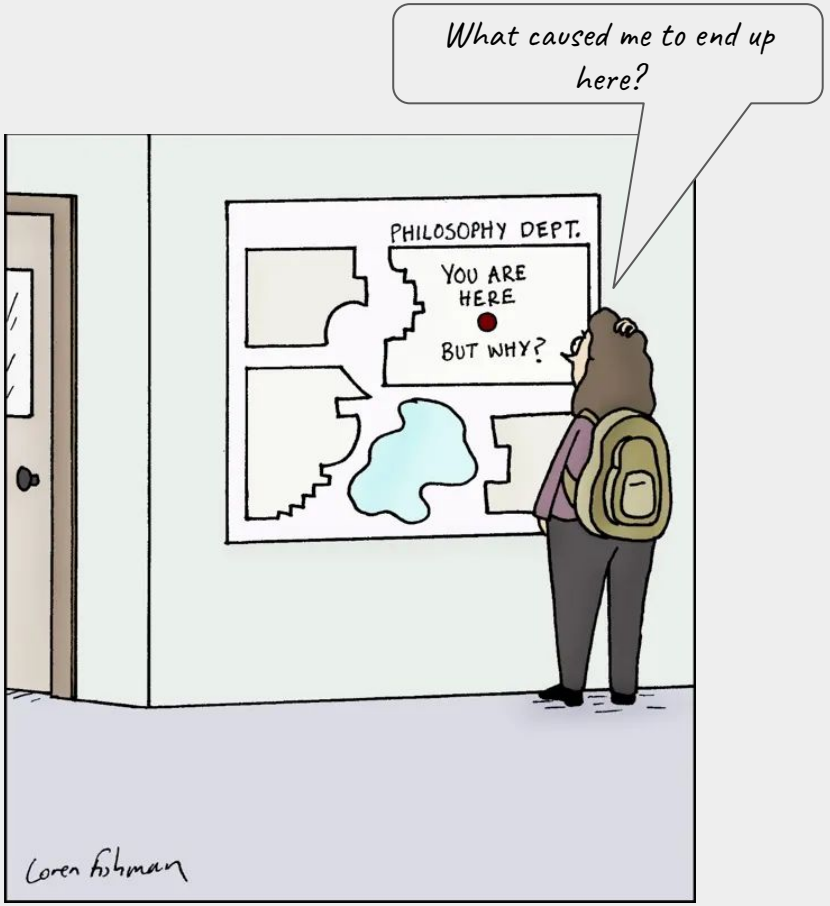
Explainable AI - Lecture 6

Partial Dependence Plots and Accumulated Local Effects

norwegian open ai lab
NTNU

Before we start...

What do SHAP explanations tell us?



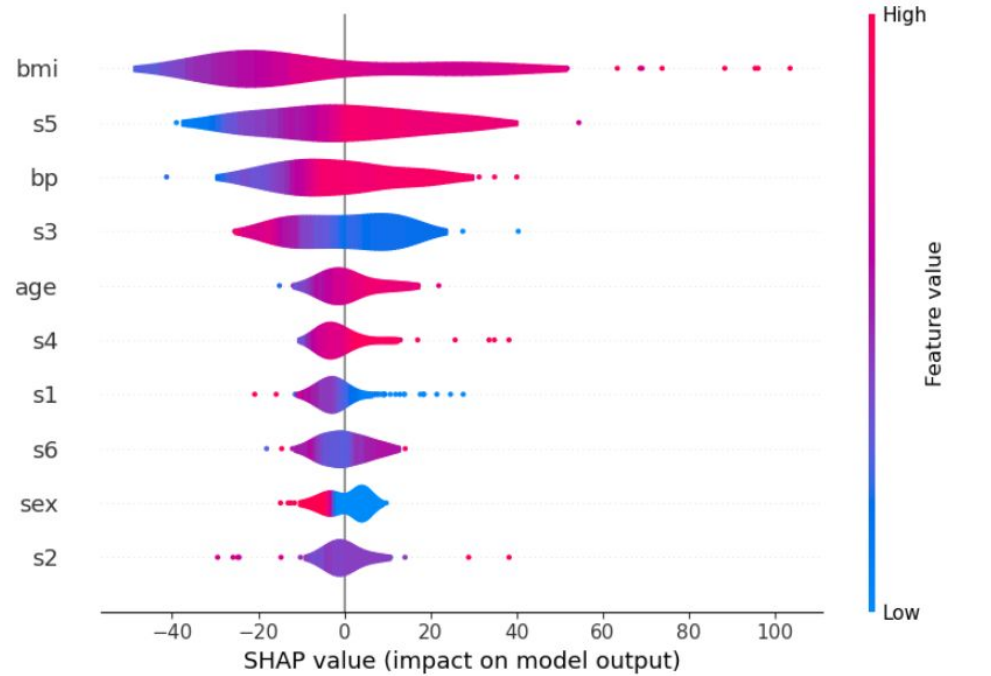
SHAP tells us...

SHAP tells us...

How the different features drive the prediction away from the base value.

SHAP can also tell us...

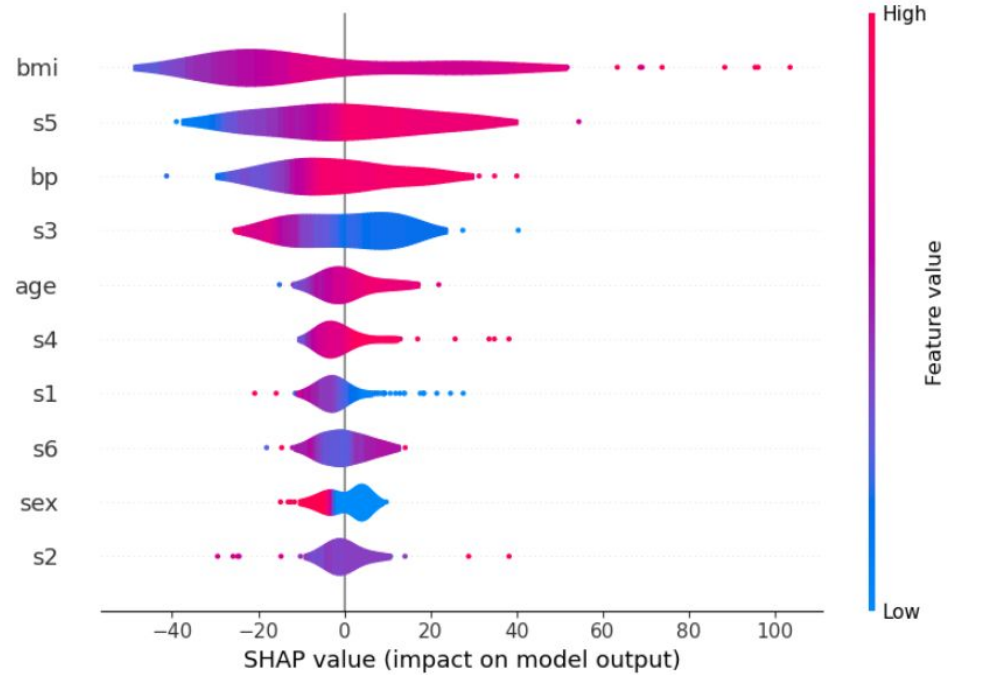
```
shap.summary_plot(shap_values, x_test, plot_type="violin")
```



SHAP can also tell us...

The trend of how the different features drive the prediction away from the base value, for an entire dataset.

```
shap.summary_plot(shap_values, x_test, plot_type="violin")
```



Feature importance attribution

SHAP and LIME tell us how important the model's input features are for the prediction, *given a data point.*

These are model-agnostic *local* explanation methods, since they produce an explanation *per data point.*

If we want to explain a model overall, we need *global* explanation methods.

Today, we'll study two methods for generating global model agnostic feature importance attribution.

Global feature importance attribution

Our goal is: by only probing the model, describe how its input features affect the model's predictions.

We will study two methods, Partial Dependence Plots (PDP) Accumulated Local Effects (ALE).

Both answer the question "how much does each feature affect model predictions on average?"

Today's mission:

Only probing the model, find out how the input features affect the predictions.

Partial Dependence Plots

The partial dependence function

We start by choosing a set S of features we want to explain, typically one or two: this is x_S

The rest of the features belong to the set C of features that we don't explain: this is X_C . Notice the use the notation for random variables.

Together, the sets S and C contain all the model's input features.

The partial dependence function

We start by choosing a set S of features we want to explain, typically one or two: this is x_S

The rest of the features belong to the set C of features that we don't explain: this is X_C . Notice the use the notation for random variables.

Together, the sets S and C contain all the model's input features.

The partial dependence function for the model f is defined as

$$PD_S(\mathbf{x}_S) = E_{X_C}\left[f(\mathbf{x}_S, X_C)\right] = \int f(\mathbf{x}_S, X_C)\, dP(X_C)$$

How is this a function that depends only on features in S? I.e. why doesn't it depend on X_C ?

The partial dependence function

We start by choosing a set S of features we want to explain, typically one or two: this is x_S

The rest of the features belong to the set C of features that we don't explain: this is X_C. Notice the use the notation for random variables.

Together, the sets S and C contain all the model's input features.

The partial dependence function for the model f is defined as

$$PD_S(\mathbf{x}_S) = E_{X_C}\left[f(\mathbf{x}_S, X_C)\right] = \int f(\mathbf{x}_S, X_C)\, dP(X_C)$$

By calculating the expectation over X_C, i.e. marginalising over the distribution of X_C, we get a function that depends only on features in S.

Estimating the partial dependence function

$x = (x_1, x_2, ..., x_d)$: data features

S: the features of interest

$C = \{1,2,...,d\} \setminus S$: the complement of the features in S.

How can we calculate the partial dependence function?

$$PD_S(\mathbf{x}_S) = E_{X_C}\left[f(\mathbf{x}_S, X_C)\right] \approx \frac{1}{N}\sum_{i=1}^{N} f(\mathbf{x}_S, \mathbf{x}_C^{(i)})$$

Estimating the partial dependence function

$x = (x_1, x_2, ..., x_d)$: data features

S: the features of interest

$C = \{1,2,...,d\} \setminus S$: the complement of the features in S.

We can estimate the expectation using the Monte Carlo method, meaning random sampling from a data set describing the features:

$$PD_S(\mathbf{x}_S) = E_{X_C}\left[f(\mathbf{x}_S, X_C)\right] \approx \frac{1}{N}\sum_{i=1}^{N} f(\mathbf{x}_S, \mathbf{x}_C^{(i)})$$

Notice that the sum is over different data points' values for the features $C \Leftarrow$ we're marginalising over X_C.

The partial dependence function is a function of S, so in order to evaluate the function, we need to insert values for x_S.

Estimating the PD function for one binary feature

Using the Loan data as example, we can choose as S the binary feature 'married', $x_S = \{0,1\}$.

```
with open("../data/LoanApprovalPrediction.csv", "r") as file:
    data_frame = pd.read_csv(file)
# Remove all rows with nan values
data_frame = data_frame.dropna()
```

```
print(len(data_frame.columns))
```

```
13
```

How many features are there in the set C?

Estimating the PD function for one binary feature

Using the Loan data as example, we can choose as S the binary feature 'married', $x_S = \{0,1\}$.

There are 11 remaining features, belonging to the set C.

For each possible value of x_S , we now have to average over all values in the set C.

$$PD_S(\mathbf{x}_S) = E_{X_C}\left[f(\mathbf{x}_S, X_C)\right] \approx \frac{1}{N}\sum_{i=1}^{N} f(\mathbf{x}_S, \mathbf{x}_C^{(i)})$$

If there are N=600 data points in total, how many data points do we have to evaluate?

Estimating the PD function for one binary feature

Using the Loan data as example, we can choose as S the binary feature 'married', $x_S = \{0,1\}$.

There are 11 remaining features, belonging to the set C.

For each possible value of x_S , we now have to average over all values in the set C.

$$PD_S(\mathbf{x}_S) = E_{X_C}\left[f(\mathbf{x}_S, X_C)\right] \approx \frac{1}{N}\sum_{i=1}^{N} f(\mathbf{x}_S, \mathbf{x}_C^{(i)})$$

If there are N=600 data points in total, we thus end up with a dataset for the PD estimate of 2x600=1200 data points.

Let's look at an example for illustration:

Estimating the PD function for one binary feature

S = feature 'married', possible values x_S={0,1} C = remaining features N = 600 data points

married	gender	dependents	...	income	loan amount
0	1	3		5849	108
0	0	1		4583	66
0	1	2		2583	120
... (until row 600)					
1	1	3		5849	108
1	0	1		4583	66
1	1	2		2583	120
... (until row 600)	...	...	...	...	...

$x_1 = 0$

$x_1 = 1$

$$PD_S(\mathbf{x}_S) = E_{X_C}\left[f(\mathbf{x}_S, X_C)\right] \approx \frac{1}{N}\sum_{i=1}^{N} f(\mathbf{x}_S, \mathbf{x}_C^{(i)})$$

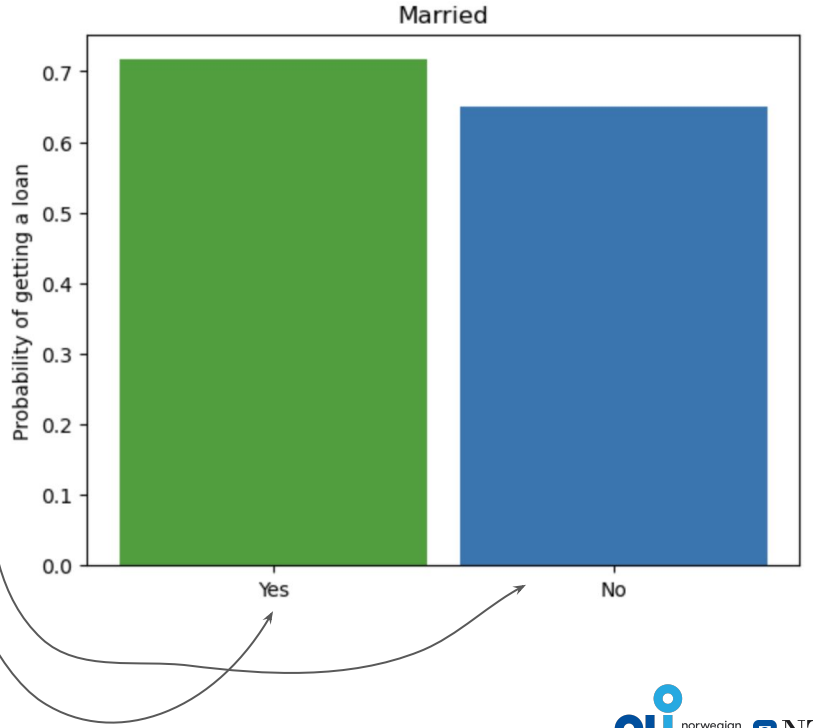
Estimating the PD function for one binary feature

S = feature 'married', possible values x_S={0,1}

C = remaining features

N = 600 data points

married	gender	dependents	...	income	loan amount	pred
0	1	3		5849	108	0.75
0	0	1		4583	66	0.7
0	1	2		2583	120	0.55
...						
1	1	3		5849	108	0.75
1	0	1		4583	66	0.7
1	1	2		2583	120	0.55
...	...	...	...	...	...	



$$PD_S(\mathbf{x}_S) = E_{X_C}\left[f(\mathbf{x}_S, X_C)\right] \approx \frac{1}{N}\sum_{i=1}^{N} f(\mathbf{x}_S, \mathbf{x}_C^{(i)})$$

Estimating the PD function for one categorical feature

S = feature 'property_area', possible values x_S={0,1,2} C = remaining features N = 600 data points

property area	gender	dependents	...	income	loan amount	pred
0	1	3		5849	108	0.75
0	0	1		4583	66	0.7
...						
1	1	3		5849	108	0.75
1	0	1		4583	66	0.7
2	1	3		5849	108	0.75
2	0	1		4583	66	0.7
...	...	...	...	...	...	

How many rows does this augmented dataset have?

$$PD_S(\mathbf{x}_S) = E_{X_C}\left[f(\mathbf{x}_S, X_C)\right] \approx \frac{1}{N}\sum_{i=1}^{N} f(\mathbf{x}_S, \mathbf{x}_C^{(i)})$$

Estimating the PD function for one categorical feature

S = feature 'property_area', possible values x_S={0,1,2} C = remaining features N = 600 data points

property area	gender	dependents	...	income	loan amount	pred
0	1	3		5849	108	0.75
0	0	1		4583	66	0.7
...						
1	1	3		5849	108	0.75
1	0	1		4583	66	0.7
2	1	3		5849	108	0.75
2	0	1		4583	66	0.7
...	...	...	...	...	...	

This augmented dataset has 1800 rows

Get data and preprocess

This code is provided in a notebook

As always:

Handle and preprocess categorical and continuous features.

Get the target column.

Split into training and test data.

