PROJECT REPORT

“FARMERLY”

Manjunath Hegde

SUMMARY

The innovative "Farmerly" initiative represents a groundbreaking data-driven solution strategically crafted to tackle the multifaceted challenges confronting farmers as they endeavor to optimize crop selection. Rooted in the principles of sustainable development, this project seamlessly aligns with the United Nations' ambitious Sustainable Development Goals (SDGs), specifically targeting Goal 2—Zero Hunger, and Goal 8—Promote sustained, inclusive, and sustainable economic growth, full and productive employment, and decent work for all.

At its core, Farmerly harnesses the power of advanced machine learning models to provide farmers with tailored recommendations for crop selection. By analyzing diverse environmental factors, the system offers intelligent insights that empower farmers to make informed decisions, thereby enhancing agricultural productivity and contributing to the overarching objective of eradicating hunger. Notably, the platform goes beyond mere crop recommendations, extending its support to include yield predictions and credit rating assessments for farmers.

In essence, Farmerly represents a holistic approach to addressing agricultural challenges, leveraging cutting-edge technology to create a sustainable and inclusive ecosystem for farmers. By fostering economic growth and ensuring food security, this initiative stands as a beacon of progress, actively contributing to the realization of the United Nations' Sustainable Development Goals on a global scale.

Enhancing Agriculture through Smart Farming and Technology

Table of Contents

6. **Feature Scaling**
    - Min-Max Scaling and Standardization
7. **Machine Learning Model**
    - Evaluating Different Models
    - Choosing the Best Suitable Model (Random Forest)
8. **Predictive Analysis**
    - User Input for Crop Recommendation
    - Yield Prediction
9. **Calculating Yield**
    - Merging Datasets
    - Training a Model for Yield Prediction
10. **Calculating Credit**
    - Calculating Credit for Farmers
11. **Future Scope**
    - Website Integration
    - Database Integration
    - Model and Data Dumping
12. **Containerization and Deployment**
    - Dockerizing the Application
    - Hosting on Cloud-Based Platform (AWS/Heroku)
13. **Conclusion**
    - Summary of Achievements
    - Lessons Learned
    - Future Enhancements

1. Introduction

Background

The agriculture sector plays a crucial role in global economies, contributing to food security and livelihoods. However, farmers face challenges such as unpredictable weather, resource management, and financial constraints. This project aims to address these issues by leveraging data science, smart farming techniques, and technology.

Significance of the Project

The project aligns with United Nations Sustainable Development Goals (SDGs) by addressing Goal 2 (Zero Hunger) and Goal 8 (Sustainable Economic Growth). The use of technology in agriculture enhances productivity, reduces poverty, and contributes to global food security.

2. Project Overview

Selected UN SDGs

1. **Goal 2 - Zero Hunger:** Enhancing food production through smart farming practices.
2. **Goal 8 - Sustainable Economic Growth:** Promoting economic growth in the agricultural sector.

Objectives

- **Smart Farming Tips:** Providing farmers with data-driven insights for optimal crop selection.
- **Technology to Help Farmers:** Using machine learning to predict crop suitability based on environmental factors.
- **Financial Help for Farmers:** Offering financial support by analyzing income and expenditure data.

- **Putting Everything Together:** Integrating farming tips and financial assistance for informed decision-making.

3. Data Integration & Processing

Importing Libraries

```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
%matplotlib inline
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, LabelEncoder
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score
    import requests
    import warnings
    warnings.simplefilter(action='ignore', category=FutureWarning)
    warnings.simplefilter(action='ignore', category=UserWarning)
```

14. **Importing Libraries:**
    - `numpy`, `pandas`, `seaborn`, and `matplotlib.pyplot` are imported for data manipulation and visualization.
    - `%matplotlib inline` is a magic command that allows for inline plotting in Jupyter notebooks.
    - `train_test_split` from scikit-learn is imported to split the dataset into training and testing sets.
    - `StandardScaler` and `LabelEncoder` from scikit-learn are imported for feature scaling and label encoding.
    - `RandomForestClassifier` is imported for implementing a random forest classification model.
    - `accuracy_score` is imported to evaluate the accuracy of the model.
15. **Disabling Warnings:**
    - Two types of warnings, `FutureWarning` and `UserWarning`, are suppressed using the `warnings` module to enhance the code's readability.
16. **Machine Learning Workflow:**
    - The code suggests that the dataset needs to be prepared for a machine learning task.
    - It likely involves loading the data into a pandas DataFrame and preprocessing steps, such as handling missing values and encoding categorical variables.

