Data-driven Design and Analyses of Structures and Materials (3dasm)

Lecture 11

Miguel A. Bessa | **M.A.Bessa@tudelft.nl** | Associate Professor

OPTION 1. Run this notebook **locally in your computer**:

1. Confirm that you have the 3dasm conda environment (see Lecture 1).
2. Go to the 3dasm_course folder in your computer and pull the last updates of the **repository**:

```
git pull
```

3. Open command window and load jupyter notebook (it will open in your internet browser):

```
conda activate 3dasm
jupyter notebook
```

4. Open notebook of this Lecture.

OPTION 2. Use **Google's Colab** (no installation required, but times out if idle):

1. go to **https://colab.research.google.com**
2. login
3. File > Open notebook
4. click on Github (no need to login or authorize anything)
5. paste the git link: **https://github.com/bessagroup/3dasm_course**
6. click search and then click on the notebook for this Lecture.

Outline for today

- Continuation of tutorial on supervised learning (again, no theory today!)
    - Multidimensional regression with Gaussian Processes and Artificial Neural Networks
- Classification with supervised learning: quick tutorial using Decision Trees and Support Vector Machines

Reading material: This notebook + (GPs in Section 17.3 of book; ANNs in Chapter 13)

Today's lecture is also going to be more practical

Similarly to the previous lecture, we will focus on the practical aspects **without deriving the models yet**.

- Regression via supervised learning in multiple dimensions:
    - Today we will just focus only on **Gaussian Processes** using **scikit-learn**

As we will see today, using ML models for multidimensional regression is very similar to the one-dimensional case we covered in the previous lectures.

Creating a noiseless multidimensional dataset

Before we start, we need to consider a particular dataset for which we want to do regression.
As before, instead of downloading a particular dataset, let's create our own.

- There is a nice website listing many different benchmark functions used in optimization: **https://www.sfu.ca/~ssurjano/optimization.html**

We will learn some of them using Gaussian Processes...

In [2]:

```
# Uncomment line below if you want to enable live rotation of the surface plots.
#%matplotlib notebook
#
from matplotlib import cm # to change colors of surface plots
#
# Let's define some functions that are often used to benchmark
# algorithms (especially in optimization)
def schwefel( x ):
    function_name = 'Schwefel' # to output name of function
    x = np.asarray_chkfinite(x)  # ValueError if any NaN or Inf
    if x.ndim == 1:
        x = np.reshape(x,(-1,2)) # reshape into 2d array
    #
    n_points, n_features = np.shape(x)
    y = np.empty((n_points,1))
    #
    for ii in range(n_points):
        y[ii] = 418.9829*n_features - sum( x[ii,:] * np.sin( np.sqrt( abs( x[ii,:] ))))
    return (np.atleast_1d(y), function_name)
#
# The next cell (hidden in presentation) includes a few more functions (so that you can play with this notebook).
```

Consider equally spaced sampling points to evaluate the function, as we did before for the 1D case.

In [4]:
```
# Uniform grid of points for two-dimensional input
n_points_per_dimension = 50
x1 = np.linspace(-10.0, 10.0, n_points_per_dimension)
x2 = np.linspace(-10.0, 10.0, n_points_per_dimension)
# x3 = np.linspace(-10.0, 10.0, n_points_per_dimension) # if the input was 3D

X1_grid, X2_grid = np.meshgrid(x1, x2) # creates a grid of points that is necessary to plot surfaces
#X1_grid, X2_grid, X3_grid = np.meshgrid(x1, x2, x3) # e.g. for 3D input

print("Note that the X1_grid (and X2_grid) are 2D arrays with the following size:\n", np.shape(X1_grid))

# Input points reshaped for Pandas dataframe
Data_X = np.array([X1_grid, X2_grid]).reshape(2, -1).T
#Data_X = np.array([X1_grid, X2_grid, X3_grid]).reshape(3, -1).T
```

```
Note that the X1_grid (and X2_grid) are 2D arrays with the following size:
 (50, 50)
```

Now evaluate the function of interest (e.g. Schwefel function) at these input points.

In [5]:
```
# Output data created from one of the benchmark functions:
# levy, ackley, rosenbrock, schwefel, griewank, rastrigin
Data_y, function_name = schwefel(Data_X) # calculate the output vector Data_y (1D output) from input data (2D input)

print("The output of the function is a vector with size:\n", np.shape(Data_y))

# Reshape output data for use in Surface plot:
Y_grid = np.reshape(Data_y,np.shape(X1_grid))

print("So, we reshape the output vector into a 2D array needed to plot surfaces:\n", np.shape(Y_grid))

# Set the color scheme used in every plot:
set_cm = cm.cool # viridis, inferno, copper, PuBu, cool, coolwarm, hsv
```

```
The output of the function is a vector with size:
 (2500, 1)
So, we reshape the output vector into a 2D array needed to plot surfaces:
 (50, 50)
```

Let's plot the function in a few different ways:

- Left subplot: 3D surface of the function
- Right subplot: a wireframe of the surface (no colors)