Here: Make a torch loader, since we'll be using a pytorch model.

```
with open("data/LoanApprovalPrediction.csv", "r") as file:
    data_frame = pd.read_csv(file)
# Remove all rows with nan values
data_frame = data_frame.dropna()
```

```
categorical_feature_names= ["Gender", "Married", "Education", "Self_Employed", "Property_Area", "Credit_History", "Dependents"]
continuous_feature_names= ["ApplicantIncome", "CoapplicantIncome", "LoanAmount", "Loan_Amount_Term"]
drop_column = "Loan_ID" # Not included as the value is unique for each row
label_column = "Loan_Status"
```

```
categories_per_feature = [
    sorted(data_frame[col].unique().tolist()) for col in categorical_feature_names]
ohencoder = OneHotEncoder(categories=categories_per_feature)
scaler = StandardScaler()
labelencoder = LabelEncoder()
```

```
def preprocess_data(df: pd.DataFrame, continuous_feature_names: list[str], categorical_feature_names: list[str], label_column=str) -> tuple
    """
    Parameters:
        df (pd.DataFrame): the data in pandas dataframe format
        continuous_feature_names (list[str]): list of the feature names where the feature is continuous
        categorical_feature_names (list[str]): list of the feature names where the feature is categorical
        label_column (str): name of the label
    Returns:
        tuple[pd.DataFrame, pd.Dataframe]: preprocessed feature values and preprocessed labels
    """
    # Fit data, using a scaler for continuous data, oheencoder for categorical data and a labelencoder for the target values (label)
    continuous_features = scaler.transform(df[continuous_feature_names])
    categorical_features = ohencoder.transform(df[categorical_feature_names]).toarray()
    labels = labelencoder.transform(df[label_column])

    # Convert to tensors and using the available device
    continuous_features = torch.tensor(continuous_features, dtype=torch.float32, device=device)
    categorical_features = torch.tensor(categorical_features, dtype=torch.float32, device=device)
    labels = torch.tensor(labels, dtype=torch.float32, device=device)
    preprocessed_features = torch.hstack((continuous_features, categorical_features))

    return preprocessed_features, labels
```

```
df = data_frame.copy(deep=True) # we will use the raw data later and do not want to modify in place
scaler.fit(df[continuous_feature_names])
ohencoder.fit(df[categorical_feature_names])
labelencoder.fit(df[label_column])
preprocessed_data, labels = preprocess_data(df, continuous_feature_names, categorical_feature_names, label_column)

X_train, X_test, y_train, y_test = train_test_split(preprocessed_data, labels, random_state=42, test_size=0.2)
train_dataset = TensorDataset(X_train, y_train)
train_loader = DataLoader(train_dataset, batch_size=32)
```

Create and train a model

The notebook contains code for making a small multilayer perceptron model.

Here, the model is trained for one epoch, which is sufficient for getting an acceptable accuracy.

You can switch the model for something else, like a different architecture, or a boosted decision tree, whatever :)

```python
class Network(nn.Module):
    def __init__(self, input_dim: int):
        super().__init__()
        self.f1 = nn.Linear(input_dim, 64)
        self.f2 = nn.Linear(64, 1)

    def forward(self, x: torch.Tensor):
        x = F.relu(self.f1(x))
        x = self.f2(x)
        return x
```

```python
def train_one_epoch(train_loader, model):
    model.train()
    for X, y in train_loader:
        optimizer.zero_grad()
        outputs = model(X).squeeze(-1)
        loss = loss_fn(outputs, y)
        loss.backward()
        optimizer.step()

    model.eval()
    with torch.no_grad():
        outputs = model(X_test).squeeze(-1)
        loss = loss_fn(outputs, y_test)
        preds = torch.sigmoid(outputs)
        accuracy = sum((preds > 0.5) == y_test)/len(y_test)
    return loss, accuracy
```

Create the upsampled data

Given a dataframe, a feature of interest, and values the feature can take:

Create one copy of the data frame per value of the feature, replacing the feature value with that value.

How many data frames do we get here?

How many data points in total for the upsampled data?

```python
def create_modified_df(df: pd.DataFrame, feature: str, values: list) -> [dict[str, pd.DataFrame]]:
    """
    Parameters:
        df (pd.DataFrame): data
        feature (str): name of the feature that should be modified
        values (list): the possible values of the feature
    Returns:
        dict[str, pd.DataFrame]: set of dataframes where the the rows of the feature are set to the same value.
    """
    dfs = {}
    for value in values:
        df.loc[:, feature] = value
        dfs[value] = df.copy(deep=True)
    return dfs
```

```python
feature_name = "Married"
categories = set(df[feature_name].values.flatten())
mod_dfs = create_modified_df(data_frame.copy(deep=True), feature_name, categories)

print("Original number of data points:", df.shape[0])
print(f"Possible values for the feature {feature_name}: {categories}")
print("Number of dataframes for upsampled data:", len(mod_dfs))
print("Total number of data points:", [sum(len(mod_dfs[_key]) for _key in mod_dfs.keys())][0])
```

Create the upsampled data

Given a dataframe, a feature of interest, and values the feature can take:

Create one copy of the data frame per value of the feature, replacing the feature value with that value.

We get as many data frames as there are unique values of the feature, each containing the original number of data points.

```python
def create_modified_df(df: pd.DataFrame, feature: str, values: list) -> [dict[str, pd.DataFrame]]:
    """
    Parameters:
        df (pd.DataFrame): data
        feature (str): name of the feature that should be modified
        values (list): the possible values of the feature
    Returns:
        dict[str, pd.DataFrame]: set of dataframes where the the rows of the feature are set to the same value.
    """
    dfs = {}
    for value in values:
        df.loc[:, feature] = value
        dfs[value] = df.copy(deep=True)
    return dfs
```

```python
feature_name = "Married"
categories = set(df[feature_name].values.flatten())
mod_dfs = create_modified_df(data_frame.copy(deep=True), feature_name, categories)

print("Original number of data points:", df.shape[0])
print(f"Possible values for the feature {feature_name}: {categories}")
print("Number of dataframes for upsampled data:", len(mod_dfs))
print("Total number of data points:", [sum(len(mod_dfs[_key]) for _key in mod_dfs.keys())][0])
```

```
Original number of data points: 505
Possible values for the feature Married: {'Yes', 'No'}
Number of dataframes for upsampled data: 2
Total number of data points: 1010
```

Get the marginalised probabilities

```python
def get_marginalized_probability(df:pd.DataFrame, continuous_feature_names:list,
                                 categorical_feature_names:list, label_column:str)->float:
    """
    Parameters:
        df (pd.DataFrame): data
        continuous_feature_names (list): names of continuous feaures
        categorical_feature_names (list): names of categorical features
        label_column (str): header of the target column
    Returns:
        float: the mean sigmoid prediction for the dataframe.
    """
    preprocessed_data, labels = preprocess_data(df, continuous_feature_names, categorical_feature_names, label_column)

    outputs = model(preprocessed_data).squeeze(-1)
    preds = torch.sigmoid(outputs)
    return float(preds.mean().item())
```

Given a dataframe, and information about feature types and the target column, *what does this function do?*

Get the marginalised probabilities

```python
def get_marginalized_probability(df:pd.DataFrame, continuous_feature_names:list,
                                 categorical_feature_names:list, label_column:str)->float:
    """
    Parameters:
        df (pd.DataFrame): data
        continuous_feature_names (list): names of continuous feaures
        categorical_feature_names (list): names of categorical features
        label_column (str): header of the target column
    Returns:
        float: the mean sigmoid prediction for the dataframe.
    """
    preprocessed_data, labels = preprocess_data(df, continuous_feature_names, categorical_feature_names, label_column)

    outputs = model(preprocessed_data).squeeze(-1)
    preds = torch.sigmoid(outputs)
    return float(preds.mean().item())
```

Given a dataframe, and information about feature types and the target column, this function returns the mean of the model's sigmoid activated outputs (probabilities for the target).

Get the marginalised probabilities

```
probabilities= {}
for category in categories:
    df = mod_dfs[category].copy(deep=True)
    probability = get_marginalized_probability(df, continuous_feature_names, categorical_feature_names, label_column)
    probabilities[category] = probability
```

```
print(f"Marginalised probabilities per value of feature {feature_name}:")
print(probabilities)
```

For each value of the categorical feature, get the marginalized probability across the modified dataset.

For S = feature 'married', possible values x_S={0,1},

How many marginalized probabilities do we get?

Get the marginalised probabilities

```
probabilities= {}
for category in categories:
    df = mod_dfs[category].copy(deep=True)
    probability = get_marginalized_probability(df, continuous_feature_names, categorical_feature_names, label_column)
    probabilities[category] = probability
```

```
print(f"Marginalised probabilities per value of feature {feature_name}:")
print(probabilities)
```

```
Marginalised probabilities per value of feature Married:
{'Yes': 0.7166070938110352, 'No': 0.6497240662574768}
```

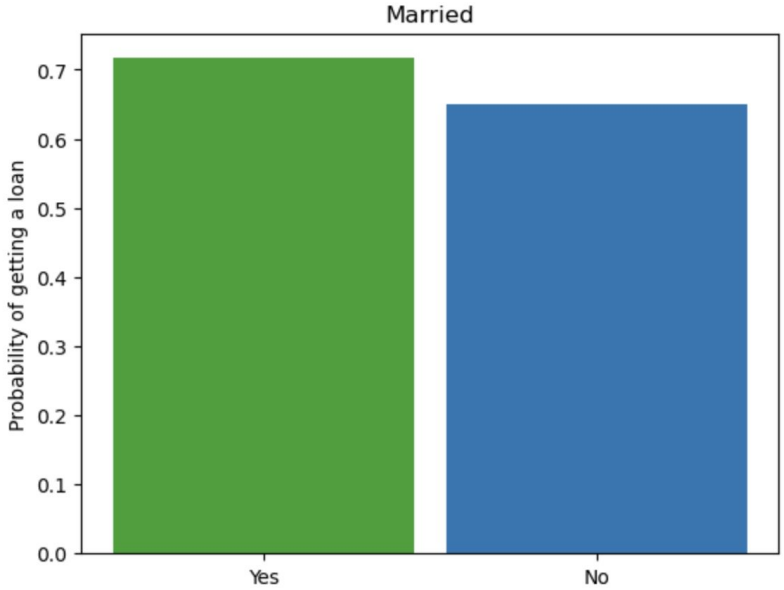
For each value of the categorical feature, get the marginalized probability across the modified dataset.

For S = feature 'married', possible values x_S={0,1}, we get two marginalized probabilities.

Plot the marginalised probabilities

```
y = probabilities.values()
x = probabilities.keys()
```

```
fig, ax = plt.subplots()
#ax.bar(x, y, width=0.9, linewidth=0.7, color=["tab:blue", "tab:green", "tab:orange"]) # property area
ax.bar(x, y, width=0.9, linewidth=0.7, color=["tab:green", "tab:blue"]) # married
plt.title(feature_name)
plt.ylabel("Probability of getting a loan")
plt.show()
```



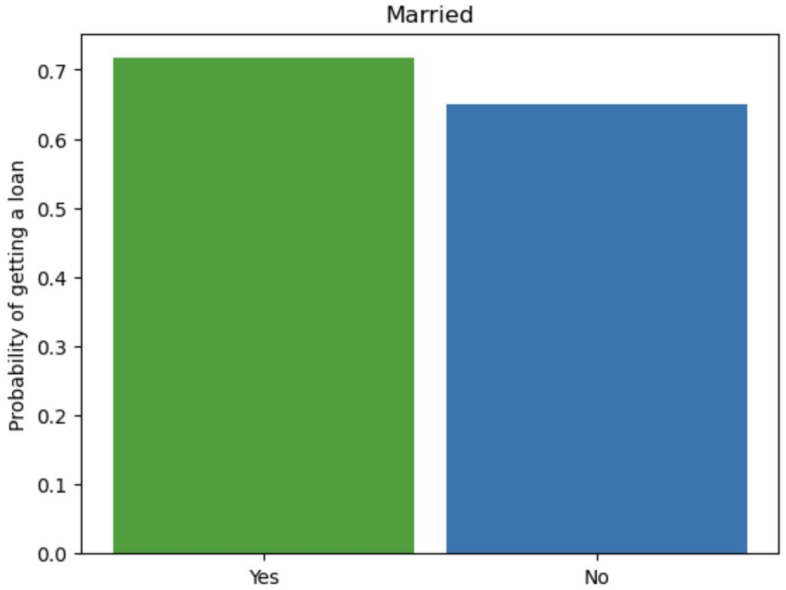
That's it! This is the partial dependence plot for the categorical feature married.

What do the axes show?

Plot the marginalised probabilities

```
y = probabilities.values()
x = probabilities.keys()
```

```
fig, ax = plt.subplots()
#ax.bar(x, y, width=0.9, linewidth=0.7, color=["tab:blue", "tab:green", "tab:orange"]) # property area
ax.bar(x, y, width=0.9, linewidth=0.7, color=["tab:green", "tab:blue"]) # married
plt.title(feature_name)
plt.ylabel("Probability of getting a loan")
plt.show()
```



That's it! This is the partial dependence plot for the categorical feature married.

We have the feature values on the *x* axis and the corresponding marginalised probabilities on the *y* axis.

Estimating the PD function for one continuous feature

Sticking to the Loan data, we choose as S the continuous feature LoanAmount, with values in the range x_S = (150, 81000).

The remaining features again belong to the set C.

Before, we averaged over all values in the features C. *How to proceed now that x_S has as many unique values as there are data points?*

$$PD_S(\mathbf{x}_S) = E_{X_C}\left[f(\mathbf{x}_S, X_C)\right] \approx \frac{1}{N}\sum_{i=1}^{N} f(\mathbf{x}_S, \mathbf{x}_C^{(i)})$$

Estimating the PD function for one continuous feature

Sticking to the Loan data, we choose as S the continuous feature LoanAmount, with values in the range $x_S = (150, 81000)$.

The remaining features again belong to the set C.

We divide x_S into n intervals, and evaluate $\text{PD}_S(x_S)$ for each value of x_S. The larger n we choose, the more computationally demanding the total calculation is, and the smoother curves we get.

$$PD_S(\mathbf{x}_S) = E_{X_C}\left[f(\mathbf{x}_S, X_C)\right] \approx \frac{1}{N}\sum_{i=1}^{N} f(\mathbf{x}_S, \mathbf{x}_C^{(i)})$$

We are free to choose the intervals however we like. Common choices are

- some spacing between the minimum and maximum values of x_S
- some spacing between the 5th and 95th percentiles of x_S
- …

Code :)

```
plot_data = dict()
n = 5

for feature_name in continuous_feature_names:
    df = data_frame.copy(deep=True)
    lower_bound = np.quantile(df[feature_name], 0.005)
    upper_bound = np.quantile(df[feature_name], 0.995)
    values = np.linspace(lower_bound, upper_bound, n, dtype=int)
    #values = np.linspace(df[feature_name].min(), df[feature_name].max(), n, dtype=int)
    mod_dfs = create_modified_df(df, feature_name, values)

    probabilities = []
    for value in values:
        probability = get_marginalized_probability(mod_dfs[value].copy(deep=True),
                                                   continuous_feature_names, categorical_feature_names, label_column)
        probabilities.append(probability)

    plot_data[feature_name] = (values, probabilities)
```

We can reuse the functions we made before. The only thing we have to do is decide on the intervals for the feature of interest, i.e. make the lower_bound, upper_bound and fill the values array.

np.linspace makes evenly (linearly) spaced intervals.

Code :)

```
plot_data = dict()
n = 5

for feature_name in continuous_feature_names:
    df = data_frame.copy(deep=True)
    lower_bound = np.quantile(df[feature_name], 0.005)
    upper_bound = np.quantile(df[feature_name], 0.995)
    values = np.linspace(lower_bound, upper_bound, n, dtype=int)
    #values = np.linspace(df[feature_name].min(), df[feature_name].max(), n, dtype=int)
    mod_dfs = create_modified_df(df, feature_name, values)

    probabilities = []
    for value in values:
        probability = get_marginalized_probability(mod_dfs[value].copy(deep=True),
                                                   continuous_feature_names, categorical_feature_names, label_column)
        probabilities.append(probability)

    plot_data[feature_name] = (values, probabilities)
```

Printing the values is not very useful, which is why we plot them instead.

```
print(f"Interval values for feature {feature}: {plot_data[feature][0]}")
print(f"Corresponding marginalized probabilities: {plot_data[feature][1]}")
```

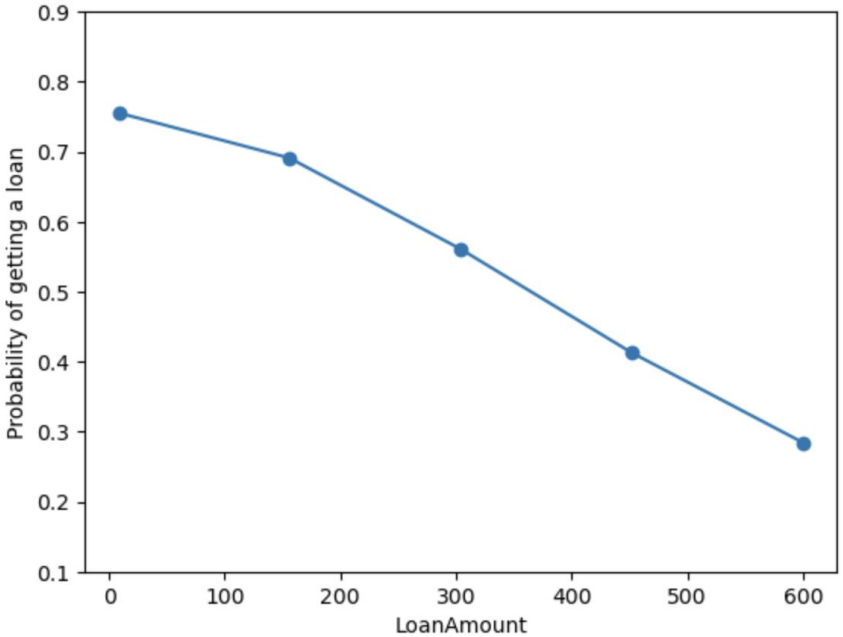
```
Interval values for feature LoanAmount: [ 25 152 279 406 533]
Corresponding marginalized probabilities: [0.751140296459198, 0.6936378479003906, 0.5855261087417603, 0.4578362703323364, 0.3388989865779
8767]
```

Estimating the PD function for one continuous feature

S = feature 'LoanAmount'

$n = 5$,

values = `np.linspace(df[feature_name].min(), df[feature_name].max(), n, dtype=int)`

