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

Lecture 12

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**.

- We will continue to focus on regression via supervised learning with the same methods but now considering multiple dimensions:
    - **Gaussin Processes** using **scikit-learn**
    - **Artificial Neural Networks** (ANNs) using **keras** and **tensorflow**
- We will also do a quick example on classification via supervised learning with:
    - **Decision trees** using **scikit-learn**
    - **Support vector machines** using **scikit-learn**

Remember: there are many (many!) more ML models out there. We are covering a few simple ones... The hope is that you feel comfortable with applying and deriving different models!

1. Multidimensional regression

Similarly to what we did for one-dimension, now we are going to show how to do multidimensional regression with supervised learning.
As you will see, it's pretty much the same thing...

- We will use less obvious functions to generate our datasets for subsequent learning.
- There is a nice website listing many different benchmark functions used in optimization: **https://www.sfu.ca/~ssurjano/optimization.html**

Let's try to learn some of them...

In [3]:

```
import pandas as pd
# read pandas DataFrame from "docs" folder of the past Lecture
df = pd.read_pickle("../Lecture11/docs/data_noiseless_schwefel_2D_regression.pkl")
function_name = 'Schwefel' # The dataframe was created using the Schwefel function

print("Show that this is the same DataFrame as in Lecture 11:\n", df)
```

```
Show that this is the same DataFrame as in Lecture 11:
              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]
```

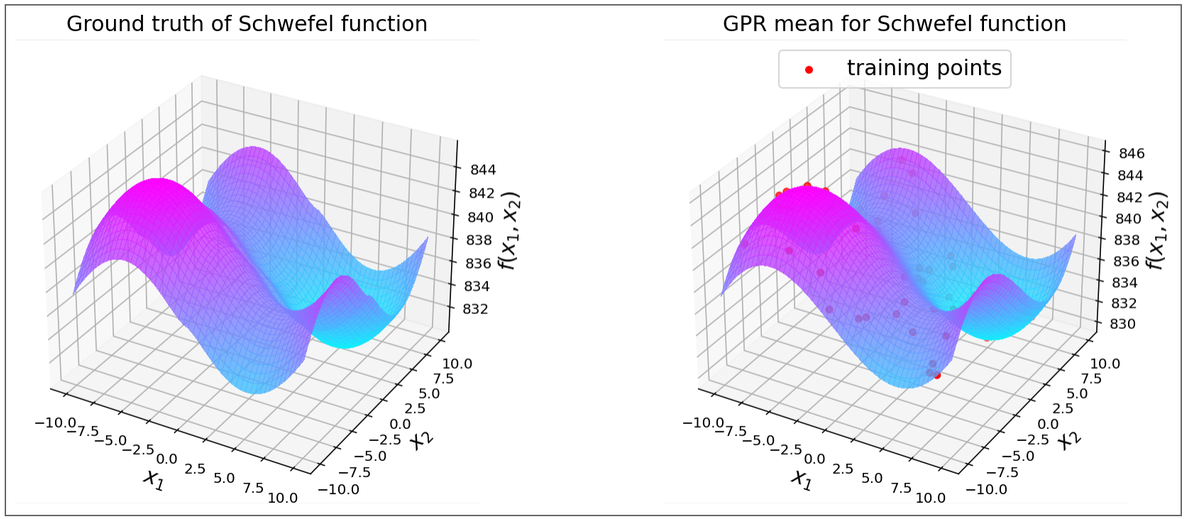
Considering the same dataset as in the previous lecture, we then:

- Split the data into the same training and testing sets as in Lecture 11.
- Scale the data using the same StandardScaler from scikit-learn.

(Since this is the same code as used in the previous lecture, it is only shown below as notes)

In [5]:
```
fig3 # show GP prediction for noiseless dataset of Schwefel function obtained in previous Lecture.
```

Out[5]:



This is the prediction we obtained using GPs last Lecture.

Multidimensional regression with Artificial Neural Networks

Today we will use ANNs to make predictions on the same data considering the same training points.
As we will see, ANNs behave in a somewhat opposite way: training for data scarce problems can be challenging, but ANNs are very scalable for "big data" problems.
Below is a code including an option to perform grid search (or not). The code is basically the same as what we saw in Lecture 10 but now with 2D features.

In [6]:

```
          # As expected, the code for multidimensional ANNs is basically the same as the 1D case.
from tensorflow import keras
from tensorflow.keras.optimizers import Adam # import the optimizer you want to use to calculate the parameters
from keras.models import Sequential # to create a feedforward neural network
from keras.layers.core import Dense # to create a feedforward neural network with dense layers
from keras.wrappers.scikit_learn import KerasRegressor # a new version will use scikeras
from keras.callbacks import EarlyStopping # a strategy for complexity control
from sklearn.model_selection import GridSearchCV # simple (brute force) approach to find better hyperparameters.
#
# Function to create the ANN model
def create_ANN(input_dimensions=1,neurons1=10,neurons2=10,neurons3=10,neurons4=10,activation='relu',optimizer='adam'):
    # create model
    model = Sequential()
    model.add(Dense(neurons1, input_dim=input_dimensions, activation=activation)) # first hidden layer
    model.add(Dense(neurons2, activation=activation)) # second hidden layer
    #model.add(Dense(neurons3, activation=activation)) # thrid hidden layer
    #model.add(Dense(neurons4, activation=activation)) # fourth hidden layer, etc.
    model.add(Dense(1))
    model.compile(loss='mse', optimizer=optimizer)
    return model
#
```

