```
import streamlit as st
import os
import pandas as pd
import numpy as np
from scipy.signal import savgol_filter
import tsfel
from sklearn.preprocessing import MinMaxScaler, StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, classification_report, mean_squared_error,
r2_score
from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier,
AdaBoostClassifier, RandomForestRegressor, GradientBoostingRegressor
from sklearn.tree import DecisionTreeClassifier, DecisionTreeRegressor
from sklearn.model_selection import RandomizedSearchCV, GridSearchCV
from xgboost import XGBClassifier, XGBRegressor
from sklearn.svm import SVC, SVR
import plotly.graph_objects as go
import re
import streamlit.components.v1 as components
import plotly.express as px


def display_classification_report(accuracy, report, model_name=None):
    model_info = f"<h2 style='color: #000000; text-align: center; font-family: Arial,
sans-serif;'>{model_name}</h2>" if model_name else ""
    report_html = f"""
    <div style="border: 1px solid #E0E0E0; border-radius: 10px; padding: 20px;
background-color: #FAFAFA; box-shadow: 0 4px 8px 0 rgba(0, 0, 0, 0.2); font-family: Arial,
sans-serif;">
        {model_info}
        <h3 style='color: #000000; text-align: center;'>Accuracy: {accuracy:.2f}</h3>
        <h4 style='color: ##000000;'>Report:</h4>
        <pre style="font-size: small; background-color: #FFFFFF; padding: 15px; border-radius:
5px; overflow: auto; white-space: pre-wrap; border: 1px solid #E0E0E0;">{report}</pre>
    </div>
    """
    components.html(report_html, height=430, scrolling=True)



def display_regression_metrics(mse, r2, model_name=None):
    model_info = f"<h2 style='color: #000000; text-align: center; font-family: Arial,
sans-serif;'>{model_name}</h2>" if model_name else ""
    metrics_html = f"""
```

```
    <div style="border: 1px solid #E0E0E0; border-radius: 10px; padding: 20px;
background-color: #FAFAFA; box-shadow: 0 4px 8px 0 rgba(0, 0, 0, 0.2); font-family: Arial,
sans-serif;">
        {model_info}
        <h3 style='color: #000000; text-align: center;'>MSE: {mse:.2f}</h3>
        <h3 style='color: #000000; text-align: center;'>R2 Score: {r2:.2f}</h3>
    </div>
    """
    components.html(metrics_html, height=250, scrolling=True)



def load_data_from_folder(folder_path):
    data = []
    for subdir, _, files in os.walk(folder_path):
        for file in files:
            if file.endswith('.csv'):
                file_path = os.path.join(subdir, file)
                df = pd.read_csv(file_path, header=None)
                df['filename'] = file
                data.append(df)
    data = pd.DataFrame(data)
    return data



def count_xls_files(folder_path):
    xls_count = 0
    for root, dirs, files in os.walk(folder_path):
        for file in files:
            if file.endswith('.csv'):
                xls_count += 1
                file_path = os.path.join(root, file)
    return xls_count



def df_maker_smooth(path, window_size, poly_order, target_expression):
    all_features_df = pd.DataFrame()
    for subfolder in os.listdir(path):
        subfolder_path = os.path.join(path, subfolder)
        if os.path.isdir(subfolder_path):
            for file in os.listdir(subfolder_path):
                if file.endswith(".csv"):
```

```
                file_path = os.path.join(subfolder_path, file)
                df = pd.read_csv(file_path)
                df['filename'] = file
                signal = df.iloc[:, 0].values
                smoothened_signal = golay_filter(signal, window_size, poly_order)
                smoothened_signal_df = pd.DataFrame(smoothened_signal)
                df['target'] = df['filename'].str.extract(f'({target_expression})(\d+)')[1].astype(float)
                cfg = tsfel.get_features_by_domain()
                features = tsfel.time_series_features_extractor(cfg, smoothened_signal_df.iloc[:, 0],
1000000)
                features_transposed = features.T.reset_index(drop=True).T
                features_transposed['target'] = df['target']
                all_features_df = pd.concat([all_features_df, features_transposed],
ignore_index=True)
    return all_features_df



def df_maker_unsmooth(path, target_expression):
    all_features_df = pd.DataFrame()
    for subfolder in os.listdir(path):
        subfolder_path = os.path.join(path, subfolder)
        if os.path.isdir(subfolder_path):
            for file in os.listdir(subfolder_path):
                if file.endswith(".csv"):
                    file_path = os.path.join(subfolder_path, file)
                    df = pd.read_csv(file_path)
                    df['filename'] = file
                    signal = df.iloc[:, 0].values
                    df['target'] = df['filename'].str.extract(f'({target_expression})(\d+)')[1].astype(float)
                    cfg = tsfel.get_features_by_domain()
                    features = tsfel.time_series_features_extractor(cfg, df.iloc[:, 0], 1000000)
                    features_transposed = features.T.reset_index(drop=True).T
                    features_transposed['target'] = df['target']
                    all_features_df = pd.concat([all_features_df, features_transposed],
ignore_index=True)
    return all_features_df



def reduce_features(features, threshold):
    corr_matrix = features.corr().abs()
    upper = corr_matrix.where(np.triu(np.ones(corr_matrix.shape), k=1).astype(bool))
    to_drop = [column for column in upper.columns if any(upper[column] > threshold)]
```

```
    reduced_features = features.drop(columns=to_drop)
    return reduced_features



def golay_filter(signal, window_size, poly_order):
    return savgol_filter(signal, window_length=window_size, polyorder=poly_order)



def scale_data(data, scaler_type):
    if scaler_type == "MinMax":
        scaler = MinMaxScaler()
    else:
        scaler = StandardScaler()
    scaled_data = pd.DataFrame(scaler.fit_transform(data.drop(columns=['target'])),
columns=data.columns[:-1])
    scaled_data['target'] = data['target']
    return scaled_data, scaler



def split(scaled_data, test_size):
    X = scaled_data.drop(columns=['target'])
    y = scaled_data['target']
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=test_size, random_state=42)
    return X_train, X_test, y_train, y_test



def models(learning_type, selected_model, X_train, X_test, y_train, y_test):
    if learning_type == "Classification":
        models = {
            "Random Forest Classifier": RandomForestClassifier(),
            "Gradient Boosting Classifier": GradientBoostingClassifier(),
            "AdaBoost Classifier": AdaBoostClassifier(),
            "Decision Tree Classifier": DecisionTreeClassifier(),
            "SVC": SVC()
        }
        if selected_model != "All Models":
            model = models[selected_model]
            model.fit(X_train, y_train)
            y_pred = model.predict(X_test)
            accuracy = accuracy_score(y_test, y_pred)
```

```
            report = classification_report(y_test, y_pred)
            return accuracy, report, model
        else:
            results = {}
            reports = {}
            trained_models = {}
            for name, model in models.items():
                model.fit(X_train, y_train)
                y_pred = model.predict(X_test)
                accuracy = accuracy_score(y_test, y_pred)
                report = classification_report(y_test, y_pred)
                results[name] = accuracy
                reports[name] = report
                trained_models[name] = model
            return results, reports, trained_models
    else:
        models = {
            "Random Forest Regressor": RandomForestRegressor(),
            "Gradient Boosting Regressor": GradientBoostingRegressor(),
            "Decision Tree Regressor": DecisionTreeRegressor(),
            "XGBoost Regressor": XGBRegressor(),
            "SVR": SVR()
        }
        if selected_model != "All Models":
            model = models[selected_model]
            model.fit(X_train, y_train)
            y_pred = model.predict(X_test)
            mse = mean_squared_error(y_test, y_pred)
            r2 = r2_score(y_test, y_pred)
            return mse, r2, model
        else:
            results = {}
            reports = {}
            trained_models = {}
            for name, model in models.items():
                model.fit(X_train, y_train)
                y_pred = model.predict(X_test)
                mse = mean_squared_error(y_test, y_pred)
                r2 = r2_score(y_test, y_pred)
                results[name] = (mse, r2)
                trained_models[name] = model
            return results, reports, trained_models
```

```
def plot_classification_accuracy(results):
    accuracy_df = pd.DataFrame(columns=['Accuracy'])
    for model_name, accuracy in results.items():
        accuracy_df.loc[model_name] = [accuracy]

    st.write("Model Performance Comparison (Classification):")
    st.write(accuracy_df)

    fig = go.Figure()
    fig.add_trace(go.Bar(
        x=accuracy_df.index,
        y=accuracy_df['Accuracy'],
        name='Accuracy',
        marker_color='green'
    ))
    fig.update_layout(
        xaxis_tickangle=-45,
        title='Model Performance Comparison (Classification)',
        xaxis_title='Model',
        yaxis_title='Accuracy'
    )
    st.plotly_chart(fig)



def plot_regression_metrics(results):
    mse_r2_df = pd.DataFrame(columns=['MSE', 'R2 Score'])
    for model_name, scores in results.items():
        mse, r2 = scores
        mse_r2_df.loc[model_name] = [mse, r2]

    st.write("Model Performance Comparison (Regression):")
    st.write(mse_r2_df)

    fig = go.Figure()
    fig.add_trace(go.Bar(
        x=mse_r2_df.index,
        y=mse_r2_df['MSE'],
        name='MSE',
        marker_color='blue'
    ))
    fig.add_trace(go.Bar(
        x=mse_r2_df.index,
```

```
        y=mse_r2_df['R2 Score'],
        name='R2 Score',
        marker_color='orange'
    ))
    fig.update_layout(
        barmode='group',
        xaxis_tickangle=-45,
        title='Model Performance Comparison (Regression)',
        xaxis_title='Model',
        yaxis_title='Value'
    )
    st.plotly_chart(fig)



def display_feature_importance(model, X_train):
    if hasattr(model, 'feature_importances_'):
        importances = model.feature_importances_
        feature_names = X_train.columns
        feature_importance_df = pd.DataFrame({
            'Feature': feature_names,
            'Importance': importances
        }).sort_values(by='Importance', ascending=False)

        st.session_state.feature_importance_df = feature_importance_df

        fig = go.Figure([go.Bar(x=feature_importance_df['Feature'],
y=feature_importance_df['Importance'])])
        fig.update_layout(title='Feature Importance', xaxis_title='Feature', yaxis_title='Importance')

        st.plotly_chart(fig)
    else:
        st.warning("Selected model does not support feature importance.")

