Python Scripts to process hyperspectral images

Montpellier, France, 31/03/2025

Yassine JANATI IDRISSI, CIRAD, Montpellier, France
Karima MEGHAR, CIRAD, Montpellier, France

CC BY NC SA

This report has been written in the framework of the RTB Breeding project, Quality-component (under CIRAD coordination), as a continuation of work initiated under the RTBfoods project.

To be cited as:

Yassine JANATI IDRISSI and Karima MEGHAR (2025). *Python Scripts to process hyperspectral images.* Montpellier, France: RTB Breeding, Tool, 6 p. https://doi.org/10.18167/agritrop/00872

Ethics: The activities, which led to the production of this document, were assessed and approved by the CIRAD Ethics Committee (H2020 ethics self-assessment procedure). When relevant, samples were prepared according to good hygiene and manufacturing practices. When external participants were involved in an activity, they were priorly informed about the objective of the activity and explained that their participation was entirely voluntary, that they could stop the interview at any point and that their responses would be anonymous and securely stored by the research team for research purposes.

Acknowledgments: This work was supported by the RTB Breeding project, through a sub-grant from the International Potato Center (CIP) to the French Agricultural Research Centre for International Development (CIRAD), Montpellier, France, incorporated in the grant agreement INV-041105 between CIP and the Bill & Melinda Gates Foundation (BMGF).

Image cover page © CIRAD.

This document has been reviewed by	
Final validation by	
Eglantine FAUVELLE (CIRAD)	31/08/2025

TABLE OF CONTENTS

Key Words: Hyperspectral imaging, Spectra, Classification model, Random forest

Part 1: Scripts used to process hyperspectral images

Importing libraries

```python
import os
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import cv2
from spectral.io import envi
```

Importing images

```python
folder_path = r"path to hyperspectral images (.raw and .hdr)"
## Image dimensions
# x : x dimension (if it differs from one image to another, take the min)
# y : y dimension (all our images are 640)
# z : z dimension, number of spectral variables (in all our images this is 244)
target_shape = (x, y, z)

# List for storing imported data
data_list = []

# Loop to browse files in folder
for file_name in os.listdir(folder_path):
    # Checks if the file is an .hdr file
    if file_name.endswith(".hdr"):
        # Recovers base name without extension
        base_name = file_name[:-4]  # Remove ".hdr"

        # Correspondence with .raw file
        raw_file = base_name + ".raw"
        raw_path = os.path.join(folder_path, raw_file)
        hdr_path = os.path.join(folder_path, file_name)

        # Checks whether the corresponding .raw file exists
        if os.path.exists(raw_path):
            try:
                # Load hyperspectral data
                data = envi.open(hdr_path, raw_path)
                data_nparr = np.array(data.load(), dtype=np.float16)

                if data_nparr.shape != target_shape:
                    print(f"Original dimensions of {file_name} : {data_nparr.shape}")

                    if data_nparr.shape[0] > target_shape[0]:
                        data_nparr = data_nparr[:target_shape[0], :, :]
                        print(f"Image {file_name} resized to {data_nparr.shape}")

                    elif data_nparr.shape[0] < target_shape[0]:
                        print(f"Error : {file_name} is too small to fit.")
                        continue  # Go to next image

                # Adds adjusted data to the list
                data_list.append(data_nparr)
                print(f"Successfully imported : {file_name} & {raw_file}")

            except Exception as e:
                print(f"Error when importing {file_name} & {raw_file} : {e}")
        else:
            print(f"Missing RAW file for : {file_name}")
```

Reflectance correction

1. Importing white and black references

```python
dark_ref = envi.open('black reference .hdr path', 'black reference .raw path')
white_ref = envi.open('blache reference .hdr path', 'blache reference .raw path')
white_nparr = np.array(white_ref.load(), dtype=np.float16)
dark_nparr = np.array(dark_ref.load(), dtype=np.float16)
```

```python
# Resizing references
example_shape = data_list[0].shape

white_nparr_resized = np.zeros(example_shape, dtype=white_nparr.dtype)
for band in range(white_nparr.shape[2]):
    white_nparr_resized[:, :, band] = cv2.resize(white_nparr[:, :, band],(example_shape[1], example_shape[0]),interpolation=cv2.INTER_NEAREST)

dark_nparr_resized = np.zeros(example_shape, dtype=dark_nparr.dtype)
for band in range(dark_nparr.shape[2]):
    dark_nparr_resized[:, :, band] = cv2.resize(dark_nparr[:, :, band],(example_shape[1], example_shape[0]),interpolation=cv2.INTER_NEAREST)
```