In [7]:

```
            # If you don't want to use Early Stopping, then comment the command below.
early_stopping = EarlyStopping(monitor='val_loss', # use validation error to check if training should stop early
                               min_delta=0.0, # minimum change in the monitored quantity to qualify as an
                                              # improvement (if 0, then even small improvements count)
                               patience=30, # be patient for a few epochs to verify evolution of loss
                               mode='min') # in min mode, training stops when val_loss stops decreasing
# Summary of this early_stopping criterion:
# stop training when there is no improvement in the loss for 30 consecutive epochs.

neurons1=200 # number of neurons in hidden layer 1
neurons2=10  # number of neurons in hidden layer 2
epochs = 1000 # number of epochs
batch_size = len(X_train) # number of samples in each batch
optimizer = 'adam' # optimizer
ANN_model = KerasRegressor(build_fn=create_ANN, # the function where we define our ANN
                           input_dimensions=np.shape(X_train)[1], # the input dimensions (here it is 2)
                           neurons1=neurons1, # number of neurons in the first hidden layer
                           neurons2=neurons2, # number of neurons in the second hidden layer
                           batch_size=batch_size, # define the number of samples in each batch
                           epochs=epochs, # number of epochs
                           optimizer=optimizer, # optimizer
                           callbacks=[early_stopping], # include early stopping
                           validation_data=(X_test_scaled, y_test_scaled)) # validation data to compute loss
# Train the model:
history = ANN_model.fit(X_train_scaled, y_train_scaled)
```

```
/tmp/ipykernel_98709/2277623882.py:15: DeprecationWarning: KerasRegressor is deprecated, use Sci-Keras (https://github.com/adriangb/scikeras) instead.
  ANN_model = KerasRegressor(build_fn=create_ANN, # the function where we define our ANN
2022-03-19 21:27:18.732392: I tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:939] successful NUMA node read from SysFS had negative value (-1), but there must be at least one NUMA node, so returning NUMA node zero
2022-03-19 21:27:18.778307: I tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:939] successful NUMA node read from SysFS had negative value (-1), but there must be at least one NUMA node, so returning NUMA node zero
2022-03-19 21:27:18.778914: I tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:939] successful NUMA node read from SysFS had negative value (-1), but there must be at least one NUMA node, so returning NUMA node zero
2022-03-19 21:27:18.779394: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1900] Ignoring visible gpu device (device: 0, name: GeForce GTX 560M, pci bus id: 0000:01:00.0, compute capability: 2.1) with Cuda compute capability 2.1. The minimum required Cuda capability is 3.5.
```

```
2022-03-19 21:27:18.779932: I tensorflow/core/platform/cpu_feature_guard.cc:151] This TensorFlow bina
ry is optimized with oneAPI Deep Neural Network Library (oneDNN) to use the following CPU instruction
s in performance-critical operations:  SSE4.1 SSE4.2 AVX
To enable them in other operations, rebuild TensorFlow with the appropriate compiler flags.
```

