Course M-GFP3: Imaging and non-imaging spectroscopy:
Term Paper

Strategies to enhance predictive modeling of soil organic carbon (SOC) using the LUCAS topsoil spectral library.

Deepak, Khuzaima, Luis

2025-02-20

Table of contents

1 Packages

```
# Use autoreload to automatically reload modules
%load_ext autoreload
%autoreload
```

```
import matplotlib.pyplot as plt
import numpy as np
```

```python
import pandas as pd
import torch
import torch.nn as nn
import torch.optim as optim
from scipy.signal import savgol_filter
from scipy.stats import pearsonr
from sklearn.cross_decomposition import PLSRegression
from sklearn.decomposition import PCA
from sklearn.metrics import mean_squared_error, pairwise_distances
from sklearn.model_selection import cross_val_score, train_test_split
from sklearn.preprocessing import StandardScaler
```

```python
import own_functions
```

2 Data

Data splitting (5 P):

- Split your data into a calibration data set (~70%) and an independent test data set (~30%).
- Show that both are representative of the full data set.
- For procedures with randomized approaches, please define and note the seed (in R: set.seed( )) to make the split reproducible for the instructors.
- From this point onward, the composition of the test data set must remain constant and unchanged for all subsequent tasks

2.1 Load and Clean

```python
# Load data
data = pd.read_csv('France_spc.csv')

# Remove unnecessary column
data = data.drop(columns=['Unnamed: 0'])

print(f"Data rows: {data.shape[0]}, columns: {data.shape[1]}")
display(data.head())
```

```
Data rows: 2807, columns: 1000
```

	500	502	504	506	508	510	512	514	516	518
0	0.137399	0.139045	0.140758	0.142544	0.144388	0.146281	0.148221	0.150205	0.152239	0.154322
1	0.141740	0.142851	0.144007	0.145208	0.146450	0.147726	0.149025	0.150348	0.151702	0.153081
2	0.140713	0.142216	0.143778	0.145392	0.147053	0.148756	0.150488	0.152257	0.154059	0.155892

	500	502	504	506	508	510	512	514	516	518
3	0.128922	0.129908	0.130919	0.131959	0.133019	0.134102	0.135196	0.136307	0.137433	0.138571
4	0.161760	0.163229	0.164741	0.166298	0.167895	0.169530	0.171194	0.172890	0.174611	0.176356

```
target = pd.read_csv('France_lab.csv')
target = target['SOC']
print(f"Target rows: {target.shape[0]}")
```

```
Target rows: 2807
```

2.2 Sampling and Splitting

```
# Extract features and target as numpy array
X = data.values
y = target.values
```

```
# Extract features and target as numpy array
X = data.values
y = target.values

### Sampling strategies
# Step 1: Generate or Load Data
np.random.seed(100)  # Set seed for reproducibility

# Step 2: Random Split (70% Calibration, 30% Test)
X_train_random, X_test_random, y_train_random, y_test_random = train_test_split(X, y, test_s

# Step 3: Apply Kennard-Stone to select 70% of the data
n_train = int(0.7 * X.shape[0])

# Get indices
ks_indices = own_functions.kennard_stone(X, n_train)

# Select Training data
X_train_ks = X[ks_indices,:]
y_train_ks = y[ks_indices]

# Select Test
test_indices = np.setdiff1d(np.arange(X.shape[0]), ks_indices)
X_test_ks = X[test_indices]
y_test_ks = y[test_indices]

# Step 4: PCA for Visualization
```

