# CS667: IoT
# Project Report: Smart Recommendation System for Agriculture

Vishal Kumar vishalkmr22@iitk.ac.in
Abhishek Kumar kumara22@iitk.ac.in
Pavan Kumar Ponnaganti pavankp24@iitk.ac.in

November 2024

## 1 Introduction

This project focuses on a smart recommendation system for agriculture. It leverages IoT and machine learning models, including Tiny Neural Network (TNN), Decision Tree, Random Forest, and Random Forest with Grid Search. A Kaggle dataset was used to simulate sensor data, and models were trained to predict fertilizer recommendations and crop production insights.

## 2 Step 1: Training the Models

### 2.1 TNN for Filtering and Prediction

TNN was primarily used to filter and predict data entries when sensors are unavailable. Below is the training process for TNN:

**Data:** Dataset with soil parameters and target labels
**Result:** Trained TNN model for filtering and prediction
**Steps:**

1. Standardize features using `StandardScaler`.
2. Define a neural network with:
   - Input layer with 6 features.
   - Hidden layer with 5 neurons (ReLU activation).
   - Output layer with 1 neuron (Softmax activation).
3. Compile the model using Adam optimizer.
4. Train the model for 10 epochs with batch size of 32.

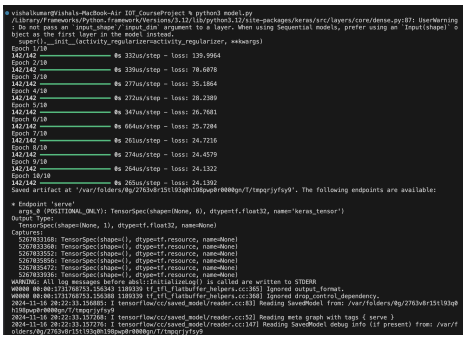


Figure 1: Execution of TNN model and conversion to tflite

Listing 1: Tiny Neural Network Training Code

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

# Model Definition
model = Sequential([
    Dense(6, input_dim=6, activation='relu'),
    Dense(5, activation='relu'),
    Dense(1, activation='softmax')
])

# Compile the model
model.compile(optimizer='adam', loss='categorical_crossentropy',
    metrics=['accuracy'])

# Train the model
model.fit(X_scaled, y, epochs=10, batch_size=32)
```

## 2.2 Decision Tree Model

The Decision Tree model was used for lightweight, interpretable predictions.

Listing 2: Decision Tree Training Code

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import classification_report

# Initialize Decision Tree Classifier
dt = DecisionTreeClassifier(random_state=42)
dt.fit(X_train, y_train)

# Predictions
y_pred = dt.predict(X_test)

# Evaluate Model
print(classification_report(y_test, y_pred))
```

### 2.3 Random Forest Model

Random Forest was utilized for robust and high-accuracy predictions.

Listing 3: Random Forest Training Code

```
from sklearn.ensemble import RandomForestClassifier

# Initialize Random Forest Classifier
rf = RandomForestClassifier(n_estimators=100, random_state=42)
rf.fit(X_train, y_train)

# Predictions
y_pred_rf = rf.predict(X_test)

# Evaluate Model
print(classification_report(y_test, y_pred_rf))
```

### 2.4 Random Forest with Grid Search

Hyperparameter tuning was conducted using Grid Search to optimize the Random Forest model.

Listing 4: Random Forest with Grid Search

```
from sklearn.model_selection import GridSearchCV

# Define Hyperparameters
param_grid = {
    'n_estimators': [100, 200, 300],
    'max_depth': [10, 20, 30]
}

# Grid Search
grid_search = GridSearchCV(RandomForestClassifier(), param_grid, cv
    =3)
grid_search.fit(X_train, y_train)

# Best Parameters and Predictions
print(grid_search.best_params_)
best_model = grid_search.best_estimator_
y_pred_best = best_model.predict(X_test)
```

## 3 Step 2: Model Deployment and Results

### 3.1 Performance Comparison

Below is a comparative analysis of the model performances based on F2 Score, Weighted Precision, and R2 Score:

### 3.2 Data Insights from Plots

The following insights were observed from the dataset:

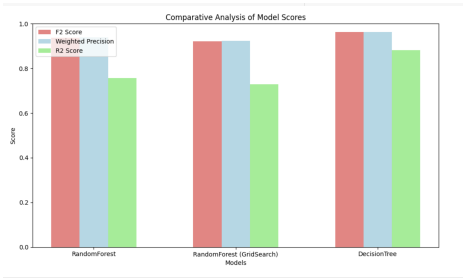


Figure 2: Comparative Analysis of Model Scores

- Comparative analysis of Model Scores
- Distribution of phosphorus levels in the soil:

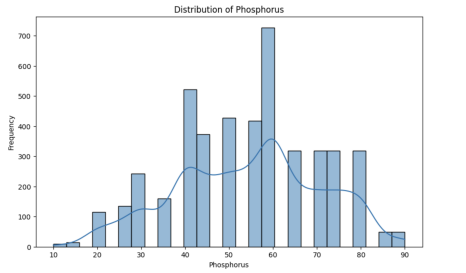


Figure 3: Distribution of Phosphorus

- Distribution of potassium levels in the soil:

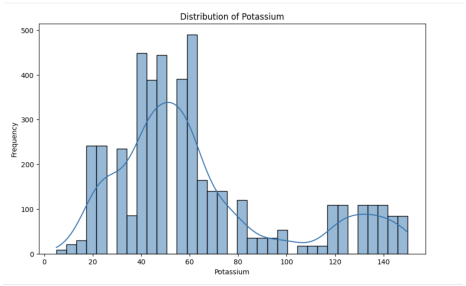


Figure 4: Distribution of Pottasium

- Relation between soil color and fertilizer type:

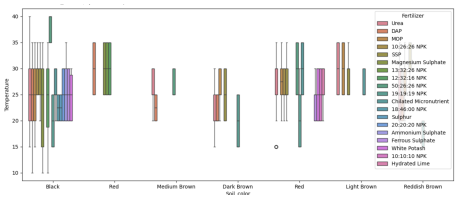

Figure 5: Soil w.r.t Temperature

- Cont Plot and Fertilizer:

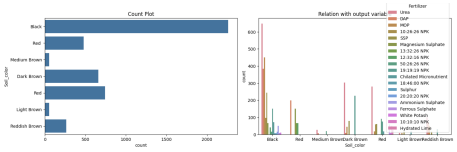

Figure 6: Count Plot and Fertilizers

### 3.3 Use Case of TNN When Sensors Are Off

TNN was deployed to filter and predict rows from the dataset. It can operate independently of sensors, using historical data to provide recommendations.

## 4 Step 3: Deployment on Raspberry Pi

The trained TNN model was converted to TFLite for deployment:

Listing 5: TFLite Conversion Code

```
import tensorflow as tf

# Convert to TFLite
converter = tf.lite.TFLiteConverter.from_keras_model(model)
tflite_model = converter.convert()

# Save the model
with open('tnn_model.tflite', 'wb') as f:
    f.write(tflite_model)
```

## 5 Hardware setup

The hardware setup in the uploaded picture includes the following components and connections:

## 5.1 Components

1. **Raspberry Pi**:
   - Acts as the central processing unit for the IoT system.
   - Used to run the machine learning model and handle MQTT communication.
2. **NodeMCU (ESP8266)**:
   - Used for Wi-Fi connectivity and as an MQTT client for transmitting data to/from the Raspberry Pi.
3. **Arduino Uno**:
   - Used to simulate sensor data.
   - Communicates with the NodeMCU via UART (serial communication).
4. **USB Power Supply**:
   - Provides power to the Raspberry Pi, Arduino, and NodeMCU.

## 5.2 Connections

1. **Raspberry Pi**:
   - Powered through the USB-C cable.
   - USB connection likely connects a flash drive or external peripheral.
2. **NodeMCU**:
   - Connected to a USB port for power.
   - Serial communication wires connect the NodeMCU to the Arduino.
3. **Arduino Uno**:
   - Powered through a USB connection.
   - Serial communication wires (Red and Blue):
     - **TX (Transmit) of Arduino → RX (Receive) of NodeMCU**
     - **RX (Receive) of Arduino → TX (Transmit) of NodeMCU**
   - Ground pins of Arduino and NodeMCU are connected to ensure proper communication.

### 5.3 Communication Workflow

1. **Arduino Uno** simulates sensor data and sends it over serial communication to the **NodeMCU**.
2. **NodeMCU (ESP8266)** forwards the data via Wi-Fi to the **Raspberry Pi** using the MQTT protocol.
3. **Raspberry Pi** receives the data, processes it, and runs the machine learning model for predictions.

## 6 Hardware Setup and Workflow

The following images illustrate the hardware setup and workflow of the system:

- **Hardware Setup:**

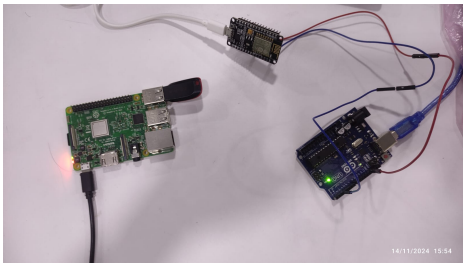

Figure 7: Hardware setup: Raspberry Pi, NodeMCU, and Arduino Uno.

