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

ALE and PDP for correlated features

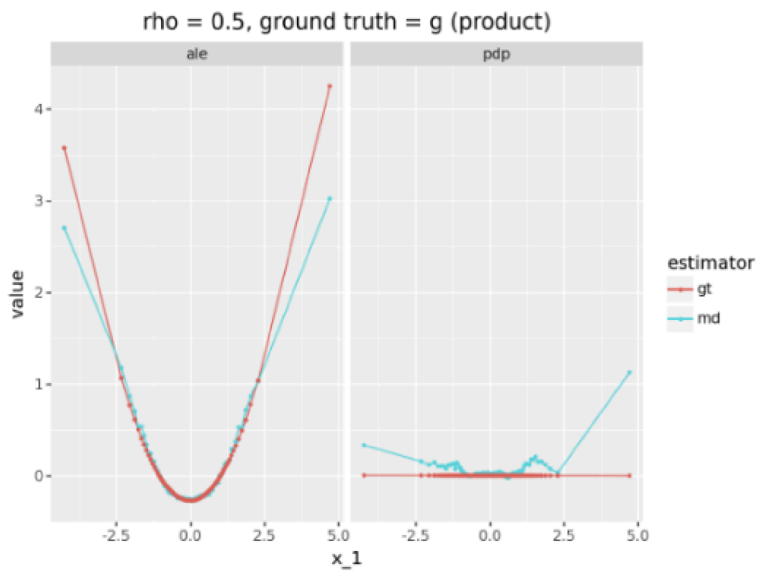
Florian Walla (florian.walla@hotmail.com)

Guido Grützner (guido.gruetzner@quantakt.com)

1. Introduction

Partial Dependendence Plots (PDP) and Accumulated Local Effects (ALE) are two standard visualisation methods to explain multivariate functions respectively machine learning models. This notebook demonstrates ALE and PDP Plots in a simple laboratory setting. Its goal is to illustrate some properties practitioners should be aware of when using these methods in more complex real life applications.

After defining terms and providing some theoretical insights, the first part of the notebook defines functions for calculating PDP and ALE, for simulating data and plotting the results. Subsequently the resulting plots will be discussed and compared to theoretical benchmarks.



- ALE and PDP are quite different summaries of the underlying function. Even without any estimation issues the two methods will regularly produce substantially different results. In the plot, this is represented by the quite different ground thruth estimators (red lines) in the left (ALE) and right column (PDP).
- A PDP is well defined analytically, whether data is correlated or not. But its estimate from a finite sample may be distorted due to extrapolation issues of the model in spite of the model's otherwise excellent fit. This is the "impossible" or "improbable" data problem.
- The ALE is somewhat less exposed to the extrapolation issue due to its more localised estimation approach. Nevertheless in extreme cases it will suffer the same estimation problems as the PDP.
- Both methods are by design extremely reductive summaries of the true underlying model. If their application produces helpful insights or not, depends very much on the model under consideration and the specific question to be analysed.

2. Data generating process

Throughout the notebook, the underlying data generation process will be a bivariate normal distribution (X_1, X_2) with mean vector $\mu = [0, 0]$, unit variance and a correlation coefficient of ρ. Furthermore, we analyse two ground truth functions

$$f(x_1, x_2) = x_1 + 2x_2$$

and

$$g(x_1, x_2) = x_1 x_2.$$

As machine learning model a Gradient Boosting regressor model with a very good fit to the data will be used.

3. Theoretical considerations

For an introduction to PDP and ALE, we refer to Molnar's Interpretable Machine Learning book. Here, we provide a brief summary and derive the analytical, i.e. exact and not sampling based, expressions for PDP and ALE. These analytical results provide benchmarks which can be compared to the simulation results.

The results are summarized in the table below:

	ALE	PDP
f (sum)	$x_1 + C$	x_1
g (product)	$\frac{\rho}{2}x_1^2 + C$	0

Definition of PDP

For a function (or machine learning model) $f : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$, the PDP is the function $f_{PDP} : \mathbb{R} \to \mathbb{R}$ such that

$$f_{PDP}(x_1) = E_{X_2}[f(x_1, X_2)]$$

where E_{X_2} is the expectation with respect to the marginal distribution of (X_1, X_2). It is obviously possible to define a PDP in terms of the second input as well, but here we will use only x_1 as dependent variable for the PDP. For a broader definition of PDP for more input variables we refer to Molnar's Interpretable Machine Learning book.

In applications the expectation is estimated over a grid of x_1 values combined with the empirical mean of function values over the X_2 sample. For details see the function `calc_pdp` below.

PDP of the ground truth functions

Recall that the marginal distributions of a bivariate normal are again normal with exactly the same parameters as the bivariate factors. In our case this means that the marginal distribution, which is also denoted by X_2, is a mean zero unit normal random variable and the PDPs are

$$f_{PDP}(x_1) = E_{X_2}[x_1 + 2X_2] = x_1 + 2E_{X_2}[X_2] = x_1.$$

and

$$g_{PDP}(x_1) = E_{X_2}[x_1 X_2] = x_1 E_{X_2}[X_2] = 0.$$

Notice that in both cases the PDP function does not change with the correlation of the input variables. This is not particular to the chosen input functions, but always true, if the marginal distribution does not depend on the dependency structure of the inputs.

Definition of ALE

As the PDP, the ALE is a function of a single variable and defined as

$$f_{ALE}(x_1) = \int_c^{x_1} E_{X_2|X_1}\left[\frac{\partial f}{\partial x_1}(X_1, X_2) \mid X_1 = z\right] dz$$

where $E_{X_2|X_1}$ is the conditional expectation of X_2 with respect to X_1 and $\frac{\partial f}{\partial x_1}$ the partial derivative of f with respect to the first argument.

The estimation is again straightforward. Derivatives are approximated by finite differences and conditional expectations by local empirical averages defined over x_1-intervals from a grid. See the function `calc_ale` below for details.

ALE of the ground truth functions

For the function f the gradient is

$$\frac{\partial f}{\partial x_1} = \frac{\partial}{\partial x_1}(x_1 + 2x_2) = 1$$

and the conditional expectation

$$E_{X_2|X_1}\left[\frac{\partial f}{\partial x_1}(X_1, X_2) \mid X_1 = z\right] = E_{X_2|X_1}\left[1 \mid X_1 = z\right] = 1.$$

Accordingly the ALE function is

$$f_{ALE}(x_1) = \int_{-C}^{x_1} 1\, dz = x_1 + C.$$

The arbitrary constant of integration C is chosen such that $E_{X_1}[f_{ALE}] = 0$, i.e. such that f_{ALE} is centred.

For the function g one obtains

$$\frac{\partial g}{\partial x_1} = \frac{\partial}{\partial x_1}(x_1 x_2) = x_2.$$

Recall (or find in Wikipedia) that the conditional distribution of $X_2 \mid X_1$ is again a normal distribution with mean ρX_1. Using this result the conditional expectation for the ALE function can be evaluated as

$$E_{X_2|X_1}\left[\frac{\partial g}{\partial x_1}(X_1, X_2) \mid X_1 = z\right] = E_{X_2|X_1}\left[X_2 \mid X_1 = z\right] = \rho z.$$

The ALE function is now

$$g_{ALE}(x_1) = \int_{-C}^{x_1} \rho z\, dz = \frac{\rho}{2} x_1^2 + C.$$

Since $E_{X_1}[\frac{\rho}{2} X_1^2] = \frac{\rho}{2}$, we choose $C = -\frac{\rho}{2}$.

Notice that ALE and PDP for f are identical (and independent of ρ) but that the results for g are quite different if $\rho \neq 0$.

In the case of function g the insight provided by both methods, ALE and PDP, is arguably limited. This pertains to correlated as well as to independent inputs.

4. Functions

In the block below the respective ground truth function, i.e. f or g, is defined as `GTfun`. The calculation will demonstrate the results for different choices of ρ.

There are two reasons to include the ground truth function. First, its plots can be compared to the theoretical benchmarks. Agreement of theoretical results and simulation serves as validation of the implementation and the theoretical derivation. Second, by definition, the ground truth function can not suffer from extrapolation problems. So any difference between the regressor model and the ground truth will be due to estimation issues in the regressor model.

In [3]:
```python
import math
import numpy as np
import pandas as pd
import plotnine as p9
from sklearn.base import BaseEstimator, RegressorMixin
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.metrics import (mean_squared_error, r2_score)
import warnings
```

In [4]:
```python
def GTfun(X, mode='sum'):
    if (mode == 'sum') or (mode == "f"):
        return X["x_1"] + 2 * X["x_2"]
    elif (mode == 'product') or (mode == "g"):
        return X["x_1"] * X["x_2"]
    else:
        raise ValueError("Invalid mode. Choose 'sum', 'f', 'product', ore 'g'")


class GroundTruth(BaseEstimator, RegressorMixin):
    def __init__(self, mode='sum'):
        self.mode = mode
    def fit(self, X, y):
        return self

    def predict(self, X):
        return GTfun(X, mode=self.mode)

reg_gt_sum = GroundTruth(mode='sum')
reg_gt_product = GroundTruth(mode='product')
```

4.1. PDP

In [5]:
```
def calc_pdp(model, df, x,
             n_bins=10,
             use_predict_proba=False):
    """Function to calculate PDP values, see e.g. https://christophm.github.io/interpretable-ml-book/pdp.html
    Parameters:
        model: an model object used for predication, e.g. a sklearn object
        df: pandas data frame used for calculation of the PDP
        n_bins: int: number of bins used for the calculation of PDP
        use_predict_proba: boolean should be True for classification problems and False for Regression problems
    """
    # calculate quantiles as grid for x values
    quantiles_x_grid = np.append(0, np.arange(1 / n_bins, 1 + 1 / n_bins, 1 / n_bins))
    x_grid = df[x].quantile(quantiles_x_grid)

    # Using a loop over all intervals, starting from the lowest:
    pdp = np.ones(n_bins+1)
    for i in range(0, n_bins+1):

        df_new= df.copy()
        df_new[x] = x_grid.iloc[i]

        if use_predict_proba:
            predictions = model.predict_proba(df_new)
        else:
            predictions = model.predict(df_new)

        pdp[i] =  np.mean(predictions)

    return x_grid, pdp
```