17. **Train-Test Split:**
    - The `train_test_split` function is likely used to split the dataset into training and testing sets, a common practice in machine learning.
18. **Feature Scaling and Label Encoding:**
    - Standard scaling of features using `StandardScaler` and label encoding of categorical variables using `LabelEncoder` are common preprocessing steps to prepare the data for machine learning models.
19. **Random Forest Classifier:**
    - The RandomForestClassifier is employed to build a classification model. Random Forest is an ensemble learning method that combines multiple decision trees to improve overall model performance.
20. **Model Training:**
    - The script probably involves fitting the RandomForestClassifier to the training data using the `fit` method.
21. **Model Evaluation:**
    - The `accuracy_score` is likely used to evaluate the model's accuracy on the test set, providing a performance metric for the trained classifier.
22. **Visualization:**
    - Seaborn and Matplotlib may be used for visualizing the results, such as plotting confusion matrices or feature importance.

Importing Dataset

```
1. crop = pd.read_csv('Farmerly.csv')
2. X = crop.iloc[:, :-1].values
3. y = crop.iloc[:, -1].values
```

4. **Reading the CSV File:**
    - `crop = pd.read_csv('Farmerly.csv')`: This line reads the CSV file 'Farmerly.csv' into a pandas DataFrame named 'crop'. The DataFrame is a two-dimensional, tabular data structure that can hold and manipulate the data.
5. **Extracting Feature Matrix (X) and Target Variable (y):**
    - `X = crop.iloc[:, :-1].values`: This line extracts the feature matrix (X) from the DataFrame 'crop'. It uses the `iloc` method to select all rows (`:`) and all columns except the last one (`[:-1]`). The `.values` converts the

selected data into a NumPy array, which is a common format for input data in machine learning.

- **`y = crop.iloc[:, -1].values`**: This line extracts the target variable (y) from the DataFrame 'crop'. It uses **`iloc`** to select all rows and only the last column (**`[:, -1]`**). Similar to X, the **`.values`** converts the selected data into a NumPy array.

Data Cleaning

```
from sklearn.impute import SimpleImputer
imputer = SimpleImputer(missing_values = np.nan, strategy = 'mean')

imputer.fit(X[:, 0:1])
X[: , 0:1] = imputer.transform(X[:, 0:1])
```

1. **Importing SimpleImputer:**
   - **`from sklearn.impute import SimpleImputer`**: This line imports the **`SimpleImputer`** class from scikit-learn, a library for machine learning in Python. The **`SimpleImputer`** is used for handling missing data by imputing or filling in the missing values.
2. **Creating an Imputer Instance:**
   - **`imputer = SimpleImputer(missing_values=np.nan, strategy='mean')`**: This line creates an instance of **`SimpleImputer`**. The **`missing_values`** parameter specifies the placeholder for missing values (in this case, **`np.nan`**), and the **`strategy`** parameter defines the imputation strategy, which is set to 'mean'. The 'mean' strategy replaces missing values with the mean of the non-missing values in the specified column.
3. **Fitting the Imputer:**
   - **`imputer.fit(X[:, 0:1])`**: This line fits the imputer on the selected data, which is the first column (**`0:1`**) of the feature matrix **`X`**. The imputer calculates and stores the mean of the non-missing values in that column.
4. **Transforming and Imputing Missing Values:**
   - **`X[:, 0:1] = imputer.transform(X[:, 0:1])`**: This line transforms the selected column in the feature matrix **`X`** by replacing missing values with the mean calculated during the fitting step. The transformed column is then assigned back to the original feature matrix.

Exploratory Data Analysis

Heatmap to check null/missing values

```
sns.heatmap(crop.isnull(),cmap="coolwarm")
plt.show()
```

1. **Seaborn Heatmap:**
   - `sns.heatmap(crop.isnull(), cmap="coolwarm")`: This line uses the Seaborn library to create a heatmap. The `crop.isnull()` part generates a DataFrame of the same shape as 'crop', where each entry is either `True` if the corresponding element in 'crop' is missing (null), or `False` if it's not missing. The `heatmap` function then visualizes this binary pattern, using the 'coolwarm' colormap to represent missing values (True) as distinct colors.
2. **Matplotlib Show:**
   - `plt.show()`: This line uses Matplotlib to display the created heatmap. The `plt.show()` function is a standard way to render and display the plot.

Distribution of temperature and PH

```
plt.figure(figsize=(12,5))
plt.subplot(1, 2, 1)
sns.distplot(crop['temperature'],color="purple",bins=15,hist_kws={'alpha
':0.2})
plt.subplot(1, 2, 2)
sns.distplot(crop['ph'],color="green",bins=15,hist_kws={'alpha':0.2})
```

1. **Setting Figure Size:**
   - `plt.figure(figsize=(12, 5))`: This line sets the size of the entire figure to be displayed. The width is set to 12 units, and the height is set to 5 units.
2. **Creating Subplots:**
   - `plt.subplot(1, 2, 1)`: This line creates a subplot grid with 1 row and 2 columns and selects the first subplot (index 1) for the subsequent plot. The first subplot will be on the left side of the figure.

- `plt.subplot(1, 2, 2)`: This line selects the second subplot (index 2) for the subsequent plot. The second subplot will be on the right side of the figure.

