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

Outline

Structure

Authors

Part I

Main

1 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.

2 Markdown structure

After each major task, we have provided a **detailed overview** to summarize what was done and the results of that particulat task.

3 Packages

```
# Use autoreload to automatically reload modules
%load_ext autoreload
%autoreload 2
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

```
import own_functions
```

```
import geopandas as gpd
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns
import torch
import torch.nn as nn
import torch.optim as optim
from autogluon.tabular import TabularDataset, TabularPredictor
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, r2_score
from sklearn.model_selection import cross_val_score, train_test_split
from sklearn.preprocessing import StandardScaler
```

```
import autogluon
```

4 Data

TASK 1

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

4.1 Load and Clean

```
# Load data
data = pd.read_csv('data/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	0.15432
1	0.141740	0.142851	0.144007	0.145208	0.146450	0.147726	0.149025	0.150348	0.151702	0.15308
2	0.140713	0.142216	0.143778	0.145392	0.147053	0.148756	0.150488	0.152257	0.154059	0.15589
3	0.128922	0.129908	0.130919	0.131959	0.133019	0.134102	0.135196	0.136307	0.137433	0.13857
4	0.161760	0.163229	0.164741	0.166298	0.167895	0.169530	0.171194	0.172890	0.174611	0.17635

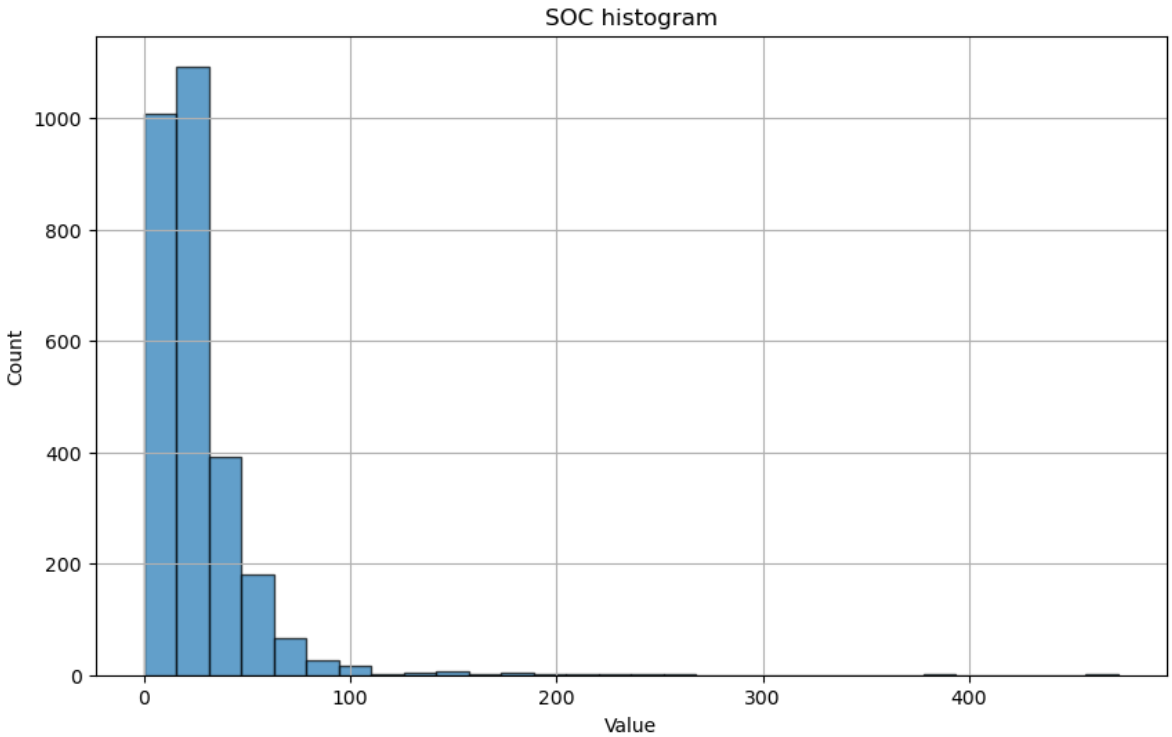
```
target_raw = pd.read_csv('data/France_lab.csv')
lat_lon = target_raw[['GPS_LAT', 'GPS_LONG']]
target = target_raw['SOC']
print(f"Target rows: {target.shape[0]}")
```

```
Target rows: 2807
```

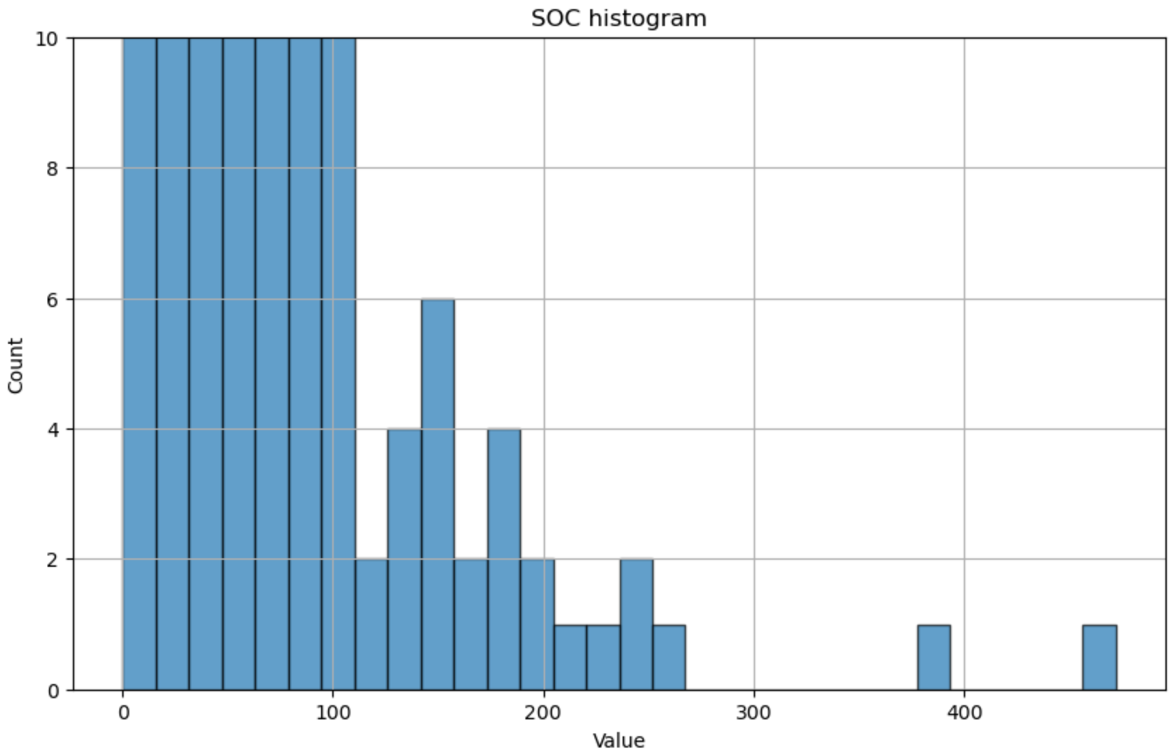
```
target_raw
```

	Unnamed: 0	SAMPLE_ID	CLAY	SILT	SAND	SOC	CaCO3	N	P	K	CEC	C
0	1	10000	40.0	52.0	8.0	15.6	1	1.4	42.6	491.1	24.6	V
1	2	10001	26.0	18.0	56.0	19.8	1	1.6	19.5	279.1	20.6	V
2	3	10002	22.0	41.0	37.0	33.5	1	2.6	37.8	399.1	15.0	V
3	4	10004	27.0	47.0	26.0	66.1	21	6.6	147.7	1080.6	30.5	V
4	5	10005	16.0	32.0	52.0	38.1	0	2.6	49.6	293.9	7.8	V
...	...	...	...	...	...	...	...	...	...	...	...	.
2802	2803	9994	19.0	62.0	19.0	9.1	0	1.2	44.5	131.8	9.7	V
2803	2804	9995	16.0	41.0	42.0	13.4	0	1.4	33.0	184.4	7.2	V
2804	2805	9996	13.0	29.0	58.0	8.7	3	1.3	104.9	425.4	7.7	V
2805	2806	9997	20.0	38.0	42.0	30.6	0	3.0	56.1	107.8	12.6	V
2806	2807	9998	11.0	56.0	34.0	5.9	0	0.7	39.7	172.1	3.8	V

```
# plot soc histogram
plt.figure(figsize=(10, 6))
plt.hist(target, bins=30, edgecolor='k', alpha=0.7)
plt.title('SOC histogram')
plt.xlabel('Value')
plt.ylabel('Count')
plt.grid(True)
plt.show()
```




```
# plot soc histogram
plt.figure(figsize=(10, 6))
plt.hist(target, bins=30, edgecolor='k', alpha=0.7)
plt.title('SOC histogram')
plt.xlabel('Value')
plt.ylabel('Count')
plt.ylim((0,10))
plt.grid(True)
plt.show()
```



4.2 Sampling and Splitting

```
# 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

print(f"Random Split: {X_train_random.shape[0]} training samples, {X_test_random.shape[0]}

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

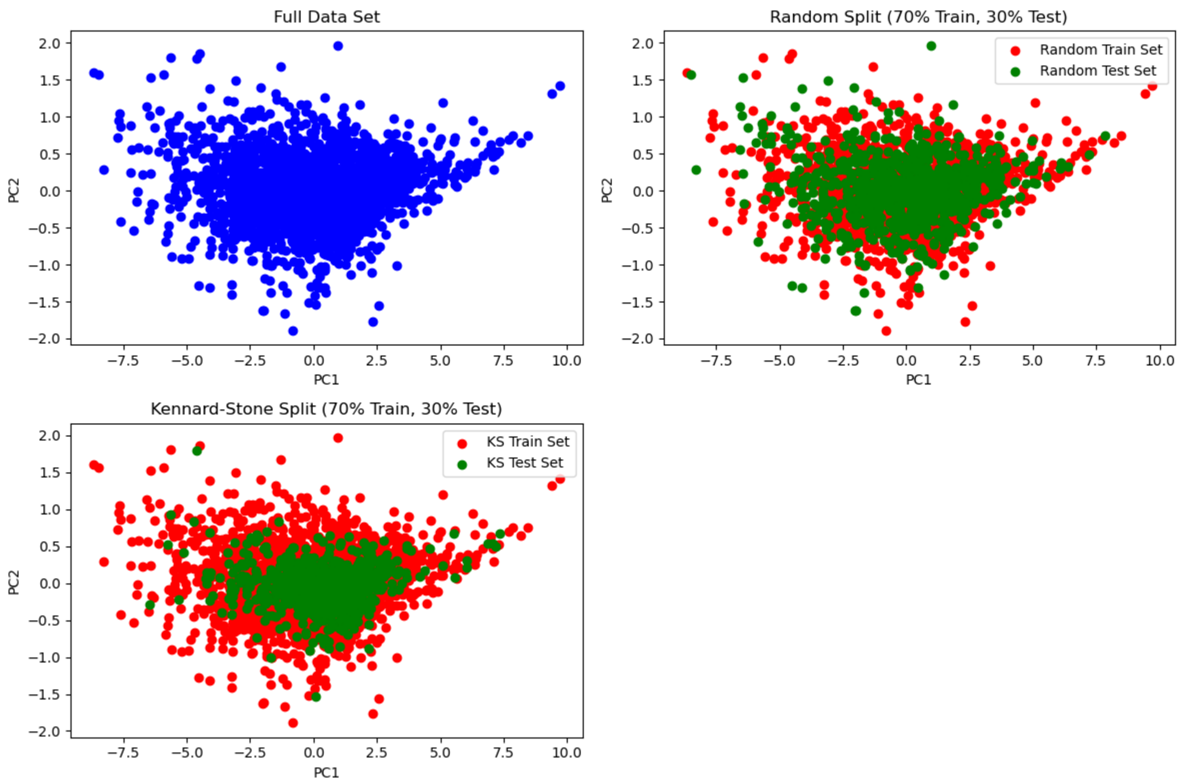
```python
# Get indices
ks_indices = own_functions.kennard_stone(X, n_train)

# 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]

print(f"Kennard-Stone: {X_train_ks.shape[0]} training samples, {X_test_ks.shape[0]} test s

# Step 4: PCA for Visualization
pca = PCA(n_components=2)

# Fit PCA on full data
X_pca = pca.fit_transform(X)

# Transform 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
```

```
Random Split: 1964 training samples, 843 test samples
Kennard-Stone: 1964 training samples, 843 test samples
```

```python
# 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)
```

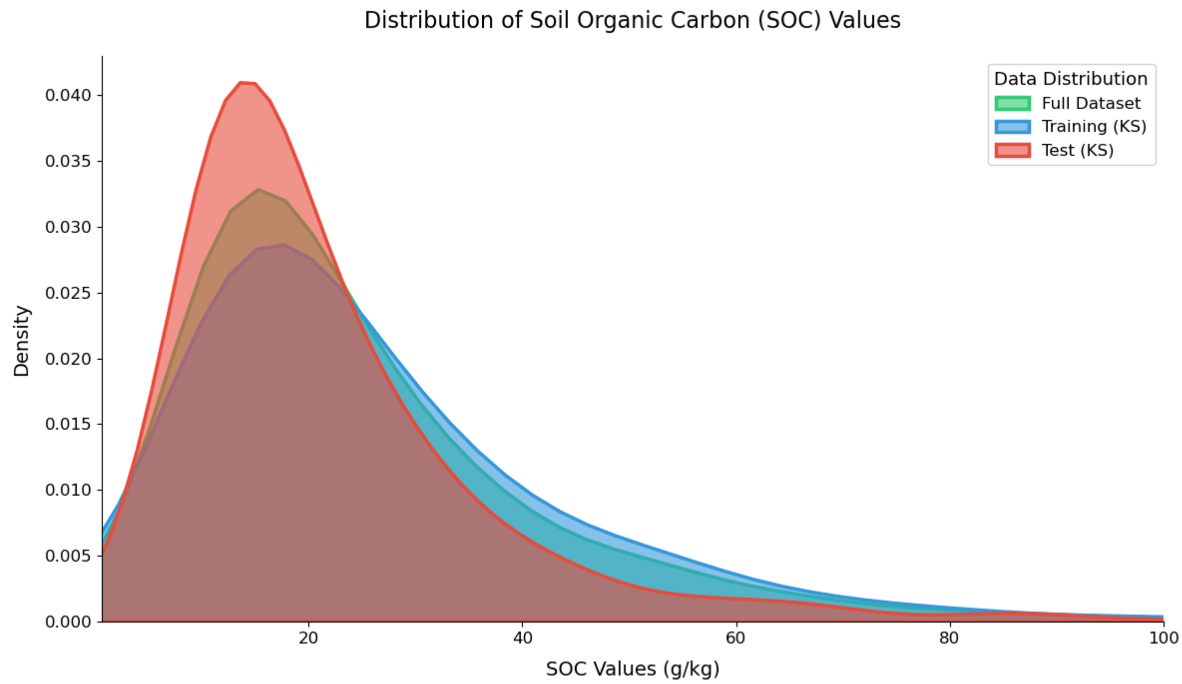
```
Random Split:
Train set shape: (1964, 1000)
Test set shape: (843, 1000)
```

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




```python
# Set up a larger figure for better visibility
plt.figure(figsize=(12, 7))

# Create DataFrames for all datasets (including full dataset)
df_full = pd.DataFrame({'SOC': y, 'Set': 'Full Dataset'})
df_train = pd.DataFrame({'SOC': y_train_ks, 'Set': 'Training (KS)'})
df_test = pd.DataFrame({'SOC': y_test_ks, 'Set': 'Test (KS)'})
df_combined = pd.concat([df_full, df_train, df_test])

# Define a nice color palette
colors = {'Full Dataset': '#2ecc71', 'Training (KS)': '#3498db', 'Test (KS)': '#e74c3c'}

# Plot each KDE separately to ensure proper legend handling
for dataset, color in colors.items():
    subset = df_combined[df_combined['Set'] == dataset]
```

```
    sns.kdeplot(data=subset, x='SOC',
                fill=True,
                alpha=0.6,
                linewidth=2.5,
                color=color,
                label=dataset,
                bw_adjust=1.2)  # Adjust bandwidth for smooth but still representative cur

# Enhance the plot aesthetics with better visibility
plt.title('Distribution of Soil Organic Carbon (SOC) Values', fontsize=16, pad=20)
plt.xlabel('SOC Values (g/kg)', fontsize=14, labelpad=10)
plt.ylabel('Density', fontsize=14, labelpad=10)
plt.tick_params(axis='both', which='major', labelsize=12)

# Create a more visible legend with explicit labels
plt.legend(title='Data Distribution', fontsize=12, title_fontsize=13,
          loc='upper right', frameon=True, framealpha=0.9)

# Set x-axis limits to focus on the main distribution (exclude extreme outliers)
q1, q99 = np.percentile(y, [1, 99])
plt.xlim([max(0, q1-5), min(100, q99+10)])

# Remove top and right spines for cleaner look
sns.despine()

plt.tight_layout()
plt.show()
```




```
X_train = X_train_ks
y_train = y_train_ks

X_test = X_test_ks
y_test = y_test_ks
```

```
#TODO: change text below: the KDE was only done for ks testing looking at training and tes
```

4.3 1. Overview for first task Data splitting (5 P)

4.3.1 Data

The dataset used for this study is based on **spectral data** and **Soil Organic Carbon (SOC)** measurements collected from LUVAS data base of country**France**. The dataset consists of **2807 samples** (rows) and **1000 spectral bands** (columns), with reflectance values ranging from **500 nm to 2498 nm**. This data was collected to assess the spatial distribution of SOC using spectral reflectance, and it includes both the **SOC values** along with so many other lab parameters and the corresponding **GPS coordinates** for geospatial analysis.

4.3.2 Data Preprocessing

- We remove columns that is not required to the analysis, leaving behind a final working dataset.
- The target variable, **SOC**, is extracted, along with the **GPS coordinates** for each sample, to allow for potential geospatial analysis.

Overview: After preprocessing, we have a clean dataset containing spectral data and SOC values with corresponding GPS coordinates.

4.3.3 Data Sampling

We employed two different sampling strategies to ensure that our model is robust and can generalize well:

1. **Random Split:** We performed a random split of the dataset, allocating **70%** for training (calibration) and **30%** for testing. This ensures that the model is trained on a representative portion of the data while still having a separate test set to evaluate its performance.

2. **Kennard-Stone Algorithm:** We applied the **Kennard-Stone algorithm** to select a calibration set that maximizes diversity in spectral space. The remaining samples are used as the independent test set. This approach ensures that the training data covers a broad range of spectral variations.

Overview: The data was split using two methods—random sampling and the Kennard-Stone algorithm—to ensure both representative and diverse training sets.

4.3.4 Distribution Analysis

After splitting the dataset, we performed an analysis to confirm that the calibration and test sets properly represent the distribution of **SOC** values in the original dataset:

- We conducted **Principal Component Analysis (PCA)** to visualize the distribution of the data and verify that both the training and test sets are representative of the overall dataset.
- Additionally, we generated **Kernel Density Estimation (KDE)** plots to compare the **SOC distributions** across three groups:
  - The **full dataset**
  - The **randomly selected training set**
  - The **Kennard-Stone-based training set**

Overview: The distribution analysis confirmed that both sampling strategies preserved the statistical characteristics of the dataset, ensuring that our model will be trained on a representative set.

4.3.5 Final Dataset Preparation

After confirming that the dataset was appropriately split and balanced, we are now ready to proceed with modeling using the **Kennard-Stone-selected calibration and test sets**.

Overview: The **Kennard-Stone** algorithm samples the extreme data points first and places them in the calibration data set, which makes the test data set less variable. But at the same time, the aim is to ensure that the calibration data set covers all eventualities. So we have used Kennard Stone data as final calibration and validation data.

TASK 2 # Basemodel

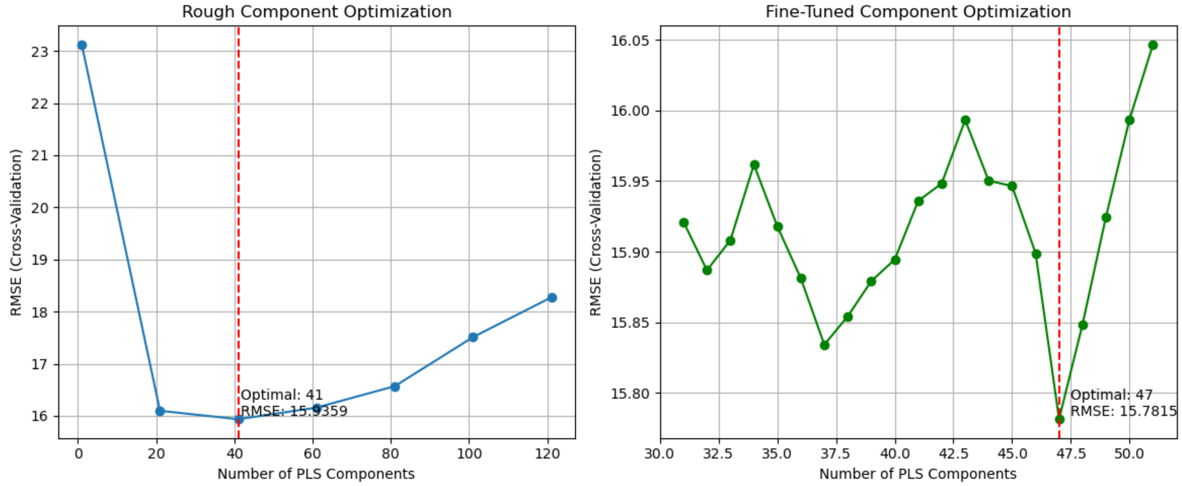
Baseline model (5 P): - Develop a global baseline PLSR model using the *calibration dataset* - (entire VNIR range from 500 nm to 2499 nm in steps of 2 nm) - *without* applying any *spectral preprocessing*. - The target variable is soil organic carbon (SOC). - Perform *internal optimization* to *determine* 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.

4.4 Finding optimal number of components

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

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

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

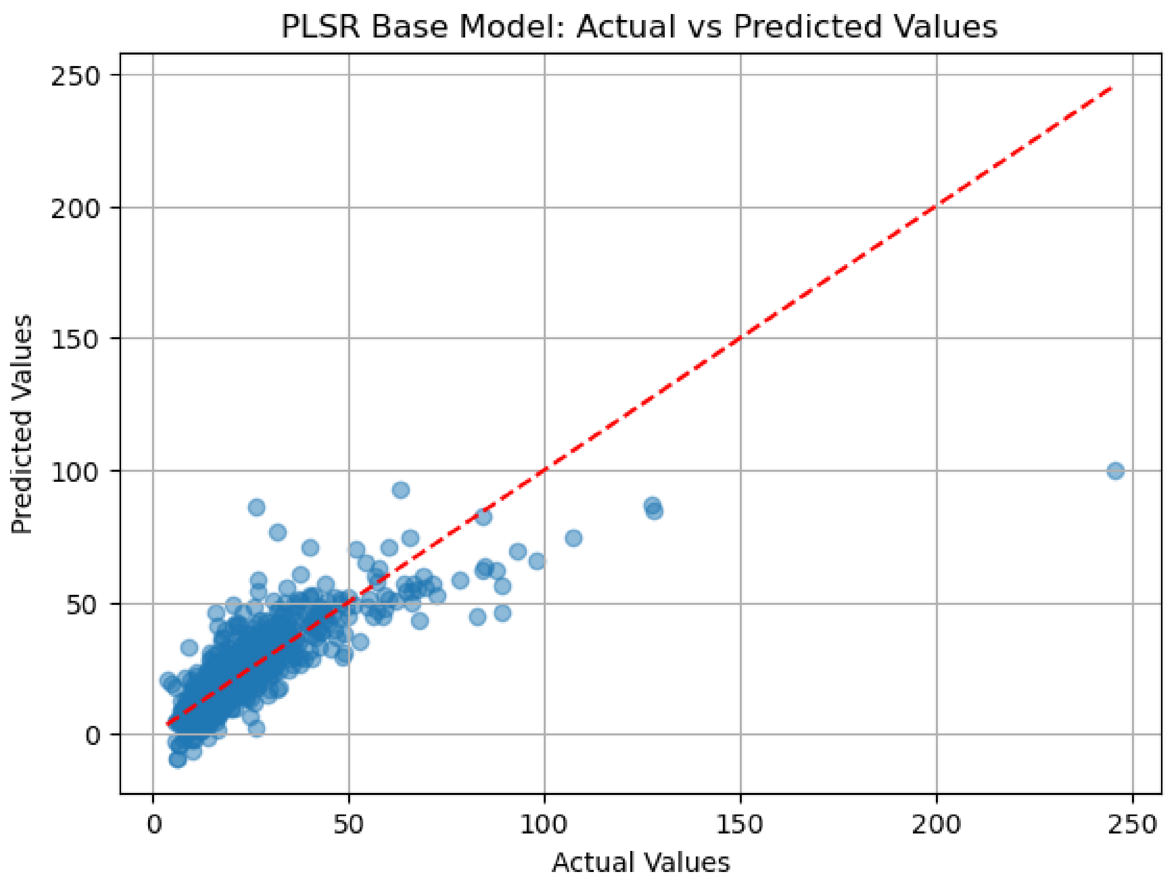