4.2. ALE

In [6]:
```
# Function which calculates ALE values
def calc_ale(model, df, x,
             n_bins=10,
             use_predict_proba=False):
    """Function to calculate ALE values, see e.g. https://christophm.github.io/interpretable-ml-book/ale.html

    Parameters:
        model: an model object used for predication, e.g. a sklearn object
        x: string column of df for which pdp has to be calculated. Column must be of numeric datatype.
        df: pandas data frame used for calculation of the ALE
        n_bins: int: number of bins used for the calculation of ALE
        use_predict_proba: boolean should be True for classification problems and False for Regression problems
                           if True, the "second probability (predict_proba)[:,1]" is used
    """
    # calculate quantiles as grid for x values
    quantiles_x_grid = np.append(0, np.arange(1 / n_bins, 1 + 1 / n_bins, 1 / n_bins))
    x_grid = df[x].quantile(quantiles_x_grid)
    # Determine in which Interval (intervals given by grid values) each observation of the pandas df lies in
    x_bin_no = pd.cut(df[x], bins = x_grid, labels=False, include_lowest=True) # number of bin x lies in


    # Using a loop over all intervals, starting from the lowest:
    #    Calculate the difference between the model prediction at the right und left boundary
    #    average that differences over all observations in the
    # Note that you can rid of the for loop to fasten up calculation. It is here for didactic reasons
    local_effects = np.ones(n_bins)
    for i in range(0, n_bins ):

        df_obs_in_intervall = df[x_bin_no == i].copy()

        # setze x-Variable gleich linker Intervallgrenze
        df_left_interval_boundary = df_obs_in_intervall.copy()
        df_left_interval_boundary[x] = x_grid.iloc[i]
        # setze x-Variable gleich rechter Intervallgrenze
        df_right_interval_boundary = df_obs_in_intervall.copy()
        df_right_interval_boundary[x] = x_grid.iloc[i+1]


        if use_predict_proba:
            predictions_left_boundary = model.predict_proba(df_left_interval_boundary )[:,1]
            predictions_right_boundary = model.predict_proba(df_right_interval_boundary )[:,1]
        else:
            predictions_left_boundary = model.predict(df_left_interval_boundary )
            predictions_right_boundary = model.predict(df_right_interval_boundary )

        local_effects[i] =  np.mean(predictions_right_boundary - predictions_left_boundary)

    # Sum the averages of all intervals the get the "uncentered" ALE
    uncentered_acc_local_effects_temp = np.cumsum(local_effects)
    # add zero since the ALE for the 0 % quantile is always zero
    uncentered_acc_local_effects = np.insert(uncentered_acc_local_effects_temp, 0, 0)

    # Average all uncentered ALEs; Substract that average from the uncentered ALEs to get "centered ALES"
    df_copy=df.copy()
    df_copy['x_bin_no'] = x_bin_no
    df_ale_average = df_copy.groupby(['x_bin_no']).size().reset_index().rename(columns={0:"n"})
    df_ale_average['uncentered_acc_local_effects'] = uncentered_acc_local_effects_temp
    average_ale = np.average(df_ale_average['uncentered_acc_local_effects'], weights=df_ale_average['n'])

    accumulated_local_effects = uncentered_acc_local_effects - average_ale


    return x_grid, accumulated_local_effects, uncentered_acc_local_effects
```

4.3. Simulating Data, Estimating Gradient Boosting Regressor, Applying PDP and ALE, plotting the results

The next code block

- sets up the simulation of the data generating process
- trains a machine learning model on the data
- estimates a ALE and PDP both for the model and the ground truth
- produces a plot with the results

In [8]:
```
def simulate_data(rho,N, seed=42):
    mean = [0, 0]
    sigma = 1
    cov = [[1, rho * sigma], [rho * sigma, sigma**2]]


    # simulating data
    x_1, x_2 = np.random.default_rng(seed=seed).multivariate_normal(mean, cov, N).T

    df = pd.DataFrame({"x_1": x_1, "x_2": x_2})
    return df
```

In [9]:
```
def wrapper_display_ale_pdp_varying_rho(rho = -0.99, N = 100_000,
n_bins = 100,  show_metrics = False, seed = 42, mode='sum', return_details = False):


    df_x = simulate_data(rho=rho, N=N, seed=seed)

    # initializing Ground Truth
    reg_gt = GroundTruth(mode=mode)

    # predict ground truth
    y = reg_gt.predict(df_x)

    df = df_x.copy()
    df['y'] = y

    list_x_vars = ["x_1", "x_2"]
    list_y_var = ["y"]

    # initialize ML Model
    reg_md = GradientBoostingRegressor(random_state=0)
    # train test split
    last_row_train = math.floor(0.8*N)
    df_train = df[:last_row_train].copy()
    df_test = df[last_row_train:].copy()

    # ## Estimate Regressors
    reg_md.fit(df_train[list_x_vars], np.ravel(df_train[list_y_var]))

    if show_metrics:
        df_test['prediction'] = reg_md.predict(df_test[list_x_vars])
        print("MSE " + str(mean_squared_error(df_test[list_y_var], df_test['prediction'])))
        print("R2 " + str(r2_score(df_test[list_y_var], df_test['prediction'])))

    # calculate pdp values
    x_grid_pdp_x_1, pdp_md = calc_pdp(model = reg_md,df = df_train[list_x_vars],
                                      x = 'x_1', n_bins=n_bins,use_predict_proba=False)
    x_grid_pdp_x_1, pdp_gt = calc_pdp(model = reg_gt,df = df_train[list_x_vars],
                                      x = 'x_1', n_bins=n_bins,use_predict_proba=False)

    # calculate ale values
    x_grid_ale_x_1, ale_md, _ = calc_ale(model = reg_md,df = df_train[list_x_vars],
                                         x = 'x_1', n_bins=n_bins,use_predict_proba=False)
    x_grid_ale_x_1, ale_gt, _ = calc_ale(model = reg_gt,df = df_train[list_x_vars],
                                         x = 'x_1', n_bins=n_bins,use_predict_proba=False)

    # combine all in one pandas df
    # note that the x grids of ALE and PDP are the same (as we use the same quantiles for both)
    assert np.allclose(x_grid_ale_x_1, x_grid_pdp_x_1)
    df_plot = pd.DataFrame({"x_1" :x_grid_ale_x_1, "ale_md": ale_md, "pdp_md": pdp_md,
                            "ale_gt": ale_gt, "pdp_gt": pdp_gt})
    df_plot_melt = pd.melt(df_plot, id_vars=['x_1'])
    df_plot_melt['xai_method'] = df_plot_melt['variable'].str[:3]
    df_plot_melt['estimator'] = df_plot_melt['variable'].str[-2:]


    # plot
    warnings.filterwarnings("ignore", message=".*containing missing values.*")
    warnings.filterwarnings("ignore", message=".*The default of observed=False*")
    if mode == "f" or mode == "sum":
        label_string = "f (sum)"
    if mode == "g" or mode == "product":
        label_string = "g (product)"
    plot = (p9.ggplot(df_plot_melt, p9.aes(x='x_1', y='value', color="estimator")) +
        p9.geom_line()+
        p9.geom_point(size=0.5) +
        p9.facet_grid('.~xai_method', scales="free_y") +
        p9.ggtitle(f'rho = {rho}, ground truth = {label_string}')
     )


    if return_details:
        return x_grid_pdp_x_1, pdp_md, pdp_gt,  ale_md, ale_gt
    else:
        return plot
```

The following function is just a convenience functions that calls the wrapper function from above twice - once for each ground truth function.

In [11]:
```
def plot_sum_and_plot_product(rho, seed =42, **kwargs):
    p_sum = wrapper_display_ale_pdp_varying_rho(rho = rho, N= N,
                                                show_metrics=show_metrics,
                                                seed = seed, mode='sum', **kwargs)
    p_product =  wrapper_display_ale_pdp_varying_rho(rho = rho, N= N,
                                                     show_metrics=show_metrics,
                                                     seed = seed, mode='product', **kwargs)
    p_sum.show()
    p_product.show()
```

5. Simulation results

Now we can plot our simulation results.

Each plot will show the graphs of four functions, two ALE graphs in the left panel and two PDP graphs in the right panel, each respectively for the ground truth function itself (red, `_gt`) and the estimate of the ground truth by the Gradient Boosting regressor (blue, `_md`).