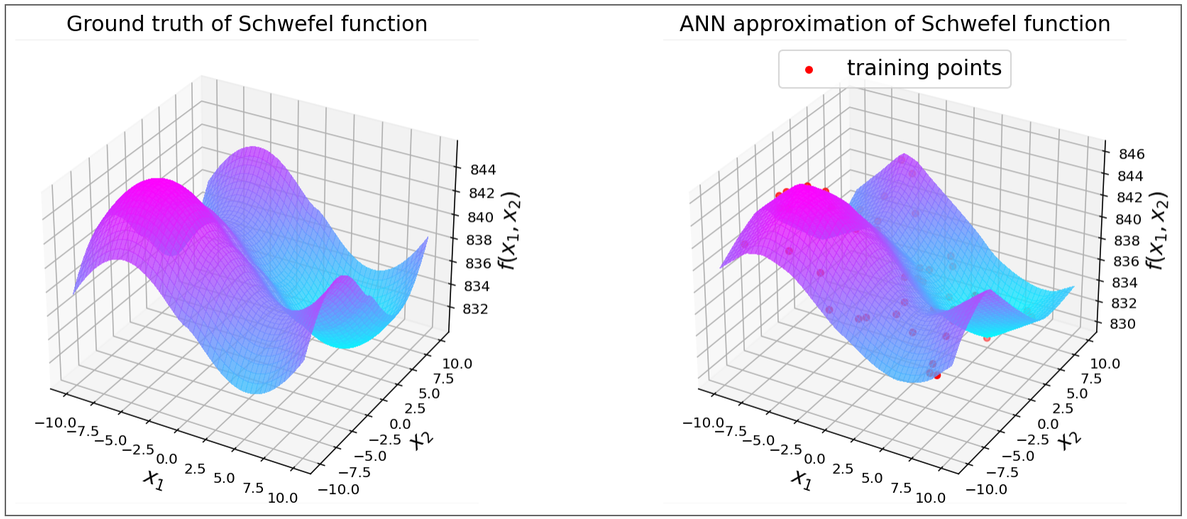
```
Epoch 1/1000
1/1 [==============================] - 1s 518ms/step - loss: 1.0381 - val_loss: 0.9904
Epoch 2/1000
1/1 [==============================] - 0s 67ms/step - loss: 0.9972 - val_loss: 0.9479
Epoch 3/1000
1/1 [==============================] - 0s 67ms/step - loss: 0.9562 - val_loss: 0.9096
Epoch 4/1000
1/1 [==============================] - 0s 59ms/step - loss: 0.9202 - val_loss: 0.8760
Epoch 5/1000
1/1 [==============================] - 0s 62ms/step - loss: 0.8874 - val_loss: 0.8499
Epoch 6/1000
1/1 [==============================] - 0s 53ms/step - loss: 0.8619 - val_loss: 0.8254
Epoch 7/1000
1/1 [==============================] - 0s 69ms/step - loss: 0.8375 - val_loss: 0.8016
Epoch 8/1000
1/1 [==============================] - 0s 57ms/step - loss: 0.8131 - val_loss: 0.7781
Epoch 9/1000
1/1 [==============================] - 0s 62ms/step - loss: 0.7890 - val_loss: 0.7549
Epoch 10/1000
1/1 [==============================] - 0s 69ms/step - loss: 0.7653 - val_loss: 0.7324
Epoch 11/1000
1/1 [==============================] - 0s 59ms/step - loss: 0.7419 - val_loss: 0.7101
Epoch 12/1000
1/1 [==============================] - 0s 57ms/step - loss: 0.7187 - val_loss: 0.6881
Epoch 13/1000
1/1 [==============================] - 0s 59ms/step - loss: 0.6957 - val_loss: 0.6667
Epoch 14/1000
1/1 [==============================] - 0s 59ms/step - loss: 0.6731 - val_loss: 0.6460
```

```
Epoch 995/1000
1/1 [==============================] - 0s 60ms/step - loss: 4.6734e-04 - val_loss: 0.0213
Epoch 996/1000
1/1 [==============================] - 0s 61ms/step - loss: 4.6479e-04 - val_loss: 0.0213
Epoch 997/1000
1/1 [==============================] - 0s 66ms/step - loss: 4.6224e-04 - val_loss: 0.0213
Epoch 998/1000
1/1 [==============================] - 0s 55ms/step - loss: 4.5968e-04 - val_loss: 0.0213
Epoch 999/1000
1/1 [==============================] - 0s 66ms/step - loss: 4.5718e-04 - val_loss: 0.0212
Epoch 1000/1000
1/1 [==============================] - 0s 62ms/step - loss: 4.5465e-04 - val_loss: 0.0212
```

Let's plot the result for ANN as we did for GPR.

In [9]:
```
fig5 # surface plots for ANN
```

Out[9]:



The approximation does not seem too bad (and it can get a better!).

- However, it can be challenging to make ANNs as good as GPR for a small number of training points (and there is a theoretical result that explains why!).

Finally, let's calculate the R2 score for the ANN.

In [10]:
```
from sklearn.metrics import r2_score # Import error metrics
# Compute R2 for the ANN model
y_test_ANNpred_scaled = history.model.predict(X_test_scaled)
y_test_ANNpred = scaler_y.inverse_transform(y_test_ANNpred_scaled) # scale predictions back to original scale

ANN_r2_value = r2_score(y_test, y_test_ANNpred)

print('R2 score for ANN = ', ANN_r2_value)
```

```
R2 score for ANN =  0.9784113541511101
```

In the midterm you will explore training of GPR and ANN using a different number of training points and different hyperparameters.
You will also consider
For now, let's just see what happens when we consider noise in our dataset.

1.3. GPR and ANN with noisy datasets

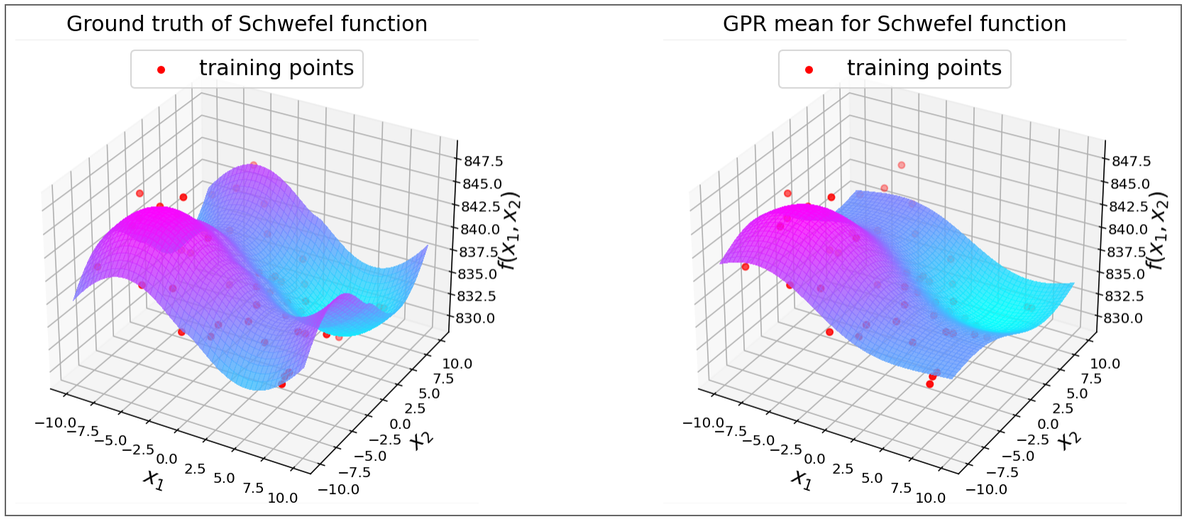
Since this is so similar to what we did in the 1D case, I will not go through the next cells in detail.
You are encouraged to look into the next cells by yourself (not shown in presentation).