3. **Plotting the First Distribution:**
   - `sns.distplot(crop['temperature'], color="purple", bins=15, hist_kws={'alpha':0.2})`: This line uses Seaborn to create a distribution plot (histogram) of the 'temperature' column from the 'crop' DataFrame. The plot is colored purple, divided into 15 bins, and the transparency of the bars is adjusted with the `alpha` parameter to make overlapping bars more visible.
4. **Plotting the Second Distribution:**
   - `sns.distplot(crop['ph'], color="green", bins=15, hist_kws={'alpha':0.2})`: This line creates a distribution plot for the 'ph' column from the 'crop' DataFrame. The plot is colored green, divided into 15 bins, and the transparency of the bars is adjusted similarly.
5. **Displaying the Plot:**
   - `plt.show()`: This line displays the entire figure with both subplots. This is a standard way to render and show the plots in Matplotlib.

Visualize what Data says

```
sns.countplot(y='label',data=crop, palette="plasma_r")
```

1. **Seaborn Countplot:**
   - `sns.countplot(y='label', data=crop, palette="plasma_r")`: This line generates a countplot using Seaborn. The 'label' column of the 'crop' DataFrame is used as the categorical variable on the y-axis, and the count of occurrences for each category is represented by the height of the bars.
2. **Specifying Data and Palette:**
   - `y='label'`: This parameter specifies that the 'label' column should be plotted on the y-axis.
   - `data=crop`: This parameter indicates that the data for the countplot is taken from the 'crop' DataFrame.
   - `palette="plasma_r"`: This parameter sets the color palette for the plot. In this case, "plasma_r" is a perceptually uniform color map, and the "_r" suffix indicates that the colors are reversed.
3. **Interpretation:**

- The countplot provides a visual representation of the distribution of categories in the 'label' column. Each bar represents the count of occurrences for a specific category.
- The use of a horizontal bar chart (`y='label'`) is suitable when dealing with a categorical variable with a large number of categories, as it prevents long category names from overlapping.

4. **Color Palette:**
   - The chosen color palette ("plasma_r") enhances the visual appeal of the plot. The "_r" suffix reverses the order of colors in the palette.

```
sns.pairplot(crop, hue = 'label')
```

1. **Seaborn Pairplot:**
   - `sns.pairplot(crop, hue='label')`: This line generates a pairplot using Seaborn. The 'crop' DataFrame is passed as the data source. The `hue='label'` parameter is used to color the scatterpoints based on the values in the 'label' column, which is a categorical variable.
2. **Specifying Data and Hue:**
   - `crop`: This parameter specifies the DataFrame containing the data to be plotted.
   - `hue='label'`: This parameter adds a color dimension to the plot, where different colors are used for different values in the 'label' column. This is particularly useful when exploring relationships between variables in the context of different categories.
3. **Interpretation:**
   - The pairplot creates scatterplots for all possible pairs of numerical columns in the DataFrame. Each point in the scatterplot represents an observation, and the position of the point is determined by the values of two variables. The color of the points is determined by the category in the 'label' column.
4. **Diagonal Axes:**
   - The diagonal axes of the pairplot show histograms or kernel density estimates for each numerical variable. This provides a univariate distribution of each variable.
5. **Off-Diagonal Axes:**
   - The off-diagonal scatterplots show bivariate relationships between pairs of variables. The points are colored based on the 'label' column, allowing for the visual exploration of how the distribution of points varies across different categories.

4. Data Encoding

```
crop_dict = {
    'rice': 1,
    'maize': 2,
    'jute': 3,
    'cotton': 4,
    'coconut': 5,
    'papaya': 6,
    'orange': 7,
    'apple': 8,
    'muskmelon': 9,
    'watermelon': 10,
    'grapes': 11,
    'mango': 12,
    'banana': 13,
    'pomegranate': 14,
    'lentil': 15,
    'blackgram': 16,
    'mungbean': 17,
    'mothbeans': 18,
    'pigeonpeas': 19,
    'kidneybeans': 20,
    'chickpea': 21,
    'coffee': 22
}
crop['crop_num']=crop['label'].map(crop_dict)
```

1. **Crop Dictionary:**
    - **`crop_dict = {...}`**: This dictionary maps crop names to numerical labels. Each crop is assigned a unique integer value. For example, 'rice' is mapped to 1, 'maize' to 2, and so on.
2. **Mapping Labels to Numbers:**
    - **`crop['crop_num'] = crop['label'].map(crop_dict)`**: This line creates a new column 'crop_num' in the 'crop' DataFrame. It uses the **`map`** function to map the values in the 'label' column to their corresponding numerical labels as defined in the 'crop_dict'. The result is a new column where each crop label is replaced with its numerical representation.
    - For example, if a row in the 'label' column contains 'rice', the corresponding value in the 'crop_num' column will be 1.

Purpose:

- By converting categorical labels into numerical representations, the data becomes compatible with models that expect numerical input features.