```
Optimal number of components: 47
```

4.5 Evaluating Base Model

```
plsr_base_model = PLSRegression(n_components=plsr_base_components["optimal_n"])
plsr_base_model.fit(X_train, y_train)

plsr_base_eval = own_functions.evaluate_model(plsr_base_model,
                             X_test=X_test,
                             y_test=y_test,
                             print_metrics=True,
                             show_plot=True,
                             plot_kwargs={'model_name': 'PLSR Base Model'}
                             )
```

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



4.6 2. Overview for Baseline Model Development (5 P)

4.6.1 Baseline Model

We have considered or rather its part of task to consider a baseline Partial Least Squares Regression (PLSR), in this regard we have developed the model using the **calibration dataset** (entire VNIR range from 500 nm to 2498 nm in steps of 2 nm) without applying any spectral preprocessing. The target variable for this model is **Soil Organic Carbon (SOC)** which is clearly explained in overview of Task 1.

4.6.2 Internal Optimization for PLS Components

- **Objective:** Determine the optimal number of latent PLS variables as the only hyperparameter , no of latent component to enhance model performance.

- **Process:** The optimization was conducted in two stages to efficiently identify the optimal number of PLS components:

  1. **Coarse Optimization:** Initially, a broader search was performed with a step size of 20 components. This step helped in quickly narrowing down the range of potential optimal values.
  2. **Fine-Tuned Optimization:** Within the range identified in the coarse optimization, a finer search was conducted by evaluating components in the range of -10 to +10 around the initially identified optimum. This refinement allowed for precise determination of the optimal number of components.

- **Overview::** Through this two-stage optimization process, the optimal number of PLS components was determined to be **47**, which was identified during the fine-tuned stage starting from an initial estimate of ***40*** components.

4.6.3 Model Evaluation

- **Application:** The optimized model was applied to the independent test set which was determined in previous task.
- **Metrics Computed:**

Metric	Value
RMSE	10.0547
R^2	0.6552
Bias	-0.1772
RPD	1.7030

- **Visualization:** A scatter plot was generated to visualize the observed vs. predicted values.
- **Performance Assessment:** The model's performance was assessed based on the computed metrics and visual inspection of the scatter plot.

Overview: The baseline PLSR model was optimized and evaluated, providing insights into its predictive performance for SOC using spectral data. The optimal number of PLS components was determined to be **47**, and the model's performance was assessed using various validation metrics.

TASK 3 # 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.

4.7 Strategy 1: Varying Preprocessing Strategy

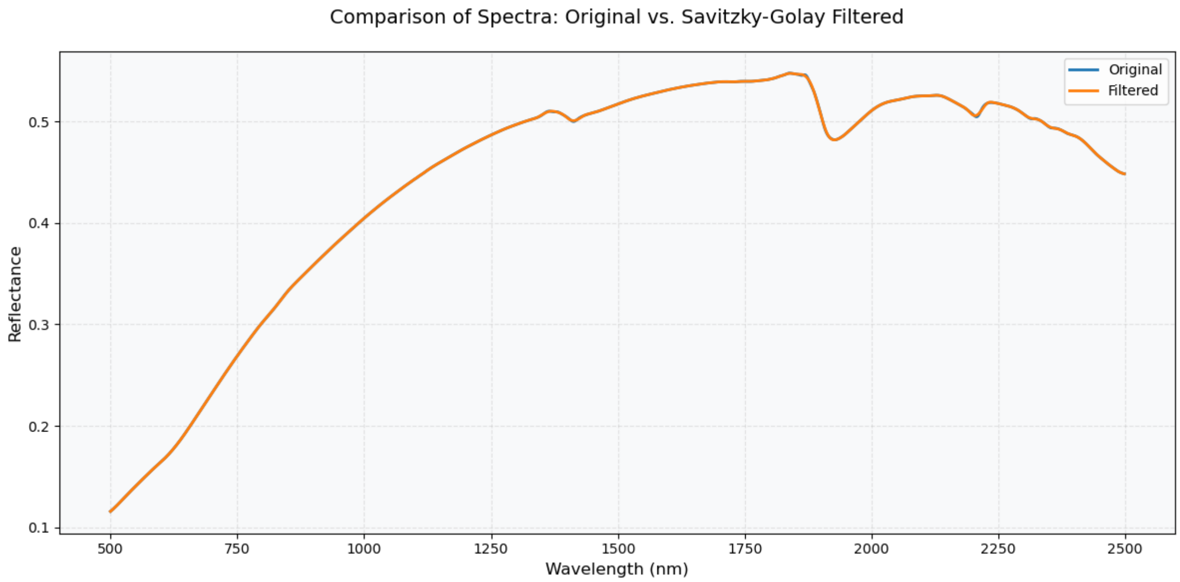
The first strategy to improve the baseline model involves exploring different spectral preprocessing techniques to enhance the model's predictive performance. We tested the following preprocessing methods:

- **Savitzky-Golay Smoothing:**
- **Standard Normal Variate (SNV) Transformation:**
- **Absorbance Transformation:**

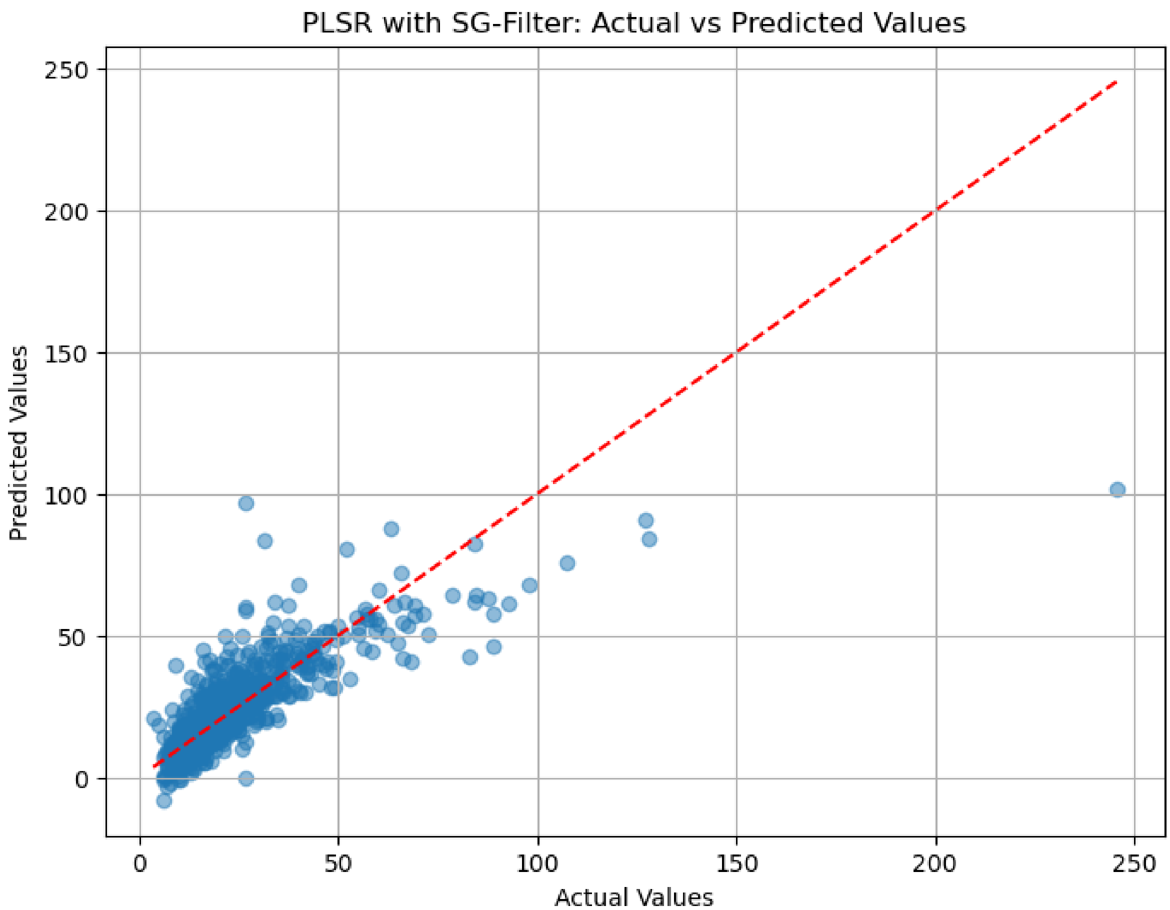
4.7.1 Savitzgy-Golay

```
# Applying Savitzky-Golay filter to calibration and test data
X_train_sg = own_functions.apply_savitzky_golay(X_train, window_length=31, polyorder=4, de
X_test_sg = own_functions.apply_savitzky_golay(X_test, window_length=31, polyorder=4, deri

# Plot spectra comparison
own_functions.plot_spectra_comparison(
    X_train[2],
    X_train_sg[2],
    wavelengths=range(500, 2500, 2),
    labels=['Original', 'Filtered'],
    title='Comparison of Spectra: Original vs. Savitzky-Golay Filtered',
)

plsr_sgolay_components = own_functions.optimize_pls_components(X_train=X_train_sg,
                                        y_train=y_train,
                                        max_components=100,
                                        step=20,
                                        fine_tune=True,
                                        show_progress=False,
                                        plot_results=False
                                        )

plsr_sg_model = PLSRegression(n_components=plsr_sgolay_components["optimal_n"])
plsr_sg_model.fit(X_train_sg, y_train)
```

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




```
Optimal number of components: 37
Root Mean Squared Error (RMSE): 10.1694
R²: 0.6473
Bias: -0.1207
RPD: 1.6838
```

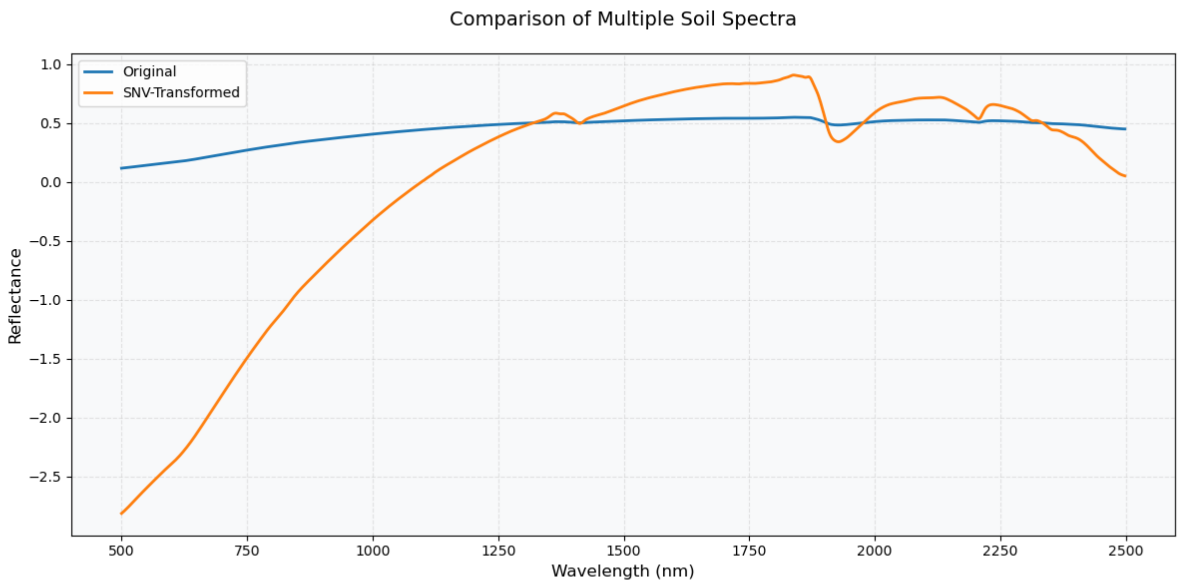


4.7.2 Standard Normal Variate

```
# Applying Savitzky-Golay filter to calibration and test data
X_train_snv = own_functions.standard_normal_variate(X_train)
X_test_snv = own_functions.standard_normal_variate(X_test)
```

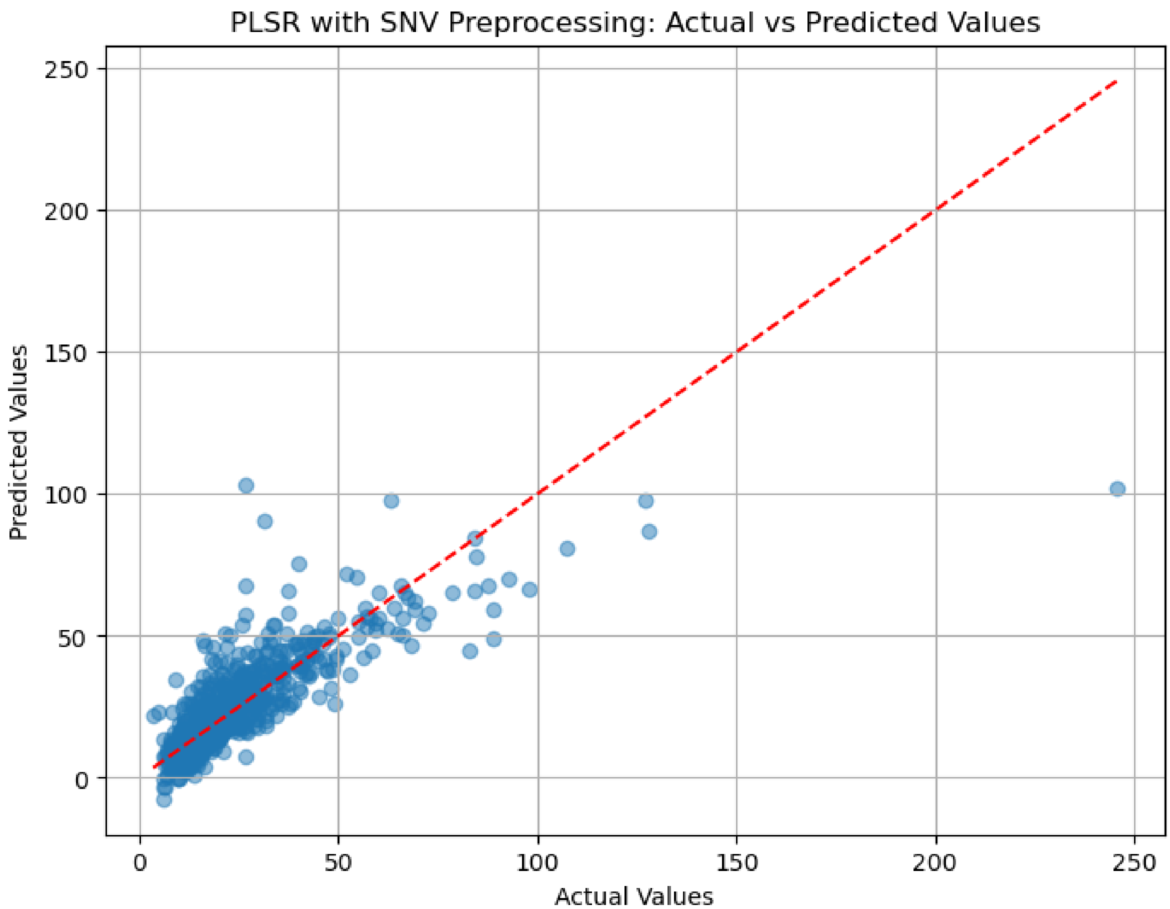
```
# Example usage with multiple spectra:
own_functions.plot_spectra_comparison(
    X_train[2],
    X_train_snv[2],
    wavelengths=range(500, 2500, 2),
    labels=['Original', 'SNV-Transformed'],
    title='Comparison of Multiple Soil Spectra'
)
```

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

plsr_snv_model = PLSRegression(n_components=plsr_snv_components["optimal_n"])
plsr_snv_model.fit(X_train_snv, y_train)

plsr_snv_eval = own_functions.evaluate_model(plsr_snv_model,
                             X_test=X_test_snv,
                             y_test=y_test,
                             print_metrics=True,
                             show_plot=True,
                             plot_kwargs={'model_name': 'PLSR with SNV Preprocessing',
                             'figsize': (8, 6)}
                             )
```




```
Optimal number of components: 40
Root Mean Squared Error (RMSE): 10.1517
R²: 0.6485
```

```
Bias: 0.3009
RPD: 1.6867
```

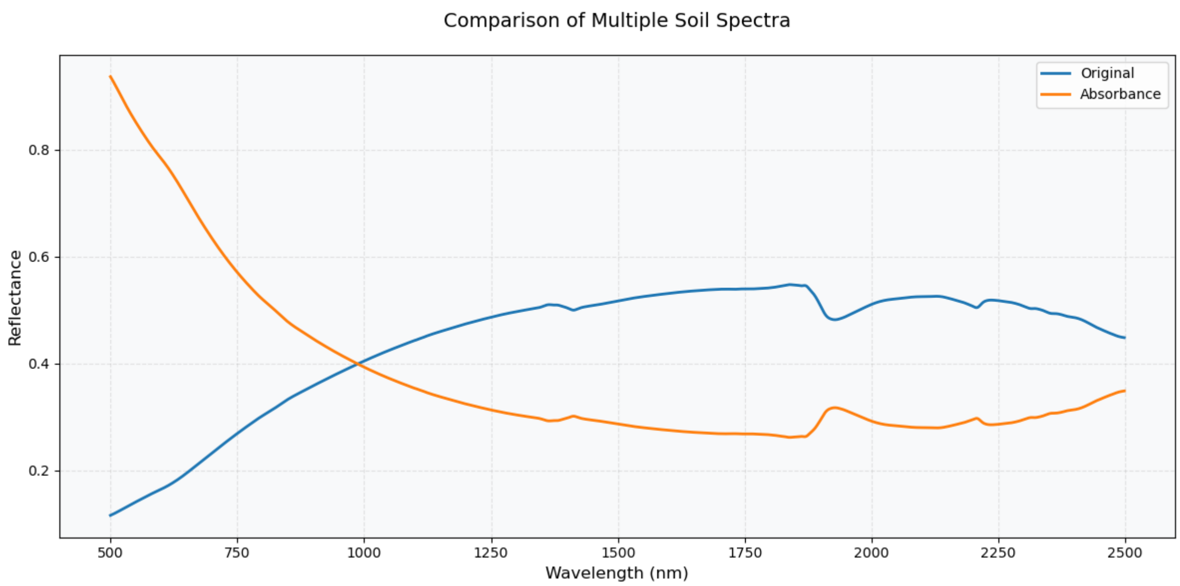


4.7.3 Absorbance

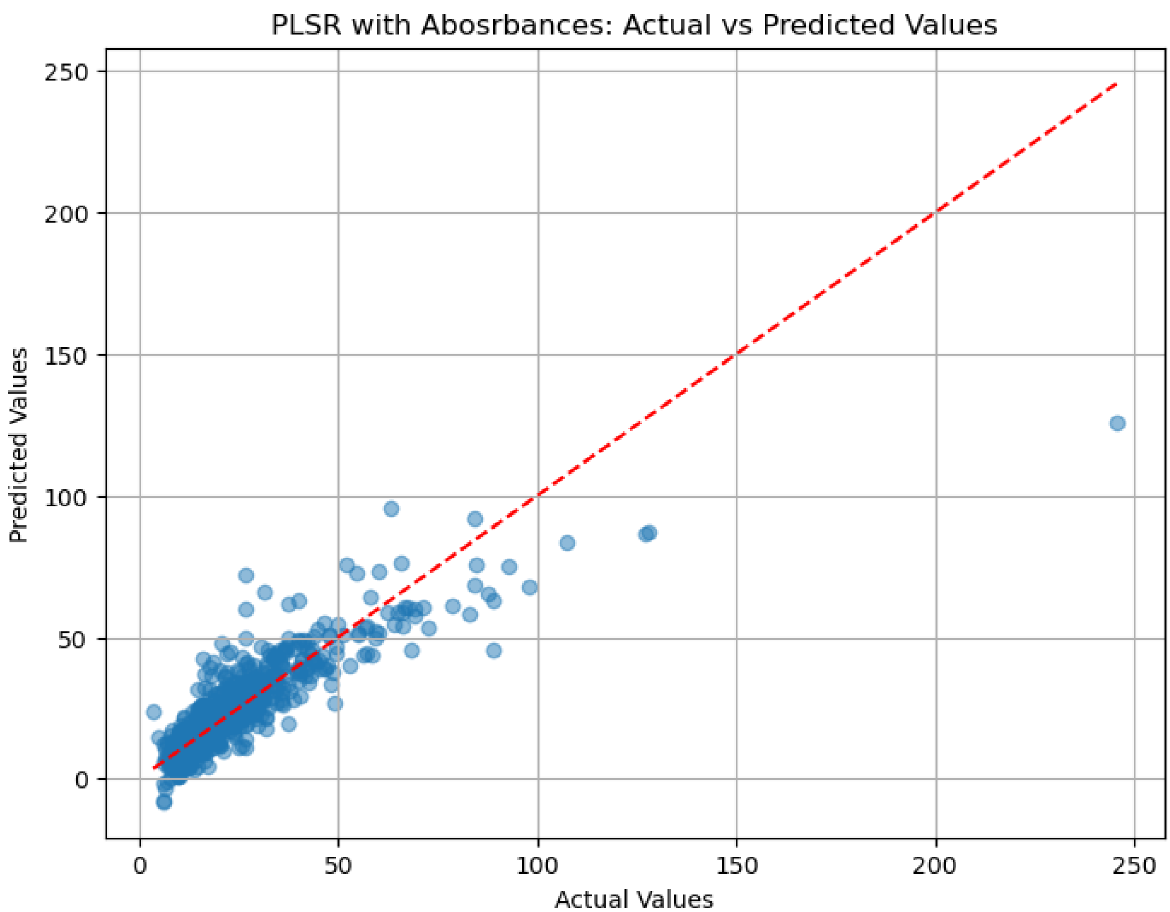
```python
# Calculate pseudo abosrbance
X_train_absorb = np.log10(1/X_train)
X_test_absorb = np.log10(1/X_test)
```

```python
# Plot Spectra
own_functions.plot_spectra_comparison(
    X_train[2],
    X_train_absorb[2],
    wavelengths=range(500, 2500, 2),
```

```
    labels=['Original', 'Absorbance'],
    title='Comparison of Multiple Soil Spectra'
)

plsr_absorb_components = own_functions.optimize_pls_components(X_train=X_train_absorb,
                                      y_train=y_train,
                                      max_components=100,
                                      step=20,
                                      fine_tune=True,
                                      show_progress=False,
                                      plot_results=False
                                      )


plsr_absorb_model = PLSRegression(n_components=plsr_absorb_components["optimal_n"])
plsr_absorb_model.fit(X_train_absorb, y_train)

plsr_absorb_eval = own_functions.evaluate_model(plsr_absorb_model,
                             X_test=X_test_absorb,
                             y_test=y_test,
                             print_metrics=True,
                             show_plot=True,
                             plot_kwargs={'model_name': 'PLSR with Abosrbances',
                             'figsize': (8, 6)}
                             )
```



```
Optimal number of components: 48
Root Mean Squared Error (RMSE): 8.6069
R²: 0.7474
Bias: 0.1506
RPD: 1.9895
```



Please not that in first plot, it is absorbance instead of reflectance.

4.8 Strategy 2: Testing Different Models

Our next strategy is to test different models. All models will be trained using the absorbance transformed data, as it led to the best performance so far.

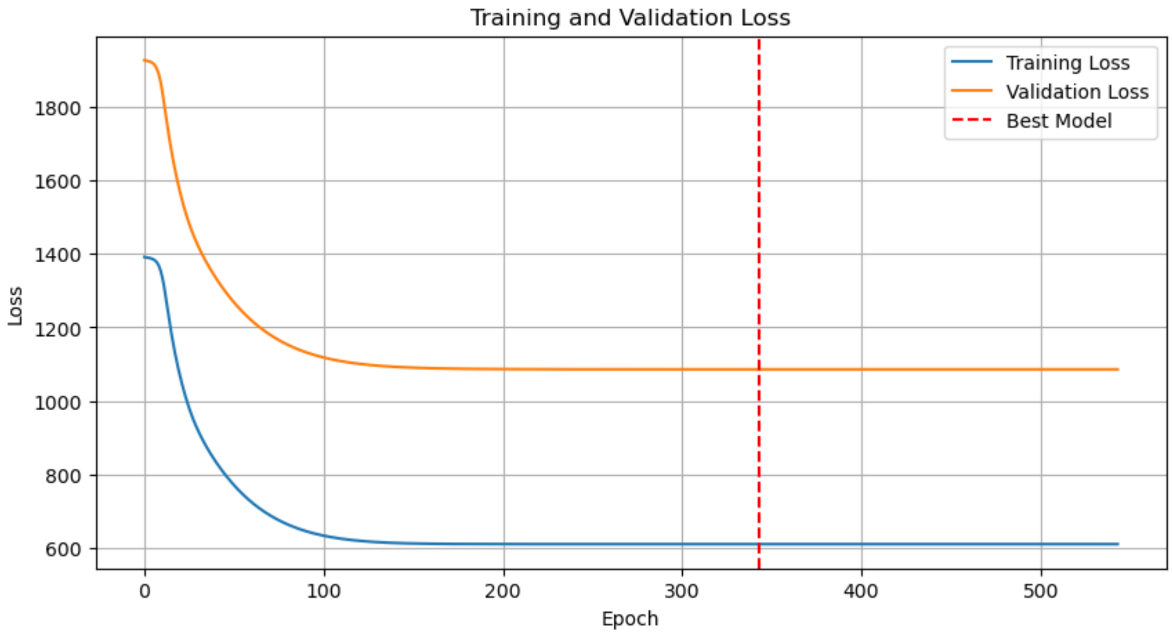
We will test the following models: - **Long Short-Term Memory (LSTM)** - **PLSR + LSTM** - **PLSR + AutoMl (Autogluon)**

4.8.1 LSTM

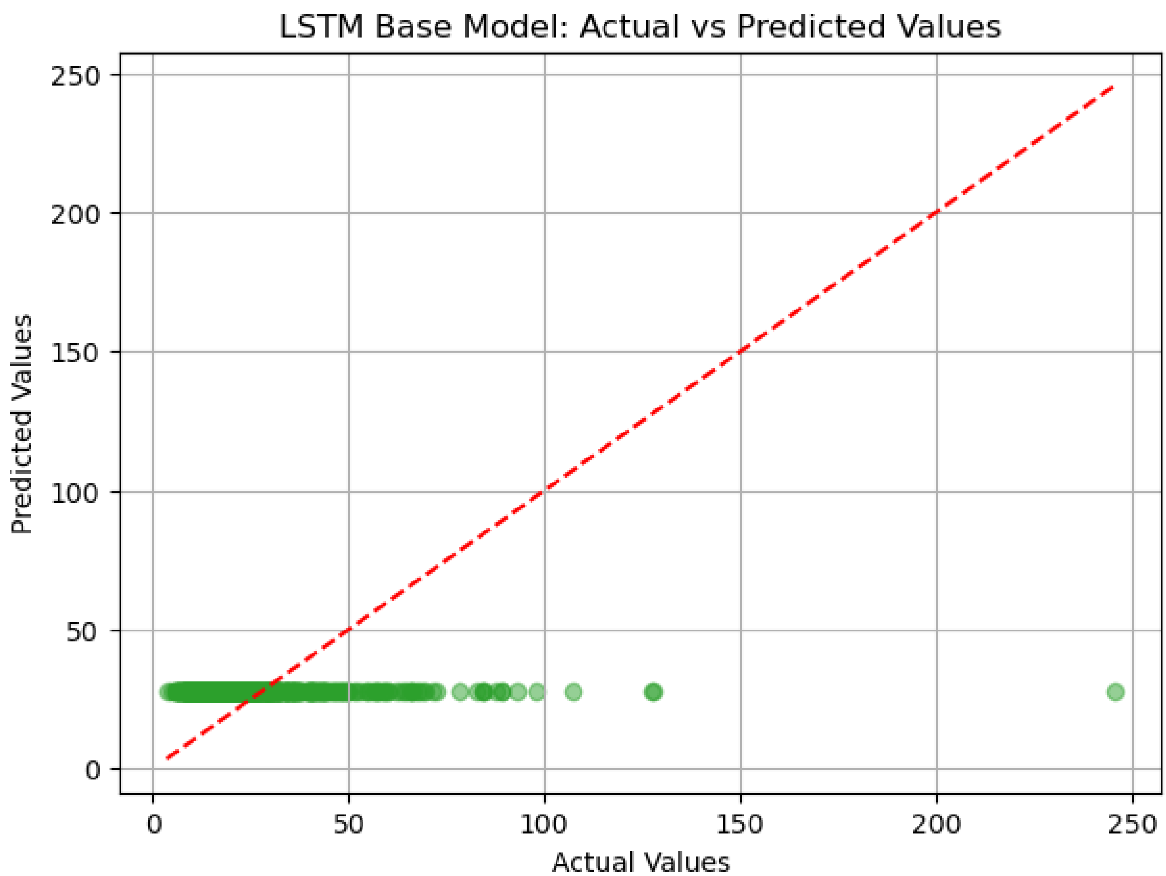
We do a simple split of our calibration (training) data into training and validation using 20 % for validation. We us this validation set to stop training and select the best model (best epoch). The same strategay was applied in all following LSTM applications.

```
# drop rate 0.2 best
# First split training data into train and validation sets
X_train_final, X_val, y_train_final, y_val = train_test_split(X_train_absorb, y_train,
                                                              test_size=0.2,
                                                              random_state=110)


# Training enhanced LSTM model
LSTM_base_model = own_functions.train_lstm(
    X_train=X_train_final,
    X_val=X_val,
    X_test=X_test,
    y_train=y_train_final,
    y_val=y_val,
    y_test=y_test,
    hidden_size=256,
    num_layers=5,
    num_epochs=3000,
    learning_rate=0.001,
    patience=200,  # Early stopping patience
    dropout=0.2
)



# Evaluate LSTM model
_ = own_functions.evaluate_model(LSTM_base_model,
                             X_test=X_test_absorb, y_test=y_test,
                             print_metrics=True, show_plot=True,
                             plot_kwargs={'model_name': 'LSTM Base Model',
                                          'color': "#2ca02c"}
                             )
```

```
Early stopping triggered at epoch 544
tensor(327.5641)
```



Root Mean Squared Error (RMSE): 18.0987
R^2: -0.1172
Bias: 5.8615
RPD: 0.9461



4.8.2 LSTM with PLSR components

Next we will use the PLSR model trained on the absorbance transformed data to extract the latent variables and use them as input to the LSTM model. This is done to see if the latent variables extracted by the PLSR model can be used to improve the performance of the LSTM model.