In [12]:

```
                # Create GP model as we did in the previous Lecture.
kernel = ConstantKernel(1.0, (1e-4, 1e4)) * RBF(10, (1e-2, 1e2)) # This is the standard RBF kernel
gp_noisy_model = GaussianProcessRegressor(kernel=kernel, alpha=noise_train**2, # note the ALPHA here!
                                          n_restarts_optimizer=20)

# The ONLY difference is that we will train the model on SCALED inputs and outputs
gp_noisy_model.fit(X_train_scaled, y_noisy_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_noisy_data_GPpred_scaled, sigma_noisy_data_GPpred_scaled = gp_noisy_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_noisy_data_GPpred = scaler_y.inverse_transform(y_noisy_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_noisy_data_GPpred = scaler_y.inverse_transform(sigma_noisy_data_GPpred_scaled.reshape(-1, 1)) - np.mean(y_noisy_train)
my_sigma_data_GPpred = sigma_data_GPpred_scaled.reshape(-1, 1)*np.std(y_train)

# Predict for test set (for error metric)
y_noisy_test_GPpred_scaled, sigma_noisy_test_GPpred_scaled = gp_noisy_model.predict(X_test_scaled, return_std=True)
# Scale mean and std back, as above.
y_noisy_test_GPpred = scaler_y.inverse_transform(y_noisy_test_GPpred_scaled)
sigma_noisy_test_GPpred = scaler_y.inverse_transform(sigma_noisy_test_GPpred_scaled.reshape(-1, 1)) - np.mean(y_noisy_train)
```

In [14]:
```
fig6 # plot surfaces
```

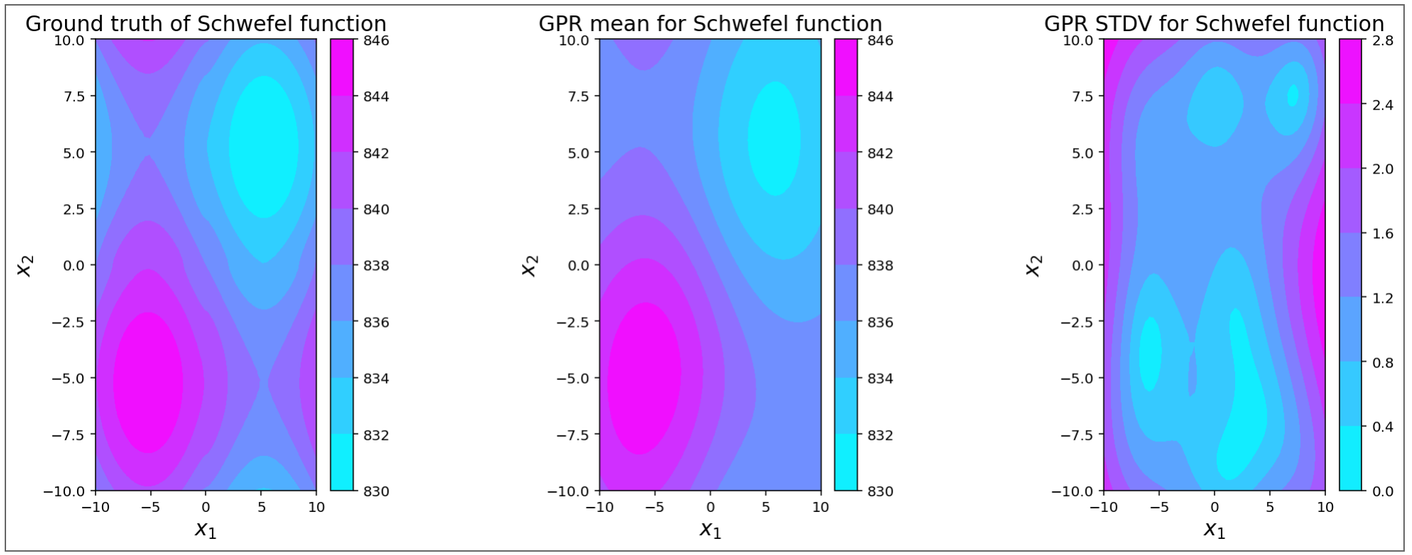
Out[14]:



In [16]:

```
fig7 # plot contours
```

Out[16]:



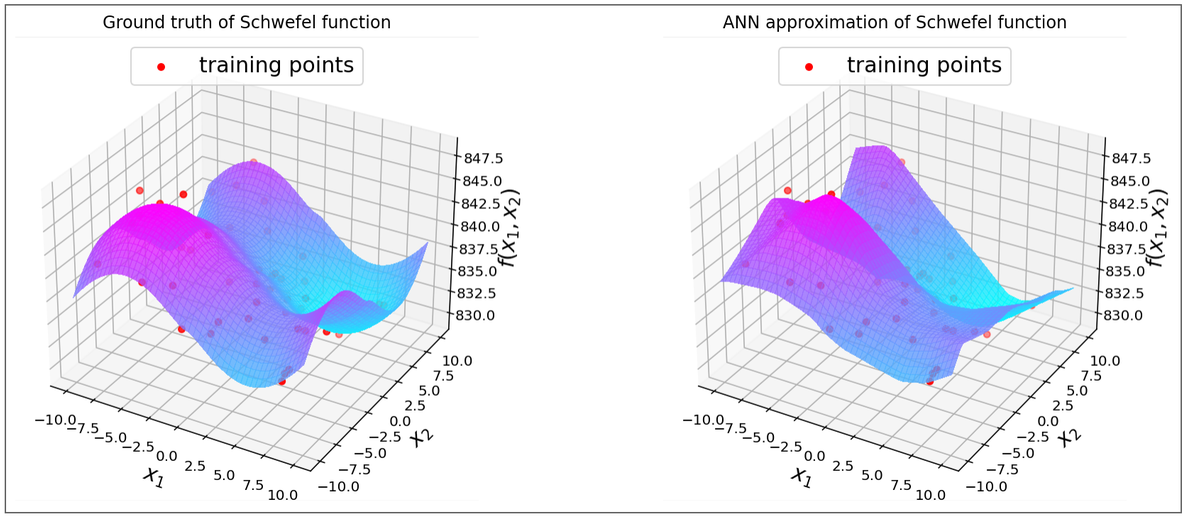
In [17]:

```
from sklearn.metrics import r2_score # Import error metrics
# Error metric
# Compute R2 for the GP model
gp_noisy_r2_value = r2_score(y_test, y_noisy_test_GPpred)

print('R2 score for GPR in noisy dataset =', gp_noisy_r2_value)
```

```
R2 score for GPR in noisy dataset = 0.8829059855273038
```

In [20]:
```
fig8 # plot surfaces for ANN
```

Out[20]:



In [21]:

```
# Compute R2 for the ANN model
y_noisy_test_ANNpred = history.model.predict(X_test_scaled)
ANN_noisy_r2_value = r2_score(y_test, scaler_y.inverse_transform(y_noisy_test_ANNpred))
# note the transformation of the outputs back to the original scale

print('R2 score for ANN = ', ANN_noisy_r2_value)
```

```
R2 score for ANN =  0.823715494682924
```

Don’t forget to ”play” with this notebook

- **Try learning different functions**: go to the beginning of the notebook and change the benchmark function to be learned (e.g. Levy function)
- Use different noise levels
- Try different kernels for the GPR
- Do a Grid Search to find better parameters for the ANN
- Investigate the role of "early stopping" in the ANN

There are so many things that you can do to go deeper in the topic! Have some fun with it!
Important Note: Despite this notebook being dedicated to 3D surfaces (2 features and 1 target), the algorithms are general. In principle, they can perform regression for any dimensionality of features and targets (although for GPR if you want more than one output you have to consider modifications to the method). The codes do not change significantly, but visualizing data can be a challenge. The simplest solution is to project the space back to a 3D space. You will do that in your projects.

2. Multidimensional classification

Here's some good news:

- If you understand regression with supervised learning... Then, performing classification with supervised learning is very similar!

But first: what is classification?

- Classification problems occur when the target (output) is **discrete**, instead of being a continuous variable like we did in regression problems.

- A simple classification problem that is very common to consider when first learning about this topic is the **Iris dataset** created by UCI researchers in 1936.
- I downloaded the .CSV file from **Kaggle** and it is included in the docs folder of this GitHub repository.

Therefore, we can load that .CSV file into a Pandas DataFrame!

In [22]:

```
# Load the Iris dataset.
iris_data = pd.read_csv('docs/Iris.csv')

print(iris_data)
```

```
      Id  SepalLengthCm  SepalWidthCm  PetalLengthCm  PetalWidthCm  \
0      1            5.1           3.5            1.4           0.2
1      2            4.9           3.0            1.4           0.2
2      3            4.7           3.2            1.3           0.2
3      4            4.6           3.1            1.5           0.2
4      5            5.0           3.6            1.4           0.2
..   ...            ...           ...            ...           ...
145  146            6.7           3.0            5.2           2.3
146  147            6.3           2.5            5.0           1.9
147  148            6.5           3.0            5.2           2.0
148  149            6.2           3.4            5.4           2.3
149  150            5.9           3.0            5.1           1.8

            Species
0       Iris-setosa
1       Iris-setosa
2       Iris-setosa
3       Iris-setosa
4       Iris-setosa
..              ...
145  Iris-virginica
146  Iris-virginica
147  Iris-virginica
148  Iris-virginica
149  Iris-virginica
```

```
[150 rows x 6 columns]
```

A SHORT NOTE

In fact, the Iris dataset is **so common**, that scikit-learn already has it saved and it can be loaded directly from the scikit-learn code, as shown below.