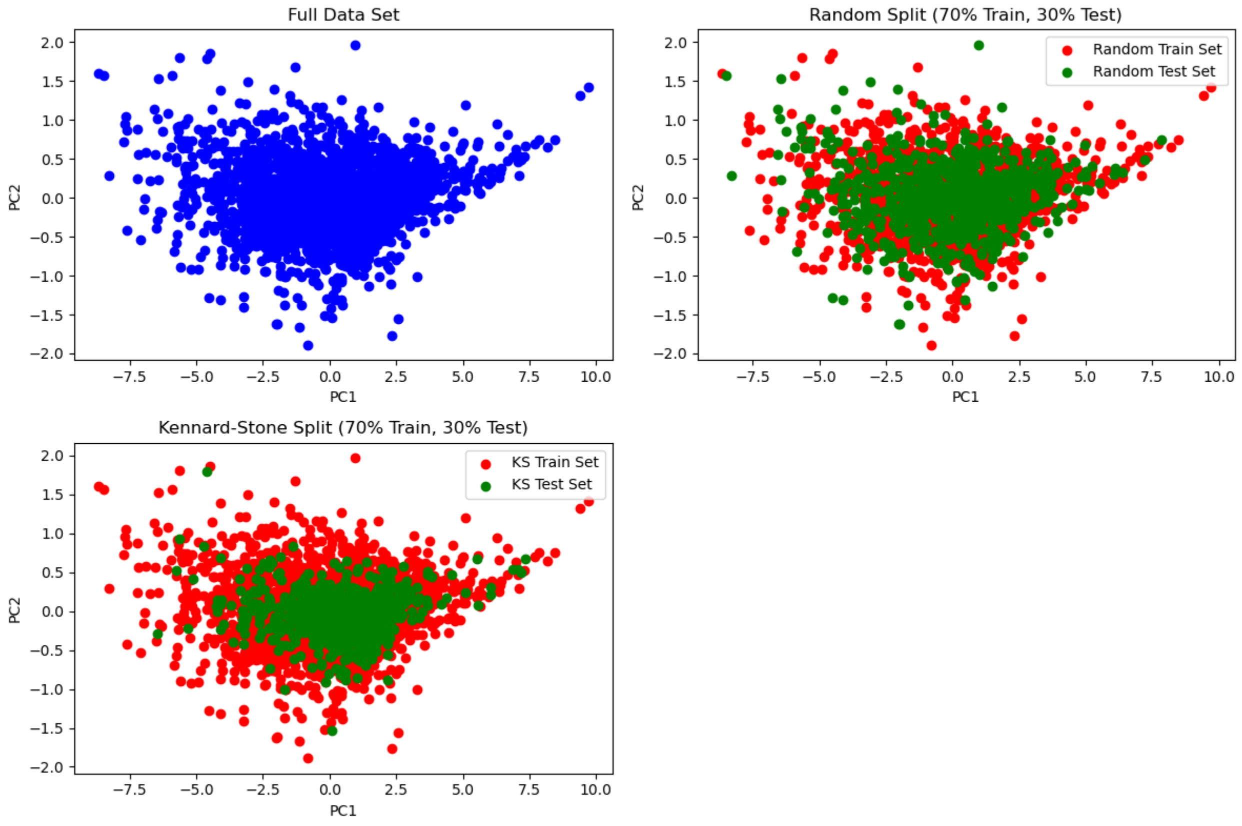
```
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X)  # PCA on full data
X_train_random_pca = pca.transform(X_train_random)  # PCA on random calibration set
X_test_random_pca = pca.transform(X_test_random)  # PCA on random test set
X_cal_ks_pca = pca.transform(X_train_ks)  # PCA on Kennard-Stone calibration set
X_test_ks_pca = pca.transform(X_test_ks)  # PCA on Kennard-Stone test set
```

```
import own_functions
```

```
#TODO: Show that both test and train are representative of the full dataset
```

```
# Step 5: Plot Results
own_functions.plot_pca_comparison(X_full=X,
                                  X_train_random=X_train_random,
                                  X_test_random=X_test_random,
                                  X_train_ks=X_train_ks,
                                  X_test_ks=X_test_ks)
```

```
Random Split:
Train set shape: (1964, 1000)
Test set shape: (843, 1000)

Kennard-Stone Split:
Train set shape: (1964, 1000)
Test set shape: (843, 1000)
```



2.3 Preprocessing

```python
# Function to calculate RMSE using cross-validation
def calculate_rmse(n_components, X_train, y_train):
    pls = PLSRegression(n_components=n_components)
    # Use negative mean squared error as cross-validation scoring
    mse = cross_val_score(pls, X_train, y_train, cv=5, scoring='neg_mean_squared_error')
    rmse = np.sqrt(-mse).mean()  # Take square root of MSE and average it across folds
    return rmse
```

3 Basemodel

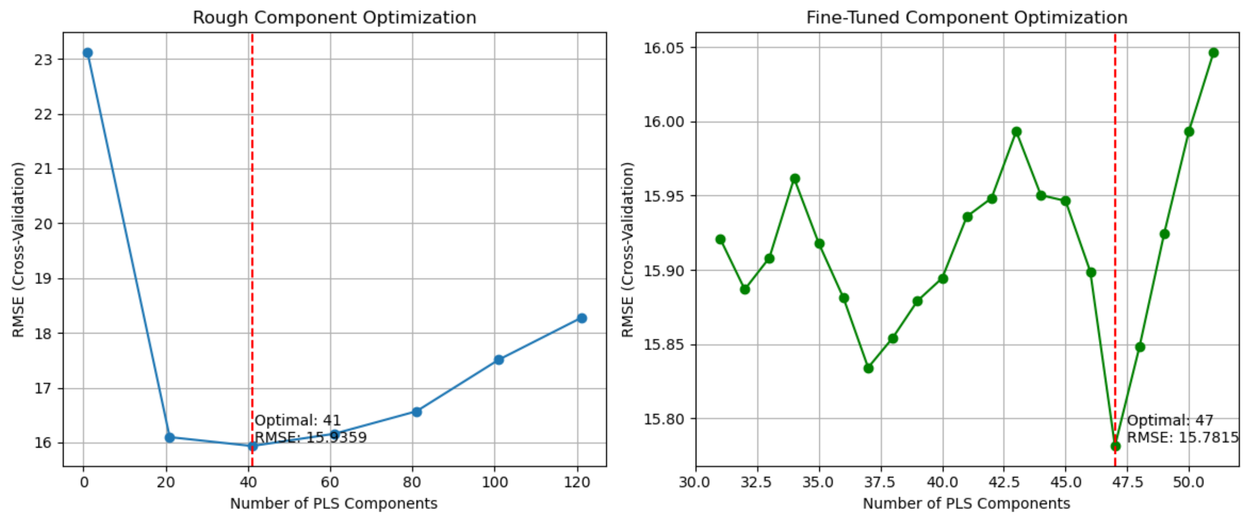
Baseline model (5 P): - Develop a global baseline PLSR model using the *calibration dataset* - (entire VNIR range from 500 nm to 2499 nm in steps of 2 nm) - *without* applying any *spectral preprocessing.* - The target variable is soil organic carbon (SOC). - Perform *internal optimization* to *determine* the *optimal number* of *latent PLS variables* - *report your selected value.* - Apply the optimized model* to the independent test set*.* - *Compute the validation metrics* (R^2, RMSE, bias, and RPD) - visualize* the results in a *scatter plot* (observed vs. predicted values) - and assess the model's performance.