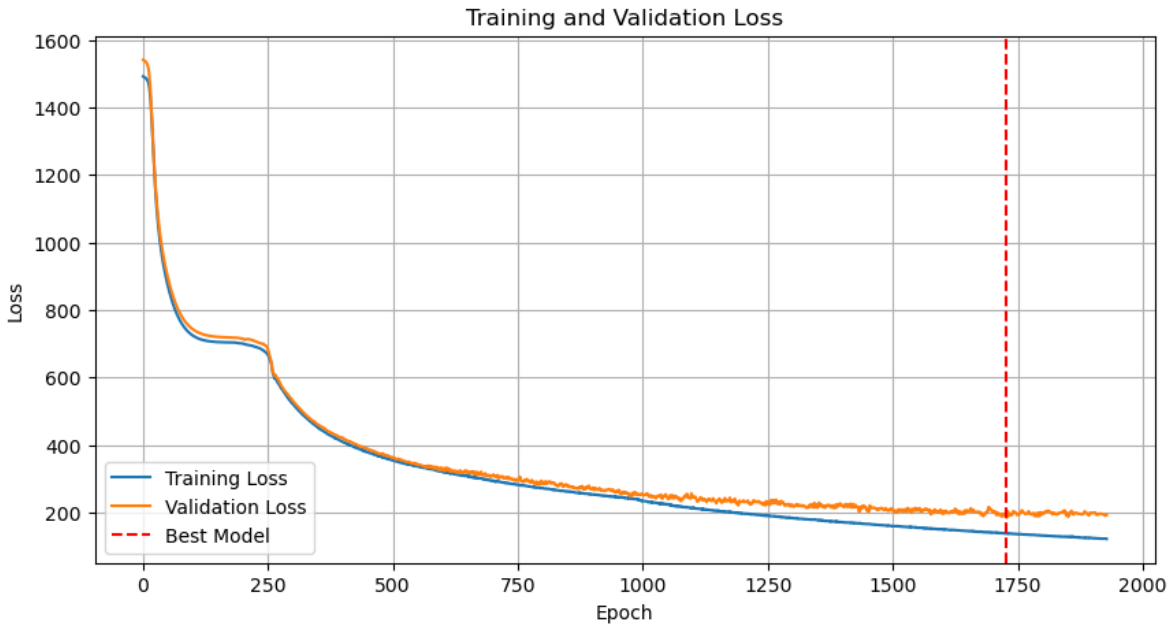
```
# Fit PLSR on the training data and transform the training set
X_train_pls = plsr_absorb_model.transform(X_train_absorb)

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

# Show number of components
print(f"PLSR components: {plsr_absorb_model.n_components}")
```

```python
print(f"Shape of transformed training data: {X_train_pls.shape}")
```

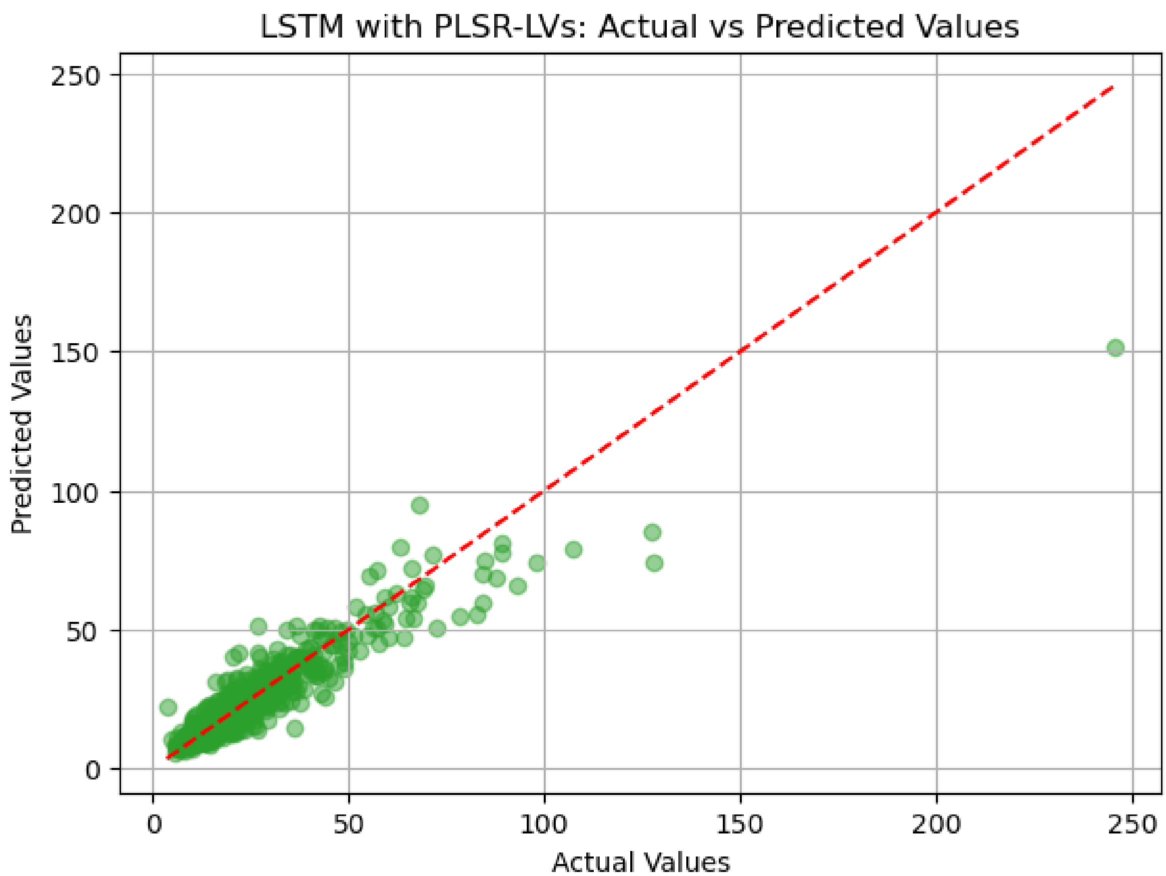
```
PLSR components: 48
Shape of transformed training data: (1964, 48)
```

```python
# First split training data into train and validation sets
X_train_final, X_val, y_train_final, y_val = train_test_split(X_train_pls, y_train,
                                                              test_size=0.2,
                                                              random_state=42)

# Training enhanced LSTM model
LSTM_plsr_model= own_functions.train_lstm(
    X_train=X_train_final,
    X_val=X_val,
    X_test=X_test_pls,
    y_train=y_train_final,
    y_val=y_val,
    y_test=y_test,
    hidden_size=256,
    num_layers=5,
    num_epochs=3000,
    learning_rate=0.001,
    patience=200,  # Early stopping patience
    dropout=0.2,
    show_training=False
)


lstm_plsr_eval = own_functions.evaluate_model(LSTM_plsr_model,
                             X_test=X_test_pls, y_test=y_test,
                             print_metrics=True, show_plot=True,
                            plot_kwargs={'model_name': 'LSTM with PLSR-LVs',
                                          'color': "#2ca02c"}
                             )
```

```
Early stopping triggered at epoch 1929
tensor(42.8820)
```



Root Mean Squared Error (RMSE): 6.5484
R^2: 0.8537
Bias: -0.7114
RPD: 2.6149



4.8.3 Plsr + AutoML

Using the latent variables from the plsr model led to a significant performance increase in the LSTM model. Therefore we will use the latent variables from the PLSR model also to train an AutoML model using Autogluon. Autogluon is a AutoML library that can be used to train a wide range of models. It will apply 15 different models, including but not limited to: - Random Forest - XGBoost - CatBoost - K-Nearest Neighbors - Neural Networks - Weighted Ensemble ***

For autogluon we need to setup our data in dataframes.

```
# Convert numpy arrays to DataFrame with wavelength columns
wavelengths = range(500, 2500, 2)  # Your wavelength range
train_df = pd.DataFrame(X_train_pls, columns=range(1, X_train_pls.shape[1] + 1))
test_df = pd.DataFrame(X_test_pls, columns=range(1, X_train_pls.shape[1] + 1))
```

```python
# Add target variable
train_df['SOC'] = y_train
test_df['SOC'] = y_test
```

Now we can simply run the TabularPredictor.

```python
# More advanced configuration with safety measures
auto_predictor = TabularPredictor(
    label='SOC',
    problem_type="regression",
    eval_metric='root_mean_squared_error',
).fit(
    train_df,
    presets='best_quality',
    num_gpus=0,
    num_cpus=1,
    memory_limit='auto',
    auto_stack=False,
    verbosity=0
)
```

We are provided with a model leaderboard.

```python
# Get leaderboard
auto_leaderbord = auto_predictor.leaderboard(test_df)

# Show leaderboard
print("\nModel Leaderboard:")
print(auto_leaderbord[1:10,["model", "score_test",
                           "fit_time","pred_time_test"]])
```

```
predictor path is c:\Users\luis_\Desktop\Alles\Uni\Leipzig\WS_24_25\spectroscopy\final_proje

Model Leaderboard:
                  model  score_test  score_val             eval_metric  \
0   NeuralNetFastAI_r191   -6.465653  -7.858265  root_mean_squared_error
1    WeightedEnsemble_L2   -6.565789  -7.667403  root_mean_squared_error
2          CatBoost_r137   -7.100844 -10.370113  root_mean_squared_error
3               CatBoost   -7.319287  -9.634744  root_mean_squared_error
4          CatBoost_r177   -7.464913  -9.398295  root_mean_squared_error
5        NeuralNetFastAI   -7.833119  -8.503627  root_mean_squared_error
6           CatBoost_r13   -8.104972  -8.996269  root_mean_squared_error
```

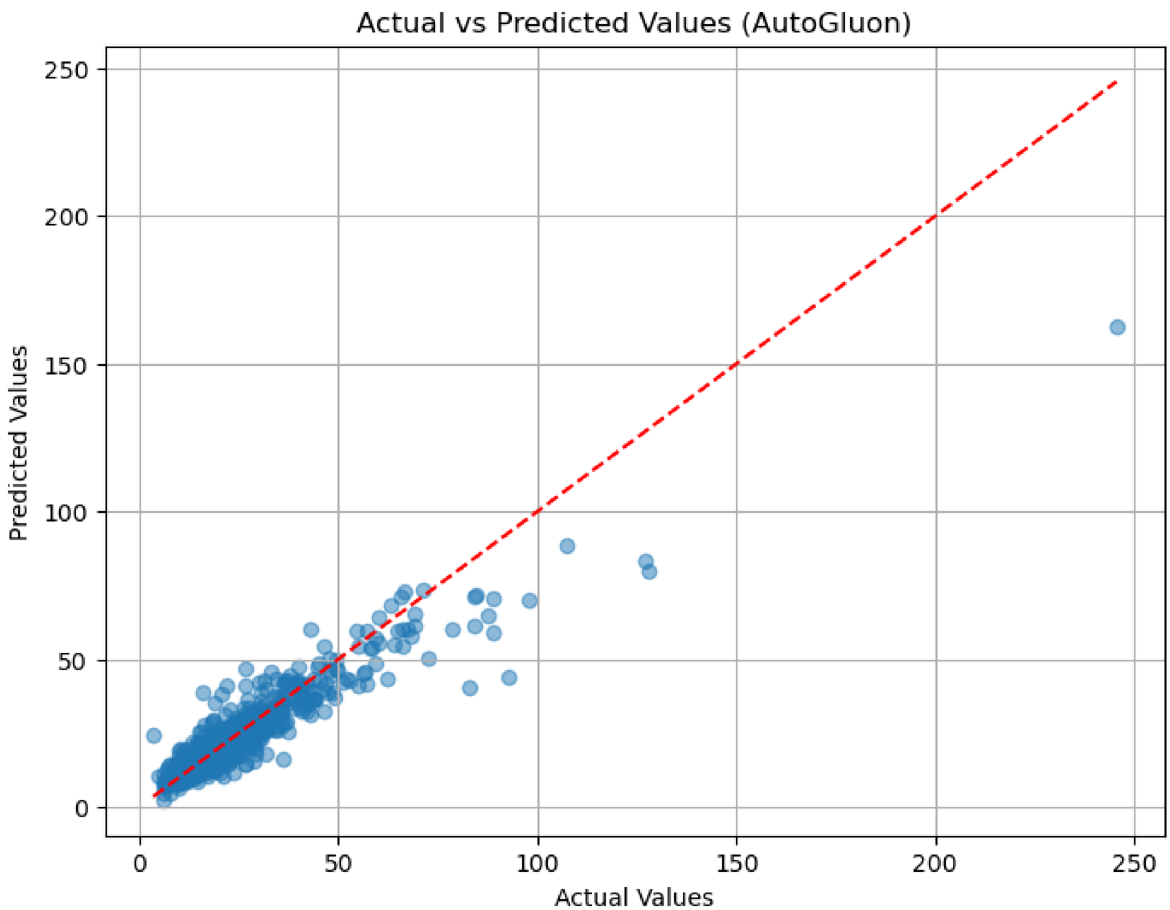
```
7           CatBoost_r9   -8.638408  -9.252907  root_mean_squared_error
8               XGBoost   -8.678744 -11.231818  root_mean_squared_error
9           XGBoost_r33   -9.264493 -10.793384  root_mean_squared_error
10      RandomForestMSE  -10.015084 -12.752302  root_mean_squared_error
11       ExtraTrees_r42  -10.321236 -12.743379  root_mean_squared_error
12        ExtraTreesMSE  -10.338205 -12.501554  root_mean_squared_error
13       KNeighborsUnif  -11.313581 -14.232448  root_mean_squared_error
14  NeuralNetFastAI_r102  -11.314371 -13.623935  root_mean_squared_error
15       KNeighborsDist  -11.326214 -14.656925  root_mean_squared_error

    pred_time_test  pred_time_val     fit_time  pred_time_test_marginal  \
0         0.062636       0.025176     9.271056                 0.062636
1         0.193922       0.033624  1533.309282                 0.021792
2         0.080354       0.006184   367.305470                 0.080354
3         0.063215       0.005027   482.195087                 0.063215
4         0.027581       0.006836    86.732648                 0.027581
5         0.040360       0.002593     3.638255                 0.040360
6         0.109494       0.008448  1524.012893                 0.109494
7         0.057354       0.025461  1030.988951                 0.057354
8         0.031350       0.008112    11.257859                 0.031350
9         0.109050       0.012634    55.511277                 0.109050
10        0.215906       0.480414     4.535789                 0.215906
11        0.216460       0.122088     1.322315                 0.216460
12        0.212351       0.099897     1.110185                 0.212351
13        0.036509       0.023081     0.022188                 0.036509
14        0.130343       0.071021    15.547421                 0.130343
15        0.029869       0.024546     0.022627                 0.029869

    pred_time_val_marginal  fit_time_marginal  stack_level  can_infer  \
0                 0.025176           9.271056            1       True
1                 0.000000           0.025332            2       True
2                 0.006184         367.305470            1       True
3                 0.005027         482.195087            1       True
4                 0.006836          86.732648            1       True
5                 0.002593           3.638255            1       True
6                 0.008448        1524.012893            1       True
7                 0.025461        1030.988951            1       True
8                 0.008112          11.257859            1       True
9                 0.012634          55.511277            1       True
10                0.480414           4.535789            1       True
11                0.122088           1.322315            1       True
12                0.099897           1.110185            1       True
13                0.023081           0.022188            1       True
```

```
14                0.071021          15.547421              1       True
15                0.024546           0.022627              1       True

    fit_order
0           9
1          16
2          13
3           4
4           8
5           6
6          15
7          10
8           7
9          11
10          3
11         12
12          5
13          1
14         14
15          2
```

We automatically use the best predictor, using the predict function.

```python
# Get predictions and evaluate
y_pred_auto = auto_predictor.predict(test_df.drop(columns=['SOC']))

# Evaluate using your existing function
autogluon_eval = own_functions.evaluate_model(
    auto_predictor,
    X_test=test_df.drop(columns=['SOC']),
    y_test=test_df['SOC'],
    print_metrics=True,
    show_plot=True,
    plot_kwargs={'model_name': 'AutoGluon', 'figsize': (8, 6), 'color': '#d62728'}
)
```

```
Root Mean Squared Error (RMSE): 6.5658
R²: 0.8530
Bias: -0.5860
RPD: 2.6080
```



4.9 Stragtegy 3: Testing auxillary data

Our 3rd strategy is to test the effect of adding auxiliary data to the model. We will use the best performing model from the previous strategies, the PLSR + LSTM using absorbance transformed data.

We will use the following auxiliary data:

- **DLR SoilSuite: spectral composite data**
- **ISRIC SoilGrids: soil information**

4.9.1 DLR SoilSuite Spectral Data

The DLR SoilSuite dataset is a recently released collection of different image data products that provide information about the spectral and statistical properties of European soils and other bare surfaces such as rocks. It utilizes Sentinel-2 data to generate composite images based on observations recorded between January 2018 and December 2022 across Europe.

Properties:

- 20 meter resolution
- 10 Sentinel-2 bands (B02, B03, B04, B05, B06, B07, B08, B08a, B11, B12)
- 5-year composite (2018-2022)
- Continental coverage (Europe)
- Cloud cover filter: excludes scenes with >80% cloud cover
- Sun elevation filter: excludes scenes with <20 degrees elevation

Used variables: - MREF: Mean reflactances excluding snow and cloud pixels - SDREF: Standard deviation of reflectances, indicating spectral variability

We also used MREF to calculate a suite of 10 spectral vegetation and soil indices to add to our data: - **Vegetation Indices**: NDVI, GNDVI, EVI, SAVI, OSAVI - **Soil Indices**: BI, NDSoI, MBI, NSDS, NSDSI1

Note: The product also includes bands on bare surface properties, these were not used however as for many points in the LUCAS dataset, no data was available.

For more Information go to: https://geoservice.dlr.de/eoc/ogc/stac/v1/collections/S2-soilsuite-europe-2018-2022-P5Y

Read data.

```
dlr_data = gpd.read_parquet("data/auxiliary_data_results_full_updated.parquet")

dlr_data
```

	y	x	spatial_ref	time	MREF_B02	MREF_B03	MREF_B04	MREF_
0	2548010.0	3902010.0	3035	2018-03-01	567	927	946	1516
1	2545990.0	3908010.0	3035	2018-03-01	399	688	706	1206
2	2555990.0	3907990.0	3035	2018-03-01	556	936	937	1369
3	2537990.0	3899990.0	3035	2018-03-01	232	466	446	934
4	2577990.0	3896010.0	3035	2018-03-01	431	766	805	1357
...	...	...	...	...	...	...	...	...
2800	2636010.0	3598010.0	3035	2018-03-01	446	753	731	1300
2801	2869990.0	3372010.0	3035	2018-03-01	298	620	463	1110
2802	2620010.0	3788010.0	3035	2018-03-01	485	803	833	1367

	y	x	spatial_ref	time	MREF_B02	MREF_B03	MREF_B04	MREF_
2803	2795990.0	3357990.0	3035	2018-03-01	498	752	882	1234
2804	2910010.0	3566010.0	3035	2018-03-01	485	779	916	1301

Select the appropriate variables.

```
dlr_measurements = ["MREF_B02", "MREF_B03", "MREF_B04", "MREF_B05", "MREF_B06", "MREF_B07"
                        "MREF-STD_B02", "MREF-STD_B03", "MREF-STD_B04", "MREF-STD_B05", "M
                        ]
```

Two of our Lucas samples have incorrect coordinates with both lattitude and longitude beeing 88.88888. These samples had to be excluded. Also we need to find the correct indices to add the auxillary data to the Lucas data.

```
# check if any lon is 88.8888 and lat is 88.8888
target_raw[target_raw['GPS_LONG'].isin([88.888888])]
```

	Unnamed: 0	SAMPLE_ID	CLAY	SILT	SAND	SOC	CaCO3	N	P	K	CEC	GPS
513	514	10589	13.0	43.0	44.0	69.9	1	2.4	0.0	55.0	15.3	X
935	936	11065	34.0	50.0	17.0	21.6	7	2.4	30.5	288.6	28.4	X

Get valid indices and select the auxillary data.

```
original_ks_indices = ks_indices
original_test_indices = test_indices

# Create a mapping from original indices to rows in the auxiliary data
point_to_row = dict(zip(dlr_data['point_index'], dlr_data.index))

# Filter indices to only include those with data in dlr_data
valid_ks_indices = [idx for idx in original_ks_indices if idx in point_to_row]
valid_test_indices = [idx for idx in original_test_indices if idx in point_to_row]

# Map original indices to row positions in dlr_data
mapped_ks_indices = [point_to_row[idx] for idx in valid_ks_indices]
mapped_test_indices = [point_to_row[idx] for idx in valid_test_indices]

# Now use these mapped indices to select data
dlr_train_ks = dlr_data.iloc[mapped_ks_indices]
```

```
dlr_test_ks = dlr_data.iloc[mapped_test_indices]
```

```
Original indices: 1964 training, 843 test
Valid indices with aux data: 1962 training, 843 test
```

Calculate vegetation and soil indices.

```
# Calculate indices
indices_train_ks = own_functions.compute_indices(dlr_train_ks)
indices_test_ks = own_functions.compute_indices(dlr_test_ks)
```

Combine DLR spectral and indeices data.

```
# Combine indices with original data
dlr_train_ks = pd.concat([dlr_train_ks[dlr_measurements], indices_train_ks], axis=1)
dlr_test_ks = pd.concat([dlr_test_ks[dlr_measurements], indices_test_ks], axis=1)

# Print shapes
print(f"Original indices: {len(original_ks_indices)} training, {len(original_test_indices)
print(f"Valid indices with aux data: {len(valid_ks_indices)} training, {len(valid_test_ind
```

```
# Convert to numpy arrays
dlr_train_ks = dlr_train_ks.values
dlr_test_ks = dlr_test_ks.values

# Print number of rows containing NaN values
print(f"NaN values in DLR Train data: {np.isnan(dlr_train_ks).sum()}")

dlr_train_ks = np.nan_to_num(dlr_train_ks, nan=0, posinf=0, neginf=0)
dlr_test_ks = np.nan_to_num(dlr_test_ks, nan=0, posinf=0, neginf=0)

print(f"Auxillary DLR Train data shape: {dlr_train_ks.shape}")
print(f"Auxillary DLR Test data shape: {dlr_test_ks.shape}")
```

```
NaN values in DLR Train data: 48
Auxillary DLR Train data shape: (1962, 38)
Auxillary DLR Test data shape: (843, 38)
```

Select the matching datapoints from the LUCAS-Data. Excluding two samples from the training data with incorrect coordinates.

```
# Map from original index to position in X_train_absprb
original_to_train_pos = {orig_idx: train_pos for train_pos, orig_idx in enumerate(ks_indic

# Find the positions in X_train_absprb that correspond to valid_ks_indices
train_absprb_positions = []
for idx in valid_ks_indices:
    if idx in original_to_train_pos:
        train_absprb_positions.append(original_to_train_pos[idx])

# Now use these positions to select from X_train_absprb
X_train_absprb_dlr = X_train_absorb[train_absprb_positions, :]
y_train_dlr = y_train[train_absprb_positions]

# Do the same for test data
original_to_test_pos = {orig_idx: test_pos for test_pos, orig_idx in enumerate(test_indice

test_absprb_positions = []
for idx in valid_test_indices:
    if idx in original_to_test_pos:
        test_absprb_positions.append(original_to_test_pos[idx])

X_test_absprb_dlr = X_test_absorb[test_absprb_positions, :]
y_test_dlr = y_test[test_absprb_positions]

# Print shapes
print(f"Original data shapes: {X_train_absorb.shape}, {X_test_absorb.shape}")
print(f"X_train_absorb_dlr shape: {X_train_absprb_dlr.shape}")
print(f"X_test_absprb_dlr shape: {X_test_absprb_dlr.shape}")
print(f"y_train_dlr shape: {y_train_dlr.shape}")
print(f"y_test_dlr shape: {y_test_dlr.shape}")
```