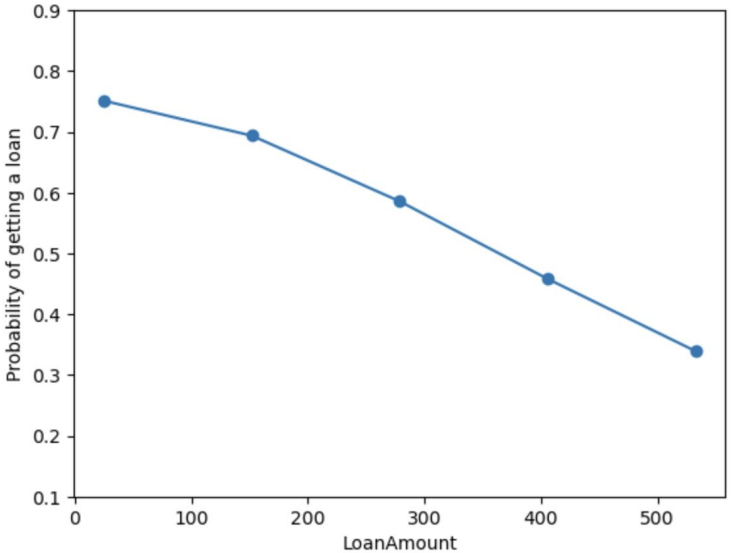


Estimating the PD function for one continuous feature

S = feature 'LoanAmount'

$n = 5$,

```
lower_bound = np.quantile(df[feature_name], 0.005)
upper_bound = np.quantile(df[feature_name], 0.995)
values = np.linspace(lower_bound, upper_bound, n, dtype=int)
```



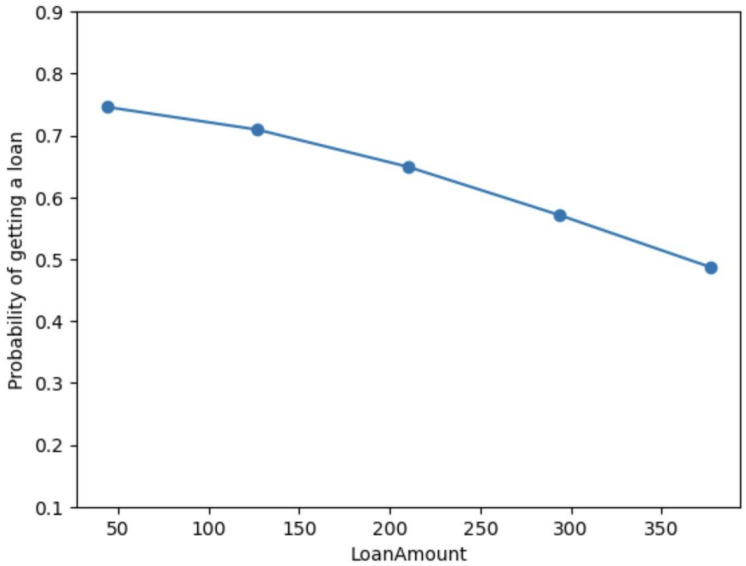
norwegian open ai lab NTNU

Estimating the PD function for one continuous feature

S = feature 'LoanAmount'

$n = 5$,

```
lower_bound = np.quantile(df[feature_name], 0.025)
upper_bound = np.quantile(df[feature_name], 0.975)
values = np.linspace(lower_bound, upper_bound, n, dtype=int)
```



Including a narrow part of the data distribution implies excluding extreme values.

What is good about this?

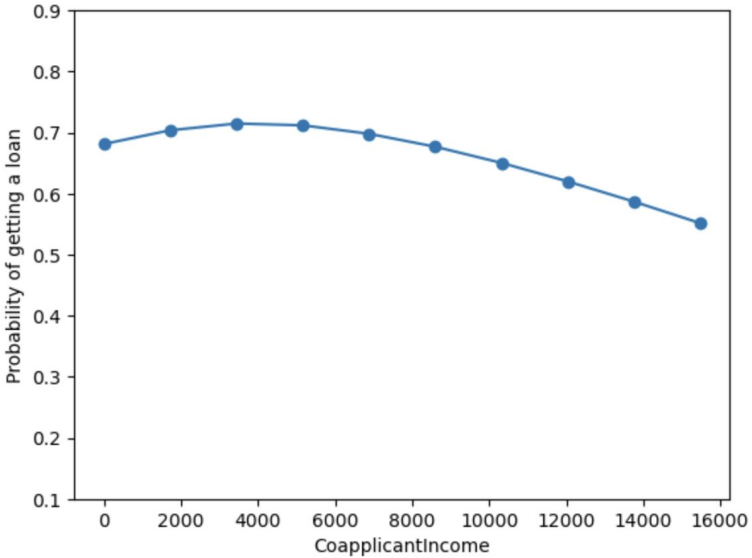
What is bad about this?

Estimating the PD function for one continuous feature

S = feature 'CoapplicantIncome'

n = 10,

```
lower_bound = np.quantile(df[feature_name], 0.005)
upper_bound = np.quantile(df[feature_name], 0.995)
values = np.linspace(lower_bound, upper_bound, n, dtype=int)
```



What do we see here?

Partial dependence plots (PDP)

In one sentence what does the PDP show?

$$PD_S(\mathbf{x}_S) = E_{X_C}\left[f(\mathbf{x}_S, X_C)\right] \approx \frac{1}{N}\sum_{i=1}^{N} f(\mathbf{x}_S, \mathbf{x}_C^{(i)})$$

Partial dependence plots (PDP)

The PDP shows the expected model prediction given values of the feature of interest.

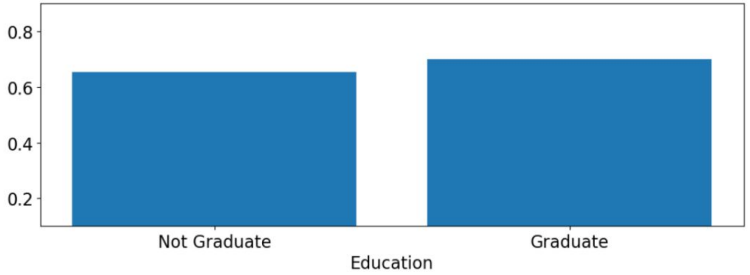
For categorical features, the PD function produces one value per category of S, often plotted as bars.

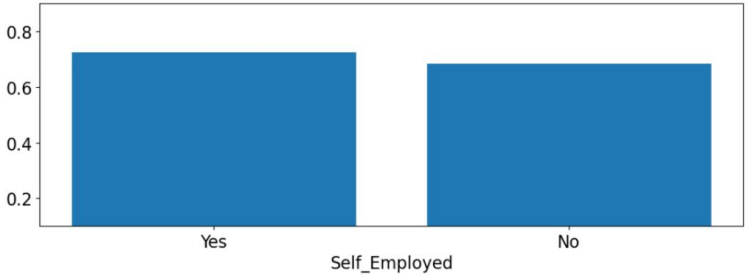
For continuous features, we have to choose intervals and the PD function produces one value per interval. When plotting, we interpolate between the intervals.

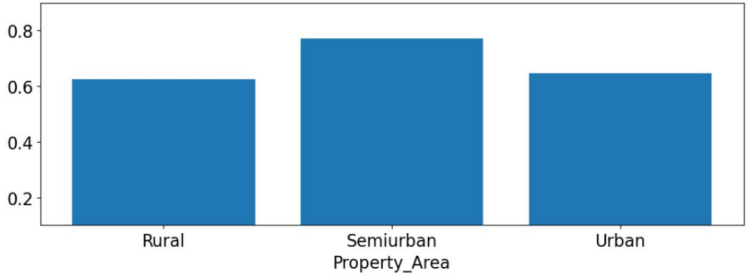
Since this is a plot based explanation method, it's best to stick to a number of features where the plots are interpretable, i.e. one or two features at a time.

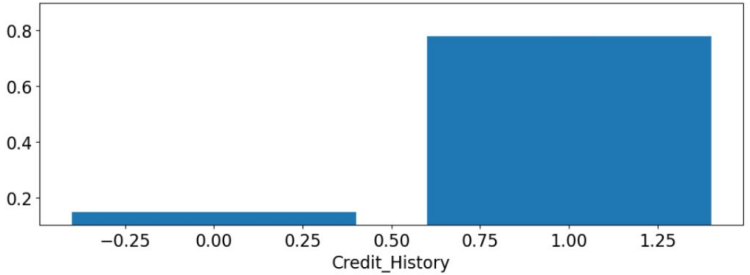
$$PD_S(\mathbf{x}_S) = E_{X_C}\left[f(\mathbf{x}_S, X_C)\right] \approx \frac{1}{N}\sum_{i=1}^{N} f(\mathbf{x}_S, \mathbf{x}_C^{(i)})$$

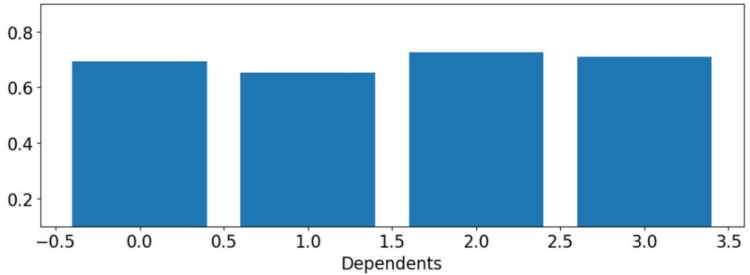
Interpreting partial dependence plots (PDP)







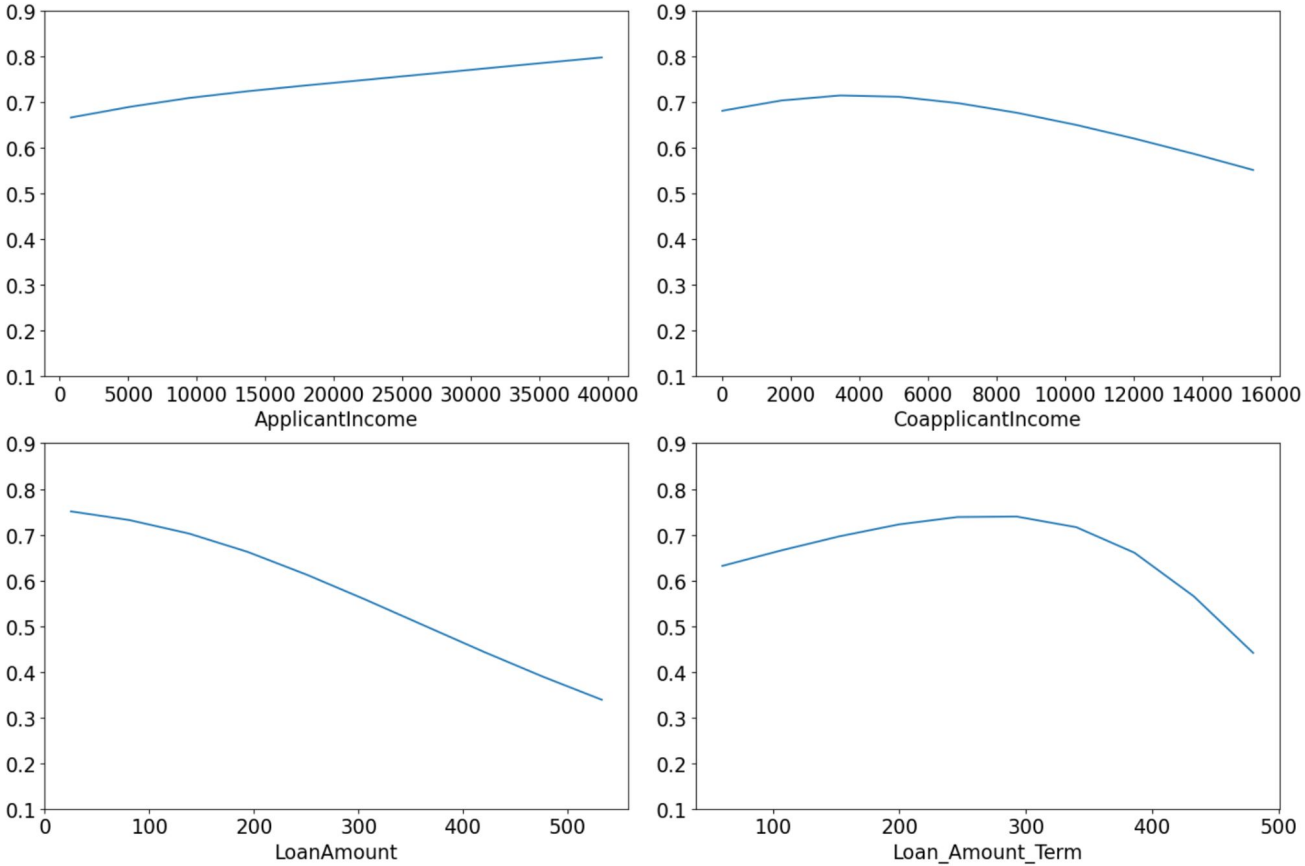




Of these, what is the most important feature for the model?

What does it mean when the bars in a PD plot are ~equal?

Interpreting partial dependence plots (PDP)



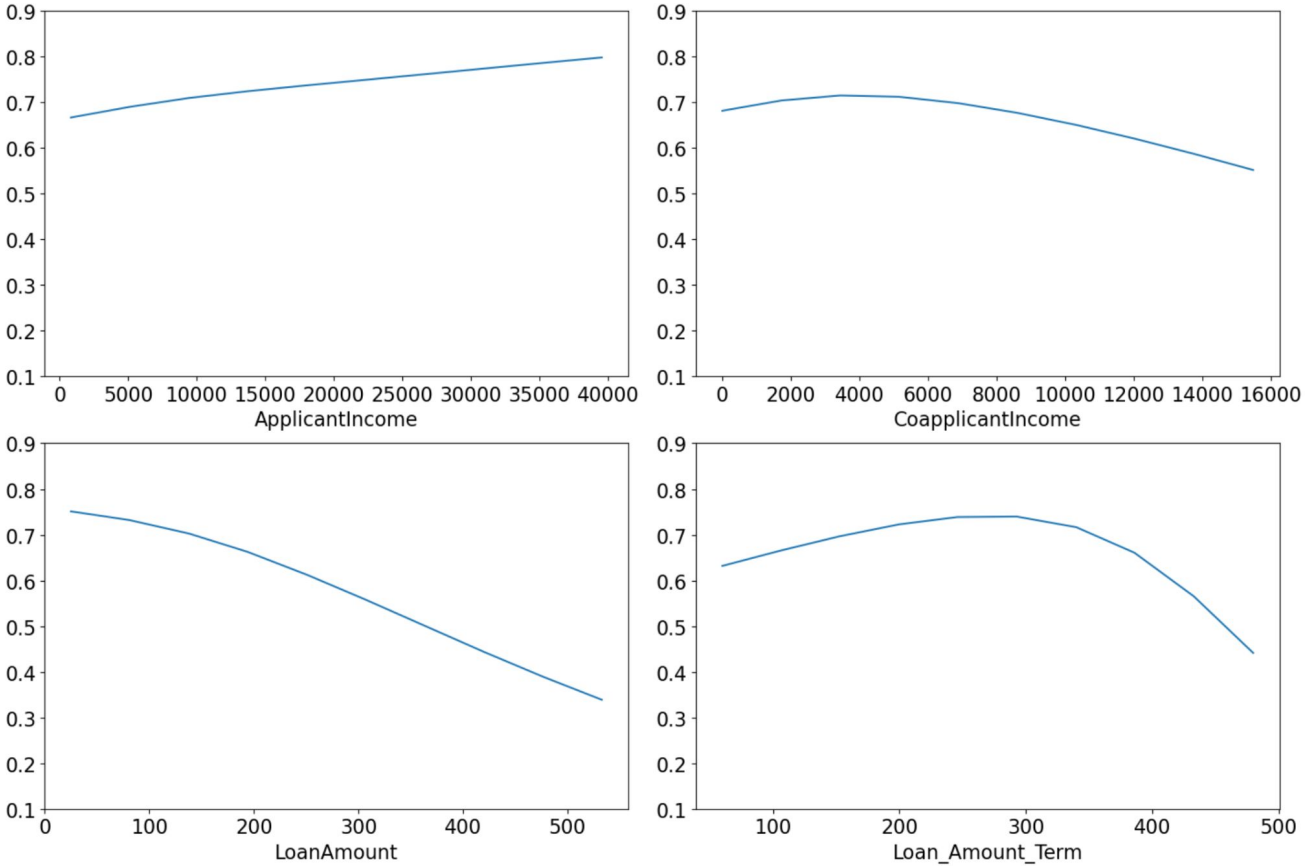
Feature importance for continuous features:

What does a flat curve indicate?

What does a steep curve indicate?

What else do we see?
(hint: regions)

Interpreting partial dependence plots (PDP)



Feature importance for continuous features:

A flat curve indicates an unimportant feature

A steep curve indicates a more important feature.

What else do we see?

Partial dependence plots

Where do we place this in the taxonomy?

Post-hoc?

Model-agnostic or model-specific?

Local or global?

Post-hoc methods	Model-agnostic	Model-specific
Local		
Global		

Partial dependence plots

Where do we place this in the taxonomy?

Post-hoc: *We got a model and want to explain its behaviour*

Model-agnostic or model-specific?

Local or global?

Post-hoc methods	Model-agnostic	Model-specific
Local		
Global		

Partial dependence plots

Where do we place this in the taxonomy?

Post-hoc: *We got a model and want to explain its behaviour*

Model-agnostic: *The inner structure of the model is not important; we care only about the inputs and outputs.*

Local or global?

Post-hoc methods	Model-agnostic	Model-specific
Local		
Global		

Partial dependence plots

Where do we place this in the taxonomy?