2. Apply correction to each image in data_list

```python
corrected_images = []  # List for storing corrected images

# Loop to browse each image in data_list
for i, data_nparr in enumerate(data_list):
    # checking dimensions
    if data_nparr.shape != white_nparr_resized.shape:
        print(f"Error: Incompatible image dimensions {i}.")
        continue

    # Correction
    corrected_nparr = np.divide(
        np.subtract(data_nparr, dark_nparr_resized),
        np.subtract(white_nparr_resized, dark_nparr_resized)
    )
    corrected_nparr = np.clip(corrected_nparr, 0, 1)
    corrected_images.append(corrected_nparr)
```

Binarization, creation of a binary masks

```python
binary_masks = []  # List for storing binary masks
thresholds = []  # List for storing Otsu thresholds

# Define a small core so that the effect is not intense
kernel = np.ones((3, 3), np.uint8)  # 3x3 core
# Loop to scroll through each corrected image
for i, corrected_image in enumerate(corrected_images):
    # Select band 41 (index 40 as NumPy uses indices starting from 0)
    gray_image = corrected_image[:, :, 40]

    # Normalize image (make sure values are between 0 and 255 for cv2)
    gray_image_normalized = (gray_image * 255).astype(np.uint8)

    # Apply Otsu's binarization method
    otsu_threshold, binary_mask = cv2.threshold(gray_image_normalized, 0, 255, cv2.THRESH_BINARY + cv2.THRESH_OTSU)

    # Apply erosion (light effect so as not to be too intense)
    binary_mask_eroded = cv2.erode(binary_mask, kernel, iterations=2)

    # Apply dilatation (light effect to restore shapes)
    binary_mask_final = cv2.dilate(binary_mask_eroded, kernel, iterations=2)

    # Add binary mask to list
    binary_masks.append(binary_mask_final)
    thresholds.append(otsu_threshold)
```

Application of binary masks, and creation of segmented hyperspectral cubes

```python
masked_cubes = []  # List for storing hidden hyperspectral cubes

# Browse each corrected image and its bitmask
for i, (corrected_image, binary_mask) in enumerate(zip(corrected_images, binary_masks)):
    cube_images = []  # Temporary list for masked bands in this image

    # Apply the binary mask to all spectral bands
    for band_idx in range(corrected_image.shape[2]):
        # Extract the current spectral band
        current_band = corrected_image[:, :, band_idx]

        # Apply binary mask (black mask zones set pixels to 0)
        masked_band = np.where(binary_mask == 255, current_band, 0)

        # Add hidden band to temporary list
        cube_images.append(masked_band)

    # Convert the list of masked images into a hyperspectral cube
    cube_hyperspectral = np.stack(cube_images, axis=-1)
    masked_cubes.append(cube_hyperspectral)  # Add hidden cube to main list
```

Extraction of average spectra

```python
# Define the boundaries of the three regions in the images
regions = [{"ymin": , "ymax": , "label": "p"},  # Proximal
           {"ymin": , "ymax": , "label": "m"},  # Central
           {"ymin": , "ymax": , "label": "d"},  # Distal
]

# List of labels for the DataFrame to be trained between [], (If you know or prefer indexes for the samples, to train them, otherwise without labels will be from 1 sample to n sample)
labels = []

# Initialiser une structure pour stocker les spectres moyens de toutes les images
all_spectra = []

# Loop through each hyperspectral image
for idx, cube_hyperspectral in enumerate(masked_cubes):
    image_spectra = {}  # Dictionary to store the spectra of this image

    # Calculate the average spectrum for each region
    for region in regions:
        ymin, ymax, label = region["ymin"], region["ymax"], region["label"]

        # Extract sub-region from hyperspectral image
        subregion = cube_hyperspectral[ymin:ymax, :, :]

        # Create a mask to exclude values equal to 0
        mask = (subregion > 0)
        # Calculate the average spectrum
        spectre_moyen = np.zeros(cube_hyperspectral.shape[2])
        for band_idx in range(cube_hyperspectral.shape[2]):
            spectre_moyen[band_idx] = np.mean(subregion[:, :, band_idx][mask[:, :, band_idx]])

        # Store the average spectrum in the dictionary
        image_spectra[label] = spectre_moyen

    # Add the average spectra of this image to the main list
    all_spectra.append(image_spectra)

# Structuring data for the DataFrame
data = []
for idx, image_spectra in enumerate(all_spectra):
    for region_label, spectre in image_spectra.items():
        # Create a complete label based on image and region
        full_label = f'_G{idx+1:03d}FR{1 if idx < 6 else 2}{region_label}'
        data.append(spectre)

# Creating the DataFrame
Spectra = pd.DataFrame(data, index=labels)
```