3.1 Finding optimal number of commponents

```
import own_functions
```

```
plsr_base_components = own_functions.optimize_pls_components(X_train=X_train_ks,
                                        y_train=y_train_ks,
                                        max_components=140,
                                        step=20,
                                        fine_tune=True,
                                        show_progress=True,
                                        plot_results=True
                                        )
```

```
Rough Optimization:   0%|          | 0/7 [00:00<?, ?it/s]

Fine Tuning:   0%|          | 0/21 [00:00<?, ?it/s]
```

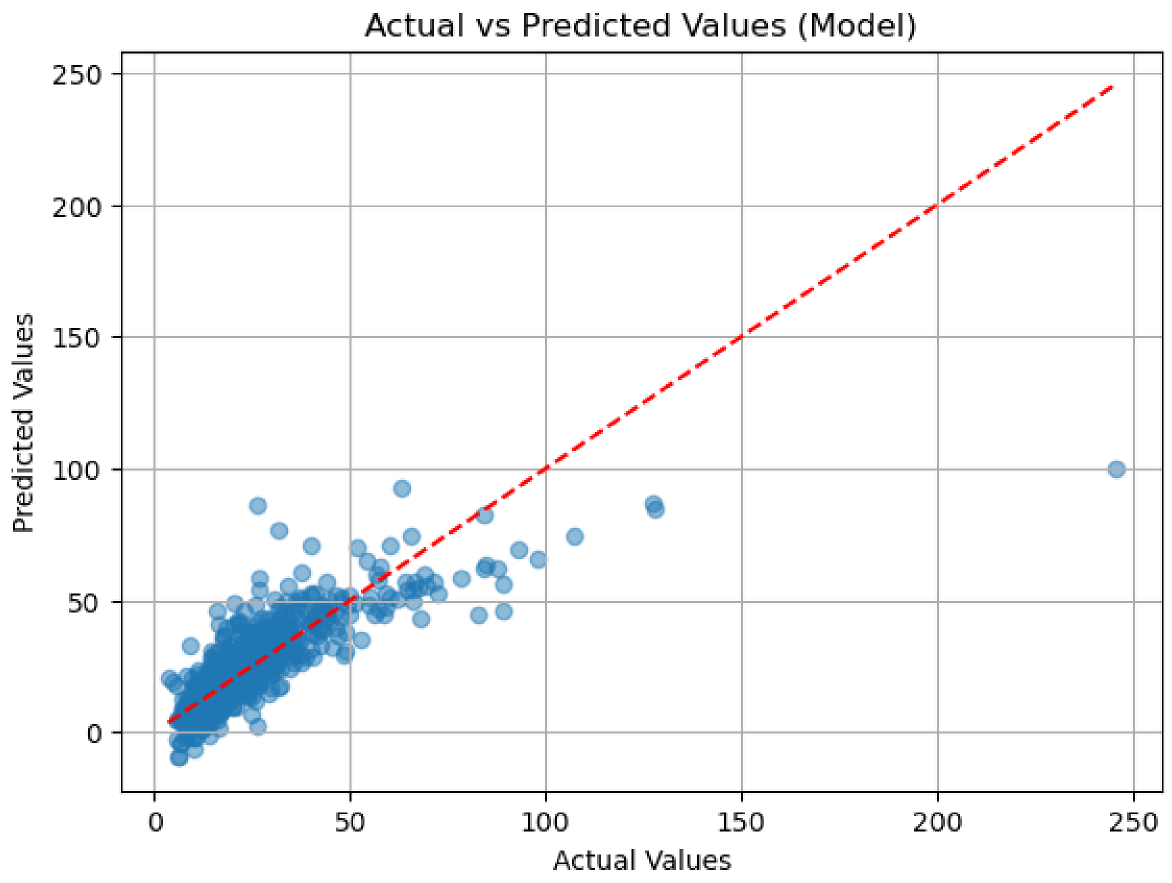


3.2 Evaluating Base Model

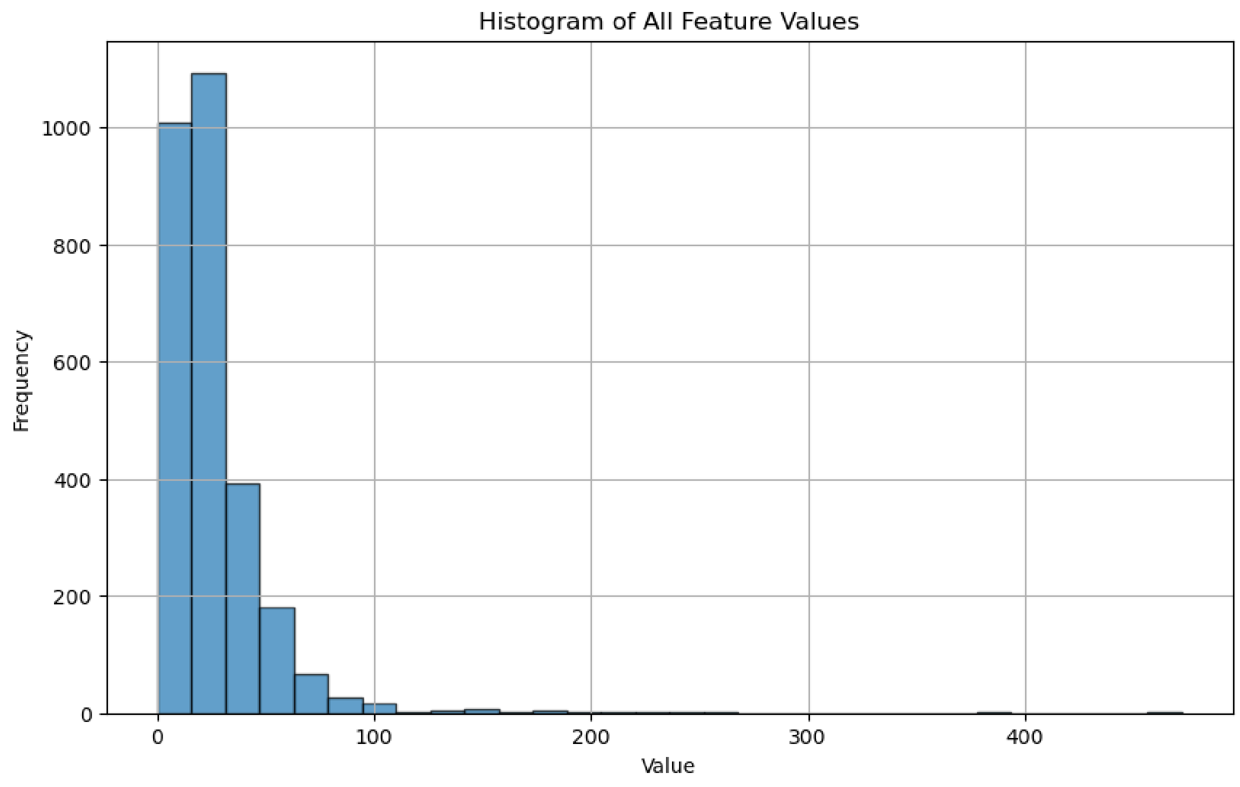
```
import own_functions
```

```
plsr_base_model = PLSRegression(n_components=plsr_base_components["optimal_n"])
plsr_base_model.fit(X_train_ks, y_train_ks)

plsr_base_eval = own_functions.evaluate_model(plsr_base_model,
                             X_test=X_test_ks,
                             y_test=y_test_ks,
                                                  print_metrics=True,
                                                  show_plot=True
                                                  )
```

```
Root Mean Squared Error (RMSE): 10.0547
R²: 0.6552
Bias: -0.1772
RPD: 1.7030
```



```
all_values = y.flatten()

# Plot histogram of all feature values
plt.figure(figsize=(10, 6))