Post-hoc: *We got a model and want to explain its behaviour*

Model-agnostic: *The inner structure of the model is not important; we care only about the inputs and outputs.*

Global: *We average the predictions over the whole dataset per feature to get a global trend.*

Post-hoc methods	Model-agnostic	Model-specific
Local		
Global		

Partial dependence plots

Where do we place this in the taxonomy?

Post-hoc: *We got a model and want to explain its behaviour*

Model-agnostic: *The inner structure of the model is not important; we care only about the inputs and outputs.*

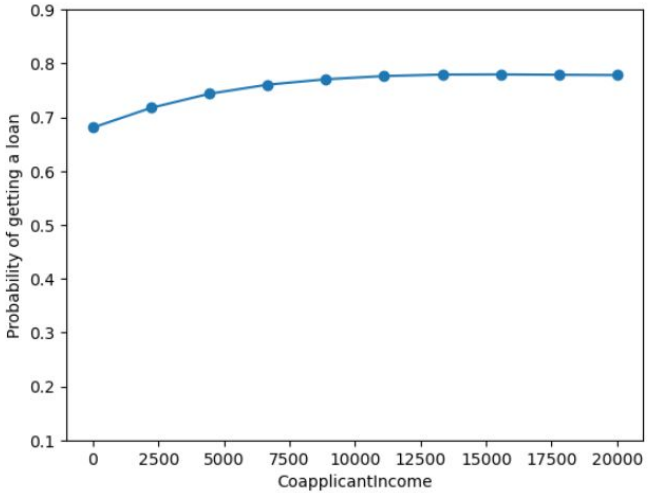
Global: *We average the predictions over the whole dataset per feature to get a global trend.*

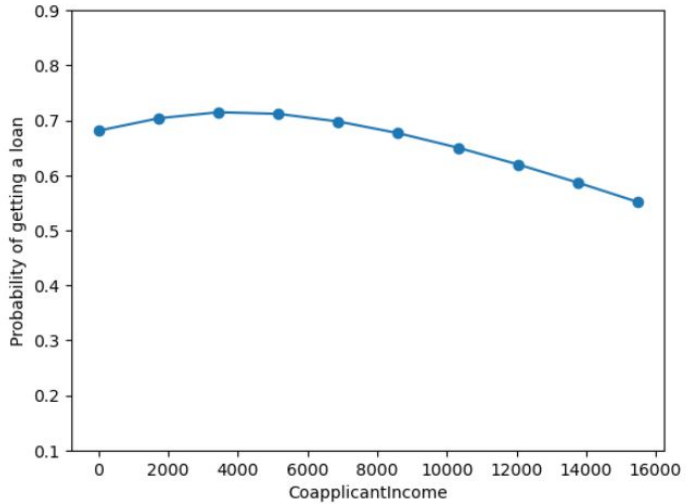
Post-hoc methods	Model-agnostic	Model-specific
Local		
Global	PDP	

Homework, part 1

Calculate PDP values for the feature CoapplicantIncome, and see if you can make the curve change trend just by changing the included range.

What does this tell us? What do you know about outliers in machine learning?





Partial dependence plots (PDP)

What does a partial dependence plot show? Use the word "marginal" in your answer.

$$PD_S(\mathbf{x}_S) = E_{X_C}\left[f(\mathbf{x}_S, X_C)\right] \approx \frac{1}{N}\sum_{i=1}^{N} f(\mathbf{x}_S, \mathbf{x}_C^{(i)})$$

Partial dependence plots (PDP)

A partial dependence plot shows the marginal effect one or two features have on the predicted outcome of a model.

Included features S are the features for which we generate PDPs.

Calculate the expected value over excluded features C, i.e. average model prediction with excluded feature values sampled from dataset.

What assumption does this imply?

$$PD_S(\mathbf{x}_S) = E_{X_C}\left[f(\mathbf{x}_S, X_C)\right] \approx \frac{1}{N}\sum_{i=1}^{N} f(\mathbf{x}_S, \mathbf{x}_C^{(i)})$$

Partial dependence plots (PDP)

A partial dependence plot shows the marginal effect one or two features have on the predicted outcome of a model.

Included features S are the features for which we generate PDPs.

Calculate the expected value over excluded features C, i.e. average model prediction with excluded feature values sampled from dataset.

This implies an assumption of no or weak correlation between features in S and C.

$$PD_S(\mathbf{x}_S) = E_{X_C}\left[f(\mathbf{x}_S, X_C)\right] \approx \frac{1}{N}\sum_{i=1}^{N} f(\mathbf{x}_S, \mathbf{x}_C^{(i)})$$

Accumulated Local Effects

What is the main weakness of partial dependence plots? When does this method really struggle?

Hint: What was our assumption along the way?

What is the main weakness of partial dependence plots? When does this method really struggle?

Hint: What was our assumption along the way?

PDPs aren't really useful when the features are correlated.

Highly correlated data for illustration

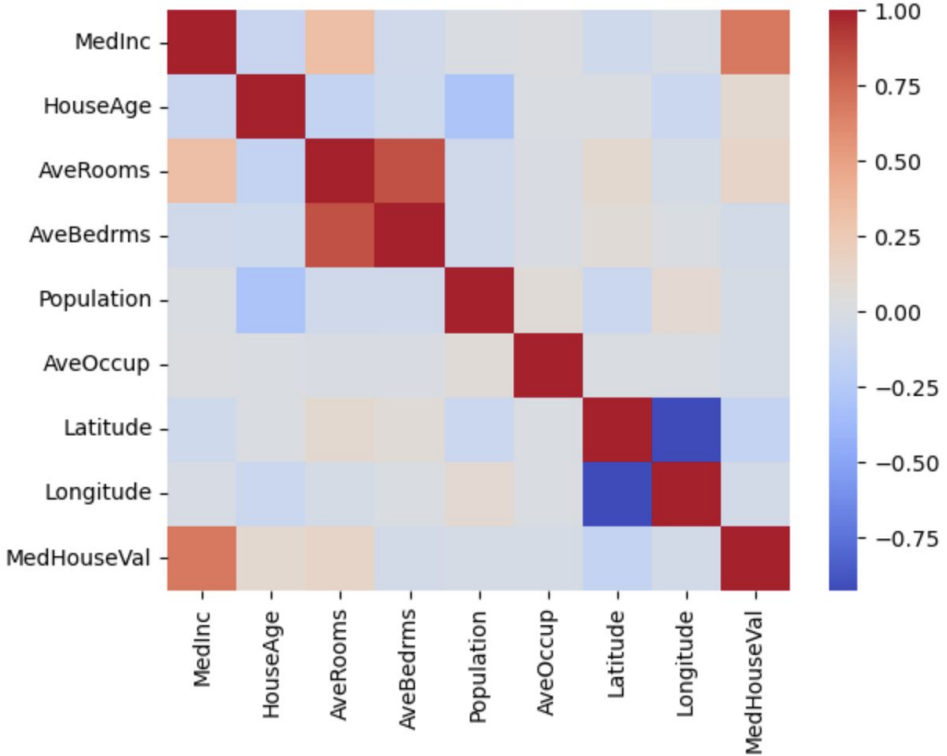
```
X, y = fetch_california_housing(as_frame=True, return_X_y=True)
train_df = pd.concat([X, y], axis=1).dropna()
X = train_df[X.columns]
y = train_df[y.name]
print("Target:", y.name)
X_train, X_test, y_train, y_test= train_test_split(X, y, test_size=0.2, random_state=42)
df = X.assign(MedHouseVal=y)
feature_names = X_train.columns.tolist()
```

```
Target: MedHouseVal
```

Our f now predicts `MedHouseVal`

Highly correlated data for illustration

```
sns.heatmap(df.corr(numeric_only=True), cmap="coolwarm")
plt.show()
```



Our f now predicts `MedHouseVal` based on these features, and their correlations.

What are the most highly correlated features here?

PDP on highly correlated data

Say we want to know the effect of the number of bedrooms (`AveBedrms`) on the predicted `MedHouseVal`.

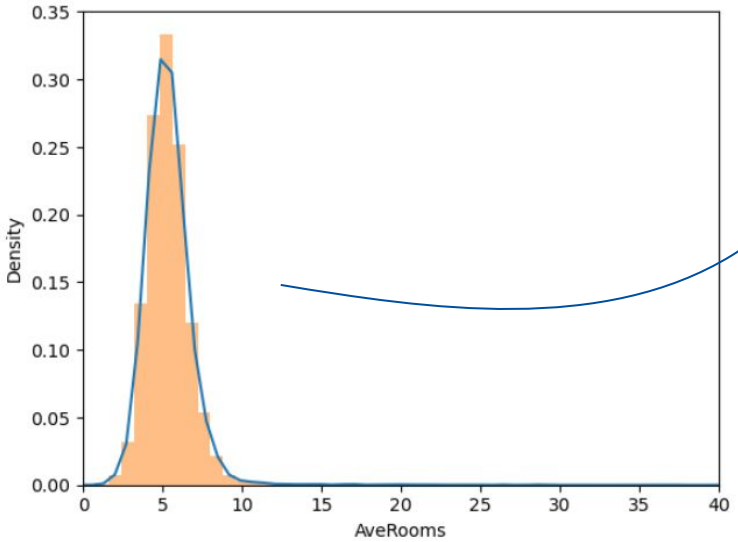
Using PDP to get this involves:

1) S = `AveBedrms`
2) Per value interval of S: Replace feature with value and average predictions.
3) Draw curve

PDP on highly correlated data

S = AveBedrms
C = (AveRooms, ...)

Let's look at the *marginal* distribution of AveRooms - meaning the distribution ignoring other feature values

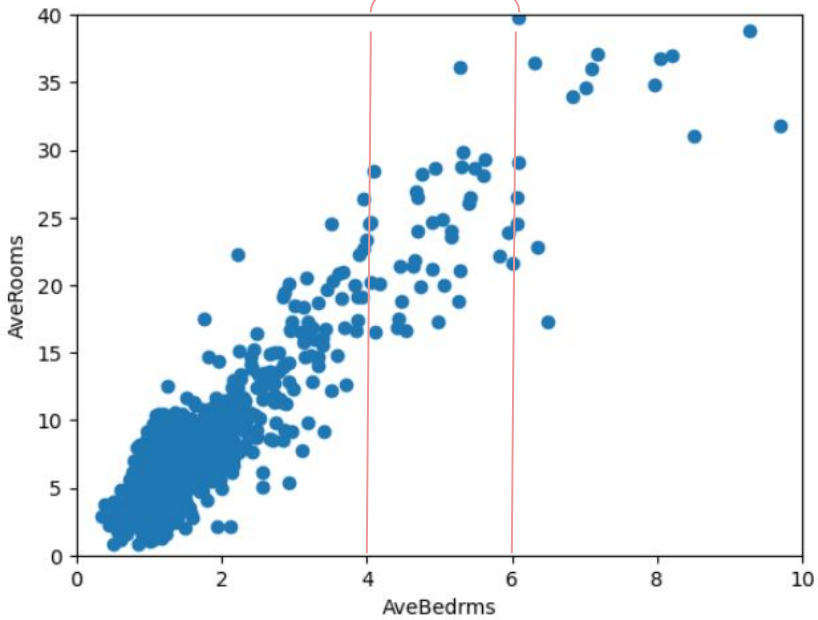


If we sample from this distribution, we will get mostly houses with around 5 rooms

PDP on highly correlated data

S = AveBedrms

C = (AveRooms, ...)



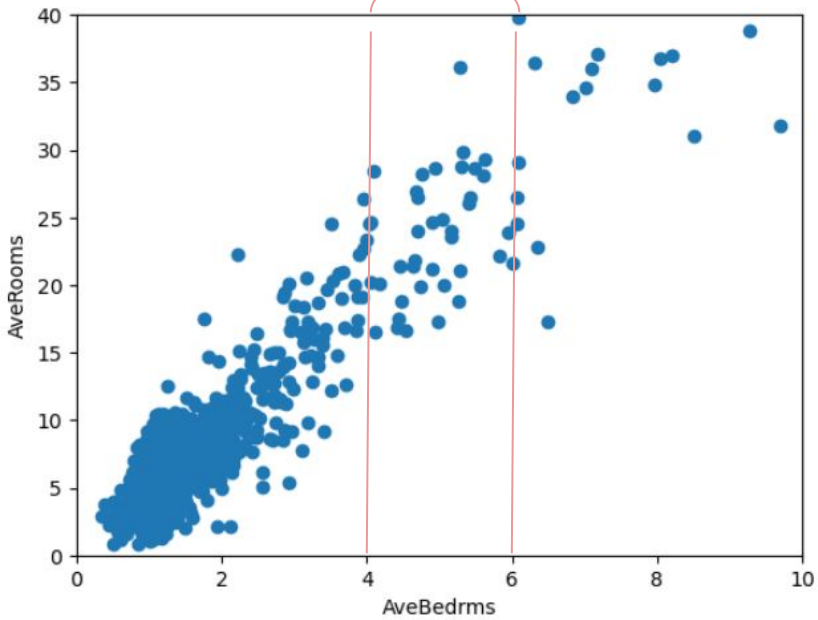
What de we get if we sample AveRooms uniformly?

What if we sampled AveRooms conditional upon the range AveBedrms = (4, 6)

PDP on highly correlated data

S = AveBedrms

C = (AveRooms, ...)



Remember, a typical value for AveRooms is 5.

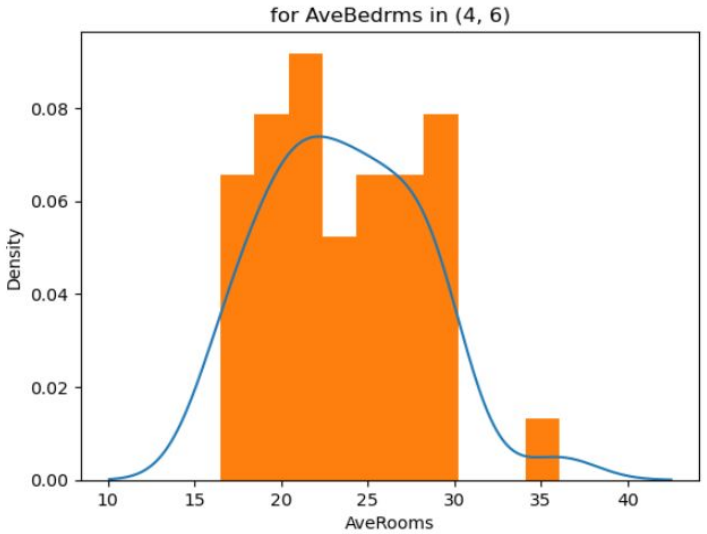
A house with 5 rooms and 4-6 bedrooms sounds weird...

PDP on highly correlated data

S = AveBedrms

C = (AveRooms, ...)

```
low, high = 4, 6
df_range = df.loc[df[feature2].between(low, high)]
sns.kdeplot(data=df_range, x=feature1)
plt.hist(df_range[feature1], density=True, bins=10)
plt.title("for AveBedrms in (4, 6)")
plt.show()
```

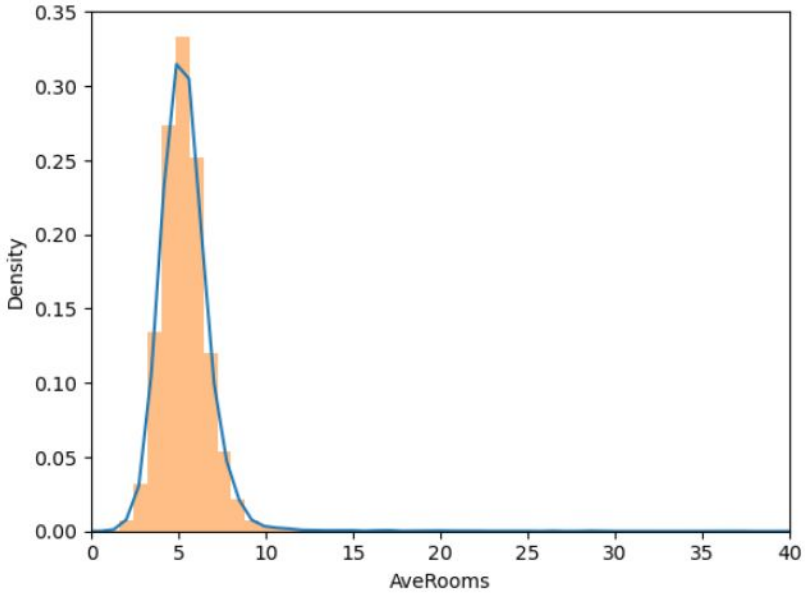


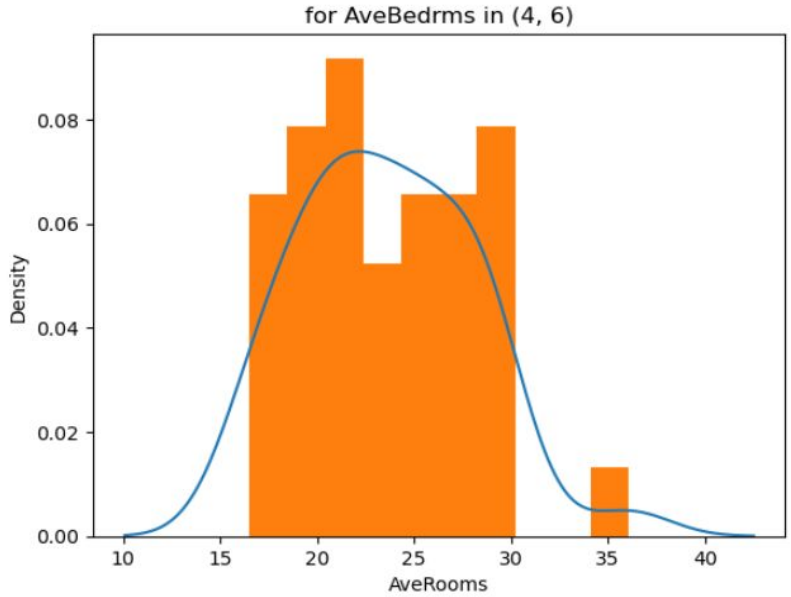
For the range AveBedrms = (4, 6), the distribution of AveRooms looks like this!