Saving mean spectra

```python
file_path = r"path to file to save.xlsx"

# Create folder if none exists
os.makedirs(os.path.dirname(file_path), exist_ok=True)

# Save DataFrame in Excel
Spectra.to_excel(file_path, index=True)
```

Part 2: Scripts used to develop the classification model
(from importing data to saving the classification model)

Importing libraries

```python
import os
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from imblearn.over_sampling import SMOTE
import seaborn as sns
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, confusion_matrix, f1_score, matthews_corrcoef, precision_score
from sklearn.model_selection import train_test_split
import joblib
```

Data import

```python
# Set file path
file_path = r"path to template/12_dataset-HSI-WAB-boiled-cassava-CIAT-CIRAD-may2024-v250331.xlsx"

# Load the "Spectra" sheet and extract spectral data
df_spectra = pd.read_excel(file_path, sheet_name="Spectra", header=0, index_col=0)

# Select spectral data columns (index = 14 as Python starts at 0)
df_spectra = df_spectra.iloc[:, 14:]

# Load the "Laboratory data" sheet and extract response Y by name
df_Y = pd.read_excel(file_path, sheet_name="Laboratory data", header=0, index_col=0)
# Select "WAB (%) at 30 min" column
df_Y = df_Y[["WAB (%) at 30 min"]]
```

