**🧠 Machine L earning Model Training - Line-by-Line Explanation**

**📦 Part 1: Data Preprocessing**

**Splitting Features and Target**

X = df.drop(columns=['label'])

y = df['label']

* **X**: All input features (N, P, K, temperature, humidity, ph, rainfall)
* **y**: Target variable (crop names like 'rice', 'maize', etc.)
* We separate them because ML models learn to map X → y

**Encoding Crop Names to Numbers**

label\_encoder = LabelEncoder()

y\_encoded = label\_encoder.fit\_transform(y)

* **Problem**: ML models can't work with text ('rice', 'maize')
* **Solution**: Convert to numbers (0, 1, 2, 3...)
* fit\_transform(): Learns all unique crop names AND converts them
* Example: 'rice' → 0, 'maize' → 1, 'chickpea' → 2

**First Split: Training + Validation vs. Testing**

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

X, y\_encoded, test\_size=0.2, random\_state=42, stratify=y\_encoded

)

**Breaking it down:**

* X\_train\_full: 80% of features for training
* X\_test: 20% of features for final testing (model never sees this!)
* y\_train\_full: 80% of labels for training
* y\_test: 20% of labels for testing
* test\_size=0.2: Keep 20% for testing
* random\_state=42: Ensures same split every time (reproducibility)
* stratify=y\_encoded: Keeps same crop proportions in both sets

**Why?** We need unseen data to test if the model truly learned patterns, not just memorized.

**Second Split: Training vs. Validation**

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

X\_train\_full, y\_train\_full, test\_size=0.2, random\_state=42, stratify=y\_train\_full

)

Now we split the training data further:

* **X\_train** (64% of original): Actually train the model
* **X\_val** (16% of original): Check performance during training
* **X\_test** (20% of original): Final evaluation (still untouched)

**Final split:**

* Training: 64%
* Validation: 16%
* Testing: 20%

**Feature Scaling**

scaler = StandardScaler()

X\_train\_scaled = scaler.fit\_transform(X\_train)

X\_val\_scaled = scaler.transform(X\_val)

X\_test\_scaled = scaler.transform(X\_test)

**Why scale?**

* Features have different ranges:
  + N: 0-140
  + pH: 3.5-9.9
  + Rainfall: 20-300
* StandardScaler makes all features have:
  + Mean = 0
  + Standard deviation = 1

**Important:**

* fit\_transform() on **training** data: Learn mean/std AND scale
* transform() on **val/test** data: Use training's mean/std to scale
* **Why?** Prevents data leakage (test data shouldn't influence scaling)

**Quick Info Check**

n\_classes = len(np.unique(y\_encoded))

print({

'train': X\_train.shape,

'val': X\_val.shape,

'test': X\_test.shape,

'classes': n\_classes

})

Prints dataset sizes and number of crop types (classes).

**🔬 Part 2: Model Evaluation Function**

def evaluate\_model(name: str, model, X\_train, y\_train, X\_val, y\_val,

X\_test, y\_test, label\_encoder, cmap\_color: str) -> Dict[str, Any]:

This function evaluates any ML model and returns performance metrics.

**Make Predictions**

y\_pred\_train = model.predict(X\_train)

y\_pred\_val = model.predict(X\_val)

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

Get predictions on all three datasets.

**Calculate Accuracy**

train\_acc = accuracy\_score(y\_train, y\_pred\_train)

val\_acc = accuracy\_score(y\_val, y\_pred\_val)

test\_acc = accuracy\_score(y\_test, y\_pred\_test)

**Accuracy** = (Correct predictions) / (Total predictions)

* Example: 990 correct out of 1000 = 99% accuracy

**Calculate Additional Metrics**

precision = precision\_score(y\_test, y\_pred\_test, average='weighted')

recall = recall\_score(y\_test, y\_pred\_test, average='weighted')

f1 = f1\_score(y\_test, y\_pred\_test, average='weighted')

**Precision**: Of all crops we predicted as "rice", how many were actually rice? **Recall**: Of all actual rice samples, how many did we correctly identify? **F1 Score**: Harmonic mean of precision and recall (balanced metric)

* average='weighted': Accounts for class imbalance

**Print Results**

print(f"\n{name}")

print(f"Train Acc: {train\_acc:.4f} | Val Acc: {val\_acc:.4f} | Test Acc: {test\_acc:.4f}")

print(f"Precision: {precision:.4f} | Recall: {recall:.4f} | F1: {f1:.4f}")

Display all metrics with 4 decimal places.

**Confusion Matrix Visualization**

cm = confusion\_matrix(y\_test, y\_pred\_test)

plt.figure(figsize=(6,5))

sns.heatmap(cm, annot=True, fmt='d', cmap=cmap\_color, cbar=False)

plt.title(f"{name} - Confusion Matrix (Test)")

plt.xlabel("Predicted")

plt.ylabel("Actual")

plt.tight\_layout()

plt.show()

**Confusion Matrix**: Shows where model gets confused

* Rows: Actual crops
* Columns: Predicted crops
* Diagonal: Correct predictions
* Off-diagonal: Mistakes

**Return Results**

return {

'Model': name,

'Train Accuracy': train\_acc,

'Val Accuracy': val\_acc,

'Test Accuracy': test\_acc,

'F1 Score': f1

}

Returns dictionary for comparison table.