It makes much more sense for a room with 4-6 bedrooms to have around 20-25 rooms in total.

PDP on highly correlated data

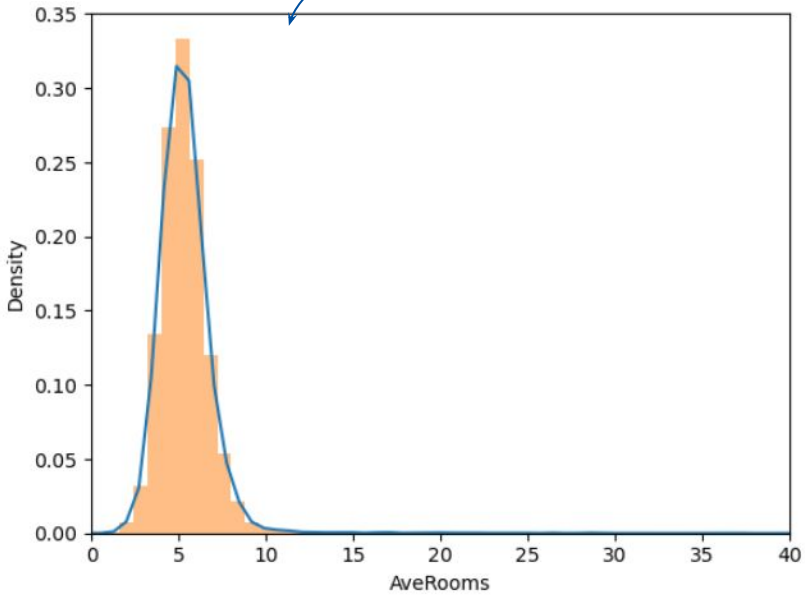
Which of these distributions would the PD function use for AveBedrms in (4,6)?

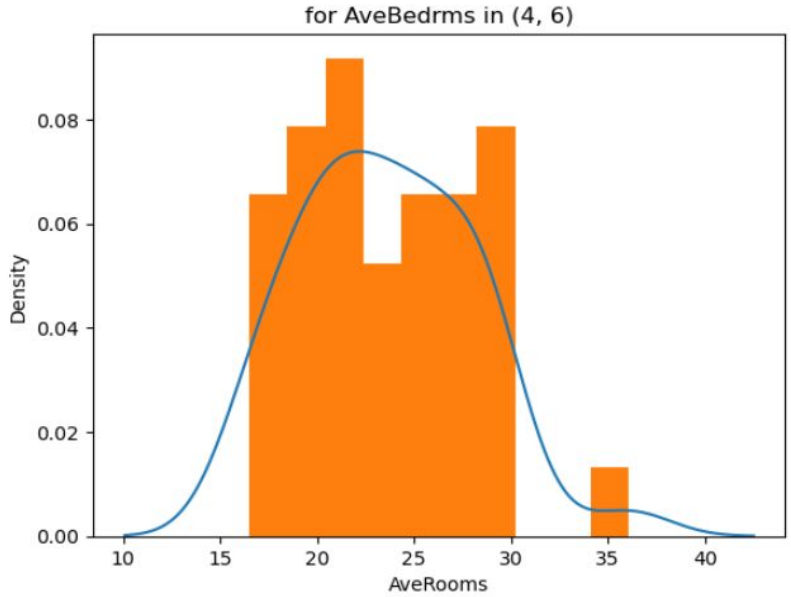




PDP on highly correlated data

The PD function uses the *marginal* distribution (it ignores feature correlations), thus including highly unrealistic data points!





Just use the conditional distribution?

How should we estimate the effect of a feature on the model prediction? Using the conditional distribution of the features?

If we average the predictions of all houses with 4-6 bedrooms, we estimate the *combined* effect of rooms and bedrooms, because of their correlation.

Just use the conditional distribution?

If we average the predictions of all houses with 4-6 bedrooms, we estimate the *combined* effect of rooms and bedrooms, because of their correlation.

Suppose that the number of bedrooms has no effect on the predicted value of a house, only the number of rooms has. Using the conditional distribution would still show that the number of bedrooms increases the predicted value, since the number of rooms increases with it.

Using the conditional distributions would give us the combined effect.

The conditional distribution avoids including unlikely data instances, *but* mixes the effect of a feature with the effects of all correlated features.

From the conditional distribution to ALE

The conditional distribution avoids including unlikely data instances, *but* mixes the effect of a feature with the effects of all correlated features.

Accumulated Local Effects (ALE) solves this problem by using the conditional distribution, but calculating *differences* in predictions instead of averages.

From the conditional distribution to ALE

The conditional distribution avoids including unlikely data instances, *but* mixes the effect of a feature with the effects of all correlated features.

Accumulated Local Effects (ALE) solves this problem by using the conditional distribution, but calculating *differences* in predictions instead of averages.

To get the effect of 4-6 AveBedrms:

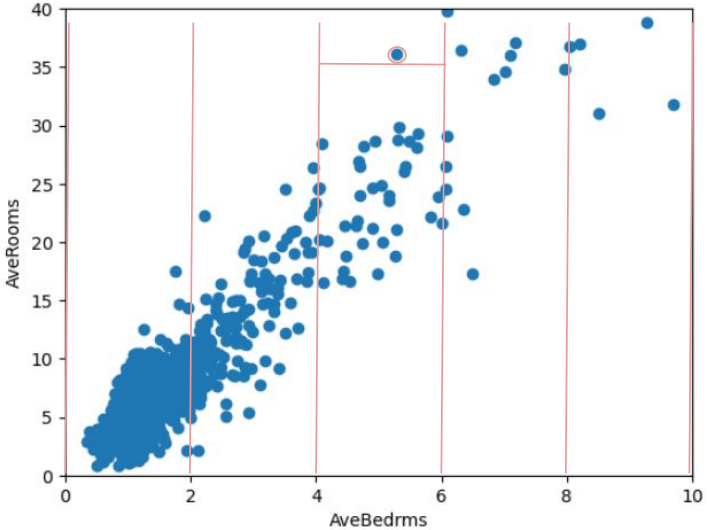
1) include all houses with 4-6 AveBedrms
2) calculate the model prediction pretending the house had AveBedrms=6, and subtract the prediction pretending the house had AveBedrms=4.

This yields the isolated effect of AveBedrms, without effects from correlated features.

From the conditional distribution to ALE

To get the effect of 4-6 AveBedrms:

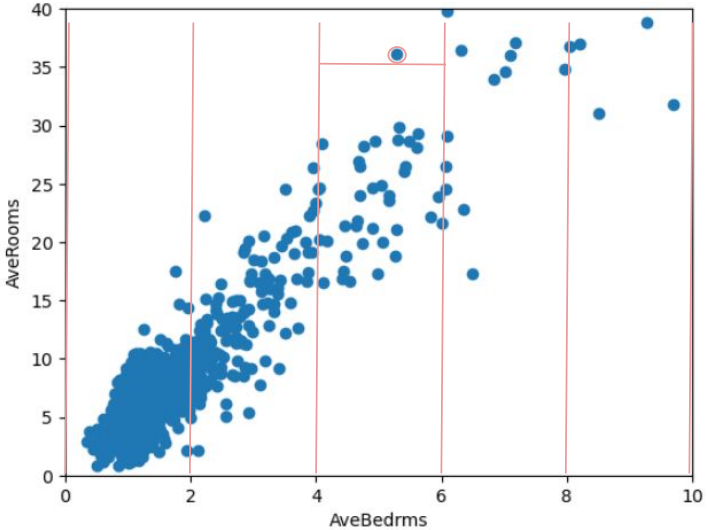
1) include all houses with between 4 and 6 AveBedrms.
2) calculate the model prediction pretending the house has AveBedrms=6, and subtract the prediction pretending the house has AveBedrms=4.



From the conditional distribution to ALE

To get the effect of 4-6 AveBedrms:

1) include all houses with between 4 and 6 AveBedrms.
2) calculate the model prediction pretending the house has AveBedrms=6, and subtract the prediction pretending the house has AveBedrms=4.



For each datapoint in the interval: calculate the difference in the prediction when replacing the feature with the upper and lower limit of the interval.

This means that we have to define neighborhoods for feature *S*.

The use of differences blocks the effect of other features.

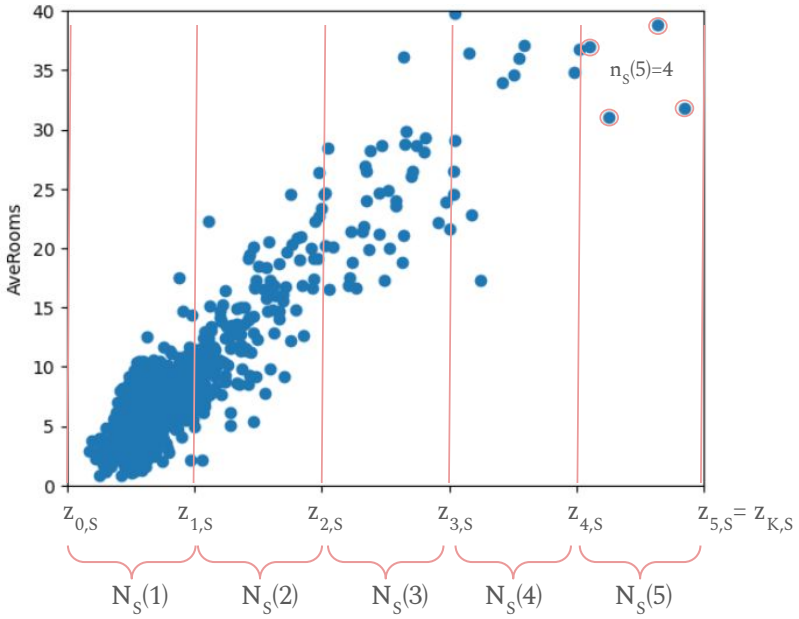
Estimating Accumulated Local Effects (ALE)

We choose a feature of interest S, with range $[x_{\min,S}, x_{\max,S}]$. In this range, we make K intervals, and denote

$$x_{min,S} = z_{0,S} \quad x_{max,S} = z_{K,S}$$

Let k=1,…,K be the index of the intervals, and $n_S(k)$ denote the number of data points where the value of feature S falls into the kth interval $N_S(k)$, so that

$$\sum_{k=1}^{K} n_S(k) = n$$



ALE pseudocode

```
def ale(X: pd.DataFrame, model: Callable, feature: str, grid_size: int):

# Make bins and assign the values of X[feature] to the correct bin
bins = split the feature's value range into grid_size quantile bins make sure that bins are unique
feat_cut = for each sample, find out which bin it belongs to # tip: use pd.cut

X_left, X_right = X.copy(), X.copy # keep all the feature values
X_left[feature] , X_right[feature] = the left / right value of the feature's corresponding bin
y_left, y_right = model.predict(X_left), model.predict(X_right)

#(more to come)
```

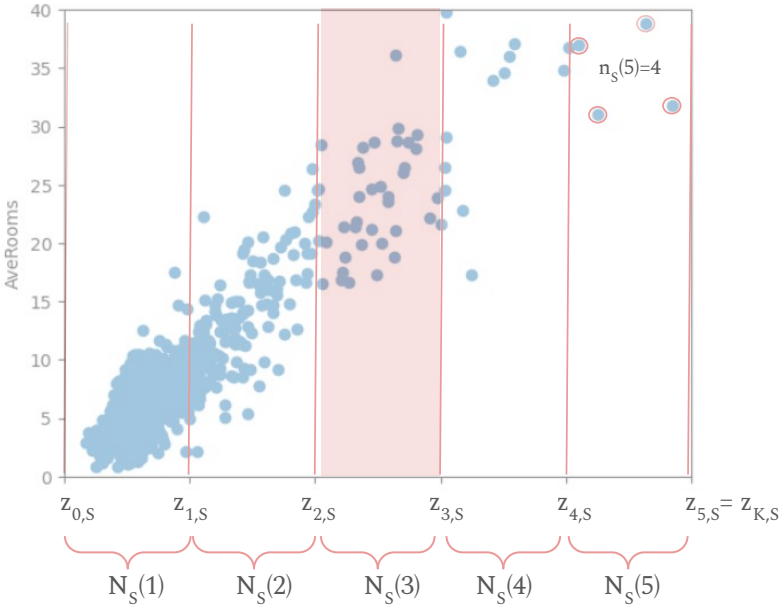
Estimating the Local Effect

For a particular value x, $k_S(x)$ denotes the index of the interval into which x falls, so that

$$x \in (z_{k_S(x)-1,S}, z_{k_S(x),S}]$$

The **local effect** of the feature S on the prediction in the interval $N_S(k)$ is

$$\mathrm{LE}_S(x) = \sum_{i:x_S^{(i)} \in N_S(k)} \left[f(x_S = z_{k,S}, \mathbf{x}_{\setminus j}^{(i)}) - f(x_S = z_{k-1,S}, \mathbf{x}_{\setminus j}^{(i)}) \right]$$



Estimating the Local Effect

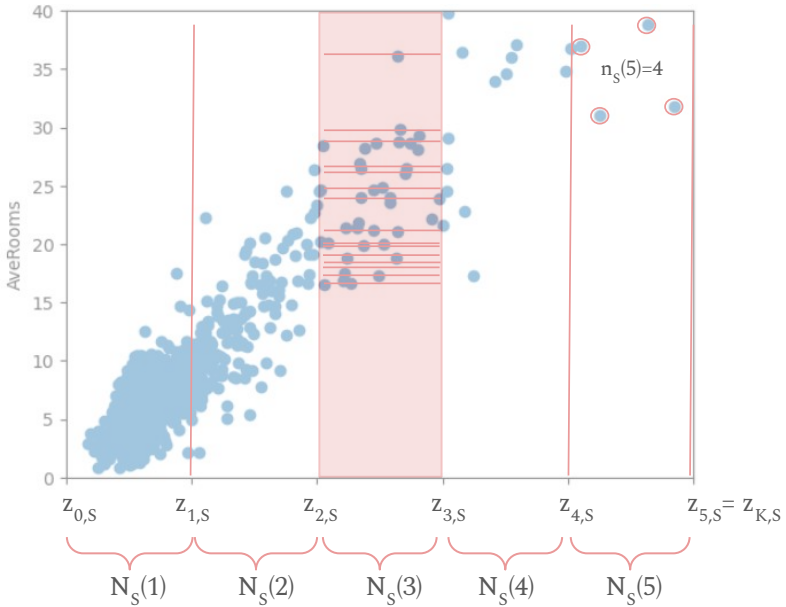
For a particular value x, $k_S(x)$ denotes the index of the interval into which x falls, so that

$$x \in (z_{k_S(x)-1,S}, z_{k_S(x),S}]$$

The **local effect** of the feature S on the prediction in the interval $N_S(k)$ is

$$\mathrm{LE}_S(x) = \sum_{i:x_S^{(i)} \in N_S(k)} \left[\underbrace{f(x_S = z_{k,S}, \mathbf{x}_{\setminus j}^{(i)})}_{\text{prediction on the right border}} - \underbrace{f(x_S = z_{k-1,S}, \mathbf{x}_{\setminus j}^{(i)})}_{\text{prediction on the left border}} \right]$$

sum over all indices i such that the data point's value for feature S falls into the interval



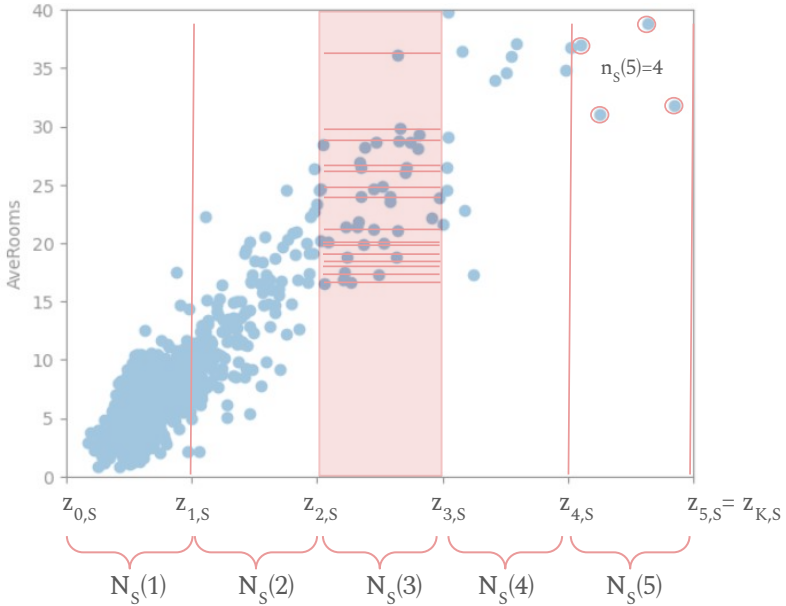
Estimating the Local Effect

The **local effect** of the feature S on the prediction in the interval $N_S(k)$ is

$$\mathrm{LE}_S(x) = \sum_{i:x_S^{(i)} \in N_S(k)} \left[f(x_S = z_{k,S}, \mathbf{x}_{\setminus j}^{(i)}) - f(x_S = z_{k-1,S}, \mathbf{x}_{\setminus j}^{(i)}) \right]$$

Effect refers to the change in prediction when changing the value of feature *S*, and

Local refers to calculating the effect in an interval.



