# 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

# 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:

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

# Explaining Individual Examples

**Let's look again at the last equation:**

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

- Assuming $P(X)$ is approximated by using a sample...
- ...Then $\mathbb{E}_{x' \in P(X)} \left[\theta^T x'\right]$ 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:**

$$\theta^T x - \mathbb{E}_{x' \in P(X)}\left[\theta^T x'\right] = \theta^T(x - \mathbb{E}_{x' \in P(X)}[x'])$$
$$= \sum_{j=1}^{n} \theta_j (x_j - \mathbb{E}_{x'_j \in P(X_j)}[x'_j])$$

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 $x$

# Explaining Individual Examples

**Due to linearity, the formula can be rewritten as:**

$$\theta^T x - \mathbb{E}_{x' \in P(X)}\left[\theta^T x'\right] = \theta^T (x - \mathbb{E}_{x' \in P(X)}[x'])$$
$$= \sum_{j=1}^{n} \theta_j (x_j - \mathbb{E}_{x'_j \in P(X_j)}[x'_j])$$

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 explaination 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:

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

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

# Shapely Values

**The result of our marginalization:**

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

...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:
  - Forall $j$ 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 slighly 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 [4]: 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)
```

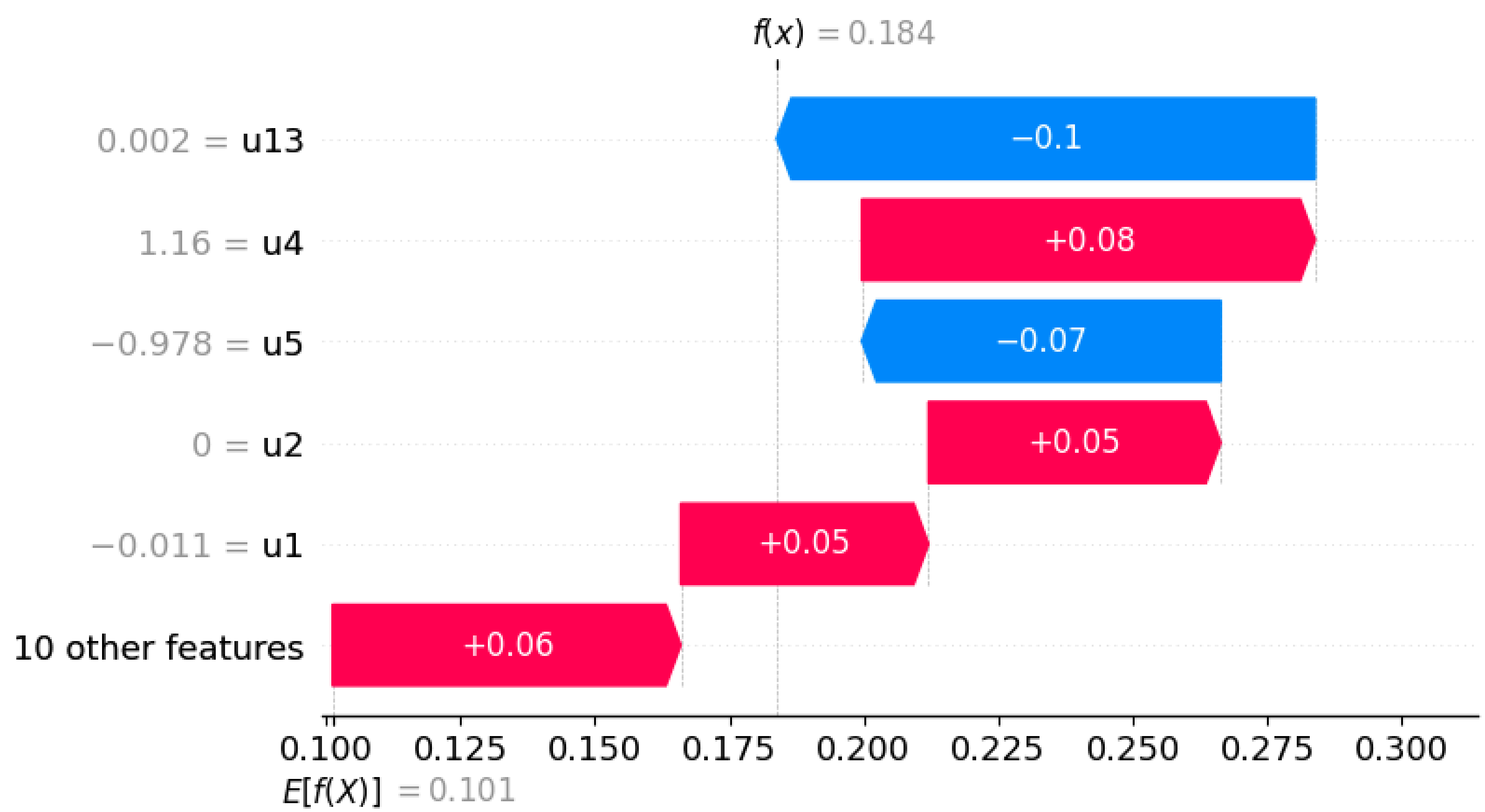
```
100%|██████████████████████████████████████████| 100/100 [00:02<0
0:00, 47.23it/s]
```

- We'll focus on the test data, since we want to find the true correlates
- For classifiers, it 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 [5]: shap.plots.waterfall(shap_values[0], max_display=6)
```