5. Step 5 - Data : Crop, Split

Cropping Data

```
X = crop.drop(['crop_num','label'],axis=1)
y = crop['crop_num']
```

1. **Features (X):**
    - **`X = crop.drop(['crop_num', 'label'], axis=1)`**: This line creates a new DataFrame 'X' that contains the features for the machine learning model. It is obtained by dropping two columns from the 'crop' DataFrame: 'crop_num' and 'label'. The **`axis=1`** parameter indicates that the operation is performed along columns.
    - In other words, 'X' will include all the columns from the original 'crop' DataFrame except for 'crop_num' and 'label'. These remaining columns are the features used for training the machine learning model.
2. **Target Variable (y):**
    - **`y = crop['crop_num']`**: This line creates a Series 'y' that contains the target variable for the machine learning model. It is obtained by selecting the 'crop_num' column from the 'crop' DataFrame.
    - 'y' represents the numerical labels corresponding to the crops. This column will be used as the target variable that the machine learning model aims to predict.

Split - Training set, Testing set

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

1. **Importing the Necessary Function:**
    - **`from sklearn.model_selection import train_test_split`**: This line imports the **`train_test_split`** function from scikit-learn. This function is commonly used to split a dataset into training and testing sets.
2. **Splitting the Dataset:**

- `X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)`: This line uses the `train_test_split` function to split the features (`X`) and target variable (`y`) into training and testing sets. The parameters are as follows:
    - `X`: The features (input variables) of the dataset.
    - `y`: The target variable (output variable) of the dataset.
    - `test_size=0.2`: This parameter specifies the proportion of the dataset to include in the test split. Here, 20% of the data will be used for testing, and 80% for training.
    - `random_state=42`: This parameter sets a seed for the random number generator, ensuring reproducibility. The same seed will produce the same split each time the code is run.

3. **Resulting Variables:**
    - `X_train`: The training set of features.
    - `X_test`: The testing set of features.
    - `y_train`: The training set of target variable values.
    - `y_test`: The testing set of target variable values.

6. Feature Scaling

Scaling

1. Min Max Scaler

```
from sklearn.preprocessing import MinMaxScaler
ms = MinMaxScaler()

X_train = ms.fit_transform(X_train)
X_test = ms.transform(X_test)
```

7. **Importing the Scaler:**
    - `from sklearn.preprocessing import MinMaxScaler`: This line imports the `MinMaxScaler` class from scikit-learn, which is a scaler used for feature scaling.
8. **Creating an Instance of the Scaler:**
    - `ms = MinMaxScaler()`: This line creates an instance of the `MinMaxScaler` and assigns it to the variable `ms`. The scaler will be used to transform the

features in a way that scales them to a specified range, usually between 0 and 1.

9. **Scaling the Training Set:**
   - `X_train = ms.fit_transform(X_train)`: This line fits the `MinMaxScaler` to the training set (`X_train`) and then transforms the features. The `fit_transform` method computes the minimum and maximum values for each feature in the training set and scales the features accordingly.
10. **Scaling the Testing Set:**
   - `X_test = ms.transform(X_test)`: This line uses the same scaler (`ms`) to transform the testing set (`X_test`). However, it's important to note that we only use the `transform` method on the testing set, not `fit_transform`. This ensures that the scaling applied to the testing set is based on the statistics (minimum and maximum values) computed from the training set. This helps prevent data leakage by ensuring that the scaling applied to the testing set is consistent with the training set.

Purpose:

11. The purpose of feature scaling is to bring all features to the same scale, which is important for many machine learning algorithms. The `MinMaxScaler` specifically scales the features to a specified range (commonly between 0 and 1), making the data more suitable for models that are sensitive to the scale of input features, such as neural networks and support vector machines.

Standardization

```
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()

sc.fit(X_train)
X_train = sc.transform(X_train)
X_test = sc.transform(X_test)
```

1. **Importing the Scaler:**
   - `from sklearn.preprocessing import StandardScaler`: This line imports the `StandardScaler` class from scikit-learn, which is a scaler used for standardizing features.
2. **Creating an Instance of the Scaler:**

- `sc = StandardScaler()`: This line creates an instance of the `StandardScaler` and assigns it to the variable `sc`. The scaler will be used to standardize the features, meaning it transforms the data such that it has a mean of 0 and a standard deviation of 1.

3. **Fitting the Scaler to the Training Set:**
   - `sc.fit(X_train)`: This line fits the `StandardScaler` to the training set (`X_train`). During this step, the scaler computes the mean and standard deviation for each feature in the training set.
4. **Transforming the Training Set:**
   - `X_train = sc.transform(X_train)`: This line uses the fitted scaler (`sc`) to transform the features in the training set. The `transform` method standardizes the features based on the mean and standard deviation computed during the fitting step.
5. **Transforming the Testing Set:**
   - `X_test = sc.transform(X_test)`: Similarly, this line uses the same fitted scaler to transform the features in the testing set (`X_test`). This ensures that the same scaling parameters (mean and standard deviation) computed from the training set are applied to the testing set.

