What is a Decision Tree?

- A Decision Tree is a supervised learning algorithm used for both classification and regression tasks.
- It models decisions based on a tree-like structure, where each internal node represents a feature, each branch represents a decision rule, and each leaf node

represents an outcome.

```
#Decision Tree
from sklearn.tree import DecisionTreeRegressor
from sklearn.multioutput import MultiOutputRegressor
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
import numpy as np

# Train Decision Tree Regressor model for multiple target variables
dtr = DecisionTreeRegressor(random_state=42)

# Wrap Decision Tree Regressor in MultiOutputRegressor to handle multiple outputs
multioutput_dtr = MultiOutputRegressor(dtr)

# Fit the Decision Tree Regressor model
multioutput_dtr.fit(X_train, y_train)

# Predict for the test set
y_pred_dtr = multioutput_dtr.predict(X_test)
y_pred_train_dtr = multioutput_dtr.predict(X_train)

# Calculate performance metrics
mse_dtr = mean_squared_error(y_test, y_pred_dtr)
mae_dtr = mean_absolute_error(y_test, y_pred_dtr)
rmse_dtr = np.sqrt(mse_dtr)
r2_dtr = r2_score(y_test, y_pred_dtr)
r2_train_dtr = r2_score(y_train, y_pred_train_dtr)

print("Decision Tree Regressor Performance Metrics:")
print(f"Mean Squared Error (MSE): {mse_dtr:.4f}")
print(f"Mean Absolute Error (MAE): {mae_dtr:.4f}")
print(f"Root Mean Squared Error (RMSE): {rmse_dtr:.4f}")
print(f"R2 Score: {r2_dtr:.4f}")
print(f"R2 Score on Training Data: {r2_train_dtr:.4f}")
```

## 2. How Decision Trees Work:

- Recursive Splitting:
  - The algorithm splits the dataset into subsets based on the value of input features.
  - Splits are made to minimize a chosen criterion (e.g., Mean Squared Error for regression).
- Tree Structure:
  - The tree is constructed recursively until a stopping condition is met (e.g.,



maximum depth, minimum samples per leaf).

### 3. Why Use Decision Trees for This Dataset?

- **Interpretability:** Decision Trees are easy to visualize and interpret, making them accessible for understanding the relationships between spectral measurements and soil nutrient levels.
- **Handling Non-Linearity:** Capable of modeling complex, non-linear relationships without requiring feature transformation.
- **No Feature Scaling Required:** Decision Trees do not require normalization or standardization of features, simplifying data preprocessing.

```
Decision Tree Regressor Performance Metrics:  
Mean Squared Error (MSE): 1114.8290  
Mean Absolute Error (MAE): 4.6089  
Root Mean Squared Error (RMSE): 33.3891  
R2 Score: 0.7596  
R2 Score on Training Data: 0.9823
```

## 8. KNN

### 1. What is KNN?

- K-Nearest Neighbors (KNN) is a simple, non-parametric algorithm used for both classification and regression tasks.
- For regression, KNN predicts the value of a target variable by averaging the values of the k-nearest data points in the feature space.

```

from sklearn.neighbors import KNeighborsRegressor
from sklearn.multioutput import MultiOutputRegressor
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
import numpy as np

# Train K-Nearest Neighbors Regressor model for multiple target variables
knn = KNeighborsRegressor()

# Wrap knn in MultiOutputRegressor to handle multiple outputs
multioutput_knn = MultiOutputRegressor(knn)

# Fit the knn model
multioutput_knn.fit(X_train, y_train)

# Predict for the test set
y_pred_knn = multioutput_knn.predict(X_test)
y_pred_train_knn = multioutput_knn.predict(X_train)

# Calculate performance metrics
mse_knn = mean_squared_error(y_test, y_pred_knn)
mae_knn = mean_absolute_error(y_test, y_pred_knn)
rmse_knn = np.sqrt(mse_knn)
r2_knn = r2_score(y_test, y_pred_knn)
r2_train_knn = r2_score(y_train, y_pred_train_knn)

```

## 2. How KNN Works:

- Distance Calculation: The algorithm calculates the distance between the test data point and all the training data points using a distance metric (e.g., Euclidean distance).
- Identify Neighbors: Select the k closest data points (neighbors) to the test point.
- Prediction: For regression, KNN takes the average of the target values of the k-nearest neighbors to make predictions.

## 3. Why Use KNN for This Dataset?

- No Assumptions: KNN makes no assumptions about the underlying distribution of soil nutrient data.
- Multi-target Capability: KNN can predict multiple soil nutrients (targets) simultaneously by averaging the values of neighbors for each nutrient.
- Data-driven: KNN performs well with datasets that have complex relationships

between features and targets, like soil properties measured via spectral data.

#### 4. Application to Soil Dataset:

- Dataset: 18 features (A, B, C, ..., T) based on spectral measurements, and 13 target variables (pH, EC, etc.).
- Process:
  - Normalize/Scale the feature values for better distance calculation.
  - Use KNN regression to predict soil nutrient values based on the closest training samples in feature space.
- Performance Tuning:
  - Tune k (number of neighbors) and distance metric (Euclidean, Manhattan) to improve accuracy.

#### 5. KNN Summary:

Advantages: Simple and intuitive. Handles multi-target regression well.

Disadvantages: Can be computationally expensive with large datasets. Sensitive to feature scaling and choice of k.

## 9. Gradient Boosting

### 1. What is Gradient Boosting?

- Gradient Boosting is an ensemble machine learning technique used for both classification and regression tasks.
- It builds models sequentially, where each new model attempts to correct the errors made by the previous models.

```

from sklearn.ensemble import GradientBoostingRegressor

# Train Gradient Boosting Regressor model for multiple target variables
gbr = GradientBoostingRegressor(random_state=42)

# Wrap GBR in MultiOutputRegressor to handle multiple outputs
multioutput_gbr = MultiOutputRegressor(gbr)

# Fit the GBR model
multioutput_gbr.fit(X_train, y_train)

# Predict for the test set
y_pred_gbr = multioutput_gbr.predict(X_test)
y_pred_train_gbr = multioutput_gbr.predict(X_train)

# Calculate performance metrics
mse_gbr = mean_squared_error(y_test, y_pred_gbr)
mae_gbr = mean_absolute_error(y_test, y_pred_gbr)
rmse_gbr = np.sqrt(mse_gbr)
r2_gbr = r2_score(y_test, y_pred_gbr)
r2_train_gbr = r2_score(y_train, y_pred_train_gbr)

```

## 2. How Gradient Boosting Works:

- Initial Model: Start with a simple model (e.g., mean value for regression).
- Iterative Learning:
  - In each iteration, a new model (usually a decision tree) is trained to minimize the errors or residuals of the previous model.
- Gradient Descent: The algorithm uses gradient descent to optimize predictions by minimizing a loss function (e.g., Mean Squared Error for regression).
- Final Prediction: The final prediction is the sum of the predictions from all models, weighted by their learning rate.

## 3. Why Use Gradient Boosting for This Dataset?

- Handles Complex Relationships: Gradient Boosting excels at modeling non-linear relationships, which are often present in soil data.

- Multi-target Support: You can use Gradient Boosting for multi-output regression to predict multiple soil nutrients (pH, EC, etc.) simultaneously.
- Feature Importance: Gradient Boosting can rank the importance of the spectral measurements (A, B, C, ..., T) in predicting soil nutrients.