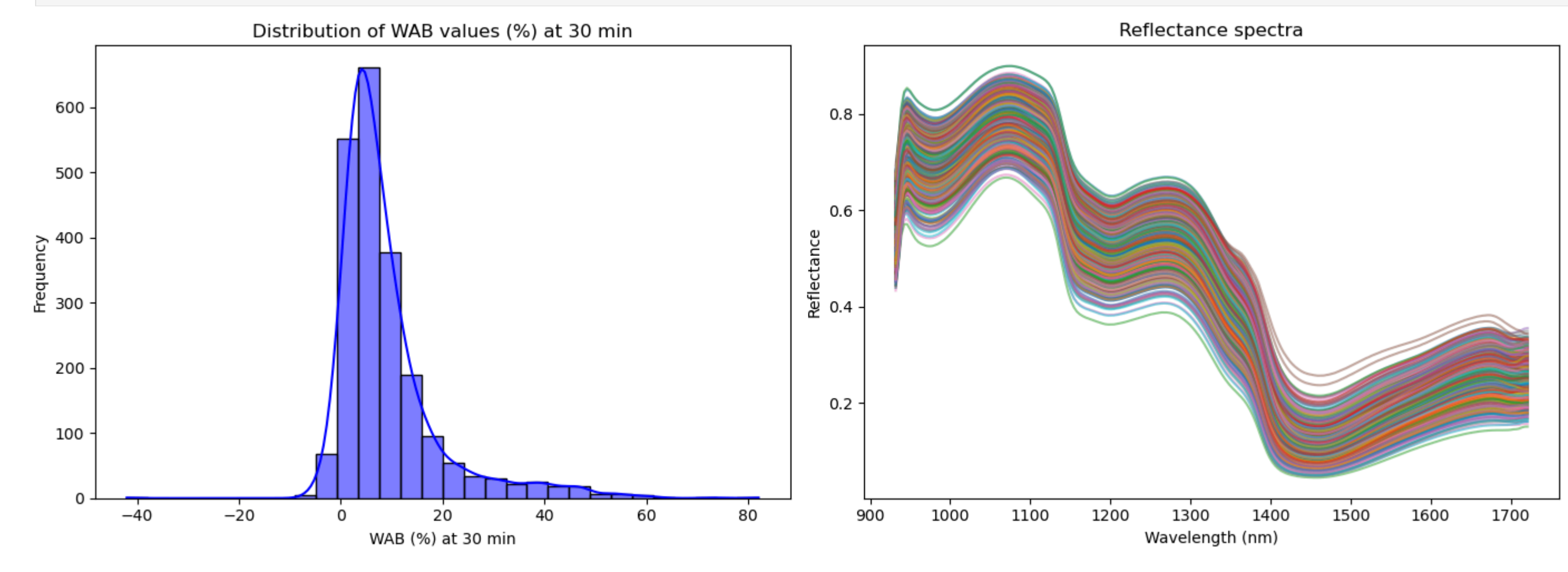
Data visualization (spectra and histogram of WAB value distribution)

```python
# Create figure with 2 side-by-side subplots
fig, axes = plt.subplots(1, 2, figsize=(14, 5))

# Histogram of WAB values
sns.histplot(df_Y["WAB (%) at 30 min"], bins=30, kde=True, ax=axes[0], color="blue")
axes[0].set_title("Distribution of WAB values (%) at 30 min")
axes[0].set_xlabel("WAB (%) at 30 min")
axes[0].set_ylabel("Frequency")

# Reflectance spectra
axes[1].plot(df_spectra.columns.astype(float), df_spectra.T, alpha=0.5)
axes[1].set_title("Reflectance spectra")
axes[1].set_xlabel("Wavelength (nm)")
axes[1].set_ylabel("Reflectance")

# Ajuster l'espacement entre les graphiques
plt.tight_layout()
plt.show()
```




```python
# Descriptive statistics
statistiques = df_Y["WAB (%) at 30 min"].describe()
print(statistiques)
```

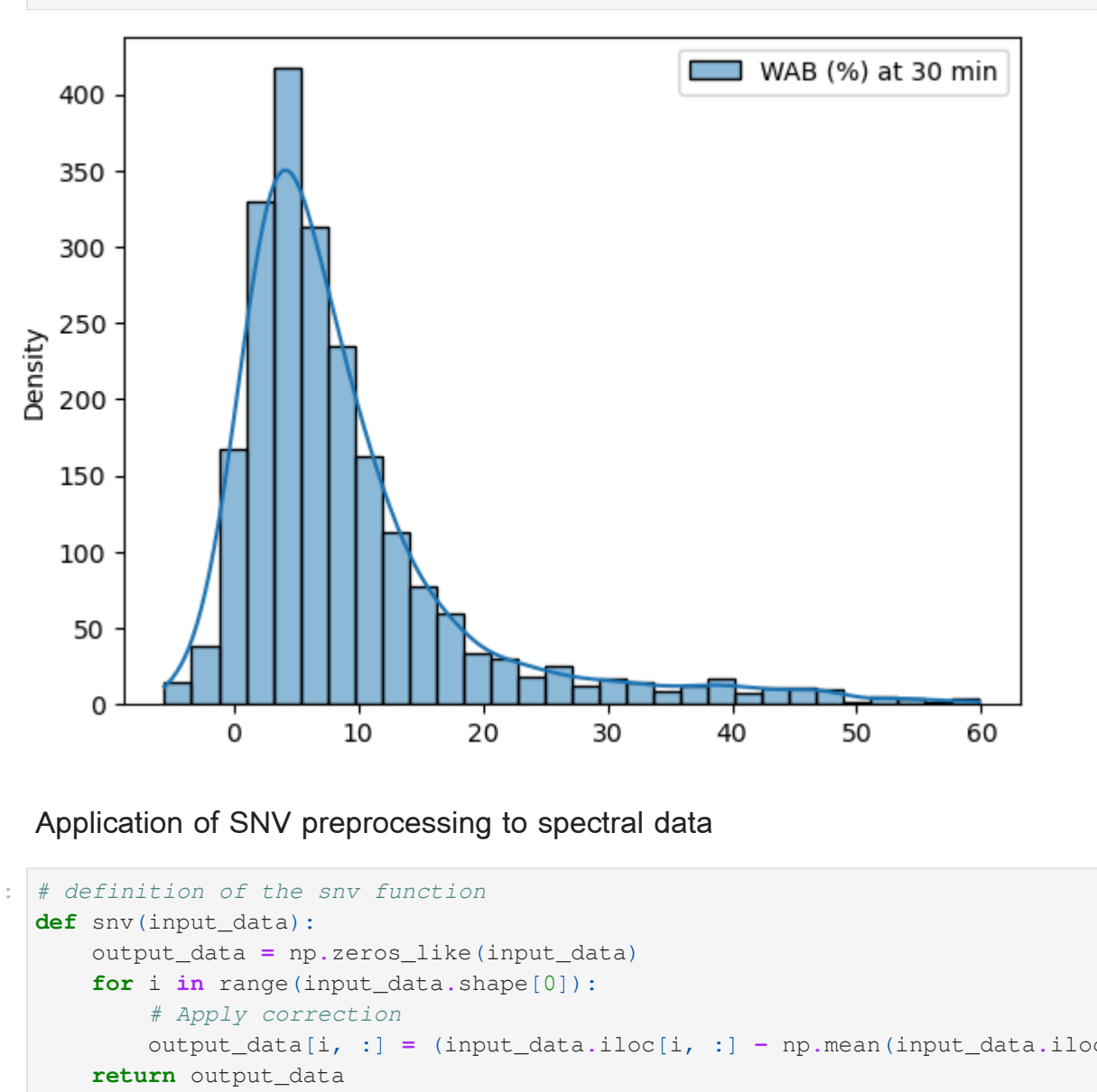
```
count    2175.000000
mean        9.103202
std        10.219647
min       -42.031464
25%         3.157459
50%         6.188119
75%        11.221271
max        82.064976
Name: WAB (%) at 30 min, dtype: float64
```