Purpose:

- The purpose of standardization is to ensure that the features have comparable scales, which can be crucial for certain machine learning algorithms, such as support vector machines, k-means clustering, and principal component analysis. Standardizing features helps prevent features with larger scales from dominating those with smaller scales during model training.

7. Machine Learning Model

Evaluating the different Models

```
from sklearn.linear_model import LogisticRegression
from sklearn.naive_bayes import GaussianNB
from sklearn.svm import SVC
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.tree import ExtraTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import BaggingClassifier
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.ensemble import AdaBoostClassifier
from sklearn.metrics import accuracy_score

models = {
    'Logistic Regression': LogisticRegression(),
    'Support Vector Machine': SVC(),
    'K-Nearest Neighbors': KNeighborsClassifier(),
    'Decision Tree': DecisionTreeClassifier(),
    'Random Forest': RandomForestClassifier(),
    'Bagging': BaggingClassifier(),
    'AdaBoost': AdaBoostClassifier(),
    'Gradient Boosting': GradientBoostingClassifier(),
    'Extra Trees': ExtraTreeClassifier(),
}


for name, md in models.items():
    md.fit(X_train,y_train)
    ypred = md.predict(X_test)
print(f"{name}  with accuracy : {accuracy_score(y_test,ypred)}")
```

This code defines a dictionary of various classification models from scikit-learn, trains each model on a training set (`X_train`, `y_train`), makes predictions on a testing set (`X_test`), and evaluates the accuracy of each model using the `accuracy_score` metric. Here's a breakdown of the code:

1. **Importing Models and Metrics:**

- The code starts by importing various classification models (`LogisticRegression`, `SVC`, `KNeighborsClassifier`, etc.) and the `accuracy_score` metric from scikit-learn.

2. **Creating a Dictionary of Models:**
   - `models = {...}`: This dictionary contains instances of different classification models, with each model associated with a human-readable name. The models include logistic regression, support vector machines, k-nearest neighbors, decision trees, random forests, bagging, AdaBoost, gradient boosting, and extra trees.
3. **Model Training and Evaluation:**
   - The code then iterates through each model in the `models` dictionary using a `for` loop.
     - `md.fit(X_train, y_train)`: This line fits the current model (`md`) to the training set (`X_train`, `y_train`).
     - `ypred = md.predict(X_test)`: This line uses the trained model to make predictions on the testing set (`X_test`).
     - `accuracy_score(y_test, ypred)`: This computes the accuracy of the model by comparing the predicted labels (`ypred`) with the true labels (`y_test`).
4. **Printing Model Performance:**
   - The code then prints the name of each model along with its accuracy on the testing set.

Purpose:

- The purpose of this code is to compare the performance of multiple classification models on the given dataset. Accuracy is used as the evaluation metric, which represents the proportion of correctly predicted instances.

Output:

- For each model, the code prints a message indicating the model's name and its accuracy on the testing set.

Choosing the best Suitable Model

```
rfc = RandomForestClassifier()
rfc.fit(X_train,y_train)
ypred = rfc.predict(X_test)
accuracy_score(y_test,ypred)
```

This code involves using a RandomForestClassifier from scikit-learn to train a random forest model, make predictions on a testing set, and evaluate the accuracy of the model. Here's a breakdown of each part:

1. **Creating a RandomForestClassifier:**
   - `rfc = RandomForestClassifier()`: This line creates an instance of the RandomForestClassifier. The random forest is an ensemble learning method that constructs multiple decision trees during training and outputs the mode of the classes (classification) of the individual trees as the prediction.
2. **Training the Model:**
   - `rfc.fit(X_train, y_train)`: This line fits (trains) the RandomForestClassifier on the training set (`X_train`, `y_train`). The model learns to make predictions based on the patterns present in the training data.
3. **Making Predictions:**
   - `ypred = rfc.predict(X_test)`: This line uses the trained random forest model to make predictions on the testing set (`X_test`). The resulting predictions are stored in the variable `ypred`.
4. **Evaluating Accuracy:**
   - `accuracy_score(y_test, ypred)`: This line calculates the accuracy of the model by comparing the predicted labels (`ypred`) with the true labels (`y_test`). The `accuracy_score` function from scikit-learn is used for this purpose.

Purpose:

- The purpose of this code is to train a RandomForestClassifier on a given dataset, make predictions on a separate testing set, and assess the accuracy of the model. Accuracy is a common metric used for classification tasks, representing the proportion of correctly predicted instances.

Output:

- The output of the `accuracy_score` function is the accuracy of the random forest model on the testing set.