#### 4. Gradient Boosting Summary:

- Advantages:
  - High accuracy: Especially for datasets with complex feature-target relationships.
  - Feature Importance: Provides insights into which spectral measurements are most important for predicting each nutrient.
- Disadvantages:
  - Training Time: Gradient Boosting can be slow due to its sequential nature.
  - Overfitting: Requires careful tuning of hyperparameters (learning rate, number of trees) to prevent overfitting.

## 10. AdaBoost

### 1. What is AdaBoost?

- Adaptive Boosting (AdaBoost) is an ensemble learning technique that combines multiple weak learners (usually decision trees) to create a strong predictive model.
- It focuses on improving the accuracy of weak classifiers by assigning higher weights to misclassified instances in subsequent models.

```

import numpy as np
from sklearn.ensemble import AdaBoostRegressor
from sklearn.multioutput import MultiOutputRegressor
from sklearn.tree import DecisionTreeRegressor
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score

base_model = DecisionTreeRegressor(max_depth=3)

# Initialize the AdaBoost model without hyperparameter tuning
adaboost_model = AdaBoostRegressor(estimator=base_model, random_state=42)

# Wrap AdaBoost in MultiOutputRegressor
multioutput_adaboost = MultiOutputRegressor(adaboost_model)

# Fit the model
multioutput_adaboost.fit(X_train, y_train)

# Make predictions using the model
y_pred_adaboost = multioutput_adaboost.predict(X_test)
y_pred_train_adaboost = multioutput_adaboost.predict(X_train)

# Calculate performance metrics for test data
mse_adaboost = mean_squared_error(y_test, y_pred_adaboost)
mae_adaboost = mean_absolute_error(y_test, y_pred_adaboost)
rmse_adaboost = np.sqrt(mse_adaboost)
r2_adaboost = r2_score(y_test, y_pred_adaboost)

# Calculate performance metrics for training data
r2_train_adaboost = r2_score(y_train, y_pred_train_adaboost)

```

## 2. How AdaBoost Works:

- Initial Model: Start with a simple weak learner (e.g., a decision stump).
- Weight Adjustment:
  - Each sample in the training set is assigned a weight. Initially, all weights are equal.
  - After each iteration, misclassified samples receive higher weights, while correctly classified samples have their weights decreased.
- Sequential Learning:

- New models are trained iteratively, focusing more on the previously misclassified instances.
- Final Prediction: The final prediction is a weighted sum of the predictions from all models, where the weight corresponds to each model's accuracy.

### 3. Why Use AdaBoost for This Dataset?

- Handles Complexity: AdaBoost can effectively capture complex relationships in soil nutrient data, improving prediction accuracy.
- Robustness: The algorithm is robust to overfitting when using simple base learners, making it suitable for the diverse features in your dataset.
- Feature Importance: AdaBoost provides insights into the importance of various spectral measurements (A, B, C, ..., T) in predicting soil nutrients.

### 4. AdaBoost Summary:

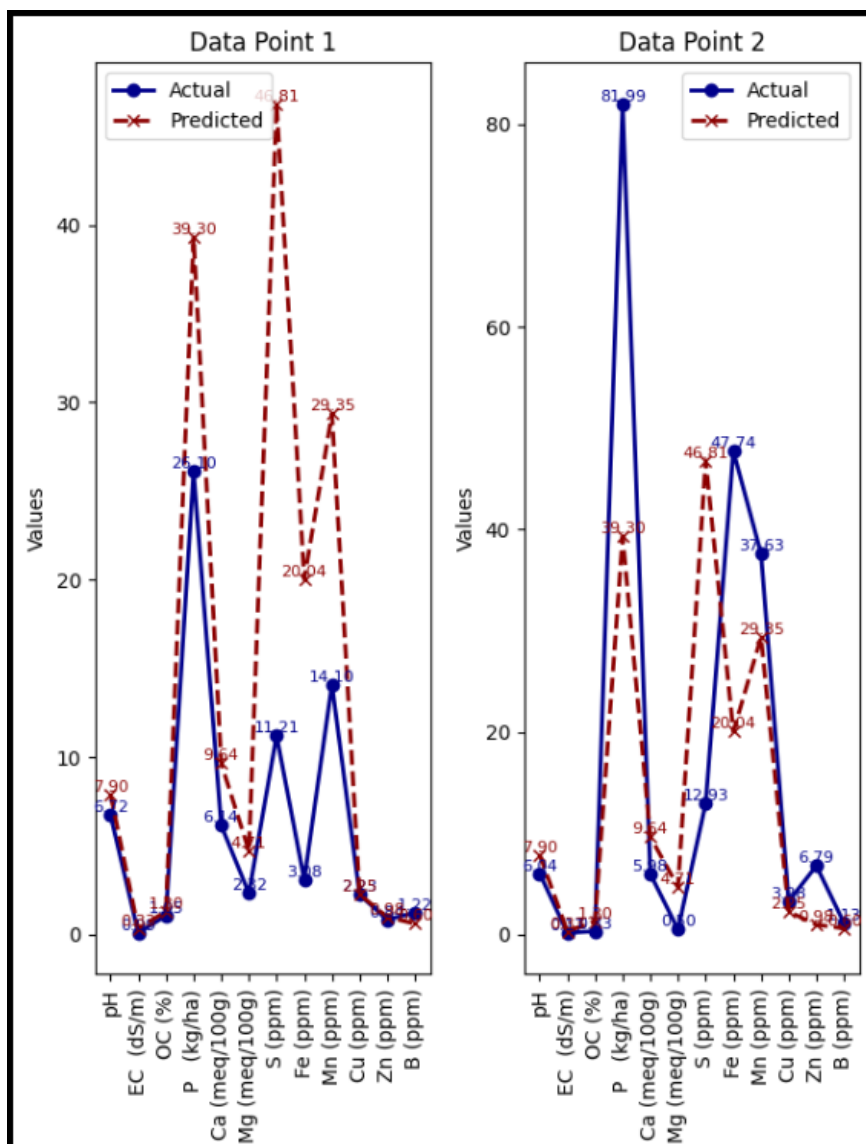
- Advantages:
  - Increased Accuracy: Boosting weak learners improves overall model performance.
  - Ease of Implementation: Simple to implement with a variety of base classifiers.
- Disadvantages:
  - Sensitive to Noisy Data: Can be affected by outliers and noisy data since it focuses on hard-to-classify instances.
  - Computational Cost: May require more computational resources as it iteratively adjusts weights.