- **Sending Data from Arduino:**

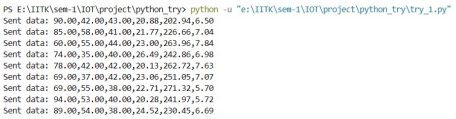


Figure 8: Arduino sending simulated data to NodeMCU.

- **Sending Data to Raspberry Pi:**

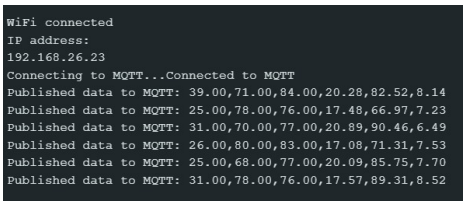


Figure 9: NodeMCU sending aggregated data to Raspberry Pi via MQTT.

- **Data from nodemcu by raspberry pi**

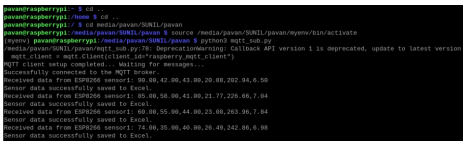


Figure 10: Receiving data from nodemcu by raspberry pi

- **Data to server**


Figure 11: Sending aggregate data to server from raspberry

# 7 Data Processing at server

The data finally received on server after running the ML models shows the predicted fertilizer and crop on the interface below

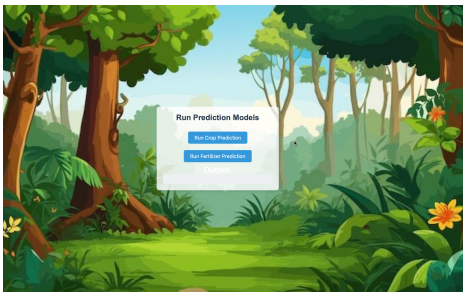


Figure 12: Web Interface

Following is the code for server created

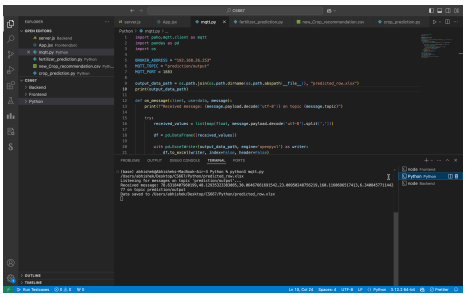

Figure 13: Mqtt.py

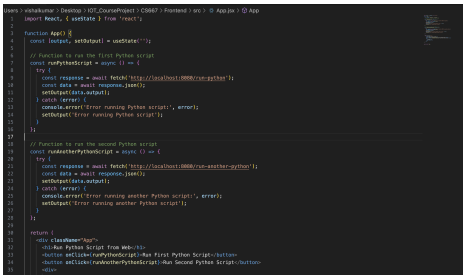

Figure 14: Code for App.jsx

# 8 Mathematical Formulations of the Models

In this section, we describe the mathematical foundations of the machine learning models used in our project.

## 8.1 Tiny Neural Network (TNN)

The Tiny Neural Network (TNN) was used to filter and predict data when sensor input was unavailable. Its formulation is as follows:

### Forward Pass

For each layer $l$ in the network:

$$Z^{(l)} = W^{(l)} \cdot A^{(l-1)} + b^{(l)}$$

$$A^{(l)} = f(Z^{(l)})$$

Where:

- $W^{(l)}$: Weights matrix for layer $l$.
- $b^{(l)}$: Bias vector for layer $l$.

- $A^{(l-1)}$: Activation from the previous layer.
- $f$: Activation function.

**Activation Functions**

- **ReLU (Rectified Linear Unit):**

$$f(x) = \max(0, x)$$

- **Softmax (Output Layer):**

$$f(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$$

**Loss Function**

The loss function used for classification is the categorical cross-entropy:

$$L = -\frac{1}{m} \sum_{i=1}^{m} \sum_{k=1}^{K} y_{i,k} \log(\hat{y}_{i,k})$$

Where:

- $m$: Number of samples.
- $K$: Number of classes.
- $y_{i,k}$: True label (one-hot encoded).
- $\hat{y}_{i,k}$: Predicted probability.

## 8.2 Decision Tree

The Decision Tree model was used for lightweight predictions and is based on splitting the data to maximize information gain.

**Splitting Criterion**

- **Gini Impurity:**

$$Gini = 1 - \sum_{k=1}^{K} p_k^2$$

  Where $p_k$ is the proportion of samples belonging to class $k$.

- **Entropy (optional):**

$$Entropy = -\sum_{k=1}^{K} p_k \log_2(p_k)$$

**Information Gain**

For a split $S$ on feature $X$:

$$IG(S, X) = Entropy(S) - \sum_i \frac{|S_i|}{|S|} Entropy(S_i)$$

Where:

- $S$: Current dataset.
- $S_i$: Subset of $S$ after splitting on $X$.

