DAV
Deutsche
Aktuarvereinigung e.V.

German Association of Actuaries (DAV) — Working Group "Explainable Artificial Intelligence"

ICE and PDP for a classification problem with continuous variables

Dr. Benjamin Müller (benjamin1985.mueller@t-online.de)

1. Introduction

In this notebook we focus on the model-agnostic explainability methods **"Individual Conditional Expectation" (ICE)** and **"Partial Dependency Plot" (PDP)** as local respectively as a global explainability method. Aim of this notebook is to reimplement/understand the output of the XAI method *PartialDependenceDisplay* of the module *sklearn.inspection* for a quite simple binary classification problem and a Gradient Boosting model.

Important remark: A requirement file *requirements_hastie_ICE_PDP.txt* is attached. It contains all necessary python libraries which was used during the development of this notebook based on python version 3.10.0.

2. Importing Python functionalities and data preparation

In [1]:

```
# load libraries from requirement.txt:
#!pip install -r requirements_hastie_ICE_PDP.txt

# loading packages and functionalities
import pandas as pd
import numpy as np
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.inspection import PartialDependenceDisplay
from sklearn.datasets import make_hastie_10_2
from scipy.stats.mstats import mquantiles
from sklearn.metrics import mean_absolute_error as mae
from sklearn.metrics import mean_squared_error as mse
```

For generating a dataset we use the make_hastie_10_2 method of the package *sklearn.datasets* (c.p. https://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_hastie_10_2.html resp. Chapter 10 in https://hastie.su.domains/Papers/ESLII.pdf). For reproducebility we choose the random_state = 0. It generates 12.000 rows with 10 numerical/continuous variables as input features and a target vector with values in $\{-1, 1\}$, the two classes.

In [2]:

```
X, y = make_hastie_10_2(random_state=0)
# design matrix:
X = pd.DataFrame(X)
# target value:
y = pd.DataFrame(y)
# renaming the columns:
X.columns = ['column_' + str(i) for i in X.columns]
```

In [3]:

```
# print basic descriptive statistics (e.g., mean, min, max) for all numerical features:
X.describe()
```

Out[3]:

	column_0	column_1	column_2	column_3	column_4	column_5	column_6	column_7	colu
count	12000.000000	12000.000000	12000.000000	12000.000000	12000.000000	12000.000000	12000.000000	12000.000000	12000.00
mean	-0.010536	0.001064	0.005144	-0.002716	0.003345	-0.000346	-0.000138	0.007013	0.00
std	0.998010	1.001357	1.007711	0.996624	0.988042	0.996305	1.008420	0.994275	0.99
min	-4.659953	-3.979925	-3.905825	-3.666662	-3.532992	-3.694285	-4.852118	-3.888444	-4.44
25%	-0.684446	-0.670143	-0.687029	-0.676335	-0.664112	-0.673421	-0.684526	-0.654683	-0.65
50%	-0.015387	0.014320	-0.001378	-0.000677	0.005678	-0.001715	-0.012512	0.006161	0.02
75%	0.655068	0.667477	0.689084	0.669405	0.664691	0.668397	0.681539	0.678485	0.67
max	3.844825	3.852020	4.285856	4.241772	3.803844	3.687019	3.902132	3.491550	3.83

In [4]:

```
# counting the classes:
y.value_counts()
```

Out[4]:

```
-1.0    6068
 1.0    5932
Name: count, dtype: int64
```

3. Modelling

As machine learning model we choose a simple gradient boosting model from scikit-learn:

In [5]:

```
clf = GradientBoostingClassifier(n_estimators=100, learning_rate=1.0,
                                 max_depth=2, random_state=0).fit(X,
                                                                  np.ravel(y)
                                                                 )
```

We do not focus on the quality of the model or testing the model on unseen data, since our primary aim is to reproduce the values of the method *PartialDependenceDisplay* from the module *sklearn.inspection*.