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

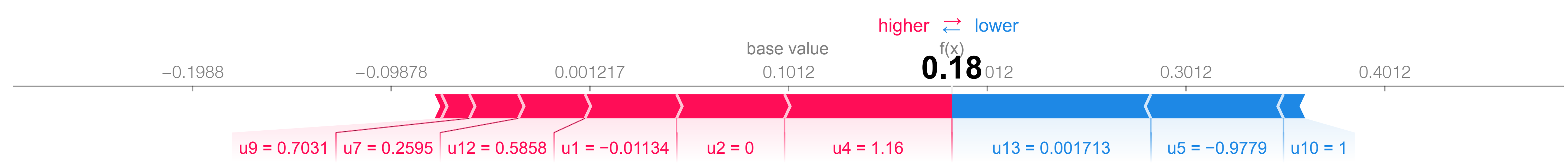
# Force Plots

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

Here we have again a plot for example 0:

In [6]:
```
shap.initjs()
shap.plots.force(shap_values[0])
```

Out[6]:




...And have a plot for example 99

In [7]:
```
shap.plots.force(shap_values[99])
```

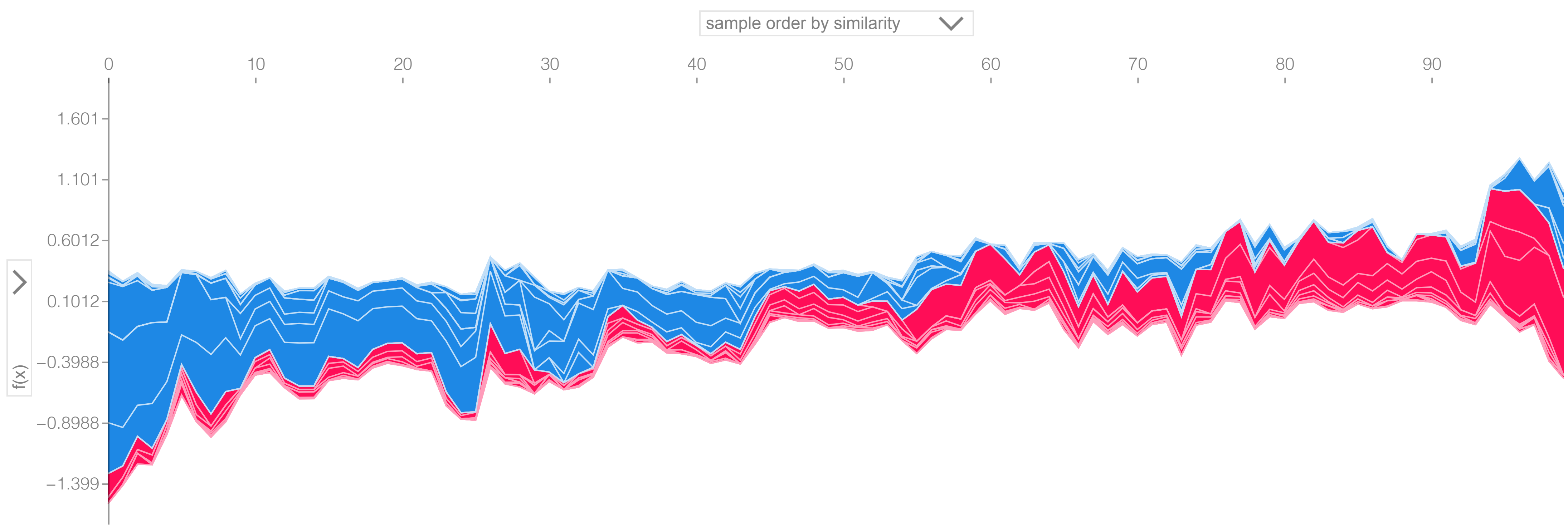
Out[7]:

# Global Force Plots

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

In [8]:
```
shap.plots.force(shap_values)
```

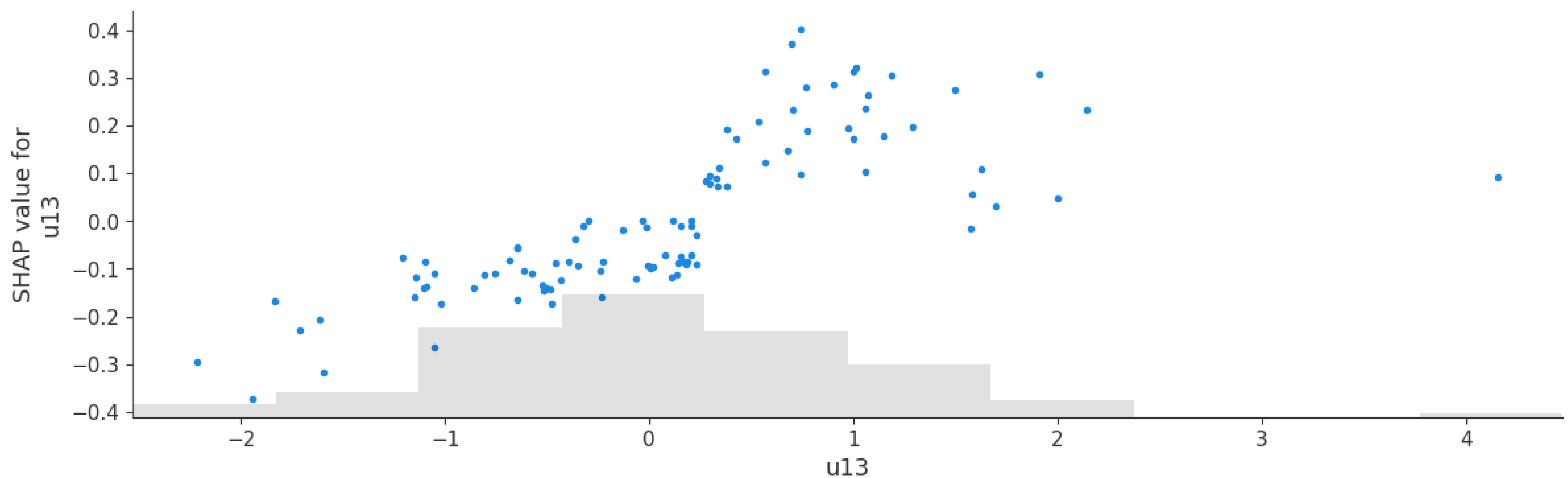
Out[8]:



# Scatter Plots

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

In [9]:
```
plt.figure(figsize=figsize)
shap.plots.scatter(shap_values[:, 'u13'], ax=plt.gca())
```