plt.hist(all_values, bins=30, edgecolor='k', alpha=0.7)
plt.title('Histogram of All Feature Values')
plt.xlabel('Value')
plt.ylabel('Frequency')
plt.grid(True)
plt.show()
```



4 Model Improvement Strategies (5 P per strategy):

- Develop and evaluate three distinct strategies to improve the baseline model,
  - using the **same independent test set for validation**.
- For each strategy, report the validation metrics
  - (R^2, RMSE, bias, and RPD),
  - visualize the best result in a scatter plot (observed vs.predicted values)
  - assess the performance of these alternative models.
  - Use the same independent test set for all strategies to ensure that validation metrics are directly comparable.

IMPORTANT: Testing two or more spectral preprocessing methods is considered one strategy, not multiple strategies. Similarly, testing one or more alternative regression algorithms counts as one strategy, not multiple.

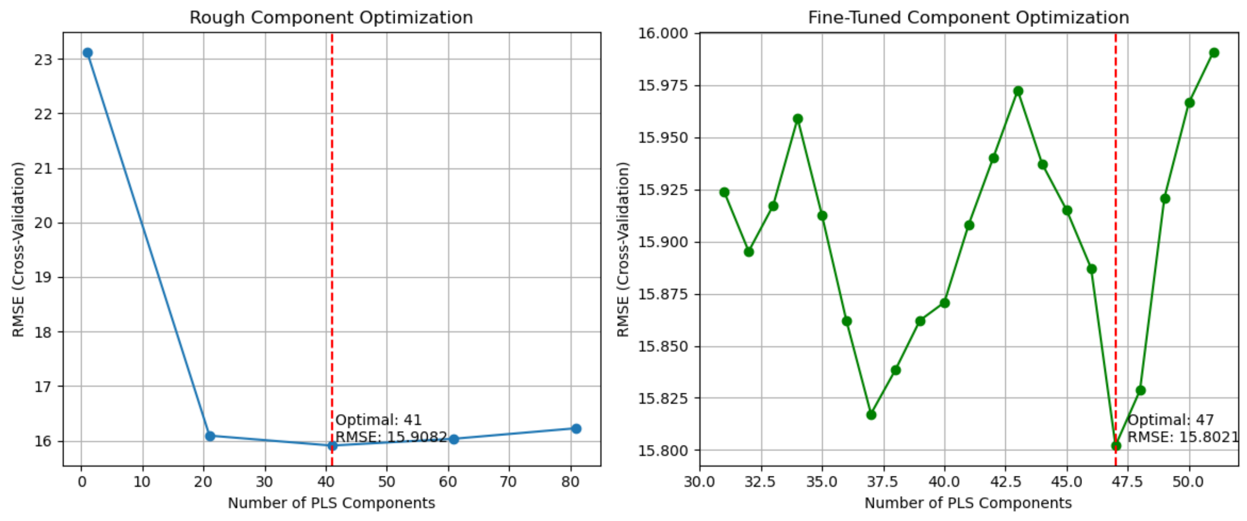
4.1 Varying Preprocessing Strategy

4.1.1 Savitzgy-Golay

```
#TODO: Is scaling necessary?

# Applying Savitzky-Golay filter to calibration and test data
X_train_ks_sg = own_functions.apply_savitzky_golay(X_train_ks, window_length=11, polyorder=2
X_test_ks_sg = own_functions.apply_savitzky_golay(X_test_ks, window_length=11, polyorder=2,

# Standardize data
scaler_ks_sg = StandardScaler()
X_train_ks_sg_scaled = scaler_ks_sg.fit_transform(X_train_ks_sg)
X_test_ks_sg_scaled = scaler_ks_sg.transform(X_test_ks_sg)
```