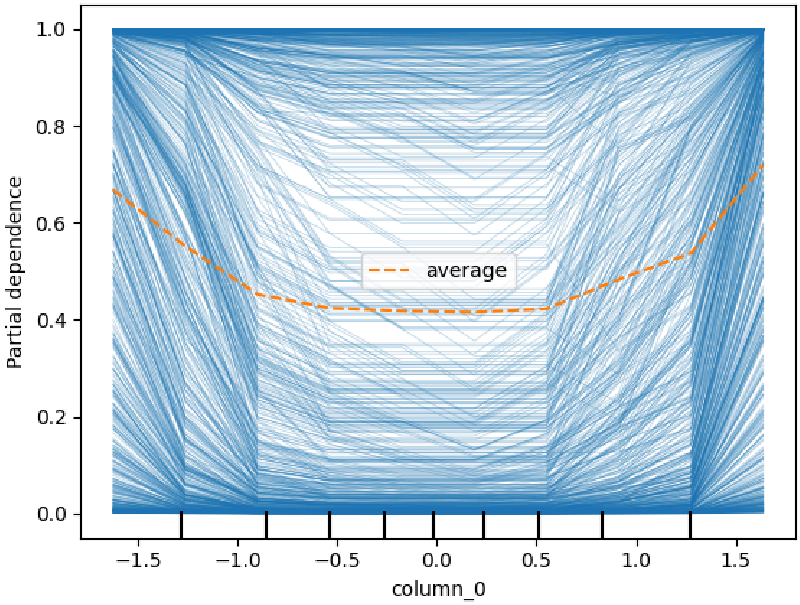
a. Computing the ICE and PDP values with the method PartialDependenceDisplay

In [6]:

```
# choice of feature:
features = ['column_0']

# number of bins for splitting the range of the chosen feature:
grid_resolution = 10

# plotting ICE curves and PDP, store values in x_num:
x_num=PartialDependenceDisplay.from_estimator(clf, X, features, kind='both',
                                              method='brute',
                                              grid_resolution=grid_resolution)
```



In [7]:

```
# ICE values:
ice_values = x_num.pd_results[0]['individual'][0]
# printing the shape:
print('Shape of ice values: ', ice_values.shape)
# printing the values:
print(ice_values)
```

```
Shape of ice values:  (12000, 10)
[[9.98161772e-01 9.84523417e-01 9.40178502e-01 ... 9.84782160e-01
  9.89600803e-01 9.99465247e-01]
 [4.43412068e-01 8.53633983e-02 2.25384913e-02 ... 4.84384332e-02
  1.06101980e-01 6.18055342e-01]
 [9.99840614e-01 9.98641121e-01 9.92560270e-01 ... 9.93556656e-01
  9.97226487e-01 9.99915868e-01]
 ...
 [1.65363500e-02 3.08098732e-03 7.62957974e-04 ... 1.68278514e-03
  3.91506338e-03 1.14982374e-01]
 [9.81560220e-01 8.90504612e-01 5.43335281e-01 ... 5.78975055e-01
  6.69115717e-01 9.64996674e-01]
 [9.97542676e-01 9.86774859e-01 9.48543816e-01 ... 9.63439771e-01
  9.93555571e-01 9.99524452e-01]]
```

In [8]:

```
# PDP values:
pdp_values = x_num.pd_results[0]['average'][0]
# printing the shape:
print('Shape of pdp values: ', pdp_values.shape)
# printing the values:
print(pdp_values)
```

```
Shape of pdp values:  (10,)
[0.66766807 0.5518011  0.45219821 0.42385902 0.41799563 0.4151173
 0.42241194 0.48290582 0.53730847 0.71947852]
```

b. Reimplementation

For reproducing the ice and pdp values for a given feature and given design matrix X we have to do the following steps:

1. Calculate grid for the feature:

   a) Compute the 5 and 95 percent quantiles $x_{0.05}$ and $x_{0.95}$.

   b) Compute grid := $[g_1, \dots, g_n]$ for a given grid resolution n as

$$g_i = x_{0.05} + \frac{i-1}{n}(x_{0.95} - x_{0.05})\,, \quad i = 1, \dots, n$$

2. Iterate over grid entries g_i in grid:

   a) compute the ice values for each fixed grid_entry g_i as

      (i) Set $X_{tmp} := X$.

      (ii) Change entries in column feature to the fixed grid_entry g_i:

$$X_{tmp}[feature] = g_i$$

      (iii) Predict the model output on this temporary data set:

$$y_{pred} = \hat{f}\,(X_{tmp})$$

   b) compute the pdp values as mean of y_{pred}

c. Example

The i-th step in the second main step gives us the i-th column of ice values respectively the i-th pdp value. In the following example we choose *column_0* as feature, $n = 10$ as grid resolution and compute exemplarily the fifth column of the ice values an the fifth pdp value. We compare the self-calculated values with the output of the method *PartialDependenceDisplay*.