**🏆 Part 3: Training and Comparing Models**

**Define 6 Different Models**

models = {

'Logistic Regression': LogisticRegression(max\_iter=1000),

'Decision Tree': DecisionTreeClassifier(random\_state=42),

'Random Forest': RandomForestClassifier(random\_state=42),

'KNN': KNeighborsClassifier(),

'SVM': SVC(probability=False),

'Naive Bayes': GaussianNB(),

}

**Each model has different strengths:**

1. **Logistic Regression**: Fast, simple, linear relationships
2. **Decision Tree**: Easy to interpret, captures non-linear patterns
3. **Random Forest**: Multiple trees voting, very accurate
4. **KNN**: Classifies based on nearest neighbors
5. **SVM**: Finds optimal boundary between classes
6. **Naive Bayes**: Fast, works well with independent features

**Color Scheme for Visualizations**

colors = {

'Logistic Regression': 'Blues',

'Decision Tree': 'Greens',

'Random Forest': 'Oranges',

'KNN': 'Purples',

'SVM': 'Reds',

'Naive Bayes': 'coolwarm',

}

Different color for each model's confusion matrix.

**Train All Models and Collect Results**

results = []

for name, model in models.items():

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

res = evaluate\_model(

name, model,

X\_train\_scaled, y\_train,

X\_val\_scaled, y\_val,

X\_test\_scaled, y\_test,

label\_encoder,

colors[name]

)

results.append(res)

**Step-by-step:**

1. Loop through each model
2. model.fit(): Train the model on training data
3. evaluate\_model(): Test it and get metrics
4. results.append(): Save results for comparison

**Create Comparison Table**

results\_df = pd.DataFrame(results)

print("\nOverall Comparison:\n", results\_df)

Converts list of dictionaries to a neat table showing all model performances.

**Visualize Comparison**

plt.figure(figsize=(8,4))

sns.heatmap(results\_df.set\_index('Model')[['Train Accuracy', 'Val Accuracy', 'Test Accuracy']],

annot=True, fmt='.3f', cmap='viridis')

plt.title('Model Performance Comparison')

plt.tight\_layout()

plt.show()

Creates a heatmap comparing all models side-by-side.

* Darker colors = higher accuracy
* Easy to spot the best performer

**Select Best Model**

best\_row = results\_df.sort\_values('Test Accuracy', ascending=False).iloc[0]

best\_name = best\_row['Model']

print("\nBest model:", best\_name)

best\_model = models[best\_name]

**Breaking it down:**

1. sort\_values('Test Accuracy', ascending=False): Sort by test accuracy (highest first)
2. .iloc[0]: Get the top row
3. best\_row['Model']: Extract the model name
4. models[best\_name]: Get the actual trained model object

**📊 Summary Workflow**

1. Load Data

↓

2. Split: 80% train, 20% test

↓

3. Split train further: 80% train, 20% validation

↓

4. Scale features (normalize)

↓

5. Train 6 different models

↓

6. Evaluate each on train/val/test

↓

7. Compare all models

↓

8. Select best performer

↓

9. Use best model for app

**🎯 Key Concepts**

**Why Three Datasets?**

* **Training**: Model learns patterns
* **Validation**: Check if learning is going well (tune hyperparameters)
* **Testing**: Final honest evaluation (model never saw this)

**Why Scaling?**

* Ensures all features are treated equally
* Some algorithms (KNN, SVM) are sensitive to feature scales

**Why Multiple Models?**

* No single "best" algorithm for all problems
* Try several, pick the winner
* Random Forest usually wins for tabular data (like this)

**Why Stratify?**

* Keeps same crop proportions in all splits
* Prevents having all rice samples in training and none in testing

**💡 Expected Output**

Typically Random Forest wins with:

* **Train Accuracy**: ~100% (might overfit slightly)
* **Validation Accuracy**: ~99%
* **Test Accuracy**: ~99%
* **F1 Score**: ~99%

This model then gets saved and used in the Streamlit app!

**App.py**

**🌱 Crop Recommendation System - Detailed Explanation**

**Overview**

This is a **web-based machine learning application** built with Streamlit that recommends the best crop to grow based on soil and climate conditions.

