

```

In [1]: # =====
# Notebook setup
# =====

%load_ext autoreload
%autoreload 2

# Control figure size
figsize=(14, 4)

from util import util
from matplotlib import pyplot as plt
import numpy as np
import seaborn as sn
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import roc_auc_score
from sklearn.model_selection import GridSearchCV
import xgboost
from sklearn.inspection import permutation_importance
import shap
import pickle
import os

# Generate synthetic data
data, name_map = util.generate_data(size=500, seed=42)
num_cols = [c for c in data.columns[:-1] if len(data[c].unique()) > 2]
cat_cols = [c for c in data.columns[:-1] if len(data[c].unique()) == 2]

# Data pre-processing
X, y = data[data.columns[:-1]].copy(), data[data.columns[-1]].copy()
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
scaler = StandardScaler()
X_train[num_cols] = scaler.fit_transform(X_train[num_cols])
X_test[num_cols] = scaler.transform(X_test[num_cols])

# Train a GBT model
base_est = xgboost.XGBClassifier(tree_method='hist', importance_type='total')
param_grid={'max_depth': [2, 3, 4], 'n_estimators': list(range(20, 41, 5))}
gscv = GridSearchCV(base_est, param_grid=param_grid)
gscv.fit(X, y)
xbm, xbm_params = gscv.best_estimator_, gscv.best_params_

```

```

/Users/michelelombardi/Library/Caches/pypoetry/virtualenvs/06-at-RVmPecHn-py
3.11/lib/python3.11/site-packages/tqdm/auto.py:21: TqdmWarning: IPProgress no
t found. Please update jupyter and ipywidgets. See https://ipywidgets.readth
edocs.io/en/stable/user_install.html
  from .autonotebook import tqdm as notebook_tqdm

```

Additive Feature Attribution

What we Gained, What we Lost

When we switched from Logistic Regression to GBTs we gained a lot

- A reliable proxy model
- A well defined and transparent feature importance definition
- Sparse and reliable importance scores

However, we also lost something:

With Linear Regression, we used to be able to:

- Identify the *direction* of the correlation (through the coefficient sign)
- ...And explain *individual examples*, by looking at the difference:

$$\theta^T x - \mathbb{E}_{x' \in P(X)} [\theta^T x']$$

Explaining Individual Examples

Let's look again at the last equation:

$$\theta^T x - \mathbb{E}_{x' \in P(X)} [\theta^T x']$$

- Assuming $P(X)$ is approximated by using a sample...
- ...Then $\mathbb{E}_{x' \in P(X)} [\theta^T x']$ is just the average prediction on the data

I.e. it is the prediction we could make *without access to any input value*

Therefore, the difference above represents the gap between:

- ...What we can predict given all information on one example
- ...And what we can predict with no such information

It's the collective *value of all available information*

Explaining Individual Examples

Due to linearity, the formula can be rewritten as:

$$\theta^T x - \mathbb{E}_{x' \in P(X)} [\theta^T x'] = \theta^T (x - \mathbb{E}_{x' \in P(X)} [x']) \quad (1)$$

$$= \sum_{j=1}^n \theta_j (x_j - \mathbb{E}_{x'_j \in P(X_j)} [x'_j]) \quad (2)$$

Meaning that we can assign a value *to every input attribute*:

- If we know the attribute, the model output moves from the trivial prediction
- ...And the change is given by $\phi_j(x) = \theta_j(x_j - \mathbb{E}_{x'_j \in P(X_j)}[x'_j])$

We call $\phi_j(x)$ the *effect* of attribute j for the example x

Can we generalize this process to non-linear models?

Additive Feature Attribution

Given an example x , we can try to build an additive attribution model:

$$g(z, x) = \phi_0 + \sum_{j=1}^n \phi_j(x) z_j \quad \text{with: } z_j \in \{0, 1\}$$

- Where z_j is called a *simplified input*
- ...And represents the fact that the j -th attribute is known or unknown

Intuitively, we build a linear explanation for the model local behavior

- Several ML explainability approaches can be seen as attempts at this
- ...Most notably the original LIME method

Shapely Values

How do we build the additive attribution model?