## CHAPTER 6: RESULTS AND DISCUSSION

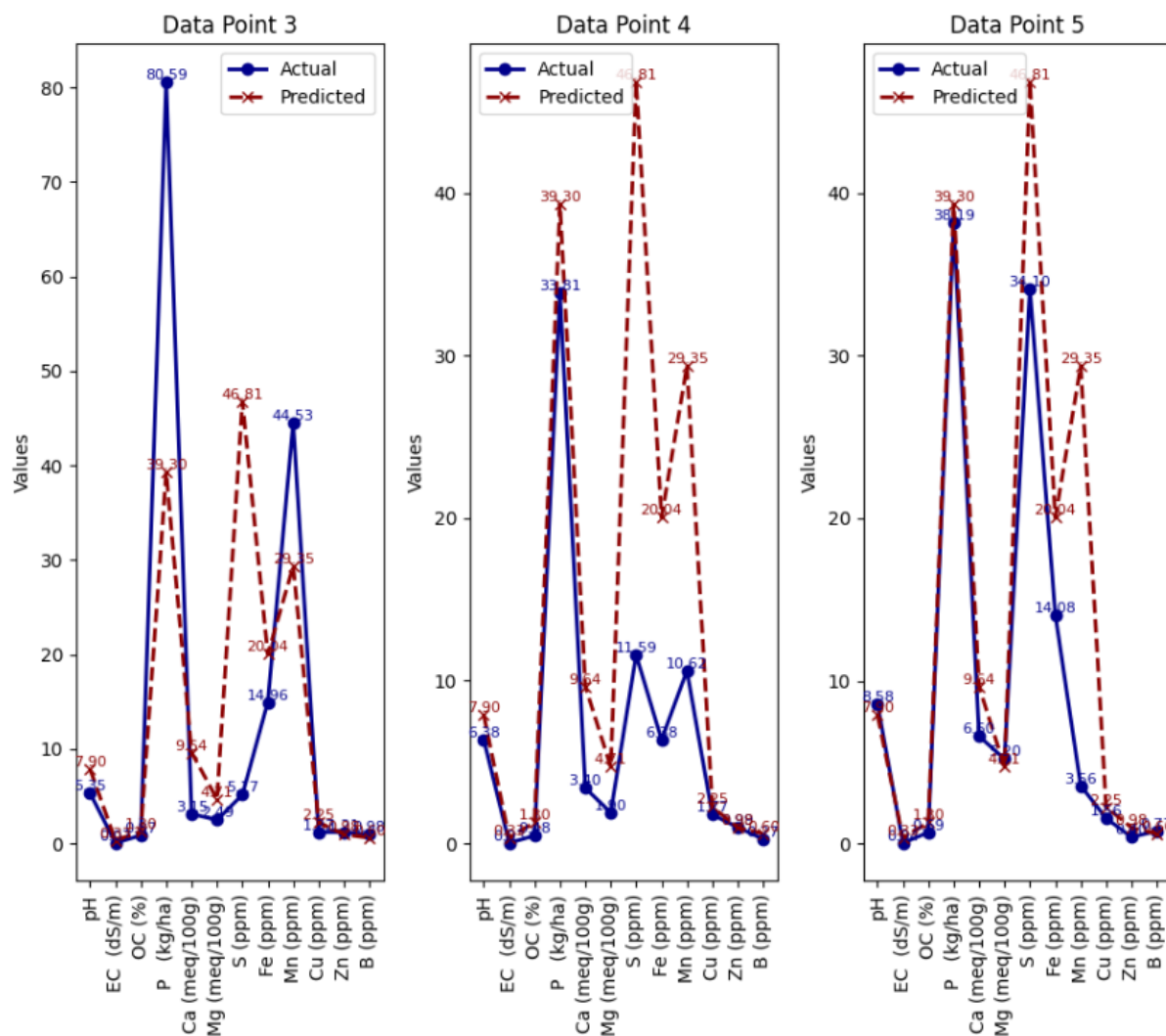
Regressor	MSE	MAE	RMSE	R2 Train	R2 Test
Decision Tree	913.833517	3.835030	30.229679	0.982259	0.699833
KNN	931.504173	4.159473	30.520553	0.982259	0.795390
Gradient Boosting	655.665083	5.684622	25.605958	0.968538	0.800789
AdaBoost	1055.254133	10.958940	32.484675	0.738951	0.533276

### Graphical Representation of the Actual vs Predicted Values:

Random Forest Regressor:







Actual values for 'K (kg/ha)':

0 444.00

1 372.00

2 132.00

3 221.76

4 234.08

Name: K (kg/ha), dtype: float64

Predicted values for 'K (kg/ha)':

0 294.0627

1 294.0627

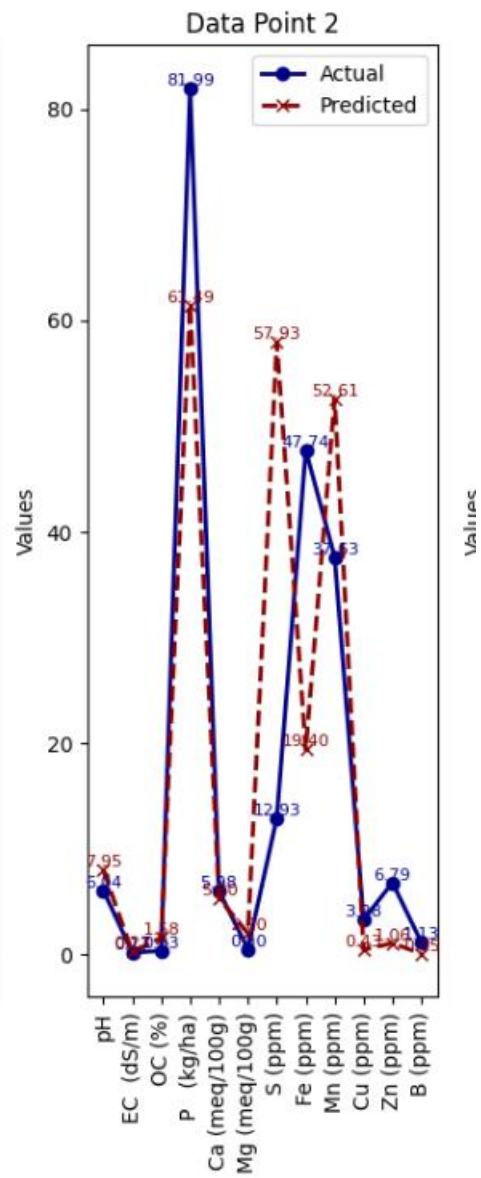
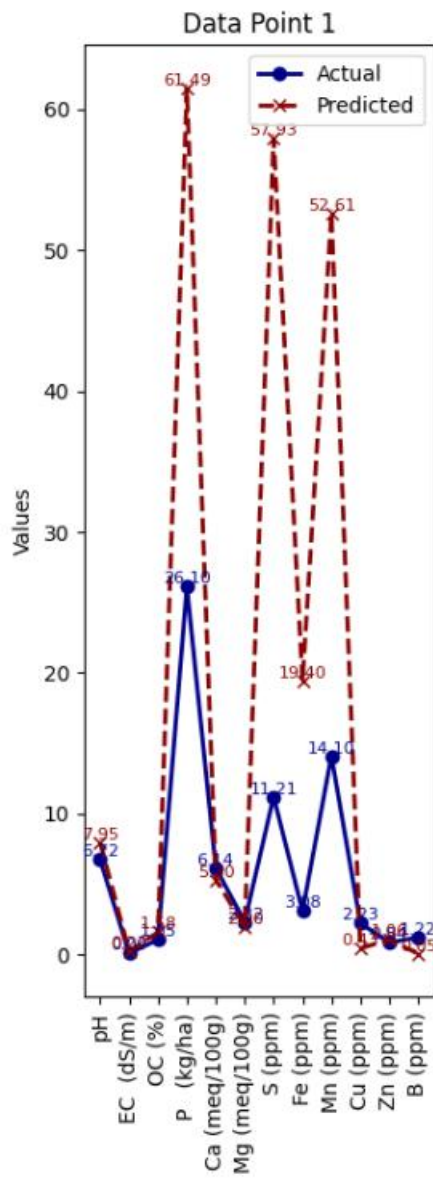
2 294.0627