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

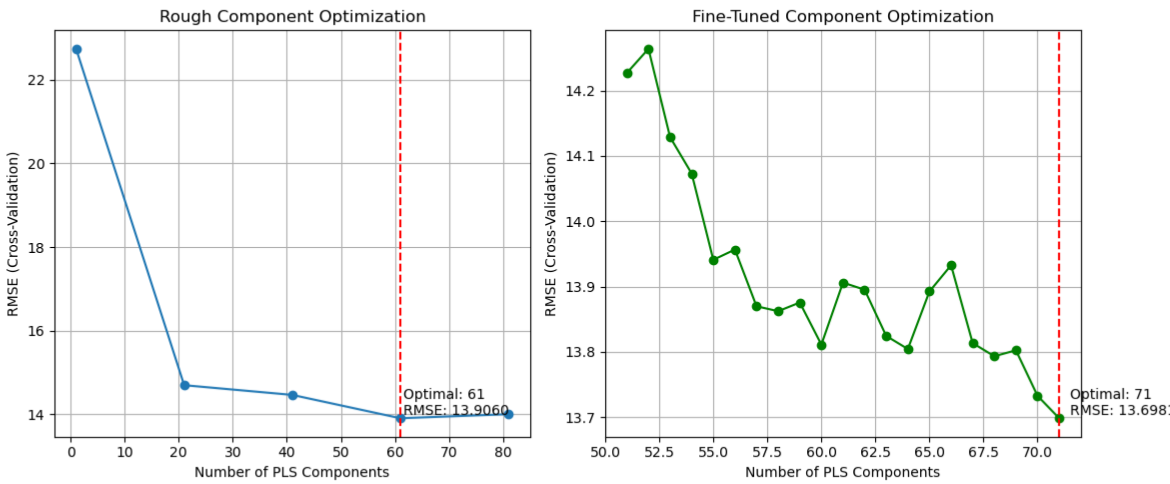
```
# Add the auxiliary data to the PLSR transformed data
X_train_combined = np.hstack((X_train_absprb_dlr, dlr_train_ks))
X_test_combined = np.hstack((X_test_absprb_dlr, dlr_test_ks))

# Print shapes
print(f"X_train_combined shape: {X_train_combined.shape}")
print(f"X_test_combined shape: {X_test_combined.shape}")
```

```
X_train_combined shape: (1962, 1038)
X_test_combined shape: (843, 1038)
```

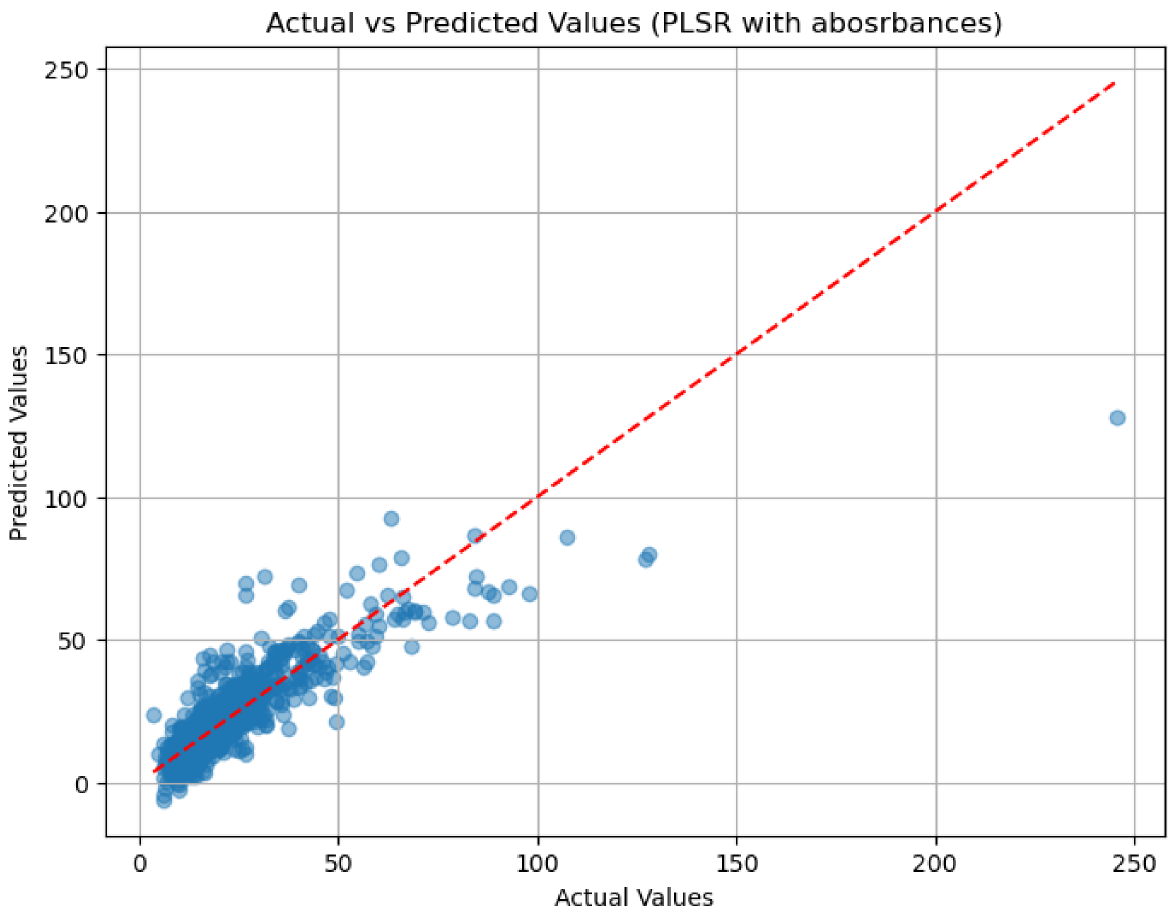
Fit PLSR on the combined data.

```python
plsr_dlr_components = own_functions.optimize_pls_components(X_train=X_train_combined,
                                      y_train=y_train_dlr,
                                      max_components=100,
                                      step=20,
                                      fine_tune=True,
                                      show_progress=False,
                                      plot_results=False
                                      )


plsr_absorb_dlr_model = PLSRegression(plsr_dlr_components["optimal_n"])
plsr_absorb_dlr_model.fit(X_train_combined, y_train_dlr)

plsr_absorb_dlr_eval = own_functions.evaluate_model(plsr_absorb_dlr_model,
                             X_test=X_test_combined,
                             y_test=y_test_dlr,
                             print_metrics=True,
                             show_plot=True,
                             plot_kwargs={'model_name': 'PLSR with abosrbances',
                             'figsize': (8, 6), 'color': '#2ca02c'}
                             )
```

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

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




```
Root Mean Squared Error (RMSE): 8.9545
R²: 0.7265
Bias: 0.2841
RPD: 1.9123
```



Use Latent Variables from PLSR.

```
# transform the data using plsr_absorb_dlr_model
X_train_plsr_absorb_dlr = plsr_absorb_dlr_model.transform(X_train_combined)
X_test_plsr_absorb_dlr = plsr_absorb_dlr_model.transform(X_test_combined)
```

Fit LSTM on the latent variables of the combined data.

```
# First split training data into train and validation sets
X_train_final, X_val, y_train_final, y_val = train_test_split(X_train_plsr_absorb_dlr, y_t
                                                              test_size=0.2,
                                                              random_state=42)


# Training enhanced LSTM model
LSTM_plsr_dlr_model= own_functions.train_lstm(
```

```python
    X_train=X_train_final,
    X_val=X_val,
    X_test=X_test_plsr_absorb_dlr,
    y_train=y_train_final,
    y_val=y_val,
    y_test=y_test_dlr,
    hidden_size=256,
    num_layers=5,
    num_epochs=3000,
    learning_rate=0.001,
    patience=200,  # Early stopping patience
    dropout=0.2
)


lstm_plsr_eval = own_functions.evaluate_model(LSTM_plsr_dlr_model,
                             X_test=X_test_plsr_absorb_dlr, y_test=y_test_dlr,
                             print_metrics=True, show_plot=True)
```

```
Epoch [10/3000], Train Loss: 1540.5464, Val Loss: 1257.4043
Epoch [20/3000], Train Loss: 1344.9900, Val Loss: 1036.0135
Epoch [30/3000], Train Loss: 1098.5538, Val Loss: 813.5641
Epoch [40/3000], Train Loss: 986.3824, Val Loss: 711.8333
Epoch [50/3000], Train Loss: 917.4670, Val Loss: 647.8571
Epoch [60/3000], Train Loss: 866.9979, Val Loss: 601.1006
Epoch [70/3000], Train Loss: 830.6962, Val Loss: 567.8345
Epoch [80/3000], Train Loss: 805.1898, Val Loss: 544.7816
Epoch [90/3000], Train Loss: 787.6553, Val Loss: 529.2339
Epoch [100/3000], Train Loss: 775.8893, Val Loss: 519.0675
Epoch [110/3000], Train Loss: 768.2424, Val Loss: 512.6531
Epoch [120/3000], Train Loss: 763.3832, Val Loss: 508.7702
Epoch [130/3000], Train Loss: 760.3764, Val Loss: 506.5227
Epoch [140/3000], Train Loss: 758.4575, Val Loss: 505.1368
Epoch [150/3000], Train Loss: 757.2153, Val Loss: 503.8857
Epoch [160/3000], Train Loss: 755.8500, Val Loss: 499.4169
Epoch [170/3000], Train Loss: 753.3842, Val Loss: 498.8031
Epoch [180/3000], Train Loss: 751.7171, Val Loss: 495.5575
Epoch [190/3000], Train Loss: 749.3278, Val Loss: 493.8198
Epoch [200/3000], Train Loss: 745.2280, Val Loss: 490.5216
Epoch [210/3000], Train Loss: 740.8985, Val Loss: 486.6480
Epoch [220/3000], Train Loss: 731.9058, Val Loss: 475.0211
Epoch [230/3000], Train Loss: 712.8406, Val Loss: 456.1924
```

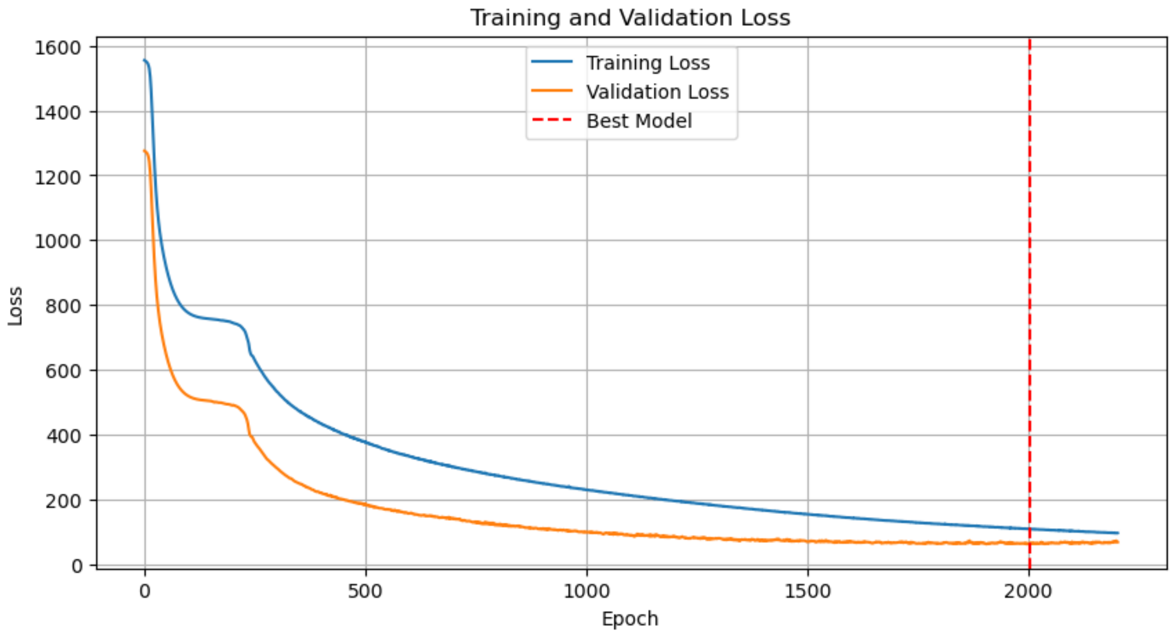
```
Epoch [240/3000], Train Loss: 654.4607, Val Loss: 397.9888
Epoch [250/3000], Train Loss: 630.2448, Val Loss: 376.5938
Epoch [260/3000], Train Loss: 607.6003, Val Loss: 358.4547
Epoch [270/3000], Train Loss: 585.6476, Val Loss: 338.9770
Epoch [280/3000], Train Loss: 566.1488, Val Loss: 322.1868
Epoch [290/3000], Train Loss: 549.3708, Val Loss: 309.4305
Epoch [300/3000], Train Loss: 532.5668, Val Loss: 296.8417
Epoch [310/3000], Train Loss: 519.7497, Val Loss: 284.9230
Epoch [320/3000], Train Loss: 505.5452, Val Loss: 273.9857
Epoch [330/3000], Train Loss: 493.5374, Val Loss: 264.8949
Epoch [340/3000], Train Loss: 484.1887, Val Loss: 256.2636
Epoch [350/3000], Train Loss: 473.1741, Val Loss: 249.6683
Epoch [360/3000], Train Loss: 465.3259, Val Loss: 243.4194
Epoch [370/3000], Train Loss: 456.6234, Val Loss: 236.2714
Epoch [380/3000], Train Loss: 449.8513, Val Loss: 230.1771
Epoch [390/3000], Train Loss: 441.3017, Val Loss: 223.6157
Epoch [400/3000], Train Loss: 433.4195, Val Loss: 220.1736
Epoch [410/3000], Train Loss: 427.6105, Val Loss: 214.7301
Epoch [420/3000], Train Loss: 422.0114, Val Loss: 213.1962
Epoch [430/3000], Train Loss: 414.6510, Val Loss: 206.7473
Epoch [440/3000], Train Loss: 407.7838, Val Loss: 203.2201
Epoch [450/3000], Train Loss: 402.4257, Val Loss: 199.4562
Epoch [460/3000], Train Loss: 396.7086, Val Loss: 196.9667
Epoch [470/3000], Train Loss: 390.8503, Val Loss: 193.0206
Epoch [480/3000], Train Loss: 386.5664, Val Loss: 189.7172
Epoch [490/3000], Train Loss: 381.8917, Val Loss: 187.7487
Epoch [500/3000], Train Loss: 376.5102, Val Loss: 182.8676
Epoch [510/3000], Train Loss: 371.9299, Val Loss: 178.7989
Epoch [520/3000], Train Loss: 366.1584, Val Loss: 175.5018
Epoch [530/3000], Train Loss: 361.6959, Val Loss: 173.6944
Epoch [540/3000], Train Loss: 356.5685, Val Loss: 171.1390
Epoch [550/3000], Train Loss: 351.8480, Val Loss: 169.6963
Epoch [560/3000], Train Loss: 347.9784, Val Loss: 165.1118
Epoch [570/3000], Train Loss: 344.4390, Val Loss: 163.6105
Epoch [580/3000], Train Loss: 340.5009, Val Loss: 161.4051
Epoch [590/3000], Train Loss: 337.0747, Val Loss: 159.3816
Epoch [600/3000], Train Loss: 333.2801, Val Loss: 157.3070
Epoch [610/3000], Train Loss: 329.4747, Val Loss: 155.0859
Epoch [620/3000], Train Loss: 325.9943, Val Loss: 152.1785
Epoch [630/3000], Train Loss: 323.2853, Val Loss: 151.4859
Epoch [640/3000], Train Loss: 319.2904, Val Loss: 148.9192
Epoch [650/3000], Train Loss: 316.3323, Val Loss: 147.7457
Epoch [660/3000], Train Loss: 312.4438, Val Loss: 146.3408
```

```
Epoch [670/3000], Train Loss: 309.1076, Val Loss: 143.9397
Epoch [680/3000], Train Loss: 306.0854, Val Loss: 143.7634
Epoch [690/3000], Train Loss: 303.6172, Val Loss: 142.2303
Epoch [700/3000], Train Loss: 300.4765, Val Loss: 141.3450
Epoch [710/3000], Train Loss: 297.2296, Val Loss: 139.1321
Epoch [720/3000], Train Loss: 293.1558, Val Loss: 133.8564
Epoch [730/3000], Train Loss: 291.3673, Val Loss: 132.3879
Epoch [740/3000], Train Loss: 289.5096, Val Loss: 131.1984
Epoch [750/3000], Train Loss: 285.8670, Val Loss: 127.8757
Epoch [760/3000], Train Loss: 283.5089, Val Loss: 126.0784
Epoch [770/3000], Train Loss: 280.9081, Val Loss: 127.8975
Epoch [780/3000], Train Loss: 277.1148, Val Loss: 127.4610
Epoch [790/3000], Train Loss: 275.3409, Val Loss: 127.2116
Epoch [800/3000], Train Loss: 272.9195, Val Loss: 122.2731
Epoch [810/3000], Train Loss: 270.0008, Val Loss: 119.5473
Epoch [820/3000], Train Loss: 268.3408, Val Loss: 118.9863
Epoch [830/3000], Train Loss: 265.6374, Val Loss: 117.0228
Epoch [840/3000], Train Loss: 263.4792, Val Loss: 116.3368
Epoch [850/3000], Train Loss: 261.1463, Val Loss: 114.9678
Epoch [860/3000], Train Loss: 258.1842, Val Loss: 113.9380
Epoch [870/3000], Train Loss: 256.4466, Val Loss: 111.2624
Epoch [880/3000], Train Loss: 253.9297, Val Loss: 110.3496
Epoch [890/3000], Train Loss: 251.4222, Val Loss: 109.4902
Epoch [900/3000], Train Loss: 249.0736, Val Loss: 107.4268
Epoch [910/3000], Train Loss: 247.4741, Val Loss: 107.8101
Epoch [920/3000], Train Loss: 245.3165, Val Loss: 107.8644
Epoch [930/3000], Train Loss: 243.1395, Val Loss: 106.0409
Epoch [940/3000], Train Loss: 241.0189, Val Loss: 105.7801
Epoch [950/3000], Train Loss: 239.6714, Val Loss: 104.6199
Epoch [960/3000], Train Loss: 237.4846, Val Loss: 101.2453
Epoch [970/3000], Train Loss: 234.7814, Val Loss: 101.6119
Epoch [980/3000], Train Loss: 233.5699, Val Loss: 101.5904
Epoch [990/3000], Train Loss: 231.6402, Val Loss: 99.9985
Epoch [1000/3000], Train Loss: 229.6795, Val Loss: 98.4479
Epoch [1010/3000], Train Loss: 227.5080, Val Loss: 97.6271
Epoch [1020/3000], Train Loss: 226.2702, Val Loss: 96.4655
Epoch [1030/3000], Train Loss: 224.1594, Val Loss: 96.8722
Epoch [1040/3000], Train Loss: 222.3503, Val Loss: 93.1366
Epoch [1050/3000], Train Loss: 220.5237, Val Loss: 95.1915
Epoch [1060/3000], Train Loss: 219.4306, Val Loss: 92.7521
Epoch [1070/3000], Train Loss: 217.0003, Val Loss: 91.0391
Epoch [1080/3000], Train Loss: 215.0062, Val Loss: 92.9009
Epoch [1090/3000], Train Loss: 212.7579, Val Loss: 89.9007
```

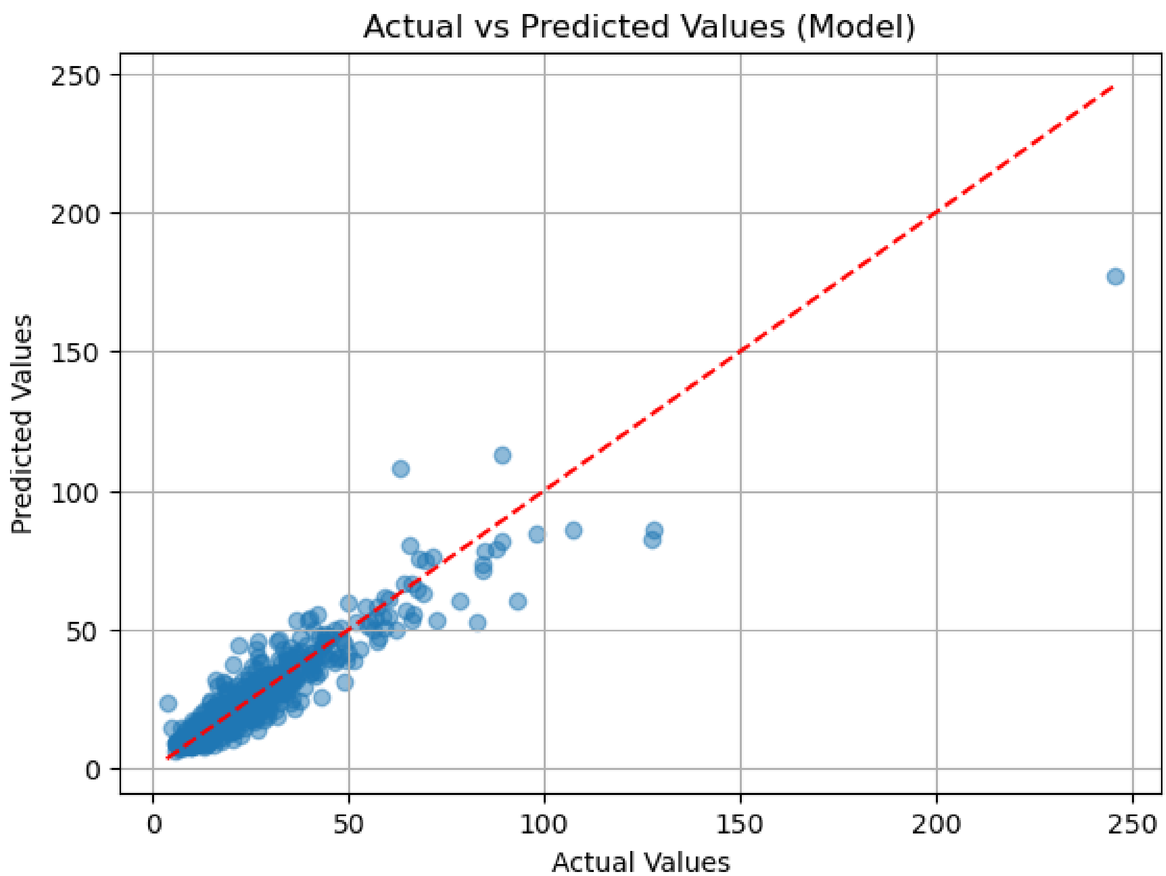
```
Epoch [1100/3000], Train Loss: 211.8514, Val Loss: 88.0061
Epoch [1110/3000], Train Loss: 210.1269, Val Loss: 85.9341
Epoch [1120/3000], Train Loss: 208.7714, Val Loss: 91.5790
Epoch [1130/3000], Train Loss: 206.4010, Val Loss: 90.6554
Epoch [1140/3000], Train Loss: 204.3830, Val Loss: 89.6532
Epoch [1150/3000], Train Loss: 203.6959, Val Loss: 89.2974
Epoch [1160/3000], Train Loss: 201.4929, Val Loss: 88.8583
Epoch [1170/3000], Train Loss: 200.3690, Val Loss: 84.1039
Epoch [1180/3000], Train Loss: 197.8281, Val Loss: 85.3187
Epoch [1190/3000], Train Loss: 196.5895, Val Loss: 85.1077
Epoch [1200/3000], Train Loss: 195.0304, Val Loss: 85.1213
Epoch [1210/3000], Train Loss: 194.0274, Val Loss: 80.5765
Epoch [1220/3000], Train Loss: 192.9698, Val Loss: 82.4580
Epoch [1230/3000], Train Loss: 190.5625, Val Loss: 79.4048
Epoch [1240/3000], Train Loss: 188.9924, Val Loss: 81.8780
Epoch [1250/3000], Train Loss: 187.3711, Val Loss: 81.3156
Epoch [1260/3000], Train Loss: 185.7895, Val Loss: 82.0416
Epoch [1270/3000], Train Loss: 185.1033, Val Loss: 81.3030
Epoch [1280/3000], Train Loss: 183.4679, Val Loss: 77.2549
Epoch [1290/3000], Train Loss: 181.8413, Val Loss: 78.1357
Epoch [1300/3000], Train Loss: 180.1181, Val Loss: 79.9820
Epoch [1310/3000], Train Loss: 178.8289, Val Loss: 75.7877
Epoch [1320/3000], Train Loss: 177.4787, Val Loss: 78.2868
Epoch [1330/3000], Train Loss: 175.8099, Val Loss: 78.2056
Epoch [1340/3000], Train Loss: 174.2967, Val Loss: 76.2412
Epoch [1350/3000], Train Loss: 173.4266, Val Loss: 73.7899
Epoch [1360/3000], Train Loss: 171.6779, Val Loss: 76.7758
Epoch [1370/3000], Train Loss: 170.3346, Val Loss: 75.5946
Epoch [1380/3000], Train Loss: 168.8669, Val Loss: 74.0210
Epoch [1390/3000], Train Loss: 167.3740, Val Loss: 72.5895
Epoch [1400/3000], Train Loss: 166.0790, Val Loss: 71.5685
Epoch [1410/3000], Train Loss: 164.8049, Val Loss: 71.9062
Epoch [1420/3000], Train Loss: 163.7440, Val Loss: 72.5149
Epoch [1430/3000], Train Loss: 162.1865, Val Loss: 74.8183
Epoch [1440/3000], Train Loss: 160.9719, Val Loss: 72.8818
Epoch [1450/3000], Train Loss: 160.1599, Val Loss: 72.2733
Epoch [1460/3000], Train Loss: 159.6580, Val Loss: 71.9436
Epoch [1470/3000], Train Loss: 157.3667, Val Loss: 70.6574
Epoch [1480/3000], Train Loss: 156.1098, Val Loss: 70.6578
Epoch [1490/3000], Train Loss: 155.0821, Val Loss: 72.4272
Epoch [1500/3000], Train Loss: 153.7685, Val Loss: 68.9280
Epoch [1510/3000], Train Loss: 152.5436, Val Loss: 70.6565
Epoch [1520/3000], Train Loss: 151.6387, Val Loss: 67.5730
```

```
Epoch [1530/3000], Train Loss: 150.6447, Val Loss: 68.7568
Epoch [1540/3000], Train Loss: 149.8463, Val Loss: 69.0988
Epoch [1550/3000], Train Loss: 148.3323, Val Loss: 68.6356
Epoch [1560/3000], Train Loss: 146.8783, Val Loss: 66.5376
Epoch [1570/3000], Train Loss: 145.7830, Val Loss: 68.7561
Epoch [1580/3000], Train Loss: 144.7695, Val Loss: 67.1103
Epoch [1590/3000], Train Loss: 143.6030, Val Loss: 67.5270
Epoch [1600/3000], Train Loss: 142.4209, Val Loss: 67.8633
Epoch [1610/3000], Train Loss: 141.7666, Val Loss: 66.5600
Epoch [1620/3000], Train Loss: 140.4418, Val Loss: 67.7374
Epoch [1630/3000], Train Loss: 139.2329, Val Loss: 70.1780
Epoch [1640/3000], Train Loss: 139.0233, Val Loss: 67.3184
Epoch [1650/3000], Train Loss: 137.2160, Val Loss: 65.6899
Epoch [1660/3000], Train Loss: 136.1907, Val Loss: 68.1165
Epoch [1670/3000], Train Loss: 135.7569, Val Loss: 65.2837
Epoch [1680/3000], Train Loss: 134.5427, Val Loss: 65.4321
Epoch [1690/3000], Train Loss: 133.5053, Val Loss: 64.5273
Epoch [1700/3000], Train Loss: 132.7533, Val Loss: 66.5384
Epoch [1710/3000], Train Loss: 132.0370, Val Loss: 68.9226
Epoch [1720/3000], Train Loss: 130.9869, Val Loss: 67.9919
Epoch [1730/3000], Train Loss: 130.3685, Val Loss: 65.8093
Epoch [1740/3000], Train Loss: 128.9625, Val Loss: 66.5856
Epoch [1750/3000], Train Loss: 127.8757, Val Loss: 63.5284
Epoch [1760/3000], Train Loss: 127.2936, Val Loss: 63.4254
Epoch [1770/3000], Train Loss: 126.4377, Val Loss: 64.9431
Epoch [1780/3000], Train Loss: 125.1934, Val Loss: 65.1679
Epoch [1790/3000], Train Loss: 124.5525, Val Loss: 64.6615
Epoch [1800/3000], Train Loss: 124.3395, Val Loss: 63.8481
Epoch [1810/3000], Train Loss: 123.4211, Val Loss: 66.0345
Epoch [1820/3000], Train Loss: 122.2801, Val Loss: 64.0493
Epoch [1830/3000], Train Loss: 121.5827, Val Loss: 63.5406
Epoch [1840/3000], Train Loss: 120.9762, Val Loss: 64.8369
Epoch [1850/3000], Train Loss: 120.1384, Val Loss: 63.7654
Epoch [1860/3000], Train Loss: 119.0760, Val Loss: 64.8412
Epoch [1870/3000], Train Loss: 118.3706, Val Loss: 66.1343
Epoch [1880/3000], Train Loss: 117.5391, Val Loss: 63.1404
Epoch [1890/3000], Train Loss: 117.0470, Val Loss: 64.0964
Epoch [1900/3000], Train Loss: 115.9382, Val Loss: 64.4309
Epoch [1910/3000], Train Loss: 115.0538, Val Loss: 64.1527
Epoch [1920/3000], Train Loss: 114.5444, Val Loss: 65.5393
Epoch [1930/3000], Train Loss: 113.6631, Val Loss: 65.1711
Epoch [1940/3000], Train Loss: 113.4395, Val Loss: 67.2524
Epoch [1950/3000], Train Loss: 112.1825, Val Loss: 64.2757
```

```
Epoch [1960/3000], Train Loss: 112.1844, Val Loss: 64.3860
Epoch [1970/3000], Train Loss: 111.3415, Val Loss: 63.7314
Epoch [1980/3000], Train Loss: 110.6227, Val Loss: 64.0904
Epoch [1990/3000], Train Loss: 109.3487, Val Loss: 64.2847
Epoch [2000/3000], Train Loss: 108.9872, Val Loss: 63.1711
Epoch [2010/3000], Train Loss: 107.8140, Val Loss: 65.8118
Epoch [2020/3000], Train Loss: 107.0848, Val Loss: 65.5678
Epoch [2030/3000], Train Loss: 107.0298, Val Loss: 67.7580
Epoch [2040/3000], Train Loss: 105.8401, Val Loss: 64.5578
Epoch [2050/3000], Train Loss: 105.5651, Val Loss: 65.6982
Epoch [2060/3000], Train Loss: 105.2389, Val Loss: 64.8927
Epoch [2070/3000], Train Loss: 103.9555, Val Loss: 64.0075
Epoch [2080/3000], Train Loss: 103.1114, Val Loss: 67.7239
Epoch [2090/3000], Train Loss: 102.6616, Val Loss: 67.5062
Epoch [2100/3000], Train Loss: 102.9571, Val Loss: 67.2030
Epoch [2110/3000], Train Loss: 101.0403, Val Loss: 66.6137
Epoch [2120/3000], Train Loss: 100.5723, Val Loss: 68.2896
Epoch [2130/3000], Train Loss: 99.5712, Val Loss: 66.7092
Epoch [2140/3000], Train Loss: 99.5899, Val Loss: 64.0705
Epoch [2150/3000], Train Loss: 99.4821, Val Loss: 67.0173
Epoch [2160/3000], Train Loss: 98.4651, Val Loss: 67.6521
Epoch [2170/3000], Train Loss: 97.5777, Val Loss: 64.8495
Epoch [2180/3000], Train Loss: 97.1009, Val Loss: 65.3012
Epoch [2190/3000], Train Loss: 96.2967, Val Loss: 67.1722
Epoch [2200/3000], Train Loss: 95.9820, Val Loss: 69.6837
Early stopping triggered at epoch 2202

Final Test Metrics:
test_loss: 37.0165
rmse: 6.0841
r2: 0.8738
bias: -0.3307
rpd: 2.8144
```