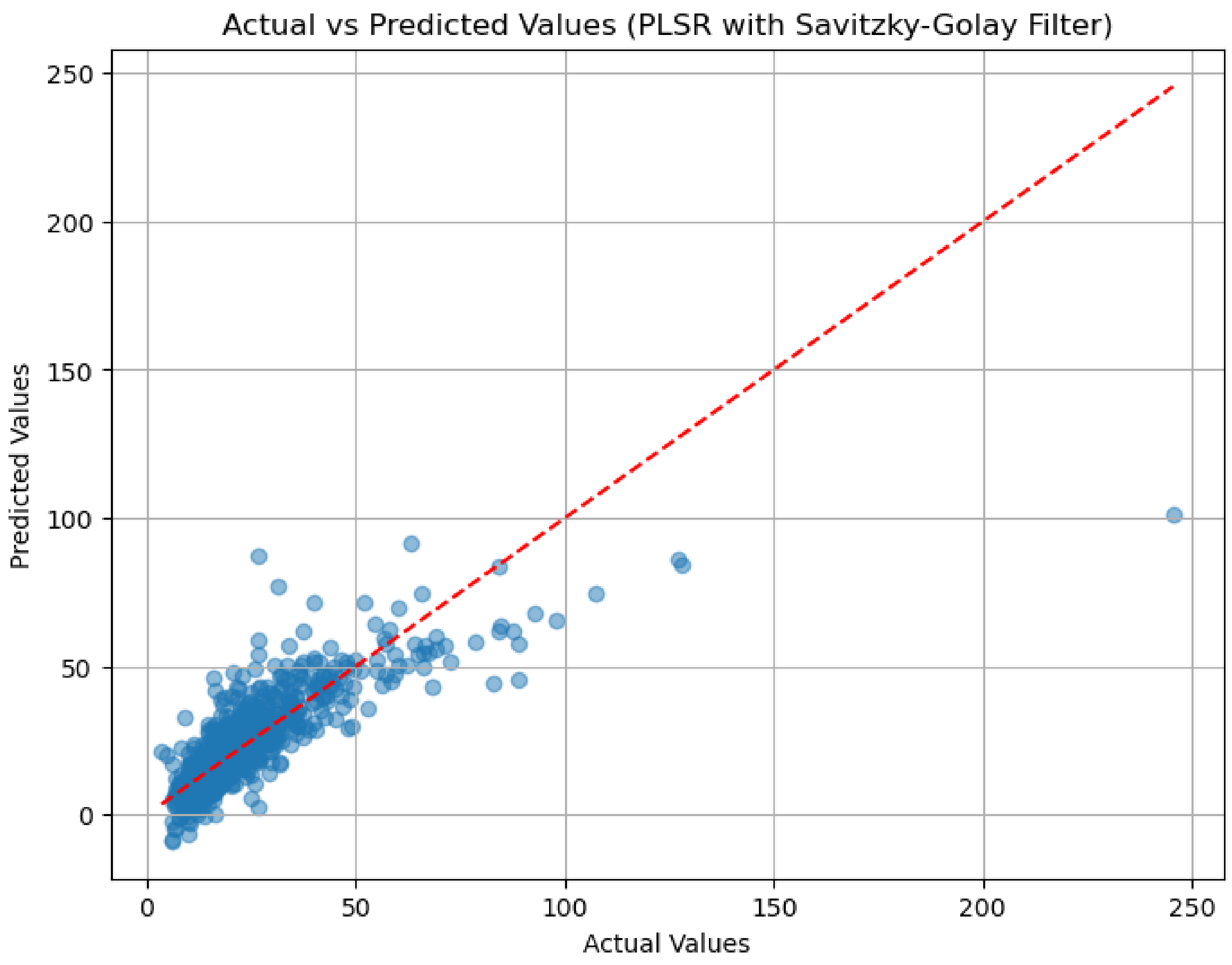
```
plsr_sgolay_components = own_functions.optimize_pls_components(X_train=X_train_ks_sg_scaled,
                                      y_train=y_train_ks,
                                      max_components=100,
                                      step=20,
                                      fine_tune=True,
                                      show_progress=True,
                                      plot_results=True
                                      )
```

```
Rough Optimization:   0%|          | 0/5 [00:00<?, ?it/s]

Fine Tuning:   0%|          | 0/21 [00:00<?, ?it/s]
```



```
plsr_sg_model = PLSRegression(n_components=plsr_sgolay_components["optimal_n"])
plsr_sg_model.fit(X_train_ks_sg_scaled, y_train_ks)

plsr_sg_eval = own_functions.evaluate_model(plsr_sg_model,
                             X_test=X_test_ks_sg_scaled,
                             y_test=y_test_ks,
                             print_metrics=True,
                             show_plot=True,
                             plot_kwargs={'model_name': 'PLSR with Savitzky-Golay Filter',
                                          'figsize': (8, 6)}
                             )
```

```
Root Mean Squared Error (RMSE): 10.0590
R²: 0.6549
Bias: -0.1640
RPD: 1.7023
```