    return feature_importance_df

def feature_name(model, X_train, top_n_features):
    if hasattr(model, 'feature_importances_'):
        importances = model.feature_importances_
        feature_names = X_train.columns
        top_features_df = pd.DataFrame({
            'Feature': feature_names,
            'Importance': importances
        }).sort_values(by='Importance', ascending=False)
```

```
        top_features_df = top_features_df['Feature'].head(top_n_features)

    return top_features_df

def hyperparameter_tuning(model, X_train, X_test, y_train, y_test, method='Randomized
Search'):
    # Define parameter grids for different models
    param_grids = {
        RandomForestClassifier: {
            'n_estimators': [50, 100, 200, 300],
            'max_depth': [3, 5, 10, None],
            'min_samples_split': [2, 5, 10],
            'min_samples_leaf': [1, 2, 4],
            'bootstrap': [True, False]
        },
        RandomForestRegressor: {
            'n_estimators': [50, 100, 200, 300],
            'max_depth': [3, 5, 10, None],
            'min_samples_split': [2, 5, 10],
            'min_samples_leaf': [1, 2, 4],
            'bootstrap': [True, False]
        },
        GradientBoostingClassifier: {
            'n_estimators': [50, 100, 200],
            'learning_rate': [0.01, 0.1, 0.05],
            'max_depth': [3, 4, 5],
            'subsample': [0.7, 0.8, 0.9, 1.0]
        },
        GradientBoostingRegressor: {
            'n_estimators': [50, 100, 200],
            'learning_rate': [0.01, 0.1, 0.05],
            'max_depth': [3, 4, 5],
            'subsample': [0.7, 0.8, 0.9, 1.0]
        },
        XGBRegressor: {
            'n_estimators': [50, 100, 200],
            'learning_rate': [0.01, 0.1, 0.05],
            'max_depth': [3, 4, 5],
            'subsample': [0.7, 0.8, 0.9, 1.0]
        },
        DecisionTreeClassifier: {
            'max_depth': [None, 10, 20, 30],
            'min_samples_split': [2, 5, 10],
            'min_samples_leaf': [1, 2, 4]
```

```
        },
        DecisionTreeRegressor: {
        'max_depth': [3, 5, 10, None],
        'min_samples_split': [2, 5, 10],
        'min_samples_leaf': [1, 2, 4]
        },
        SVR: {
            'kernel': ['linear', 'poly', 'rbf', 'sigmoid'],
            'C': [0.1, 1, 10, 100],
            'gamma': ['scale', 'auto']
        },
        SVC: {
            'C': [0.1, 1, 10],
            'kernel': ['linear', 'rbf', 'poly'],
            'degree': [2, 3, 4]
        },
        AdaBoostClassifier: {
            'n_estimators': [50, 100, 200],
            'learning_rate': [0.01, 0.1, 0.05],
            'algorithm': ['SAMME', 'SAMME.R']
        }
    }

    # Get the parameter grid for the model
    param_grid = param_grids[type(model)]

    if method == 'Randomized Search':
        tuner = RandomizedSearchCV(estimator=model, param_distributions=param_grid,
n_iter=100, cv=3, verbose=2, random_state=42, n_jobs=-1)
    elif method == 'Grid Search':
        tuner = GridSearchCV(estimator=model, param_grid=param_grid, cv=3, verbose=2,
n_jobs=-1)
    else:
        raise ValueError("Invalid method. Choose either 'Randomized Search' or 'Grid Search'.")

    tuner.fit(X_train, y_train)
    best_model = tuner.best_estimator_

    if isinstance(best_model, (RandomForestRegressor, GradientBoostingRegressor,
XGBRegressor, DecisionTreeRegressor, SVR)):
        y_pred = best_model.predict(X_test)
        score1 = mean_squared_error(y_test, y_pred)
        score2 = r2_score(y_test, y_pred)
```

```
    elif isinstance(best_model, (RandomForestClassifier, GradientBoostingClassifier,
DecisionTreeClassifier, AdaBoostClassifier, SVC)):
        y_pred = best_model.predict(X_test)
        score1 = accuracy_score(y_test, y_pred)
        score2 = classification_report(y_test, y_pred)
    else:
        raise TypeError("Unsupported model type for evaluation.")

    return best_model, score1, score2, y_pred

def barplot(y_test,y_pred):
    #Plot
    fig=go.Figure()
    fig.add_trace(go.Bar(y=y_test,name='Actual',marker_color='blue'))
    fig.add_trace(go.Bar(y=y_pred,name='Predicted',marker_color='orange'))
    fig.update_layout(title=f'Actual vs Predicted',
                      xaxis_title='index',
                      yaxis_title='Target')
    st.plotly_chart(fig)

def main():
    st.title("CT-PIU Gap Measurement")

    if "score1" not in st.session_state:
        st.session_state.score1 = None
    if "score2" not in st.session_state:
        st.session_state.score2 = None
    if "model" not in st.session_state:
        st.session_state.model = None
    if "result" not in st.session_state:
        st.session_state.result = None
    if "reports" not in st.session_state:
        st.session_state.reports = None
    if "xls_count" not in st.session_state:
        st.session_state.xls_count = None
    if "target_expression" not in st.session_state:
        st.session_state.target_expression = None
    if "data_loaded" not in st.session_state:
        st.session_state.data_loaded = False

    folder_path = st.text_input("Enter the path to the main folder")

    if folder_path:
        if st.button("Count .xls Files and Calculate Max Values"):
```

```
            if os.path.isdir(folder_path):
                xls_count = count_xls_files(folder_path)
                st.session_state.xls_count = xls_count
                st.success(f"Total number of .xls files: {xls_count}")

    if st.session_state.xls_count is not None:
        target_expression = st.text_input("Enter the target expression (e.g., __OD__)", "_OD_")
        if st.button("Load Data"):
            st.session_state.target_expression = target_expression
            st.session_state.data_loaded = True

    if st.session_state.data_loaded:
        remove_noise = st.sidebar.radio("Remove Noise and Feature Extraction", ('Yes', 'No'))
        if remove_noise == "Yes":
            window_size = st.sidebar.slider("Window Size", 3, 51, step=2, value=5)
            poly_order = st.sidebar.slider("Polynomial Order", 1, 5, value=2)
            if "smooth_df" not in st.session_state:
                with st.spinner("Processing..."):
                    st.session_state.smooth_df = df_maker_smooth(folder_path, window_size,
poly_order, st.session_state.target_expression)
            df = st.session_state.smooth_df
            st.write("Noise removed using Savitzky-Golay filter.")
            st.write(df)
            st.write("Shape", df.shape)
        else:
            if "unsmooth_df" not in st.session_state:
                with st.spinner("Processing..."):
                    st.session_state.unsmooth_df = df_maker_unsmooth(folder_path,
st.session_state.target_expression)
            df = st.session_state.unsmooth_df
            st.write("Using raw data.")
            st.write(df)
            st.write("Shape", df.shape)

        threshold = st.sidebar.slider("Correlation Threshold", 0.0, 1.0, 0.9)
        if st.sidebar.button("Correlation"):
            st.session_state.reduced_features = reduce_features(df, threshold)
            reduced_features = st.session_state.reduced_features
            st.write("Reduced features based on correlation.", reduced_features)
            st.write("Shape", reduced_features.shape)
        else:
            if "reduced_features" in st.session_state:
                reduced_features = st.session_state.reduced_features
            else:
```

```
            reduced_features = df

        scale_data_flag = st.sidebar.checkbox("Scale Data", value=False)
        if scale_data_flag:
            st.write(st.session_state.reduced_features)
            scaler_type = st.sidebar.selectbox("Scaler Type", ["Standard", "MinMax"])
            scaled_data, scaler = scale_data(reduced_features, scaler_type)
            st.write("Data scaled using", scaler_type, "scaler.", scaled_data)
        else:
            scaled_data = reduced_features

        learning_type = st.sidebar.selectbox("Learning Type", ["Classification", "Regression"])
        test_size = st.sidebar.slider("Test Size Split Ratio", 0.1, 0.5, 0.2)

        if st.sidebar.button("Split"):
            X_train, X_test, y_train, y_test = split(scaled_data, test_size)
            st.session_state.X_train = X_train
            st.session_state.X_test = X_test
            st.session_state.y_train = y_train
            st.session_state.y_test = y_test
            st.write(st.session_state.X_train.shape)

        if learning_type == "Classification":
            models_list = [
                "Random Forest Classifier",
                "Gradient Boosting Classifier",
                "AdaBoost Classifier",
                "Decision Tree Classifier",
                "SVC"
            ]
        else:
            models_list = [
                "Random Forest Regressor",
                "Gradient Boosting Regressor",
                "Decision Tree Regressor",
                "XGBoost Regressor",
                "SVR"
            ]

        selected_model = st.sidebar.selectbox("Select Model for feature importance:", models_list
+ ["All Models"])

        if st.sidebar.button("Train"):
```

```
        if "X_train" in st.session_state and "X_test" in st.session_state and "y_train" in
st.session_state and "y_test" in st.session_state:
            if selected_model == "All Models" and learning_type == "Classification":
                st.session_state.result, st.session_state.reports, trained_models = models(
                    learning_type, selected_model, st.session_state.X_train, st.session_state.X_test,
                    st.session_state.y_train, st.session_state.y_test)
                plot_classification_accuracy(st.session_state.result)
                for model_name, report in st.session_state.reports.items():
                    display_classification_report(st.session_state.result[model_name], report,
model_name)
            elif selected_model == "All Models" and learning_type == "Regression":
                st.session_state.result, _, trained_models = models(
                    learning_type, selected_model, st.session_state.X_train, st.session_state.X_test,
                    st.session_state.y_train, st.session_state.y_test)
                plot_regression_metrics(st.session_state.result)
            else:
                st.session_state.score1, st.session_state.score2, st.session_state.model = models(
                    learning_type, selected_model, st.session_state.X_train, st.session_state.X_test,
                    st.session_state.y_train, st.session_state.y_test)
                if learning_type == "Regression":
                    display_regression_metrics(st.session_state.score1, st.session_state.score2)
                else:
                    display_classification_report(st.session_state.score1, st.session_state.score2)
        else:
            st.error("Please split the data before training the model.")