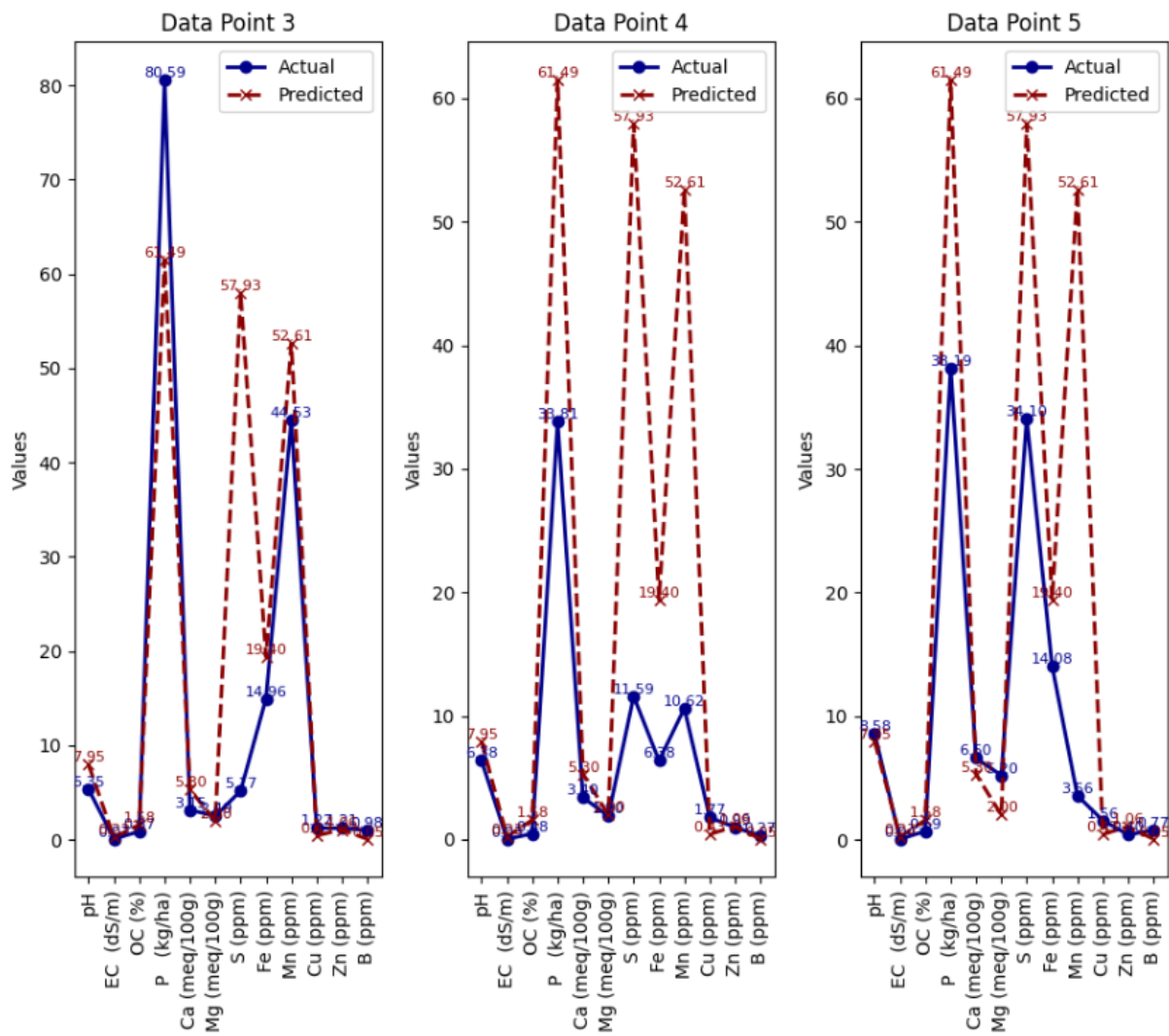
3 294.0627

4 294.0627

Name: K (kg/ha), dtype: float64

Decision Tree Regressor





Actual values for 'K (kg/ha)':

```
0    444.00
1    372.00
2    132.00
3    221.76
4    234.08
```