Root Mean Squared Error (RMSE): 6.0841
R^2: 0.8738
Bias: -0.3307
RPD: 2.8144



4.9.2 ISRIC SoilGrids Data

The ISRIC SoilGrids dataset is a comprehensive global soil information system that uses machine learning to map the spatial distribution of soil properties worldwide. It's developed and maintained by ISRIC - World Soil Information.

Properties:

- 250 meter resolution global coverage
- Based on over 230,000 soil profile observations from the WoSIS database

Modeling Data Sources: SoilGrids models are trained using two primary types of data: 1. **Soil profile observations** from the World Soil Information Service (WoSIS) database, containing over 230,000 profiles collected by various national and international soil survey programs worldwide 2. **Environmental covariates** selected from more than 400 layers including: - Climate data (temperature, precipitation, seasonality) - Remote sensing products

(MODIS, Landsat) - Terrain attributes (elevation, slope, aspect) - Land cover information - Lithology and parent material data - Vegetation indices and biomass estimates

These data sources are combined using machine learning techniques to predict soil properties globally with quantified uncertainty.

For our study, we used the following soil properties from SoilGrids: - **Bulk density** (bdod): at 0-5cm, 5-15cm, and 15-30cm depths - **Clay content** (clay): percentage of clay particles at 0-5cm, 5-15cm, and 15-30cm depths - **pH in water** (phh2o): soil pH at 0-5cm, 5-15cm, and 15-30cm depths - **Sand content** (sand): percentage of sand particles at 0-5cm, 5-15cm, and 15-30cm depths - **Silt content** (silt): percentage of silt particles at 0-5cm, 5-15cm, and 15-30cm depths

These soil physical and chemical properties provide valuable information about soil texture, structure, and composition, which are known to be correlated with soil organic carbon content. By including these properties as auxiliary data, we aim to enhance our predictive models.

Importantly, we deliberately excluded organic carbon-related variables from SoilGrids to avoid circular dependencies in our modeling, as soil organic carbon (SOC) is our target variable.

For more information, visit: https://www.isric.org/explore/soilgrids

Or the Paper: https://soil.copernicus.org/articles/7/217/2021/ ***

Load data.

```
# read pd from csv
isric_data = pd.read_csv('data/soilgrids_parallel.csv')
isric_data.head()
```

	point_index	lon	lat	bdod_0_to_5cm_mean	bdod_5_to_15cm_mean	bdod_15_to_
0	1	4.584692	45.816720	125.0	134.0	141.0
1	0	4.680379	45.893933	128.0	138.0	141.0
2	3	4.601575	45.908022	133.0	140.0	144.0
3	2	4.671533	45.983716	129.0	139.0	143.0
4	6	4.439863	46.224665	102.0	116.0	121.0

Select variables.

```
isric_measurements = ["bdod_0_to_5cm_mean", "bdod_5_to_15cm_mean", "bdod_15_to_30cm_mean",
                      "clay_0_to_5cm_mean", "clay_5_to_15cm_mean", "clay_15_to_30cm_mean",
                      "phh2o_0_to_5cm_mean", "phh2o_5_to_15cm_mean", "phh2o_15_to_30cm_mea
                      "sand_0_to_5cm_mean", "sand_5_to_15cm_mean", "sand_15_to_30cm_mean",
                      "silt_0_to_5cm_mean", "silt_5_to_15cm_mean", "silt_15_to_30cm_mean"]
```

The rest of thie code for the approach is the same as for the DLR-Data:

```
# Create a mapping from original indices to rows in the isriciliary data
point_to_row = dict(zip(isric_data['point_index'], isric_data.index))

# Filter indices to only include those with data in isric_data
valid_ks_indices = [idx for idx in original_ks_indices if idx in point_to_row]
valid_test_indices = [idx for idx in original_ks_indices if idx in point_to_row]

# Map original indices to row positions in isric_data
mapped_ks_indices = [point_to_row[idx] for idx in valid_ks_indices]
mapped_test_indices = [point_to_row[idx] for idx in valid_test_indices]

# Now use these mapped indices to select data
isric_train_ks = isric_data.iloc[mapped_ks_indices]
isric_test_ks = isric_data.iloc[mapped_test_indices]

# Print shapes
print(f"Original indices: {len(ks_indices)} training, {len(test_indices)} test")
print(f"Valid indices with isric data: {len(valid_ks_indices)} training, {len(valid_test_i

# select only relevant columns
isric_train_ks = isric_train_ks[isric_measurements].values
isric_test_ks = isric_test_ks[isric_measurements].values

isric_train_ks = np.nan_to_num(isric_train_ks, nan=0, posinf=0, neginf=0)
isric_test_ks = np.nan_to_num(isric_test_ks, nan=0, posinf=0, neginf=0)

print(f"Auxillary isric Train data shape: {isric_train_ks.shape}")
print(f"Auxillary isric Test data shape: {isric_test_ks.shape}")
```

```
Original indices: 1964 training, 843 test
Valid indices with isric data: 1964 training, 843 test
Auxillary isric Train data shape: (1964, 15)
Auxillary isric Test data shape: (843, 15)
```

```
original_train_indices = ks_indices  # These are the indices used to create X_train_pls

# Map from original index to position in X_train_pls
original_to_train_pos = {orig_idx: train_pos for train_pos, orig_idx in enumerate(original
```

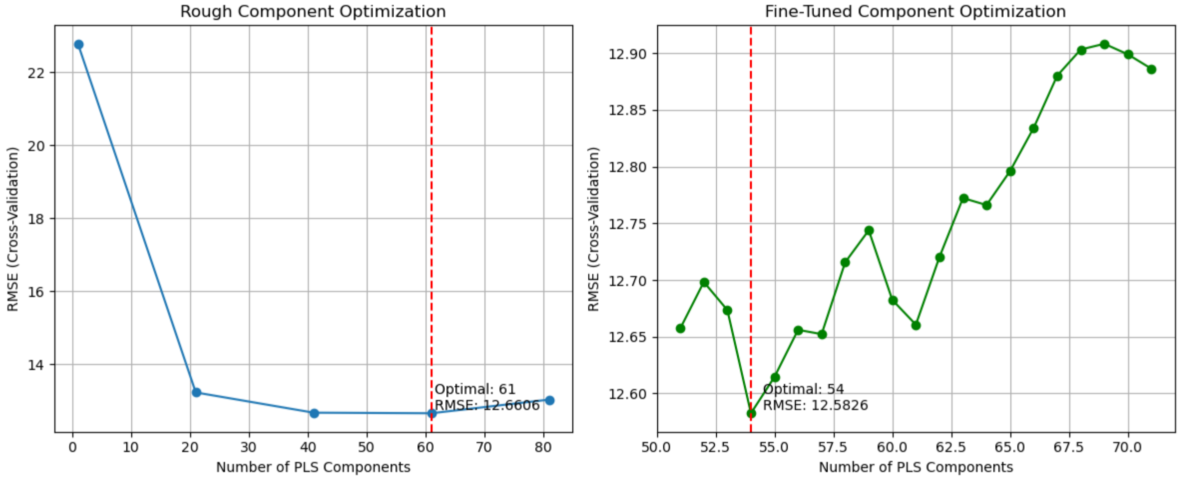
```python
# Find the positions in X_train_pls that correspond to valid_ks_indices
train_pls_positions = []
for idx in valid_ks_indices:
    if idx in original_to_train_pos:
        train_pls_positions.append(original_to_train_pos[idx])

# Now use these positions to select from X_train_pls
X_train_pls_isric = X_train_absorb[train_pls_positions, :]
y_train_isric = y_train[train_pls_positions]

# Do the same for test data
original_test_indices = test_indices  # These are the indices used to create X_test_pls
original_to_test_pos = {orig_idx: test_pos for test_pos, orig_idx in enumerate(original_te

test_pls_positions = []
for idx in valid_test_indices:
    if idx in original_to_test_pos:
        test_pls_positions.append(original_to_test_pos[idx])

X_test_pls_isric = X_test_absorb[test_pls_positions, :]
y_test_isric = y_test[test_pls_positions]

# Print shapes
print(f"X_train_pls_isric shape: {X_train_pls_isric.shape}")
print(f"X_test_pls_isric shape: {X_test_pls_isric.shape}")
print(f"y_train_isric shape: {y_train_isric.shape}")
print(f"y_test_isric shape: {y_test_isric.shape}")

# Add the isriciliary data to the PLSR transformed data
X_train_combined = np.hstack((X_train_pls_isric, isric_train_ks))
X_test_combined = np.hstack((X_test_pls_isric, isric_test_ks))

# standardize the data
scaler = StandardScaler()
X_train_combined = scaler.fit_transform(X_train_combined)
X_test_combined = scaler.transform(X_test_combined)

# Print shapes
print(f"X_train_combined shape: {X_train_combined.shape}")
print(f"X_test_combined shape: {X_test_combined.shape}")
```

```
X_train_pls_isric shape: (1964, 1000)
```

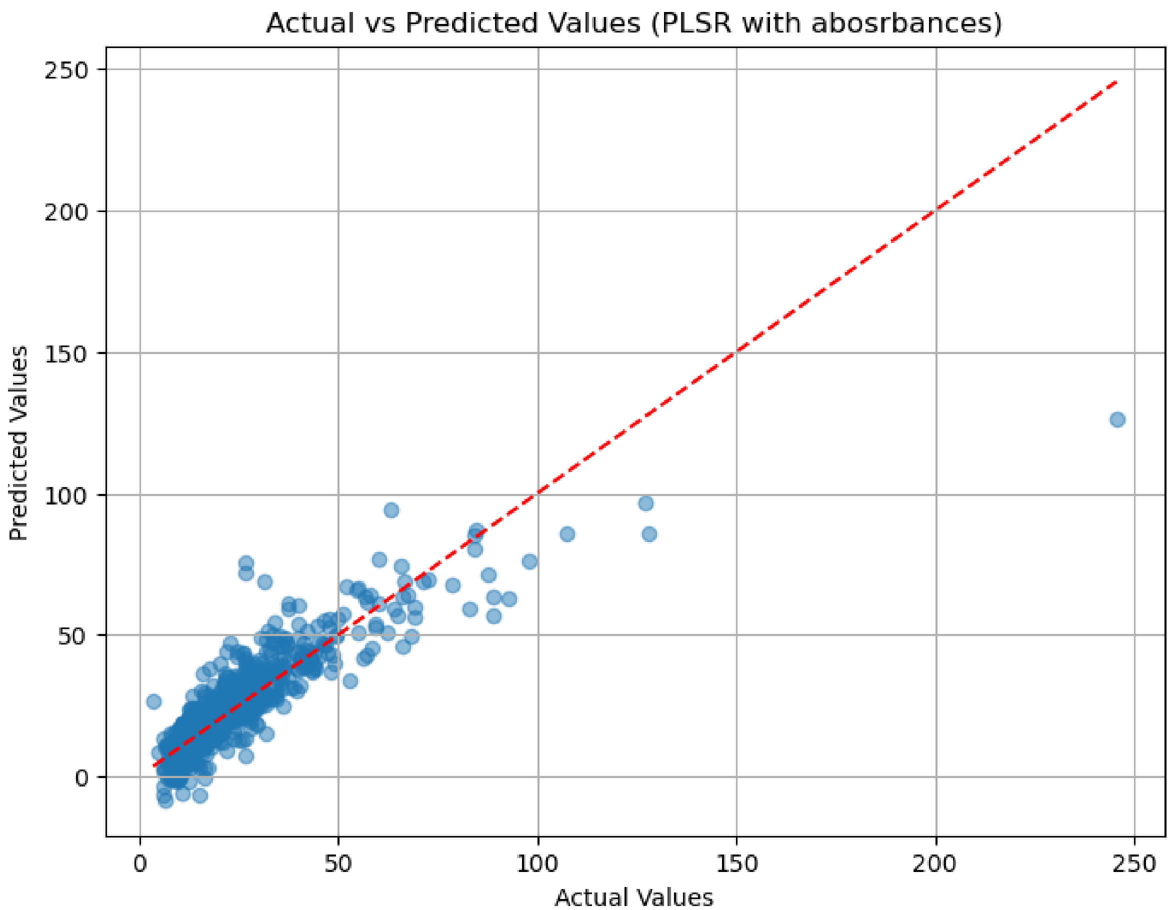
```
X_test_pls_isric shape: (843, 1000)
y_train_isric shape: (1964,)
y_test_isric shape: (843,)
X_train_combined shape: (1964, 1015)
X_test_combined shape: (843, 1015)
```

```
plsr_isric_components = own_functions.optimize_pls_components(X_train=X_train_combined,
                                      y_train=y_train_isric,
                                      max_components=100,
                                      step=20,
                                      fine_tune=True,
                                      show_progress=True,
                                      plot_results=True
                                      )


plsr_absorb_aux_model = PLSRegression(n_components=plsr_isric_components["optimal_n"])
plsr_absorb_aux_model.fit(X_train_combined, y_train_isric)

plsr_absorb_aux_eval = own_functions.evaluate_model(plsr_absorb_aux_model,
                             X_test=X_test_combined,
                             y_test=y_test_isric,
                             print_metrics=True,
                             show_plot=True,
                             plot_kwargs={'model_name': 'PLSR with abosrbances',
                             'figsize': (8, 6)}
                             )
```