    if selected_model != "All Models" and st.sidebar.button("Show Feature Importance"):
        if "model" in st.session_state:
            st.session_state.feature_importance_df=
display_feature_importance(st.session_state.model, st.session_state.X_train)
            st.session_state.show_feature_selection = True
        else:
            st.error("Train a single model first to see feature importance.")

    # After showing feature importance, allow user to input number of top features
    if st.session_state.get('show_feature_selection', False):
        if "model" in st.session_state:
            num_top_features = st.sidebar.number_input("Enter the number of top features to
select:", min_value=1, max_value=len(st.session_state.feature_importance_df), value=10)
            top_features =
st.session_state.feature_importance_df['Feature'].head(num_top_features).tolist()

#             st.session_state.top_feature_names = feature_name(model, X_train, top_features)
        if learning_type == "Classification":
```

```
            models_list = [
                "Random Forest Classifier",
                "Gradient Boosting Classifier",
                "AdaBoost Classifier",
                "Decision Tree Classifier",
                "SVC"
            ]
        else:
            models_list = [
                "Random Forest Regressor",
                "Gradient Boosting Regressor",
                "Decision Tree Regressor",
                "XGBoost Regressor",
                "SVR"
            ]

        selected_model_imp = st.sidebar.selectbox("Select Model after feature importance:",
models_list)
        if st.sidebar.button("Train on Top N Features"):
            if "X_train" in st.session_state and "X_test" in st.session_state and "y_train" in
st.session_state and "y_test" in st.session_state:
                # Filter X_train and X_test to keep only top features
                st.session_state.X_train_top = st.session_state.X_train[top_features]
                st.session_state.X_test_top = st.session_state.X_test[top_features]

                if learning_type == "Regression":
                    st.session_state.score1, st.session_state.score2, st.session_state.model =
models(
                    learning_type, selected_model_imp, st.session_state.X_train_top,
st.session_state.X_test_top,
                    st.session_state.y_train, st.session_state.y_test)
                    display_regression_metrics(st.session_state.score1, st.session_state.score2)
                else:
                    st.session_state.score1, st.session_state.score2, st.session_state.model =
models(
                    learning_type, selected_model_imp, st.session_state.X_train_top,
st.session_state.X_test_top,
                    st.session_state.y_train, st.session_state.y_test)
                    display_classification_report(st.session_state.score1, st.session_state.score2)
            else:
                st.error("Please split the data before training the model.")
        # else:
        #     st.error("Train a single model first to see feature importance.")
```

```
        if st.sidebar.checkbox("Hyperparamter Tuning", value=False):
            method = st.sidebar.selectbox("Hyperparameter Tuning Method", ["Randomized
Search", "Grid Search"])
            if st.sidebar.button("Train After Hyperparameter Tuning"):
                with st.spinner("Performing Hyperparameter Tuning..."):
                    # method = st.sidebar.selectbox("Hyperparameter Tuning Method",
["Randomized Search", "Grid Search"])
                    # if method == "Randomized Search":
                    #     # Define hyperparameter grid for randomized search
                    #     param_grid = {
                    #         'n_estimators': [50, 100, 200, 300],
                    #         'max_depth': [3, 5, 10, None],
                    #         'min_samples_split': [2, 5, 10],
                    #         'min_samples_leaf': [1, 2, 4],
                    #         'bootstrap': [True, False]
                    #     }
                    # else:
                    #     # Define hyperparameter grid for grid search
                    #     param_grid = {
                    #         'n_estimators': [50, 100, 200, 300],
                    #         'max_depth': [3, 5, 10, None],
                    #         'min_samples_split': [2, 5, 10],
                    #         'min_samples_leaf': [1, 2, 4],
                    #         'bootstrap': [True, False]
                    #     }
                    st.session_state.model, st.session_state.score1, st.session_state.score2,
st.session_state.y_pred = hyperparameter_tuning(st.session_state.model,
st.session_state.X_train[top_features], st.session_state.X_test[top_features],
st.session_state.y_train, st.session_state.y_test, method)
                    if learning_type == "Regression":
                        display_regression_metrics(st.session_state.score1, st.session_state.score2)
                        barplot(st.session_state.y_test, st.session_state.y_pred)
                    else:
                        display_classification_report(st.session_state.score1, st.session_state.score2)
                        barplot(st.session_state.y_test, st.session_state.y_pred)
            else:
                st.warning("Perform Hyperparameter Tuning and click Train After Hyperparameter
Tuning.")

if __name__ == "__main__":
    main()
```

```
import streamlit as st
import os
import pandas as pd
import numpy as np
from scipy.signal import savgol_filter
import tsfel
from sklearn.preprocessing import MinMaxScaler, StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, classification_report, mean_squared_error,
r2_score
from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier,
AdaBoostClassifier, RandomForestRegressor, GradientBoostingRegressor
from sklearn.tree import DecisionTreeClassifier, DecisionTreeRegressor
from sklearn.model_selection import RandomizedSearchCV, GridSearchCV
from xgboost import XGBClassifier, XGBRegressor
from sklearn.svm import SVC, SVR
import plotly.graph_objects as go
import re
import streamlit.components.v1 as components
import plotly.express as px


def display_classification_report(accuracy, report, model_name=None):
    model_info = f"<h2 style='color: #000000; text-align: center; font-family: Arial,
sans-serif;'>{model_name}</h2>" if model_name else ""
    report_html = f"""
    <div style="border: 1px solid #E0E0E0; border-radius: 10px; padding: 20px;
background-color: #FAFAFA; box-shadow: 0 4px 8px 0 rgba(0, 0, 0, 0.2); font-family: Arial,
sans-serif;">
        {model_info}
        <h3 style='color: #000000; text-align: center;'>Accuracy: {accuracy:.2f}</h3>
        <h4 style='color: ##000000;'>Report:</h4>
        <pre style="font-size: small; background-color: #FFFFFF; padding: 15px; border-radius:
5px; overflow: auto; white-space: pre-wrap; border: 1px solid #E0E0E0;">{report}</pre>
    </div>
    """
    components.html(report_html, height=430, scrolling=True)



def display_regression_metrics(mse, r2, model_name=None):
    model_info = f"<h2 style='color: #000000; text-align: center; font-family: Arial,
sans-serif;'>{model_name}</h2>" if model_name else ""
    metrics_html = f"""
```

```
    <div style="border: 1px solid #E0E0E0; border-radius: 10px; padding: 20px;
background-color: #FAFAFA; box-shadow: 0 4px 8px 0 rgba(0, 0, 0, 0.2); font-family: Arial,
sans-serif;">
        {model_info}
        <h3 style='color: #000000; text-align: center;'>MSE: {mse:.2f}</h3>
        <h3 style='color: #000000; text-align: center;'>R2 Score: {r2:.2f}</h3>
    </div>
    """
    components.html(metrics_html, height=250, scrolling=True)



def load_data_from_folder(folder_path):
    data = []
    for subdir, _, files in os.walk(folder_path):
        for file in files:
            if file.endswith('.csv'):
                file_path = os.path.join(subdir, file)
                df = pd.read_csv(file_path, header=None)
                df['filename'] = file
                data.append(df)
    data = pd.DataFrame(data)
    return data



def count_xls_files(folder_path):
    xls_count = 0
    for root, dirs, files in os.walk(folder_path):
        for file in files:
            if file.endswith('.csv'):
                xls_count += 1
                file_path = os.path.join(root, file)
    return xls_count



def df_maker_smooth(path, window_size, poly_order, target_expression):
    all_features_df = pd.DataFrame()
    for subfolder in os.listdir(path):
        subfolder_path = os.path.join(path, subfolder)
        if os.path.isdir(subfolder_path):
            for file in os.listdir(subfolder_path):
                if file.endswith(".csv"):
```

```
                file_path = os.path.join(subfolder_path, file)
                df = pd.read_csv(file_path)
                df['filename'] = file
                signal = df.iloc[:, 0].values
                smoothened_signal = golay_filter(signal, window_size, poly_order)
                smoothened_signal_df = pd.DataFrame(smoothened_signal)
                df['target'] = df['filename'].str.extract(f'({target_expression})(\d+)')[1].astype(float)
                cfg = tsfel.get_features_by_domain()
                features = tsfel.time_series_features_extractor(cfg, smoothened_signal_df.iloc[:, 0],
1000000)
                features_transposed = features.T.reset_index(drop=True).T
                features_transposed['target'] = df['target']
                all_features_df = pd.concat([all_features_df, features_transposed],
ignore_index=True)
    return all_features_df



def df_maker_unsmooth(path, target_expression):
    all_features_df = pd.DataFrame()
    for subfolder in os.listdir(path):
        subfolder_path = os.path.join(path, subfolder)
        if os.path.isdir(subfolder_path):
            for file in os.listdir(subfolder_path):
                if file.endswith(".csv"):
                    file_path = os.path.join(subfolder_path, file)
                    df = pd.read_csv(file_path)
                    df['filename'] = file
                    signal = df.iloc[:, 0].values