8. Predictive Analysis

```
def recommendation(N,P,k,temperature,humidity,ph,rainfal):
    features = np.array([[N,P,k,temperature,humidity,ph,rainfal]])
    transformed_features = ms.fit_transform(features)
    transformed_features = sc.fit_transform(transformed_features)
    prediction = rfc.predict(transformed_features).reshape(1,-1)

    return prediction[0]
```

This function, named **`recommendation`**, takes several input parameters (N, P, K, temperature, humidity, pH, rainfall), transforms these features using previously applied MinMaxScaler (**`ms`**) and StandardScaler (**`sc`**), and then uses a pre-trained RandomForestClassifier (**`rfc`**) to make a crop recommendation. Here's a breakdown of each part:

1. **Input Parameters:**
   - **`N, P, K, temperature, humidity, ph, rainfall`**: These parameters represent the nutrient values (N, P, K), temperature, humidity, pH level, and rainfall, respectively. These are the features used for making a crop recommendation.
2. **Creating Features Array:**
   - **`features = np.array([[N, P, K, temperature, humidity, ph, rainfall]])`**: This line creates a NumPy array containing the input features. The features are organized as a nested list within a NumPy array to match the expected input format.
3. **Scaling Features:**
   - **`transformed_features = ms.fit_transform(features)`**: This line applies MinMaxScaler (**`ms`**) to transform the features. It scales the features to a specified range (commonly between 0 and 1).
   - **`transformed_features = sc.fit_transform(transformed_features)`**: This line applies StandardScaler (**`sc`**) to further standardize the features. It ensures that the features have a mean of 0 and a standard deviation of 1.
4. **Making Predictions:**
   - **`prediction = rfc.predict(transformed_features).reshape(1, -1)`**: This line uses the pre-trained RandomForestClassifier (**`rfc`**) to make predictions on the transformed features. The **`reshape(1, -1)`** is used to ensure that the result is a 1D array (flattened) because the **`predict`** method typically returns a 1D array for a single sample.
5. **Returning the Recommendation:**

- `return prediction[0]`: This line returns the predicted crop recommendation. The `[0]` indexing is used to extract the single element from the 1D array.

Purpose:

- The function serves the purpose of providing a crop recommendation based on the given input features. It leverages a pre-trained RandomForestClassifier that has been fitted on a dataset with similar features and crop labels.

```
N = float(input("Enter N: "))
P = float(input("Enter P: "))
K = float(input("Enter K: "))
temperature = float(input("Enter temperature: "))
humidity = float(input("Enter humidity: "))
ph = float(input("Enter pH: "))
rainfall = float(input("Enter rainfall: "))

predict = recommendation(N, P, K, temperature, humidity, ph, rainfall)

if predict[0] in crop_dict:
    crop = crop_dict[predict[0]]
    print("{} is the recommended crop for cultivation.".format(crop))
else:
    print("Sorry, we are not able to recommend a proper crop for this environment.")
```

This code is an interactive script that takes user input for specific environmental parameters (N, P, K, temperature, humidity, pH, rainfall), utilizes the `recommendation` function to predict a crop based on those inputs, and then prints the recommended crop or a message if no recommendation is available. Here's an explanation of each part:

1. **User Input:**
   - `N = float(input("Enter N: "))`: This line prompts the user to enter the value for the nitrogen content (`N`). The `input` function is used to take user input, and `float` is used to convert the input to a floating-point number.
   - Similar lines exist for other environmental parameters (P, K, temperature, humidity, pH, rainfall).
2. **Calling the `recommendation` Function:**

- `predict = recommendation(N, P, K, temperature, humidity, ph, rainfall)`: This line calls the `recommendation` function with the user-provided environmental parameters. The function returns the predicted crop label.

3. **Mapping Predicted Crop Label to Crop Name:**
   - `if predict[0] in crop_dict:`: This line checks if the predicted crop label is present in the `crop_dict` (the dictionary used for mapping crop labels to crop names).
   - `crop = crop_dict[predict[0]]`: If the predicted label is found, it retrieves the corresponding crop name from the `crop_dict`.
4. **Printing the Recommendation:**
   - `print("{} is the recommended crop for cultivation.".format(crop))`: If a recommendation is available, it prints the recommended crop name.
   - `else: print("Sorry, we are not able to recommend a proper crop for this environment.")`: If no recommendation is available, it prints a message indicating that a proper crop recommendation couldn't be made.

Purpose:

- The script aims to provide a user-friendly interface for obtaining crop recommendations based on environmental parameters. Users input values for key environmental factors, and the system uses a pre-trained model to predict a suitable crop for cultivation.

9. Calculating Yield

```
farmers_data = pd.read_csv('farmers_data.csv')
merged_data = pd.merge(crop, farmers_data, on='crop', how='inner')
```

This code involves reading data from two CSV files, `crop` and `farmers_data`, and then merging them based on a common column 'crop'. Here's a breakdown of each part:

1. **Reading CSV Files:**
    - `crop = pd.read_csv('crop.csv')`: This line reads the data from a CSV file named 'crop.csv' into a Pandas DataFrame called `crop`.
    - `farmers_data = pd.read_csv('farmers_data.csv')`: Similarly, this line reads the data from a CSV file named 'farmers_data.csv' into a Pandas DataFrame called `farmers_data`.
2. **Merging DataFrames:**
    - `merged_data = pd.merge(crop, farmers_data, on='crop', how='inner')`: This line merges the two DataFrames (`crop` and `farmers_data`) based on the common column 'crop'. The parameters used in the `merge` function are as follows:
        - `on='crop'`: Specifies the column on which the merging is based, in this case, the 'crop' column.
        - `how='inner'`: Specifies the type of merge to be performed. 'Inner' merge retains only the rows where the 'crop' column is present in both DataFrames.
3. **Resulting DataFrame:**
    - The resulting `merged_data` DataFrame contains all columns from both `crop` and `farmers_data`, with rows matched based on the 'crop' column. Each row in the merged DataFrame corresponds to a unique combination of 'crop' values present in both original DataFrames.