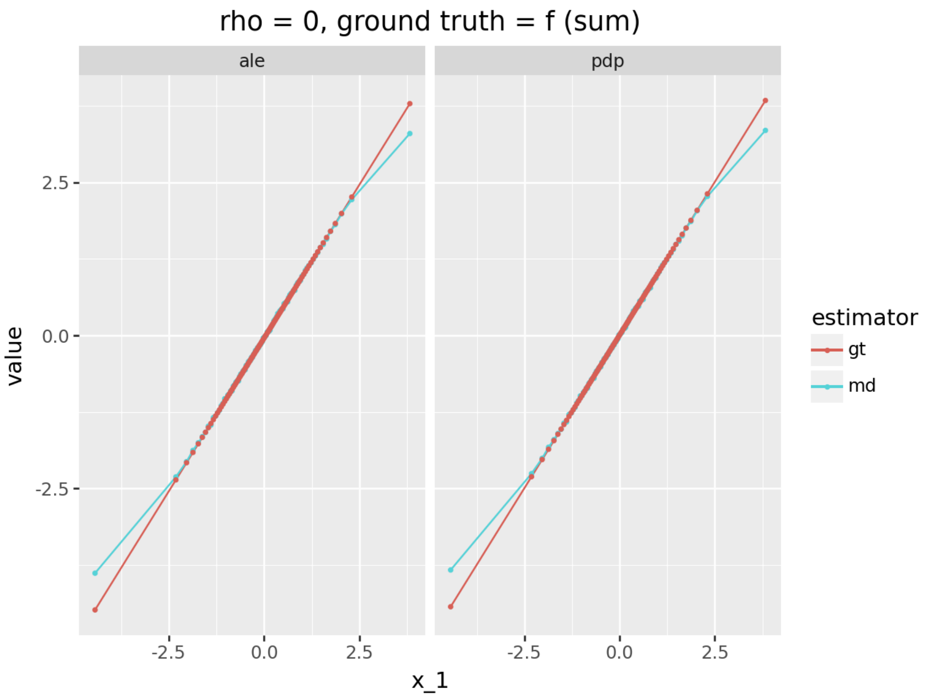
In [12]:
```
N = 100_000
```

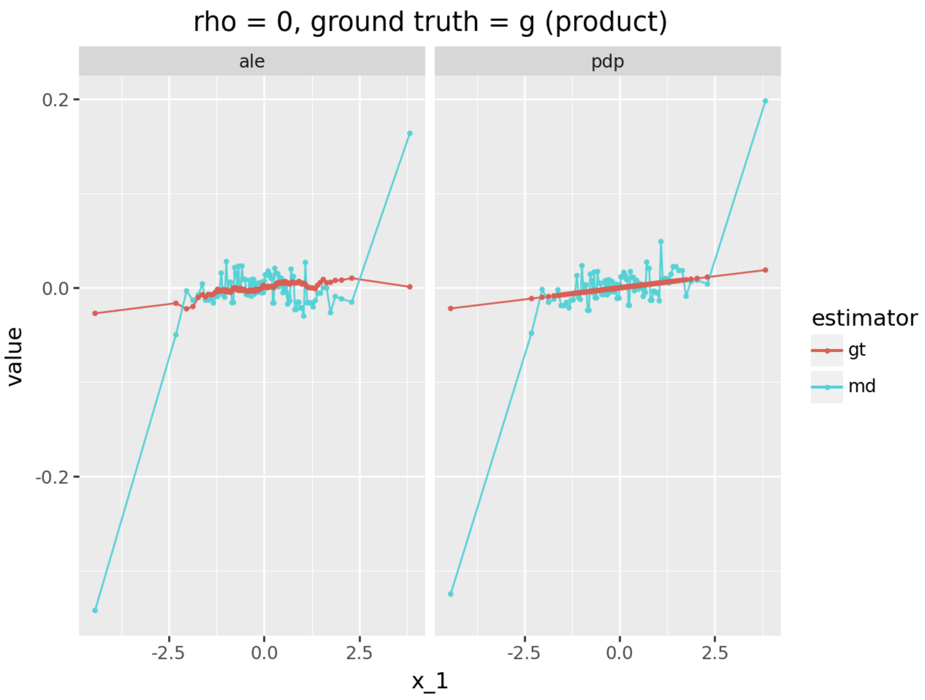
```
show_metrics = True
```

Lets start with uncorrelated data. We see a good fit of both methods (R^2 > 95%) for both ground truth functions.

```
In [13]: plot_sum_and_plot_product(rho= 0, n_bins=100, seed = 4)
```

```
MSE 0.004750144237195441
R2 0.9990703243186256
MSE 0.0438616367942629
R2 0.9568270995587546
```





In case of the ground truth function g we see rather big deviation for the first and last quantile for the model derived estimates. We will show in the appendix that this is due to model performance at the outermost grid points. The deviation also depends on the choosen seed.

Moreover, there seems to be an upward trend in the ground truth function. However, this is due due the fact that the empirical mean is not exactly zero as the analytical expected value:

```
In [14]: df_sim = simulate_data(rho=0, N=N, seed = 4)
         df_sim['x_2'].mean()
```

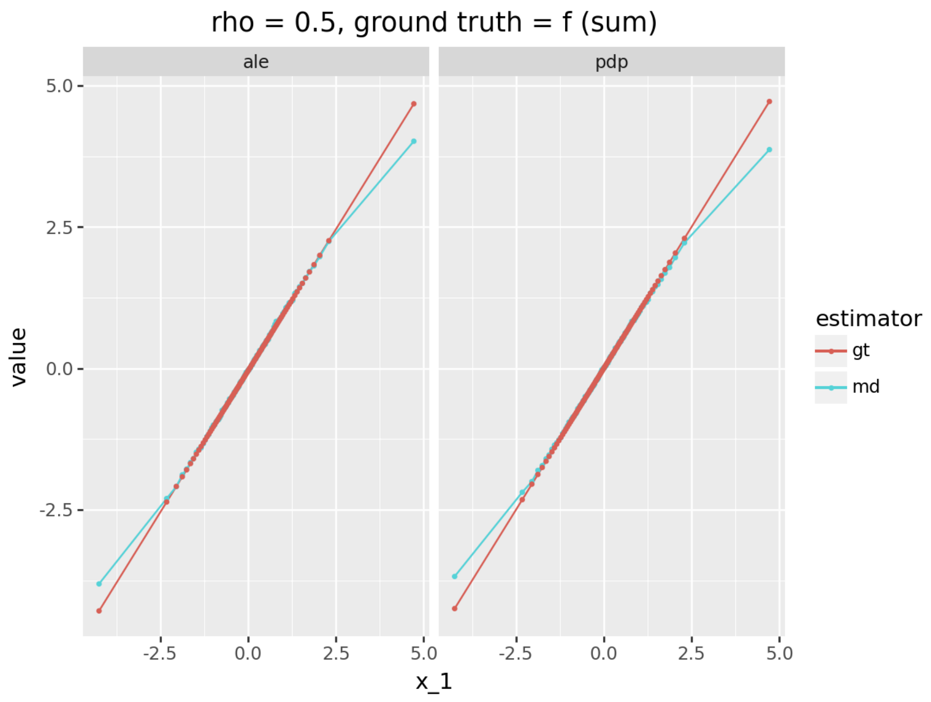
```
Out[14]: np.float64(0.004234110336476717)
```

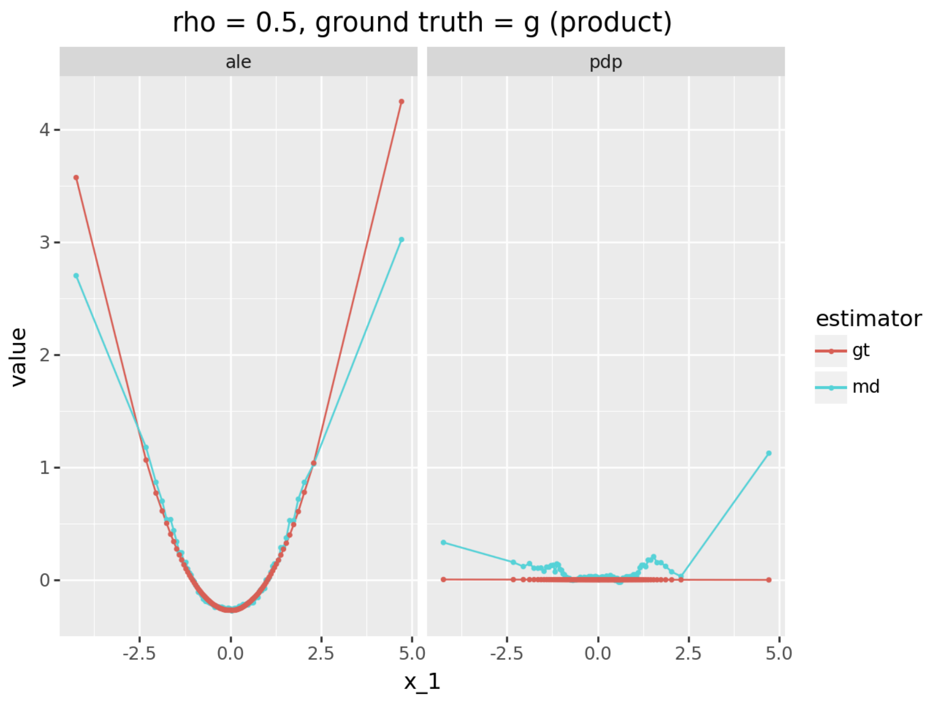
Since for the g ground truth the pdp is calculated by multiplying the grid point with all values of x_2, the sign is determined by the sign of the grid point and the positive sign of the empirical mean, resulting in a slight upward slope changing of the ground truth estimate at zero.

If we increase the correlation to 0.5, we observe that for the product ground truth function g, the estimates between ALE and PDP differ significantly, resulting in distinct shapes. This is expected according to the analytical results presented above. The model-derived estimates remain close to their ground truth counterparts, though some deviations begin to appear.

In [15]:
```
plot_sum_and_plot_product(rho= 0.5, n_bins=100, seed = 4)
```

```
MSE 0.006786643724377091
R2 0.9990257163021148
MSE 0.016266686928808196
R2 0.9871324029704325
```