                    df['target'] = df['filename'].str.extract(f'({target_expression})(\d+)')[1].astype(float)
                    cfg = tsfel.get_features_by_domain()
                    features = tsfel.time_series_features_extractor(cfg, df.iloc[:, 0], 1000000)
                    features_transposed = features.T.reset_index(drop=True).T
                    features_transposed['target'] = df['target']
                    all_features_df = pd.concat([all_features_df, features_transposed],
ignore_index=True)
    return all_features_df



def reduce_features(features, threshold):
    corr_matrix = features.corr().abs()
    upper = corr_matrix.where(np.triu(np.ones(corr_matrix.shape), k=1).astype(bool))
    to_drop = [column for column in upper.columns if any(upper[column] > threshold)]
```

```
    reduced_features = features.drop(columns=to_drop)
    return reduced_features



def golay_filter(signal, window_size, poly_order):
    return savgol_filter(signal, window_length=window_size, polyorder=poly_order)



def scale_data(data, scaler_type):
    if scaler_type == "MinMax":
        scaler = MinMaxScaler()
    else:
        scaler = StandardScaler()
    scaled_data = pd.DataFrame(scaler.fit_transform(data.drop(columns=['target'])),
columns=data.columns[:-1])
    scaled_data['target'] = data['target']
    return scaled_data, scaler



def split(scaled_data, test_size):
    X = scaled_data.drop(columns=['target'])
    y = scaled_data['target']
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=test_size, random_state=42)
    return X_train, X_test, y_train, y_test



def models(learning_type, selected_model, X_train, X_test, y_train, y_test):
    if learning_type == "Classification":
        models = {
            "Random Forest Classifier": RandomForestClassifier(),
            "Gradient Boosting Classifier": GradientBoostingClassifier(),
            "AdaBoost Classifier": AdaBoostClassifier(),
            "Decision Tree Classifier": DecisionTreeClassifier(),
            "SVC": SVC()
        }
        if selected_model != "All Models":
            model = models[selected_model]
            model.fit(X_train, y_train)
            y_pred = model.predict(X_test)
            accuracy = accuracy_score(y_test, y_pred)
```

```
            report = classification_report(y_test, y_pred)
            return accuracy, report, model
        else:
            results = {}
            reports = {}
            trained_models = {}
            for name, model in models.items():
                model.fit(X_train, y_train)
                y_pred = model.predict(X_test)
                accuracy = accuracy_score(y_test, y_pred)
                report = classification_report(y_test, y_pred)
                results[name] = accuracy
                reports[name] = report
                trained_models[name] = model
            return results, reports, trained_models
    else:
        models = {
            "Random Forest Regressor": RandomForestRegressor(),
            "Gradient Boosting Regressor": GradientBoostingRegressor(),
            "Decision Tree Regressor": DecisionTreeRegressor(),
            "XGBoost Regressor": XGBRegressor(),
            "SVR": SVR()
        }
        if selected_model != "All Models":
            model = models[selected_model]
            model.fit(X_train, y_train)
            y_pred = model.predict(X_test)
            mse = mean_squared_error(y_test, y_pred)
            r2 = r2_score(y_test, y_pred)
            return mse, r2, model
        else:
            results = {}
            reports = {}
            trained_models = {}
            for name, model in models.items():
                model.fit(X_train, y_train)
                y_pred = model.predict(X_test)
                mse = mean_squared_error(y_test, y_pred)
                r2 = r2_score(y_test, y_pred)
                results[name] = (mse, r2)
                trained_models[name] = model
            return results, reports, trained_models
```

```
def plot_classification_accuracy(results):
    accuracy_df = pd.DataFrame(columns=['Accuracy'])
    for model_name, accuracy in results.items():
        accuracy_df.loc[model_name] = [accuracy]

    st.write("Model Performance Comparison (Classification):")
    st.write(accuracy_df)

    fig = go.Figure()
    fig.add_trace(go.Bar(
        x=accuracy_df.index,
        y=accuracy_df['Accuracy'],
        name='Accuracy',
        marker_color='green'
    ))
    fig.update_layout(
        xaxis_tickangle=-45,
        title='Model Performance Comparison (Classification)',
        xaxis_title='Model',
        yaxis_title='Accuracy'
    )
    st.plotly_chart(fig)



def plot_regression_metrics(results):
    mse_r2_df = pd.DataFrame(columns=['MSE', 'R2 Score'])
    for model_name, scores in results.items():
        mse, r2 = scores
        mse_r2_df.loc[model_name] = [mse, r2]

    st.write("Model Performance Comparison (Regression):")
    st.write(mse_r2_df)

    fig = go.Figure()
    fig.add_trace(go.Bar(
        x=mse_r2_df.index,
        y=mse_r2_df['MSE'],
        name='MSE',
        marker_color='blue'
    ))
    fig.add_trace(go.Bar(
        x=mse_r2_df.index,
```

```
        y=mse_r2_df['R2 Score'],
        name='R2 Score',
        marker_color='orange'
    ))
    fig.update_layout(
        barmode='group',
        xaxis_tickangle=-45,
        title='Model Performance Comparison (Regression)',
        xaxis_title='Model',
        yaxis_title='Value'
    )
    st.plotly_chart(fig)



def display_feature_importance(model, X_train):
    if hasattr(model, 'feature_importances_'):
        importances = model.feature_importances_
        feature_names = X_train.columns
        feature_importance_df = pd.DataFrame({
            'Feature': feature_names,
            'Importance': importances
        }).sort_values(by='Importance', ascending=False)

        st.session_state.feature_importance_df = feature_importance_df

        fig = go.Figure([go.Bar(x=feature_importance_df['Feature'],
y=feature_importance_df['Importance'])])
        fig.update_layout(title='Feature Importance', xaxis_title='Feature', yaxis_title='Importance')

        st.plotly_chart(fig)
    else:
        st.warning("Selected model does not support feature importance.")

    return feature_importance_df

def feature_name(model, X_train, top_n_features):
    if hasattr(model, 'feature_importances_'):
        importances = model.feature_importances_
        feature_names = X_train.columns
        top_features_df = pd.DataFrame({
            'Feature': feature_names,
            'Importance': importances
        }).sort_values(by='Importance', ascending=False)
```

```
        top_features_df = top_features_df['Feature'].head(top_n_features)

    return top_features_df

def hyperparameter_tuning(model, X_train, X_test, y_train, y_test, method='Randomized
Search'):
    # Define parameter grids for different models
    param_grids = {
        RandomForestClassifier: {
            'n_estimators': [50, 100, 200, 300],
            'max_depth': [3, 5, 10, None],
            'min_samples_split': [2, 5, 10],
            'min_samples_leaf': [1, 2, 4],
            'bootstrap': [True, False]
        },
        RandomForestRegressor: {
            'n_estimators': [50, 100, 200, 300],
            'max_depth': [3, 5, 10, None],
            'min_samples_split': [2, 5, 10],
            'min_samples_leaf': [1, 2, 4],
            'bootstrap': [True, False]
        },
        GradientBoostingClassifier: {
            'n_estimators': [50, 100, 200],
            'learning_rate': [0.01, 0.1, 0.05],
            'max_depth': [3, 4, 5],
            'subsample': [0.7, 0.8, 0.9, 1.0]
        },
        GradientBoostingRegressor: {
            'n_estimators': [50, 100, 200],
            'learning_rate': [0.01, 0.1, 0.05],
            'max_depth': [3, 4, 5],
            'subsample': [0.7, 0.8, 0.9, 1.0]
        },
        XGBRegressor: {
            'n_estimators': [50, 100, 200],
            'learning_rate': [0.01, 0.1, 0.05],
            'max_depth': [3, 4, 5],
            'subsample': [0.7, 0.8, 0.9, 1.0]
        },
        DecisionTreeClassifier: {
            'max_depth': [None, 10, 20, 30],
            'min_samples_split': [2, 5, 10],
            'min_samples_leaf': [1, 2, 4]
```

```
        },
        DecisionTreeRegressor: {
        'max_depth': [3, 5, 10, None],
        'min_samples_split': [2, 5, 10],
        'min_samples_leaf': [1, 2, 4]
        },
        SVR: {
            'kernel': ['linear', 'poly', 'rbf', 'sigmoid'],
            'C': [0.1, 1, 10, 100],
            'gamma': ['scale', 'auto']
        },
        SVC: {
            'C': [0.1, 1, 10],
            'kernel': ['linear', 'rbf', 'poly'],
            'degree': [2, 3, 4]
        },
        AdaBoostClassifier: {
            'n_estimators': [50, 100, 200],
            'learning_rate': [0.01, 0.1, 0.05],
            'algorithm': ['SAMME', 'SAMME.R']
        }
    }

    # Get the parameter grid for the model
    param_grid = param_grids[type(model)]

    if method == 'Randomized Search':
        tuner = RandomizedSearchCV(estimator=model, param_distributions=param_grid,
n_iter=100, cv=3, verbose=2, random_state=42, n_jobs=-1)
    elif method == 'Grid Search':
        tuner = GridSearchCV(estimator=model, param_grid=param_grid, cv=3, verbose=2,
n_jobs=-1)
    else:
        raise ValueError("Invalid method. Choose either 'Randomized Search' or 'Grid Search'.")

    tuner.fit(X_train, y_train)
    best_model = tuner.best_estimator_

    if isinstance(best_model, (RandomForestRegressor, GradientBoostingRegressor,
XGBRegressor, DecisionTreeRegressor, SVR)):
        y_pred = best_model.predict(X_test)
        score1 = mean_squared_error(y_test, y_pred)
        score2 = r2_score(y_test, y_pred)
```

```
    elif isinstance(best_model, (RandomForestClassifier, GradientBoostingClassifier,
DecisionTreeClassifier, AdaBoostClassifier, SVC)):
        y_pred = best_model.predict(X_test)
        score1 = accuracy_score(y_test, y_pred)
        score2 = classification_report(y_test, y_pred)
    else:
        raise TypeError("Unsupported model type for evaluation.")