In [9]:

```
# chosen feature:
feature = 'column_0'

# 1st step: calculation of grid:

n = 10 # grid resolution (number of entries in grid)
```

```
# calculation of quantiles:
array_for_percentile = [0.05, 0.95]
p = mquantiles(X[feature], prob=array_for_percentile)

# grid as list:
grid = [p[0] + i/(grid_resolution-1)*( p[1] - p[0] )
        for i in range(0, grid_resolution)]

print("5%-Quantil: ", p[0])
print("95%-Quantil: ", p[1])
print('Einträge im Gitter: ', len(grid))

# 2nd step (compute ice values and pdp value for fixed grid entry):

# grid_entry:
grid_entry = 5

# a) (i):
X_tmp = X.copy()

# a) (ii):
X_tmp[feature] = grid[grid_entry-1]

# a) (iii)
y_pred = clf.predict_proba(X_tmp) # predictions (IMPORTANT: predict method must
                                  # be 'predict_proba' in this case)

# ice values (self implemented):
# IMPORTANT: prediction delivers two columns as output (for each class)
y_pred = np.array([val[1] for val in y_pred])

# ice values (of x_num):
ice_values_grid_entry_5 = np.array([ice_values[i][grid_entry-1]
                                    for i in range(0, len(ice_values))])

# comparison of the values:
print('MSE: ', "%0.8f" % mse(y_pred, ice_values_grid_entry_5))

# b) pdp values:
print('pdp value (self implemented) : ', np.mean(y_pred))
print('pdp_value of x_num  : ', pdp_values[grid_entry-1])
```

```
5%-Quantil:  -1.6249705477983925
95%-Quantil:  1.6377365934566899
Einträge im Gitter:  10
MSE:  0.00000000
pdp value (self implemented) :  0.4179956318992344
pdp_value of x_num  :  0.41799563189923233
```

d. Full comparison

At the end we will compare the self implemented values for all grid entries. The procedure is the same:

In [10]:

```
# complete comparison over grid
columns_result = [feature + '_grid(i)=' + str(grid_nr)
                  for grid_nr in range(1, n + 1)]

# initialize dataframe for ice output:
output_df = pd.DataFrame(columns=columns_result)

for grid_entry in range(1, n + 1):
    X_tmp=X.copy()

    # insert fixed grid value in feature column:
    X_tmp[feature] = grid[grid_entry - 1]
    y_pred = clf.predict_proba(X_tmp) # predictions
    y_pred = np.array([val[1] for val in y_pred]) # ice values
    col = feature + '_grid(i)=' + str(grid_entry) # name of column in output
    output_df[col] = y_pred

# comparison of ice values:
ice_values_self = output_df.values
mae_ice =  mae(ice_values, ice_values_self)
print("Mean absolute error between all values: {:e}".format(mae_ice))

# comparison of pdp values:
pdp_values_self = output_df.mean().values

print('MSE pdp values: ', mse(pdp_values_self, pdp_values))
print('MAE pdp values: ', mae(pdp_values_self, pdp_values))
```

```
Mean absolute error between all values: 0.000000e+00
MSE pdp values:  3.422608622769445e-30
MAE pdp values:  1.559863349598345e-15
```

4. Remarks on actuarial diligence

In general it is necessary to prove the assumptions of the explainability methods before applying them. For instance for the methods ICE and PDP it is necessary that the feature of interest is independent to the other features (c.p. https://christophm.github.io/interpretable-ml-book/). Since independence implies uncorrelatedness (or vice versa: correlatedness implies dependence) one can check if the features are correlated. For this purpose you can simply use the Pearson correlation coefficient. Another possibility is to use a Kendall tau independence test to proof the independence of two continuous variables (c.p. https://en.wikipedia.org/wiki/Kendall_rank_correlation_coefficient).

A second important thing is that an actuary should not apply an explainability method without understanding it. If you don't know how the method works, you apply a **black box explainability method** to a **black box machine learning model**. Therefore it is important to understand how the explainability methods work. A reimplementation as above could help here.

5. Conclusion

In general the method *PartialDependenceDisplay* for continuous variables works identical to categorical variables. To reduce the computational time the method do not use all different values of the chosen feature. Instead it determines an equidistant grid with given resolution size on the basis of the 5 and 95 percent quantile of the feature values. Afterwards for each grid point the method predicts values on the basis of a temporary data set (Idea: Insert fixed grid value in feature column for all rows!). Very important is that the method *predict_proba* must be used in the case of a classification problem. Moreover, it is important that the prediction method delivers per row two outputs, the probability for each of the two classes in a binary problem. We have seen in the example that the second column of the prediction output is identical to the ice values of the package. For the calculation of the pdp values the mean of the ice values must be built columnwise.

References

[1] https://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_hastie_10_2.html

[2] T. Hastie, R. Tibshirani and J. Friedman, “The Elements of Statistical Learning”, Second Edition, Springer, 2009 (Link: https://hastie.su.domains/Papers/ESLII.pdf)

[3] https://scikit-learn.org/stable/modules/generated/sklearn.inspection.PartialDependenceDisplay.html

[4] https://christophm.github.io/interpretable-ml-book/)

[5] https://en.wikipedia.org/wiki/Kendall_rank_correlation_coefficient