outliers elimination

```python
filtered_indexes = df_Y[(df_Y["WAB (%) at 30 min"] >= -20) & (df_Y["WAB (%) at 30 min"] <= 60)].index

# Apply filtering to both DataFrames
Y_df_filtered = df_Y.loc[filtered_indexes]
df_filtered = df_spectra.loc[filtered_indexes]

# Display of distribution after filtering
sns.histplot(Y_df_filtered, bins=30, kde=True)
plt.ylabel("Density")
plt.show()
```



Application of SNV preprocessing to spectral data

```python
# definition of the snv function
def snv(input_data):
    output_data = np.zeros_like(input_data)
    for i in range(input_data.shape[0]):
        # Apply correction
        output_data[i, :] = (input_data.iloc[i, :] - np.mean(input_data.iloc[i, :])) / np.std(input_data.iloc[i, :])
    return output_data
```

```python
df_snv = snv(df_filtered)
```

Define target variable (class)

```python
Y_df_filtered['Quality_Class'] = np.where(Y_df_filtered["WAB (%) at 30 min"] < 12, 0, 1)
```

data oversampling with the SMOTE method

```python
# Define explanatory variables (X) and target (y)
X = df_snv
y = Y_df_filtered['Quality_Class'].values  # Classes (0 or 1)

smote = SMOTE(random_state=42) # to fix the random effect
X_resampled, y_resampled = smote.fit_resample(X, y)
```

Split data into train/test (80% training, 20% test)

```python
X_train, X_test, y_train, y_test = train_test_split(X_resampled, y_resampled, test_size=0.2, random_state=42, stratify=y_resampled)
```

Randomforest Classifier model training

```python
RFc = RandomForestClassifier(n_estimators = 200, max_depth = 30, min_samples_split = 10, min_samples_leaf = 4, max_features = 'sqrt', random_state=42)
RFc.fit(X_train, y_train)
```

```
RandomForestClassifier
RandomForestClassifier(max_depth=30, min_samples_leaf=4, min_samples_split=10,
                       n_estimators=200, random_state=42)
```