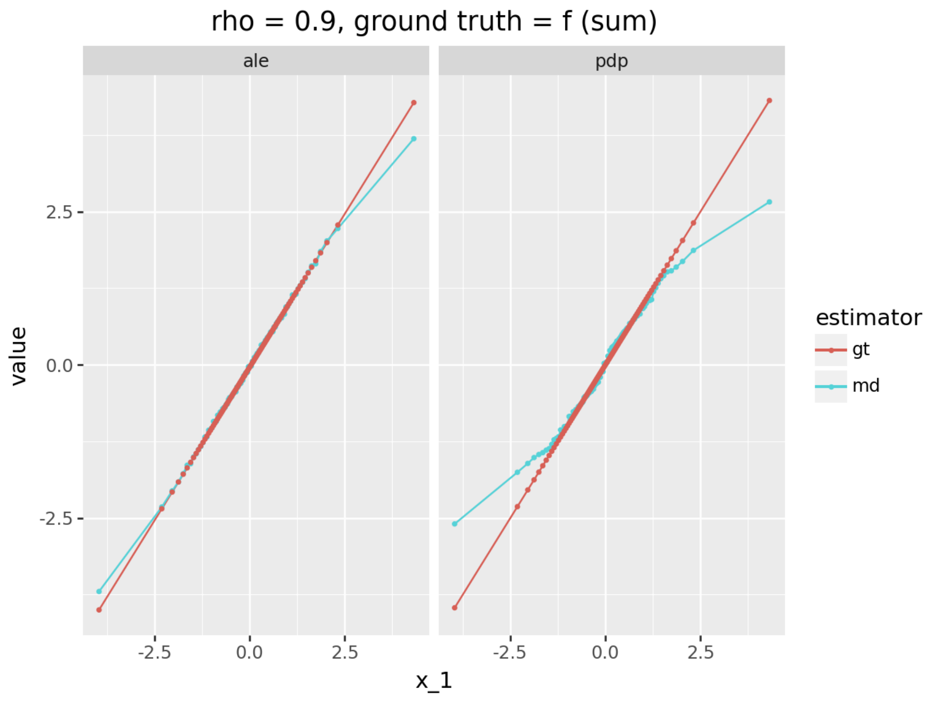


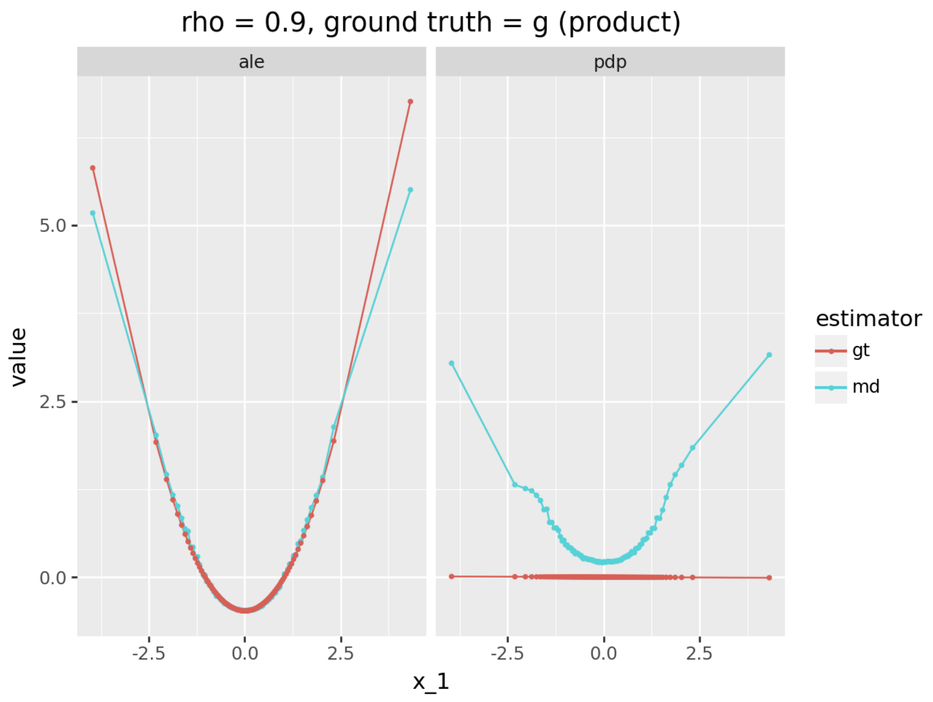
Now, we are increasing the correlation to 0.9. In the case of the ground truth function f, we observe some differences between the ground truth and model-derived estimates, with the discrepancies being somewhat larger for the PDP estimates compared to the ALE estimates.

For the ground truth function g, the model-based PDP estimates deviate significantly from the ground truth, displaying a completely different shape, in spite of the R2=99% quality of the fit. In a real-world application, this could lead to incorrect conclusions.

```
In [16]: plot_sum_and_plot_product(rho= 0.9, n_bins=100, seed = 4)
```

```
MSE 0.0042638827052461975
R2 0.9995011891628777
MSE 0.008441479473109454
R2 0.9953494049356651
```

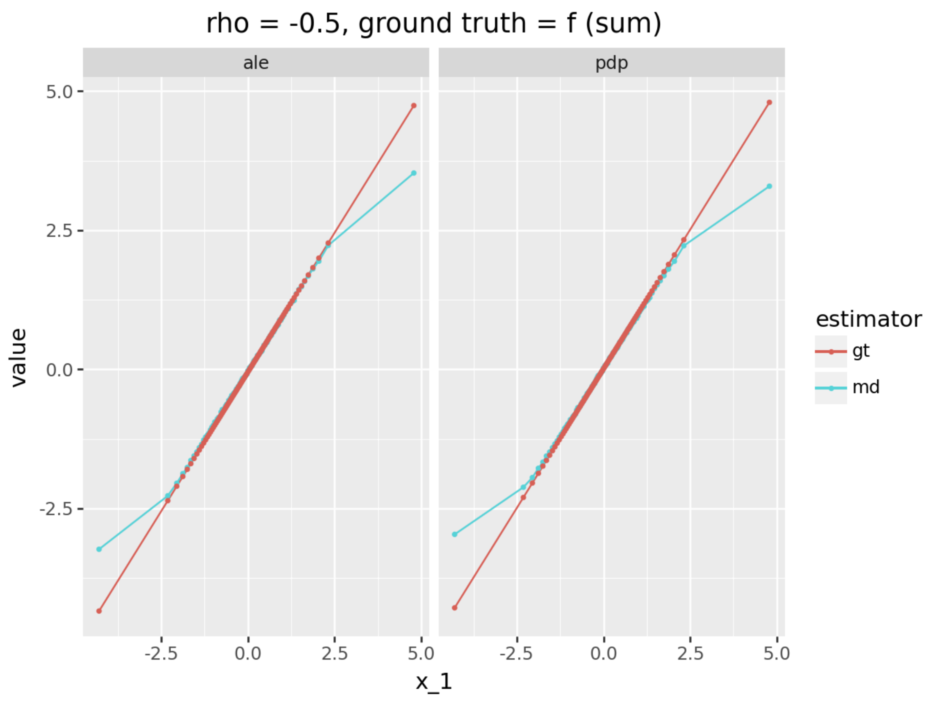


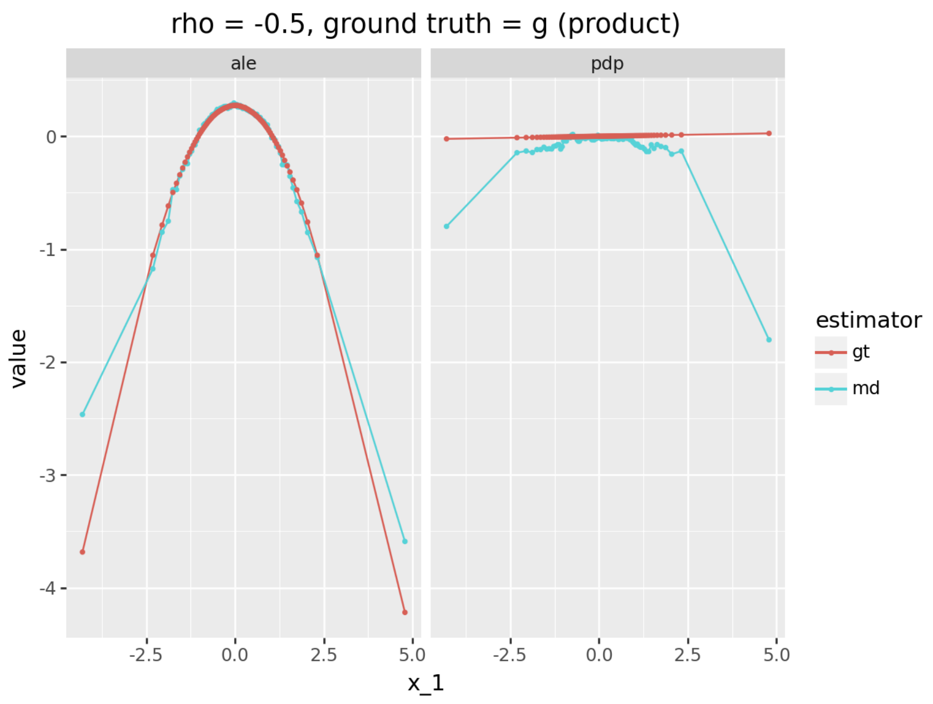


A similiar pattern can be seen if we analyse negative correlations.

```
In [17]: plot_sum_and_plot_product(rho= -0.5, n_bins=100, seed = 4)
```

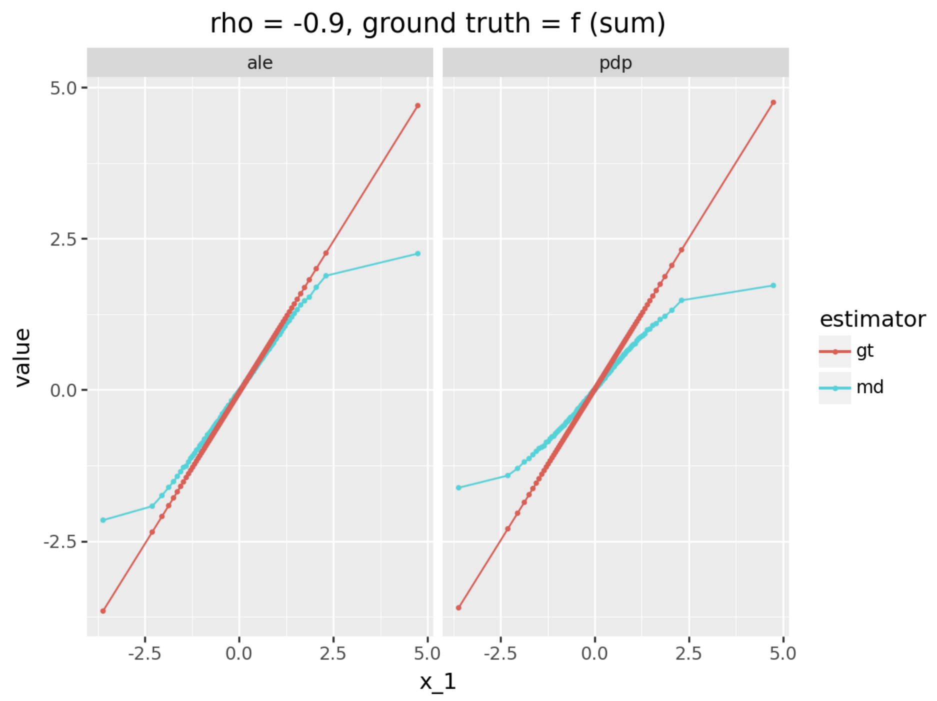
```
MSE 0.004355779782403611
R2 0.9985766646720536
MSE 0.016278090848896948
R2 0.9871233820155876
```

