Exercise 3
Feature Effects

In this exercise, we are going to explore feature effects using Individual Conditional Expectation, Partial Dependence Plots, and Accumulated Local Effects. After completing the tasks, you will have a detailed understanding of how these methods work. Therefore, you can explain results better and get a feeling for when which method is more suitable.

- You can get a bonus point for this exercise if you pass at least 80% of the tests. Code is automatically tested after pushing. If your tests fail, either your code is wrong, or you solved the task differently. In the second case, we will manually check your solution.
- Four collected bonus points result in a 0.3/0.4 increase of the final grade.
- You are allowed to work in groups with up to three people. You do not need to specify your name/matriculation number again, as it was already collected from the entrance test (assuming you added the "user_info.txt" file in your entrance test repository).
- Follow the 'README.md' for further instructions.
- Make sure that all plots are saved and committed to the repository. The plots are generated by simply executing the three python files.
- Finish this exercise by November 15th, 2023, at 17:00 German time.

General Note: The following tasks heavily rely on the lecture, and it is hardly possible to finish the tasks without the slides. Hence, it is recommended to complete this exercise sheet alongside reading the slides or watching the videos. Additionally to the tasks' description and the slides, functions' docstrings provide code-specific information, which are worth reading.

1 Individual Conditional Expectation

Individual Conditional Expectation (ICE) plots one line per instance that shows how the instance's prediction changes when a feature changes. The following tasks guide you to implement ICE from scratch.

Associated file: *tasks/ice.py*.

1.1 Calculation

Write the ICE algorithm as described in the lecture. In particular, in *calculate_ice* the following steps need to be added:

- We need to sample grid values $x_s^{*(1)}, ..., x_s^{*(n)}$ along x_s. Here, we will use all unique observations of x_s as grid points.
- Iteratively, replace all observed values for feature x_s with the previously selected grid points and evaluate the model on these artificially created data points.

1.2 Preparation

Use the function *calculate_ice* from the previous task to prepare the data for plotting. Iterate in *prepare_ice* over the number of rows s.t. you obtain as many curves as you have data points. Use the parameter *centered* to apply c-ICE. Adjust all y values s.t. the first y value starts at 0.

In both centered and uncentered cases, please note that it is necessary to sort the x values (and to order the y values accordingly) for plotting.

1.3 Plotting

After preparing the ICE data, we now want to plot it. Use all x and y values from *prepare_ice*. Set labels using *dataset* and plot all lines with an alpha (e.g. 0.2) value.

2 Partial Dependency Plot

The partial dependence plot (short PDP or PD plot) shows the marginal effect one or two features have on the predicted outcome of a machine learning model. A partial dependence plot can show whether the relationship between the target and a feature is linear, monotonic or more complex.

Associated file: *tasks/ice.py*.

2.1 Preparation

We now want to use ICE data to plot a PDP. Call *calculate_ice* and iterate over the rows. For each value of x_s, average the corresponding y values. Again, sort the x values (and order the y values accordingly) for plotting.

2.2 Plotting

Use *prepare_pdp* to plot the data. Set the labels with *dataset* and plot the line with no transparency.

3 Accumulated Local Effects

Accumulated local effects describe how features influence the prediction of a machine learning model on average. ALE plots are a faster and unbiased alternative to PDPs, which you will, after solving the tasks, notice when running the tests.

Associated file: *tasks/ale.py*.

3.1 Binning

Since ALE works with bins, we have to select bounds first. Usually the bins are chosen s.t. every bin has roughly the same number of data points. However, to make it more simple, we use the same length for every interval.

Complete the function *get_bounds* to select bounds by using *n_intervals* on numpy's linspace. Those bounds are used for the ALE within the next tasks.

3.2 Calculation

Write the ALE algorithm from scratch inside the function *calculate_ale* using the bounds from the previous task and the help of the lecture slides on the ALE algorithm. For each interval k:

- Select the observations inside the interval (if zero observations are found then return 0 for this interval). In general x_s can be counted as included if it is larger than the lower bound and smaller than or equal the upper bound. However, make sure that elements on the first bound are included as well.
- Intervention: Use the relevant observations to create two new subsets:
  - *X_min*: Values at s-th feature position are replaced by the lower interval value z_{k-1}.
  - *X_max*: Values at s-th feature position are replaced by the higher interval value z_k.
- Prediction: Get the predictions for both *X_min* and *X_max* and calculate the difference between the predictions for *X_max* and *X_min.* Sum up the differences and divide by the number of observations in the interval to get a single value for the current interval *k*.

Use *np.cumsum* to perform the aggregation step, i.e. to get an accumulated value for each interval. Finally, if the *center* parameter is set, center the ALE values for each interval by subtracting the average over the values of all intervals.

Hint: *zip(bounds, bounds[1:])* is a nice trick to iterate over the bounds.

3.3 Preparation

The function *calculate_ale* returns both the used bounds and the ALE values. To use this data for plotting, we need to get the centers of the bounds, which will be used as x values. Use the upper and the lower bound for each interval in the function *prepare_ale* to retrieve those centers.