Model performance parameters

Function for calculating specificity and sensitivity

```python
def specificity_sensitivity(y_true, y_pred):
    tn, fp, fn, tp = confusion_matrix(y_true, y_pred).ravel()
    specificity = tn / (tn + fp) if (tn + fp) != 0 else 0
    sensitivity = tp / (tp + fn) if (tp + fn) != 0 else 0
    return specificity * 100, sensitivity * 100
```

```python
# Predictions on training and test sets
y_train_pred = RFc.predict(X_train)
y_test_pred = RFc.predict(X_test)
# Calculating metrics for the train set
accuracy_train = accuracy_score(y_train, y_train_pred) * 100
specificity_train, sensitivity_train = specificity_sensitivity(y_train, y_train_pred)
precision_train = precision_score(y_train, y_train_pred) * 100
f1_train = f1_score(y_train, y_train_pred) * 100
mcc_train = matthews_corrcoef(y_train, y_train_pred) * 100
# Calculating metrics for the test set
accuracy_test = accuracy_score(y_test, y_test_pred) * 100
specificity_test, sensitivity_test = specificity_sensitivity(y_test, y_test_pred)
precision_test = precision_score(y_test, y_test_pred) * 100
f1_test = f1_score(y_test, y_test_pred) * 100
mcc_test = matthews_corrcoef(y_test, y_test_pred) * 100
# Metrics table creation
metrics_df = pd.DataFrame({
    "Metrics": ["Accuracy", "Specificity", "Sensitivity", "Precision", "F1-Score", "MCC"],
    "Train (%)": [accuracy_train, specificity_train, sensitivity_train, precision_train, f1_train, mcc_train],
    "Test (%)": [accuracy_test, specificity_test, sensitivity_test, precision_test, f1_test, mcc_test]})

# Round to 2 decimal places
metrics_df = metrics_df.round(2)

# Final table display
print("\n🔹 Performance summary table :")
print(metrics_df)
```

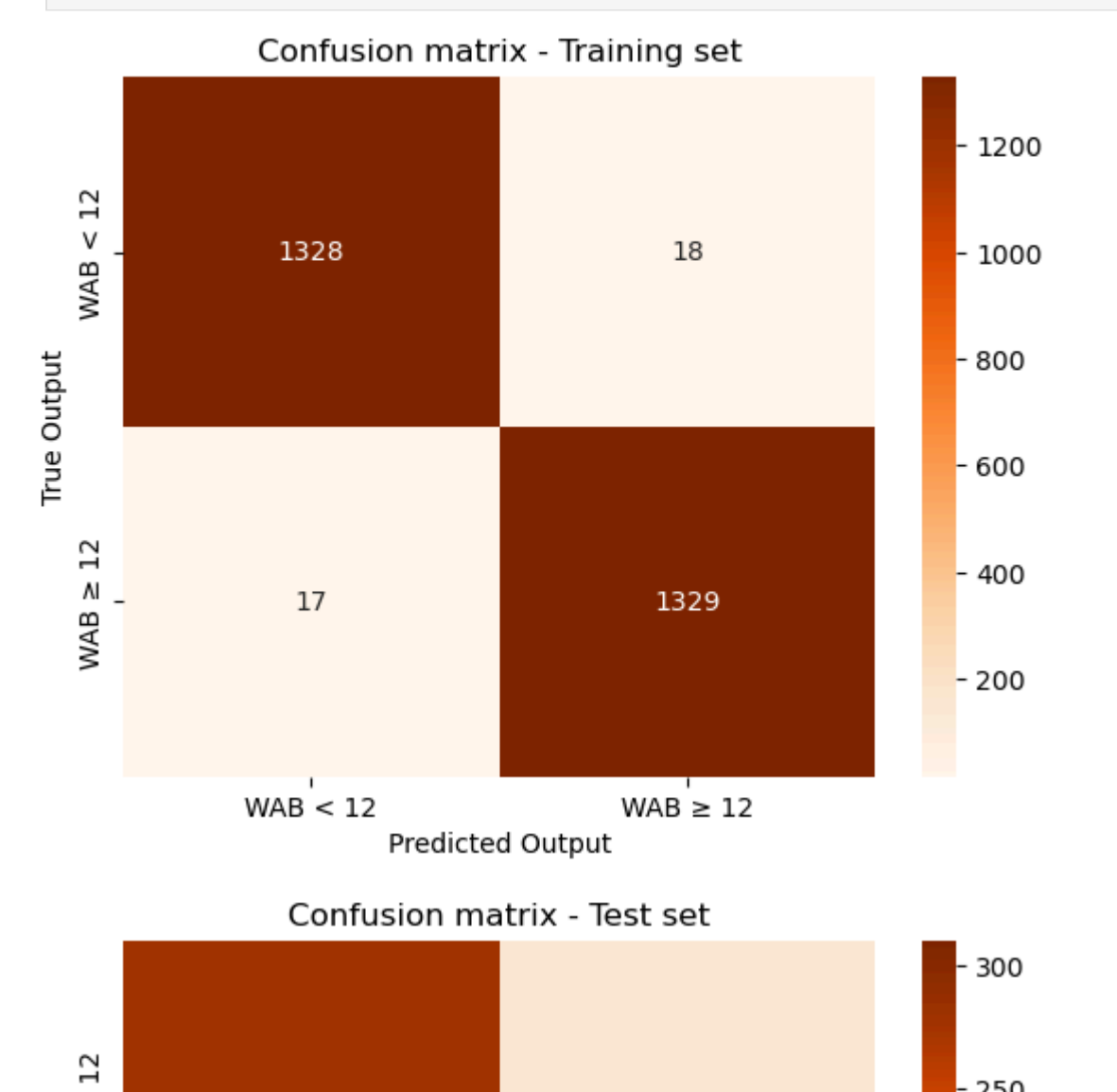
```
🔹 Performance summary table :
       Metrics  Train (%)  Test (%)
0     Accuracy      98.70     87.54
1  Specificity      98.66     83.09
2  Sensitivity      98.74     91.99
3    Precision      98.66     84.47
4     F1-Score      98.70     88.07
5          MCC      97.40     75.37
```