When you start a new problem, the first step is:

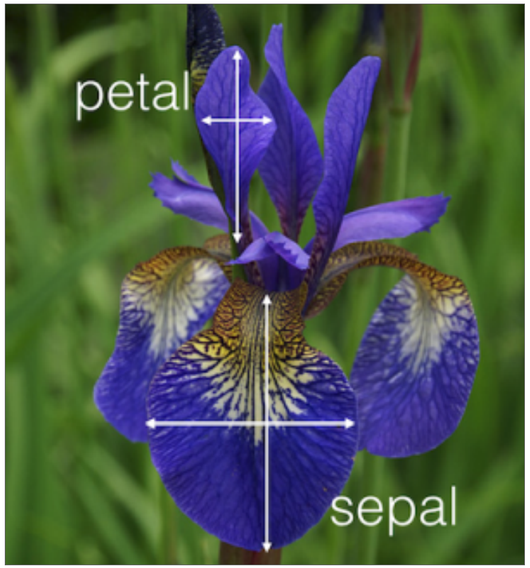
- Get to know the dataset.

What are the features? And the targets? What kind of data preprocessing should we do? What do we know *before* using machine learning tools to analyze the dataset?

Here's a nice image to understand the dataset.
The dataset is quite simple! It has:

- 4 Features (inputs): sepal length, sepal width, petal length, petal width
- 1 Target (output): iris species (3 different species with 50 samples each)

So, the dataset has 150 samples of 3 different species of iris flower (50 samples per species) and each flower has 4 features that characterize it: the length and width of the petal and the sepal (see figure on the right).
Our task: Learn from part of the dataset how to classify a new iris flower into one of the 3 species!
Note: This example is straight out of Scikit-learn (but simplified)



In [24]:

```
                       # Let's only use the first two features (ignoring the other 2).
X_2d_iris_data = iris_data.loc[:, ['SepalLengthCm','SepalWidthCm'] ].values # just use 2 inputs (features)
y_string_iris_data = iris_data.loc[:, 'Species'].values

print('y_string_iris_data = ',y_string_iris_data)
```

```
y_string_iris_data =  ['Iris-setosa' 'Iris-setosa' 'Iris-setosa' 'Iris-setosa' 'Iris-setosa'
 'Iris-setosa' 'Iris-setosa' 'Iris-setosa' 'Iris-setosa' 'Iris-setosa'
 'Iris-setosa' 'Iris-setosa' 'Iris-setosa' 'Iris-setosa' 'Iris-setosa'
 'Iris-setosa' 'Iris-setosa' 'Iris-setosa' 'Iris-setosa' 'Iris-setosa'
 'Iris-setosa' 'Iris-setosa' 'Iris-setosa' 'Iris-setosa' 'Iris-setosa'
 'Iris-setosa' 'Iris-setosa' 'Iris-setosa' 'Iris-setosa' 'Iris-setosa'
 'Iris-setosa' 'Iris-setosa' 'Iris-setosa' 'Iris-setosa' 'Iris-setosa'
 'Iris-setosa' 'Iris-setosa' 'Iris-setosa' 'Iris-setosa' 'Iris-setosa'
 'Iris-setosa' 'Iris-setosa' 'Iris-setosa' 'Iris-setosa' 'Iris-setosa'
 'Iris-setosa' 'Iris-setosa' 'Iris-setosa' 'Iris-setosa' 'Iris-setosa'
 'Iris-versicolor' 'Iris-versicolor' 'Iris-versicolor' 'Iris-versicolor'
 'Iris-versicolor' 'Iris-versicolor' 'Iris-versicolor' 'Iris-versicolor'
 'Iris-versicolor' 'Iris-versicolor' 'Iris-versicolor' 'Iris-versicolor'
 'Iris-versicolor' 'Iris-versicolor' 'Iris-versicolor' 'Iris-versicolor'
 'Iris-versicolor' 'Iris-versicolor' 'Iris-versicolor' 'Iris-versicolor'
 'Iris-versicolor' 'Iris-versicolor' 'Iris-versicolor' 'Iris-versicolor'
 'Iris-versicolor' 'Iris-versicolor' 'Iris-versicolor' 'Iris-versicolor'
 'Iris-versicolor' 'Iris-versicolor' 'Iris-versicolor' 'Iris-versicolor'
 'Iris-versicolor' 'Iris-versicolor' 'Iris-versicolor' 'Iris-versicolor'
 'Iris-versicolor' 'Iris-versicolor' 'Iris-versicolor' 'Iris-versicolor'
 'Iris-versicolor' 'Iris-versicolor' 'Iris-versicolor' 'Iris-versicolor'
 'Iris-versicolor' 'Iris-versicolor' 'Iris-virginica' 'Iris-virginica'
 'Iris-virginica' 'Iris-virginica' 'Iris-virginica' 'Iris-virginica'
 'Iris-virginica' 'Iris-virginica' 'Iris-virginica' 'Iris-virginica'
```

```
 'Iris-virginica' 'Iris-virginica' 'Iris-virginica' 'Iris-virginica'
 'Iris-virginica' 'Iris-virginica' 'Iris-virginica' 'Iris-virginica'
 'Iris-virginica' 'Iris-virginica' 'Iris-virginica' 'Iris-virginica'
 'Iris-virginica' 'Iris-virginica' 'Iris-virginica' 'Iris-virginica'
 'Iris-virginica' 'Iris-virginica' 'Iris-virginica' 'Iris-virginica'
 'Iris-virginica' 'Iris-virginica' 'Iris-virginica' 'Iris-virginica'
 'Iris-virginica' 'Iris-virginica' 'Iris-virginica' 'Iris-virginica'
 'Iris-virginica' 'Iris-virginica' 'Iris-virginica' 'Iris-virginica'
 'Iris-virginica' 'Iris-virginica' 'Iris-virginica' 'Iris-virginica'
 'Iris-virginica' 'Iris-virginica' 'Iris-virginica' 'Iris-virginica']
```