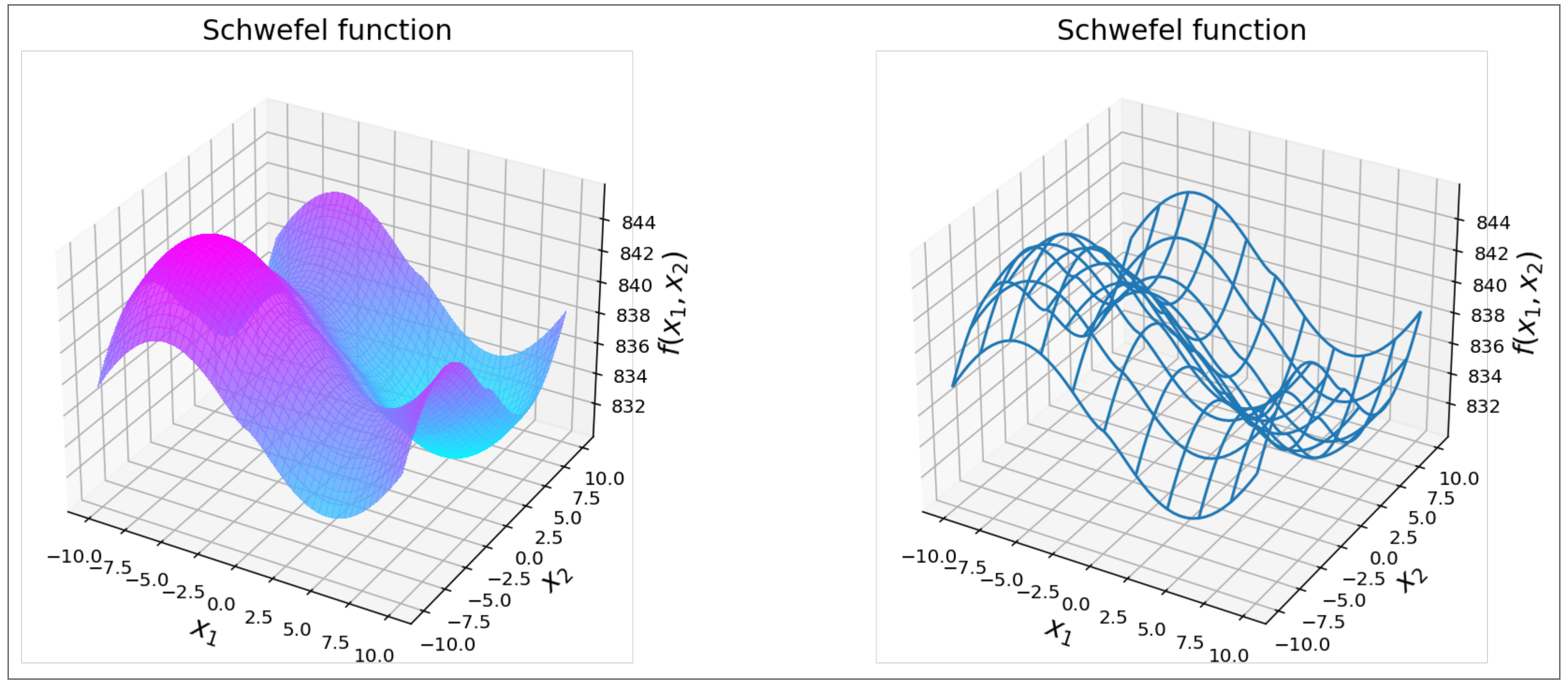
In [6]:

```
fig1 = plt.figure(figsize=plt.figaspect(0.5)); ax1 = [];
# Subplot 1 (left) of Figure 1
ax1.append(fig1.add_subplot(1, 2, 1, projection='3d')) # just a way to use the same variable for all axes of fig1.
# Surface plot:
surf = ax1[0].plot_surface(X1_grid, X2_grid, Y_grid, cmap=set_cm, alpha=0.8, linewidth=0, antialiased=False)
ax1[0].set_xlabel('$x_1$', fontsize=15)
ax1[0].set_ylabel('$x_2$', fontsize=15)
ax1[0].set_zlabel('$f(x_1,x_2)$', fontsize=15)
ax1[0].set_title("%s function" % function_name, fontsize=15)

# Subplot 2 (right) of Figure 1
ax1.append(fig1.add_subplot(1, 2, 2, projection='3d'))
# Plot a 3D wireframe (no colors)
ax1[1].plot_wireframe(X1_grid, X2_grid, Y_grid, rstride=5, cstride=5)
ax1[1].set_xlabel('$x_1$', fontsize=15)
ax1[1].set_ylabel('$x_2$', fontsize=15)
ax1[1].set_zlabel('$f(x_1,x_2)$', fontsize=15)
ax1[1].set_title("%s function" % function_name, fontsize=15)
#plt.tight_layout() # if we want to enlarge the figures, but sometimes this leads to label occlusion.
fig1.set_size_inches(15, 6)
plt.close(fig1)
```

In [7]:

```
fig1
```

Out[7]:

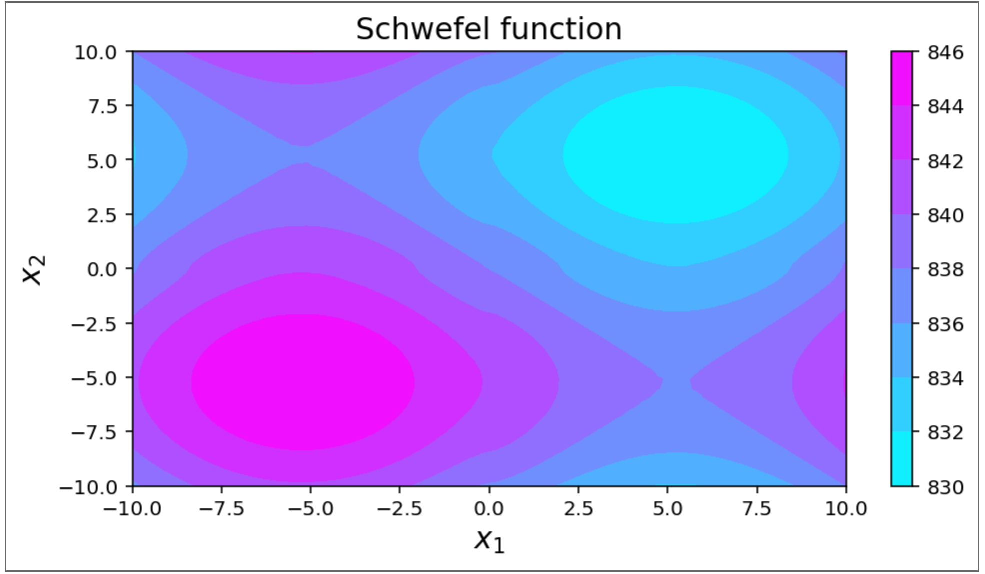


In the next figure, we show a contour plot of the same function.