**🏗️ System Architecture**

**1. Core Components**

**Libraries Used**

* **Streamlit**: Creates the interactive web interface
* **Pandas & NumPy**: Data manipulation and numerical operations
* **Matplotlib & Seaborn**: Data visualization (static charts)
* **Plotly**: Interactive charts and graphs
* **Scikit-learn**: Machine learning functionality
* **Joblib**: Saves and loads trained models

**Pre-trained Models Loaded**

model = joblib.load('best\_crop\_model.pkl') # The ML model

scaler = joblib.load('scaler.pkl') # Normalizes input data

label\_encoder = joblib.load('label\_encoder.pkl') # Converts crop names to numbers

**📊 Data & Features**

**Input Features (7 parameters)**

1. **N** - Nitrogen content in soil
2. **P** - Phosphorus content in soil
3. **K** - Potassium content in soil
4. **Temperature** - In degrees Celsius
5. **Humidity** - Relative humidity percentage
6. **pH** - Soil acidity/alkalinity level
7. **Rainfall** - In millimeters

**Output**

* **Recommended crop** from 18 possible crops (rice, maize, chickpea, etc.)

**🎯 How It Works**

**Step 1: User Input**

Users adjust sliders in the sidebar to enter their:

* Soil nutrient levels (N, P, K)
* Climate conditions (temperature, humidity, rainfall)
* Soil pH

**Step 2: Prediction Process**

# 1. Create array from user inputs

input\_array = np.array([[N, P, K, temp, humidity, ph, rainfall]])

# 2. Scale the data (normalize to similar ranges)

input\_scaled = scaler.transform(input\_array)

# 3. Get prediction from model

prediction\_encoded = model.predict(input\_scaled)[0]

# 4. Convert number back to crop name

predicted\_crop = label\_encoder.inverse\_transform([prediction\_encoded])[0]

**Step 3: Display Results**

Shows the recommended crop with:

* Input summary
* Ideal growing conditions
* Water needs, temperature preferences, soil type

**📱 User Interface (4 Tabs)**

**Tab 1: 🏠 Recommendation**

* **Main prediction display** with recommended crop
* **Input summary** showing all entered values
* **Ideal growing conditions** for the crop
* **Interactive gauges** for temperature and humidity

**Tab 2: 📊 Data Analysis**

* **Crop distribution chart** - How many samples of each crop in dataset
* **Feature distributions** - Histogram for any selected parameter
* **Correlation heatmap** - Shows relationships between features
* **Scatter plots** - Compare any two features with color coding

**Tab 3: 📈 Model Performance**

* **Accuracy comparison** of different ML models:
  + Random Forest (99%)
  + SVM (98%)
  + KNN (97%)
  + Decision Tree (96%)
  + Logistic Regression (95%)
  + Naive Bayes (90%)

**Tab 4: ℹ️ Crop Info**

* **Detailed crop requirements** for any selected crop
* **Average conditions** needed (temp, humidity, pH, etc.)
* **Acceptable ranges** visualization

**🎨 Design Features**

**Custom Styling**

# Green agricultural theme

- Main color: #2E8B57 (Sea Green)

- Rounded prediction boxes

- Clean metric displays

- Professional layout

**Caching for Performance**

@st.cache\_resource # Loads models once, reuses them

@st.cache\_data # Loads data once, reuses it

This makes the app faster by not reloading data on every interaction.

**🌾 Crop Information Database**

The system has built-in knowledge about **18 crops**:

* **Cereals**: Rice, Maize
* **Pulses**: Chickpea, Lentil, Mungbean, Blackgram, Mothbeans, Pigeonpeas, Kidneybeans
* **Fruits**: Apple, Banana, Grapes, Mango, Orange, Papaya, Pomegranate, Watermelon, Muskmelon

Each crop has info on:

* **Water needs**: Low/Medium/High
* **Temperature preference**: Cool/Moderate/Warm/Hot
* **Soil type**: Sandy, Clayey, Loamy, etc.

**🔧 Technical Workflow**

**Before Running This App:**

1. Train ML models on crop dataset
2. Save models using joblib
3. Ensure Crop\_recommendation.csv exists

**When App Runs:**

1. Load models and data (cached)
2. Display UI with sliders
3. Wait for user input
4. Process prediction when button clicked
5. Show results with visualizations

**💡 Key Machine Learning Concepts**

**Scaling/Normalization**

Different features have different ranges:

* N: 0-140
* Temperature: 8-43°C
* Rainfall: 20-300mm

Scaling ensures all features are treated equally by the model.

**Label Encoding**

ML models work with numbers, not text:

* "rice" → 0
* "maize" → 1
* "chickpea" → 2 (etc.)

**Prediction**

The model learned patterns like:

* High rainfall + warm temp → Rice
* Low rainfall + cool temp → Chickpea Then applies these patterns to new inputs.

**🎯 Use Cases**

1. **Farmers**: Decide what crop to plant based on current conditions
2. **Agricultural advisors**: Quick recommendations for clients
3. **Students**: Learn about crop requirements and ML applications
4. **Researchers**: Analyze crop-climate relationships

**🚀 Running the App**

streamlit run app.py

Opens in browser at http://localhost:8501

**📝 Summary**

This is a **complete end-to-end ML application** that:

* ✅ Takes real-world agricultural data
* ✅ Uses machine learning to make predictions
* ✅ Presents results in an intuitive, interactive interface
* ✅ Provides educational insights about crops and data
* ✅ Helps make practical farming decisions

The combination of ML accuracy (99%) with user-friendly design makes it a powerful tool for agricultural decision-making!