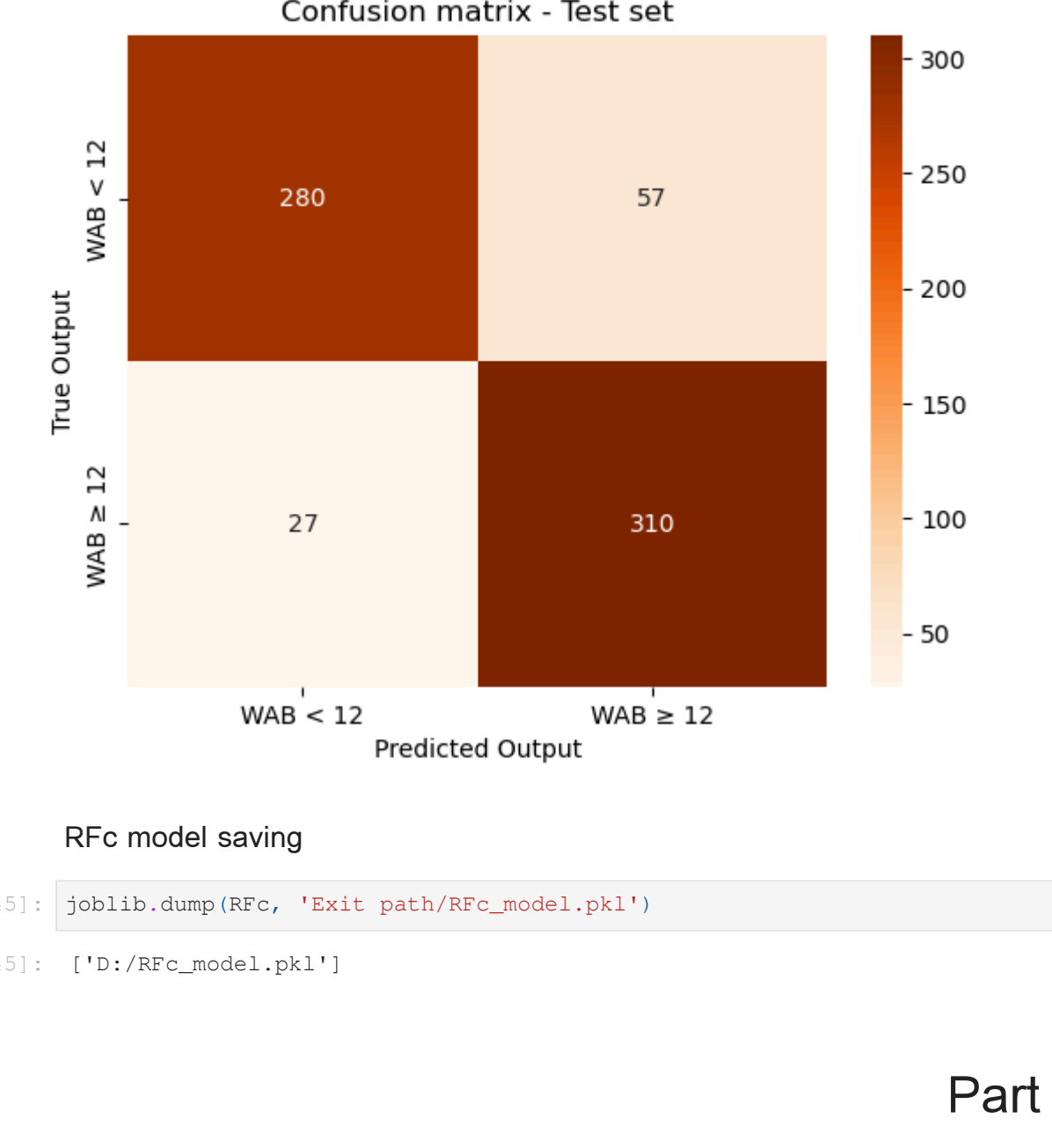
```python
conf_matrix_train = confusion_matrix(y_train, y_train_pred)
conf_matrix_test = confusion_matrix(y_test, y_test_pred)
# Function to display confusion matrices with seaborn
def plot_confusion_matrix(cm, labels, title):
    sns.heatmap(cm, annot=True, fmt='d', xticklabels=labels, yticklabels=labels, cmap='Oranges')
    plt.xlabel("Predicted Output")
    plt.ylabel("True Output")
    plt.title(title)
    plt.show()

# Class labels
labels = ["WAB < 12", "WAB ≥ 12"]

# Train set
plot_confusion_matrix(conf_matrix_train, labels, "Confusion matrix - Training set")

# Test set
plot_confusion_matrix(conf_matrix_test, labels, "Confusion matrix - Test set")
```