In [8]:
```
                fig2, ax2 = plt.subplots()
cset = ax2.contourf(X1_grid, X2_grid, Y_grid, cmap=set_cm)
ax2.set_xlabel('$x_1$', fontsize=15)
ax2.set_ylabel('$x_2$', fontsize=15)
ax2.set_title("%s function" % function_name, fontsize=15)
fig2.colorbar(cset, ax=ax2)
```

Out[8]:

```
<matplotlib.colorbar.Colorbar at 0x7f87b1c4e3d0>
```



Exercise 1

Go to this **cell** and choose a different function to be learned. Run the notebook until here to see how the function looks like.

Note on the Pandas DataFrame included in the ”docs” folder

- The "docs" folder contains a file called "data_noiseless_schwefel_2D_regression.csv" with a Pandas DataFrame with data obtained from the Schwefel function in the domain $x \in [-10, 10]$ without considering noise.

So, let's load this DataFrame and use it for training a Gaussian Process.

In [9]:

```
import pandas as pd
df = pd.read_pickle("docs/data_noiseless_schwefel_2D_regression.pkl") # read pandas DataFrame
function_name = 'Schwefel' # The dataframe was created using the Schwefel function

print("Show the DataFrame in the docs folder:\n", df)
```

```
Show the DataFrame in the docs folder:
               x1    x2          y1
0    -10.000000 -10.0  837.552129
1     -9.591837 -10.0  838.185887
2     -9.183673 -10.0  838.777493
3     -8.775510 -10.0  839.323512
4     -8.367347 -10.0  839.820614
...         ...   ...         ...
2495   8.367347  10.0  836.110986
2496   8.775510  10.0  836.608088
2497   9.183673  10.0  837.154107
2498   9.591837  10.0  837.745713
2499  10.000000  10.0  838.379471

[2500 rows x 3 columns]
```

Although we introduced pandas in Lecture 2, you should keep exploring how it works...
For example, there are a few different ways to access the data of a DataFrame.
The notes below (not shown in the presentation), show you 3 different ways:

1. Direct way to select columns & rows by how they were labeled originaly

1. DataFrame.loc to select columns & rows by Name

1. DataFrame.iloc to select columns & rows by Index Positions (integer numbers)

In [12]:

```python
# Separate the features and targets into two different variables:
X_data = df.loc[:,['x1','x2']].values # note that we ask for all the values, not a subset of the DataFrame
y_data = df.loc[:,'y1'].values

# Recreate the grid of points that is necessary to plot surfaces (by reshaping)
X1_grid = np.reshape(X_data[:,0],( round( np.sqrt(len(X_data)) ), round( np.sqrt(len(X_data)) ) ))
X2_grid = np.reshape(X_data[:,1],( round( np.sqrt(len(X_data)) ), round( np.sqrt(len(X_data)) ) ))
Y_grid = np.reshape(y_data,( round( np.sqrt(len(X_data)) ), round( np.sqrt(len(X_data)) ) ))

print("Features loaded from the saved DataFrame:\n", X_data)
print("\nTarget loaded from the saved DataFrame:\n", y_data)
```

```
Features loaded from the saved DataFrame:
 [[-10.         -10.        ]
 [ -9.59183673 -10.        ]
 [ -9.18367347 -10.        ]
 ...
 [  9.18367347  10.        ]
 [  9.59183673  10.        ]
 [ 10.          10.        ]]

Target loaded from the saved DataFrame:
 [837.55212937 838.18588668 838.77749275 ... 837.15410725 837.74571332
 838.37947063]
```

Multidimensional regression with Gaussian Processes

Let's create a Gaussian Process Regression model for the multidimensional data we loaded from that Pandas dataframe.

As usual, we split the dataset into training and testing sets.

- For now, we split the data with the following ratio: 2% for training set, and 98% for testing set

Note: the commonly used ratio is 75% for training and 25% for testing, but we will use Gaussian Processes and they can learn from very little data.

In [13]:

```
                # Train/test split like we did before!
from sklearn.model_selection import train_test_split

testset_ratio = 0.98 # ratio of test set points from the dataset

X_train, X_test, y_train, y_test = train_test_split(X_data, y_data,
                                                    test_size=testset_ratio,random_state=seed)
#
```

Now we can train the GP model on the training data and test it on the testing data, just like we did in the previous lectures.

```
In [14]:
               from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.gaussian_process.kernels import RBF, Matern, ExpSineSquared, ConstantKernel

# Define the kernel function
kernel = ConstantKernel(1.0, (1e-4, 1e4)) * RBF(10, (1e-2, 1e2)) # This is the standard RBF kernel
#kernel = 1.0 * RBF(10, (1e-2, 1e2)) # Same kernel as above
                                    #(scikit-learn assumes constant
                                    # variance if you just write RBF
                                    # without the constant kernel or
                                    # without multiplying by 1.0)

# Other examples of kernels:
#kernel = ExpSineSquared(length_scale=3.0, periodicity=3.14,
#                        length_scale_bounds=(0.1, 10.0),
#                        periodicity_bounds=(0.1, 10)) * RBF(3.0, (1e-2, 1e2))
#kernel = Matern(length_scale=1.0, length_scale_bounds=(1e-2, 1e2),nu=1.5)

gp_model = GaussianProcessRegressor(kernel=kernel, alpha=1e-3, n_restarts_optimizer=20) # using a small alpha

# Fit to data using Maximum Likelihood Estimation of the parameters
gp_model.fit(X_train, y_train) # here I am not scaling the target variable; only the features.

# Make the prediction on the entire dataset (for plotting), both for mean and standard deviation
y_data_GPpred, sigma_data_GPpred = gp_model.predict(X_data, return_std=True)

# Predict for test set (for error metric)
y_test_GPpred, sigma_test_GPpred = gp_model.predict(X_test, return_std=True)
```

```
/home/mabessa/miniconda3/envs/3dasm/lib/python3.9/site-packages/sklearn/gaussian_process/kernels.py:4
30: ConvergenceWarning: The optimal value found for dimension 0 of parameter k1__constant_value is cl
ose to the specified upper bound 10000.0. Increasing the bound and calling fit again may find a bette
r value.
  warnings.warn(
```