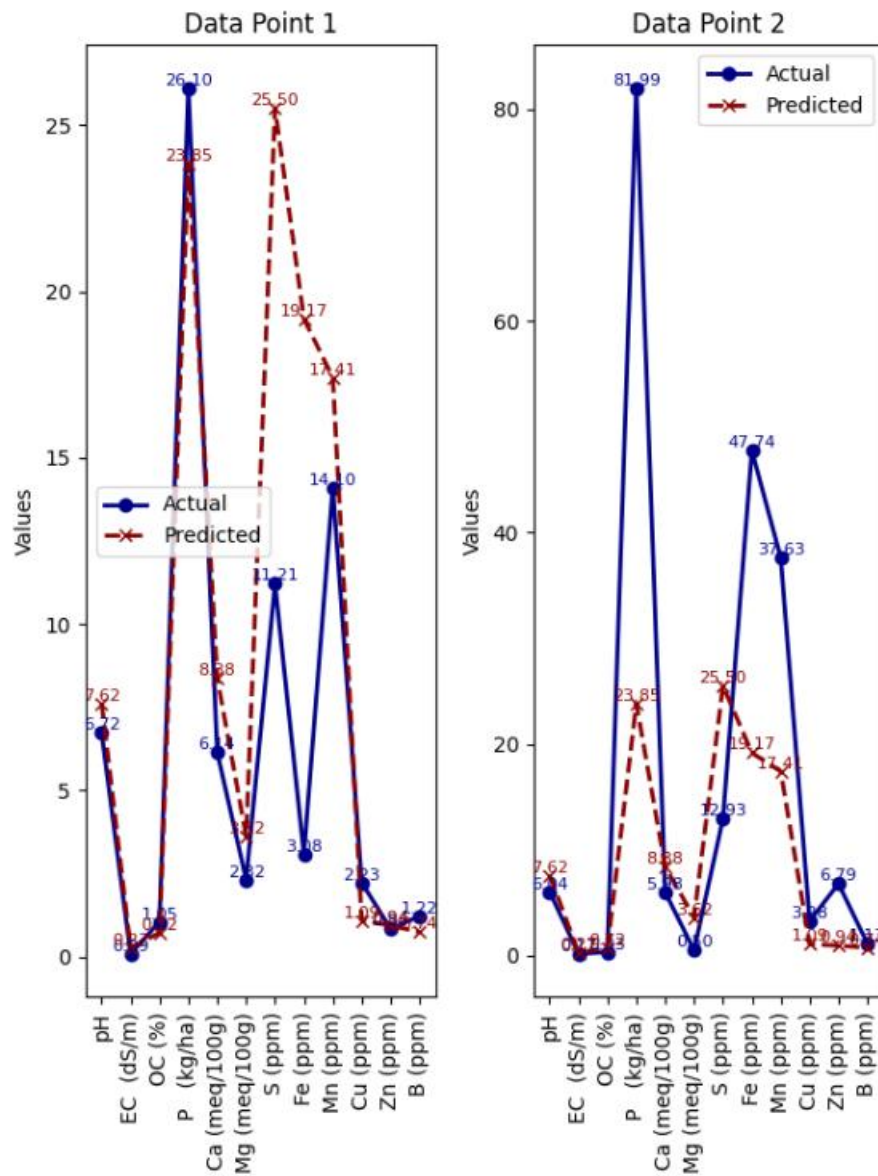
Name: K (kg/ha), dtype: float64

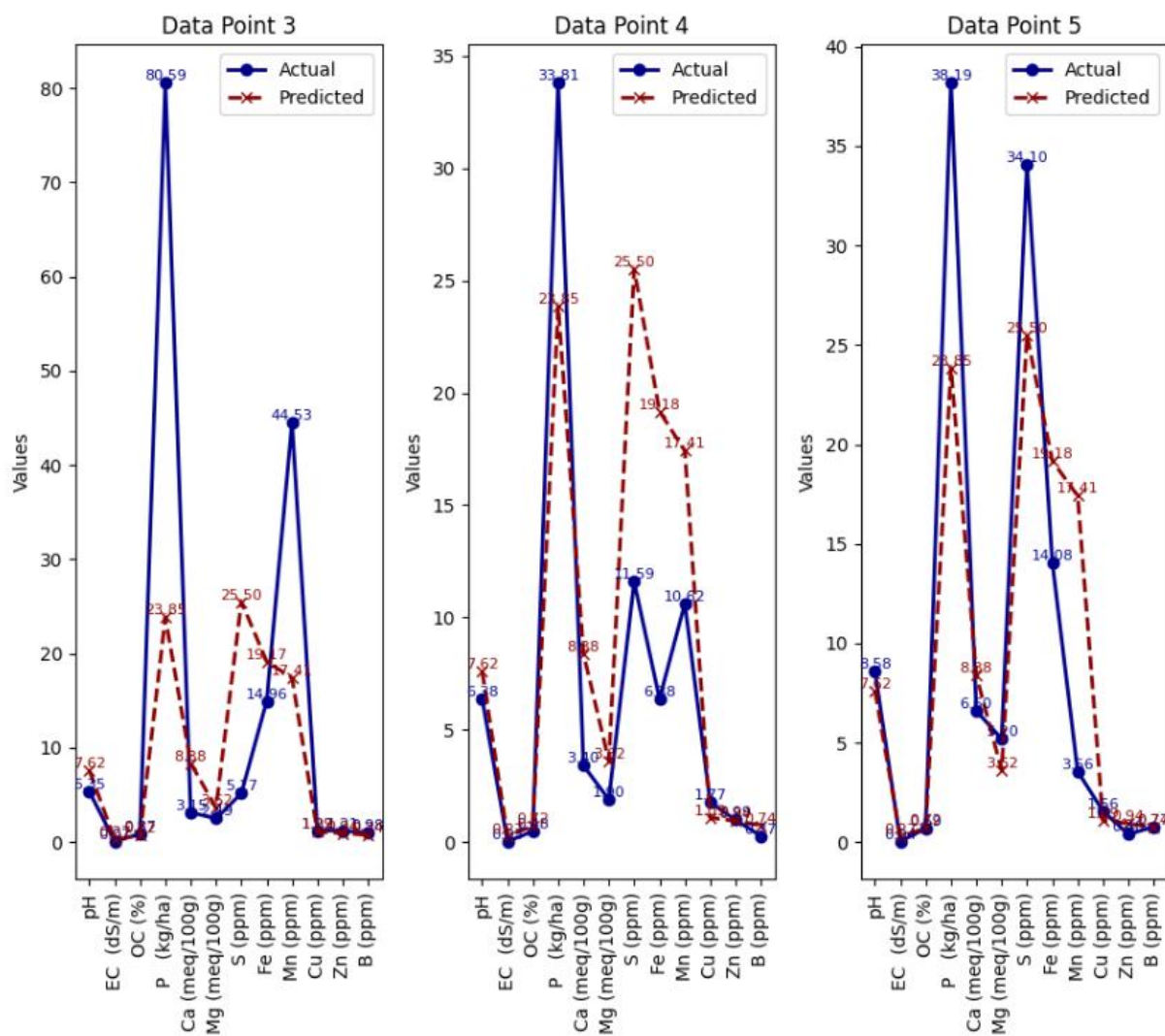
Predicted values for 'K (kg/ha)':

```
0    438.12
1    438.12
2    438.12
3    438.12
4    438.12
```

Name: K (kg/ha), dtype: float64

## K-Nearest Neighbours





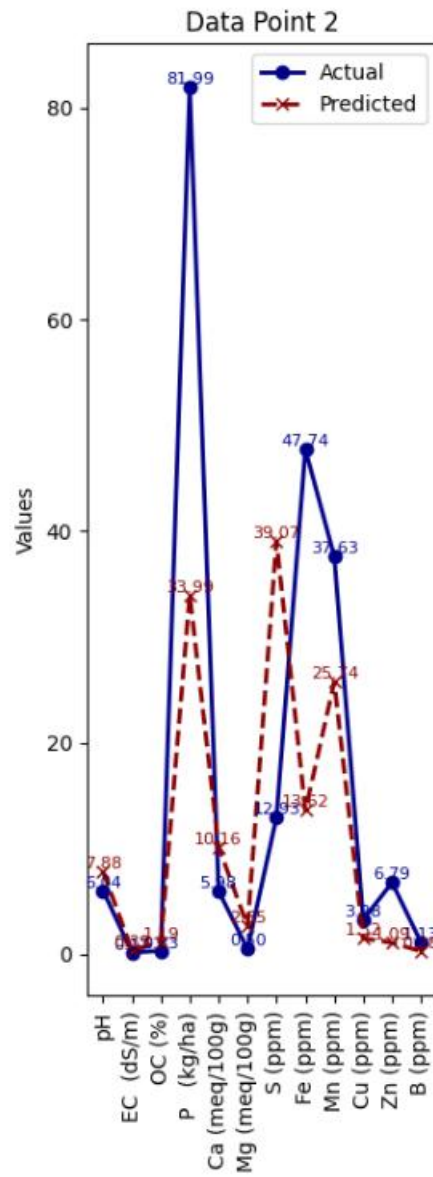
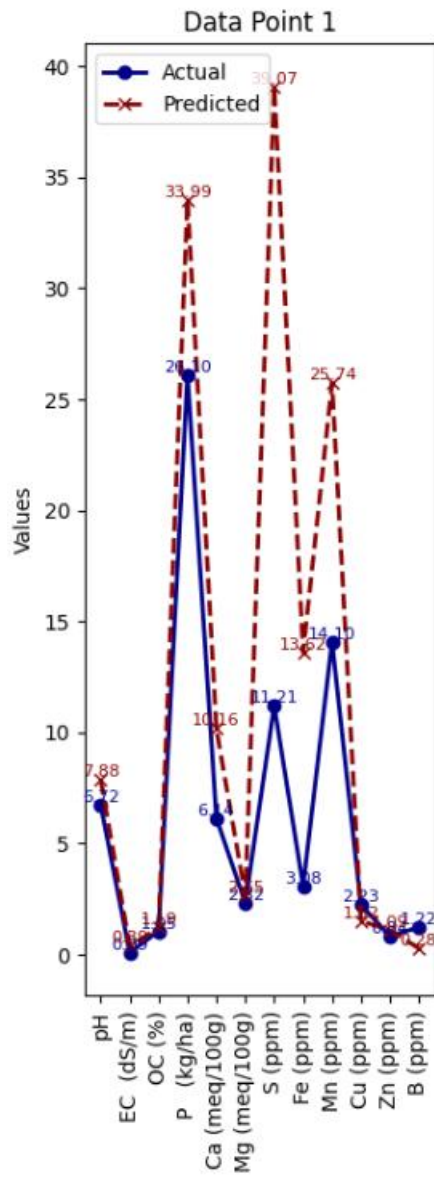
```

Actual values for 'K (kg/ha)':
0    444.00
1    372.00
2    132.00
3    221.76
4    234.08
Name: K (kg/ha), dtype: float64

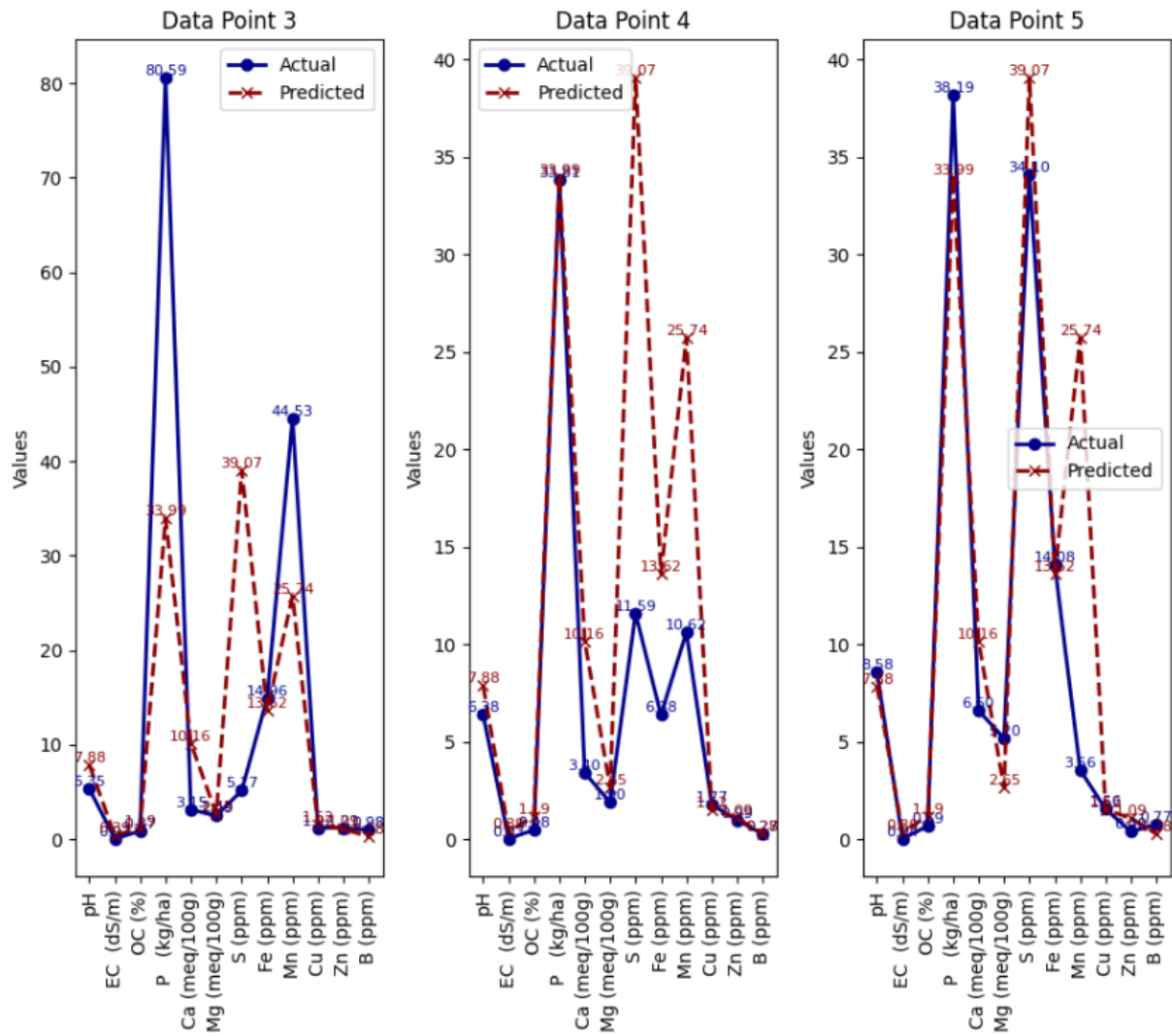
Predicted values for 'K (kg/ha)':
0    380.362655
1    380.364277
2    380.359866
3    380.368646
4    380.385198
Name: K (kg/ha), dtype: float64