Purpose:

- The purpose of merging the DataFrames is to combine information from two separate datasets based on a common identifier ('crop' in this case). This can be useful when you have related information distributed across multiple datasets and you want to consolidate it into a single DataFrame for analysis.

```
X_yield = merged_data.drop(['crop_num', 'label', 'yield'], axis=1)
y_yield = merged_data['yield']

X_yield_train, X_yield_test, y_yield_train, y_yield_test =
train_test_split(X_yield, y_yield, test_size=0.2, random_state=42)

rfc_yield = RandomForestClassifier()
rfc_yield.fit(X_yield_train, y_yield_train)

yield_predictions = rfc_yield.predict(X_yield_test)
accuracy_yield = accuracy_score(y_yield_test, yield_predictions)
print(f"Accuracy for yield prediction: {accuracy_yield}")
```

This code involves preparing data for predicting crop yields using a RandomForestClassifier. It follows these steps:

1. **Creating Feature and Target Variables:**
   - `X_yield = merged_data.drop(['crop_num', 'label', 'yield'], axis=1)`: This line creates a DataFrame `X_yield` that contains the features for predicting crop yields. It drops the columns 'crop_num', 'label', and 'yield' from the `merged_data` DataFrame.
   - `y_yield = merged_data['yield']`: This line creates a Series `y_yield` that contains the target variable for predicting crop yields, which is the 'yield' column from the `merged_data` DataFrame.
2. **Splitting the Dataset:**
   - `X_yield_train, X_yield_test, y_yield_train, y_yield_test = train_test_split(X_yield, y_yield, test_size=0.2, random_state=42)`: This line uses `train_test_split` to split the data into training and testing sets. The features (`X_yield`) and target variable (`y_yield`) are divided into training and testing sets. The `random_state` parameter ensures reproducibility.
3. **Creating and Training the RandomForestClassifier:**
   - `rfc_yield = RandomForestClassifier()`: This line creates an instance of the RandomForestClassifier for predicting crop yields.
   - `rfc_yield.fit(X_yield_train, y_yield_train)`: This line fits (trains) the RandomForestClassifier on the training set (`X_yield_train`, `y_yield_train`).
4. **Making Predictions and Calculating Accuracy:**
   - `yield_predictions = rfc_yield.predict(X_yield_test)`: This line uses the trained RandomForestClassifier to make predictions on the testing set (`X_yield_test`).
   - `accuracy_yield = accuracy_score(y_yield_test, yield_predictions)`: This line calculates the accuracy of the yield predictions by comparing the predicted yields (`yield_predictions`) with the true yields (`y_yield_test`) using the `accuracy_score` metric.
5. **Printing Accuracy:**
   - `print(f"Accuracy for yield prediction: {accuracy_yield}")`: This line prints the accuracy of the yield predictions.

Purpose:

- The purpose of this code is to train a RandomForestClassifier to predict crop yields based on certain features. It evaluates the model's performance on a separate testing set and prints the accuracy of the yield predictions.

10. Calculating Credit

```
farmers_data['credit'] = farmers_data['yield_per_acre'] *
farmers_data['total_area']
farmers_data['scaled_credit'] = (farmers_data['credit'] -
farmers_data['credit'].min()) / (farmers_data['credit'].max() -
farmers_data['credit'].min())
```

This code involves creating two new columns, 'credit' and 'scaled_credit', in the **`farmers_data`** DataFrame based on calculations involving the existing columns 'yield_per_acre' and 'total_area'. Here's an explanation of each part:

1. **Calculating 'credit' Column:**
    - **`farmers_data['credit'] = farmers_data['yield_per_acre'] * farmers_data['total_area']`**: This line creates a new column 'credit' in the **`farmers_data`** DataFrame. It calculates the credit by multiplying the 'yield_per_acre' (yield per acre of land) by the 'total_area' (total area of land). The result is the estimated credit derived from the agricultural yield.
2. **Scaling 'credit' to 'scaled_credit':**
    - **`farmers_data['scaled_credit'] = (farmers_data['credit'] - farmers_data['credit'].min()) / (farmers_data['credit'].max() - farmers_data['credit'].min())`**: This line creates a new column 'scaled_credit' in the **`farmers_data`** DataFrame. It scales the 'credit' values to a normalized range between 0 and 1 using the Min-Max scaling technique.
        - **`(farmers_data['credit'] - farmers_data['credit'].min())`**: This part calculates the difference between each 'credit' value and the minimum 'credit' value in the entire column.
        - **`(farmers_data['credit'].max() - farmers_data['credit'].min())`**: This part calculates the range (difference between maximum and minimum) of the 'credit' values.
        - The result is a scaled value for each 'credit' value, ensuring that the 'scaled_credit' column has values in the range [0, 1].