- We've already seen how to do it for linear models
- ...But for non-linear models the input features *interact* with each other

A possible solution: *marginalizing* over all subset of remaining features

Let \mathcal{X} be the set of all input features; then we have:

$$\phi_j(x) = \sum_{\mathcal{S} \subset \mathcal{X} \setminus j} \frac{|\mathcal{S}|!(n - |\mathcal{S}| - 1)!}{n!} (\hat{f}(x_{\mathcal{S} \cup j}) - \hat{f}(x_{\mathcal{S}}))$$

- The sum is over all subsets that do not contain feature j
- The coefficient ensures normalization
- $\hat{f}(x_{\mathcal{S}})$ is the model evaluate with only features in \mathcal{S}

Shapely Values

The result of our marginalization:

$$\phi_j(x) = \sum_{S \subset \mathcal{X} \setminus j} \frac{|\mathcal{S}|!(n - |\mathcal{S}| - 1)!}{n!} (\hat{f}(x_{S \cup j}) - \hat{f}(x_S))$$

...Are known as *Shapely values*

- They originate from game theory
- ...In a setup where we want to assign credit to multiple actors for a result
- The actors correspond to our input features, the result to the model output

Shapely values are the *only* attribution model with some key properties

SHAP

Using Shapely values for explanation become prominent with [this paper](#)

The work makes a number of contributions:

- It introduces the general idea of additive feature attribution
- It shows how several previous approaches fall into that category
- It show how Shapely values provide "ideal" attribution scores
- It introduces multiple techniques to approximate the values

Computing Shapely values can be very expensive, for two reasons:

- There is exponential number of terms in the sum
- Many ML models do not support missing values

Kernel SHAP

Those issues can be sidestepped by learning a *local linear approximator*

Given an example x , we can:

- Sample multiple simplified vectors z' of simplified inputs z from $\{0, 1\}^n$
- For every sampled vector, we construce an example:
 - For all j s.t. $z'_j = 1$, we put $x'_j = x_j$ in the example
 - We sample all x' s.t. $z'_j = 0$ from a *background set*
- We train a particular type of linear model on the obtained examples
- ...Then we compute the Shapely values using the linear formula

By sampling from the background we marginalize out "missing" attributes

Typically, we use as a background the training set or a sample of that

Kernel SHAP

The method we have just described is referred to as Kernel-SHAP

It works even if we used *kernels* computed on the original features

- E.g. we can group multiple features, or apply non-linear transformations
- In that case, the Shapely values will apply to the kernels

Other approximation/computation methods have been defined

- DeepSHAP for Deep NNs
- TreeSHAP for tree models
- ...

Note: beware of TreeShap, it is fast an exact, but it *relies on a slightly different semantic!*
Be sure to understand the method you choose to use

SHAP in Action

The authors of the SHAP paper maintain [a nice Python package](#)

...Which we are going to use to *explain* our non-linear model

```
In [2]: f = lambda x: xbm.predict_proba(x)[: ,1]
explainer = shap.KernelExplainer(f, shap.sample(X_train, 100), link='logit')
shap_values = explainer(X_test)
with open(os.path.join '..', 'data', 'shap_values.pickle'), 'wb' as fp:
    pickle.dump(shap_values, fp)
```

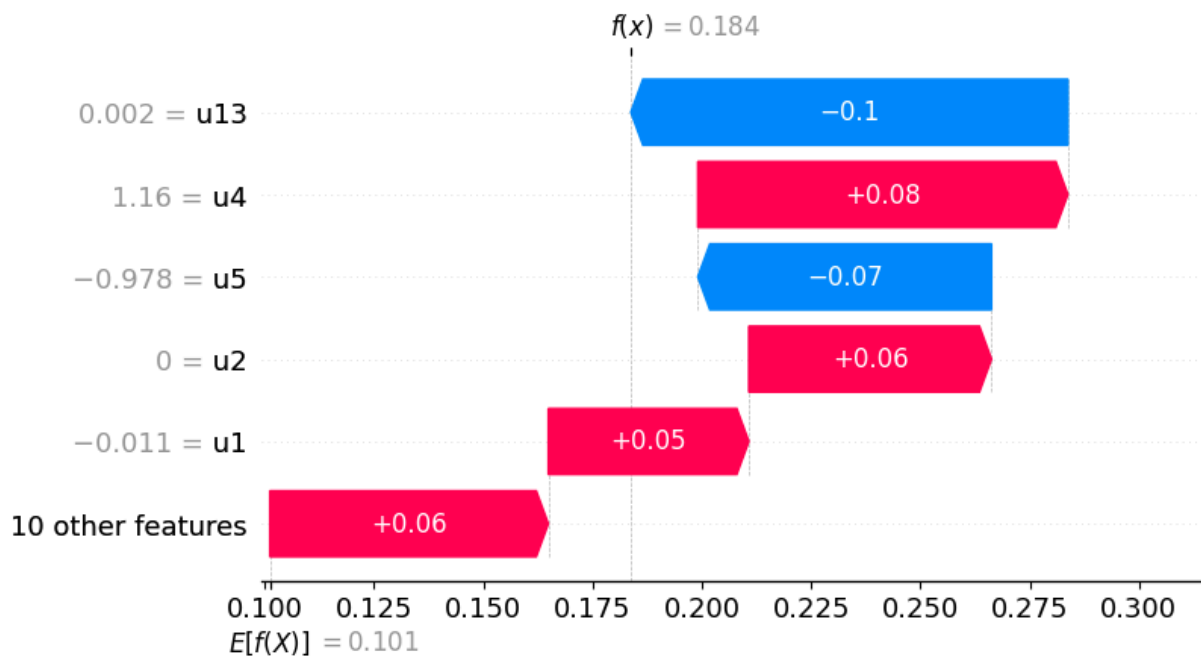
[illegible]