```

Gradient Boosting Regressor:







Actual values for 'K (kg/ha)':

```
0    444.00
1    372.00
2    132.00
3    221.76
4    234.08
```

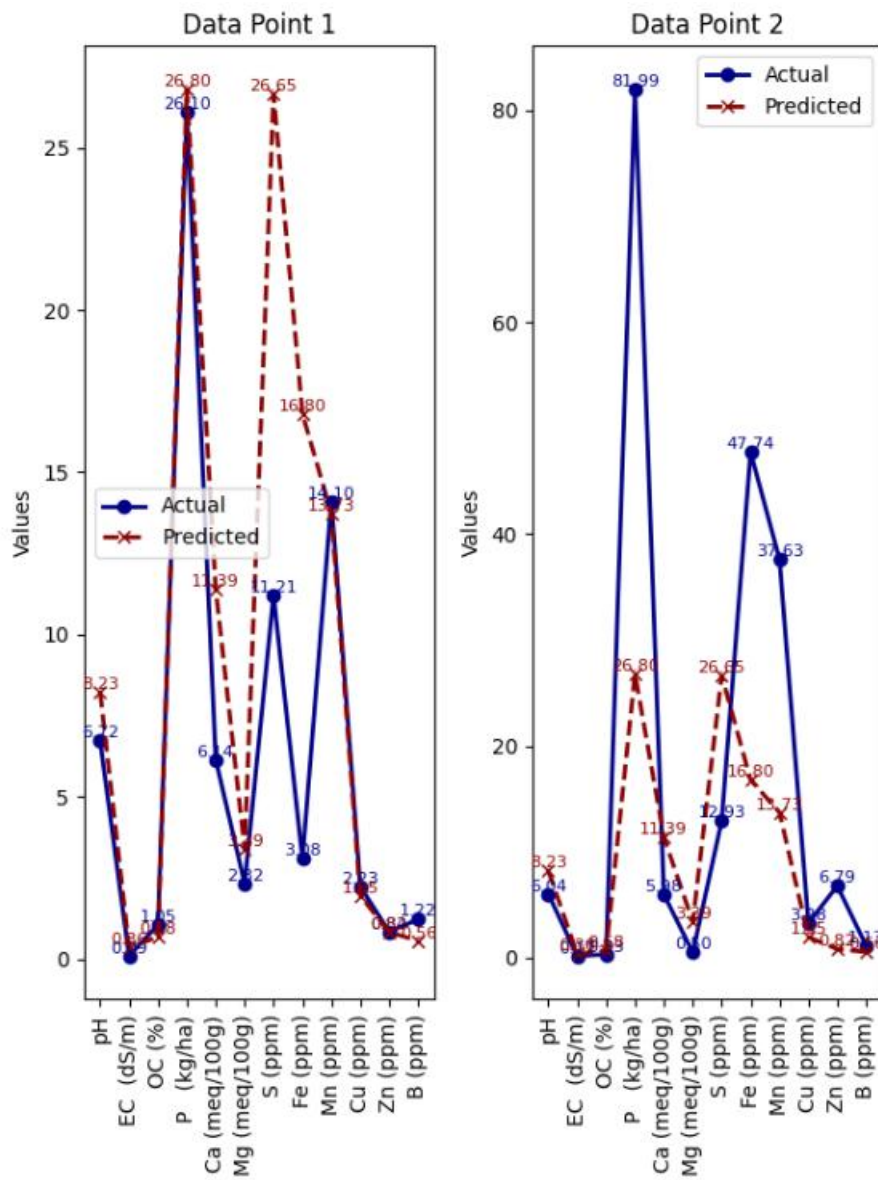
Name: K (kg/ha), dtype: float64

Predicted values for 'K (kg/ha)':

