Exercise 1: LIME

In the following, you are guided to implement LIME to interpret a Support Vector Machine (SVM). We use **two** (numeric) features and explore LIME on a multiclass classification problem with only two (numeric) features. The associated files for this exercise are *lime.py* or *lime.R* depending on your preferred programming language. In these files, helper functions for plotting (`get_grid()`, `plot_grid()` and `plot_points_in_grid()`) were already implemented.

a) Inspect Implemented Functions

First of all, make yourself familiar with the already implemented functions in the template files.

- The function `get_grid()` prepares data to visualize the feature space. It creates a $N \times N$ grid, and every point in this grid is associated with a value. This value is obtained by the model's predict method.
- The function `plot_grid()`, visualizes the prediction surface.
- The created plot is an input to the function `plot_points_in_grid()`, which adds given data points to the plot.

b) Sample Points

Your first implementation task is to sample points, which are later used to train the local surrogate model. Complete `sample_points()` by randomly sampling from a uniform distribution. Consider the lower and upper bounds from the input datapoints.

Hint: In Python, you can use the method `dataset.get_configspace().get_hyperparameters_dict()` implemented in the file *utils/dataset.py* to retrieve the lower and upper values. For an example, have a look on the already implemented function `get_grid()`.

c) Weight Points

Given a selected point $\mathbf{x}$ and the sampled points Z from the previous task, we now want to weight the points. Use the following equation with d as Euclidean distance to calculate the weight of a single point $\mathbf{z} \in Z$:

$$\phi_{\mathbf{x}}(\mathbf{z}) = exp(-d(\mathbf{x}, \mathbf{z})^2/\sigma^2). \tag{1}$$

To make plotting easier later on, the weights should be normalized between zero and one. Finally, return the normalized weights in `weight_points()`.

d) Fit Local Surrogate Model

Finally, fit a decision tree with training data and weights. Return the fitted tree in the function `fit_explainer_model()`. What could be problematic?

Exercise 2: Counterfactuals - WhatIf

Counterfactual explanations are a valuable tool to explain predictions of machine learning models. They tell the user how features need to be changed in order to predict a desired outcome. One of the simplest approaches to generate counterfactuals is to determine for a given observation x (`x_interest`) the closest data point which has a prediction equal to the desired outcome.[1] In the following exercise, you should implement this so called WhatIf approach for a binary classifier. The associated files for this exercise are *whatif.py* or *whatif.R*.

a) Implement the following steps in `generate_whatif()`:

[1] Wexler et al. (2019): "The What-If Tool: Interactive Probing of Machine Learning Models"

(i) Subset the `data` to the observations having a prediction different to the one of `x_interest` (this is equal to our desired prediction).

(ii) Calculate the pairwise Gower's distances between `x_interest` and the remaining data points in `data`. *Hint:* the `StatMatch` package in R and `gower` in Python offer implementations of Gower's distance.

(iii) Return the nearest data point as a counterfactual for `x_interest`.

Try out your function with the provided example code.

b) Which attributes from the lecture (*validity*, *sparsity*, ...) does this approach fulfill. Based on this, derive the advantages and disadvantages of the approach.

c) In order to evaluate the sparseness of the counterfactual produced by WhatIf, we could use the following approach: For each feature of the counterfactual instance assess whether setting its value to the one of `x_interest` still leads to a different prediction than `x_interest`. Complete the function `evaluate_cfexp()` using the following steps:

(i) Create an empty vector `feature_nams`.

(ii) For each feature do the following:

   i. Create a copy of the counterfactual.
   ii. Replace the feature value of this copy with the value of `x_interest`.
   iii. Evaluate if the prediction for this copy still differs to the one of `x_interest`.
   iv. If it still differs, add the name of this feature to `feature_nams`.

(iii) End for - return `feature_nams`.

Try out your function given the code example and think about possible extensions of this approach.