- We'll focus on the *test* data, since we want to find the true correlates
- For classifiers, it's easier to explain *logits* rather than probabilities
- The process can be slow, and using a small background set is recommended
- The result contains the Shapely values, the base values, and the original data

Waterfall Plots

The SHAP library allows us to build *waterfall plots*

```
In [3]: shap.plots.waterfall(shap_values[0], max_display=6)
```



- The bars represent the Shapely values, the colors their sign

Force Plots

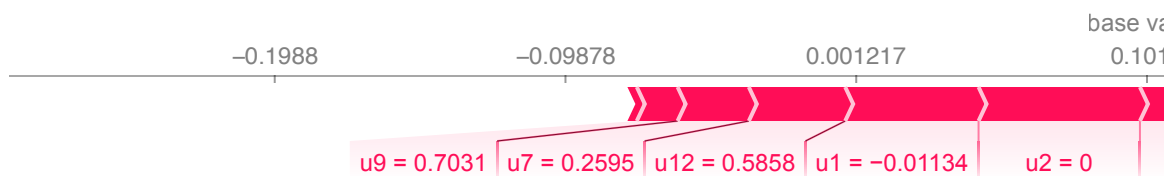
Waterfall plots can be "compacted" into *force plots*

Here we have again a plot for example 0:

```
In [4]: shap.initjs()
shap.plots.force(shap_values[0])
```



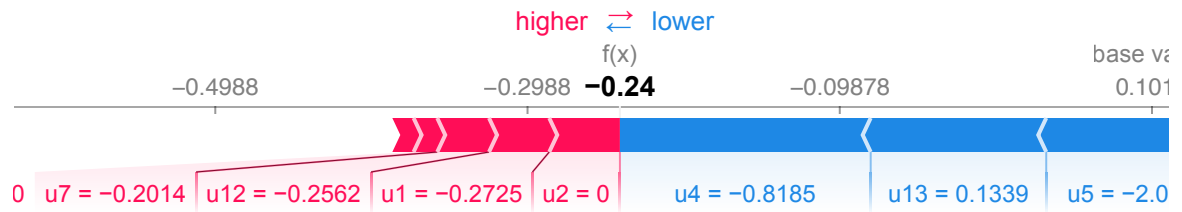
Out[4]:



...And have a plot for example 99

```
In [5]: shap.plots.force(shap_values[99])
```

Out [5]:

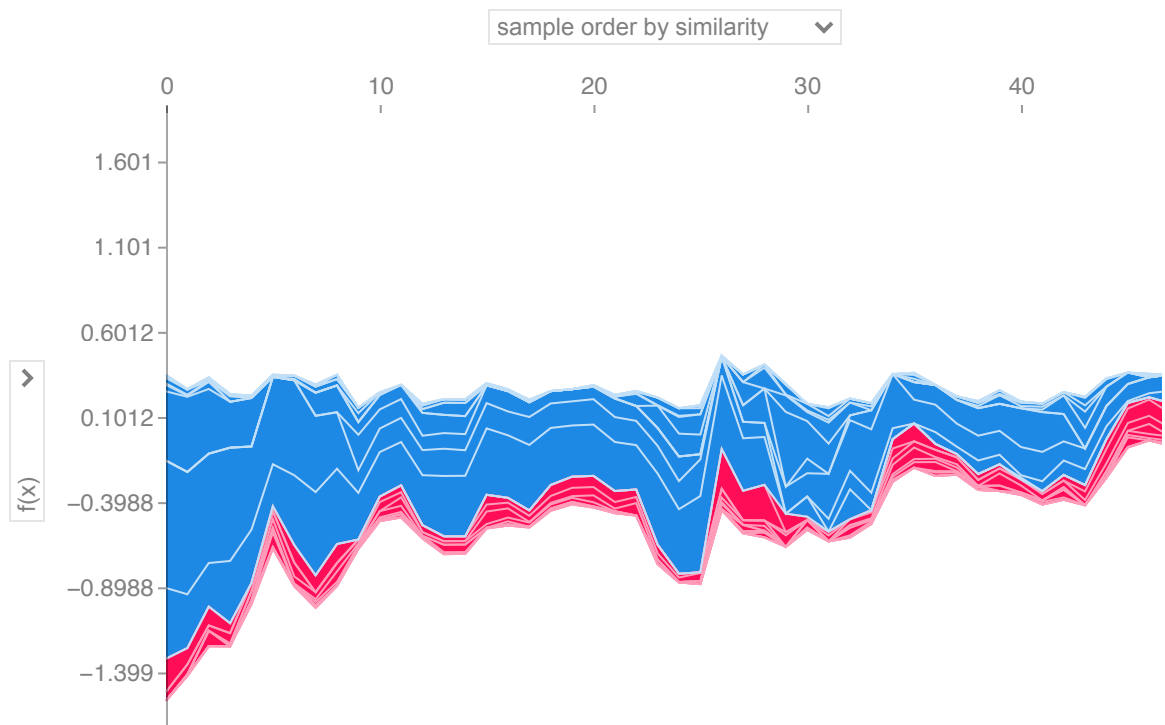


Global Force Plots

Force plots can be stacked to inspect many examples at once:

In [6]: `shap.plots.force(shap_values)`

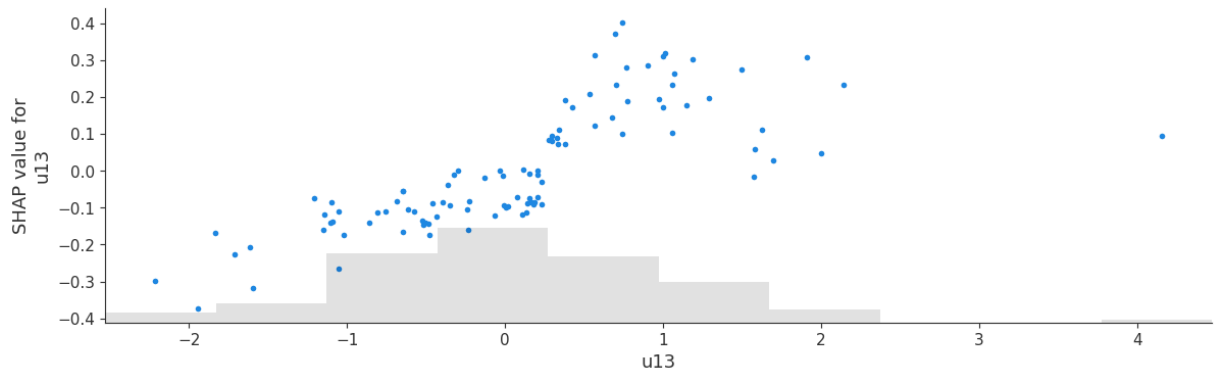
Out [6]:



Scatter Plots

We can use *scatter plots* to show the effect of a single feature

In [7]: `plt.figure(figsize=figsize)`
`shap.plots.scatter(shap_values[:, 'u13'], ax=plt.gca())`

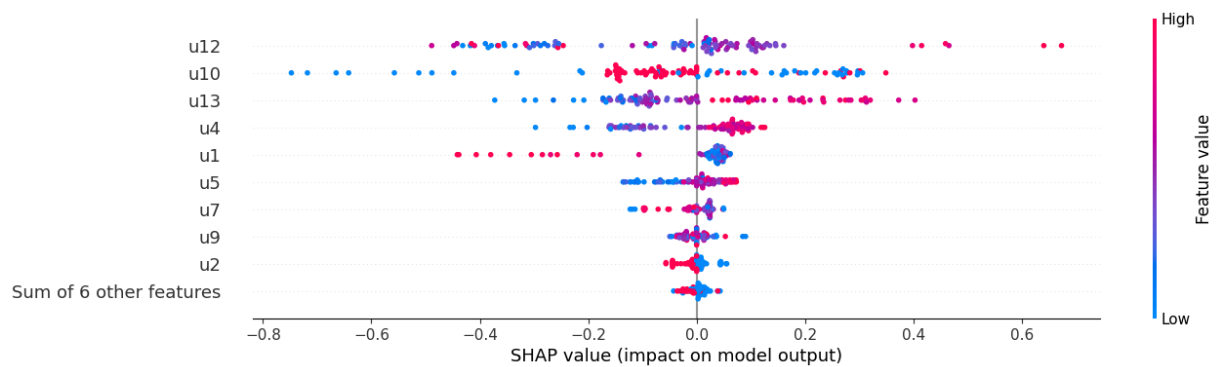


- The gray area is the histogram of the chosen feature

Beeswarm (Summary) Plot

We can stack (and color) multiple scatter plots to obtain a *beeswarm* plot:

```
In [8]: shap.plots.beeswarm(shap_values, max_display=10, plot_size=figsize)
```