## 8.3 Random Forest

The Random Forest model uses an ensemble of Decision Trees for robust predictions.

**Prediction**

- **Classification:**

$$\hat{y} = \text{mode}\{T_1(x), T_2(x), \ldots, T_n(x)\}$$

- **Regression (optional):**

$$\hat{y} = \frac{1}{n} \sum_{i=1}^{n} T_i(x)$$

**Random Subsampling**

Each tree in the forest is trained on a bootstrap sample (random sampling with replacement). A random subset of features is considered for splitting at each node.

## 8.4 Random Forest with Grid Search

To optimize the Random Forest model, Grid Search was used to tune hyperparameters like:

- $n_{estimators}$: Number of trees.
- $max_{depth}$: Maximum depth of trees.
- $min_{samples_{split}}$: Minimum samples required to split a node.

**Cross-Validation**

For each hyperparameter configuration:

$$CV_{Score} = \frac{1}{k} \sum_{i=1}^{k} Accuracy_i$$

Where:

- $k$: Number of folds in cross-validation.
- $Accuracy_i$: Accuracy on the $i$-th validation fold.

### 8.5 Evaluation Metrics

The following metrics were used to evaluate model performance:

- **Accuracy:**

$$Accuracy = \frac{\text{Number of Correct Predictions}}{\text{Total Number of Predictions}}$$

- **Precision:**

$$Precision_k = \frac{TP_k}{TP_k + FP_k}$$

- **Recall:**

$$Recall_k = \frac{TP_k}{TP_k + FN_k}$$

- **F2 Score:**

$$F2 = \frac{(1 + 2^2) \cdot Precision \cdot Recall}{2^2 \cdot Precision + Recall}$$

  Where:

  - $TP_k$: True Positives for class $k$.
  - $FP_k$: False Positives for class $k$.
  - $FN_k$: False Negatives for class $k$.

## 9 How Our Model is an Optimization to Previous Work

Our proposed model introduces several optimizations over traditional approaches, making it more efficient and suitable for IoT devices in agricultural systems.

### 9.1 Independence from Sensors

- The model can operate independently of sensors by using **TNN-predicted rows**. This ensures that even in the absence of live sensor data, the system can continue making accurate predictions using historical data or simulated inputs.
- **Optimization 1:** The system does not need to operate 24x7, as it can rely on stored or predicted data. This reduces dependency on continuous real-time sensor availability, thus saving power and communication costs.

### 9.2 Efficient Data Filtering for Advanced Models

- The use of TNN for filtering large chunks of data enables the system to preprocess and clean the data effectively. This filtered data can then be used as input for more computationally expensive models like **Decision Tree** and **Random Forest**.
- **Case 2:** By running Decision Tree, Random Forest, or Grid-Search-optimized Random Forest on filtered data, the system ensures both computational efficiency and improved prediction accuracy.

### 9.3 Cost Efficiency for IoT Devices

- The system is highly **cost-efficient** for IoT devices as it leverages lightweight models like TNN for data processing. These models require minimal computational resources, making them suitable for deployment on devices with limited processing power (e.g., Raspberry Pi or NodeMCU).
- Advanced models (e.g., Random Forest) are selectively used for crucial decision-making, further optimizing resource utilization while maintaining high prediction accuracy.

## 10 Conclusion

The project demonstrates an optimized IoT-based agricultural recommendation system that integrates hardware, machine learning, and a user-friendly web interface. The hardware setup includes a Raspberry Pi, NodeMCU (ESP8266), and Arduino Uno, where simulated sensor data is transmitted via serial communication and MQTT protocol. The system introduces significant optimizations over traditional approaches, starting with the ability to operate independently of sensors using Tiny Neural Network (TNN)-predicted rows. This ensures continuity in predictions without the need for 24x7 sensor operations, making it cost-efficient and power-saving.

The Raspberry Pi processes the data using advanced machine learning models, including Decision Tree, Random Forest, and Grid-Search-optimized Random Forest, to refine predictions from filtered data. This layered approach

enables computational efficiency while maintaining high accuracy for fertilizer recommendations and crop insights. The predictions are published via MQTT and stored in an Excel file for further analysis. A React-based web interface allows users to trigger Python scripts for real-time processing or visualization, with outputs displayed interactively.

This system is not only robust and flexible but also highly optimized for IoT devices. It leverages lightweight models like TNN for preprocessing and resource-efficient deployment through TFLite, while advanced models handle more complex decision-making. The seamless integration of hardware, backend, and frontend ensures a continuous flow of data and insights, enabling smarter agricultural decisions at reduced operational costs and even when sensors are offline. This makes it a scalable and practical solution for real-world agricultural applications.