It's also very common to convert our target (output) data from strings to integers!
In this case we have three discrete target values ('Iris-setosa', 'Iris-versicolor' and 'Iris-virginica') so we can convert to 0, 1 and 2

In [25]:
```
# Pandas makes this very easy for us. Just one line!
y_iris_data = pd.Categorical(pd.factorize(y_string_iris_data)[0])

print('Targets: y_iris_data = ',y_iris_data)
```

```
Targets: y_iris_data =  [0, 0, 0, 0, 0, ..., 2, 2, 2, 2, 2]
Length: 150
Categories (3, int64): [0, 1, 2]
```

Let's consider one of the simplest supervised learning classification algorithms: **Support Vector Machines**.

Note: we can also do classification with Gaussian Processes and with Artificial Neural Networks. But I just want to show a few other examples of algorithms because you will need to search for algorithms you don't know, understand them and decide if they are applicable or not to your problem of interest.

In [26]:

```
from sklearn import svm
# we create an instance of SVM and fit out data. We do not scale our data (this dataset is quite simple)
C = 1.0  # SVM regularization parameter
svm_model = svm.SVC(kernel='rbf', gamma=0.7, C=C)

svm_model.fit(X_2d_iris_data, y_iris_data) # HERE WE ARE USING ONLY 2 FEATURES TO FIT THE MODEL!
                                           # This is just to assess the predictive power of the SVC even though
                                           #  it is only using 2 features for training.
                                           # REMEMBER: WE USUALLY USE ALL FEATURES WHEN TRAINING A MODEL.
```

Out[26]:

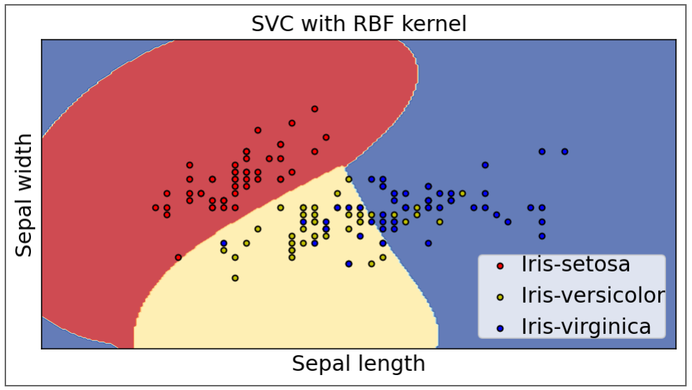
```
SVC(gamma=0.7)
```

In [27]:

```
            # Set-up 2x2 grid for plotting.
# Parameters
n_classes = 3 # number of classes for this problem
plot_colors = "ryb" # defining the 3 colors for each category
plot_step = 0.02 # defining the meshgrid step size
target_names = iris_data.loc[:, 'Species'].unique() # retrieve the names of each category

fig9, ax9 = plt.subplots()
plt.subplots_adjust(wspace=0.4, hspace=0.4)

x1_iris_data, x2_iris_data = X_2d_iris_data[:, 0], X_2d_iris_data[:, 1]

x1_iris_data_min, x1_iris_data_max = x1_iris_data.min() - 1, x1_iris_data.max() + 1 # define min & max of feature 0
x2_iris_data_min, x2_iris_data_max = x2_iris_data.min() - 1, x2_iris_data.max() + 1 # define min & max of feature 1
X1_iris_data_grid, X2_iris_data_grid = np.meshgrid(np.arange(x1_iris_data_min, x1_iris_data_max, plot_step),
                                                   np.arange(x2_iris_data_min, x2_iris_data_max, plot_step))

y_iris_data_pred = svm_model.predict(np.c_[X1_iris_data_grid.ravel(), X2_iris_data_grid.ravel()])
Y_iris_data_grid = y_iris_data_pred.reshape(X1_iris_data_grid.shape)

ax9.contourf(X1_iris_data_grid, X2_iris_data_grid, Y_iris_data_grid, cmap=cm.RdYlBu, alpha=0.8)

# Plot the training points
for i, color in zip(range(n_classes), plot_colors):
    idx = np.where(y_iris_data == i)
    ax9.scatter(X_2d_iris_data[idx, 0], X_2d_iris_data[idx, 1],
                c=color, label=target_names[i],cmap=cm.RdYlBu, edgecolor='black', s=15)

ax9.set_xlim(X1_iris_data_grid.min(), X1_iris_data_grid.max())
ax9.set_ylim(X2_iris_data_grid.min(), X2_iris_data_grid.max())
ax9.set_xlabel('Sepal length', fontsize=15)
ax9.set_ylabel('Sepal width', fontsize=15)
ax9.set_xticks(())
ax9.set_yticks(())
ax9.legend(loc='lower right', borderpad=0, handletextpad=0, fontsize=15)
ax9.set_title('SVC with RBF kernel', fontsize=15)

plt.close(fig9)
```