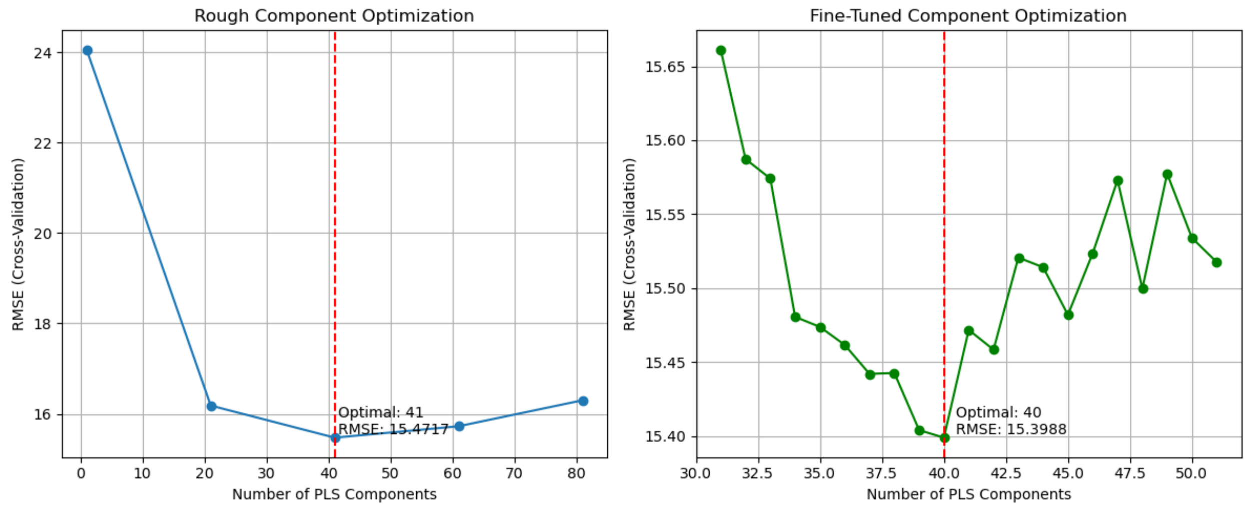
4.1.2 Standard Normal Variate

```
import own_functions
```

```
# Applying Savitzky-Golay filter to calibration and test data
X_train_ks_snv = own_functions.standard_normal_variate(X_train_ks)
X_test_ks_snv = own_functions.standard_normal_variate(X_test_ks)

# Standardize data
scaler_ks_snv = StandardScaler()
X_train_ks_snv_scaled = scaler_ks_snv.fit_transform(X_train_ks_snv)
X_test_ks_snv_scaled = scaler_ks_snv.transform(X_test_ks_snv)
```

```
plsr_snv_components = own_functions.optimize_pls_components(X_train=X_train_ks_snv_scaled,
                                      y_train=y_train_ks,
                                      max_components=100,
                                      step=20,
                                      fine_tune=True,
                                      show_progress=True,
                                      plot_results=True
                                      )
```

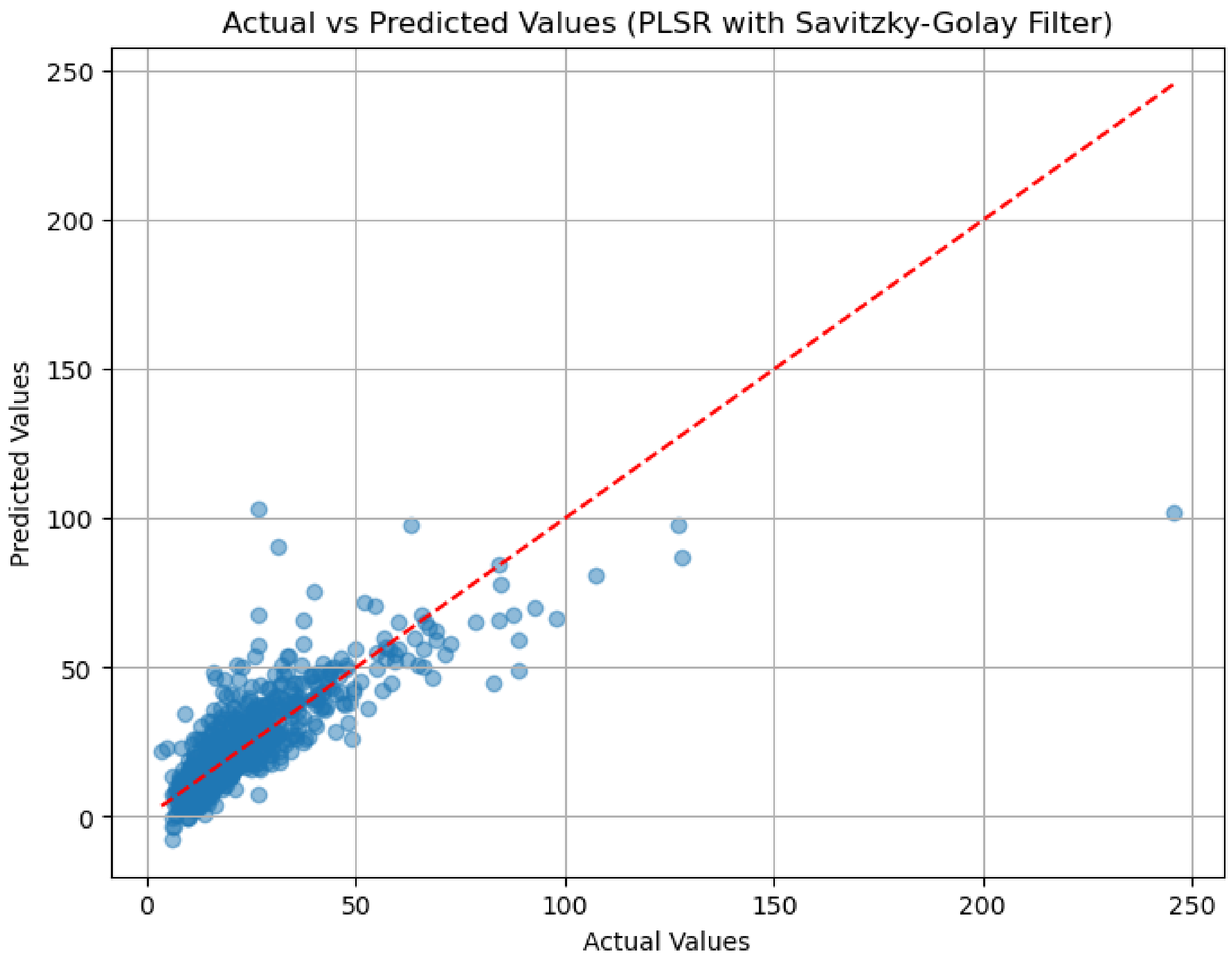
```
Rough Optimization:   0%|          | 0/5 [00:00<?, ?it/s]

Fine Tuning:   0%|          | 0/21 [00:00<?, ?it/s]
```



```
plsr_snv_model = PLSRegression(n_components=plsr_snv_components["optimal_n"])
plsr_snv_model.fit(X_train_ks_snv, y_train_ks)
```

```python
plsr_snv_eval = own_functions.evaluate_model(plsr_snv_model,
                          X_test=X_test_ks_snv,
                          y_test=y_test_ks,
                          print_metrics=True,
                          show_plot=True,
                          plot_kwargs={'model_name': 'PLSR with Savitzky-Golay Filter',
                                       'figsize': (8, 6)}
                          )
```

```
Root Mean Squared Error (RMSE): 10.1517
R²: 0.6485
Bias: 0.3009
RPD: 1.6867
```