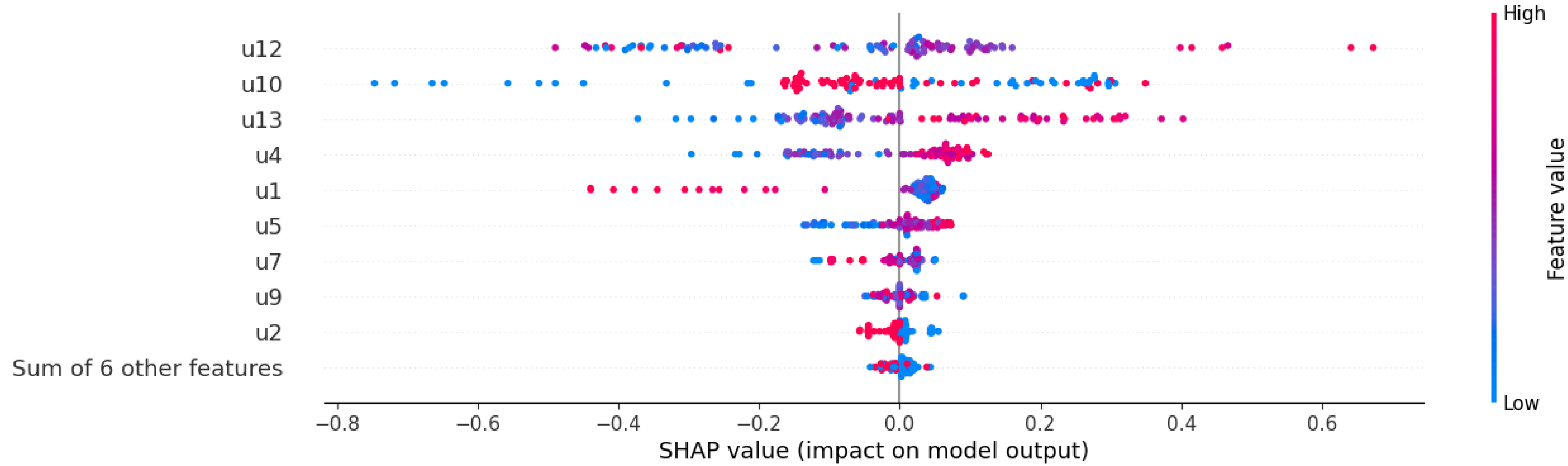



- The gray area is the histogram of the chosen feature

# Beeswarm (Summary) Plot

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

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

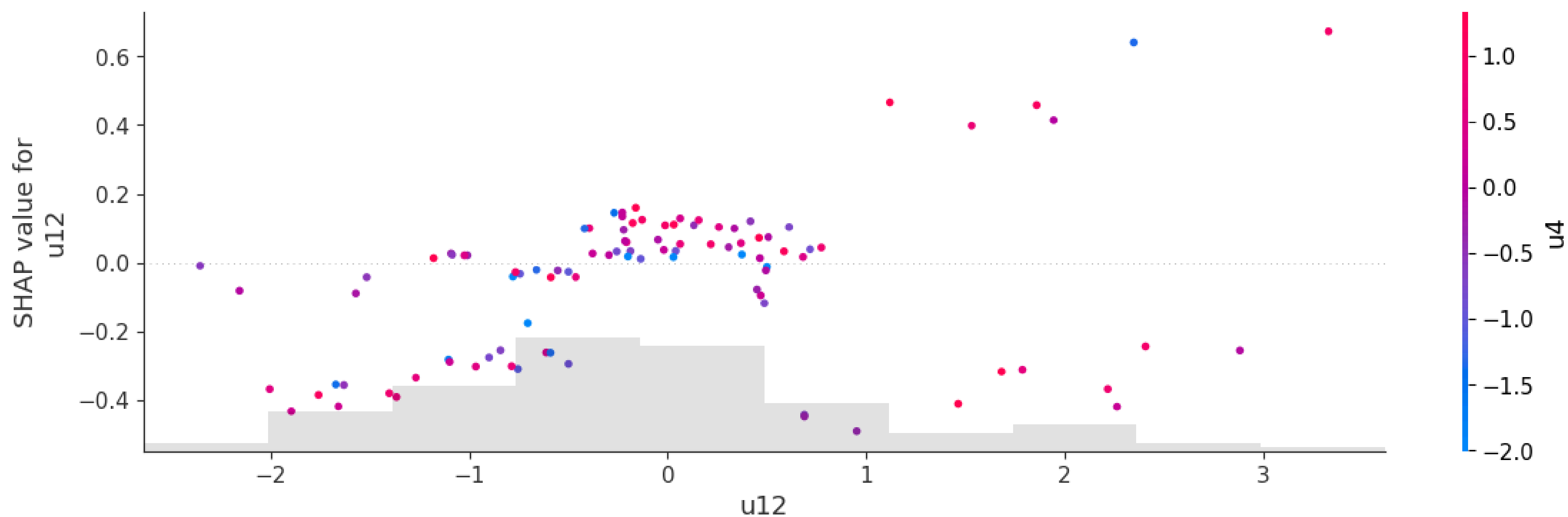



- By checking the color distribution we can indentify (e.g.) monotonic effects

# Scatter (Dependency) Plots

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