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

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




```
Root Mean Squared Error (RMSE): 8.7390
R²: 0.7395
Bias: 0.7804
RPD: 1.9594
```




```
X_train_comb_plsr = plsr_absorb_aux_model.transform(X_train_combined)
X_test_comb_plsr = plsr_absorb_aux_model.transform(X_test_combined)
```

```
# First split training data into train and validation sets
X_train_final, X_val, y_train_final, y_val = train_test_split(X_train_comb_plsr, y_train_i
                                                              test_size=0.2,
                                                              random_state=42)

# Training enhanced LSTM model
LSTM_aux_model= own_functions.train_lstm(
    X_train=X_train_final,
    X_val=X_val,
    X_test=X_test_comb_plsr,
    y_train=y_train_final,
```

```
    y_val=y_val,
    y_test=y_test_isric,
    hidden_size=256,
    num_layers=5,
    num_epochs=4000,
    learning_rate=0.001,
    patience=300,  # Early stopping patience
    dropout=0.2
)


lstm_eval = own_functions.evaluate_model(LSTM_aux_model,
                             X_test=X_test_comb_plsr, y_test=y_test_isric,
                             print_metrics=True, show_plot=True)
```

```
# First split training data into train and validation sets
X_train_final, X_val, y_train_final, y_val = train_test_split(X_train_comb_plsr, y_train_i
                                                              test_size=0.2,
                                                              random_state=42)

# Training enhanced LSTM model
LSTM_aux_model= own_functions.train_lstm(
    X_train=X_train_final,
    X_val=X_val,
    X_test=X_test_comb_plsr,
    y_train=y_train_final,
    y_val=y_val,
    y_test=y_test_isric,
    hidden_size=256,
    num_layers=5,
    num_epochs=4000,
    learning_rate=0.001,
    patience=200,  # Early stopping patience
    dropout=0.2
)


lstm_eval = own_functions.evaluate_model(LSTM_aux_model,
                             X_test=X_test_comb_plsr, y_test=y_test_isric,
                             print_metrics=True, show_plot=True)
```

```
Epoch [10/4000], Train Loss: 1475.2972, Val Loss: 1519.5110
Epoch [20/4000], Train Loss: 1280.4115, Val Loss: 1283.1000
Epoch [30/4000], Train Loss: 1040.9993, Val Loss: 1062.2501
Epoch [40/4000], Train Loss: 929.5973, Val Loss: 954.4364
Epoch [50/4000], Train Loss: 861.1890, Val Loss: 885.2343
Epoch [60/4000], Train Loss: 811.6935, Val Loss: 834.5779
Epoch [70/4000], Train Loss: 775.9415, Val Loss: 797.5146
Epoch [80/4000], Train Loss: 750.5928, Val Loss: 770.7653
Epoch [90/4000], Train Loss: 732.1863, Val Loss: 751.7930
Epoch [100/4000], Train Loss: 718.9263, Val Loss: 738.2095
Epoch [110/4000], Train Loss: 709.2322, Val Loss: 727.8226
Epoch [120/4000], Train Loss: 702.1847, Val Loss: 720.2368
Epoch [130/4000], Train Loss: 696.5098, Val Loss: 714.1851
Epoch [140/4000], Train Loss: 691.6286, Val Loss: 708.9233
Epoch [150/4000], Train Loss: 686.7952, Val Loss: 703.0333
Epoch [160/4000], Train Loss: 676.5783, Val Loss: 696.5714
Epoch [170/4000], Train Loss: 660.6702, Val Loss: 671.7918
Epoch [180/4000], Train Loss: 611.3018, Val Loss: 620.0627
Epoch [190/4000], Train Loss: 585.4724, Val Loss: 594.4952
Epoch [200/4000], Train Loss: 562.5011, Val Loss: 571.0565
Epoch [210/4000], Train Loss: 542.8493, Val Loss: 551.1712
Epoch [220/4000], Train Loss: 525.5701, Val Loss: 534.1605
Epoch [230/4000], Train Loss: 509.3923, Val Loss: 518.1349
Epoch [240/4000], Train Loss: 495.3281, Val Loss: 503.3998
Epoch [250/4000], Train Loss: 482.5118, Val Loss: 490.0721
Epoch [260/4000], Train Loss: 470.3756, Val Loss: 478.1037
Epoch [270/4000], Train Loss: 458.9582, Val Loss: 467.4412
Epoch [280/4000], Train Loss: 448.4810, Val Loss: 456.5381
Epoch [290/4000], Train Loss: 438.2216, Val Loss: 447.2844
Epoch [300/4000], Train Loss: 428.8750, Val Loss: 438.7708
Epoch [310/4000], Train Loss: 420.6180, Val Loss: 429.9001
Epoch [320/4000], Train Loss: 412.1140, Val Loss: 421.0419
Epoch [330/4000], Train Loss: 405.7183, Val Loss: 414.2930
Epoch [340/4000], Train Loss: 396.6264, Val Loss: 406.5574
Epoch [350/4000], Train Loss: 390.4545, Val Loss: 399.5245
Epoch [360/4000], Train Loss: 383.7248, Val Loss: 393.4516
Epoch [370/4000], Train Loss: 378.0196, Val Loss: 389.2910
Epoch [380/4000], Train Loss: 372.8172, Val Loss: 380.0265
Epoch [390/4000], Train Loss: 368.0840, Val Loss: 374.1557
Epoch [400/4000], Train Loss: 362.7235, Val Loss: 374.2974
Epoch [410/4000], Train Loss: 357.2463, Val Loss: 369.5172
Epoch [420/4000], Train Loss: 352.7180, Val Loss: 360.7042
Epoch [430/4000], Train Loss: 347.9333, Val Loss: 355.6990
```

```
Epoch [440/4000], Train Loss: 343.2024, Val Loss: 351.0757
Epoch [450/4000], Train Loss: 340.2503, Val Loss: 346.8278
Epoch [460/4000], Train Loss: 335.1614, Val Loss: 343.3439
Epoch [470/4000], Train Loss: 332.5359, Val Loss: 339.5449
Epoch [480/4000], Train Loss: 329.0076, Val Loss: 334.6169
Epoch [490/4000], Train Loss: 324.6738, Val Loss: 330.5406
Epoch [500/4000], Train Loss: 321.7654, Val Loss: 327.0572
Epoch [510/4000], Train Loss: 318.4358, Val Loss: 322.8938
Epoch [520/4000], Train Loss: 315.3759, Val Loss: 318.9895
Epoch [530/4000], Train Loss: 312.0256, Val Loss: 315.3359
Epoch [540/4000], Train Loss: 309.3525, Val Loss: 312.6183
Epoch [550/4000], Train Loss: 305.9377, Val Loss: 309.1159
Epoch [560/4000], Train Loss: 302.1337, Val Loss: 304.8154
Epoch [570/4000], Train Loss: 299.8845, Val Loss: 300.7813
Epoch [580/4000], Train Loss: 297.0800, Val Loss: 298.9834
Epoch [590/4000], Train Loss: 294.7407, Val Loss: 294.9838
Epoch [600/4000], Train Loss: 291.3507, Val Loss: 292.5641
Epoch [610/4000], Train Loss: 289.1307, Val Loss: 288.1119
Epoch [620/4000], Train Loss: 286.0616, Val Loss: 284.8908
Epoch [630/4000], Train Loss: 283.8226, Val Loss: 281.2082
Epoch [640/4000], Train Loss: 280.8970, Val Loss: 279.6353
Epoch [650/4000], Train Loss: 278.5081, Val Loss: 277.2468
Epoch [660/4000], Train Loss: 275.2970, Val Loss: 272.4575
Epoch [670/4000], Train Loss: 274.0508, Val Loss: 269.8115
Epoch [680/4000], Train Loss: 272.5681, Val Loss: 267.6024
Epoch [690/4000], Train Loss: 270.0841, Val Loss: 264.0202
Epoch [700/4000], Train Loss: 267.5056, Val Loss: 259.1208
Epoch [710/4000], Train Loss: 265.4495, Val Loss: 255.3375
Epoch [720/4000], Train Loss: 263.1166, Val Loss: 255.2786
Epoch [730/4000], Train Loss: 260.1707, Val Loss: 255.9149
Epoch [740/4000], Train Loss: 259.7098, Val Loss: 251.5681
Epoch [750/4000], Train Loss: 257.0754, Val Loss: 245.3743
Epoch [760/4000], Train Loss: 254.6993, Val Loss: 241.7000
Epoch [770/4000], Train Loss: 252.8703, Val Loss: 241.0565
Epoch [780/4000], Train Loss: 250.8164, Val Loss: 239.1893
Epoch [790/4000], Train Loss: 248.8749, Val Loss: 235.0323
Epoch [800/4000], Train Loss: 246.8988, Val Loss: 233.9730
Epoch [810/4000], Train Loss: 245.4286, Val Loss: 232.6353
Epoch [820/4000], Train Loss: 243.8551, Val Loss: 228.9728
Epoch [830/4000], Train Loss: 241.4019, Val Loss: 227.3049
Epoch [840/4000], Train Loss: 239.5529, Val Loss: 225.8664
Epoch [850/4000], Train Loss: 238.2764, Val Loss: 223.2610
Epoch [860/4000], Train Loss: 236.8090, Val Loss: 218.5238
```

```
Epoch [870/4000], Train Loss: 235.2233, Val Loss: 217.4158
Epoch [880/4000], Train Loss: 233.4767, Val Loss: 217.7320
Epoch [890/4000], Train Loss: 231.6711, Val Loss: 217.3340
Epoch [900/4000], Train Loss: 229.3501, Val Loss: 213.9813
Epoch [910/4000], Train Loss: 228.2005, Val Loss: 213.6611
Epoch [920/4000], Train Loss: 226.3165, Val Loss: 210.5754
Epoch [930/4000], Train Loss: 224.4331, Val Loss: 209.2676
Epoch [940/4000], Train Loss: 223.2852, Val Loss: 207.7989
Epoch [950/4000], Train Loss: 221.9013, Val Loss: 202.3709
Epoch [960/4000], Train Loss: 219.8360, Val Loss: 202.4597
Epoch [970/4000], Train Loss: 218.0010, Val Loss: 198.2457
Epoch [980/4000], Train Loss: 217.1879, Val Loss: 195.3857
Epoch [990/4000], Train Loss: 215.1354, Val Loss: 193.2548
Epoch [1000/4000], Train Loss: 213.0206, Val Loss: 190.4581
Epoch [1010/4000], Train Loss: 211.3067, Val Loss: 189.5974
Epoch [1020/4000], Train Loss: 210.3573, Val Loss: 192.0776
Epoch [1030/4000], Train Loss: 208.4079, Val Loss: 187.6253
Epoch [1040/4000], Train Loss: 206.9423, Val Loss: 185.5899
Epoch [1050/4000], Train Loss: 205.1693, Val Loss: 183.4603
Epoch [1060/4000], Train Loss: 203.9469, Val Loss: 180.7608
Epoch [1070/4000], Train Loss: 202.3957, Val Loss: 181.2276
Epoch [1080/4000], Train Loss: 201.4997, Val Loss: 178.1205
Epoch [1090/4000], Train Loss: 199.6372, Val Loss: 176.5981
Epoch [1100/4000], Train Loss: 198.5518, Val Loss: 175.3873
Epoch [1110/4000], Train Loss: 196.8762, Val Loss: 174.3162
Epoch [1120/4000], Train Loss: 196.2135, Val Loss: 173.3125
Epoch [1130/4000], Train Loss: 194.3036, Val Loss: 171.3059
Epoch [1140/4000], Train Loss: 193.0226, Val Loss: 168.4846
Epoch [1150/4000], Train Loss: 192.0900, Val Loss: 166.3214
Epoch [1160/4000], Train Loss: 190.3935, Val Loss: 165.6931
Epoch [1170/4000], Train Loss: 189.0933, Val Loss: 164.7315
Epoch [1180/4000], Train Loss: 187.7266, Val Loss: 162.8180
Epoch [1190/4000], Train Loss: 186.9007, Val Loss: 161.8801
Epoch [1200/4000], Train Loss: 185.7356, Val Loss: 159.2818
Epoch [1210/4000], Train Loss: 184.4393, Val Loss: 158.2003
Epoch [1220/4000], Train Loss: 182.7735, Val Loss: 156.4669
Epoch [1230/4000], Train Loss: 181.4821, Val Loss: 156.6378
Epoch [1240/4000], Train Loss: 180.7059, Val Loss: 155.0530
Epoch [1250/4000], Train Loss: 179.2459, Val Loss: 153.8369
Epoch [1260/4000], Train Loss: 178.3188, Val Loss: 151.6919
Epoch [1270/4000], Train Loss: 176.9521, Val Loss: 151.7181
Epoch [1280/4000], Train Loss: 176.0576, Val Loss: 150.8133
Epoch [1290/4000], Train Loss: 174.3492, Val Loss: 148.9969
```

```
Epoch [1300/4000], Train Loss: 173.7594, Val Loss: 148.8919
Epoch [1310/4000], Train Loss: 172.7791, Val Loss: 146.5303
Epoch [1320/4000], Train Loss: 171.1582, Val Loss: 146.5997
Epoch [1330/4000], Train Loss: 170.5263, Val Loss: 144.1215
Epoch [1340/4000], Train Loss: 168.8886, Val Loss: 144.9699
Epoch [1350/4000], Train Loss: 167.9973, Val Loss: 141.2650
Epoch [1360/4000], Train Loss: 166.7077, Val Loss: 139.8877
Epoch [1370/4000], Train Loss: 166.2939, Val Loss: 141.3412
Epoch [1380/4000], Train Loss: 164.6947, Val Loss: 140.1218
Epoch [1390/4000], Train Loss: 163.4467, Val Loss: 139.7124
Epoch [1400/4000], Train Loss: 162.7966, Val Loss: 137.7644
Epoch [1410/4000], Train Loss: 161.7658, Val Loss: 137.4774
Epoch [1420/4000], Train Loss: 160.7231, Val Loss: 134.2698
Epoch [1430/4000], Train Loss: 159.4350, Val Loss: 136.2363
Epoch [1440/4000], Train Loss: 158.7151, Val Loss: 134.6639
Epoch [1450/4000], Train Loss: 157.6062, Val Loss: 134.2825
Epoch [1460/4000], Train Loss: 156.2660, Val Loss: 130.8835
Epoch [1470/4000], Train Loss: 155.9137, Val Loss: 132.7157
Epoch [1480/4000], Train Loss: 154.5379, Val Loss: 131.2435
Epoch [1490/4000], Train Loss: 153.7037, Val Loss: 129.6104
Epoch [1500/4000], Train Loss: 152.9774, Val Loss: 128.5560
Epoch [1510/4000], Train Loss: 151.7004, Val Loss: 128.6219
Epoch [1520/4000], Train Loss: 150.6979, Val Loss: 126.3337
Epoch [1530/4000], Train Loss: 150.1365, Val Loss: 124.1229
Epoch [1540/4000], Train Loss: 148.3528, Val Loss: 125.6940
Epoch [1550/4000], Train Loss: 147.9403, Val Loss: 122.5712
Epoch [1560/4000], Train Loss: 146.6510, Val Loss: 121.2221
Epoch [1570/4000], Train Loss: 145.8448, Val Loss: 120.7168
Epoch [1580/4000], Train Loss: 144.8011, Val Loss: 123.2362
Epoch [1590/4000], Train Loss: 144.2140, Val Loss: 120.5647
Epoch [1600/4000], Train Loss: 143.1090, Val Loss: 119.2540
Epoch [1610/4000], Train Loss: 142.0329, Val Loss: 118.6189
Epoch [1620/4000], Train Loss: 141.2752, Val Loss: 116.8923
Epoch [1630/4000], Train Loss: 140.4433, Val Loss: 115.1083
Epoch [1640/4000], Train Loss: 139.1991, Val Loss: 116.2727
Epoch [1650/4000], Train Loss: 138.7201, Val Loss: 115.9392
Epoch [1660/4000], Train Loss: 137.7895, Val Loss: 115.5939
Epoch [1670/4000], Train Loss: 136.5574, Val Loss: 114.0230
Epoch [1680/4000], Train Loss: 135.9528, Val Loss: 114.4975
Epoch [1690/4000], Train Loss: 135.2841, Val Loss: 110.6569
Epoch [1700/4000], Train Loss: 134.8039, Val Loss: 115.1600
Epoch [1710/4000], Train Loss: 133.5406, Val Loss: 110.0300
Epoch [1720/4000], Train Loss: 132.7553, Val Loss: 108.4593
```

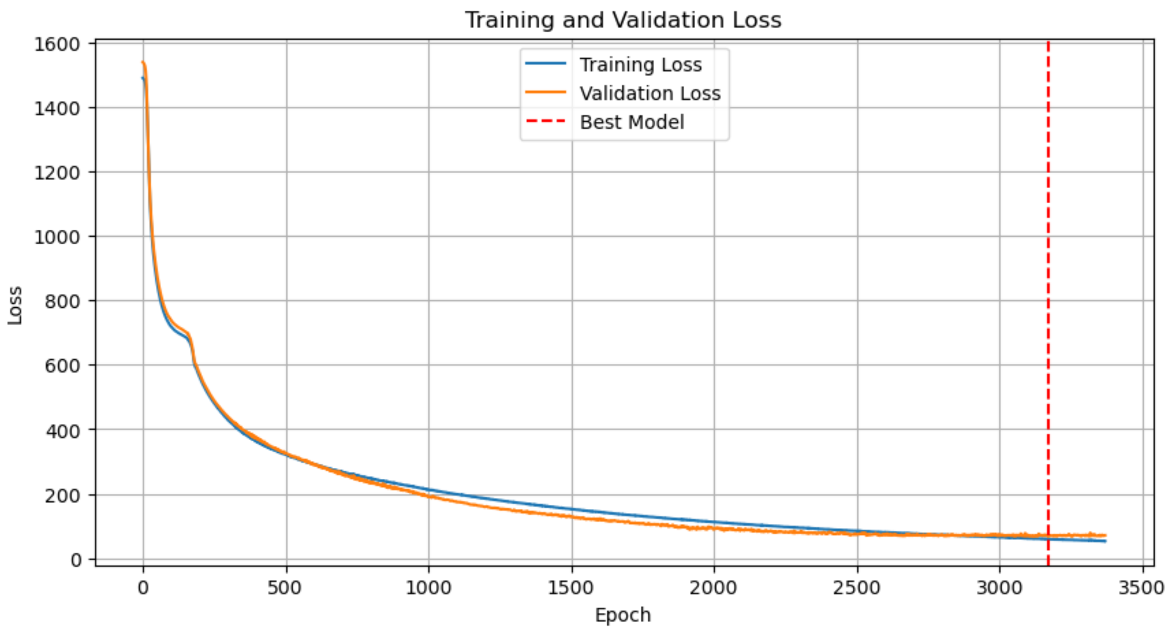
```
Epoch [1730/4000], Train Loss: 132.0244, Val Loss: 108.3886
Epoch [1740/4000], Train Loss: 131.3950, Val Loss: 107.6691
Epoch [1750/4000], Train Loss: 130.6855, Val Loss: 108.7865
Epoch [1760/4000], Train Loss: 129.5622, Val Loss: 106.1223
Epoch [1770/4000], Train Loss: 129.0035, Val Loss: 105.5860
Epoch [1780/4000], Train Loss: 128.0325, Val Loss: 103.9554
Epoch [1790/4000], Train Loss: 127.5116, Val Loss: 105.6273
Epoch [1800/4000], Train Loss: 126.4400, Val Loss: 102.6399
Epoch [1810/4000], Train Loss: 125.6601, Val Loss: 103.3804
Epoch [1820/4000], Train Loss: 125.4416, Val Loss: 105.8067
Epoch [1830/4000], Train Loss: 124.9395, Val Loss: 101.2187
Epoch [1840/4000], Train Loss: 123.2253, Val Loss: 104.2556
Epoch [1850/4000], Train Loss: 122.7832, Val Loss: 100.3161
Epoch [1860/4000], Train Loss: 121.9710, Val Loss: 98.5214
Epoch [1870/4000], Train Loss: 121.3527, Val Loss: 98.5841
Epoch [1880/4000], Train Loss: 120.6299, Val Loss: 95.7500
Epoch [1890/4000], Train Loss: 120.6749, Val Loss: 99.3560
Epoch [1900/4000], Train Loss: 119.5486, Val Loss: 97.6784
Epoch [1910/4000], Train Loss: 118.3321, Val Loss: 92.3545
Epoch [1920/4000], Train Loss: 117.9050, Val Loss: 94.4123
Epoch [1930/4000], Train Loss: 117.2739, Val Loss: 93.1113
Epoch [1940/4000], Train Loss: 116.2846, Val Loss: 88.5482
Epoch [1950/4000], Train Loss: 115.8986, Val Loss: 90.9105
Epoch [1960/4000], Train Loss: 115.3080, Val Loss: 93.9559
Epoch [1970/4000], Train Loss: 114.3435, Val Loss: 90.4082
Epoch [1980/4000], Train Loss: 113.5336, Val Loss: 94.2318
Epoch [1990/4000], Train Loss: 113.4553, Val Loss: 96.8833
Epoch [2000/4000], Train Loss: 112.2931, Val Loss: 89.5048
Epoch [2010/4000], Train Loss: 111.6759, Val Loss: 94.6542
Epoch [2020/4000], Train Loss: 111.2247, Val Loss: 91.0629
Epoch [2030/4000], Train Loss: 110.4307, Val Loss: 91.1231
Epoch [2040/4000], Train Loss: 110.0050, Val Loss: 88.7316
Epoch [2050/4000], Train Loss: 109.2240, Val Loss: 89.3921
Epoch [2060/4000], Train Loss: 108.3593, Val Loss: 89.8433
Epoch [2070/4000], Train Loss: 108.1223, Val Loss: 90.3443
Epoch [2080/4000], Train Loss: 107.0611, Val Loss: 86.9198
Epoch [2090/4000], Train Loss: 106.6175, Val Loss: 88.7713
Epoch [2100/4000], Train Loss: 105.7518, Val Loss: 88.6444
Epoch [2110/4000], Train Loss: 105.7235, Val Loss: 89.7381
Epoch [2120/4000], Train Loss: 105.1024, Val Loss: 87.4743
Epoch [2130/4000], Train Loss: 104.1024, Val Loss: 86.6269
Epoch [2140/4000], Train Loss: 103.9082, Val Loss: 88.0413
Epoch [2150/4000], Train Loss: 102.7591, Val Loss: 85.2298
```

```
Epoch [2160/4000], Train Loss: 102.4836, Val Loss: 84.3741
Epoch [2170/4000], Train Loss: 101.9873, Val Loss: 84.0700
Epoch [2180/4000], Train Loss: 101.5353, Val Loss: 83.6846
Epoch [2190/4000], Train Loss: 100.5013, Val Loss: 81.8236
Epoch [2200/4000], Train Loss: 99.8078, Val Loss: 85.8560
Epoch [2210/4000], Train Loss: 99.4545, Val Loss: 81.8619
Epoch [2220/4000], Train Loss: 99.1633, Val Loss: 81.9100
Epoch [2230/4000], Train Loss: 98.3155, Val Loss: 81.6756
Epoch [2240/4000], Train Loss: 97.9388, Val Loss: 82.4595
Epoch [2250/4000], Train Loss: 97.2953, Val Loss: 82.1583
Epoch [2260/4000], Train Loss: 96.4588, Val Loss: 79.8386
Epoch [2270/4000], Train Loss: 96.1209, Val Loss: 80.4036
Epoch [2280/4000], Train Loss: 96.0188, Val Loss: 79.1964
Epoch [2290/4000], Train Loss: 95.3891, Val Loss: 79.9859
Epoch [2300/4000], Train Loss: 94.5486, Val Loss: 79.4242
Epoch [2310/4000], Train Loss: 94.3102, Val Loss: 79.9428
Epoch [2320/4000], Train Loss: 93.6289, Val Loss: 78.5922
Epoch [2330/4000], Train Loss: 92.9661, Val Loss: 78.1629
Epoch [2340/4000], Train Loss: 92.2733, Val Loss: 80.4315
Epoch [2350/4000], Train Loss: 91.7731, Val Loss: 79.3922
Epoch [2360/4000], Train Loss: 91.7519, Val Loss: 77.7873
Epoch [2370/4000], Train Loss: 90.8322, Val Loss: 78.6464
Epoch [2380/4000], Train Loss: 90.5798, Val Loss: 75.7892
Epoch [2390/4000], Train Loss: 89.5495, Val Loss: 78.6463
Epoch [2400/4000], Train Loss: 89.3860, Val Loss: 78.1047
Epoch [2410/4000], Train Loss: 89.0039, Val Loss: 78.5845
Epoch [2420/4000], Train Loss: 88.4657, Val Loss: 75.7842
Epoch [2430/4000], Train Loss: 88.2265, Val Loss: 76.6274
Epoch [2440/4000], Train Loss: 87.3137, Val Loss: 75.7389
Epoch [2450/4000], Train Loss: 86.6753, Val Loss: 78.6994
Epoch [2460/4000], Train Loss: 87.1207, Val Loss: 79.2834
Epoch [2470/4000], Train Loss: 85.9596, Val Loss: 76.2952
Epoch [2480/4000], Train Loss: 85.3464, Val Loss: 71.3577
Epoch [2490/4000], Train Loss: 85.1605, Val Loss: 73.5715
Epoch [2500/4000], Train Loss: 84.6268, Val Loss: 73.5383
Epoch [2510/4000], Train Loss: 84.1133, Val Loss: 74.3282
Epoch [2520/4000], Train Loss: 83.5839, Val Loss: 75.7406
Epoch [2530/4000], Train Loss: 83.2210, Val Loss: 77.1210
Epoch [2540/4000], Train Loss: 82.8623, Val Loss: 73.0023
Epoch [2550/4000], Train Loss: 82.5890, Val Loss: 72.5696
Epoch [2560/4000], Train Loss: 82.0369, Val Loss: 74.2347
Epoch [2570/4000], Train Loss: 81.5016, Val Loss: 74.9981
Epoch [2580/4000], Train Loss: 81.0995, Val Loss: 70.3764
```

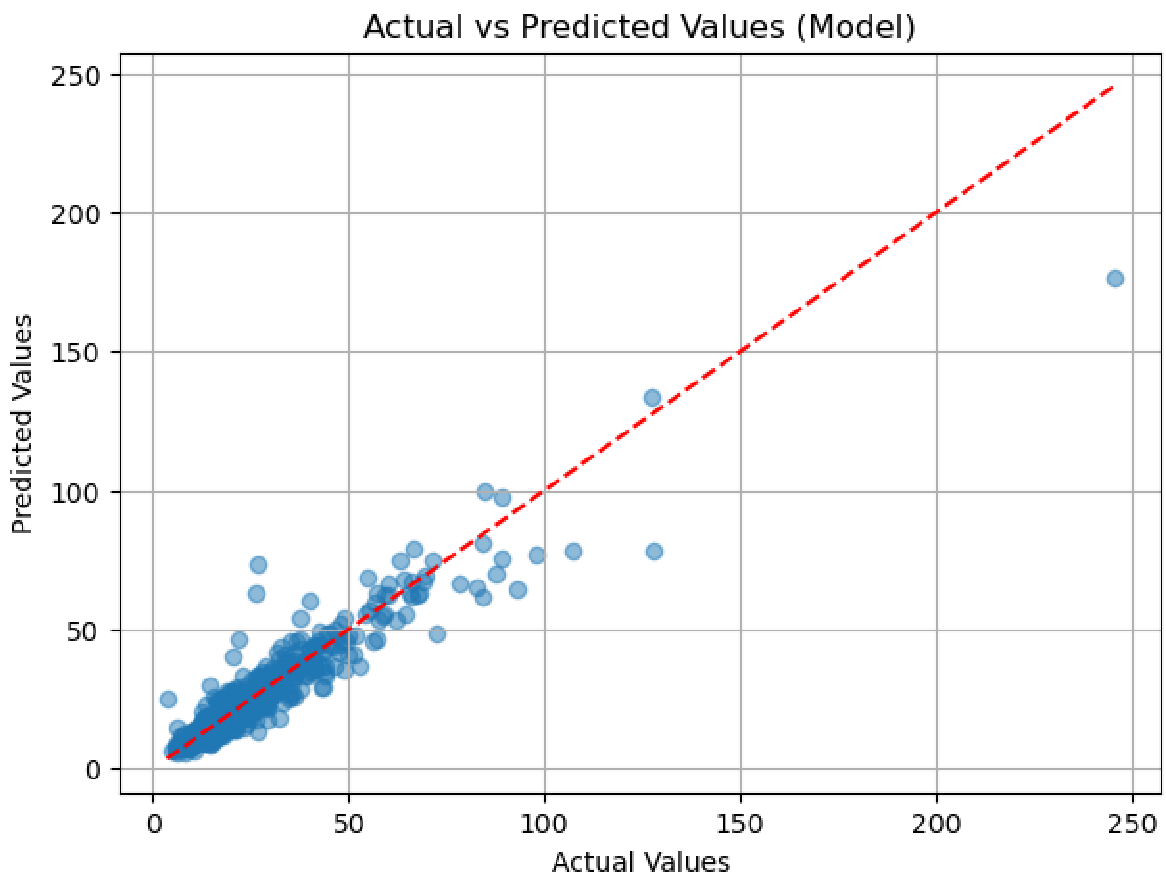
```
Epoch [2590/4000], Train Loss: 80.6859, Val Loss: 71.5359
Epoch [2600/4000], Train Loss: 80.6933, Val Loss: 73.4774
Epoch [2610/4000], Train Loss: 80.5954, Val Loss: 72.4979
Epoch [2620/4000], Train Loss: 79.7742, Val Loss: 71.6546
Epoch [2630/4000], Train Loss: 79.9925, Val Loss: 73.2060
Epoch [2640/4000], Train Loss: 78.7852, Val Loss: 73.0067
Epoch [2650/4000], Train Loss: 78.2762, Val Loss: 71.2027
Epoch [2660/4000], Train Loss: 77.8366, Val Loss: 70.4018
Epoch [2670/4000], Train Loss: 77.6922, Val Loss: 69.2568
Epoch [2680/4000], Train Loss: 76.9244, Val Loss: 73.6592
Epoch [2690/4000], Train Loss: 76.8279, Val Loss: 73.0960
Epoch [2700/4000], Train Loss: 76.0845, Val Loss: 72.2024
Epoch [2710/4000], Train Loss: 76.2772, Val Loss: 72.3820
Epoch [2720/4000], Train Loss: 75.3092, Val Loss: 71.8076
Epoch [2730/4000], Train Loss: 75.1692, Val Loss: 69.2691
Epoch [2740/4000], Train Loss: 74.9733, Val Loss: 69.8544
Epoch [2750/4000], Train Loss: 74.5515, Val Loss: 74.4492
Epoch [2760/4000], Train Loss: 74.0229, Val Loss: 71.0891
Epoch [2770/4000], Train Loss: 73.5033, Val Loss: 71.6540
Epoch [2780/4000], Train Loss: 73.3870, Val Loss: 70.4911
Epoch [2790/4000], Train Loss: 72.5135, Val Loss: 71.0347
Epoch [2800/4000], Train Loss: 72.2039, Val Loss: 73.5950
Epoch [2810/4000], Train Loss: 72.3273, Val Loss: 70.3539
Epoch [2820/4000], Train Loss: 72.8485, Val Loss: 72.0424
Epoch [2830/4000], Train Loss: 71.7018, Val Loss: 70.4526
Epoch [2840/4000], Train Loss: 70.7951, Val Loss: 73.2022
Epoch [2850/4000], Train Loss: 70.1997, Val Loss: 72.3059
Epoch [2860/4000], Train Loss: 69.9281, Val Loss: 71.0135
Epoch [2870/4000], Train Loss: 69.8212, Val Loss: 72.1294
Epoch [2880/4000], Train Loss: 69.2205, Val Loss: 70.7717
Epoch [2890/4000], Train Loss: 69.1564, Val Loss: 72.5951
Epoch [2900/4000], Train Loss: 68.2798, Val Loss: 71.3689
Epoch [2910/4000], Train Loss: 68.0547, Val Loss: 71.6414
Epoch [2920/4000], Train Loss: 67.5505, Val Loss: 72.8377
Epoch [2930/4000], Train Loss: 67.5432, Val Loss: 71.0826
Epoch [2940/4000], Train Loss: 67.1700, Val Loss: 72.1346
Epoch [2950/4000], Train Loss: 66.7856, Val Loss: 70.2226
Epoch [2960/4000], Train Loss: 66.1184, Val Loss: 70.9697
Epoch [2970/4000], Train Loss: 66.1702, Val Loss: 74.3410
Epoch [2980/4000], Train Loss: 65.8824, Val Loss: 73.6123
Epoch [2990/4000], Train Loss: 65.5826, Val Loss: 73.1546
Epoch [3000/4000], Train Loss: 65.1834, Val Loss: 70.7262
Epoch [3010/4000], Train Loss: 64.7049, Val Loss: 71.1611
```

```
Epoch [3020/4000], Train Loss: 64.1906, Val Loss: 70.3399
Epoch [3030/4000], Train Loss: 63.6143, Val Loss: 71.2310
Epoch [3040/4000], Train Loss: 63.9896, Val Loss: 69.8181
Epoch [3050/4000], Train Loss: 63.7118, Val Loss: 71.8857
Epoch [3060/4000], Train Loss: 63.2881, Val Loss: 68.5068
Epoch [3070/4000], Train Loss: 62.9631, Val Loss: 70.0213
Epoch [3080/4000], Train Loss: 62.2313, Val Loss: 69.1991
Epoch [3090/4000], Train Loss: 62.4382, Val Loss: 73.1480
Epoch [3100/4000], Train Loss: 61.7150, Val Loss: 71.7592
Epoch [3110/4000], Train Loss: 61.2196, Val Loss: 68.6339
Epoch [3120/4000], Train Loss: 60.8892, Val Loss: 68.6813
Epoch [3130/4000], Train Loss: 60.5875, Val Loss: 70.5614
Epoch [3140/4000], Train Loss: 60.7052, Val Loss: 74.2106
Epoch [3150/4000], Train Loss: 60.0216, Val Loss: 70.5668
Epoch [3160/4000], Train Loss: 59.9391, Val Loss: 72.6791
Epoch [3170/4000], Train Loss: 59.1286, Val Loss: 68.5612
Epoch [3180/4000], Train Loss: 58.9806, Val Loss: 70.7640
Epoch [3190/4000], Train Loss: 58.6160, Val Loss: 68.0443
Epoch [3200/4000], Train Loss: 58.5205, Val Loss: 68.0781
Epoch [3210/4000], Train Loss: 58.2038, Val Loss: 70.3584
Epoch [3220/4000], Train Loss: 57.5880, Val Loss: 70.8389
Epoch [3230/4000], Train Loss: 57.2102, Val Loss: 69.5093
Epoch [3240/4000], Train Loss: 57.0882, Val Loss: 70.0921
Epoch [3250/4000], Train Loss: 56.9717, Val Loss: 69.6033
Epoch [3260/4000], Train Loss: 56.3433, Val Loss: 71.2625
Epoch [3270/4000], Train Loss: 56.3955, Val Loss: 71.2912
Epoch [3280/4000], Train Loss: 55.7468, Val Loss: 70.1235
Epoch [3290/4000], Train Loss: 56.0993, Val Loss: 71.4007
Epoch [3300/4000], Train Loss: 55.3590, Val Loss: 69.3364
Epoch [3310/4000], Train Loss: 55.0446, Val Loss: 71.2662
Epoch [3320/4000], Train Loss: 55.6800, Val Loss: 76.7160
Epoch [3330/4000], Train Loss: 55.0863, Val Loss: 69.8083
Epoch [3340/4000], Train Loss: 54.1134, Val Loss: 69.4163
Epoch [3350/4000], Train Loss: 54.0234, Val Loss: 68.9033
Epoch [3360/4000], Train Loss: 53.5544, Val Loss: 71.2739
Epoch [3370/4000], Train Loss: 53.7395, Val Loss: 70.9363
Early stopping triggered at epoch 3371

Final Test Metrics:
test_loss: 32.5966
rmse: 5.7093
r2: 0.8888
bias: -0.2987
```

```
rpd: 2.9992
```




```
Root Mean Squared Error (RMSE): 5.7093
R²: 0.8888
Bias: -0.2987
RPD: 2.9992
```



4.10 3. Overview for Model Improvement Strategies (5 P per strategy)

4.10.1 Model Improvement Strategies

To enhance the baseline model, we have implemented three distinct strategies and evaluated using the same independent test set for validation.