    return best_model, score1, score2, y_pred

def barplot(y_test,y_pred):
    #Plot
    fig=go.Figure()
    fig.add_trace(go.Bar(y=y_test,name='Actual',marker_color='blue'))
    fig.add_trace(go.Bar(y=y_pred,name='Predicted',marker_color='orange'))
    fig.update_layout(title=f'Actual vs Predicted',
                      xaxis_title='index',
                      yaxis_title='Target')
    st.plotly_chart(fig)

def main():
    st.title("CT-PIU Gap Measurement")

    if "score1" not in st.session_state:
        st.session_state.score1 = None
    if "score2" not in st.session_state:
        st.session_state.score2 = None
    if "model" not in st.session_state:
        st.session_state.model = None
    if "result" not in st.session_state:
        st.session_state.result = None
    if "reports" not in st.session_state:
        st.session_state.reports = None
    if "xls_count" not in st.session_state:
        st.session_state.xls_count = None
    if "target_expression" not in st.session_state:
        st.session_state.target_expression = None
    if "data_loaded" not in st.session_state:
        st.session_state.data_loaded = False

    folder_path = st.text_input("Enter the path to the main folder")

    if folder_path:
        if st.button("Count .xls Files and Calculate Max Values"):
```

```
            if os.path.isdir(folder_path):
                xls_count = count_xls_files(folder_path)
                st.session_state.xls_count = xls_count
                st.success(f"Total number of .xls files: {xls_count}")

    if st.session_state.xls_count is not None:
        target_expression = st.text_input("Enter the target expression (e.g., __OD__)", "_OD_")
        if st.button("Load Data"):
            st.session_state.target_expression = target_expression
            st.session_state.data_loaded = True

    if st.session_state.data_loaded:
        remove_noise = st.sidebar.radio("Remove Noise and Feature Extraction", ('Yes', 'No'))
        if remove_noise == "Yes":
            window_size = st.sidebar.slider("Window Size", 3, 51, step=2, value=5)
            poly_order = st.sidebar.slider("Polynomial Order", 1, 5, value=2)
            if "smooth_df" not in st.session_state:
                with st.spinner("Processing..."):
                    st.session_state.smooth_df = df_maker_smooth(folder_path, window_size,
poly_order, st.session_state.target_expression)
            df = st.session_state.smooth_df
            st.write("Noise removed using Savitzky-Golay filter.")
            st.write(df)
            st.write("Shape", df.shape)
        else:
            if "unsmooth_df" not in st.session_state:
                with st.spinner("Processing..."):
                    st.session_state.unsmooth_df = df_maker_unsmooth(folder_path,
st.session_state.target_expression)
            df = st.session_state.unsmooth_df
            st.write("Using raw data.")
            st.write(df)
            st.write("Shape", df.shape)

        threshold = st.sidebar.slider("Correlation Threshold", 0.0, 1.0, 0.9)
        if st.sidebar.button("Correlation"):
            st.session_state.reduced_features = reduce_features(df, threshold)
            reduced_features = st.session_state.reduced_features
            st.write("Reduced features based on correlation.", reduced_features)
            st.write("Shape", reduced_features.shape)
        else:
            if "reduced_features" in st.session_state:
                reduced_features = st.session_state.reduced_features
            else:
```

```
            reduced_features = df

        scale_data_flag = st.sidebar.checkbox("Scale Data", value=False)
        if scale_data_flag:
            st.write(st.session_state.reduced_features)
            scaler_type = st.sidebar.selectbox("Scaler Type", ["Standard", "MinMax"])
            scaled_data, scaler = scale_data(reduced_features, scaler_type)
            st.write("Data scaled using", scaler_type, "scaler.", scaled_data)
        else:
            scaled_data = reduced_features

        learning_type = st.sidebar.selectbox("Learning Type", ["Classification", "Regression"])
        test_size = st.sidebar.slider("Test Size Split Ratio", 0.1, 0.5, 0.2)

        if st.sidebar.button("Split"):
            X_train, X_test, y_train, y_test = split(scaled_data, test_size)
            st.session_state.X_train = X_train
            st.session_state.X_test = X_test
            st.session_state.y_train = y_train
            st.session_state.y_test = y_test
            st.write(st.session_state.X_train.shape)

        if learning_type == "Classification":
            models_list = [
                "Random Forest Classifier",
                "Gradient Boosting Classifier",
                "AdaBoost Classifier",
                "Decision Tree Classifier",
                "SVC"
            ]
        else:
            models_list = [
                "Random Forest Regressor",
                "Gradient Boosting Regressor",
                "Decision Tree Regressor",
                "XGBoost Regressor",
                "SVR"
            ]

        selected_model = st.sidebar.selectbox("Select Model for feature importance:", models_list
+ ["All Models"])

        if selected_model == "All Models":
            if st.sidebar.button("Train"):
```

```
            if "X_train" in st.session_state and "X_test" in st.session_state and "y_train" in
st.session_state and "y_test" in st.session_state:
                st.session_state.result, st.session_state.reports, trained_models = models(
                    learning_type, selected_model, st.session_state.X_train, st.session_state.X_test,
                    st.session_state.y_train, st.session_state.y_test)
                if learning_type == "Classification":
                    plot_classification_accuracy(st.session_state.result)
                    for model_name, report in st.session_state.reports.items():
                        display_classification_report(st.session_state.result[model_name], report,
model_name)
                else:
                    plot_regression_metrics(st.session_state.result)
        else:

            if st.sidebar.button("Train"):
                if "X_train" in st.session_state and "X_test" in st.session_state and "y_train" in
st.session_state and "y_test" in st.session_state:
                    if selected_model != "All Models":
                        st.session_state.score1, st.session_state.score2, st.session_state.model =
models(
                            learning_type, selected_model, st.session_state.X_train,
st.session_state.X_test,
                            st.session_state.y_train, st.session_state.y_test)
                        if learning_type == "Classification":
                            display_classification_report(st.session_state.score1, st.session_state.score2)
                        else:
                            display_regression_metrics(st.session_state.score1, st.session_state.score2)

            if selected_model != "All Models" and st.sidebar.button("Show Feature Importance"):
                if "model" in st.session_state:
                    st.session_state.feature_importance_df =
display_feature_importance(st.session_state.model, st.session_state.X_train)
                    st.session_state.show_feature_selection = True
                else:
                    st.error("Train a single model first to see feature importance.")

            if st.session_state.get('show_feature_selection', False):
                if "model" in st.session_state:
                    num_top_features = st.sidebar.number_input("Enter the number of top features to
select:", min_value=1, max_value=len(st.session_state.feature_importance_df), value=10)
                    top_features =
st.session_state.feature_importance_df['Feature'].head(num_top_features).tolist()

                if learning_type == "Classification":
```

```
                models_list = [
                    "Random Forest Classifier",
                    "Gradient Boosting Classifier",
                    "AdaBoost Classifier",
                    "Decision Tree Classifier",
                    "SVC"
                ]
            else:
                models_list = [
                    "Random Forest Regressor",
                    "Gradient Boosting Regressor",
                    "Decision Tree Regressor",
                    "XGBoost Regressor",
                    "SVR"
                ]

            selected_model_imp = st.sidebar.selectbox("Select Model after feature importance:",
models_list)
            if st.sidebar.button("Train on Top N Features"):
                if "X_train" in st.session_state and "X_test" in st.session_state and "y_train" in
st.session_state and "y_test" in st.session_state:
                    st.session_state.X_train_top = st.session_state.X_train[top_features]
                    st.session_state.X_test_top = st.session_state.X_test[top_features]

                    if learning_type == "Regression":
                        st.session_state.score1, st.session_state.score2, st.session_state.model =
models(
                        learning_type, selected_model_imp, st.session_state.X_train_top,
st.session_state.X_test_top,
                        st.session_state.y_train, st.session_state.y_test)
                        display_regression_metrics(st.session_state.score1, st.session_state.score2)
                    else:
                        st.session_state.score1, st.session_state.score2, st.session_state.model =
models(
                        learning_type, selected_model_imp, st.session_state.X_train_top,
st.session_state.X_test_top,
                        st.session_state.y_train, st.session_state.y_test)
                        display_classification_report(st.session_state.score1, st.session_state.score2)
                else:
                    st.error("Please split the data before training the model.")

            if st.sidebar.checkbox("Hyperparamter Tuning", value=False):
                method = st.sidebar.selectbox("Hyperparameter Tuning Method", ["Randomized
Search", "Grid Search"])
```

```
            if st.sidebar.button("Train After Hyperparameter Tuning"):
                with st.spinner("Performing Hyperparameter Tuning..."):

                    st.session_state.model, st.session_state.score1, st.session_state.score2,
st.session_state.y_pred = hyperparameter_tuning(st.session_state.model,
st.session_state.X_train[top_features], st.session_state.X_test[top_features],
st.session_state.y_train, st.session_state.y_test, method)
                    if learning_type == "Regression":
                        display_regression_metrics(st.session_state.score1,
st.session_state.score2)
                        barplot(st.session_state.y_test, st.session_state.y_pred)
                    else:
                        display_classification_report(st.session_state.score1,
st.session_state.score2)
                        barplot(st.session_state.y_test, st.session_state.y_pred)
            else:
                st.warning("Perform Hyperparameter Tuning and click Train After
Hyperparameter Tuning.")

if __name__ == "__main__":
    main()
```

PRIYANKA:

```
import streamlit as st
import os
import pandas as pd
import numpy as np
from scipy.signal import savgol_filter
import tsfel
from sklearn.preprocessing import MinMaxScaler, StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, classification_report, mean_squared_error,
r2_score
from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier,
AdaBoostClassifier, RandomForestRegressor, GradientBoostingRegressor
from sklearn.tree import DecisionTreeClassifier, DecisionTreeRegressor
from sklearn.model_selection import RandomizedSearchCV, GridSearchCV
from xgboost import XGBClassifier, XGBRegressor
from sklearn.svm import SVC, SVR
import plotly.graph_objects as go
import re
import streamlit.components.v1 as components
import plotly.express as px

def select_signal_window(data):
    """
    Function to allow the user to select a window or range of the signal data.