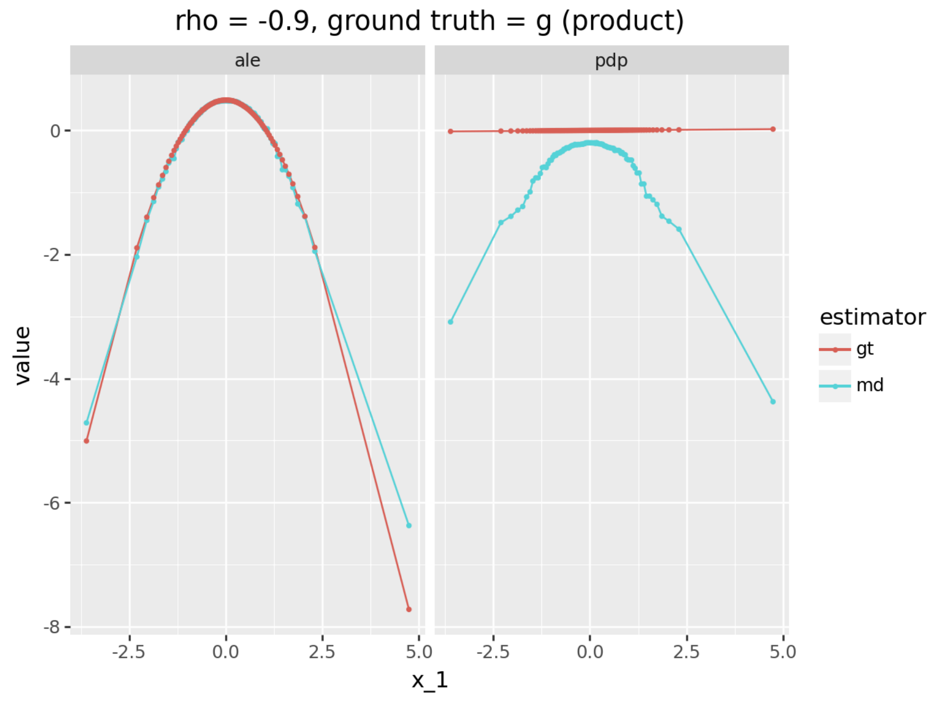




```
In [18]: plot_sum_and_plot_product(rho= -0.9, n_bins=100, seed = 4)
```

```
MSE 0.0073707441271422865
R2 0.9947603309985177
MSE 0.00844147947310946
R2 0.9953494049356651
```





6. Impossible Data

One reason why estimated PDPs often fail to accurately reflect the true functional form of the ground truth estimator is their reliance on impossible data—or, in the specific case of the bivariate normal distribution, highly atypical data. By considering the entire range of x_2 values to compute the PDP for a particular (grid) value of x_1, the PDP method implicitly assumes that the marginal distribution of x_2 given x_1 is identical to the unconditional distribution of x_2.

In the context of the bivariate normal distribution, this assumption translates to:

$$X_2 \mid X_1 = a \sim \mathcal{N}\left(\mu_2, \sigma_2^2\right)$$

However, the actual conditional distribution of X_2 given $X_1 = a$ is:

$$X_2 \mid X_1 = a \sim \mathcal{N}\left(\mu_2 + \frac{\sigma_2}{\sigma_1}\rho(a - \mu_1), (1-\rho^2)\sigma_2^2\right)$$

This would only coincide with the previous assumption if $\rho = 0$, which corresponds to the case of independence.

We will visualize this in the following. To start, we simulate the same data as before.

In [19]:
```
rho = 0.5
df_sim = simulate_data(rho=rho, N=N, seed = 4)
x_grid_pdp_x_1, pdp_md, pdp_gt, ale_md, ale_gt = \
        wrapper_display_ale_pdp_varying_rho(rho = rho, N= N, show_metrics=show_metrics,
                                            mode='product',return_details=True)
```

```
MSE 0.02001784653582598
R2 0.983471957577019
```

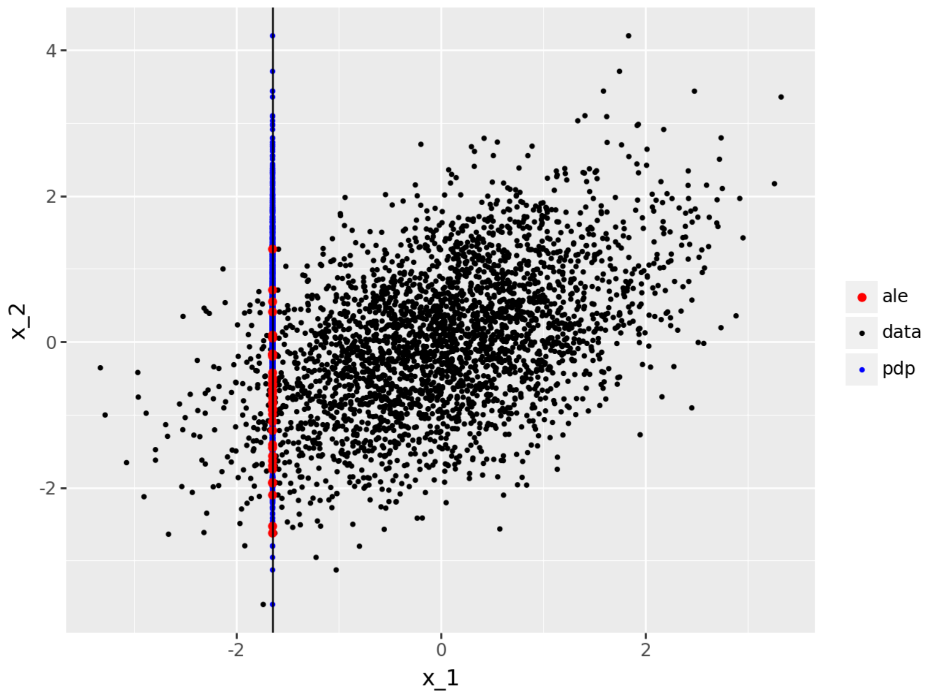
Next, we highlight the values that the ALE method uses to calculate its value for the grid point/quantile of interest-

In [20]:
```
index_quantile_of_interest = 5
quantile_of_interest = x_grid_pdp_x_1.iloc[index_quantile_of_interest]
print(quantile_of_interest)

df_sim["used_by_ale"] = \
    np.where((df_sim["x_1"] >= quantile_of_interest) &
(df_sim["x_1"] <= x_grid_pdp_x_1.iloc[index_quantile_of_interest+1]), True, False)
```

```
-1.6480496787589138
```

Now we plot the simulated data. The points used by ALE to calculate the ALE value for x_1 at the 5% quantile (represented by the vertical line) are shown in red. In contrast, PDP uses all x_2 values, shown in blue, to compute the values at the quantile of interest. However, as the plot demonstrates, some of the x_2 values are highly unlikely given the 5% quantile of x_1.

In [21]:
```
no_points_plot = min(3000, N)
df = df_sim[["x_1", "x_2", "used_by_ale"]].sample(n=no_points_plot, random_state=1)
pt_data = df[["x_1", "x_2"]]
pt_ale = pd.DataFrame({"x_1": quantile_of_interest,
                       "x_2": df.loc[df["used_by_ale"], "x_2"]})
pt_pdp = pd.DataFrame({"x_1": quantile_of_interest,
                       "x_2": df["x_2"]})
plotdf = pd.concat([pt_data, pt_pdp, pt_ale], keys=["data", "pdp", "ale"]).reset_index()
plotdf = plotdf.drop(labels="level_1", axis=1).rename({"level_0": "type"}, axis=1)

(p9.ggplot(plotdf, p9.aes(x='x_1', y='x_2', colour="type", shape="type"))
    + p9.geom_point()
    + p9.scale_color_manual(values = {"data": "black", "pdp": "blue", "ale": "red"})
    + p9.scale_shape_manual(values = {"data": ".", "pdp": ".", "ale": "o"})
    + p9.geom_vline(xintercept=quantile_of_interest, color='black')
    + p9.labs(color="", shape=""))
```

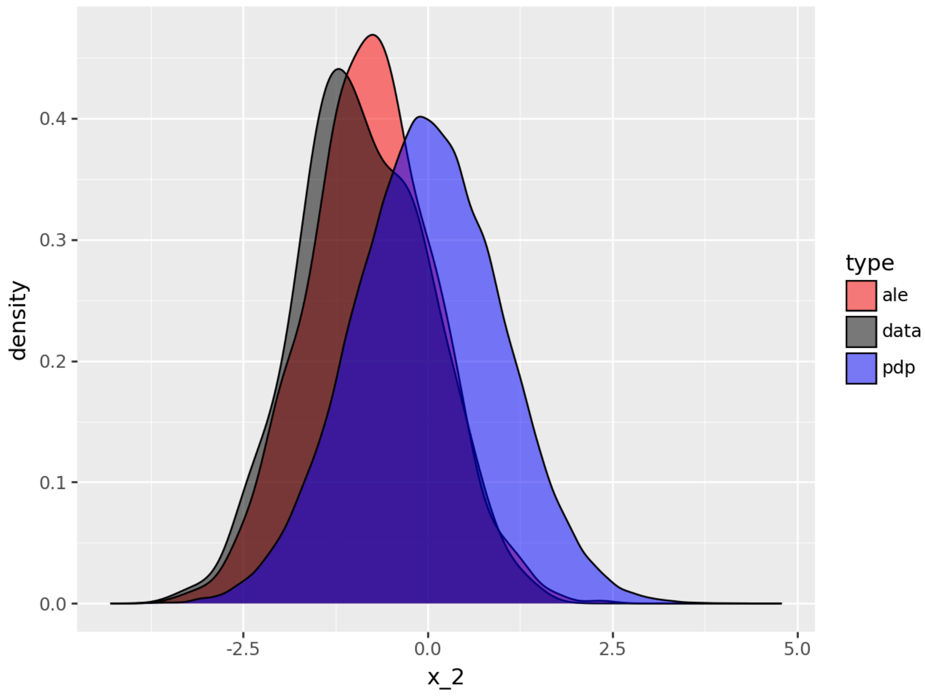


The next plot illustrates this concept from a different perspective. It shows the density of the points used for estimation at the quantile/grid value of interest by ALE (blue), PDP (red), and the density of x_2 given the quantile of interest/grid for x_1 in green