4.10.1.1 Strategy 1: Varying Preprocessing Techniques

1. **Savitzky-Golay Filter:**
   - **Process:** Applied the Savitzky-Golay filter to both calibration and test data to smooth the spectral data.
   - **Metrics:**

- RMSE: 10.1694
- R^2: 0.6473
- Bias: -0.1207
- RPD: 1.6838

2. **Standard Normal Variate (SNV):**
   - **Process:** Applied SNV to normalize spectral data, reducing variability due to differences in illumination.
   - **Metrics:**
     - RMSE: 10.1517
     - R^2: 0.6485
     - Bias: 0.3009
     - RPD: 1.6867

3. **Absorbance Transformation:**
   - **Process:** Converted reflectance data to absorbance to enhance spectral features.
   - **Metrics:**
     - RMSE: 8.6069
     - R^2: 0.7474
     - Bias: 0.1506
     - RPD: 1.9895

Overview: Preprocessing techniques such as the Savitzky-Golay filter, SNV, and absorbance transformation were applied to improve model performance. The Savitzky-Golay Filter, as well as the SNV-Transfomration did not lead to an improvement of the results. Transforming reflactance to pseudo-absorbance however, led to a strong increase in performance compard to the baseline model, from an RPD of 1.703 to 1.989. So, in further stragies we have used absorbances as independened variables.

4.10.1.2 Strategy 2: Testing Different Models

1. **PyTorch LSTM:**
   - **Process:** Implemented an LSTM model using PyTorch to capture temporal dependencies in spectral data. The LSTM model was designed to leverage the sequential nature of spectral measurements to improve predictive accuracy.
   - **Metrics:**
     - **RMSE:** 18.0296
     - **R^2:** -0.1087
     - **Bias:** 5.6444
     - **RPD:** 0.9497

2. **LSTM with PLSR Components:**

- **Process:** Combined PLSR with LSTM to leverage both dimensionality reduction and temporal modeling. Here, we used the optimal number of components obtained from PLSR, transformed the data, and applied LSTM on the transformed latent variables. This approach aimed to benefit from the strengths of both PLSR and LSTM.
- **Metrics:**
  - **RMSE:** 6.2590
  - **R²:** 0.8664
  - **Bias:** -0.4263
  - **RPD:** 2.7358

3. **PLSR + AutoML (AutoGluon):**

   - **Process:** Utilized AutoGluon, an automated machine learning library, to optimize model selection and hyperparameters. AutoGluon selected NeuralNetFastAI_r191 as the top model on the leaderboard. We applied PLSR in combination with NeuralNetFastAI_r191 to enhance predictive performance.
   - **Reference:** AutoGluon Documentation
   - **Metrics:**
     - **RMSE:** 6.5658
     - **R²:** 0.8530
     - **Bias:** -0.5860
     - **RPD:** 2.6080

Overview: Different modeling approaches, including LSTM and AutoML, were tested to improve predictive performance. While the direct application of LSTM did not yield satisfactory results so we tried to combine PLSR with LSTM in 2nd part of stragety 2 which yieded good result and using AutoML has given satisfactory results with three lines of code.

4.10.1.3 Strategy 3: Incorporating Auxiliary Data

1. **DLR Auxiliary Data:**

   - **Source:** The auxiliary data was sourced from the German Aerospace Center (DLR), specifically from their soil suite collection for Europe spanning 2018 to 2022.
   - **Data Description:** This dataset includes various soil properties and spectral measurements, providing a comprehensive view of soil characteristics across Europe.
   - **Integration:** The DLR data was integrated with the existing spectral data to enhance the model by providing additional features that capture soil variability and properties.
   - **Metrics:**
     - **RMSE:** 6.0841
     - **R²:** 0.8738

- **Bias:** -0.3307
- **RPD:** 2.8144

2. **ISRIC Soil Data:**

- **Source:** The ISRIC (International Soil Reference and Information Centre) soil data was obtained from SoilGrids, a global soil information system.
- **Data Description:** This dataset includes detailed soil properties such as bulk density, clay content, pH, sand, and silt content at various soil depths. Care was taken not to include the parameter , which would be close to SOC.
- **Integration:** The ISRIC data was merged with the spectral data to provide a richer set of features, allowing the model to better understand and predict soil organic carbon (SOC) based on both spectral and soil property information.
- **Metrics:**
  - **RMSE:** 5.7093
  - **R^2:** 0.8888
  - **Bias:** -0.2987
  - **RPD:** 2.9992

Overview: Incorporating auxiliary data from DLR and ISRIC improved model performance by providing additional context and features for more accurate predictions. These datasets stragegy worked relatively better, as it include more information

4.10.2 Consolidated Evaluation Metrics

Strategy	RMSE	R^2	Bias	RPD
Baseline Model	10.0547	0.6552	-0.1772	1.7030
Savitzky-Golay Filter	10.1694	0.6473	**-0.1207**	1.6838
Standard Normal Variate	10.1517	0.6485	0.3009	1.6867
Absorbance Transformation	8.6069	0.7474	0.1506	1.9895
PyTorch LSTM	18.0296	-0.1087	5.6444	0.9497
LSTM with PLSR	6.2590	***0.8664***	-0.4263	***2.7358***
PLSR + AutoML	6.5658	0.8530	-0.5860	2.6080
DLR Auxiliary Data	6.0841	0.8738	-0.3307	2.8144
ISRIC Soil Data	**5.7093**	**0.8888**	-0.2987	**2.9992**

Conclusion: As explained before , Incorporating auxilary data **ISRIC Soil Data** worked best compared to all strageties , and if we dont consider , inclusion of new data , then **LSTM with PLSR** worked relatively better.

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/S2

5.0.1 4.Discussion of Results

5.0.1.1 Overview of Modeling Approaches

In this study, various modeling approaches were explored to predict soil organic carbon (SOC) using soil VNIR reflectance spectroscopy. The strategies included traditional machine learning techniques such as Partial Least Squares Regression (PLSR), as well as advanced deep learning models like Long Short-Term Memory networks (LSTM). Each approach was evaluated based on validation metrics such as RMSE, R^2, bias, and RPD.

5.0.1.2 Performance Comparison

- **PLSR:** The baseline PLSR model demonstrated moderate performance with an RMSE of 10.0547 and an R^2 of 0.6552. This model served as a benchmark for evaluating the effectiveness of more complex models.
- **Deep Learning Models:**
  - **LSTM:** The standalone LSTM model did not perform well, with an RMSE of 18.0296 and a negative R^2 value, indicating poor predictive accuracy.
  - **LSTM with PLSR Components:** Combining LSTM with PLSR significantly improved performance, achieving an RMSE of 6.2590 and an R^2 of 0.8664. This approach leveraged the strengths of both dimensionality reduction and temporal modeling.
  - **PLSR + AutoML (AutoGluon):** Utilizing AutoGluon for automated machine learning further enhanced predictive accuracy, with an RMSE of 6.5658 and an R^2 of 0.8530.
- **Auxiliary Data Integration:**
  - **DLR Auxiliary Data:** Incorporating DLR auxiliary data improved model performance, resulting in an RMSE of 6.0841 and an R^2 of 0.8738.
  - **ISRIC Soil Data:** The integration of ISRIC soil data yielded the best results, with an RMSE of 5.7093 and an R^2 of 0.8888.

5.0.1.3 Comparison with Published Studies

The following table compares the performance metrics of different models from this study with those reported in similar published studies:

Study/Model	RMSE	R^2	Bias	RPD	Reference
Baseline PLSR	10.0547	0.6552	-0.1772	1.7030	This Study
LSTM	18.0296	-0.1087	5.6444	0.9497	This Study
LSTM with PLSR	6.2590	0.8664	-0.4263	2.7358	This Study
PLSR + AutoML	6.5658	0.8530	-0.5860	2.6080	This Study
DLR Auxiliary Data	6.0841	0.8738	-0.3307	2.8144	This Study
ISRIC Soil Data	5.7093	0.8888	-0.2987	2.9992	This Study
AutoEncoders (2021)	10.40	0.87	-	-	AutoEncoders Paper
CNN with SG and SNV (2024)	8.48	0.81	-	-	CNN with SG and SNV Paper
N-MBL (2024)	6.21	0.70	-	2.12	N-MBL Paper
SAE-1DCNN (2024)	3.935%	0.784	-	3.914	SAE-1DCNN Paper
HKF-GPR (2024)	5.29	0.7671	-	2.5789	HKF-GPR Paper
LSTM-CNN (2024)	1.66	0.96	-	-	LSTM-CNN Paper

5.0.1.4 Reflection on Soil VNIR Spectroscopy

Soil VNIR reflectance spectroscopy shows promise as a complementary approach for large-scale SOC assessment. Key considerations include:

- **Model Accuracy:** Achieving high accuracy is crucial for reliable SOC predictions. Our best model, which integrated ISRIC soil data, demonstrated strong performance with an R^2 of 0.8888, indicating its potential for practical applications.
- **Data Harmonization:** Combining datasets from different providers poses challenges due to variations in data quality and format. Standardizing preprocessing techniques, such as Savitzky-Golay smoothing and SNV transformation, can help mitigate these issues.
- **Practical Usability:** While advanced models like LSTM and CNN offer superior performance, they require significant computational resources and expertise. Simplifying

these models or providing user-friendly interfaces could enhance their accessibility for end users.

In conclusion, soil VNIR spectroscopy, combined with advanced modeling techniques and auxiliary data integration, holds significant potential for improving large-scale soil information systems. Future work should focus on refining these models and addressing data harmonization challenges to enhance their practical usability.

Part II

Appendix

6 Getting DLR Data

```
import rioxarray as rxr
import xarray as xr
from odc.stac import load
import pystac_client
import pandas as pd
import geopandas as gpd
import matplotlib.pyplot as plt
import numpy as np
from shapely.geometry import Point
from concurrent.futures import ThreadPoolExecutor, as_completed
from tqdm import tqdm
import os
```

```
# Create cache directory
CACHE_DIR = "data/stac_data_cache_full"
os.makedirs(CACHE_DIR, exist_ok=True)
```

```
def create_bbox(lon, lat, step=0.000001):
    """Create a bounding box around a point."""
    return [lon - step, lat - step, lon + step, lat + step]

def get_stac_data_for_point(args):
    """Process a single point (for parallel execution)"""
    catalog, collection, measurements, point_idx, lon, lat, bbox_step = args

    # Check if cached result exists
    cache_file = f"{CACHE_DIR}/point_{lon}_{lat}.parquet"
    if os.path.exists(cache_file):
        try:
            return pd.read_parquet(cache_file)
        except Exception as e:
            print(f"Error reading cache file {cache_file}: {e}")
            pass  # If cache read fails, continue with regular processing
```

```python
        try:
            # Create bounding box for this point
            bbox = create_bbox(lon, lat, bbox_step)

            # Search for items
            search = catalog.search(
                collections=collection,
                bbox=bbox,
                datetime="2018-03-01/2020-12-31"
            )

            # Convert search results to list
            items = list(search.items())

            if len(items) > 0:
                # Load the data
                dataset = load(
                    items,
                    measurements=measurements,
                    bbox=bbox,
                    resolution=20
                )

                # Convert to dataframe
                data_point = dataset.isel(time=0).to_dataframe().reset_index()

                # Add point metadata
                data_point['point_index'] = point_idx
                data_point['source_lon'] = lon
                data_point['source_lat'] = lat

                # Cache the result
                try:
                    data_point.to_parquet(cache_file)
                except Exception as e:
                    print(f"Error caching point {point_idx}: {e}")
                    pass  # If caching fails, continue anyway

                return data_point
            else:
                print(f"No items found for point {point_idx} ({lon}, {lat})")
```

```
            return None
    except Exception as e:
        print(f"Error processing point {point_idx}: {e}")
        return None

def get_all_auxiliary_data(catalog, collection, measurements, long_lat, bbox_step=0.000001
    """Retrieve STAC data for all points using parallel processing."""

    # Prepare arguments for each point
    args_list = []

    # Create argument list for all points
    for i in range(len(long_lat)):
        args_list.append((
            catalog,
            collection,
            measurements,
            i,  # Point index
            long_lat.iloc[i]['GPS_LONG'],
            long_lat.iloc[i]['GPS_LAT'],
            bbox_step
        ))

    # Process points in parallel
    results = []
    with ThreadPoolExecutor(max_workers=max_workers) as executor:
        # Submit all tasks and track with progress bar
        futures = [executor.submit(get_stac_data_for_point, args) for args in args_list]

        for future in tqdm(as_completed(futures), total=len(args_list), desc="Processing p
            result = future.result()
            if result is not None:
                results.append(result)

    # Combine all results
    if not results:
        print("No data retrieved!")
        return None

    points_df = pd.concat(results, ignore_index=True)
```

```python
    # Create geometry points for GeoDataFrame
    geometry_points = [Point(x, y) for x, y in zip(points_df['x'], points_df['y'])]

    # Convert to GeoDataFrame
    points_gdf = gpd.GeoDataFrame(points_df, geometry=geometry_points, crs=3035)

    return points_gdf
```

```python
dlr_measurements = ["MREF_B02", "MREF_B03", "MREF_B04", "MREF_B05", "MREF_B06", "MREF_B07"
                        "MREF-STD_B02", "MREF-STD_B03", "MREF-STD_B04", "MREF-STD_B05", "M
                        "SRC_B02", "SRC_B03", "SRC_B04", "SRC_B05", "SRC_B06", "SRC_B07",
                        "SRC-STD_B02", "SRC-STD_B03", "SRC-STD_B04", "SRC-STD_B05", "SRC-S
                        "SRC-CI95_B02", "SRC-CI95_B03", "SRC-CI95_B04", "SRC-CI95_B05", "S
                        "SFREQ-BSF" #, "SFREQ-BSC", "SFREQ-VPC"
                        ]
```

```python
# Load your data
target_raw = pd.read_csv('data/France_lab.csv')
long_lat = target_raw[['GPS_LONG', 'GPS_LAT']]

# Initialize STAC catalog
dlr_catalog = pystac_client.Client.open("https://geoservice.dlr.de/eoc/ogc/stac/v1")

# Define measurements (you can reduce this list if you don't need all bands)
# dlr_measurements = ["MREF_B02", "MREF_B03", "MREF_B04", "MREF_B08", "MREF_B11", "MREF_B1

# Define collection
dlr_collection = ["S2-soilsuite-europe-2018-2022-P5Y"]
```

```python
# # Process all points (consider using a subset for testing: long_lat.iloc[:10])
# results = get_all_auxiliary_data(
#     catalog=dlr_catalog,
#     collection=dlr_collection,
#     measurements=dlr_measurements,
#     long_lat=long_lat,
#     bbox_step=0.000001,
#     max_workers=4  # Adjust based on your CPU and bandwidth
# )

# # Save the results
```

```
# if results is not None:
#     results.to_parquet("data/auxiliary_data_results_full.parquet")
#     print("Data saved successfully")
```

```
Processing points:  15%|          | 413/2807 [19:17<1:14:54,  1.88s/it]

Error processing point 416: ('Connection aborted.', RemoteDisconnected('Remote end closed co

Processing points:  18%|          | 493/2807 [23:15<2:07:44,  3.31s/it]

Error processing point 497: ('Connection aborted.', RemoteDisconnected('Remote end closed co

Processing points:  18%|          | 511/2807 [24:02<1:18:04,  2.04s/it]

No items found for point 513 (88.888888, 88.888888)

Processing points:  33%|         | 933/2807 [43:10<1:02:52,  2.01s/it]

No items found for point 935 (88.888888, 88.888888)

Processing points:  59%|        | 1668/2807 [1:16:21<36:45,  1.94s/it]

Error processing point 1671: ('Connection aborted.', RemoteDisconnected('Remote end closed c

Processing points:  78%|       | 2201/2807 [1:40:42<22:58,  2.27s/it]

Error processing point 2204: ('Connection aborted.', RemoteDisconnected('Remote end closed c

Processing points:  86%|       | 2406/2807 [1:50:21<18:09,  2.72s/it]

Error processing point 2410: ('Connection aborted.', RemoteDisconnected('Remote end closed c

Processing points:  92%|      | 2582/2807 [1:58:41<14:03,  3.75s/it]

Error processing point 2586: ('Connection aborted.', RemoteDisconnected('Remote end closed c

Processing points: 100%|      | 2807/2807 [2:08:47<00:00,  2.75s/it]

Data saved successfully
```

6.1 Update missing data

```
def update_missing_dlr_data(
    target_raw_path='data/France_lab.csv',
    input_parquet="data/auxiliary_data_results_full.parquet",
    output_parquet="data/auxiliary_data_results_updated.parquet",
    cache_dir="data/stac_data_cache_full",
    max_workers=4,
    max_retries=3
):
    """
    Find and update missing DLR data points

    Args:
        target_raw_path: Path to the CSV containing all target points (with GPS_LONG, GPS_
        input_parquet: Path to the existing parquet file containing the processed results
        output_parquet: Path to save the updated parquet file (if None, overwrites input_p
        cache_dir: Directory containing cached point data
        max_workers: Maximum number of parallel workers
        max_retries: Maximum number of retry attempts for each point

    Returns:
        Updated GeoDataFrame with all available points
    """
    import pandas as pd
    import geopandas as gpd
    import os
    import time
    import random
    from shapely.geometry import Point
    from concurrent.futures import ThreadPoolExecutor, as_completed
    from tqdm import tqdm
    import pystac_client

    if output_parquet is None:
        output_parquet = input_parquet

    # Load original target points
    print(f"Loading original target points from {target_raw_path}")
    target_raw = pd.read_csv(target_raw_path)
    original_points = target_raw[['GPS_LONG', 'GPS_LAT']]
```

```python
    # Try to load existing results
    if os.path.exists(input_parquet):
        try:
            print(f"Loading existing results from {input_parquet}")
            existing_data = gpd.read_parquet(input_parquet)
            print(f"Loaded {len(existing_data)} points from existing data")

            # Extract unique source coordinates from the existing data
            processed_coords = set(zip(existing_data['source_lon'], existing_data['source_
            print(f"Found {len(processed_coords)} unique processed coordinates")
        except Exception as e:
            print(f"Error loading existing results: {e}")
            existing_data = None
            processed_coords = set()
    else:
        print(f"No existing results found at {input_parquet}")
        existing_data = None
        processed_coords = set()

    # Identify missing points
    missing_points = []
    for idx, row in original_points.iterrows():
        point_coord = (row['GPS_LONG'], row['GPS_LAT'])
        if point_coord not in processed_coords:
            missing_points.append((idx, point_coord[0], point_coord[1]))

    print(f"Found {len(missing_points)} missing points out of {len(original_points)} total

    if not missing_points:
        print("No missing points to process!")
        return existing_data

    # Initialize STAC catalog
    dlr_catalog = pystac_client.Client.open("https://geoservice.dlr.de/eoc/ogc/stac/v1")

    # Define collection
    dlr_collection = ["S2-soilsuite-europe-2018-2022-P5Y"]

    # Define measurements
    dlr_measurements = [
        "MREF_B02", "MREF_B03", "MREF_B04", "MREF_B05", "MREF_B06", "MREF_B07", "MREF_B08"
```

```
    "MREF-STD_B02", "MREF-STD_B03", "MREF-STD_B04", "MREF-STD_B05", "MREF-STD_B06", "M
    "SRC_B02", "SRC_B03", "SRC_B04", "SRC_B05", "SRC_B06", "SRC_B07", "SRC_B08", "SRC_
    "SRC-STD_B02", "SRC-STD_B03", "SRC-STD_B04", "SRC-STD_B05", "SRC-STD_B06", "SRC-ST
    "SRC-CI95_B02", "SRC-CI95_B03", "SRC-CI95_B04", "SRC-CI95_B05", "SRC-CI95_B06", "S
    "SFREQ-BSF"  # Surface Frequency - Bare Soil Frequency
]

# Create cache directory if it doesn't exist
os.makedirs(cache_dir, exist_ok=True)

# Function to process a single point with retries
def process_point_with_retries(point_info):
    idx, lon, lat = point_info

    for retry in range(max_retries):
        try:
            # Create arguments for the get_stac_data_for_point function
            args = (
                dlr_catalog,
                dlr_collection,
                dlr_measurements,
                idx,
                lon,
                lat,
                0.000001  # bbox_step
            )

            result = get_stac_data_for_point(args)
            if result is not None:
                print(f"Successfully processed point {idx} ({lon}, {lat})")
                return result

            print(f"No data found for point {idx} ({lon}, {lat}) - Attempt {retry+1}/{

            # Add randomized delay between retries
            if retry < max_retries - 1:
                sleep_time = 5 + random.random() * 10
                print(f"Waiting {sleep_time:.1f} seconds before retry...")
                time.sleep(sleep_time)

        except Exception as e:
```

```
                print(f"Error processing point {idx} ({lon}, {lat}) - Attempt {retry+1}/{m

                if retry < max_retries - 1:
                    sleep_time = 5 + random.random() * 10
                    print(f"Waiting {sleep_time:.1f} seconds before retry...")
                    time.sleep(sleep_time)

        print(f"Failed to process point {idx} ({lon}, {lat}) after {max_retries} attempts"
        return None

    # Process missing points in parallel
    new_results = []
    with ThreadPoolExecutor(max_workers=max_workers) as executor:
        # Submit all tasks and track with progress bar
        futures = [executor.submit(process_point_with_retries, point) for point in missing

        for future in tqdm(as_completed(futures), total=len(missing_points), desc="Process
            result = future.result()
            if result is not None:
                new_results.append(result)

    print(f"Successfully processed {len(new_results)} out of {len(missing_points)} missing

    # Combine with existing results
    if new_results:
        # Combine all new results
        new_points_df = pd.concat(new_results, ignore_index=True)

        # Create geometry points for GeoDataFrame
        geometry_points = [Point(x, y) for x, y in zip(new_points_df['x'], new_points_df['

        # Convert to GeoDataFrame
        new_points_gdf = gpd.GeoDataFrame(new_points_df, geometry=geometry_points, crs=303

        # Combine with existing data if available
        if existing_data is not None:
            combined_gdf = pd.concat([existing_data, new_points_gdf], ignore_index=True)
        else:
            combined_gdf = new_points_gdf

        # Save the updated results
```

```python
        combined_gdf.to_parquet(output_parquet)
        print(f"Updated data saved to {output_parquet} ({len(combined_gdf)} total points)"

        return combined_gdf
    else:
        print("No new data to add")
        return existing_data
```

```python
# Load your original data points
target_raw = pd.read_csv('data/France_lab.csv')
long_lat = target_raw[['GPS_LONG', 'GPS_LAT']]

# Update missing points
updated_results = update_missing_dlr_data(
    target_raw_path='data/France_lab.csv',
    input_parquet="data/auxiliary_data_results_full.parquet",
    output_parquet="data/auxiliary_data_results_full_updated.parquet",  # Optional: set to
    cache_dir="data/stac_data_cache_full",
    max_workers=4,
    max_retries=3
)
```

```
Loading original target points from data/France_lab.csv
Loading existing results from data/auxiliary_data_results_full.parquet
Loaded 2799 points from existing data
Found 2797 unique processed coordinates
Found 8 missing points out of 2807 total points

Processing missing points:   0%|          | 0/8 [00:00<?, ?it/s]

Successfully processed point 497 (0.57576, 46.423743)Successfully processed point 416 (-1.27

Successfully processed point 1671 (-2.833854, 48.198354)
Successfully processed point 2204 (3.056281, 46.46886)
Successfully processed point 2410 (-2.852118, 47.522143)
Successfully processed point 2586 (-0.317878, 48.826356)
No items found for point 513 (88.888888, 88.888888)
No data found for point 513 (88.888888, 88.888888) - Attempt 1/3
Waiting 13.7 seconds before retry...
No items found for point 935 (88.888888, 88.888888)
No data found for point 935 (88.888888, 88.888888) - Attempt 1/3
```

```
Waiting 13.2 seconds before retry...
No items found for point 935 (88.888888, 88.888888)
No data found for point 935 (88.888888, 88.888888) - Attempt 2/3
Waiting 7.1 seconds before retry...
No items found for point 513 (88.888888, 88.888888)
No data found for point 513 (88.888888, 88.888888) - Attempt 2/3
Waiting 8.5 seconds before retry...

Processing missing points:  88%|     | 7/8 [00:21<00:03,  3.01s/it]

No items found for point 935 (88.888888, 88.888888)
No data found for point 935 (88.888888, 88.888888) - Attempt 3/3
Failed to process point 935 (88.888888, 88.888888) after 3 attempts

Processing missing points: 100%|     | 8/8 [00:22<00:00,  2.84s/it]

No items found for point 513 (88.888888, 88.888888)
No data found for point 513 (88.888888, 88.888888) - Attempt 3/3
Failed to process point 513 (88.888888, 88.888888) after 3 attempts
Successfully processed 6 out of 8 missing points
Updated data saved to data/auxiliary_data_results_full_updated.parquet (2805 total points)
```