    Parameters:
    data (DataFrame): The loaded signal data.

    Returns:
    DataFrame: The subset of the signal data within the specified range.
    """
    st.write("Signal Data Loaded:")
    st.write(data)

    # Get the maximum index value to set the slider range
    max_index = len(data) - 1

    # Add sliders to select the start and end points of the signal window
    start_point = st.slider("Select Start Point of Signal Range", 0, max_index, 0)
    end_point = st.slider("Select End Point of Signal Range", start_point, max_index, max_index)

    # Extract the selected range
```

```
    selected_data = data.iloc[start_point:end_point+1]

    st.write("Selected Signal Data Range:")
    st.write(selected_data)

    return selected_data


def display_classification_report(accuracy, report, model_name=None):
    model_info = f"<h2 style='color: #000000; text-align: center; font-family: Arial,
sans-serif;'>{model_name}</h2>" if model_name else ""
    report_html = f"""
    <div style="border: 1px solid #E0E0E0; border-radius: 10px; padding: 20px;
background-color: #FAFAFA; box-shadow: 0 4px 8px 0 rgba(0, 0, 0, 0.2); font-family: Arial,
sans-serif;">
        {model_info}
        <h3 style='color: #000000; text-align: center;'>Accuracy: {accuracy:.2f}</h3>
        <h4 style='color: ##000000;'>Report:</h4>
        <pre style="font-size: small; background-color: #FFFFFF; padding: 15px; border-radius:
5px; overflow: auto; white-space: pre-wrap; border: 1px solid #E0E0E0;">{report}</pre>
    </div>
    """
    components.html(report_html, height=430, scrolling=True)

def display_regression_metrics(mse, r2, model_name=None):
    model_info = f"<h2 style='color: #000000; text-align: center; font-family: Arial,
sans-serif;'>{model_name}</h2>" if model_name else ""
    metrics_html = f"""
    <div style="border: 1px solid #E0E0E0; border-radius: 10px; padding: 20px;
background-color: #FAFAFA; box-shadow: 0 4px 8px 0 rgba(0, 0, 0, 0.2); font-family: Arial,
sans-serif;">
        {model_info}
        <h3 style='color: #000000; text-align: center;'>MSE: {mse:.2f}</h3>
        <h3 style='color: #000000; text-align: center;'>R2 Score: {r2:.2f}</h3>
    </div>
    """
    components.html(metrics_html, height=250, scrolling=True)

def load_data_from_folder(folder_path):
    data = []
    for subdir, _, files in os.walk(folder_path):
        for file in files:
            if file.endswith('.csv'):
                file_path = os.path.join(subdir, file)
```

```
            df = pd.read_csv(file_path, header=None)
            df['filename'] = file
            data.append(df)
    data = pd.DataFrame(data)
    return data

def count_xls_files(folder_path):
    xls_count = 0
    for root, dirs, files in os.walk(folder_path):
        for file in files:
            if file.endswith('.csv'):
                xls_count += 1
                file_path = os.path.join(root, file)
    return xls_count

def df_maker_smooth(path, window_size, poly_order, target_expression):
    all_features_df = pd.DataFrame()
    for subfolder in os.listdir(path):
        subfolder_path = os.path.join(path, subfolder)
        if os.path.isdir(subfolder_path):
            for file in os.listdir(subfolder_path):
                if file.endswith(".csv"):
                    file_path = os.path.join(subfolder_path, file)
                    df = pd.read_csv(file_path)
                    df['filename'] = file
                    signal = df.iloc[:, 0].values
                    smoothened_signal = golay_filter(signal, window_size, poly_order)
                    smoothened_signal_df = pd.DataFrame(smoothened_signal)
                    df['target'] = df['filename'].str.extract(f'({target_expression})(\d+)')[1].astype(float)
                    cfg = tsfel.get_features_by_domain()
                    features = tsfel.time_series_features_extractor(cfg, smoothened_signal_df.iloc[:, 0],
1000000)
                    features_transposed = features.T.reset_index(drop=True).T
                    features_transposed['target'] = df['target']
                    all_features_df = pd.concat([all_features_df, features_transposed],
ignore_index=True)
    return all_features_df

def df_maker_unsmooth(path, target_expression):
    all_features_df = pd.DataFrame()
    for subfolder in os.listdir(path):
        subfolder_path = os.path.join(path, subfolder)
        if os.path.isdir(subfolder_path):
            for file in os.listdir(subfolder_path):
```

```
            if file.endswith(".csv"):
                file_path = os.path.join(subfolder_path, file)
                df = pd.read_csv(file_path)
                df['filename'] = file
                signal = df.iloc[:, 0].values
                df['target'] = df['filename'].str.extract(f'({target_expression})(\d+)')[1].astype(float)
                cfg = tsfel.get_features_by_domain()
                features = tsfel.time_series_features_extractor(cfg, df.iloc[:, 0], 1000000)
                features_transposed = features.T.reset_index(drop=True).T
                features_transposed['target'] = df['target']
                all_features_df = pd.concat([all_features_df, features_transposed],
ignore_index=True)
    return all_features_df

def reduce_features(features, threshold):
    corr_matrix = features.corr().abs()
    upper = corr_matrix.where(np.triu(np.ones(corr_matrix.shape), k=1).astype(bool))
    to_drop = [column for column in upper.columns if any(upper[column] > threshold)]
    reduced_features = features.drop(columns=to_drop)
    return reduced_features

def golay_filter(signal, window_size, poly_order):
    return savgol_filter(signal, window_length=window_size, polyorder=poly_order)

def scale_data(data, scaler_type):
    if scaler_type == "MinMax":
        scaler = MinMaxScaler()
    else:
        scaler = StandardScaler()
    scaled_data = pd.DataFrame(scaler.fit_transform(data.drop(columns=['target'])),
columns=data.columns[:-1])
    scaled_data['target'] = data['target']
    return scaled_data, scaler

def split(scaled_data, test_size):
    X = scaled_data.drop(columns=['target'])
    y = scaled_data['target']
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=test_size, random_state=42)
    return X_train, X_test, y_train, y_test

def models(learning_type, selected_model, X_train, X_test, y_train, y_test):
    if learning_type == "Classification":
        models = {
            "Random Forest Classifier": RandomForestClassifier(),
```

```
            "Gradient Boosting Classifier": GradientBoostingClassifier(),
            "AdaBoost Classifier": AdaBoostClassifier(),
            "Decision Tree Classifier": DecisionTreeClassifier(),
            "SVC": SVC()
        }
        if selected_model != "All Models":
            model = models[selected_model]
            model.fit(X_train, y_train)
            y_pred = model.predict(X_test)
            accuracy = accuracy_score(y_test, y_pred)
            report = classification_report(y_test, y_pred)
            return accuracy, report, model
        else:
            results = {}
            reports = {}
            trained_models = {}
            for name, model in models.items():
                model.fit(X_train, y_train)
                y_pred = model.predict(X_test)
                accuracy = accuracy_score(y_test, y_pred)
                report = classification_report(y_test, y_pred)
                results[name] = accuracy
                reports[name] = report
                trained_models[name] = model
            return results, reports, trained_models
    else:
        models = {
            "Random Forest Regressor": RandomForestRegressor(),
            "Gradient Boosting Regressor": GradientBoostingRegressor(),
            "Decision Tree Regressor": DecisionTreeRegressor(),
            "XGBoost Regressor": XGBRegressor(),
            "SVR": SVR()
        }
        if selected_model != "All Models":
            model = models[selected_model]
            model.fit(X_train, y_train)
            y_pred = model.predict(X_test)
            mse = mean_squared_error(y_test, y_pred)
            r2 = r2_score(y_test, y_pred)
            return mse, r2, model
        else:
            results = {}
            reports = {}
            trained_models = {}
```

```
            for name, model in models.items():
                model.fit(X_train, y_train)
                y_pred = model.predict(X_test)
                mse = mean_squared_error(y_test, y_pred)
                r2 = r2_score(y_test, y_pred)
                results[name] = (mse, r2)
                trained_models[name] = model
            return results, reports, trained_models

def plot_classification_accuracy(results):
    accuracy_df = pd.DataFrame(columns=['Accuracy'])
    for model_name, accuracy in results.items():
        accuracy_df.loc[model_name] = [accuracy]

    st.write("Model Performance Comparison (Classification):")
    st.write(accuracy_df)

    fig = go.Figure()
    fig.add_trace(go.Bar(
        x=accuracy_df.index,
        y=accuracy_df['Accuracy'],
        name='Accuracy',
        marker_color='green'
    ))
    fig.update_layout(
        xaxis_tickangle=-45,
        title='Model Performance Comparison (Classification)',
        xaxis_title='Model',
        yaxis_title='Accuracy'
    )
    st.plotly_chart(fig)

def plot_regression_metrics(results):
    mse_r2_df = pd.DataFrame(columns=['MSE', 'R2 Score'])
    for model_name, scores in results.items():
        mse, r2 = scores
        mse_r2_df.loc[model_name] = [mse, r2]

    st.write("Model Performance Comparison (Regression):")
    st.write(mse_r2_df)

    fig = go.Figure()
    fig.add_trace(go.Bar(
        x=mse_r2_df.index,
```

```
        y=mse_r2_df['MSE'],
        name='MSE',
        marker_color='blue'
    ))
    fig.add_trace(go.Bar(
        x=mse_r2_df.index,
        y=mse_r2_df['R2 Score'],
        name='R2 Score',
        marker_color='orange'
    ))
    fig.update_layout(
        barmode='group',
        xaxis_tickangle=-45,
        title='Model Performance Comparison (Regression)',
        xaxis_title='Model',
        yaxis_title='Value'
    )
    st.plotly_chart(fig)

def display_feature_importance(model, X_train):
    if hasattr(model, 'feature_importances_'):
        importances = model.feature_importances_
        feature_names = X_train.columns
        feature_importance_df = pd.DataFrame({
            'Feature': feature_names,
            'Importance': importances
        }).sort_values(by='Importance', ascending=False)

        st.session_state.feature_importance_df = feature_importance_df

        fig = go.Figure([go.Bar(x=feature_importance_df['Feature'],
y=feature_importance_df['Importance'])])
        fig.update_layout(title='Feature Importance', xaxis_title='Feature', yaxis_title='Importance')

        st.plotly_chart(fig)
    else:
        st.warning("Selected model does not support feature importance.")