4.2 Testing Different Models

4.2.1 Pytorch LSTM

```
#TODO: Changed learning rate for less epochs, original: 0.001, 3000, new: 0.005, 500

X_train = X_train_ks
X_test = X_test_ks
y_train = y_train_ks
y_test = y_test_ks





# Define the PLS Regression model for dimensionality reduction
n_components = min(40, X_train.shape[1])  # Adjust based on your data; 10 is an example
pls = PLSRegression(n_components=n_components)

# Fit PLSR on the training data and transform both training and test sets
#X_train_pls = pls.fit_transform(X_train, y_train)[0]
#X_test_pls = pls.transform(X_test)# check using traning data


# Fit PLSR on the training data and transform the training set
X_train_pls, _ = pls.fit_transform(X_train, y_train)

# Transform the test set using the fitted model (trained on the training set)
X_test_pls = pls.transform(X_test)


#pls  scores loading regression coeffieicent latent  variables ,

# Convert to PyTorch tensors
X_train_tensor = torch.tensor(X_train_pls, dtype=torch.float32)
y_train_tensor = torch.tensor(y_train, dtype=torch.float32)
X_test_tensor = torch.tensor(X_test_pls, dtype=torch.float32)
y_test_tensor = torch.tensor(y_test, dtype=torch.float32)

# Define LSTM model
class LSTMModel(nn.Module):
    def __init__(self, input_size, hidden_size, output_size, num_layers):
        super(LSTMModel, self).__init__()
        self.hidden_size = hidden_size
        self.num_layers = num_layers

        # LSTM layers
        self.lstm = nn.LSTM(input_size, hidden_size, num_layers, batch_first=True)
        # Fully connected layer
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x):

        # Initialize hidden and cell states
        h0 = torch.zeros(self.num_layers, x.size(0), self.hidden_size).to(x.device)
        c0 = torch.zeros(self.num_layers, x.size(0), self.hidden_size).to(x.device)

        # Forward pass through LSTM layer
        out, _ = self.lstm(x, (h0, c0))

        # Select the last time step
        out = out[:, -1, :]

        # Forward pass through fully connected layer
        out = self.fc(out)
        return out

# Hyperparameters
input_size = X_train_pls.shape[1]  # Number of features after PLSR
hidden_size = 256
output_size = 1  # For regression
num_layers = 5
num_epochs = 500
learning_rate = 0.005

# Initialize the model, loss function, and optimizer
model = LSTMModel(input_size, hidden_size, output_size, num_layers)
criterion = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=learning_rate)

# Training loop
for epoch in range(num_epochs):
    model.train()

    # Forward pass
    outputs = model(X_train_tensor.unsqueeze(1))  # Add sequence dimension
    loss = criterion(outputs.squeeze(), y_train_tensor)

    # Backward pass and optimization
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

```
        if (epoch + 1) % 10 == 0:
            print(f'Epoch [{epoch + 1}/{num_epochs}], Loss: {loss.item():.4f}')

# Evaluate the model
model.eval()
with torch.no_grad():
    # Make predictions on the test set
    predictions = model(X_test_tensor.unsqueeze(1)).squeeze()

    # Calculate test loss
    test_loss = criterion(predictions, y_test_tensor)
    print(f'Test Loss: {test_loss.item():.4f}')

    # Convert predictions and y_test to numpy arrays
    y_pred_np = predictions.numpy()
    y_test_np = y_test_tensor.numpy()

    # Calculate RMSE
    rmse = np.sqrt(mean_squared_error(y_test_np, y_pred_np))
    print(f'Root Mean Squared Error (RMSE): {rmse:.4f}')