RFc model saving

```python
joblib.dump(RFc, 'Exit path/RFc_model.pkl')
```

```
['D:/RFc_model.pkl']
```

Part 3: Creating classification maps

importing the classification model

```python
# Import model
RFc_loaded = joblib.load('path/RFc_model.pkl')
```

```python
# Number of images to be processed
num_images = len(masked_cubes)

# Loop over all hyperspectral images
for i, cube_hyperspectral in enumerate(masked_cubes):
    # Hyperspectral cube dimensions
    height, width, num_bands = cube_hyperspectral.shape

    # Creation of ROI mask to detect valid pixels
    roi_mask = np.any(cube_hyperspectral > 0, axis=2)  # True for valid pixels

    # Initialization of the prediction map with NaN as background
    prediction_map = np.full((height, width), np.nan, dtype=np.float32)

    # Extraction of valid pixels for prediction
    selected_pixels = cube_hyperspectral[roi_mask].reshape(-1, num_bands)

    selected_pixelssnv = snv(selected_pixels)  # SNV filtering

    # Prediction with the trained model
    predicted_classes = RFc_loaded.predict(selected_pixelssnv).flatten()

    # Placement of predictions on the map only at valid pixels
    prediction_map[roi_mask] = predicted_classes
    # Classification card display
```

```
plt.figure(figsize=(10, 5))
plt.imshow(color_map, cmap="Purp", vmin=0, vmax=3)  # Blue = Class 0, Green = Class 1

# Add a white border and an intermediate color
cbar = plt.colorbar(ticks=[0, 1, 2, 3])
cbar.ax.set_yticklabels(["Red 1", "Red 2", "Red 3", "Red 4"])
plt.axis('off')  # Hide axes
plt.show()
```