    return feature_importance_df

def feature_name(model, X_train, top_n_features):
    if hasattr(model, 'feature_importances_'):
        importances = model.feature_importances_
        feature_names = X_train.columns
```

```
        top_features_df = pd.DataFrame({
            'Feature': feature_names,
            'Importance': importances
        }).sort_values(by='Importance', ascending=False)
        top_features_df = top_features_df['Feature'].head(top_n_features)

    return top_features_df

def hyperparameter_tuning(model, X_train, X_test, y_train, y_test, method='Randomized
Search'):
    # Define parameter grids for different models
    param_grids = {
        RandomForestClassifier: {
            'n_estimators': [50, 100, 200, 300],
            'max_depth': [3, 5, 10, None],
            'min_samples_split': [2, 5, 10],
            'min_samples_leaf': [1, 2, 4],
            'bootstrap': [True, False]
        },
        RandomForestRegressor: {
            'n_estimators': [50, 100, 200, 300],
            'max_depth': [3, 5, 10, None],
            'min_samples_split': [2, 5, 10],
            'min_samples_leaf': [1, 2, 4],
            'bootstrap': [True, False]
        },
        GradientBoostingClassifier: {
            'n_estimators': [50, 100, 200],
            'learning_rate': [0.01, 0.1, 0.05],
            'max_depth': [3, 4, 5],
            'subsample': [0.7, 0.8, 0.9, 1.0]
        },
        GradientBoostingRegressor: {
            'n_estimators': [50, 100, 200],
            'learning_rate': [0.01, 0.1, 0.05],
            'max_depth': [3, 4, 5],
            'subsample': [0.7, 0.8, 0.9, 1.0]
        },
        XGBRegressor: {
            'n_estimators': [50, 100, 200],
            'learning_rate': [0.01, 0.1, 0.05],
            'max_depth': [3, 4, 5],
            'subsample': [0.7, 0.8, 0.9, 1.0]
        },
```

```
        DecisionTreeClassifier: {
            'max_depth': [None, 10, 20, 30],
            'min_samples_split': [2, 5, 10],
            'min_samples_leaf': [1, 2, 4]
        },
        DecisionTreeRegressor: {
        'max_depth': [3, 5, 10, None],
        'min_samples_split': [2, 5, 10],
        'min_samples_leaf': [1, 2, 4]
        },
        SVR: {
            'kernel': ['linear', 'poly', 'rbf', 'sigmoid'],
            'C': [0.1, 1, 10, 100],
            'gamma': ['scale', 'auto']
        },
        SVC: {
            'C': [0.1, 1, 10],
            'kernel': ['linear', 'rbf', 'poly'],
            'degree': [2, 3, 4]
        },
        AdaBoostClassifier: {
            'n_estimators': [50, 100, 200],
            'learning_rate': [0.01, 0.1, 0.05],
            'algorithm': ['SAMME', 'SAMME.R']
        }
    }

    # Get the parameter grid for the model
    param_grid = param_grids[type(model)]

    if method == 'Randomized Search':
        tuner = RandomizedSearchCV(estimator=model, param_distributions=param_grid,
n_iter=100, cv=3, verbose=2, random_state=42, n_jobs=-1)
    elif method == 'Grid Search':
        tuner = GridSearchCV(estimator=model, param_grid=param_grid, cv=3, verbose=2,
n_jobs=-1)
    else:
        raise ValueError("Invalid method. Choose either 'Randomized Search' or 'Grid Search'.")

    tuner.fit(X_train, y_train)
    best_model = tuner.best_estimator_

    if isinstance(best_model, (RandomForestRegressor, GradientBoostingRegressor,
XGBRegressor, DecisionTreeRegressor, SVR)):
```

```
        y_pred = best_model.predict(X_test)
        score1 = mean_squared_error(y_test, y_pred)
        score2 = r2_score(y_test, y_pred)
    elif isinstance(best_model, (RandomForestClassifier, GradientBoostingClassifier,
DecisionTreeClassifier, AdaBoostClassifier, SVC)):
        y_pred = best_model.predict(X_test)
        score1 = accuracy_score(y_test, y_pred)
        score2 = classification_report(y_test, y_pred)
    else:
        raise TypeError("Unsupported model type for evaluation.")

    return best_model, score1, score2, y_pred

def barplot(y_test,y_pred):
    #Plot
    fig=go.Figure()
    fig.add_trace(go.Bar(y=y_test,name='Actual',marker_color='blue'))
    fig.add_trace(go.Bar(y=y_pred,name='Predicted',marker_color='orange'))
    fig.update_layout(title=f'Actual vs Predicted',
                      xaxis_title='index',
                      yaxis_title='Target')
    st.plotly_chart(fig)

# def extract_top_features(data, top_features):
#     return data[top_features]

# def evaluate_model_on_test_data(model, test_data, learning_type):
#     X_test = test_data.drop(columns=['target'])
#     y_test = test_data['target']
#     y_pred = model.predict(X_test)
#     if learning_type == "Classification":
#         accuracy = accuracy_score(y_test, y_pred)
#         report = classification_report(y_test, y_pred)
#         return accuracy, report, y_pred
#     else:
#         mse = mean_squared_error(y_test, y_pred)
#         r2 = r2_score(y_test, y_pred)
#         return mse, r2, y_pred

# def load_test_data(test_data_path):
#     test_data = pd.read_csv(test_data_path)
#     return test_data
```

```
# def process_data(folder_path, target_expression, remove_noise, window_size, poly_order,
threshold, scale_data_flag, scaler_type):
#     if remove_noise == "Yes":
#         df = df_maker_smooth(folder_path, window_size, poly_order, target_expression)
#     else:
#         df = df_maker_unsmooth(folder_path, target_expression)

#     reduced_features = reduce_features(df, threshold)

#     if scale_data_flag:
#         scaled_data, scaler = scale_data(reduced_features, scaler_type)
#         return scaled_data, scaler, reduced_features.columns
#     else:
#         return reduced_features, None, reduced_features.columns

def load_and_preprocess_test_data(test_data_path, remove_noise, window_size, poly_order,
target_expression, scaler):
    if remove_noise:
        test_df = df_maker_smooth(test_data_path, window_size, poly_order, target_expression)
    else:
        test_df = df_maker_unsmooth(test_data_path, target_expression)

    # Scaling test data using the previously chosen scaler
    if scaler:
        scaled_test_data = pd.DataFrame(scaler.transform(test_df.drop(columns=['target'])),
columns=test_df.columns[:-1])
        scaled_test_data['target'] = test_df['target']
    else:
        scaled_test_data = test_df

    return scaled_test_data

def extract_top_features(test_data, top_features):
    return test_data[top_features.tolist() + ['target']]

def evaluate_test_data(test_data, model, learning_type):
    X_test = test_data.drop(columns=['target'])
    y_test = test_data['target']

    y_pred = model.predict(X_test)

    if learning_type == "Classification":
        accuracy = accuracy_score(y_test, y_pred)
        report = classification_report(y_test, y_pred)
```

```
        return accuracy, report, y_test, y_pred
    else:
        mse = mean_squared_error(y_test, y_pred)
        r2 = r2_score(y_test, y_pred)
        return mse, r2, y_test, y_pred


def main():
    st.title("CT-PIU Gap Measurement")

    if "score1" not in st.session_state:
        st.session_state.score1 = None
    if "score2" not in st.session_state:
        st.session_state.score2 = None
    if "model" not in st.session_state:
        st.session_state.model = None
    if "result" not in st.session_state:
        st.session_state.result = None
    if "reports" not in st.session_state:
        st.session_state.reports = None
    if "xls_count" not in st.session_state:
        st.session_state.xls_count = None
    if "target_expression" not in st.session_state:
        st.session_state.target_expression = None
    if "data_loaded" not in st.session_state:
        st.session_state.data_loaded = False
    if 'scaler' not in st.session_state:
        st.session_state.scaler = None
    if 'selected_features' not in st.session_state:
        st.session_state.selected_features = None

    folder_path = st.text_input("Enter the path to the main folder")

    if folder_path:
        if st.button("Count .xls Files and Calculate Max Values"):
            if os.path.isdir(folder_path):
                xls_count = count_xls_files(folder_path)
                st.session_state.xls_count = xls_count
                st.success(f"Total number of .xls files: {xls_count}")

    if st.session_state.xls_count is not None:
        target_expression = st.text_input("Enter the target expression (e.g., __OD__)", "_OD_")
        if st.button("Load Data"):
            st.session_state.target_expression = target_expression
```

```
            st.session_state.data_loaded = True

    if st.session_state.data_loaded:
        remove_noise = st.sidebar.radio("Remove Noise and Feature Extraction", ('Yes', 'No'))
        if remove_noise == "Yes":
            window_size = st.sidebar.slider("Window Size", 3, 51, step=2, value=5)
            poly_order = st.sidebar.slider("Polynomial Order", 1, 5, value=2)
            if "smooth_df" not in st.session_state:
                with st.spinner("Processing..."):
                    st.session_state.smooth_df = df_maker_smooth(folder_path, window_size,
poly_order, st.session_state.target_expression)
            df = st.session_state.smooth_df
            df = select_signal_window(st.session_state.smooth_df)
            st.write("Noise removed using Savitzky-Golay filter.")