In [11]:
```
plt.figure(figsize=figsize)
shap.plots.scatter(shap_values[:, 'u12'], color=shap_values[:, 'u4'], ax=plt.gca())
```

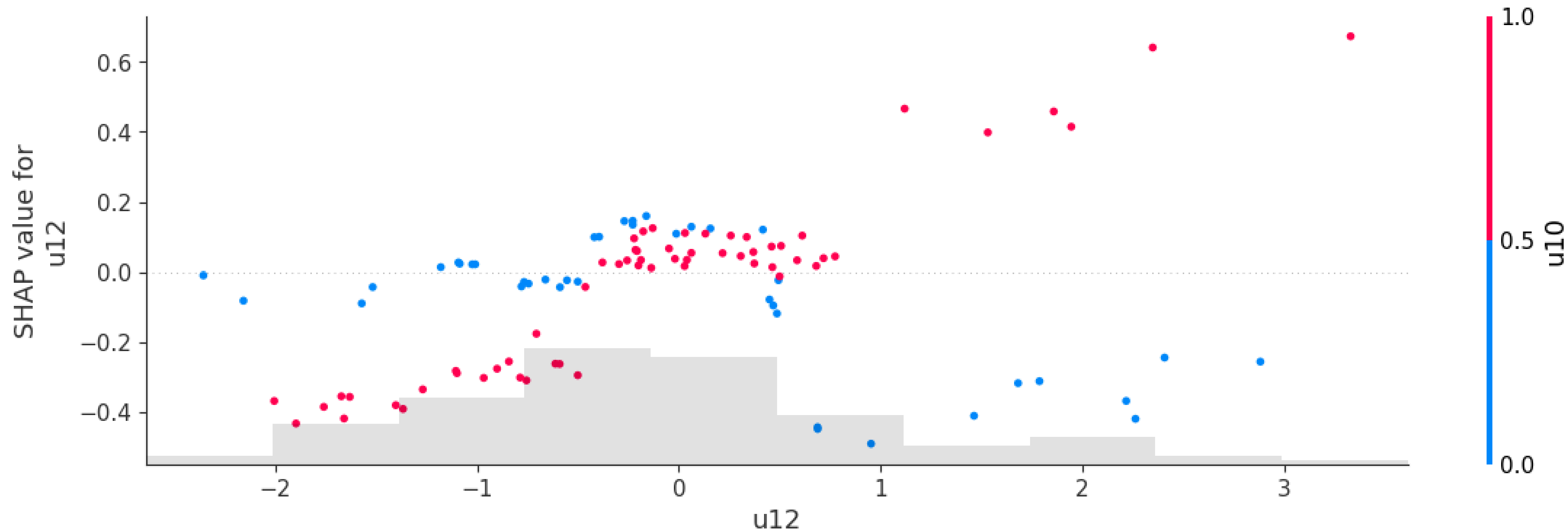



- In this case we are coloring the "u12" values by using "u4"

# Scatter (Dependency) Plots

## We can let the library choose the best coloring feature

In [12]:
```
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