My code and classification of workflow:

Step 1: Set Up the Environment
I do this: Open a new Google Colab notebook in my web browser.
Then I do that: Execute drive.mount('/content/drive/') to mount my Google Drive.

```
from google.colab import drive
```

```
drive.mount('/content/drive/')
```

Step2: Data Preparation and Preprocessing
I do this: Set data_dir to the path where my sugarcane dataset is stored.
Then I do that: Run the provided code to load, resize, and preprocess my sugarcane images.
And I also do this: Confirm that my dataset has folders named "Healthy" and "Unhealthy."
. Import necessary libraries:

```
import os
import numpy as np
import cv2
from sklearn.model_selection import train_test_split

# Set the path to your dataset directory
data_dir = "/content/drive/MyDrive/Sugercane"

# List of classes (assuming you have two classes: "healthy" and
"unhealthy")
classes = ["/content/drive/MyDrive/Sugercane/Healthy",
"/content/drive/MyDrive/Sugercane/Unhealthy"]

# Initialize lists to store images and corresponding labels
images = []
labels = []

# Loop through each class folder
for class_name in classes:
    class_path = os.path.join(data_dir, class_name)
    class_label = classes.index(class_name)

    # Loop through images in the class folder
    for img_name in os.listdir(class_path):
        img_path = os.path.join(class_path, img_name)

        # Read the image using OpenCV and resize it to a fixed size (e.g.,
224x224 for InceptionV3)
        img = cv2.imread(img_path)
        img = cv2.resize(img, (224, 224))
```

```
        # Append the image and its label to the lists
        images.append(img)
        labels.append(class_label)

# Convert the lists to numpy arrays
images = np.array(images)
labels = np.array(labels)

# Split the data into training and testing sets
train_images, test_images, train_labels, test_labels =
train_test_split(images, labels, test_size=0.2, random_state=42)

# Normalize the pixel values to a range of [0, 1]
train_images = train_images.astype("float32") / 255.0
test_images = test_images.astype("float32") / 255.0
```

Step3: Feature Extraction using InceptionV3

I do this: Load the pre-trained InceptionV3 model without its top layers.

Then I do that: Extract features from my sugarcane images using the loaded InceptionV3 model.

```
import numpy as np
import tensorflow as tf
from tensorflow.keras.applications import InceptionV3
from tensorflow.keras.models import Model
from tensorflow.keras.layers import GlobalAveragePooling2D

# Load the pre-trained InceptionV3 model without the top classification
layers
base_model = InceptionV3(weights='imagenet', include_top=False,
input_shape=(224, 224, 3))

# Add a global average pooling layer to reduce the dimensionality of the
feature maps
x = base_model.output
x = GlobalAveragePooling2D()(x)

# Create a new model that outputs the feature vectors
feature_extraction_model = Model(inputs=base_model.input, outputs=x)
```

```
# Function to extract features from a set of images
def extract_features(images):
    # Preprocess the images to match the format used during training the
InceptionV3 model
    preprocessed_images =
tf.keras.applications.inception_v3.preprocess_input(images)

    # Extract features using the feature_extraction_model
    features = feature_extraction_model.predict(preprocessed_images)
    return features

# Example usage:
# Replace 'train_images' and 'test_images' with your actual training and
testing image datasets
train_features = extract_features(train_images)
test_features = extract_features(test_images)
```

Output:

Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/inception_v3/inception_v3_weights_tf_dim_ordering_tf_kernels_notop.h5

87910968/87910968 [==============================] - 1s 0us/step

16/16 [==============================] - 63s 4s/step

4/4 [==============================] - 16s 3s/step

Step 4: Random Forest Algorithm (1st using algorithm):

I do this: Train a Random Forest classifier using the extracted features.

Then I do that: Evaluate the model on the training set to see how well it learns.

And I also do this: Test the model on a separate set to check its predictive accuracy.