    # Calculate correlation coefficient
    correlation, _ = pearsonr(y_test_np, y_pred_np)
    print(f'Correlation coefficient: {correlation:.4f}')

    # Plot actual vs. predicted values
    plt.figure(figsize=(10, 5))
    plt.scatter(y_test_np, y_pred_np, alpha=0.5)
    plt.plot([min(y_test_np), max(y_test_np)], [min(y_test_np), max(y_test_np)], color='red'
    plt.xlabel('Actual Values')
    plt.ylabel('Predicted Values')
    plt.title('Actual vs Predicted Values (PLSR + LSTM)')
    plt.grid(True)
    plt.show()
```

```
KeyboardInterrupt:
```

```
# Normalize features
scaler = StandardScaler()

#X_pca_scaled = scaler.fit_transform(X)


# Split data into training and test sets

X_train = scaler.fit_transform(X_train)
```

```
X_test = scaler.transform(X_test)


# Convert to PyTorch tensors
X_train = torch.tensor(X_train, dtype=torch.float32)
y_train = torch.tensor(y_train, dtype=torch.float32)
X_test = torch.tensor(X_test, dtype=torch.float32)
y_test = torch.tensor(y_test, dtype=torch.float32)

# Define LSTM model
class LSTMModel(nn.Module):
    def __init__(self, input_size, hidden_size, output_size, num_layers):
        super(LSTMModel, self).__init__()
        self.hidden_size = hidden_size
        self.num_layers = num_layers

        # LSTM layers
        self.lstm = nn.LSTM(input_size, hidden_size, num_layers, batch_first=True)
        # Fully connected layer
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        # Initialize hidden and cell states
        h0 = torch.zeros(self.num_layers, x.size(0), self.hidden_size).to(x.device)
        c0 = torch.zeros(self.num_layers, x.size(0), self.hidden_size).to(x.device)

        # Forward pass through LSTM layer
        out, _ = self.lstm(x, (h0, c0))
        # Select the last time step
        out = out[:, -1, :]
        # Forward pass through fully connected layer
        out = self.fc(out)
        return out

# Hyperparameters
input_size = X_train.shape[1]  # Number of features after PCA and scaling
hidden_size = 256
output_size = 1  # For regression
num_layers = 5
num_epochs = 2000
learning_rate = 0.001

# Initialize the model, loss function, and optimizer
model = LSTMModel(input_size, hidden_size, output_size, num_layers)
criterion = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=learning_rate)
```

```
# Training loop
for epoch in range(num_epochs):
    model.train()

    # Forward pass
    outputs = model(X_train.unsqueeze(1))  # Add sequence dimension
    loss = criterion(outputs.squeeze(), y_train)

    # Backward pass and optimization
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    if (epoch + 1) % 10 == 0:
        print(f'Epoch [{epoch + 1}/{num_epochs}], Loss: {loss.item():.4f}')

# Evaluate the model
model.eval()
with torch.no_grad():
    # Make predictions on the test set
    predictions = model(X_test.unsqueeze(1)).squeeze()

    # Calculate test loss
    test_loss = criterion(predictions, y_test)
    print(f'Test Loss: {test_loss.item():.4f}')

    # Convert predictions and y_test to numpy arrays
    y_pred_np = predictions.numpy()
    y_test_np = y_test.numpy()

    # Calculate RMSE
    rmse = np.sqrt(mean_squared_error(y_test_np, y_pred_np))
    print(f'Root Mean Squared Error (RMSE): {rmse:.4f}')

    # Calculate correlation coefficient
    correlation, _ = pearsonr(y_test_np, y_pred_np)
    print(f'Correlation coefficient: {correlation:.4f}')

    # Plot actual vs. predicted values
    plt.figure(figsize=(10, 5))
    plt.scatter(y_test_np, y_pred_np, alpha=0.5)
    plt.plot([min(y_test_np), max(y_test_np)], [min(y_test_np), max(y_test_np)], color='red'
    plt.xlabel('Actual Values')
    plt.ylabel('Predicted Values')
    plt.title('Actual vs Predicted Values')
    plt.grid(True)
    plt.show()
```

```
NameError: name 'torch' is not defined
```

5 Discussion of Results (5 P):

- Briefly discuss your results and interpret them based on the validation metrics for the test set.
- Compare your findings with those of published studies in a similar context.
- Evaluate whether soil VNIR reflectance spectroscopy could serve as a complementary approach for large-scale soil organic carbon assessment in Earth (system) science.

Additional Information:

The length of the discussion section really depends on your results, but as a general guideline, I would expect it to be around one page.

- **Focus on**:
  - directly comparing your different modeling approaches
  - interpreting which performed best based on the validation metrics
- If the results are not as good as expected:
  - consider discussing possible reasons and suggesting ways to improve them
  - (you might find 1-2 examples from the literature helpful here).
- Additionally, you could compare your findings with similar studies that have attempted to model SOC (or related properties) at national or continental scales using spectroscopy—ideally referencing 2-3 relevant publications.
- Finally, reflect on whether and how soil VNIR spectroscopy could contribute to large-scale soil information systems.
  - This is a more theoretical aspect, and you are free in how you approach this point.
  - Important aspects to consider might include:
    - a) Model accuracy (What would be considered a good accuracy in this context?)
    - b) Data harmonization (Challenges when combining datasets from different providers)
    - c) Practical usability (Would end users require programming skills, etc.?)

A recent publication that could provide a useful overview is: Peng et al. (2025): Spectroscopic solutions for generating new global soil information (Link: https://www.sciencedirect.com/science/article/pii/S266667