ALE pseudocode

```
def ale(X: pd.DataFrame, model: Callable, feature: str, grid_size: int):

# Make bins and assign the values of X[feature] to the correct bin
bins = split the feature's value range into grid_size quantile bins make sure that bins are unique
feat_cut = for each sample, find out which bin it belongs to # tip: use pd.cut

X_left, X_right = X.copy(), X.copy # keep all the feature values
X_left[feature] , X_right[feature] = the left / right value of the feature's corresponding bin
y_left, y_right = model.predict(X_left), model.predict(X_right)

# Get the Local Effects:
for each bin, calculate all delta=y_left-y_right and append them to a list
delta_mean = the mean values of all the delta (one value per bin)

#(more to come)
```

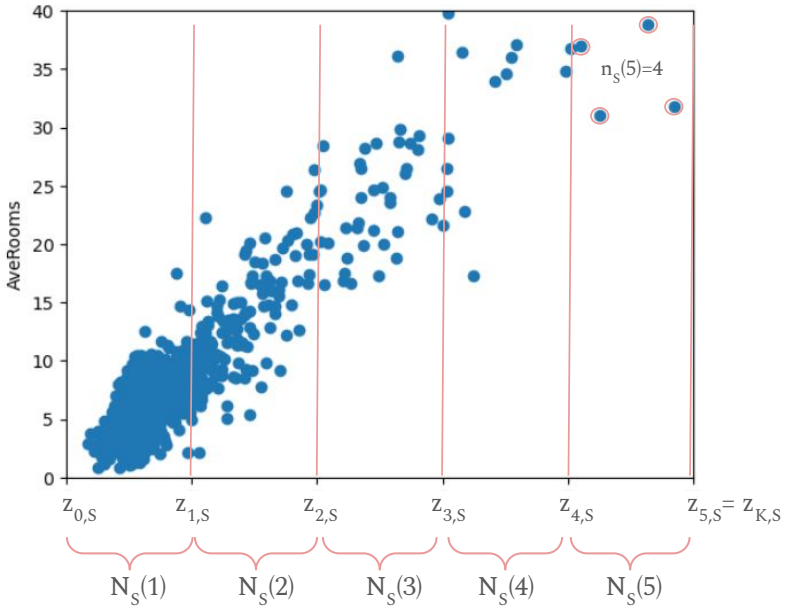
Estimating the Accumulated Local Effect

$$\mathrm{LE}_S(x) = \sum_{i:x_S^{(i)} \in N_S(k)} \left[f(x_S = z_{k,S}, \mathbf{x}_{\setminus j}^{(i)}) - f(x_S = z_{k-1,S}, \mathbf{x}_{\setminus j}^{(i)}) \right]$$

The sum means: include only data points inside the interval.

We care about the total effect of the feature, not only in a small window, but the difference all the way from the feature's minimum value to its actual value.

We therefore **accumulate** the local effects from the first interval up until the interval containing the feature value,



ALE pseudocode

```
def ale(X: pd.DataFrame, model: Callable, feature: str, grid_size: int):

# Make bins and assign the values of X[feature] to the correct bin
bins = split the feature's value range into grid_size quantile bins make sure that bins are unique
feat_cut = for each sample, find out which bin it belongs to # tip: use pd.cut

X_left, X_right = X.copy(), X.copy # keep all the feature values
X_left[feature] , X_right[feature] = the left / right value of the feature's corresponding bin
y_left, y_right = model.predict(X_left), model.predict(X_right)

# Get the Local Effects:
for each bin, calculate all delta=y_left-y_right and append them to a list
delta_mean = the mean values of all the delta (one value per bin)
# Accumulate the LEs
delta_cumsum = the cumulative sum up to that interval

#(more to come)
```

Estimating the Accumulated Local Effect

$$\mathrm{LE}_S(x) = \sum_{i:x_S^{(i)} \in N_S(k)} \left[f(x_S = z_{k,S}, \mathbf{x}_{\setminus j}^{(i)}) - f(x_S = z_{k-1,S}, \mathbf{x}_{\setminus j}^{(i)}) \right]$$

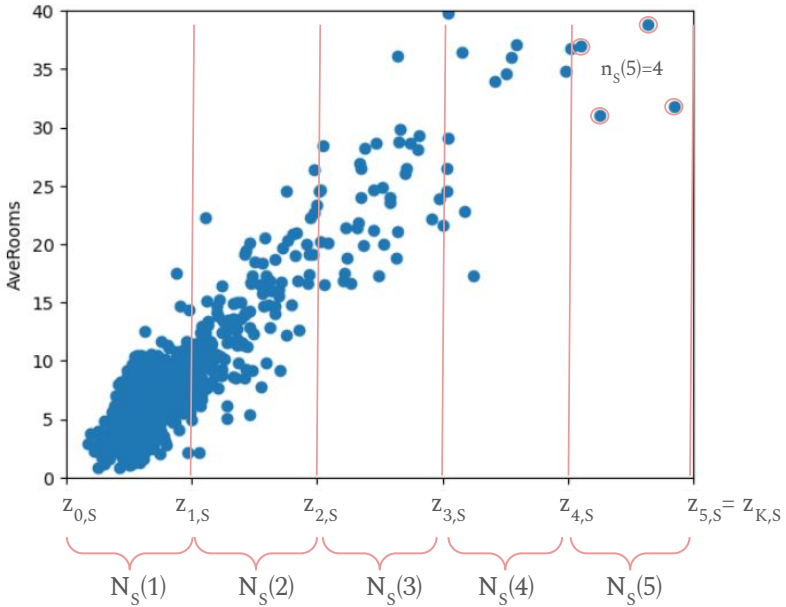
We care about the total effect of the feature, not only in a small window, but the difference all the way from the feature's minimum value to its actual value.

We therefore **accumulate** the local effects from the first interval up until the interval containing the feature value,

$$\hat{g}_{S,\mathrm{ALE}}(x) = \sum_{k=1}^{k_S(x)} \frac{1}{n_S(k)} \mathrm{LE}_S(x)$$

Imagine sliding over each interval and including all the prediction differences from each window.

This yields the (uncentered) **accumulated local effects** (ALE)



So far:

Local effects: the differences in the model's predictions when x_S increases from x_{k-1} to x_k

$$\mathrm{LE}_S(x) = \sum_{i:x_S^{(i)} \in N_S(k)} \left[f(x_S = z_{k,S}, \mathbf{x}_{\setminus j}^{(i)}) - f(x_S = z_{k-1,S}, \mathbf{x}_{\setminus j}^{(i)}) \right]$$

Accumulated Local Effects: the average prediction differences within each interval accumulated.

$$\hat{g}_{S,\mathrm{ALE}}(x) = \sum_{k=1}^{k_S(x)} \frac{1}{n_S(k)} \mathrm{LE}_S(x)$$

$\Rightarrow$ the (uncentered) ALE of a feature value in the *k*th interval is the sum of the effects of the first through the *k*th interval.

Sorry, there's more... :)

We have calculated the *uncentered* ALE.

Again, this is the cumulative change in the model's prediction from moving the feature of interest from its minimum value up to the point x, adding up local effects interval by interval.

The uncentered ALE therefore answers the question:
"how much does the feature, at a certain value, shift the model's prediction relative to the prediction with the feature's minimum value?"

while we want to know
"how much does the feature, at a certain value, shift the model's prediction away from the average prediction?"

We don't want to know the effect of the feature compared to its minimum value, but the effect of the feature on the model prediction overall

Sorry, there's more... :)

We have calculated the *uncentered* ALE.

Again, this is the cumulative change in the model's prediction from moving the feature of interest from its minimum value up to the point x, adding up local effects interval by interval.

The uncentered ALE therefore answers the question:
"how much does the feature, at a certain value, shift the model's prediction relative to the prediction with the feature's minimum value?"

while we want to know
"how much does the feature, at a certain value, shift the model's prediction away from the average prediction?"

To get this, we have to center the ALE curve so that the mean over all data indices is zero.

Sorry, there's more... :)

The uncentered ALE answers the question:
"how much has does the feature, at a certain value, shift the model's prediction relative to the feature's minimum value?"

while we want to know
"how much does the feature, at a certain value, shift the model's prediction relative to the prediction with the feature's minimum value?"

To get this, we have to center the ALE curve so that the mean over all data indices is zero.

We do this by subtracting an estimate of

$$E[\hat{g}_{S,ALE}(X_S)]$$

Meaning the expectation of the uncentered ALE for the entire distribution of X_S.

The final ALE curve

The final, centered Accumulated Local Effects is

$$\hat{f}_{S,\text{ALE}}(x) = \hat{g}_{S,\text{ALE}}(x) - \frac{1}{n}\sum_{i=1}^{n} \hat{g}_{S,ALE}(x_S^{(i)})$$

Or, equivalently

$$\hat{f}_{S,\text{ALE}}(x) = \hat{g}_{S,\text{ALE}}(x) - \frac{1}{n}\sum_{k=1}^{K} n_S(k)\hat{g}_{S,\text{ALE}}(z_{k,S})$$

if you prefer to think about the intervals.

So we subtract "the ALE curve calculated for whatever the value of feature S is".

ALE pseudocode

```
def ale(X: pd.DataFrame, model: Callable, feature: str, grid_size: int):

# Make bins and assign the values of X[feature] to the correct bin
bins = split the feature's value range into grid_size quantile bins make sure that bins are unique
feat_cut = for each sample, find out which bin it belongs to # tip: use pd.cut

X_left, X_right = X.copy(), X.copy # keep all the feature values
X_left[feature] , X_right[feature] = the left / right value of the feature's corresponding bin
y_left, y_right = model.predict(X_left), model.predict(X_right)

# Get the Local Effects:
for each bin, calculate all delta=y_left-y_right and append them to a list
delta_mean = the mean values of all the delta (one value per bin)
# Accumulate the LEs
delta_cumsum = the cumulative sum up to that interval
# center the ALE by subtracting the mean
delta_centered = delta_cumsum - np.mean(delta_cumsum)
```

Original ALE

JOURNAL ARTICLE

Visualizing the Effects of Predictor Variables in Black Box Supervised Learning Models FREE

Daniel W. Apley ✉, Jingyu Zhu

Journal of the Royal Statistical Society Series B: Statistical Methodology, Volume 82, Issue 4, September 2020, Pages 1059–1086, https://doi.org/10.1111/rssb.12377

Published: 11 June 2020 **Article history** ▾

ALE was introduced in a 2020 paper.

We have *estimated* the *first order* ALE.

The paper presents the analytical expressions for the estimates we have looked at, and details for how to calculate higher order ALEs.

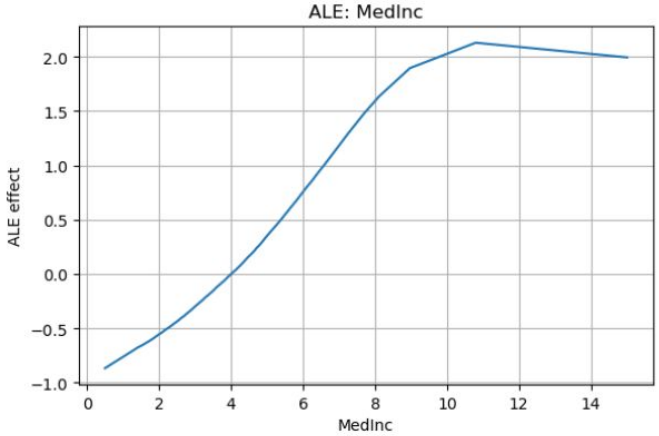
Homework: Write the ale function

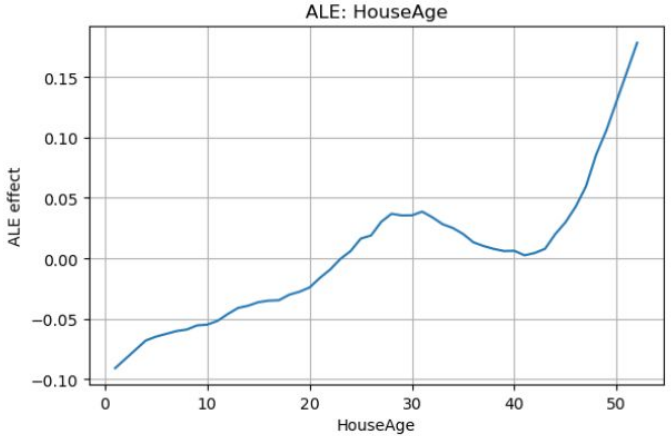
```
def ale(X: pd.DataFrame, model: Callable, feature: str, grid_size: int):

# Make bins and assign the values of X[feature] to the correct bin
bins = np.quantile(...
feat_cut = pd.cut(...
bin_codes = index of interval for each data point
feat_cut = for each sample, find out which bin it belongs to # tip: use pd.cut

X_left, X_right = X.copy(), X.copy # keep all the feature values
X_left[feature] , X_right[feature] = the left / right value of the feature's corresponding bin
y_left, y_right = model.predict(X_left), model.predict(X_right)

# Get the Local Effects:
delta is a nested array,populated using bin_codes, y_left and y_right
delta_mean = the mean values of all the delta (one value per bin)
# Accumulate the LEs
delta_cumsum = the cumulative sum up to that interval
# center the ALE by subtracting the mean
delta_centered = delta_cumsum - np.mean(delta_cumsum)

return delta_centered, bin_intervals
```

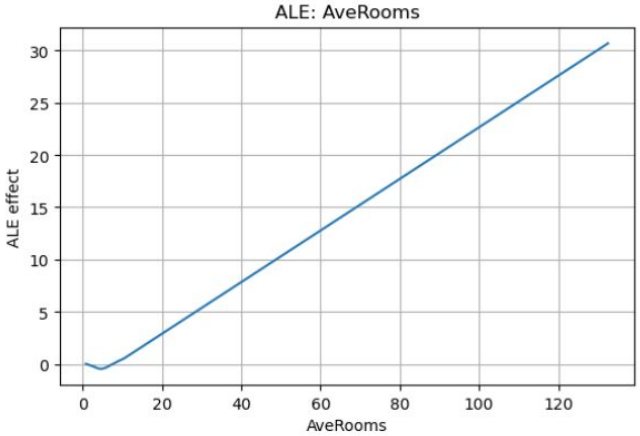
Plot results

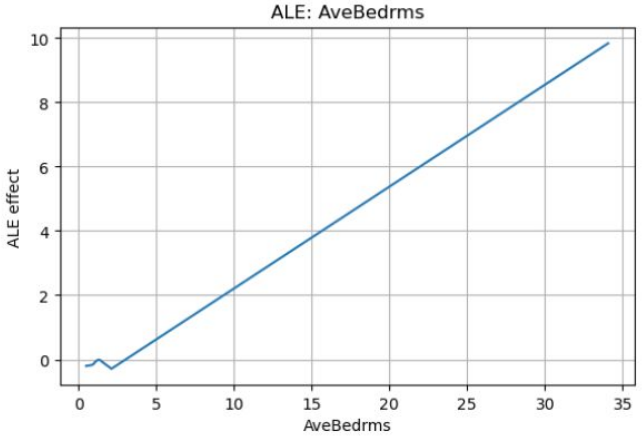
```python
n_cols = 2
n_rows = -(-len(feature_names) // n_cols)  # ceiling division
fig, axs = plt.subplots(n_rows, n_cols, figsize=(6 * n_cols, 4 * n_rows), squeeze=False)

for idx, feature_name in enumerate(feature_names):
    ax = axs[idx // n_cols][idx % n_cols]

    # Compute ALE effect
    ale_eff, bin_intervals = ale(
        X=X_test,
        model=pipe,
        feature=feature_name,
        grid_size=100,
    )

    intervals = list(bin_intervals.left) + [bin_intervals.right[-1]] # For x axis

    ax.plot(intervals, ale_eff)
    ax.set_title(f"ALE: {feature_name}")
    ax.set_xlabel(feature_name)
    ax.set_ylabel("ALE effect")
    ax.grid(True)

# Turn off any unused axes
for i in range(len(feature_names), n_rows * n_cols):
    axs[i // n_cols][i % n_cols].axis("off")

plt.tight_layout()
plt.savefig("ale.png")
plt.show()
```

With this snippet, and an `ale` function that returns the ALE effects and bin_intervals (here: from `pd.cut`), you can plot the results and get something that should look liiikeee…



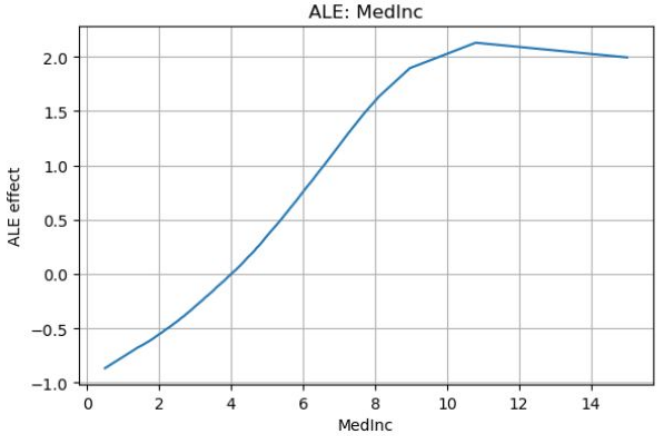


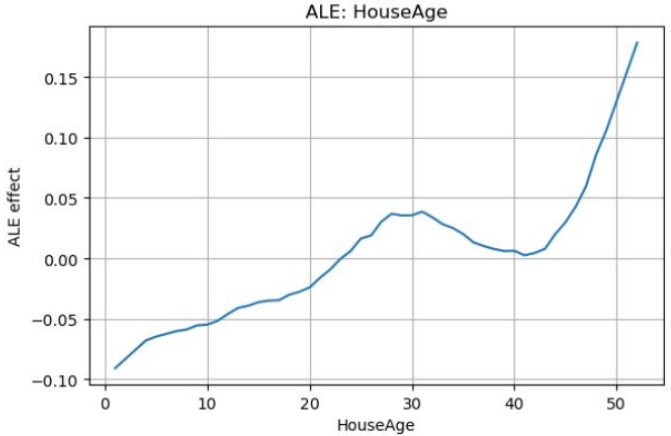


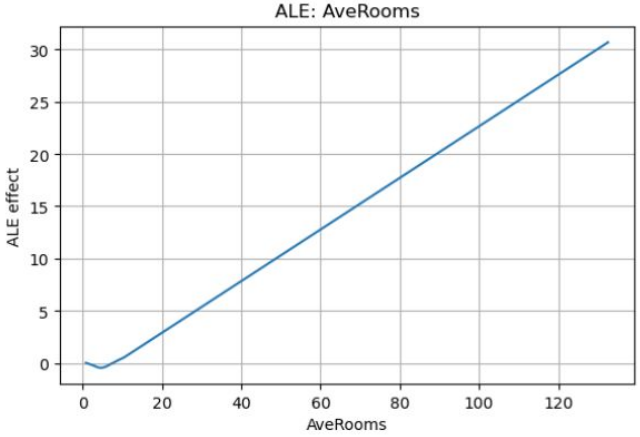


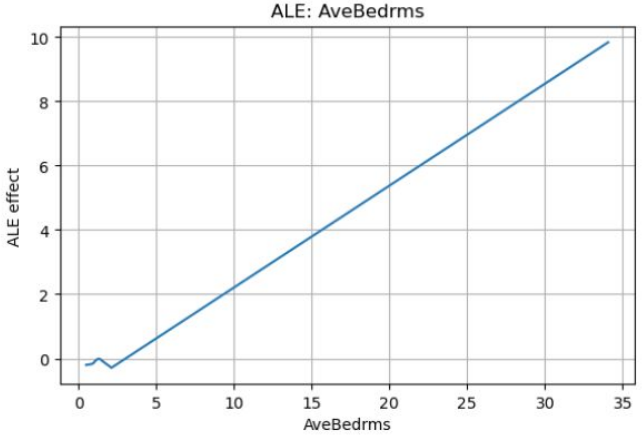
What do you see?