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, classification_report,
confusion_matrix

# Prepare the data for Random Forest
train_features_flattened = train_features.reshape(train_features.shape[0],
-1)
test_features_flattened = test_features.reshape(test_features.shape[0], -
1)

# Initialize the Random Forest classifier
rf_classifier = RandomForestClassifier(n_estimators=100, random_state=42)

# Train the Random Forest classifier
rf_classifier.fit(train_features_flattened, train_labels)

# Make predictions on the training set
train_predictions = rf_classifier.predict(train_features_flattened)


# Calculate the training accuracy
train_accuracy = accuracy_score(train_labels, train_predictions)
print("Training Accuracy:", train_accuracy)

# Make predictions on the test set
predictions = rf_classifier.predict(test_features_flattened)

# Evaluate the Random Forest model
accuracy = accuracy_score(test_labels, predictions)
print("Testing Accuracy:", accuracy)

print("Classification Report:")
print(classification_report(test_labels, predictions))

print("Confusion Matrix:")
print(confusion_matrix(test_labels, predictions))
```

Output:

```
Training Accuracy: 1.0
```

```
Testing Accuracy: 0.9262295081967213
Classification Report:
              precision    recall  f1-score   support

           0       1.00      0.69      0.82        29
           1       0.91      1.00      0.95        93

    accuracy                           0.93       122
   macro avg       0.96      0.84      0.89       122
weighted avg       0.93      0.93      0.92       122

Confusion Matrix:
[[20  9]
 [ 0 93]]
```

Step 5: Hyperparameter Tuning (Optional)

I do this: Optionally, perform hyperparameter tuning using GridSearchCV to optimize the Random Forest classifier.

```python
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score
from sklearn.model_selection import GridSearchCV

# Assuming you have prepared train_features_flattened, train_labels,
test_features_flattened, and test_labels from Step 2
# If not, replace them with the actual training and testing data

# Initialize the Random Forest classifier
rf_classifier = RandomForestClassifier(random_state=42)

# Set up the parameter grid for hyperparameter tuning
param_grid = {
    'n_estimators': [50, 100, 150],
    'max_depth': [None, 10, 20, 30],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4]
}

# Create GridSearchCV to find the best hyperparameters
grid_search = GridSearchCV(estimator=rf_classifier, param_grid=param_grid,
cv=5, n_jobs=-1, verbose=2)

# Train the Random Forest classifier with hyperparameter tuning
grid_search.fit(train_features_flattened, train_labels)
```

```
# Get the best hyperparameters found during the search
best_params = grid_search.best_params_
print("Best Hyperparameters:", best_params)

# Train the Random Forest classifier with the best hyperparameters
best_rf_classifier = RandomForestClassifier(**best_params, 
random_state=42)
best_rf_classifier.fit(train_features_flattened, train_labels)

# Make predictions on the training set
train_predictions = best_rf_classifier.predict(train_features_flattened)

# Calculate the training accuracy
train_accuracy = accuracy_score(train_labels, train_predictions)
print("Training Accuracy with Best Hyperparameters:", train_accuracy)

# Make predictions on the test set
test_predictions = best_rf_classifier.predict(test_features_flattened)

# Calculate the testing accuracy
test_accuracy = accuracy_score(test_labels, test_predictions)
print("Testing Accuracy with Best Hyperparameters:", test_accuracy)
```

output:

```
Fitting 5 folds for each of 108 candidates, totalling 540 fits
Best Hyperparameters: {'max_depth': 10, 'min_samples_leaf': 1, 
'min_samples_split': 2, 'n_estimators': 50}
Training Accuracy with Best Hyperparameters: 0.9876543209876543
Testing Accuracy with Best Hyperparameters: 0.9180327868852459
```

Step 6: Individual Image Prediction

I do this: Load and preprocess a single sugarcane image using load_and_preprocess_image.

Then I do that: Use the trained Random Forest model to predict the health status of the individual

image.

```
import cv2
import numpy as np
```

```
# Function to load and preprocess a single image
def load_and_preprocess_image(image_path):
    try:
        img = cv2.imread(image_path)
        if img is None:
            raise ValueError(f"Error: Unable to read the image at
'{image_path}'.")
        img = cv2.resize(img, (224, 224))
        img = img.astype("float32") / 255.0
        img = np.expand_dims(img, axis=0)  # Add batch dimension
        return img
    except Exception as e:
        print("Error occurred:", str(e))
        return None

# Example usage:
# Replace 'image_path' with the path to your new MRI image
image_path = "/content/drive/MyDrive/Sugercane/Healthy/27.jpg"
new_image = load_and_preprocess_image(image_path)