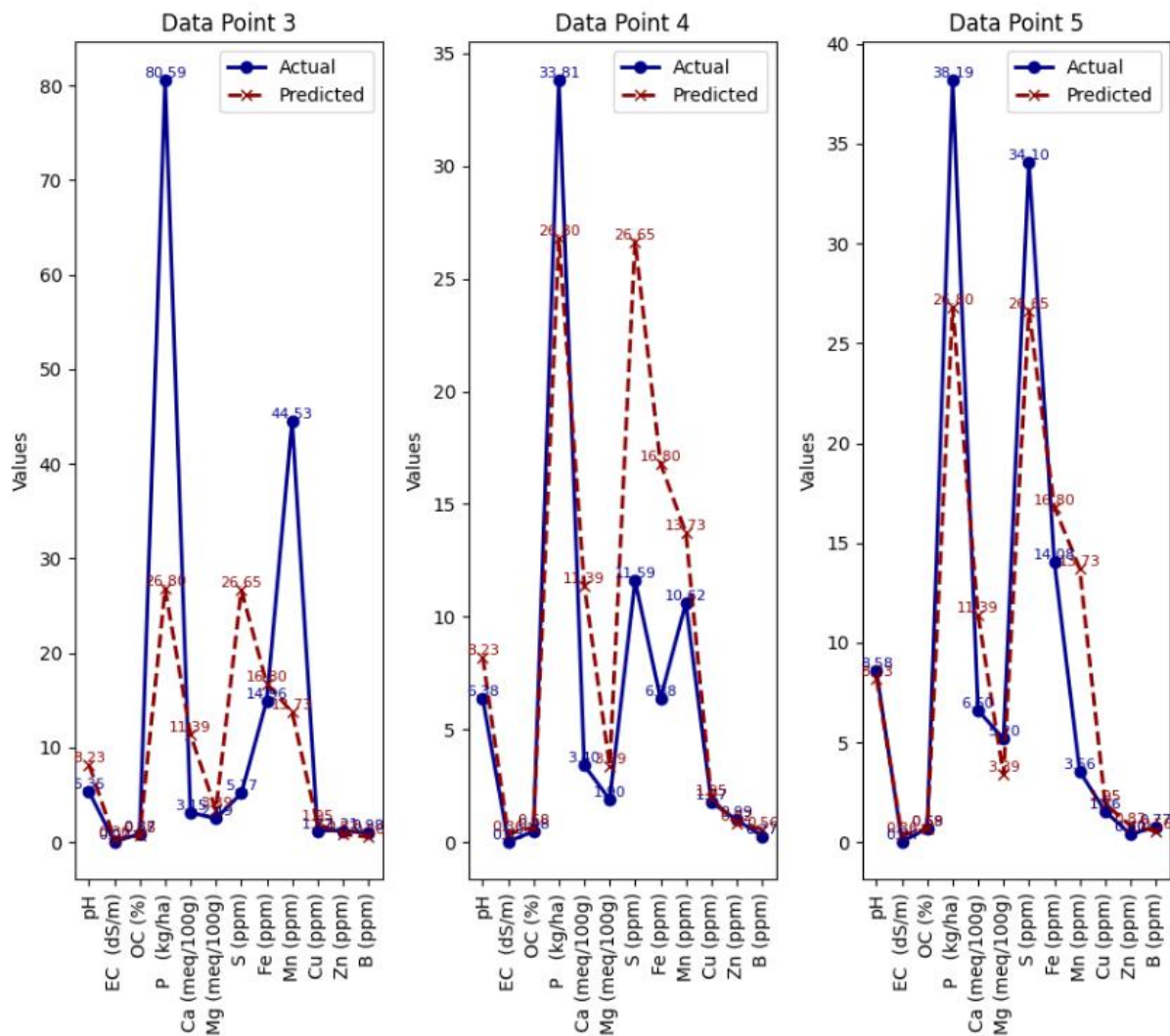
```
0    341.279203
1    341.279203
2    341.279203
3    341.279203
4    341.279203
```

Name: K (kg/ha), dtype: float64

AdaBoost :







Actual values for 'K (kg/ha)':

```
0    444.00
1    372.00
2    132.00
3    221.76
4    234.08
```

Name: K (kg/ha), dtype: float64

Predicted values for 'K (kg/ha)':

```
0    336.809904
1    336.809904
2    336.809904
3    336.809904
4    336.809904
```

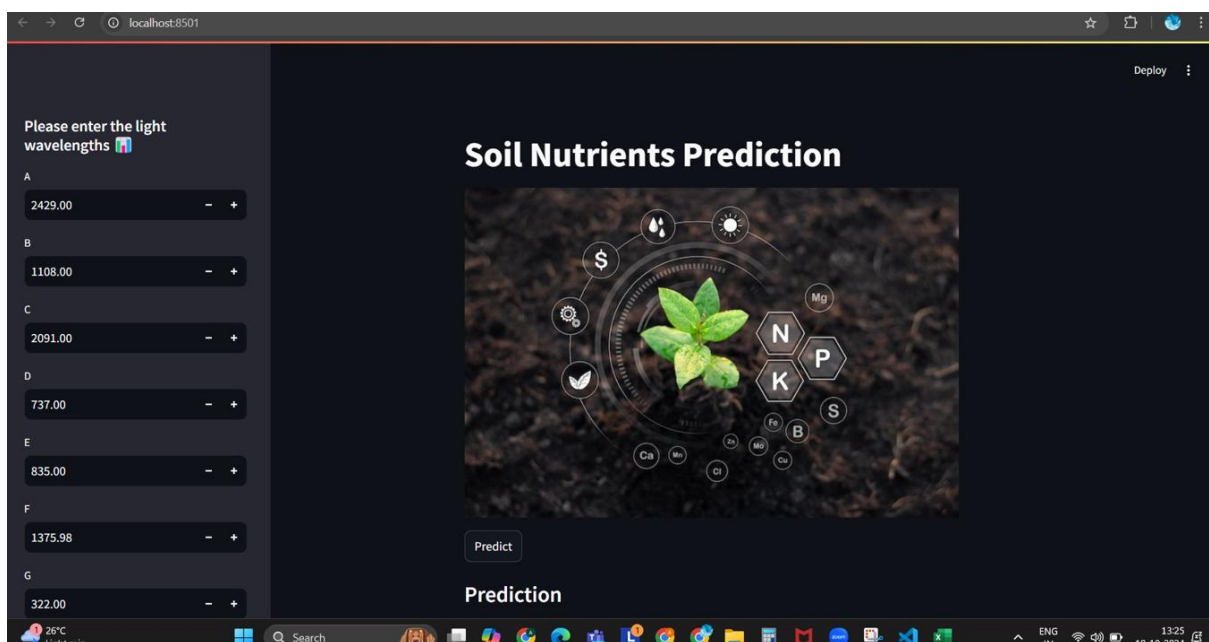
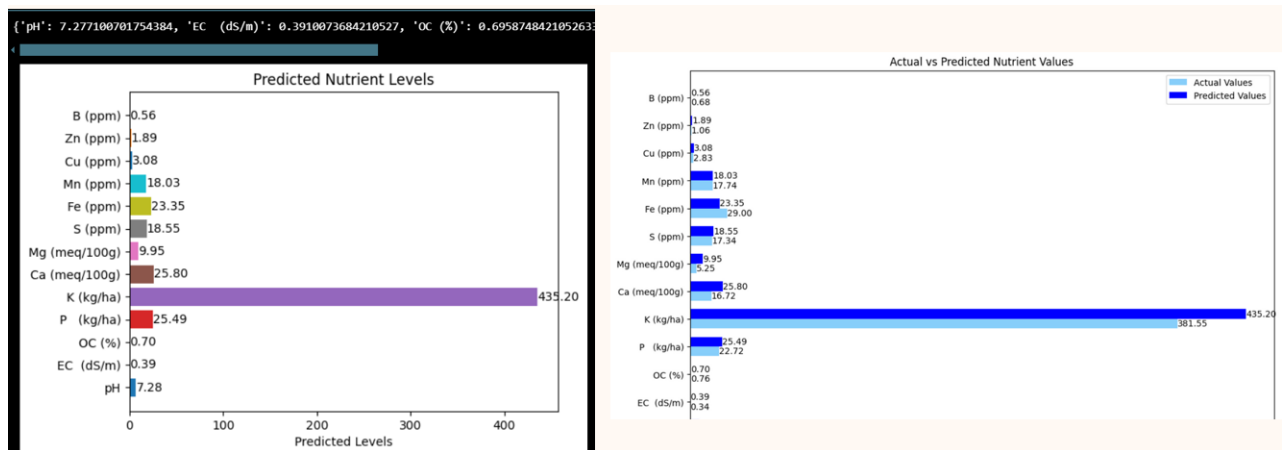
Name: K (kg/ha), dtype: float64