How do we interpret these plots?



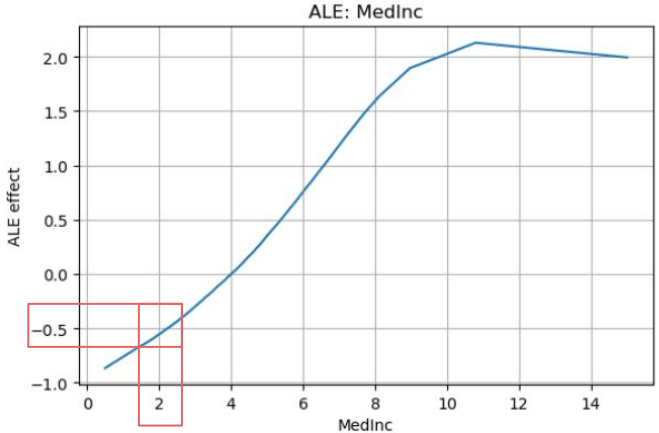


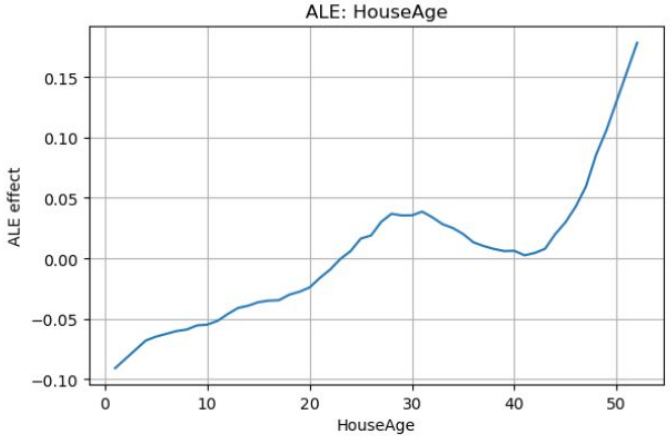


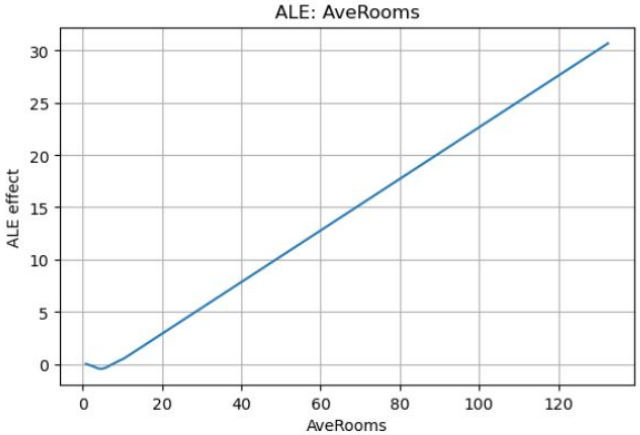


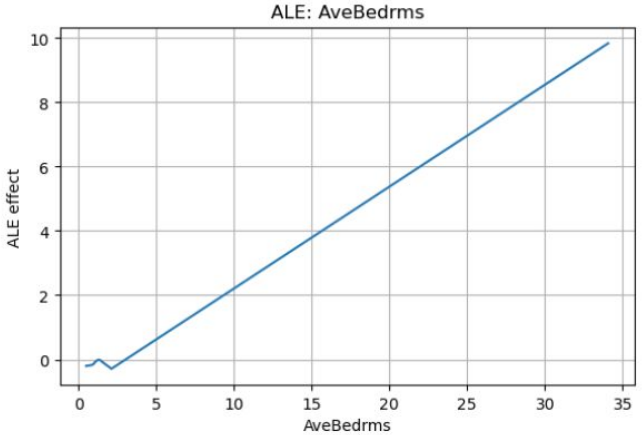
Very short story:
"how a feature affects the prediction on average"

Actual story:
The ALE effect shows the main effect of the feature at a certain value compared to the average prediction.



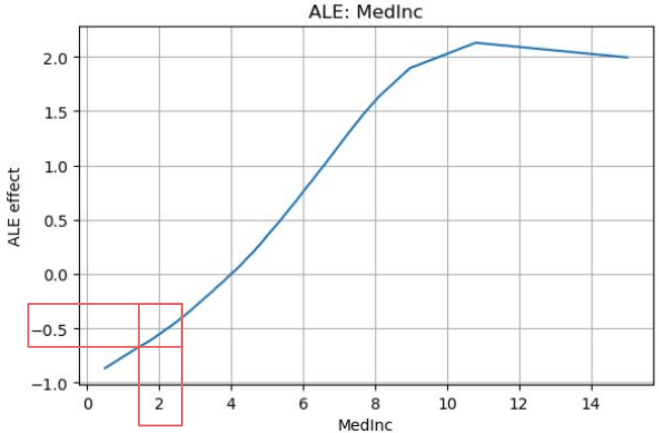


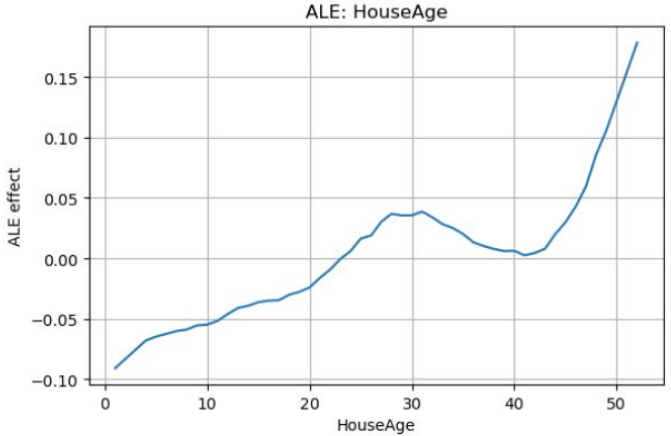


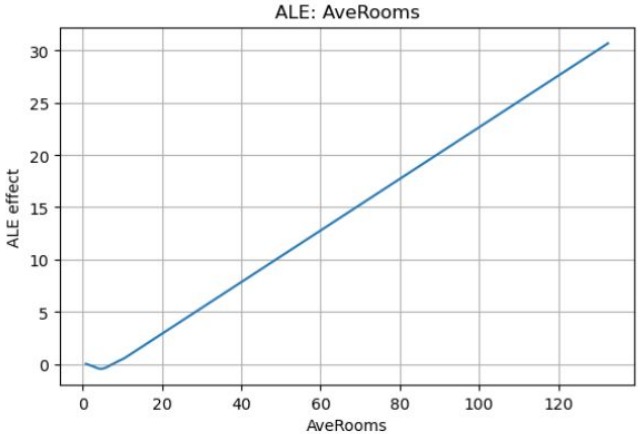


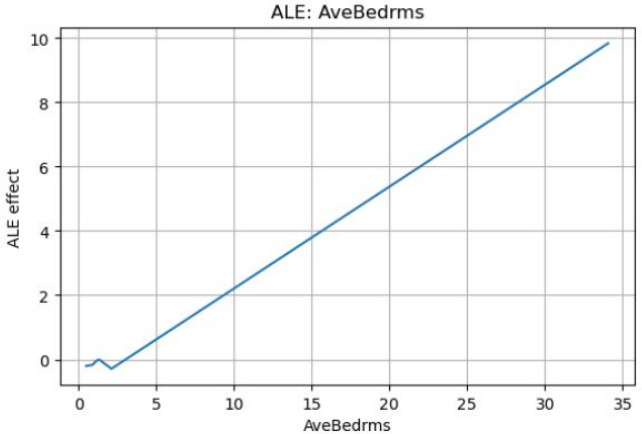
The ALE effect shows the main effect of the feature at a certain value compared to the average prediction.

Example: The ALE effect = -0.5 at MedInc = 2 *means that?*



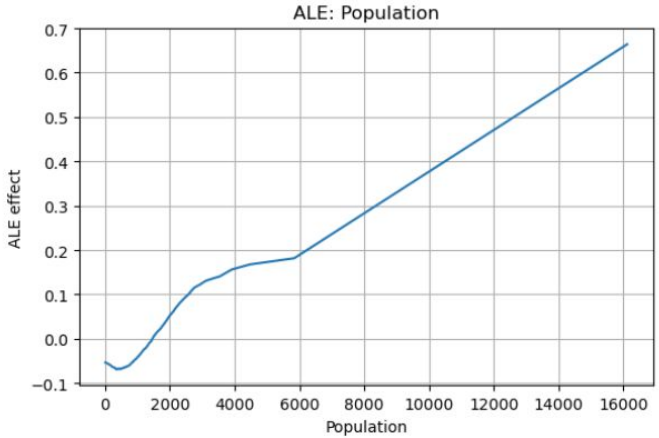


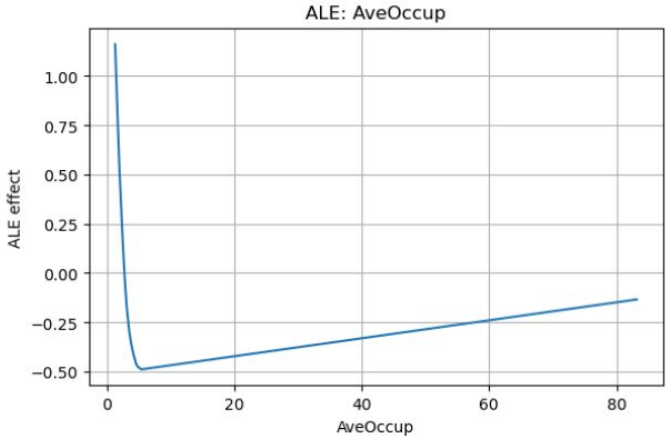


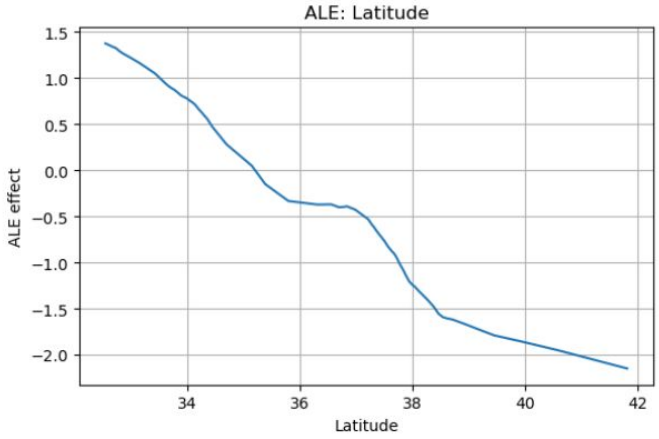


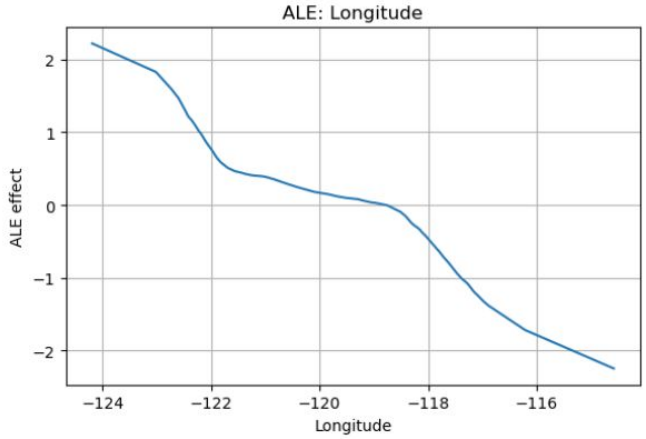
The ALE effect shows the main effect of the feature at a certain value compared to the average prediction.

Example:
ALE effect = -0.5 at MedInc = 2
⇒ when this feature has value 2, then the prediction is lower by 0.5 compared to the average prediction.







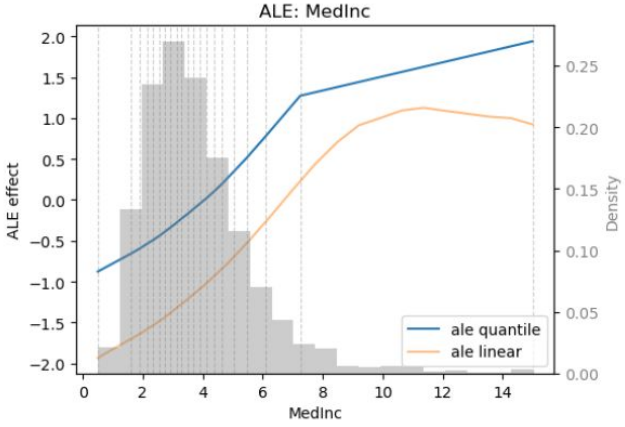


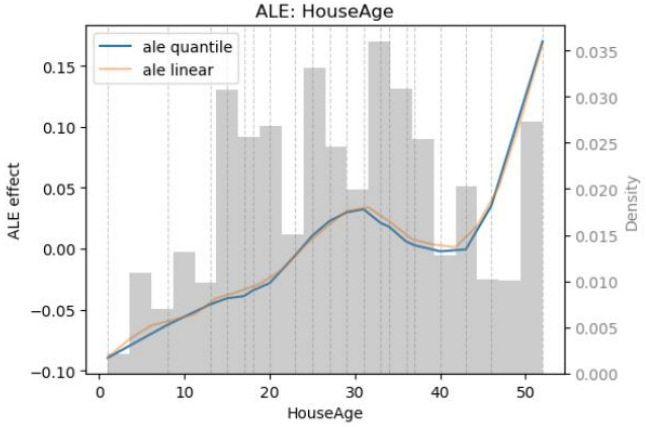
Now you :)

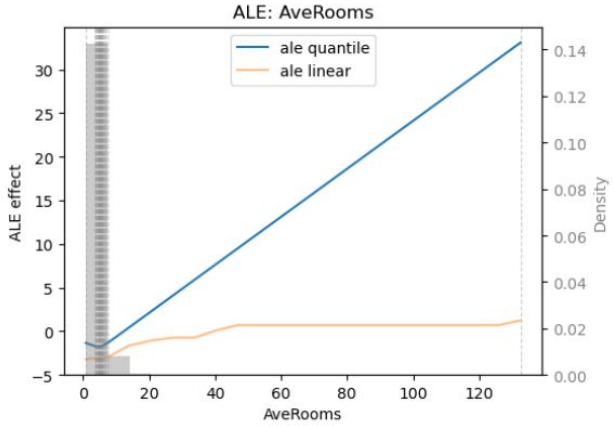
norwegian open ai lab NTNU

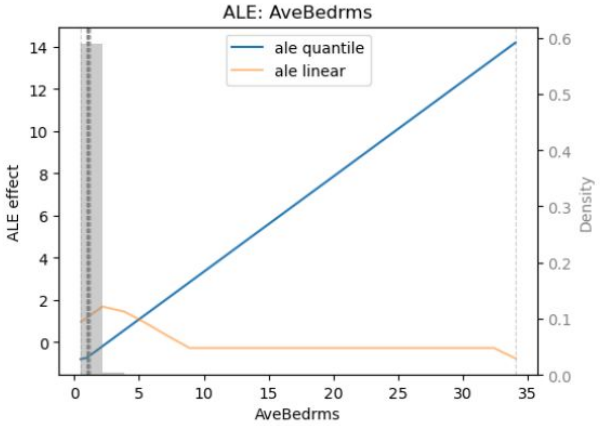
The outlier effect in ALE

The effect of dividing the data X[feature] into quantiles vs linspace







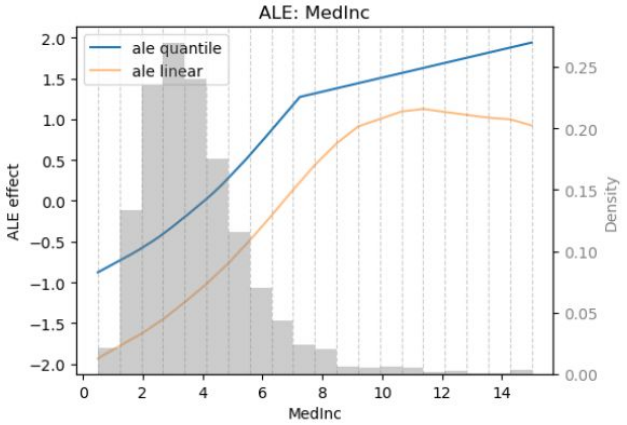


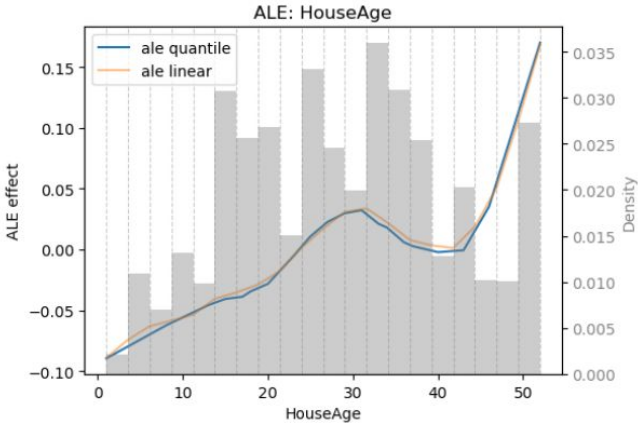
The gray bars show the data distribution.