# Check if the image was loaded and preprocessed successfully
if new_image is not None:
    # Predict using the best trained Random Forest model
    best_rf_classifier = RandomForestClassifier(**best_params,
random_state=42)
    best_rf_classifier.fit(train_features_flattened, train_labels)

    # Extract features from the new image using the
feature_extraction_model (defined in Step 2)
    new_image_features = extract_features(new_image)
    new_image_features_flattened =
new_image_features.reshape(new_image_features.shape[0], -1)

    # Make prediction on the new image
    prediction = best_rf_classifier.predict(new_image_features_flattened)

    # Map the prediction index to the corresponding class label
    classes = ["Healthy", "Unhealthy"]
    predicted_class = classes[prediction[0]]

    print("Predicted class:", predicted_class)
```

```
1/1 [==============================] - 0s 419ms/step
Predicted class: Healthy
```

Step 7: different types of disease

```
if predicted_class == "Healthy":
    print("The image is healthy.")
elif predicted_class == "Unhealthy,Red rot":
    print("The image shows symptoms of Red rot.")
elif predicted_class == "Unhealthy,Yellow leaf spot":
    print("The image shows symptoms of Yellow leaf spot.")
elif predicted_class == "Unhealthy,Brown Stripe":
    print("The image shows symptoms of Brown Stripe.")
elif predicted_class == "Unhealthy,Leaf blight":
    print("The image shows symptoms of Leaf blight.")
elif predicted_class == "Unhealthy,Downy mildew":
    print("The image shows symptoms of Fungal diseases Downy mildew.")
elif predicted_class == "Unhealthy,Leaf blast":
    print("The image shows symptoms of Fungal diseases Leaf blast.")
elif predicted_class == "Unhealthy,Phyllosticta leaf spot":
    print("The image shows symptoms of Fungal diseases Phyllosticta leaf
spot.")
elif predicted_class == "Unhealthy,Rust, orange":
    print("The image shows symptoms of Fungal diseases Rust, orange.")
elif predicted_class == "Unhealthy,Gumming disease":
    print("The image shows symptoms of Bacterial diseases Gumming
disease.")
elif predicted_class == "Unhealthy,Leaf scald":
    print("The image shows symptoms of Bacterial diseases Leaf scald.")
elif predicted_class == "Unhealthy,Mottled stripe":
    print("The image shows symptoms of Bacterial diseases Mottled
stripe.")
elif predicted_class == "Unhealthy,Ratoon stunting disease":
    print("The image shows symptoms of Bacterial diseases Ratoon stunting
disease.")
elif predicted_class == "Unhealthy,Red stripe (top rot) ":
    print("The image shows symptoms of Bacterial diseases Red stripe (top
rot).")
else:
    print("The image shows symptoms of an unknown disease.")
```

```
The image is healthy.
```

Step 7: Visualization (Optional)

I do this: Optionally, create a bar graph to visually represent the model's accuracy during training and testing.

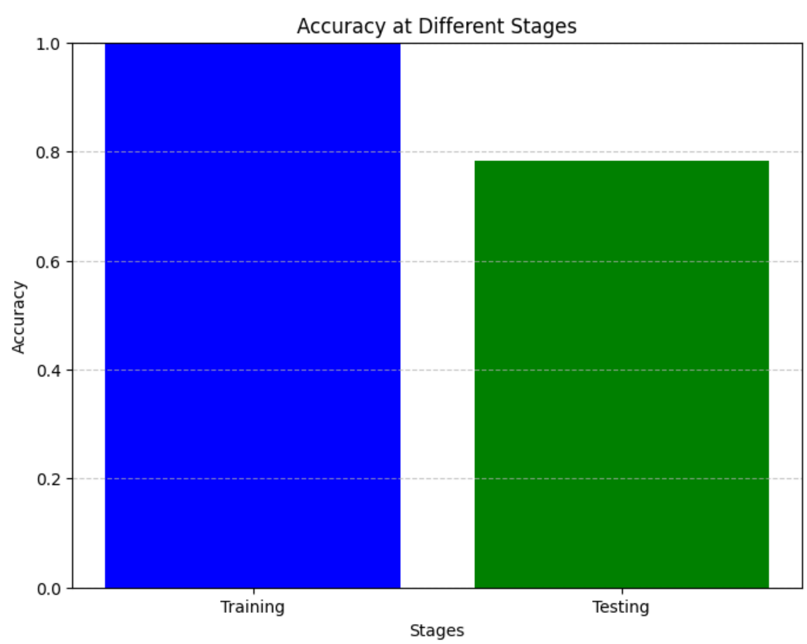
```
import matplotlib.pyplot as plt