IMPORTANT: You probably got a warning message...
This is not just a minor "detail". It has important implications!

- This is a direct consequence of us not **pre-processing** the data adequately.

Before learning how to pre-process the data, let's visualize our GPR model and compare it to the ground truth function.

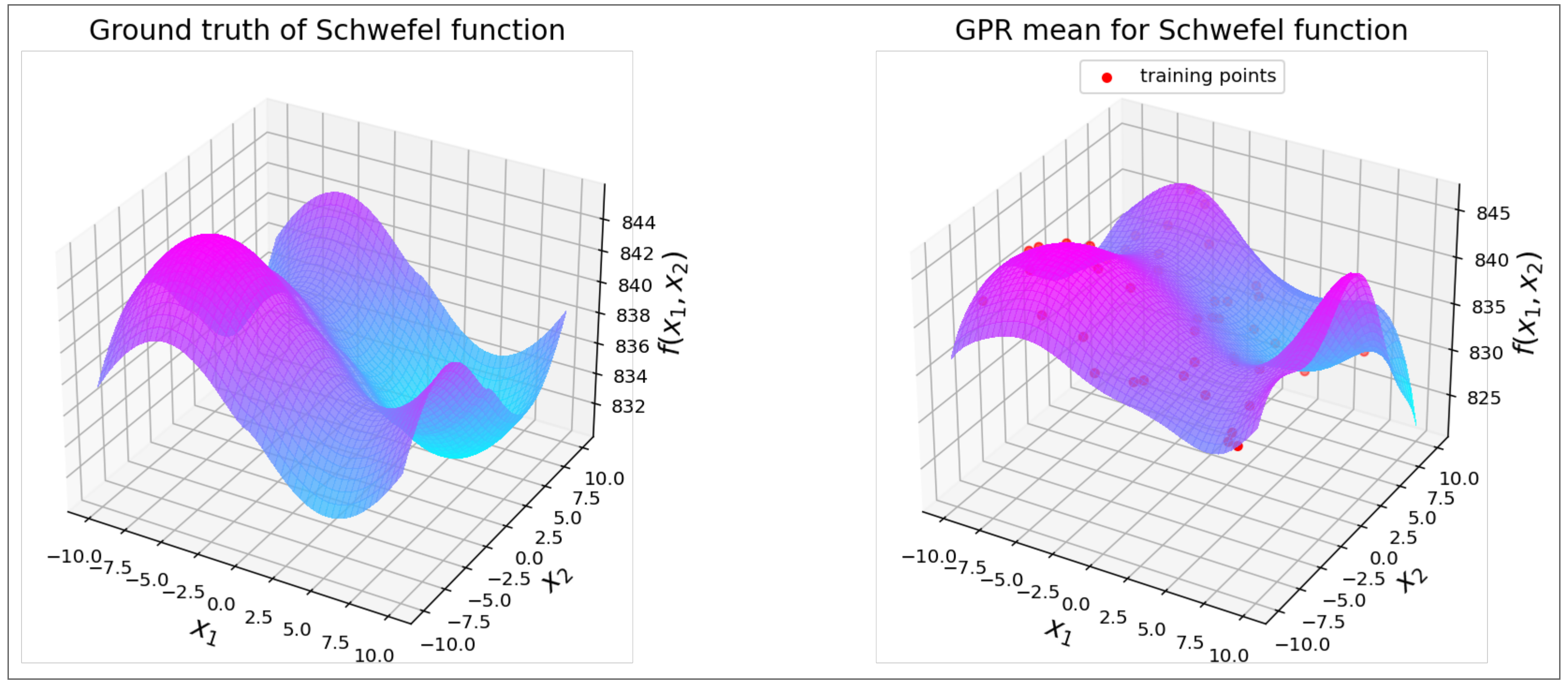
In [15]:

```
fig3_WRONG = plt.figure(figsize=plt.figaspect(0.5)); ax3_WRONG = []
# Subplot 1 (left): ground truth
ax3_WRONG.append(fig3_WRONG.add_subplot(1, 2, 1, projection='3d'))
surf = ax3_WRONG[0].plot_surface(X1_grid, X2_grid, Y_grid,
                                 cmap=set_cm, alpha=0.8, linewidth=0, antialiased=False)
ax3_WRONG[0].set_xlabel('$x_1$', fontsize=15)
ax3_WRONG[0].set_ylabel('$x_2$', fontsize=15)
ax3_WRONG[0].set_zlabel('$f(x_1,x_2)$', fontsize=15)
ax3_WRONG[0].set_title("Ground truth of %s function" % function_name, fontsize=15)

# Subplot 2 (right): GPR approximation
ax3_WRONG.append(fig3_WRONG.add_subplot(1, 2, 2, projection='3d'))
Y_grid_data_GPpred = np.reshape(y_data_GPpred,np.shape(X1_grid)) # convert targets into grid format for plotting
surf = ax3_WRONG[1].plot_surface(X1_grid, X2_grid, Y_grid_data_GPpred,
                                 cmap=set_cm, alpha=0.8, linewidth=0, antialiased=False)
ax3_WRONG[1].set_xlabel('$x_1$', fontsize=15)
ax3_WRONG[1].set_ylabel('$x_2$', fontsize=15)
ax3_WRONG[1].set_zlabel('$f(x_1,x_2)$', fontsize=15)
ax3_WRONG[1].set_title("GPR mean for %s function" % function_name, fontsize=15)
ax3_WRONG[1].scatter(X_train[:,0], X_train[:,1], y_train, marker='o', color='red', label="training points")
ax3_WRONG[1].legend(loc='upper center')
#plt.tight_layout()
fig3_WRONG.set_size_inches(15, 6)
plt.close(fig3_WRONG) # close figure to open it in next cell.
```