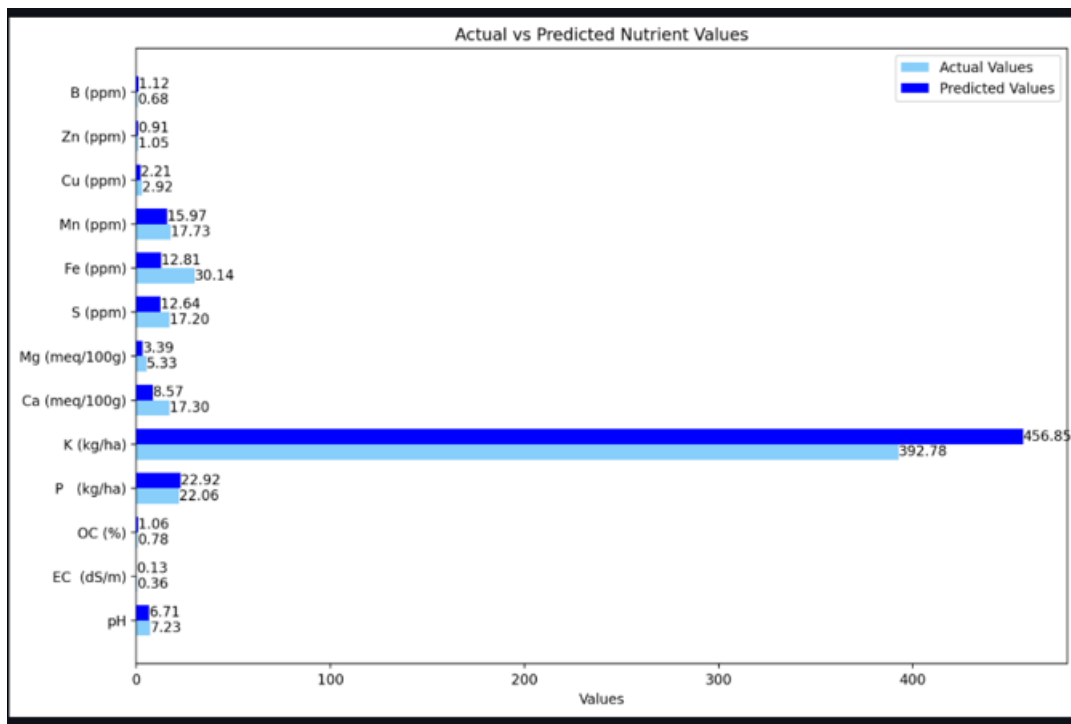
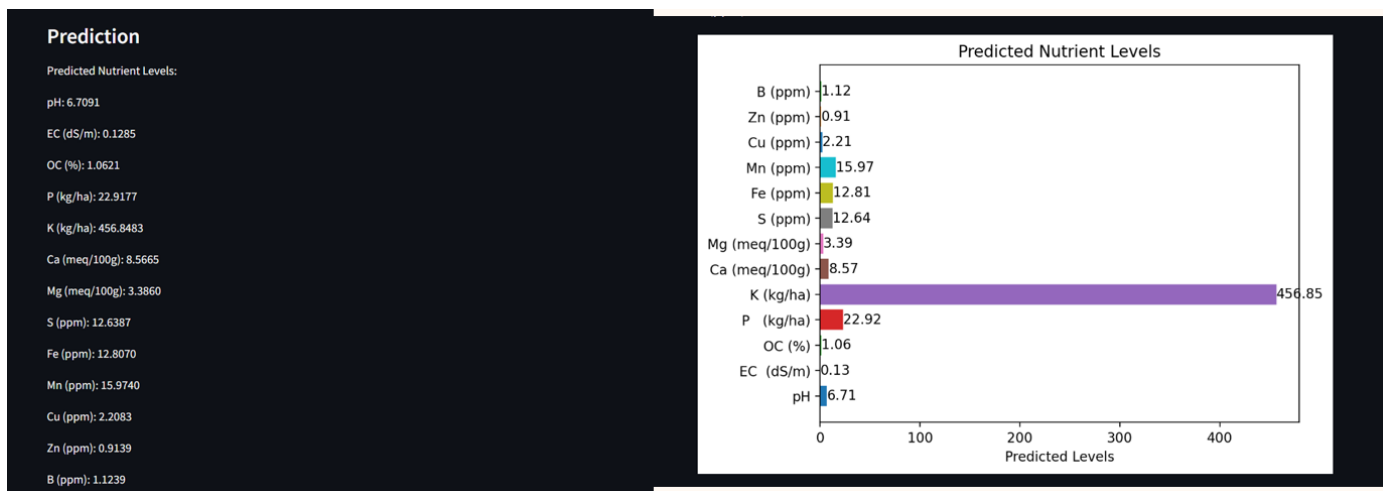
## CHAPTER 7: UI DESIGN

### Model Serialization using Pickle

- **Model Persistence:** Pickle is used to save the trained model to a file. This is critical for reusability, especially when deploying models in production environments.
- **Reusability:** Once the model is saved, it can be loaded later without retraining, saving time and computational resources.

**Actual vs Predicted Comparison:** The chart allows a side-by-side visual comparison of actual nutrient values and the model's predictions.





## How Streamlit Works:

### 1. Streamlit Framework:

- Streamlit is an open-source Python framework designed to turn Python scripts into interactive web applications with minimal effort.
- It is widely used for data visualization, machine learning model deployment, and rapid prototyping.

### 2. Key Concepts:

- Run Python Scripts:
- You create a normal Python script (app.py) and use Streamlit's functions to build a UI.

- Running `streamlit run app.py` launches a local web server, where the app is accessible in the browser.
- Reactive Interface:
- Streamlit apps automatically update when you change code or adjust user inputs.
- It reruns the entire script from top to bottom whenever an interaction (e.g., slider or button click) occurs.

### 3. Components & Features:

- Widgets for Interactivity:
  - Streamlit provides a variety of built-in widgets like sliders, buttons, file uploaders, etc., for easy interactivity.

### Data Visualizations:

- Streamlit integrates seamlessly with popular plotting libraries (e.g., Matplotlib, Plotly, Altair), allowing you to display dynamic charts.
- Auto UI Generation:
  - Streamlit automatically generates a user interface based on the functions in your Python script, without the need for HTML, CSS, or JavaScript.

### 4. Simple App Lifecycle:

- Write Code → Run Streamlit → View in Browser:
  - You write Python code as usual, focusing on logic and analytics.
  - Streamlit handles the rendering of the UI and interactions on the browser.

### 5. Deployment:

- Streamlit apps can be easily deployed using platforms like Streamlit Cloud, or packaged in Docker containers for more advanced deployments.