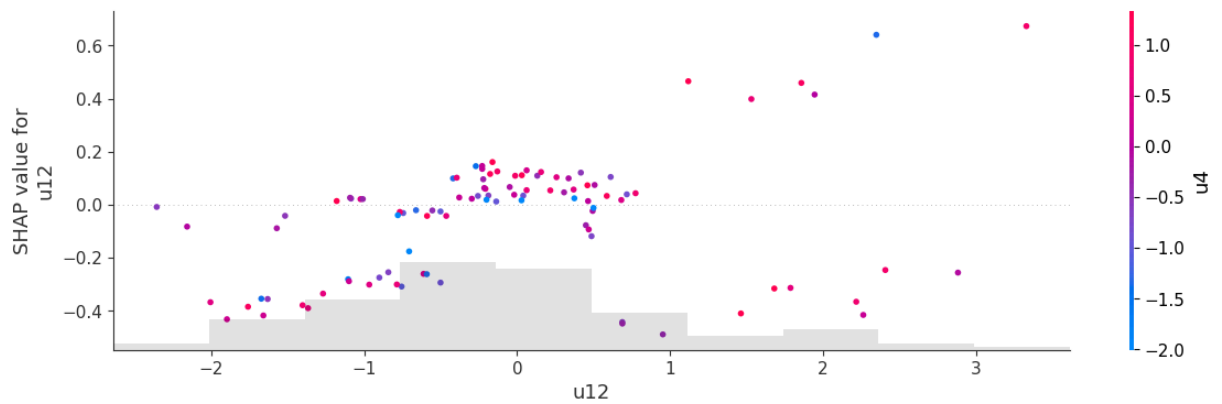


- By checking the color distribution we can identify (e.g.) monotonic effects

Scatter (Dependency) Plots

We can color scatter plots *by using another feature* to highlight dependency

```
In [9]: plt.figure(figsize=figsize)
shap.plots.scatter(shap_values[:, 'u12'], color=shap_values[:, 'u4'], ax=plt
```

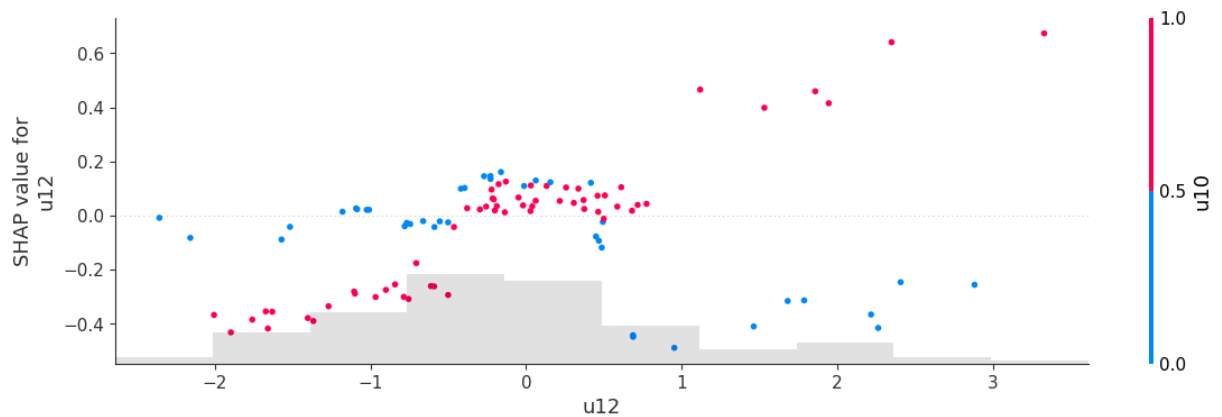


- In this case we are coloring the "u12" values by using "u4"

Scatter (Dependency) Plots

We can let the library choose the best coloring feature

```
In [10]: plt.figure(figsize=figsize)
shap.plots.scatter(shap_values[:, 'u12'], color=shap_values, ax=plt.gca())
```



- The chosen coloring feature changes how "u12" impacts the output in a noticeable way