In [16]:

```
fig3_WRONG # show figure illustrating the issues of not scaling the data appropriately!
```

Out[16]:



This does not look great, especially at the boundary!

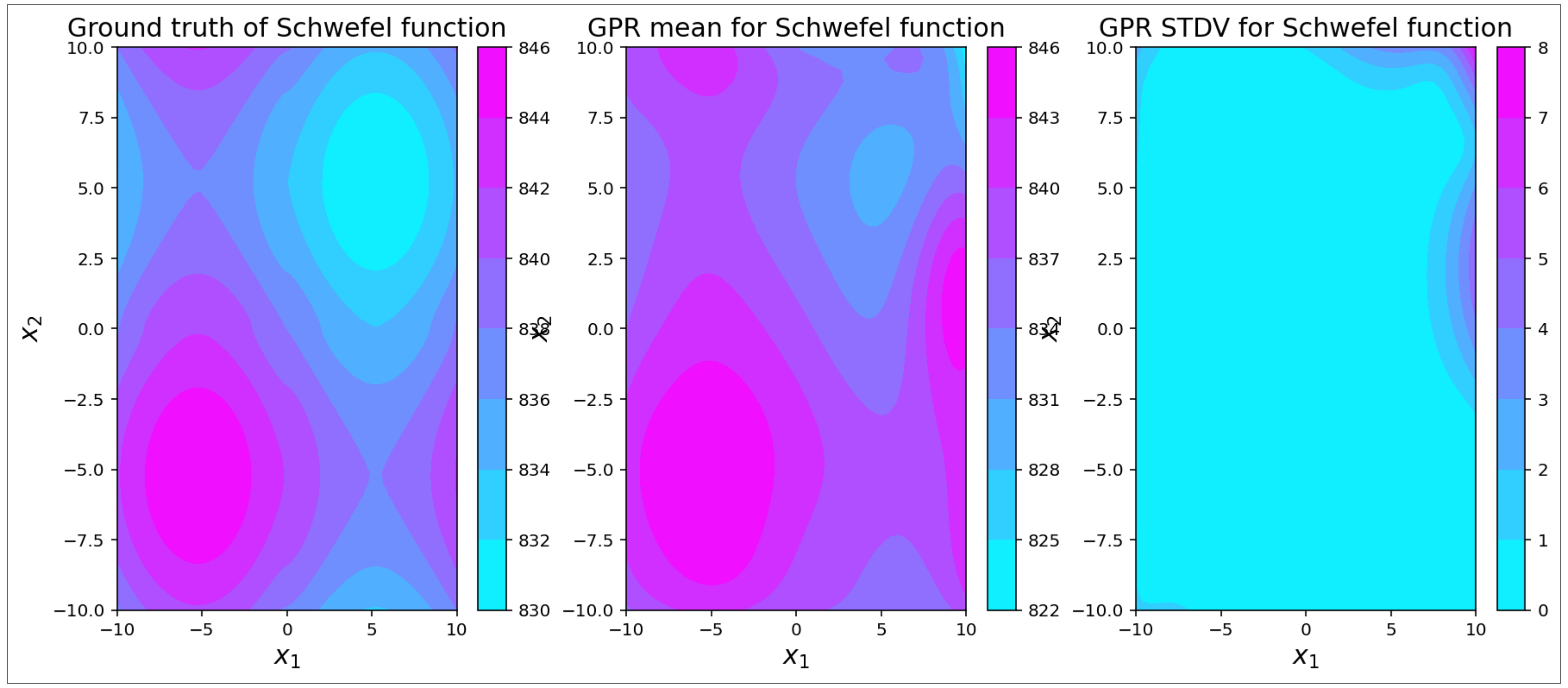
In [17]:

```
fig4_WRONG = plt.figure(figsize=plt.figaspect(0.5)); ax4_WRONG = []
# Create Contour plot:
ax4_WRONG.append(fig4_WRONG.add_subplot(1, 3, 1))
cset = ax4_WRONG[0].contourf(X1_grid, X2_grid, Y_grid, cmap=set_cm)
ax4_WRONG[0].set_xlabel('$x_1$', fontsize=15)
ax4_WRONG[0].set_ylabel('$x_2$', fontsize=15)
ax4_WRONG[0].set_title("Ground truth of %s function" % function_name, fontsize=15)
fig4_WRONG.colorbar(cset, ax=ax4_WRONG[0])

# Create Contour plot:
ax4_WRONG.append(fig4_WRONG.add_subplot(1, 3, 2))
cset = ax4_WRONG[1].contourf(X1_grid, X2_grid, Y_grid_data_GPpred, cmap=set_cm)
ax4_WRONG[1].set_xlabel('$x_1$', fontsize=15)
ax4_WRONG[1].set_ylabel('$x_2$', fontsize=15)
ax4_WRONG[1].set_title("GPR mean for %s function" % function_name, fontsize=15)
fig4_WRONG.colorbar(cset, ax=ax4_WRONG[1])

# Create Contour plot:
ax4_WRONG.append(fig4_WRONG.add_subplot(1, 3, 3))

SIGMA_grid_data_GPpred = np.reshape(sigma_data_GPpred,np.shape(X1_grid))

cset = ax4_WRONG[2].contourf(X1_grid, X2_grid, SIGMA_grid_data_GPpred, cmap=set_cm)
ax4_WRONG[2].set_xlabel('$x_1$', fontsize=15)
ax4_WRONG[2].set_ylabel('$x_2$', fontsize=15)
ax4_WRONG[2].set_title("GPR STDV for %s function" % function_name, fontsize=15)
fig4_WRONG.colorbar(cset, ax=ax4_WRONG[2])

#plt.tight_layout()
fig4_WRONG.set_size_inches(15, 6)
plt.close(fig4_WRONG) # close figure to open it in next cell.
```