            st.write(df)
            st.write("Shape", df.shape)
        else:
            if "unsmooth_df" not in st.session_state:
                with st.spinner("Processing..."):
                    st.session_state.unsmooth_df = df_maker_unsmooth(folder_path,
st.session_state.target_expression)
            df = st.session_state.unsmooth_df
            df = select_signal_window(st.session_state.unsmooth_df)
            st.write("Using raw data.")
            st.write(df)
            st.write("Shape", df.shape)

        threshold = st.sidebar.slider("Correlation Threshold", 0.0, 1.0, 0.9)
        if st.sidebar.button("Correlation"):
            st.session_state.reduced_features = reduce_features(df, threshold)
            reduced_features = st.session_state.reduced_features
            st.write("Reduced features based on correlation.", reduced_features)
            st.write("Shape", reduced_features.shape)
        else:
            if "reduced_features" in st.session_state:
                reduced_features = st.session_state.reduced_features
            else:
                reduced_features = df

        scale_data_flag = st.sidebar.checkbox("Scale Data", value=False)
        if scale_data_flag:
            st.write(st.session_state.reduced_features)
            scaler_type = st.sidebar.selectbox("Scaler Type", ["Standard", "MinMax"])
            scaled_data, scaler = scale_data(reduced_features, scaler_type)
```

```
            st.write("Data scaled using", scaler_type, "scaler.", scaled_data)
        else:
            scaled_data = reduced_features

        learning_type = st.sidebar.selectbox("Learning Type", ["Classification", "Regression"])
        test_size = st.sidebar.slider("Test Size Split Ratio", 0.1, 0.5, 0.2)

        if st.sidebar.button("Split"):
            X_train, X_test, y_train, y_test = split(scaled_data, test_size)
            st.session_state.X_train = X_train
            st.session_state.X_test = X_test
            st.session_state.y_train = y_train
            st.session_state.y_test = y_test
            st.write(st.session_state.X_train.shape)

        if learning_type == "Classification":
            models_list = [
                "Random Forest Classifier",
                "Gradient Boosting Classifier",
                "AdaBoost Classifier",
                "Decision Tree Classifier",
                "SVC"
            ]
        else:
            models_list = [
                "Random Forest Regressor",
                "Gradient Boosting Regressor",
                "Decision Tree Regressor",
                "XGBoost Regressor",
                "SVR"
            ]

        selected_model = st.sidebar.selectbox("Select Model for feature importance:", models_list
+ ["All Models"])

        if st.sidebar.button("Train"):
            if "X_train" in st.session_state and "X_test" in st.session_state and "y_train" in
st.session_state and "y_test" in st.session_state:
                if selected_model == "All Models" and learning_type == "Classification":
                    st.session_state.result, st.session_state.reports, trained_models = models(
                        learning_type, selected_model, st.session_state.X_train, st.session_state.X_test,
                        st.session_state.y_train, st.session_state.y_test)
                    plot_classification_accuracy(st.session_state.result)
                    for model_name, report in st.session_state.reports.items():
```

```
                display_classification_report(st.session_state.result[model_name], report,
model_name)
            elif selected_model == "All Models" and learning_type == "Regression":
                st.session_state.result, _, trained_models = models(
                    learning_type, selected_model, st.session_state.X_train, st.session_state.X_test,
                    st.session_state.y_train, st.session_state.y_test)
                plot_regression_metrics(st.session_state.result)
            else:
                st.session_state.score1, st.session_state.score2, st.session_state.model = models(
                    learning_type, selected_model, st.session_state.X_train, st.session_state.X_test,
                    st.session_state.y_train, st.session_state.y_test)
                if learning_type == "Regression":
                    display_regression_metrics(st.session_state.score1, st.session_state.score2)
                else:
                    display_classification_report(st.session_state.score1, st.session_state.score2)
        else:
            st.error("Please split the data before training the model.")

    if selected_model != "All Models" and st.sidebar.button("Show Feature Importance"):
        if "model" in st.session_state:
            st.session_state.feature_importance_df=
display_feature_importance(st.session_state.model, st.session_state.X_train)
            st.session_state.show_feature_selection = True
        else:
            st.error("Train a single model first to see feature importance.")

    # After showing feature importance, allow user to input number of top features
    if st.session_state.get('show_feature_selection', False):
        if "model" in st.session_state:
            num_top_features = st.sidebar.number_input("Enter the number of top features to
select:", min_value=1, max_value=len(st.session_state.feature_importance_df), value=10)
            top_features =
st.session_state.feature_importance_df['Feature'].head(num_top_features).tolist()

#               st.session_state.top_feature_names = feature_name(model, X_train, top_features)
        if learning_type == "Classification":
                models_list = [
                    "Random Forest Classifier",
                    "Gradient Boosting Classifier",
                    "AdaBoost Classifier",
                    "Decision Tree Classifier",
                    "SVC"
                ]
        else:
```

```
            models_list = [
                "Random Forest Regressor",
                "Gradient Boosting Regressor",
                "Decision Tree Regressor",
                "XGBoost Regressor",
                "SVR"
            ]

        selected_model_imp = st.sidebar.selectbox("Select Model after feature importance:",
models_list)
        if st.sidebar.button("Train on Top N Features"):
            if "X_train" in st.session_state and "X_test" in st.session_state and "y_train" in
st.session_state and "y_test" in st.session_state:
                # Filter X_train and X_test to keep only top features
                st.session_state.X_train_top = st.session_state.X_train[top_features]
                st.session_state.X_test_top = st.session_state.X_test[top_features]

                if learning_type == "Regression":
                    st.session_state.score1, st.session_state.score2, st.session_state.model =
models(
                    learning_type, selected_model_imp, st.session_state.X_train_top,
st.session_state.X_test_top,
                    st.session_state.y_train, st.session_state.y_test)
                    display_regression_metrics(st.session_state.score1, st.session_state.score2)
                else:
                    st.session_state.score1, st.session_state.score2, st.session_state.model =
models(
                    learning_type, selected_model_imp, st.session_state.X_train_top,
st.session_state.X_test_top,
                    st.session_state.y_train, st.session_state.y_test)
                    display_classification_report(st.session_state.score1, st.session_state.score2)
            else:
                st.error("Please split the data before training the model.")
        # else:
        #     st.error("Train a single model first to see feature importance.")

        if st.sidebar.checkbox("Hyperparamter Tuning", value=False):
            method = st.sidebar.selectbox("Hyperparameter Tuning Method", ["Randomized
Search", "Grid Search"])
            if st.sidebar.button("Train After Hyperparameter Tuning"):
                with st.spinner("Performing Hyperparameter Tuning..."):
                    st.session_state.model, st.session_state.score1, st.session_state.score2,
st.session_state.y_pred = hyperparameter_tuning(st.session_state.model,
```

```
st.session_state.X_train[top_features], st.session_state.X_test[top_features],
st.session_state.y_train, st.session_state.y_test, method)
                    if learning_type == "Regression":
                        display_regression_metrics(st.session_state.score1, st.session_state.score2)
                        barplot(st.session_state.y_test, st.session_state.y_pred)
                    else:
                        display_classification_report(st.session_state.score1, st.session_state.score2)
                        barplot(st.session_state.y_test, st.session_state.y_pred)
            # else:
            #     st.warning("Perform Hyperparameter Tuning and click Train After Hyperparameter
Tuning.")

        # test_data_path = st.sidebar.text_input("Enter the path to the test data file:")
        # if test_data_path:
        #     if st.sidebar.button("Evaluate on Test Data"):
        #         # test_data = load_data_from_folder(test_data_path)

        #         # Apply the same preprocessing steps as the training data
        #         test_data, _, _ = process_data(
        #             test_data_path, st.session_state.target_expression, remove_noise,
        #             window_size, poly_order, threshold, scale_data_flag, scaler_type
        #         )

        #         # Apply the same scaling to test data
        #         if st.session_state.scaler is not None:
        #             test_data = pd.DataFrame(st.session_state.scaler.transform(test_data),
columns=test_data.columns)

        #         # Select the same features as in training data
        #         test_data = test_data[top_features]
        #         # test_data = extract_top_features(test_data, top_features + ['target'])

        #         # Display processed test data
        #         if 'test_data' in st.session_state:
        #             st.write("Processed Test Data:", st.session_state.test_data)
        #             st.write("Shape:", st.session_state.test_data.shape)

        #         # if learning_type == "Classification":
        #         #     accuracy, report, y_pred =
evaluate_model_on_test_data(st.session_state.model, test_data, learning_type)
        #         #     display_classification_report(accuracy, report)
        #         # else:
        #         #     mse, r2, y_pred = evaluate_model_on_test_data(st.session_state.model,
test_data, learning_type)
```

```
        #         #     display_regression_metrics(mse, r2)
        #         # barplot(test_data['target'], y_pred)
        #     else:
        #         st.error("Please provide a valid test data file path.")

        test_data_path = st.sidebar.text_input("Enter the path to the test data folder")

        if test_data_path and st.sidebar.button("Load and Evaluate Test Data"):
            if "top_features" in st.session_state:
                with st.spinner("Processing and evaluating test data..."):
                    test_data = load_and_preprocess_test_data(test_data_path, remove_noise ==
"Yes", window_size, poly_order, st.session_state.target_expression, st.session_state.scaler)
                    test_data = extract_top_features(test_data, st.session_state.top_features)

                    st.write("Extracted test data with top features:")
                    st.write(test_data)
                    st.write("Shape:", test_data.shape)

                    if learning_type == "Classification":
                        accuracy, report, y_test, y_pred = evaluate_test_data(test_data,
st.session_state.model, learning_type)
                        display_classification_report(accuracy, report)
                    else:
                        mse, r2, y_test, y_pred = evaluate_test_data(test_data, st.session_state.model,
learning_type)
                        display_regression_metrics(mse, r2)

                    barplot(y_test, y_pred)

if __name__ == "__main__":
    main()
```