In [22]:
```
# correct data (epsilon around the quantile)
# we could use the formula for the conditional normal distribution above to calculate these values.
# However, we make our (plotting-) lives a bit easier by using all points very closely to the quantile of interest
epsilon = 0.01
df_sim_quantile_of_interest = df_sim[(df_sim["x_1"] > quantile_of_interest - epsilon) &
(df_sim["x_1"] < quantile_of_interest + epsilon) ]
```

```
# pdp just takes all the data
df_sim_ale = df_sim[df_sim['used_by_ale']]
```

In [23]:
```
plotdf = pd.concat([df_sim, df_sim_quantile_of_interest, df_sim_ale],
                   keys=["pdp", "data", "ale"]).reset_index()
plotdf = plotdf.drop(labels="level_1", axis=1).rename({"level_0": "type"}, axis=1)
(
    p9.ggplot(plotdf, p9.aes(x='x_2', fill="type"))
    + p9.geom_density(alpha=0.5)
    + p9.scale_fill_manual(values = {"data": "black", "pdp": "blue", "ale": "red"})
)
```



The plot shows that the values used by the PDP are highly unlikely given the value of x_1. By adjusting the value of ρ in the first coding cell of this section, you can observe that the divergence between the distributions increases with a stronger correlation.

This issue can have significant consequences in practical applications. For example, consider car insurance, where x_1 represents the age of the driver and x_2 represents the number of years driving without making a claim. Young drivers are naturally limited in the number of years they can drive without a claim. However, PDP would use all data points, including those from older drivers who have more claim-free years. This can result in unrealistic data points, such as an 18-year-old driver with 20 years of driving experience without a claim—an attribute effectively "borrowed" by PDP from a, say, 50-year-old driver.

7. Conclusion

In this notebook, we observed that:

- ALE and PDP are based on different underlying theories and only align in specific cases. When a linear effect is present, analytical results show that they share the same slope.
- The estimate of PDP may be distorted due to the use of impossible data, even when the model's overall fit is excellent. Based on our data generation process, this could be a significant issue in cases with interaction effects and correlated variables.
- ALE is somewhat less susceptible to the problem of impossible data because of its more localized estimation approach.
- It is possible to reimplement both ALE and PDP with just a few lines of code.

Another interesting notebook on "PDPs and Impossible Data for Categorical Variables" can be found in the "Use Case SOA GLTD Experience Study" section of the GitHub repository belonging to the working group on Explainable Artificial Intelligence. You can access it here: https://github.com/DeutscheAktuarvereinigung/WorkingGroup_eXplainableAI_Notebooks.

Appendix: Inspection of the Outermost Grid Points

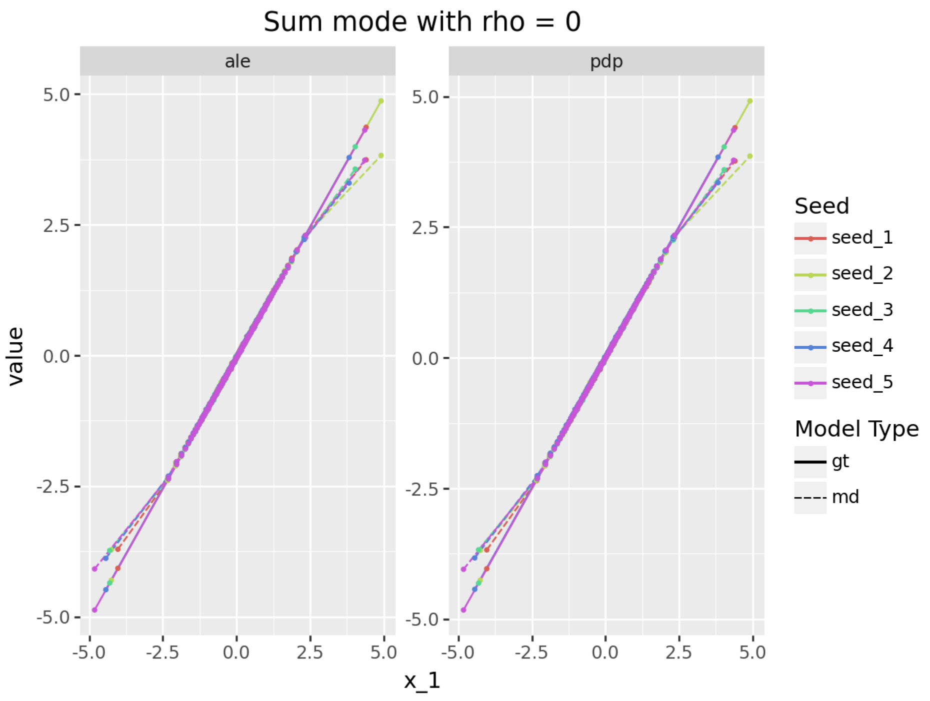
As mentioned above, we want to inspect the last/first gridpoints, which led to huge deviations for the product ground truth g in the simulation results.

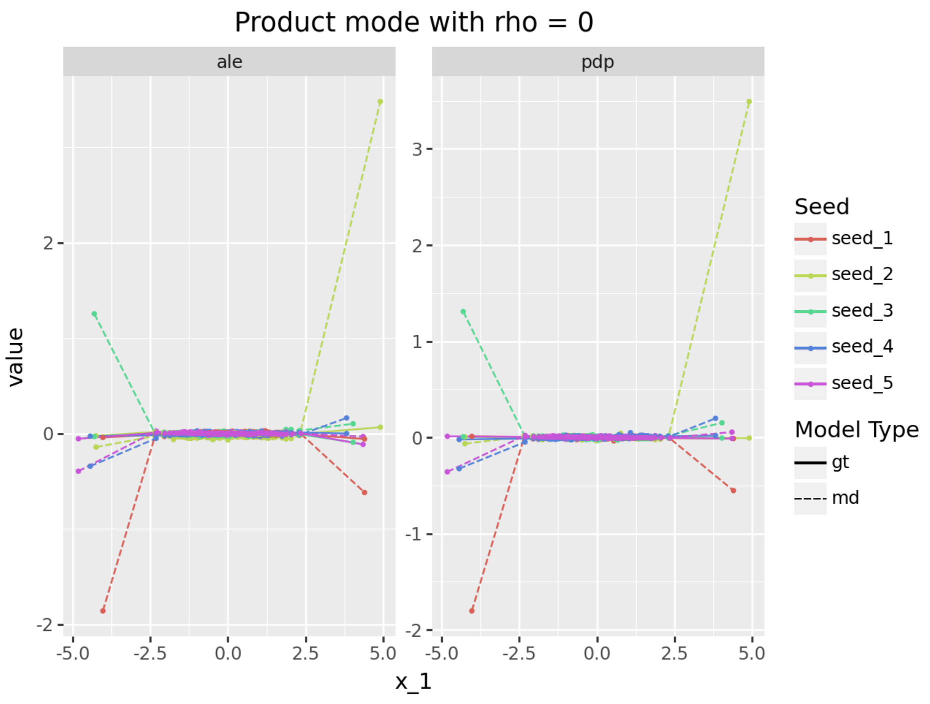
We will demonstrate that these deviations are dependent on the seed and can be attributed to poor model fit at the extreme quantiles.

We start by first producing the same plots as above, yet for varying seeds:

In [24]:
```python
def plot_sum_and_plot_product_varying_seed(rho, seeds=[1,2,3], N=100_000, show_metrics=False, **kwargs):
    """
    Plots results for both 'sum' and 'product' modes for a given rho, across multiple seeds.
    Different seeds are shown in different colors, while _md and _gt are shown with different line types.

    Parameters:
    - rho (float): Correlation coefficient.
    - seeds (list): List of random seeds to run the analysis with.
    - N (int): Number of samples to generate.
    - show_metrics (bool): Whether to show metrics like MSE and R².
    - kwargs: Additional arguments to pass to `wrapper_display_ale_pdp_varying_rho`.
    """
    # List to store plot data for each seed
    plot_data_sum = []
    plot_data_product = []

    # Loop through each seed and run the function for 'sum' and 'product' modes
    for seed in seeds:
        # Get the results for 'sum' mode
        x_grid_sum, pdp_md_sum, pdp_gt_sum, ale_md_sum, ale_gt_sum = wrapper_display_ale_pdp_varying_rho(
            rho=rho, N=N, show_metrics=show_metrics, seed=seed, mode='sum', return_details=True, **kwargs
        )

        # Get the results for 'product' mode
        x_grid_prod, pdp_md_prod, pdp_gt_prod, ale_md_prod, ale_gt_prod = wrapper_display_ale_pdp_varying_rho(
            rho=rho, N=N, show_metrics=show_metrics, seed=seed, mode='product', return_details=True, **kwargs
        )

        # Create a dataframe for 'sum' mode results
        df_sum = pd.DataFrame({
            "x_1": x_grid_sum,
            "ale_md": ale_md_sum, "pdp_md": pdp_md_sum,
            "ale_gt": ale_gt_sum, "pdp_gt": pdp_gt_sum,
            "seed": f"seed_{seed}", "mode": "sum"
        })

        # Create a dataframe for 'product' mode results
        df_prod = pd.DataFrame({
            "x_1": x_grid_prod,
            "ale_md": ale_md_prod, "pdp_md": pdp_md_prod,
            "ale_gt": ale_gt_prod, "pdp_gt": pdp_gt_prod,
            "seed": f"seed_{seed}", "mode": "product"
        })

        # Append data to the lists
        plot_data_sum.append(df_sum)
        plot_data_product.append(df_prod)

    # Concatenate all 'sum' data and 'product' data into respective DataFrames
    df_plot_sum = pd.concat(plot_data_sum)
    df_plot_product = pd.concat(plot_data_product)

    # Melt data for plotting
    df_plot_sum_melt = pd.melt(df_plot_sum, id_vars=["x_1", "seed"],
                               value_vars=["ale_md", "pdp_md", "ale_gt", "pdp_gt"],
                               var_name="variable", value_name="value")

    df_plot_product_melt = pd.melt(df_plot_product, id_vars=["x_1", "seed"],
                                   value_vars=["ale_md", "pdp_md", "ale_gt", "pdp_gt"],
                                   var_name="variable", value_name="value")

    # Create new columns for XAI method (_md or _gt) and estimator (ALE or PDP)
    df_plot_sum_melt["xai_method"] = df_plot_sum_melt["variable"].str[:3]  # ale or pdp
    df_plot_sum_melt["estimator"] = df_plot_sum_melt["variable"].str[-2:]  # md or gt

    df_plot_product_melt["xai_method"] = df_plot_product_melt["variable"].str[:3]
    df_plot_product_melt["estimator"] = df_plot_product_melt["variable"].str[-2:]

    # Plot for 'sum' mode
    plot_sum = (p9.ggplot(df_plot_sum_melt, p9.aes(x='x_1', y='value', color="seed", linetype="estimator")) +
                p9.geom_line() +
                p9.geom_point(size=0.5) +
                p9.facet_wrap('~xai_method', scales="free_y") +
                p9.ggtitle(f'Sum mode with rho = {rho}') +

                p9.labs(color="Seed", linetype="Model Type"))

    # Plot for 'product' mode
    plot_product = (p9.ggplot(df_plot_product_melt, p9.aes(x='x_1', y='value', color="seed", linetype="estimator"))
                    p9.geom_line() +
                    p9.geom_point(size=0.5) +
                    p9.facet_wrap('~xai_method', scales="free_y") +
                    p9.ggtitle(f'Product mode with rho = {rho}') +
                    p9.labs(color="Seed", linetype="Model Type"))

    # Display the plots using show()
    plot_sum.show()
    plot_product.show()
```