In [18]:

```
fig4_WRONG # plot the same figure but using countour plots
```

Out[18]:



Now we see clearly that the uncertainty is significant at the boundary!

This happened because the output values are large and we did not **pre-process** the dataset appropriately!

- It is good practice to **scale the dataset**.

Scikit-learn has implemented different **preprocessing** strategies. **Check the documentation**.

The most common strategy is to standardize the dataset using the StandardScaler:

- StandardScaler: each feature is transformed to have zero-mean and unit standard deviation. In other words, the data becomes normally distributed (it is transformed). **Standardization of datasets is a common requirement for many machine learning models**.

In [19]:

```
from sklearn.preprocessing import StandardScaler
# Scaling inputs with a Standard Scaler:
scaler_x = StandardScaler()
scaler_x.fit(X_train) # fit the scaler to the input data
#
X_train_scaled=scaler_x.transform(X_train) # scale the input training data
X_test_scaled=scaler_x.transform(X_test)  # scale the input testing data
X_data_scaled=scaler_x.transform(X_data) # scale the input data (the whole dataset)

idx = 0 # Choose the point you want to check the value (we could also plot the entire dataset.)
print("X_data[%i] is =" % idx, X_data[idx])
print("X_data_scaled[%i] is =" % idx, X_data_scaled[idx])
```

```
X_data[0] is = [-10. -10.]
X_data_scaled[0] is = [-1.79658824 -1.50411611]
```

In [20]:

```
               # Now scaling outputs also using StandardScaler:
y_train = y_train.reshape(-1, 1) # recall that we need to convert the 1D array vector into a 2D array vector
y_test = y_test.reshape(-1, 1)   # recall that we need to convert the 1D array vector into a 2D array vector
y_data = y_data.reshape(-1, 1)   # recall that we need to convert the 1D array vector into a 2D array vector
#
scaler_y = StandardScaler()
scaler_y.fit(y_train) # fit the scaler to the output data
#
y_train_scaled=scaler_y.transform(y_train) # scale the output training data
y_test_scaled=scaler_y.transform(y_test)  # scale the output testing data
y_data_scaled=scaler_y.transform(y_data) # scale the output data (the whole dataset)

idx = 0 # Choose the point you want to check the value (we could also plot the entire dataset.)
print("y_data[%i] is =" % idx, y_data[idx])
print("y_data_scaled[%i] is =" % idx, y_data_scaled[idx])
```

```
y_data[0] is = [837.55212937]
y_data_scaled[0] is = [-0.10898576]
```

In [21]:

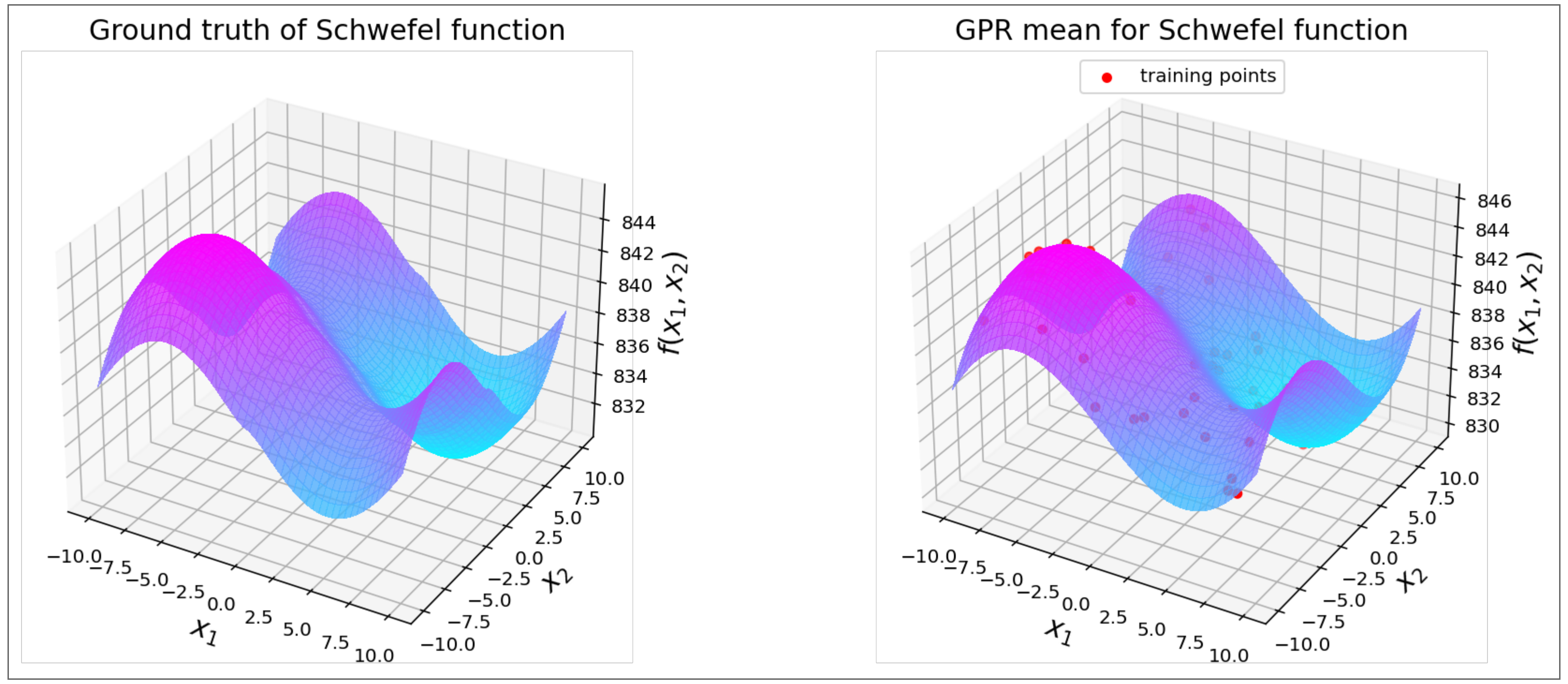
```
               # Create the GP model again (same kernel, same parameters and same optimization process!)
kernel = ConstantKernel(1.0, (1e-4, 1e4)) * RBF(10, (1e-2, 1e2)) # This is the standard RBF kernel
gp_model = GaussianProcessRegressor(kernel=kernel, alpha=1e-3, n_restarts_optimizer=20) # using a small alpha

# The ONLY difference is that we will train the model on SCALED inputs and outputs
gp_model.fit(X_train_scaled, y_train_scaled) # here I am not scaling the target variable; only the features.

# Make the prediction on the entire dataset (for plotting), both for mean and standard deviation
y_data_GPpred_scaled, sigma_data_GPpred_scaled = gp_model.predict(X_data_scaled, return_std=True)
# IMPORTANT: we need to scale the predictions for the mean of the output back to the original scale!
y_data_GPpred = scaler_y.inverse_transform(y_data_GPpred_scaled) # mean
# IMPORTANT: we need to do the same for the standard deviation of the output, but in this case we have to subtract
#             by the mean of the training data because the standard deviation is not transformed by the
#             StandardScaler in the same way as the mean! In the Homework you will derive this...
sigma_data_GPpred = scaler_y.inverse_transform(sigma_data_GPpred_scaled.reshape(-1, 1)) - np.mean(y_train)
my_sigma_data_GPpred = sigma_data_GPpred_scaled.reshape(-1, 1)*np.std(y_train)

# Predict for test set (for error metric)
y_test_GPpred_scaled, sigma_test_GPpred_scaled = gp_model.predict(X_test_scaled, return_std=True)
# Scale mean and std back, as above.
y_test_GPpred = scaler_y.inverse_transform(y_test_GPpred_scaled)
sigma_test_GPpred = scaler_y.inverse_transform(sigma_test_GPpred_scaled.reshape(-1, 1)) - np.mean(y_train)
```

In [22]:

```
fig3 = plt.figure(figsize=plt.figaspect(0.5)); ax3 = []
# Subplot 1 (left): ground truth
ax3.append(fig3.add_subplot(1, 2, 1, projection='3d'))
surf = ax3[0].plot_surface(X1_grid, X2_grid, Y_grid,
                           cmap=set_cm, alpha=0.8, linewidth=0, antialiased=False)
ax3[0].set_xlabel('$x_1$', fontsize=15)
ax3[0].set_ylabel('$x_2$', fontsize=15)
ax3[0].set_zlabel('$f(x_1,x_2)$', fontsize=15)
ax3[0].set_title("Ground truth of %s function" % function_name, fontsize=15)

# Subplot 2 (right): GPR approximation
ax3.append(fig3.add_subplot(1, 2, 2, projection='3d'))
Y_grid_data_GPpred = np.reshape(y_data_GPpred,np.shape(X1_grid)) # convert targets into grid format for plotting
surf = ax3[1].plot_surface(X1_grid, X2_grid, Y_grid_data_GPpred,
                                       cmap=set_cm, alpha=0.8, linewidth=0, antialiased=False)
ax3[1].set_xlabel('$x_1$', fontsize=15)
ax3[1].set_ylabel('$x_2$', fontsize=15)
ax3[1].set_zlabel('$f(x_1,x_2)$', fontsize=15)
ax3[1].set_title("GPR mean for %s function" % function_name, fontsize=15)
ax3[1].scatter(X_train[:,0], X_train[:,1], y_train, marker='o', color='red', label="training points")
ax3[1].legend(loc='upper center')
#plt.tight_layout()
fig3.set_size_inches(15, 6)
plt.close(fig3) # close figure to open it in next cell.
```

In [23]:
```
fig3
```

Out[23]:



We can also plot this information via contour plots, and show not only the mean predicted by GPR but also the standard deviation.
Note: you can also overlay the 95% confidence intervals in a surface plot, but usually it becomes really cluttered (so we didn't do it).