Purpose:

- The 'credit' column represents an estimated credit value for each farmer based on their agricultural yield and total land area. The 'scaled_credit' column provides a normalized version of these credit values, which can be useful for certain analyses or machine learning models that benefit from standardized input features.

11. Future Scope

Website Integration: Creating a web interface for user interaction.

Importing Libraries

```
!pip install flask
!pip install pyngrok

!wget https://bin.equinox.io/c/4VmDzA7iaHb/ngrok-stable-linux-amd64.zip
!unzip ngrok-stable-linux-amd64.zip
```

Create a Flask Application

```
import getpass
authtoken = getpass.getpass("Enter your Ngrok authtoken: ")
!./ngrok authtoken $authtoken
```

```
from flask_ngrok import run_with_ngrok
from flask import Flask, render_template, request
import numpy as np
import pickle

app = Flask(__name__)
run_with_ngrok(app)

model = pickle.load(open('model.pkl', 'rb'))
sc = pickle.load(open('standscaler.pkl', 'rb'))
ms = pickle.load(open('minmaxscaler.pkl', 'rb'))

@app.route('/')
def index():
  return render_template("index.html")

@app.route("/predict", methods=['POST'])
def predict():
    try:
        return render_template('index.html', result=result)
    except Exception as e:
        return render_template('index.html', result="Error processing the request.
Please try again.")

if __name__ == "__main__":
    app.run()
```

Database Integration: Storing and retrieving data from a database.

```
pip install Flask-SQLAlchemy
pip install mysql-connector-python
```

```
from flask import Flask, render_template, request
from flask_sqlalchemy import SQLAlchemy

app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] =
'mysql://username:password@hostname/database'
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False

db = SQLAlchemy(app)

class CropData(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    N = db.Column(db.Float)
    P = db.Column(db.Float)
    K = db.Column(db.Float)
    temperature = db.Column(db.Float)
    humidity = db.Column(db.Float)
    ph = db.Column(db.Float)
    rainfall = db.Column(db.Float)
    label = db.Column(db.String(50))
    crop_num = db.Column(db.Integer)

def recommendation(N, P, K, temperature, humidity, ph, rainfall):
    features = CropData(N=N, P=P, K=K, temperature=temperature, humidity=humidity,
ph=ph, rainfall=rainfall)
    db.session.add(features)
    db.session.commit()

@app.route('/')
def home():
    return render_template('index.html')

@app.route('/predict', methods=['POST'])
def predict():
    input_data = request.form['input_data']
    return f'Prediction: {prediction}'

if __name__ == '__main__':
    db.create_all()
    app.run(debug=True)
```

```
crop = CropData.query.all()
```

```
pickle.dump(rfc, open('model.pkl', 'wb'))
pickle.dump(ms, open('minmaxscaler.pkl', 'wb'))
```

```
pickle.dump(sc, open('standscaler.pkl', 'wb'))

db.create_all()
```

Dumping Results: Saving the model and data model for future use.

```
import pickle
pickle.dump(rfc,open('model.pkl','wb'))
pickle.dump(ms,open('minmaxscaler.pkl','wb'))
pickle.dump(sc,open('standscaler.pkl','wb'))
```

12. Containerization and Deployment

Dockerization: Creating a Docker image for the application.

Steps:

```
FROM python:3.8-slim
WORKDIR /app
COPY . /app
RUN pip install --no-cache-dir -r requirements.txt
EXPOSE 80
ENV NAME World
CMD ["python", "app.py"]
```

```
Flask==2.0.1
Flask-SQLAlchemy==2.5.1
mysql-connector-python==8.0.27
pyngrok==5.0.6
scikit-learn==0.24.2
```

```
docker build -t Farmerly .
docker run -p 4000:80 Farmerly
```

```
version: '3'
services:
  web:
    build: .
    ports:
      - "4000:80"
docker-compose up
```

Cloud Hosting: Deploying the application on a cloud-based platform (AWS/Heroku).

Steps:

```
heroku container:login
docker tag Farmerly:latest registry.heroku.com/ Farmerly /web
docker push registry.heroku.com/ Farmerly /web
heroku container:release web -a Farmerly
```

13. Conclusion

Summary of Achievements

- Successful implementation of smart farming tips and technology.
- High accuracy achieved in crop recommendation and yield prediction.
- Financial assistance provided through credit calculation.

Lessons Learned

- Importance of data preprocessing and exploratory data analysis.
- Selection and evaluation of machine learning models.
- Integration of multiple components for a comprehensive solution.

Future Enhancements

- Incorporate more environmental factors for better predictions.
- Expand the web interface for user-friendly access.
- Explore advanced machine learning techniques.