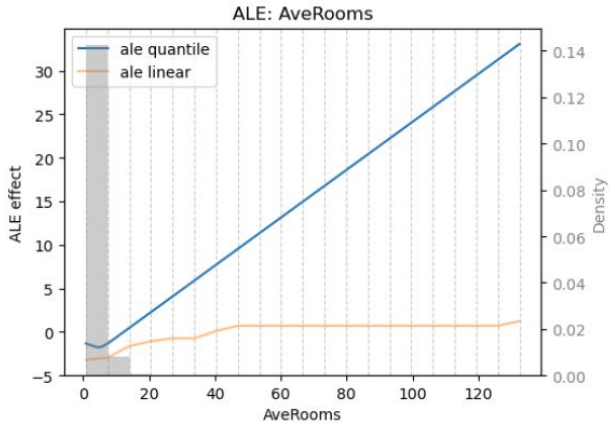
The dotted lines show the bins (here: quantiles)

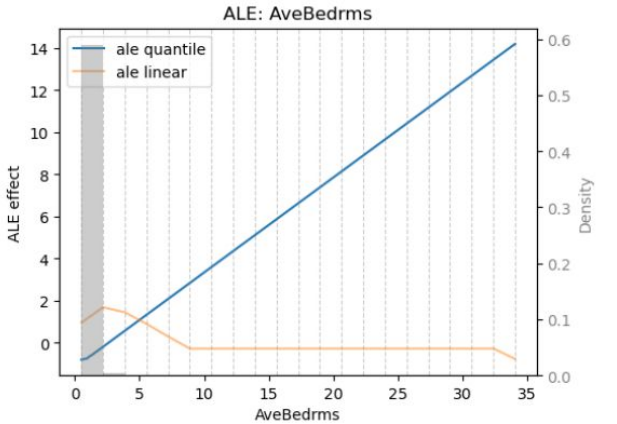
What do we see?

The effect of dividing the data X[feature] into quantiles vs linspace







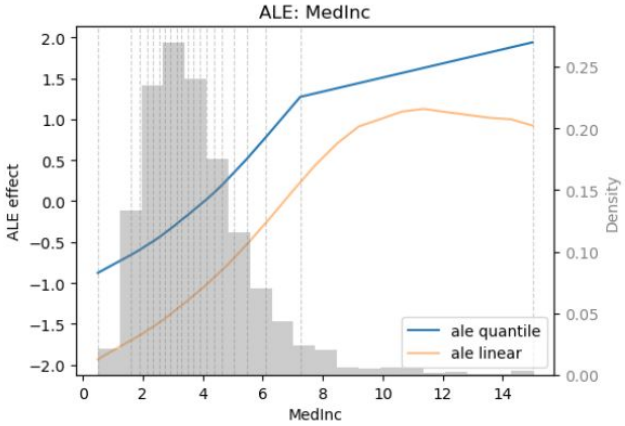


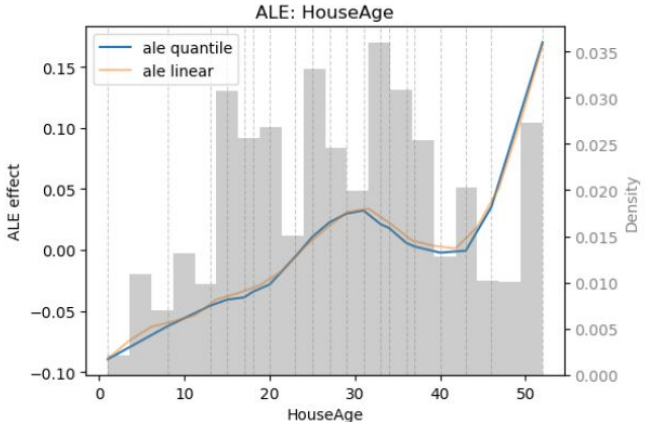
The gray bars show the data distribution.

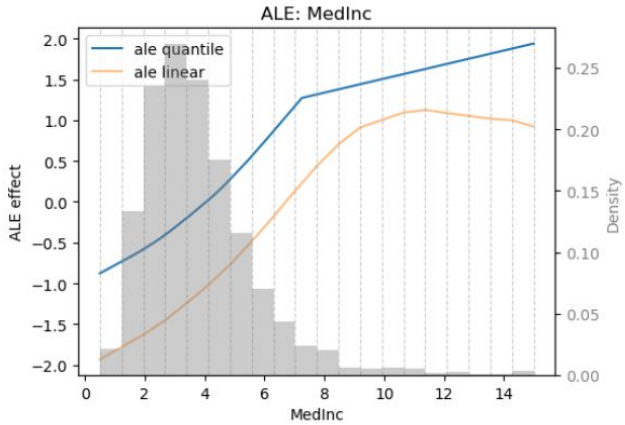
The dotted lines show the bins (here: linspace)

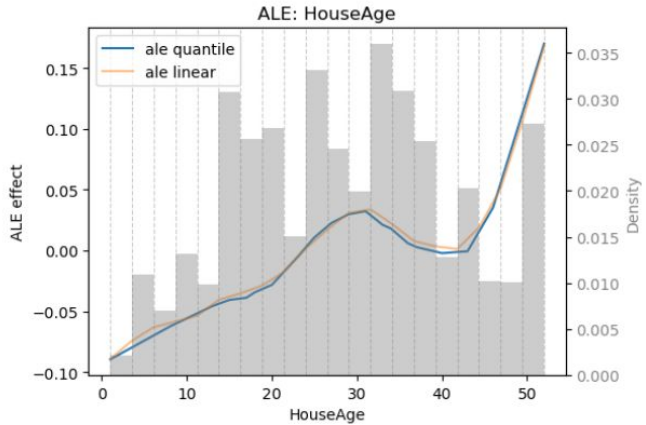
What do we see?

The effect of dividing the data X[feature] into quantiles vs linspace







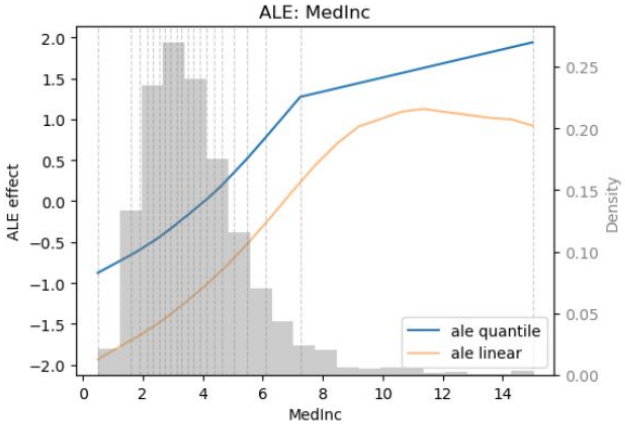


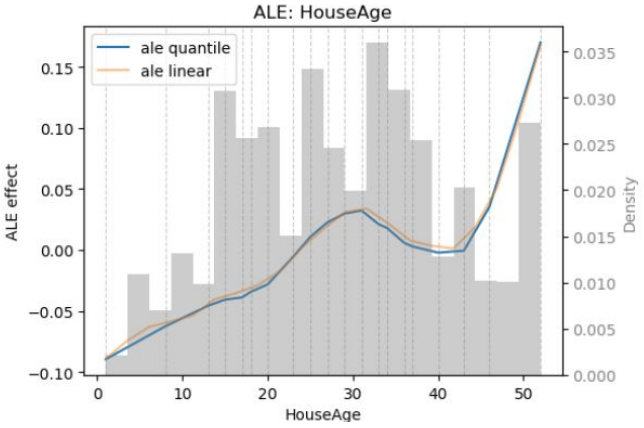
The gray bars show the data distribution.

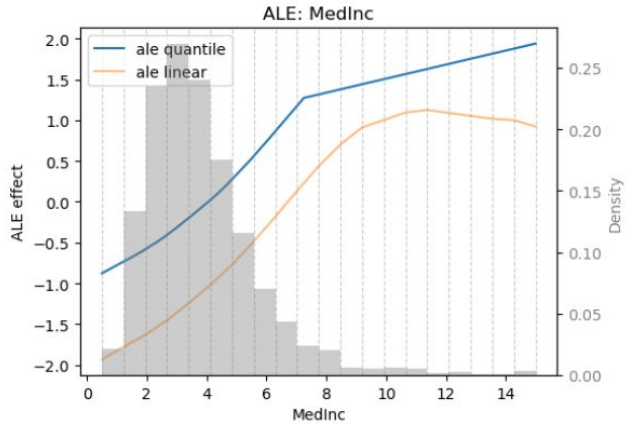
The dotted lines show the bins (top: quantiles, bottom:linspace)

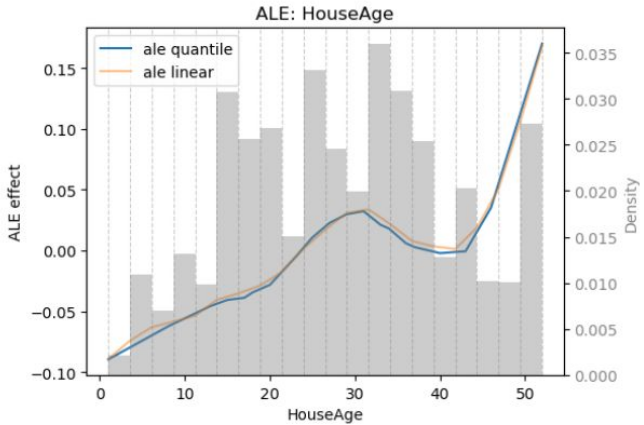
What do we see?

The effect of dividing the data X[feature] into quantiles vs linspace









The gray bars show the data distribution.

The dotted lines show the bins (top: quantiles, bottom:linspace)

We see that:

for linspace bins, outliers can dominate the bins.

for quantile bins, the ALE curve is more discontinuous.

Homework part 3:

Calculate the ALE effects for the california_housing data using both linspace and quantiles:

```
bins = np.quantile(X[feature], np.linspace(0,1,grid_size+1)) # quantiles
bins = np.linspace(X[feature].min(), X[feature].max(), grid_size+1) # linspace
```

Calculate and plot the SHAP values for the same data, on the same plot.

Find out whether the SHAP values agree more with the ALE values resulting from the linspace or the quantiles.

ALE

Where do we place this in the taxonomy?

Post-hoc?

Model-agnostic or model-specific?

Local or global?

Post-hoc methods	Model-agnostic	Model-specific
Local		
Global		

ALE

Where do we place this in the taxonomy?

Post-hoc: *We got a model and want to explain its behaviour*

Model-agnostic or model-specific?

Local or global?

Post-hoc methods	Model-agnostic	Model-specific
Local		
Global		

ALE

Where do we place this in the taxonomy?

Post-hoc: *We got a model and want to explain its behaviour*

Model-agnostic: *The inner structure of the model is not important; we care only about the inputs and outputs.*

Local or global?

Post-hoc methods	Model-agnostic	Model-specific
Local		
Global		

ALE

Where do we place this in the taxonomy?

Post-hoc: *We got a model and want to explain its behaviour*

Model-agnostic: *The inner structure of the model is not important; we care only about the inputs and outputs.*

Global: *We accumulate the predictions over the whole dataset per feature to get a global trend.*

Post-hoc methods	Model-agnostic	Model-specific
Local		
Global		

ALE

Where do we place this in the taxonomy?

Post-hoc: *We got a model and want to explain its behaviour*

Model-agnostic: *The inner structure of the model is not important; we care only about the inputs and outputs.*

Global: *We accumulate the predictions over the whole dataset per feature to get a global trend.*

Post-hoc methods	Model-agnostic	Model-specific
Local		
Global	ALE	

Summary

Similarities and differences

Both PDP and ALE give us a visualisation of how the feature we're interested in, S, affects the model prediction overall. This is done by summing over the remaining features C in different ways.

Both PDP and ALE are functions of only x_S, while our model f depends on x_S and x_C

The main difference is that

PDP is based on the *marginal effect* of a feature, averaging f for x_S while sampling the other feature values x_C independently of the value of x_S

ALE is based on the *conditional effect* of a feature, calculating prediction differences of f for a window around x_S , including the actual feature values x_C in that window.

Summary - Partial Dependence Plots (PDP)

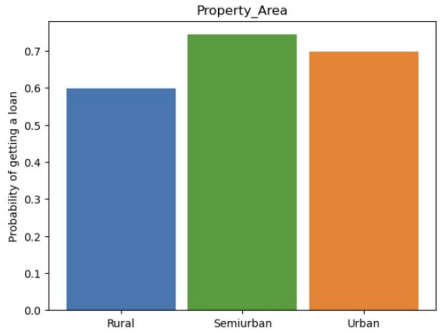
For each value of *S*, duplicate the remaining feature values, creating an upsampled dataset.
For each value of *S*, get model predictions for the the upsampled dataset.

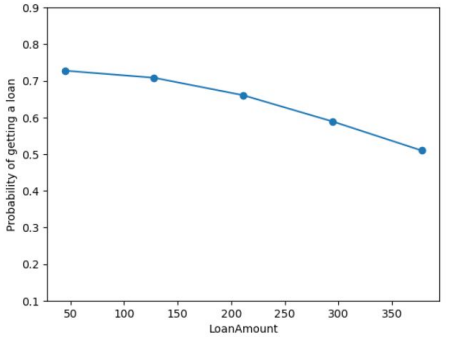
Calculate the average model prediction per value of *S*.

The PDP at a particular feature value represents the average model prediction if we force all data points to have that value for feature *S*.

PDPs are presented with the values of *S* on the *x* axis and the corresponding average predictions on the *y* axis,

- as bars for categorical *S*
- interpolating between values for continuous *S*.





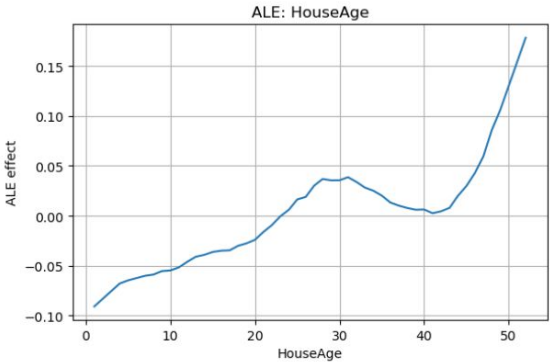
Summary - Accumulated local effects (ALE)

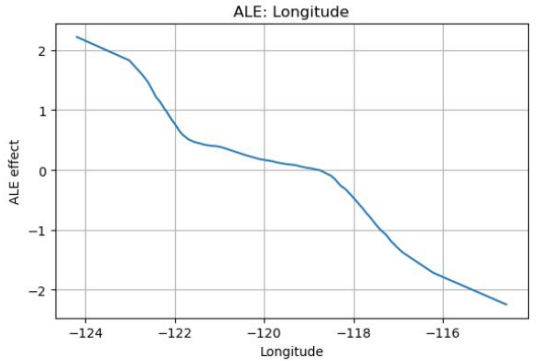
Divide the range of feature *S* into many intervals. For each interval, calculate the differences in model prediction, for data points inside the interval, replacing the value of feature *S* with the minimum and maximum value in the interval.

This way, excluded feature values C are sampled conditioned upon the value of the included feature(s) S, and the effects of excluded features C cancel out.

Accumulate the differences from the feature's minimum value to its actual value, and then centered.

The ALE value of a feature at a certain value quantises how much the feature, at this value, contributes to move the model prediction away from the average.





When do we use PDP? :)

When do we use ALE? :)

Homework :)

1. Calculate PDP values for the feature CoapplicantIncome, and see if you can make the curve change trend just by changing the included range.
   What does this tell us?
2. Write the ALE function and reproduce the plots from the slides.
3. Calculate the ALE effects for the california_housing data using both linspace and quantiles:

```
bins = np.quantile(X[feature], np.linspace(0,1,grid_size+1)) # quantiles
bins = np.linspace(X[feature].min(), X[feature].max(), grid_size+1) # linspace
```

   Calculate and plot the SHAP values for the same data, on the same plot.
   Find out whether the SHAP values agree more with the ALE values resulting from the linspace or the quantiles.

Have fun coding, and please practice saying exactly what an explanation method gives us :)

See you tomorrow!