In [25]:
```
plot_sum_and_plot_product_varying_seed(rho=0, N=N, seeds=[1,2,3,4,5],
                                       n_bins=100, show_metrics=False)
```





We observe, that huge the deviations for the last/first gridpoint occur in several, though not all seeds. Additionally, we notice that both ALE and PDP are impacted. However, the ground truth is far less affected compared to the model-derived (md) results.

This suggests a potential issue with the model's fit at the outermost grid points. Recall that, by the definition of ALE and PDP, we are evaluating the gradient booster at the maximum and minimum values of x_1 on which it was trained—an area of the feature space where a suboptimal fit would not be unexpected

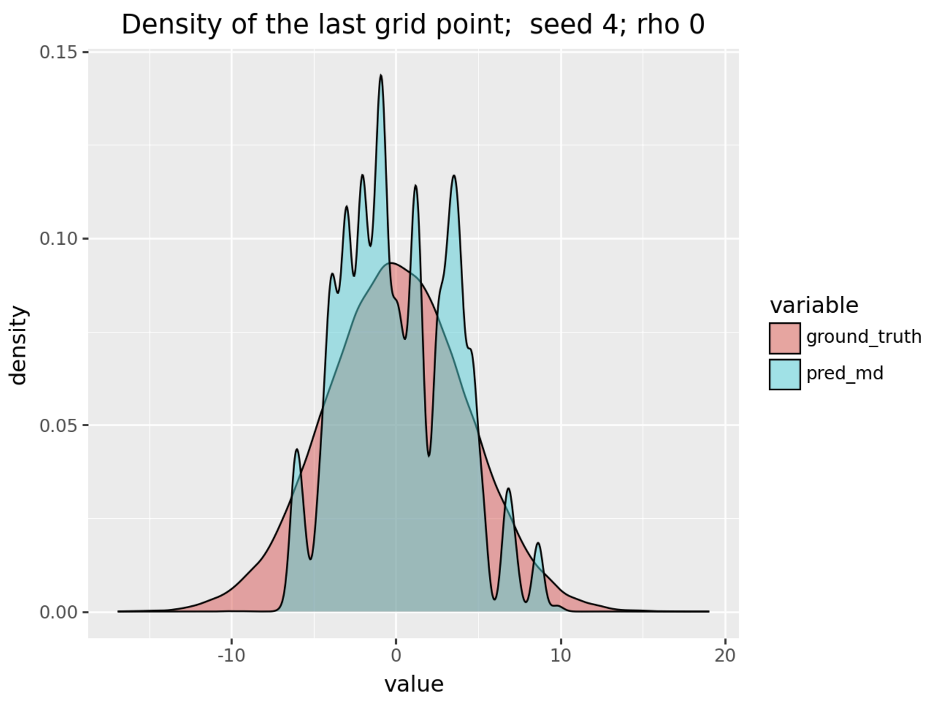
We examine the fit at the outermost grid points across multiple seeds by comparing the density of the ground truth with the model predictions.

In [26]:
```python
def plot_density_grid_points(seed=1, rho=0, N=100_000, show_metrics=True,
                mode='product', n_bins=100, x='x_1',
                list_x_vars=["x_1", "x_2"], list_y_var=["y"], iloc_index_grid = -1):

    df_x = simulate_data(rho=rho, N=N, seed=seed)


    quantiles_x_grid = np.append(0, np.arange(1 / n_bins, 1 + 1 / n_bins, 1 / n_bins))
    x_grid = df_x[x].quantile(quantiles_x_grid)

    # initializing Ground Truth
    reg_gt = GroundTruth(mode=mode)

    # predict ground truth
    y = reg_gt.predict(df_x)

    df = df_x.copy()
    df['y'] = y

    list_x_vars = ["x_1", "x_2"]
    list_y_var = ["y"]

    # initialize ML Model
    reg_md = GradientBoostingRegressor(random_state=0)


    # train test split
    last_row_train = math.floor(0.8*N)
    df_train = df[:last_row_train].copy()
    df_test = df[last_row_train:].copy()

    # ## Estimate Regressors
    reg_md.fit(df_train[list_x_vars], np.ravel(df_train[list_y_var]))

    if show_metrics:
        df_test['prediction'] = reg_md.predict(df_test[list_x_vars])
        print("MSE " + str(mean_squared_error(df_test[list_y_var], df_test['prediction'])))
        print("R2 " + str(r2_score(df_test[list_y_var], df_test['prediction'])))


    # plot last quantile
    df_predict = df_train[['x_1', 'x_2']].copy()
    df_predict['x_1'] = x_grid.iloc[iloc_index_grid]


    df_predict['pred_md'] = reg_md.predict(df_predict[['x_1', 'x_2']])
    df_predict['ground_truth'] = reg_gt.predict(df_predict[['x_1', 'x_2']])


    if iloc_index_grid == -1:
        string_plot_title = "last"
    elif iloc_index_grid == -2:
        string_plot_title = "last but one"
    elif iloc_index_grid == 0:
        string_plot_title = "first"
    elif iloc_index_grid == 1:
        string_plot_title = "second"
    else:
        string_plot_title = f"{iloc_index_grid}"

    df_predict_plot = pd.melt(df_predict[['pred_md', 'ground_truth']])
    p =(p9.ggplot(df_predict_plot, p9.aes(x='value', fill='variable'))
    + p9.geom_density(alpha=0.5)
    + p9.ggtitle(f"Density of the {string_plot_title} grid point;  seed {seed}; rho {rho}"))
    p.show()
```

We begin with the last grid point of seed 4, which was used in the simulation study. In comparison to the results of seed 2 (green, last grid point) and seed 1 (red, first grid point) shown in the plot above, we anticipate a reasonably good fit between the model and the ground truth.

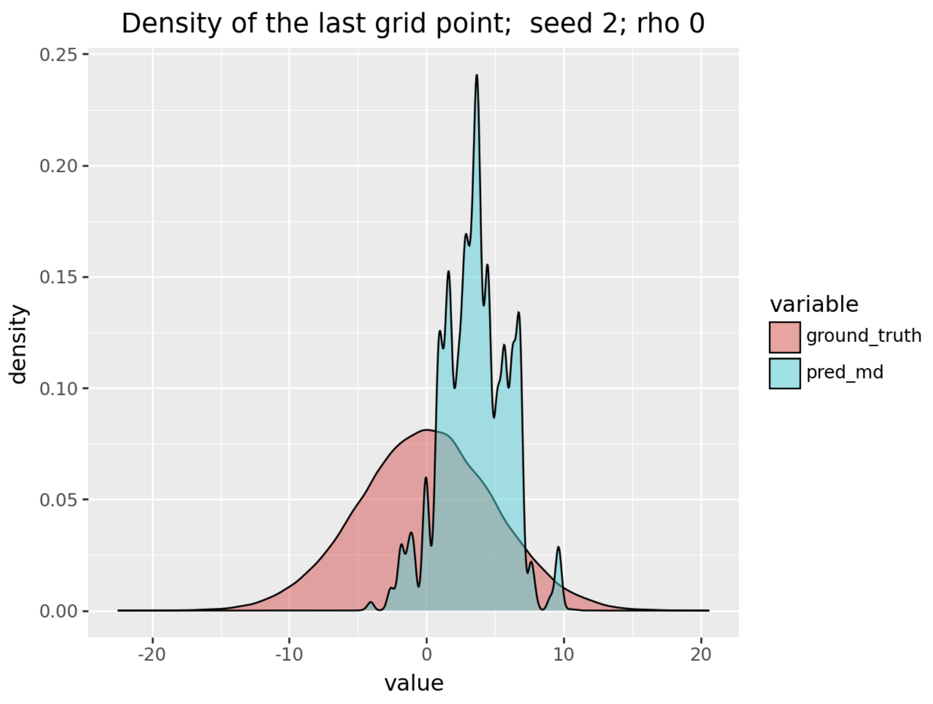
In [27]:
```python
plot_density_grid_points(seed=4, rho=0)
```

```
MSE 0.0438616367942629
R2 0.9568270995587546
```