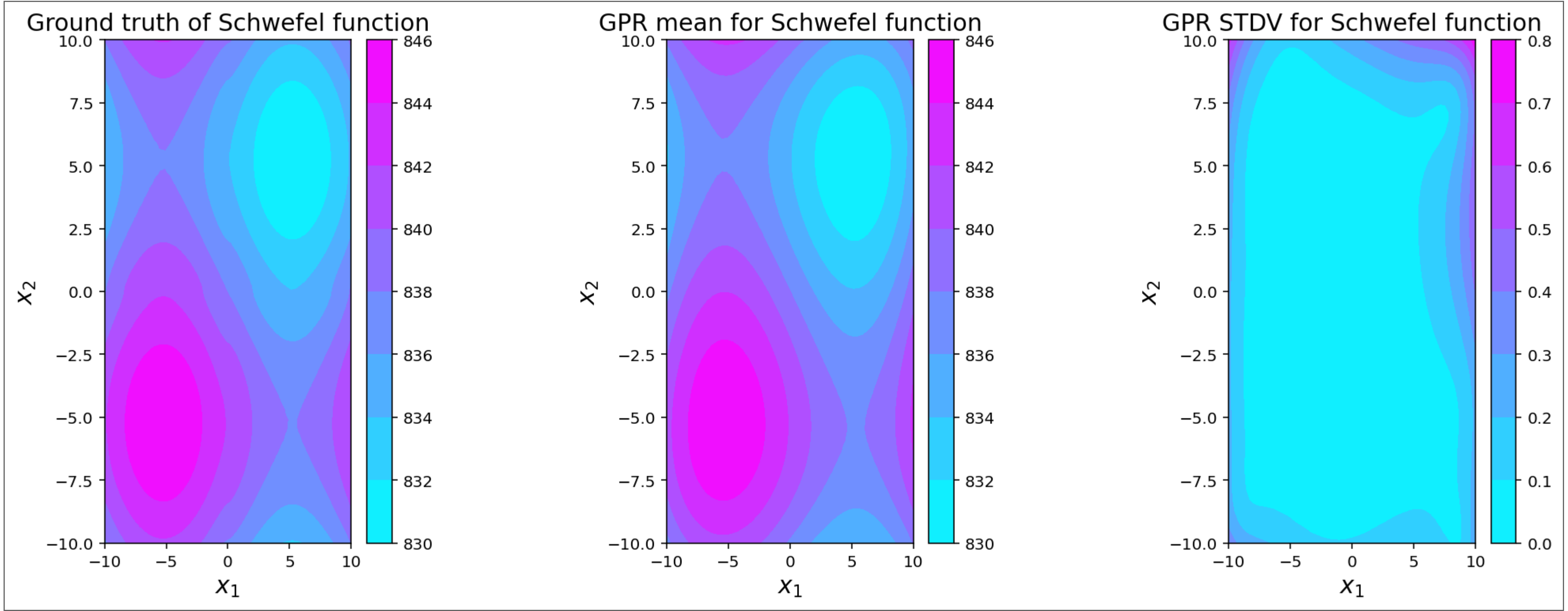
In [24]:

```
fig4 = plt.figure(figsize=plt.figaspect(0.5)); ax4 = []
# Create Contour plot:
ax4.append(fig4.add_subplot(1, 3, 1))
cset = ax4[0].contourf(X1_grid, X2_grid, Y_grid, cmap=set_cm)
ax4[0].set_xlabel('$x_1$', fontsize=15)
ax4[0].set_ylabel('$x_2$', fontsize=15)
ax4[0].set_title("Ground truth of %s function" % function_name, fontsize=15)
fig4.colorbar(cset, ax=ax4[0])

# Create Contour plot:
ax4.append(fig4.add_subplot(1, 3, 2))
cset = ax4[1].contourf(X1_grid, X2_grid, Y_grid_data_GPpred, cmap=set_cm)
ax4[1].set_xlabel('$x_1$', fontsize=15)
ax4[1].set_ylabel('$x_2$', fontsize=15)
ax4[1].set_title("GPR mean for %s function" % function_name, fontsize=15)
fig4.colorbar(cset, ax=ax4[1])

# Create Contour plot:
ax4.append(fig4.add_subplot(1, 3, 3))

SIGMA_grid_data_GPpred = np.reshape(sigma_data_GPpred,np.shape(X1_grid))

cset = ax4[2].contourf(X1_grid, X2_grid, SIGMA_grid_data_GPpred, cmap=set_cm)
ax4[2].set_xlabel('$x_1$', fontsize=15)
ax4[2].set_ylabel('$x_2$', fontsize=15)
ax4[2].set_title("GPR STDV for %s function" % function_name, fontsize=15)
fig4.colorbar(cset, ax=ax4[2])

plt.tight_layout()
fig4.set_size_inches(15, 6)
plt.close(fig4) # close figure to open it in next cell.
```

In [25]:

```
fig4 # show same results but using contour plots.
```

Out[25]:



It's quite remarkable that GPR predicts this function almost perfectly while using only 50 points! This is equivalent to using just $\sqrt{50} \approx 7$ points per dimension!

Let's now compute the R^2 error metric for the GPR approximation.

```
In [26]:
from sklearn.metrics import r2_score # Import error metrics

# Compute MSE and R2 for the GP model
gp_r2_value = r2_score(y_test, y_test_GPpred)

print('R2 score for GPR = ', gp_r2_value)
```

```
R2 score for GPR =  0.995660118433014
```

An important issue in Gaussian processes: lack of scalability

- Redo the GP regression of the Schwefel function but now using 75% of the data for training.

The notes below show the complete code. You probably noticed it took a while to train the GP...
Unfortunately, it is not uncommon for a Machine Learning problem to have millions of data points... But GPs cannot deal with this because the computational cost scales cubically with the number of datapoints: $\mathcal{O}(n^3)$.
The reason for this issue comes from the need to invert the covariance matrix, which is a $n \times n$ matrix where n is the number of training points.

- **To keep in mind**: Gaussian processes are very powerful for small datasets, but they do not scale well for large datasets.

HOMEWORK

Redo the Gaussian Processes Regression (GPR) of the Schwefel function for these conditions:

1. Training for data where you only scale the **inputs** using the StandardScaler. You get the same warning message (convergence difficulties to find the parameters of the kernel). **Why?**
2. Training for data where you only scale the **outputs** before training. This time your predictions are as good as the ones obtained when scaling both the inputs and the outputs. **Why?**
3. Create your own StandardScaler class achieving the same as scikit-learn's StandardScaler, but where the inverse_transform for the standard deviation does not need to be corrected like we did in the Lecture.

You will explore these and other things in the Midterm Project...

Have fun!