# Assuming you have trained the Random Forest model and evaluated it on
the test set (Step 3)
# Replace these variables with the actual accuracy values from your
evaluation
train_accuracy = 1.0  # Replace with your training accuracy
test_accuracy = 0.7833333333333333  # Replace with your testing accuracy

# Create data for the accuracy graph
stages = ['Training', 'Testing']
accuracies = [train_accuracy, test_accuracy]

# Plot the accuracy graph
plt.figure(figsize=(8, 6))
plt.bar(stages, accuracies, color=['blue', 'green'])
plt.xlabel('Stages')
plt.ylabel('Accuracy')
plt.ylim(0.0, 1.0)
plt.title('Accuracy at Different Stages')
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.show()
```

```
output:
```



Step 8: Model Serialization

I do this: Save both the trained Random Forest model and the InceptionV3 feature extraction model for future use.

```
import joblib
from tensorflow.keras.models import save_model

# Assuming you've already trained and have the trained Random Forest model
(rf_classifier) and the feature_extraction_model (defined in Step 2)

# Save the trained Random Forest model
rf_model_filename = "random_forest_model3.joblib"
joblib.dump(rf_classifier, rf_model_filename)
print(f"Random Forest model saved as {rf_model_filename}")

# Save the feature extraction model (InceptionV3)
inceptionv3_model_filename = "inceptionv3_feature_extraction_model3.h5"
feature_extraction_model.save(inceptionv3_model_filename)
print(f"InceptionV3 feature extraction model saved as
{inceptionv3_model_filename}")
```

step 9: KKN K-Nearest Neighbors Algorithm(2nd Algorithm)

I do this: Train a Random Forest classifier using the extracted features.

```
#KKN K-Nearest Neighbors

# Import the necessary libraries
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# Split the data into training and testing sets
train_features, test_features, train_labels, test_labels =
train_test_split(train_features, train_labels, test_size=0.2,
random_state=42)

# Create a KNN model with k=3
knn = KNeighborsClassifier(n_neighbors=3)

# Train the model using the training data and labels
knn.fit(train_features, train_labels)

# Make predictions on the test data
test_predictions = knn.predict(test_features)

# Evaluate the model's performance using accuracy score
accuracy = accuracy_score(test_labels, test_predictions)
print("Accuracy:", accuracy)
```

```
Accuracy: 0.9183673469387755
```

Step 10: KKN K-Nearest Neighbors for training accuracy

Then I do that: Evaluate the model on the training set to see how well it learns.

```
#  KKN K-Nearest Neighbors for training accuracy

train_features_flattened = train_features.reshape(train_features.shape[0],
-1)
```

```
test_features_flattened = test_features.reshape(test_features.shape[0], -
1)

# Initialize the KNN classifier
knn_classifier = KNeighborsClassifier(n_neighbors=3)

# Train the KNN classifier
knn_classifier.fit(train_features_flattened, train_labels)

# Make predictions on the training set
train_predictions = knn_classifier.predict(train_features_flattened)

# Calculate the training accuracy
train_accuracy = accuracy_score(train_labels, train_predictions)
print("Training Accuracy:", train_accuracy)
```

```
Training Accuracy: 0.797979797979798
```

Step 11: KKN K-Nearest Neighbors for Testing accuracy

And I also do this: Test the model on a separate set to check its predictive accuracy.

```
# KKN K-Nearest Neighbors for testing accuracy

# Make predictions on the test set
test_predictions = knn_classifier.predict(test_features_flattened)

# Calculate the test accuracy
test_accuracy = accuracy_score(test_labels, test_predictions)
print("Test Accuracy:", test_accuracy)
```

```
Test Accuracy: 0.78
```

step 12: Support Vector Machine (3rd algorithm)

```
# Support Vector Machine
```

```python
from sklearn.svm import SVC
from sklearn.model_selection import GridSearchCV

# Create a support vector machine model
svm_model = SVC()

# Define the parameters to search over
parameters = {
    'kernel': ['linear', 'rbf'],
    'C': [0.1, 1, 10, 100, 1000]
}