7 Getting Soil Grid Data

```
import pandas as pd
import requests
import json
import time
import random
import os
import concurrent.futures
from tqdm import tqdm
import threading

# Add a lock to prevent race conditions when saving data
save_lock = threading.Lock()

def get_soilgrids_point(lon, lat, point_idx, properties=None, max_retries=3):
    """Get SoilGrids data for a single point with correct field mapping"""
    if properties is None:
        properties = ['soc', 'clay', 'sand', 'silt', 'bdod', 'phh2o']

    url = "https://rest.isric.org/soilgrids/v2.0/properties/query"
    params = {
        'lon': lon,
        'lat': lat,
        'property': properties,
        'depth': ['0-5cm', '5-15cm', '15-30cm'],
        'value': ['mean']
    }

    for retry in range(max_retries):
        try:
            response = requests.get(url, params=params)

            # Handle rate limiting
            if response.status_code == 429:
                wait_time = 15 + random.random() * 15
```

```
            #print(f"Rate limited for point {point_idx}, waiting {wait_time:.1f} secon
            time.sleep(wait_time)
            continue

        if response.status_code == 200:
            data = response.json()

            # Start with basic info
            result = {'point_index': point_idx, 'lon': lon, 'lat': lat}

            # Extract data using the correct field structure
            if 'properties' in data and 'layers' in data['properties']:
                for layer in data['properties']['layers']:
                    # Get property name
                    prop_name = layer.get('name', 'unknown')

                    for depth in layer.get('depths', []):
                        # Get depth label (which is the string format we need)
                        depth_label = depth.get('label', 'unknown')

                        # Clean the depth label for column naming
                        clean_depth = depth_label.replace('-', '_to_')

                        # Extract values
                        for value_type, value in depth.get('values', {}).items():
                            column_name = f"{prop_name}_{clean_depth}_{value_type}"
                            result[column_name] = value

            # Debug print to verify data is being captured correctly
            #print(f"Retrieved data for point {point_idx}: {lon}, {lat}")
            return result
        else:
            #print(f"Error for point {point_idx}: Status code {response.status_code}")
            if retry < max_retries - 1:
                wait_time = 10 * (retry + 1)
                #print(f"Retrying in {wait_time} seconds...")
                time.sleep(wait_time)
            else:
                return {'point_index': point_idx, 'lon': lon, 'lat': lat,
                        'error': f"Status {response.status_code}"}
```

```
        except Exception as e:
            #print(f"Exception for point {point_idx}: {str(e)}")
            if retry < max_retries - 1:
                wait_time = 10 * (retry + 1)
                #print(f"Retrying in {wait_time} seconds...")
                time.sleep(wait_time)
            else:
                return {'point_index': point_idx, 'lon': lon, 'lat': lat,
                        'error': f"Exception: {str(e)}"}

    return {'point_index': point_idx, 'lon': lon, 'lat': lat,
            'error': "Max retries reached"}

def process_point(args):
    """Wrapper function for concurrent processing"""
    lon, lat, idx, properties = args
    # Add jitter to avoid all workers hitting the API simultaneously
    time.sleep(random.random() * 2)
    return get_soilgrids_point(lon, lat, idx, properties)

def save_checkpoint(results, filename, verbose=False):
    """Save results to a checkpoint file using a lock to prevent race conditions"""
    with save_lock:
        try:
            df_results = pd.DataFrame(results)
            # First write to a temporary file, then rename to avoid partial writes
            temp_file = f"{filename}.temp"
            df_results.to_csv(temp_file, index=False)
            os.replace(temp_file, filename)
            if verbose:
                print(f"Saved checkpoint with {len(results)} points to {filename}")
        except Exception as e:
            print(f"Error saving checkpoint: {str(e)}")

def get_soilgrids_parallel(coordinates_df, num_workers=4, lon_col='GPS_LONG', lat_col='GPS
                           properties=None, cache_file='soilgrids_parallel.csv',
                           checkpoint_interval=10, debug=False):
    """
    Retrieve soil data for multiple points in parallel using multiple workers

    Args:
```

```
        coordinates_df: DataFrame with coordinates
        num_workers: Number of parallel workers (default: 4)
        lon_col: Column name for longitude
        lat_col: Column name for latitude
        properties: List of SoilGrids properties to retrieve
        cache_file: Output file name
        checkpoint_interval: Save intermediate results every N points
        debug: Enable additional debug output
    """
    if properties is None:
        properties = ['soc', 'clay', 'sand', 'silt', 'bdod', 'phh2o']

    # Print the input data to verify it's correct
    if debug:
        print("Input coordinate data sample:")
        print(coordinates_df.head())
        print(f"Longitude column: {lon_col}, Latitude column: {lat_col}")

    # Check for existing cache to resume from
    results = []

    if os.path.exists(cache_file):
        try:
            existing_df = pd.read_csv(cache_file)
            if len(existing_df) > 0:
                results = existing_df.to_dict('records')
                processed_indices = set(existing_df['point_index'].unique())
                print(f"Found {len(processed_indices)} already processed points in {cache_
                coordinates_df = coordinates_df[~coordinates_df.index.isin(processed_indic
                print(f"Remaining points to process: {len(coordinates_df)}")
        except Exception as e:
            print(f"Error reading existing cache: {str(e)}. Starting from scratch.")

    if len(coordinates_df) == 0:
        print("All points already processed!")
        return pd.DataFrame(results)

    # Prepare arguments for parallel processing
    args_list = []
    for idx, row in coordinates_df.iterrows():
        # Verify and clean coordinate values
```

```
        try:
            lon = float(row[lon_col])
            lat = float(row[lat_col])
            args_list.append((lon, lat, idx, properties))
            if debug and len(args_list) <= 5:
                print(f"Prepared point {idx}: lon={lon}, lat={lat}")
        except (ValueError, TypeError) as e:
            print(f"Error with coordinates at index {idx}: {e}")
            print(f"Row data: {row}")

    print(f"Processing {len(args_list)} points with {num_workers} workers")

    completed_count = 0

    # Use ThreadPoolExecutor for parallel HTTP requests
    with concurrent.futures.ThreadPoolExecutor(max_workers=num_workers) as executor:
        # Submit all tasks
        future_to_args = {executor.submit(process_point, args): args for args in args_list

        # Use tqdm for a progress bar
        for future in tqdm(concurrent.futures.as_completed(future_to_args), total=len(args
            args = future_to_args[future]
            point_idx = args[2]

            try:
                result = future.result()
                if result:
                    results.append(result)
                    completed_count += 1

                    # Save intermediate results periodically
                    if completed_count % checkpoint_interval == 0:
                        save_checkpoint(results, cache_file)

            except Exception as e:
                print(f"\nError processing point {point_idx}: {str(e)}")

    # Save final results
    save_checkpoint(results, cache_file, verbose=False)

    # Verify the final output
```

```python
    try:
        final_df = pd.read_csv(cache_file)
        print(f"Final output has {len(final_df)} rows and {len(final_df.columns)} columns"
        print("Column names:", final_df.columns.tolist())
        print("First few rows:")
        print(final_df.head())
    except Exception as e:
        print(f"Error verifying final output: {str(e)}")

    return pd.DataFrame(results)

# Example usage:
# df = pd.read_csv('coordinates.csv', index_col=0)  # Set the first column as index if tha
# results = get_soilgrids_parallel(df, num_workers=4)
```

```python
# Load your data
target_raw = pd.read_csv('data/France_lab.csv')
long_lat = target_raw[['GPS_LONG', 'GPS_LAT']]
get_soilgrids_parallel(long_lat, num_workers=4, properties=['soc', 'clay', 'sand', 'silt',
```

```
Found 670 already processed points in soilgrids_parallel.csv
Remaining points to process: 2137
Processing 2137 points with 4 workers

100%|     | 2137/2137 [1:41:01<00:00,  2.84s/it]

Final output has 2807 rows and 22 columns
Column names: ['point_index', 'lon', 'lat', 'bdod_0_to_5cm_mean', 'bdod_5_to_15cm_mean', 'bdo
First few rows:
   point_index       lon        lat  bdod_0_to_5cm_mean  bdod_5_to_15cm_mean  \
0            1  4.584692  45.816720               125.0                134.0
1            0  4.680379  45.893933               128.0                138.0
2            3  4.601575  45.908022               133.0                140.0
3            2  4.671533  45.983716               129.0                139.0
4            6  4.439863  46.224665               102.0                116.0

   bdod_15_to_30cm_mean  clay_0_to_5cm_mean  clay_5_to_15cm_mean  \
0                 141.0               250.0                270.0
1                 141.0               303.0                319.0
2                 144.0               247.0                268.0
3                 143.0               249.0                254.0
```

```
4                121.0                188.0                  172.0

   clay_15_to_30cm_mean  phh2o_0_to_5cm_mean  ...  sand_0_to_5cm_mean  \
0                 302.0                 58.0  ...               375.0
1                 338.0                 62.0  ...               263.0
2                 288.0                 60.0  ...               334.0
3                 300.0                 63.0  ...               305.0
4                 205.0                 52.0  ...               505.0

   sand_5_to_15cm_mean  sand_15_to_30cm_mean  silt_0_to_5cm_mean  \
0                366.0                 366.0               375.0
1                243.0                 275.0               434.0
2                320.0                 329.0               418.0
3                295.0                 313.0               446.0
4                514.0                 494.0               306.0

   silt_5_to_15cm_mean  silt_15_to_30cm_mean  soc_0_to_5cm_mean  \
0                364.0                 332.0              489.0
1                438.0                 387.0              467.0
2                412.0                 383.0              401.0
3                451.0                 387.0              422.0
4                314.0                 301.0              786.0

   soc_5_to_15cm_mean  soc_15_to_30cm_mean  error
0               249.0                246.0    NaN
1               254.0                182.0    NaN
2               289.0                153.0    NaN
3               271.0                154.0    NaN
4               620.0                239.0    NaN

[5 rows x 22 columns]
```

	point_index	lon	lat	bdod_0_to_5cm_mean	bdod_5_to_15cm_mean	bdod_15_
0	1	4.584692	45.816720	125.0	134.0	141.0
1	0	4.680379	45.893933	128.0	138.0	141.0
2	3	4.601575	45.908022	133.0	140.0	144.0
3	2	4.671533	45.983716	129.0	139.0	143.0
4	6	4.439863	46.224665	102.0	116.0	121.0
...	...	...	...	...	...	...
2802	2803	5.058028	45.713629	131.0	147.0	150.0
2803	2806	4.784826	45.881063	128.0	137.0	141.0

	point_index	lon	lat	bdod_0_to_5cm_mean	bdod_5_to_15cm_mean	bdod_15_
2804	2805	4.381513	45.788303	124.0	135.0	137.0
2805	2800	4.718750	45.498638	119.0	129.0	135.0
2806	2799	4.578846	45.617959	NaN	NaN	NaN

7.1 Update

```
def update_missing_soilgrids_data(csv_file, output_file=None, max_retries=5, delay_between
    """
    Update missing data in a SoilGrids CSV file

    Args:
        csv_file: Path to the CSV file with missing data
        output_file: Path to save the updated CSV (default: overwrite input file)
        max_retries: Maximum number of retries for failed API calls
        delay_between_retries: Delay in seconds between retries

    Returns:
        DataFrame with the updated data
    """
    import pandas as pd
    import time
    import random
    import numpy as np

    if output_file is None:
        output_file = csv_file

    # Load the CSV file and force column types
    print(f"Loading data from {csv_file}...")
    df = pd.read_csv(csv_file, header=None)

    # Determine data types for all columns
    dtypes = df.dtypes
    print(f"Column data types: {dtypes}")

    # Identify rows with missing data (rows with mostly empty values)
    # Consider both NaN values and empty strings as missing
    missing_mask = ((df.iloc[:, 3:].isna()) | (df.iloc[:, 3:] == "")).sum(axis=1) > (df.sh
```

```
missing_indices = df[missing_mask].index

print(f"Found {len(missing_indices)} rows with missing data")

if len(missing_indices) == 0:
    print("No missing data to update!")
    return df

# Prepare a results list to store updated rows
updated_rows = []

# Process each row with missing data
for idx in missing_indices:
    row = df.iloc[idx]
    point_idx = row[0]
    lon = row[1]
    lat = row[2]

    print(f"Processing missing data for point {point_idx} at coordinates {lon}, {lat}"

    # Make API call with retries
    for retry in range(max_retries):
        try:
            result = get_soilgrids_point(lon, lat, point_idx)

            if 'error' in result:
                print(f"Attempt {retry+1}/{max_retries} failed: {result.get('error')}"

                # If we've reached the max retries, save what we have
                if retry == max_retries - 1:
                    print(f"Failed to update point {point_idx} after {max_retries} att
                    break

                # Wait before retrying
                sleep_time = delay_between_retries + random.random() * 10
                print(f"Retrying in {sleep_time:.1f} seconds...")
                time.sleep(sleep_time)
                continue

            # Create a new row with the correct data types
            updated_row = row.copy()
```

```
    # Set the basic fields (point_idx, lon, lat)
    # Convert to the same type as the original DataFrame to avoid warnings
    updated_row[0] = point_idx  # This should already be the correct type
    updated_row[1] = lon        # This should already be the correct type
    updated_row[2] = lat        # This should already be the correct type

    # Map the result fields to the appropriate columns in the dataframe
    soil_properties = ['soc', 'clay', 'sand', 'silt', 'bdod', 'phh2o']
    depths = ['0_to_5cm', '5_to_15cm', '15_to_30cm']

    # Assuming the columns in the original DataFrame follow this order:
    column_idx = 3  # Start after point_idx, lon, lat
    for prop in soil_properties:
        for depth in depths:
            column_name = f"{prop}_{depth}_mean"
            if column_name in result and column_idx < len(df.columns):
                # Try to match the data type
                value = result[column_name]
                if pd.api.types.is_float_dtype(dtypes[column_idx]):
                    value = float(value) if value is not None else np.nan
                elif pd.api.types.is_integer_dtype(dtypes[column_idx]):
                    value = int(value) if value is not None else np.nan
                updated_row[column_idx] = value
            column_idx += 1

    # Update the DataFrame
    df.iloc[idx] = updated_row
    print(f"Successfully updated point {point_idx}")

    # Add a small delay to avoid rate limiting
    time.sleep(2 + random.random() * 3)
    break

except Exception as e:
    print(f"Error updating point {point_idx}: {str(e)}")

    if retry < max_retries - 1:
        sleep_time = delay_between_retries + random.random() * 10
        print(f"Retrying in {sleep_time:.1f} seconds...")
        time.sleep(sleep_time)
    else:
```

```python
                        print(f"Failed to update point {point_idx} after {max_retries} attempt

    # Save the updated DataFrame
    print(f"Saving updated data to {output_file}...")
    df.to_csv(output_file, index=False, header=False)

    return df
```

```python
# Update the missing data
updated_df = update_missing_soilgrids_data(
    csv_file='data/soilgrids_parallel.csv',
    output_file='data/soilgrids_updated.csv',
    max_retries=5,
    delay_between_retries=20
)
```

```
Loading data from data/soilgrids_parallel.csv...
Column data types: 0     object
1     object
2     object
3     object
4     object
5     object
6     object
7     object
8     object
9     object
10    object
11    object
12    object
13    object
14    object
15    object
16    object
17    object
18    object
19    object
20    object
21    object
dtype: object
Found 103 rows with missing data
Processing missing data for point 155 at coordinates 0.669178, 49.855175
```

```
Successfully updated point 155
Processing missing data for point 191 at coordinates 2.743706, 48.544291
Successfully updated point 191
Processing missing data for point 190 at coordinates 2.557593, 48.513858
Successfully updated point 190
Processing missing data for point 216 at coordinates 3.131174, 48.695394
Successfully updated point 216
Processing missing data for point 250 at coordinates -0.054073, 46.837853
Successfully updated point 250
Processing missing data for point 297 at coordinates 2.737896, 49.699723
Successfully updated point 297
Processing missing data for point 338 at coordinates 5.884201, 48.020614
Successfully updated point 338
Processing missing data for point 365 at coordinates 4.987468, 44.229435
Successfully updated point 365
Processing missing data for point 398 at coordinates 1.13659, 43.97067
Successfully updated point 398
Processing missing data for point 397 at coordinates 1.33483, 44.15072
Successfully updated point 397
Processing missing data for point 396 at coordinates 1.4312, 44.17679
Successfully updated point 396
Processing missing data for point 451 at coordinates -4.271489, 47.855326
Successfully updated point 451
Processing missing data for point 466 at coordinates 0.381898, 46.769025
Successfully updated point 466
Processing missing data for point 491 at coordinates 0.237318, 46.266361
Successfully updated point 491
Processing missing data for point 513 at coordinates 88.888888, 88.888888
Successfully updated point 513
Processing missing data for point 511 at coordinates 3.00959, 48.2002
Successfully updated point 511
Processing missing data for point 512 at coordinates 3.18728, 48.08494
Successfully updated point 512
Processing missing data for point 555 at coordinates 2.046038, 48.134749
Successfully updated point 555
Processing missing data for point 574 at coordinates 6.0446, 49.39529
Successfully updated point 574
Processing missing data for point 605 at coordinates 1.121382, 45.981392
Successfully updated point 605
Processing missing data for point 622 at coordinates 1.139869, 43.299735
Successfully updated point 622
Processing missing data for point 627 at coordinates 2.207169, 46.193274
Successfully updated point 627
```

```
Processing missing data for point 668 at coordinates -1.359804, 47.234471
Successfully updated point 668
Processing missing data for point 695 at coordinates -0.3349, 45.15926
Successfully updated point 695
Processing missing data for point 722 at coordinates 1.347223, 43.571493
Successfully updated point 722
Processing missing data for point 759 at coordinates 4.11086, 43.98437
Successfully updated point 759
Processing missing data for point 762 at coordinates 4.80933, 43.98572
Successfully updated point 762
Processing missing data for point 788 at coordinates 0.089928, 43.929543
Successfully updated point 788
Processing missing data for point 787 at coordinates 0.870324, 43.911071
Successfully updated point 787
Processing missing data for point 838 at coordinates 3.1851, 43.36619
Successfully updated point 838
Processing missing data for point 856 at coordinates 5.092115, 47.879903
Successfully updated point 856
Processing missing data for point 904 at coordinates -1.875838, 48.306003
Successfully updated point 904
Processing missing data for point 935 at coordinates 88.888888, 88.888888
Successfully updated point 935
Processing missing data for point 923 at coordinates 6.510341, 48.672605
Successfully updated point 923
Processing missing data for point 1012 at coordinates 5.384717, 47.009183
Successfully updated point 1012
Processing missing data for point 1031 at coordinates 0.98026, 47.88937
Successfully updated point 1031
Processing missing data for point 1057 at coordinates 1.77986, 47.60863
Successfully updated point 1057
Processing missing data for point 1056 at coordinates 1.106791, 47.62817
Successfully updated point 1056
Processing missing data for point 1081 at coordinates 4.05244, 45.73408
Successfully updated point 1081
Processing missing data for point 1110 at coordinates 4.31513, 45.188658
Successfully updated point 1110
Processing missing data for point 1244 at coordinates 5.967521, 48.996453
Successfully updated point 1244
Processing missing data for point 1246 at coordinates 5.634411, 45.251803
Successfully updated point 1246
Processing missing data for point 1269 at coordinates 1.48663, 46.78984
Successfully updated point 1269
Processing missing data for point 1280 at coordinates 1.17236, 46.6196
```

```
Successfully updated point 1280
Processing missing data for point 1272 at coordinates 1.35562, 47.10523
Successfully updated point 1272
Processing missing data for point 1333 at coordinates -0.95724, 43.623243
Successfully updated point 1333
Processing missing data for point 1351 at coordinates -0.239878, 43.806771
Successfully updated point 1351
Processing missing data for point 1361 at coordinates 2.819498, 44.916088
Successfully updated point 1361
Processing missing data for point 1399 at coordinates 4.054361, 48.730593
Successfully updated point 1399
Processing missing data for point 1396 at coordinates 4.300754, 48.959924
Successfully updated point 1396
Processing missing data for point 1418 at coordinates -0.757008, 48.058626
Successfully updated point 1418
Processing missing data for point 1465 at coordinates -1.305934, 48.566394
Successfully updated point 1465
Processing missing data for point 1493 at coordinates 1.564248, 48.749983
Successfully updated point 1493
Processing missing data for point 1515 at coordinates 0.879868, 48.297504
Successfully updated point 1515
Processing missing data for point 1513 at coordinates 1.966878, 48.291714
Successfully updated point 1513
Processing missing data for point 1507 at coordinates 1.600771, 48.047341
Successfully updated point 1507
Processing missing data for point 1546 at coordinates 5.05454, 44.32302
Successfully updated point 1546
Processing missing data for point 1549 at coordinates 5.13098, 44.30854
Successfully updated point 1549
Processing missing data for point 1575 at coordinates 6.666943, 47.128374
Successfully updated point 1575
Processing missing data for point 1576 at coordinates 6.559288, 47.161011
Successfully updated point 1576
Processing missing data for point 1645 at coordinates -3.255429, 48.239248
Successfully updated point 1645
Processing missing data for point 1648 at coordinates -2.951216, 48.439631
Successfully updated point 1648
Processing missing data for point 1751 at coordinates -0.345346, 46.0298
Successfully updated point 1751
Processing missing data for point 1773 at coordinates -0.514265, 45.559151
Successfully updated point 1773
Processing missing data for point 1851 at coordinates 0.21301, 45.683683
Successfully updated point 1851
```

```
Processing missing data for point 1829 at coordinates 0.554783, 45.950545
Successfully updated point 1829
Processing missing data for point 1898 at coordinates -0.246134, 49.141341
Successfully updated point 1898
Processing missing data for point 1902 at coordinates 0.091428, 49.117383
Successfully updated point 1902
Processing missing data for point 1930 at coordinates 4.792664, 43.894531
Successfully updated point 1930
Processing missing data for point 1938 at coordinates 4.664076, 43.399791
Successfully updated point 1938
Processing missing data for point 1944 at coordinates 4.764023, 43.386851
Successfully updated point 1944
Processing missing data for point 1957 at coordinates 3.11161, 44.19392
Successfully updated point 1957
Processing missing data for point 2079 at coordinates 1.59067, 49.07755
Successfully updated point 2079
Processing missing data for point 2083 at coordinates 4.7675, 49.721748
Successfully updated point 2083
Processing missing data for point 2134 at coordinates 7.164566, 43.716186
Successfully updated point 2134
Processing missing data for point 2204 at coordinates 3.056281, 46.46886
Successfully updated point 2204
Processing missing data for point 2239 at coordinates 3.509173, 49.764728
Successfully updated point 2239
Processing missing data for point 2241 at coordinates 3.408321, 49.686713
Successfully updated point 2241
Processing missing data for point 2274 at coordinates 5.658703, 46.280918
Successfully updated point 2274
Processing missing data for point 2287 at coordinates 5.371121, 45.962301
Successfully updated point 2287
Processing missing data for point 2284 at coordinates 5.670066, 45.776498
Successfully updated point 2284
Processing missing data for point 2310 at coordinates -4.464553, 48.524331
Successfully updated point 2310
Processing missing data for point 2339 at coordinates -3.909861, 48.322179
Successfully updated point 2339
Processing missing data for point 2362 at coordinates 5.971293, 48.652804
Successfully updated point 2362
Processing missing data for point 2366 at coordinates 1.69355, 48.95858
Successfully updated point 2366
Processing missing data for point 2415 at coordinates -3.147249, 47.723656
Successfully updated point 2415
Processing missing data for point 2414 at coordinates -3.286465, 47.943738
```

```
Successfully updated point 2414
Processing missing data for point 2526 at coordinates 3.153084, 50.518051
Successfully updated point 2526
Processing missing data for point 2532 at coordinates 3.137133, 50.609016
Successfully updated point 2532
Processing missing data for point 2522 at coordinates 3.191674, 50.233321
Successfully updated point 2522
Processing missing data for point 2548 at coordinates 2.643136, 49.224024
Successfully updated point 2548
Processing missing data for point 2560 at coordinates 2.155684, 49.335503
Successfully updated point 2560
Processing missing data for point 2607 at coordinates 0.703044, 48.518116
Successfully updated point 2607
Processing missing data for point 2661 at coordinates 1.688209, 50.603511
Successfully updated point 2661
Processing missing data for point 2685 at coordinates 3.534318, 45.758041
Successfully updated point 2685
Processing missing data for point 2667 at coordinates 3.862523, 45.361973
Successfully updated point 2667
Processing missing data for point 2671 at coordinates 3.47058, 45.428731
Successfully updated point 2671
Processing missing data for point 2689 at coordinates 3.689698, 45.532144
Successfully updated point 2689
Processing missing data for point 2731 at coordinates 2.16721, 44.30777
Successfully updated point 2731
Processing missing data for point 2737 at coordinates 0.099172, 43.294687
Successfully updated point 2737
Processing missing data for point 2778 at coordinates 7.667896, 48.900893
Successfully updated point 2778
Processing missing data for point 2791 at coordinates 7.443721, 47.962394
Successfully updated point 2791
Processing missing data for point 2799 at coordinates 4.578846, 45.617959
Successfully updated point 2799
Saving updated data to data/soilgrids_updated.csv...
```