In [28]:
```
fig9 # classification plot using only 2 features ('SepalLengthCm' and 'SepalWidthCm')
```

Out[28]:



Let's now plot for all 4 features, instead of considering only the first 2.
Let's also use a different classifier. Instead of Support Vector Machines (SVM), we will use **Decision Trees**.
The following is also an example from Scikit-learn.

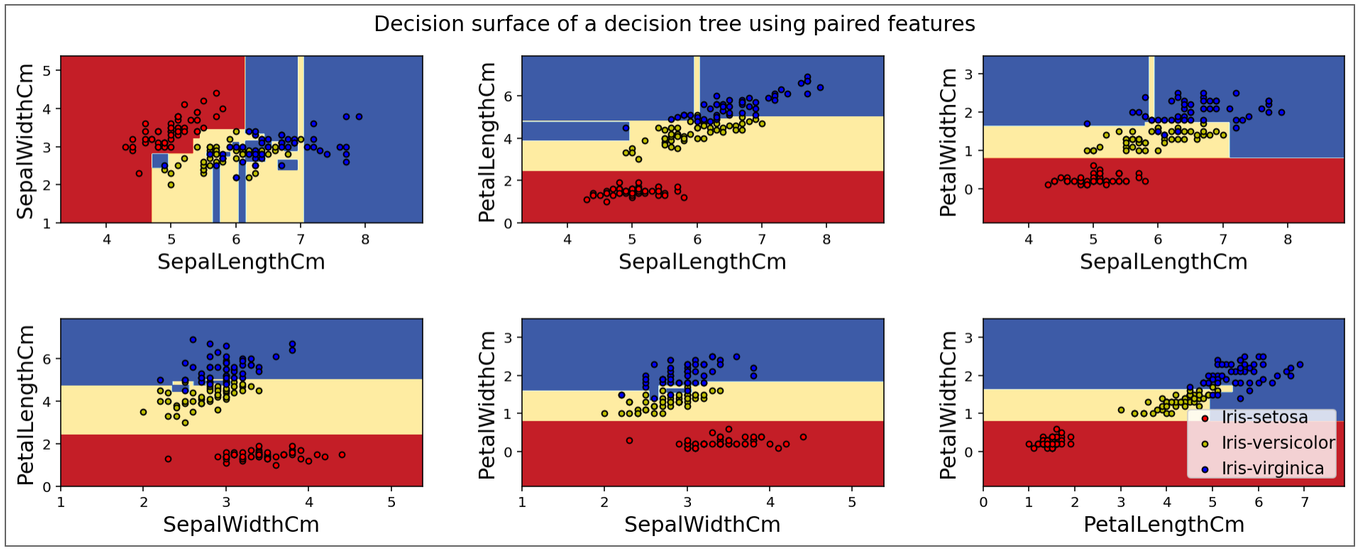
In [29]:

```
# Load all the features (the targets were loaded in a previous cell)
X_iris_data = iris_data.iloc[:, 1:5 ].values # now using all 4 features (inputs)
feature_names = iris_data.columns.values[1:5] # not including the "ID" column

print(feature_names)
print(target_names)
```

```
['SepalLengthCm' 'SepalWidthCm' 'PetalLengthCm' 'PetalWidthCm']
['Iris-setosa' 'Iris-versicolor' 'Iris-virginica']
```

In [31]:
```
fig10 # classification plot with Decision Trees train with each possible pair of 2 features
```

Out[31]:



Notes about visualization in multidimensional spaces

Usually, we **do not want** to train just using 2 pairs (like we did in the figure above)! Instead we usually use all the information available to create better predictions!

- So, we train considering all features (e.g. using all 4 features of the Iris dataset)
- After training using all features, we can plot the model predictions on slices of the space. These slices can be creted by allowing to vary 2 features and fixing all other features to a particular value.
- There are more advanced ways to visualize predictions in multidimensional spaces. For example, we can use *dimensionality reduction* techniques to reduce the number of features to fewer representative features that create more informative plots. Another example is the use of *sensitivity analysis* to select the more important features (without transforming them).

You will explore these and other things in Part 3 of the Midterm Project...

Have fun!