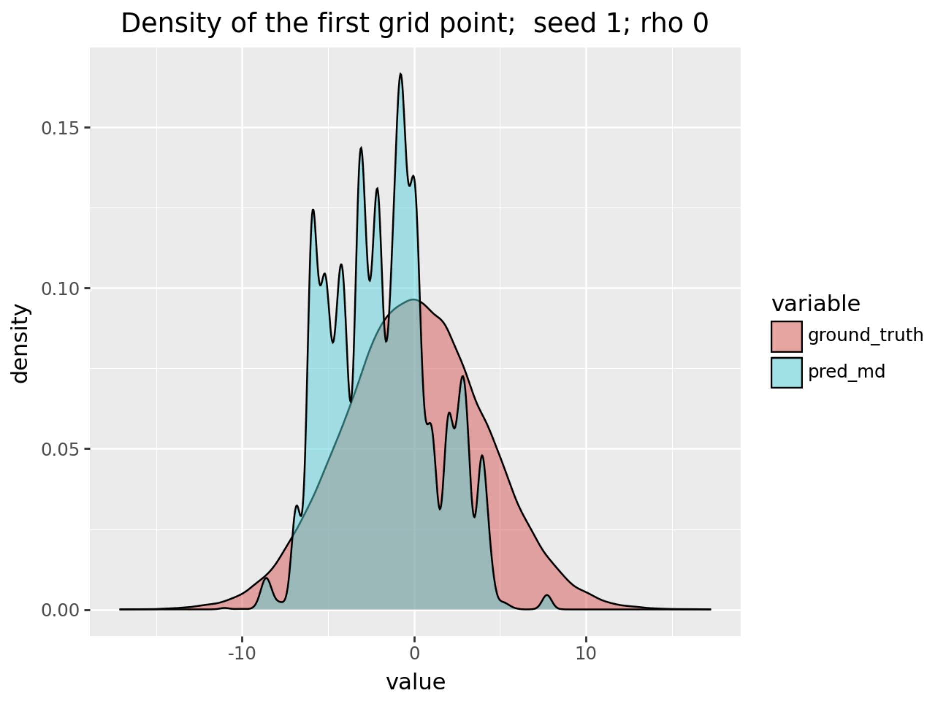
```
In [28]: plot_density_grid_points(seed=2, rho=0)
```

```
MSE 0.02955160622614025
R2 0.9707964716411317
```



```
In [29]: plot_density_grid_points(seed=1, rho=0, iloc_index_grid=0)
```

```
MSE 0.08753847734680682
R2 0.9120912517808025
```

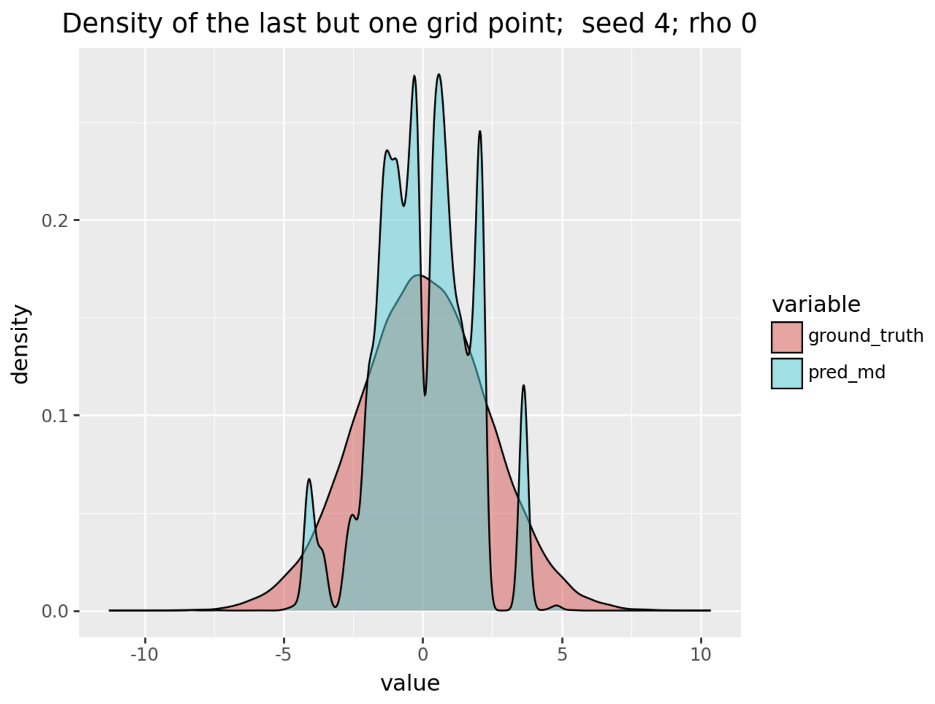


For seed 4, the density of the predictions and the ground truth indeed appears to be centered around the same mean. In contrast, for seeds 2 and 1, we observe a clear deviation of the predictions from the ground truth.

This indicates a poor model fit at the extreme grid points for those models with significant deviations. However, as shown in the following three plots, even the next grid points display a much better fit. Additionally, note that the R^2 values (shown above each plot) remain relatively high for all three seeds, although slightly lower for seed 1, where we observed large deviations at both the first and last grid points. Hence, these significant deviations can be attributed to poor model fit in sparse regions of the feature space

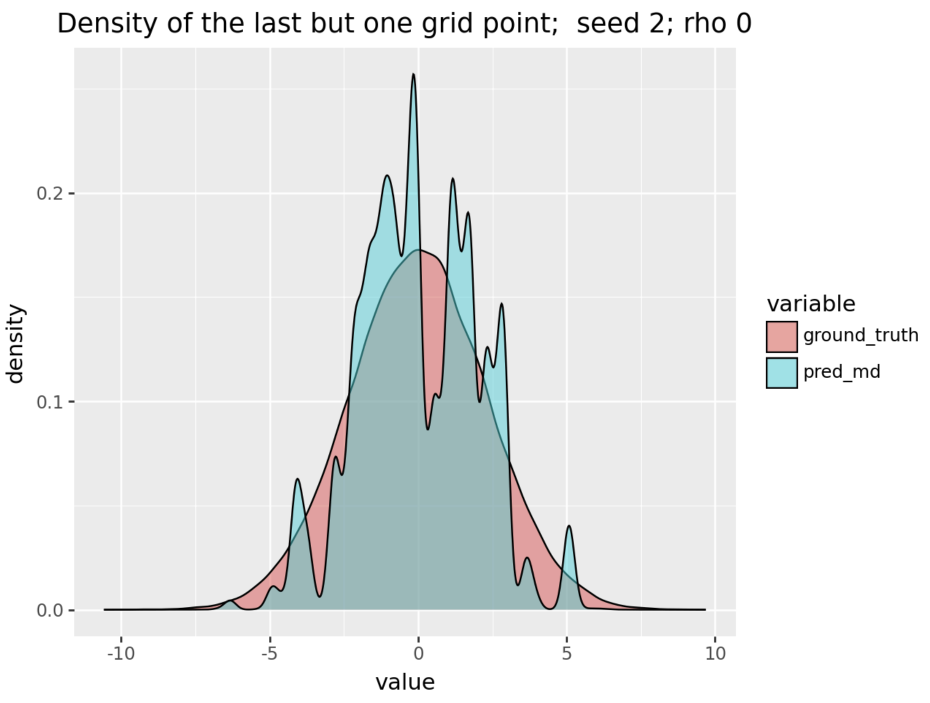
```
In [30]: plot_density_grid_points(seed=4, rho=0, iloc_index_grid=-2)
```

```
MSE 0.0438616367942629
R2 0.9568270995587546
```



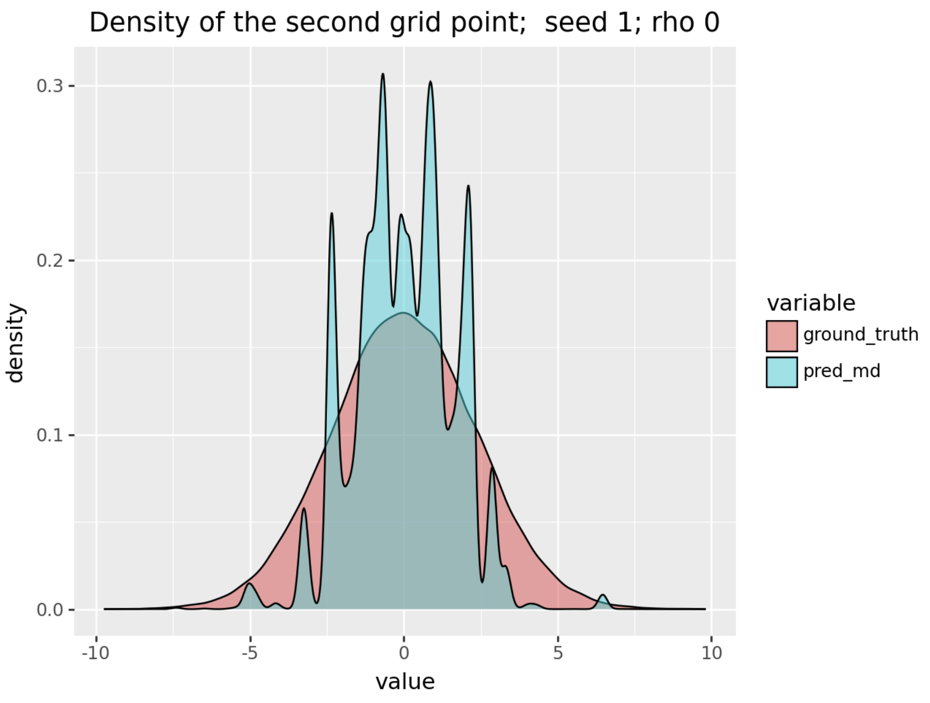
```
In [31]: plot_density_grid_points(seed=2, rho=0, iloc_index_grid=-2)
```

```
MSE 0.02955160622614025
R2 0.9707964716411317
```



```
In [32]: plot_density_grid_points(seed=1, rho=0, iloc_index_grid=1)
```

```
MSE 0.08753847734680682
R2 0.9120912517808025
```


Density of the second grid point; seed 1; rho 0
density
value
variable
ground_truth
pred_md