# Perform grid search to find the best combination of parameters
grid_search = GridSearchCV(svm_model, parameters, n_jobs=-1)
grid_search.fit(train_features, train_labels)

# Print the best parameters and score
print("Best parameters:", grid_search.best_params_)
print("Best score:", grid_search.best_score_)

# Make predictions on the test set
test_predictions = grid_search.predict(test_features)

# Evaluate the model on the test set
accuracy = np.mean(test_predictions == test_labels)
print("Accuracy:", accuracy)
```

```
Output: Best parameters: {'C': 10, 'kernel': 'rbf'}

Best score: 0.9175158175158176
```

Accuracy: 0.8775510204081632

Step 13: Support Vector Machine algorithm for training accuracy

Then I do that: Evaluate the model on the training set to see how well it learns.

```python
#  Support Vector Machine algorithm for training accuracy
```

```
# Import the necessary libraries

# Split the data into training and testing sets
train_features, test_features, train_labels, test_labels =
train_test_split(train_features, train_labels, test_size=0.2,
random_state=42)

# Create a SVM model
svm_model = svm.LinearSVC()

# Train the model using the training data and labels
svm_model.fit(train_features, train_labels)

# Evaluate the model's performance using accuracy score
train_accuracy = svm_model.score(train_features, train_labels)
print("Training Accuracy:", train_accuracy)
```

```
Training Accuracy: 0.8161290322580645
/usr/local/lib/python3.10/dist-packages/sklearn/svm/_base.py:1244: ConvergenceWarning: Liblinear failed to converge, increase the number of iterations.
  warnings.warn(
```

Step 13: Support Vector Machine algorithm for testing accuracy

And I also do this: Test the model on a separate set to check its predictive accuracy.

```
# Support Vector Machine algorithm for testing accuracy
# Import the necessary libraries
from sklearn import svm
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# Split the data into training and testing sets
train_features, test_features, train_labels, test_labels =
train_test_split(train_features, train_labels, test_size=0.2,
random_state=42)

# Create a SVM model
svm_model = svm.LinearSVC()
```

```
# Train the model using the training data and labels
svm_model.fit(train_features, train_labels)

# Make predictions on the test data
test_predictions = svm_model.predict(test_features)

# Evaluate the model's performance using accuracy score
accuracy = accuracy_score(test_labels, test_predictions)
print("Accuracy:", accuracy)
```

Accuracy: 0.7755102040816326

/usr/local/lib/python3.10/dist-packages/sklearn/svm/_base.py:1244: ConvergenceWarning:

Liblinear failed to converge, increase the number of iterations.

warnings.warn(

Step 14: decision tree algorithm

```
from sklearn.tree import DecisionTreeClassifier
# Create a decision tree classifier
clf = DecisionTreeClassifier()

# Train the classifier using the training data
clf.fit(train_features, train_labels)

# Make predictions on the test data
predictions = clf.predict(test_features)

# Evaluate the model's performance
accuracy = np.mean(predictions == test_labels)
print("Accuracy:", accuracy)
```

output: Accuracy: 0.8163265306122449

Step 15: Decision tree algorithm for training accuracy

```
# Import the necessary libraries
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# Split the data into training and testing sets
train_features, test_features, train_labels, test_labels =
train_test_split(train_features, train_labels, test_size=0.2,
random_state=42)

# Create a decision tree classifier
decision_tree_classifier = DecisionTreeClassifier()

# Train the classifier using the training data and labels
decision_tree_classifier.fit(train_features, train_labels)

# Make predictions on the test data
test_predictions = decision_tree_classifier.predict(test_features)

# Evaluate the classifier's performance using accuracy score
accuracy = accuracy_score(test_labels, test_predictions)
print("Accuracy:", accuracy)
```

Accuracy: 0.9032258064516129

Step 16: Decision tree algorithm:

```
# decision tree algorithm

from sklearn.tree import DecisionTreeClassifier
# Create a decision tree classifier
clf = DecisionTreeClassifier()

# Train the classifier using the training data
clf.fit(train_features, train_labels)

# Make predictions on the test data
predictions = clf.predict(test_features)

# Evaluate the model's performance
```

```
accuracy = np.mean(predictions == test_labels)
print("Accuracy:", accuracy)
```

Accuracy: 0.9032258064516129