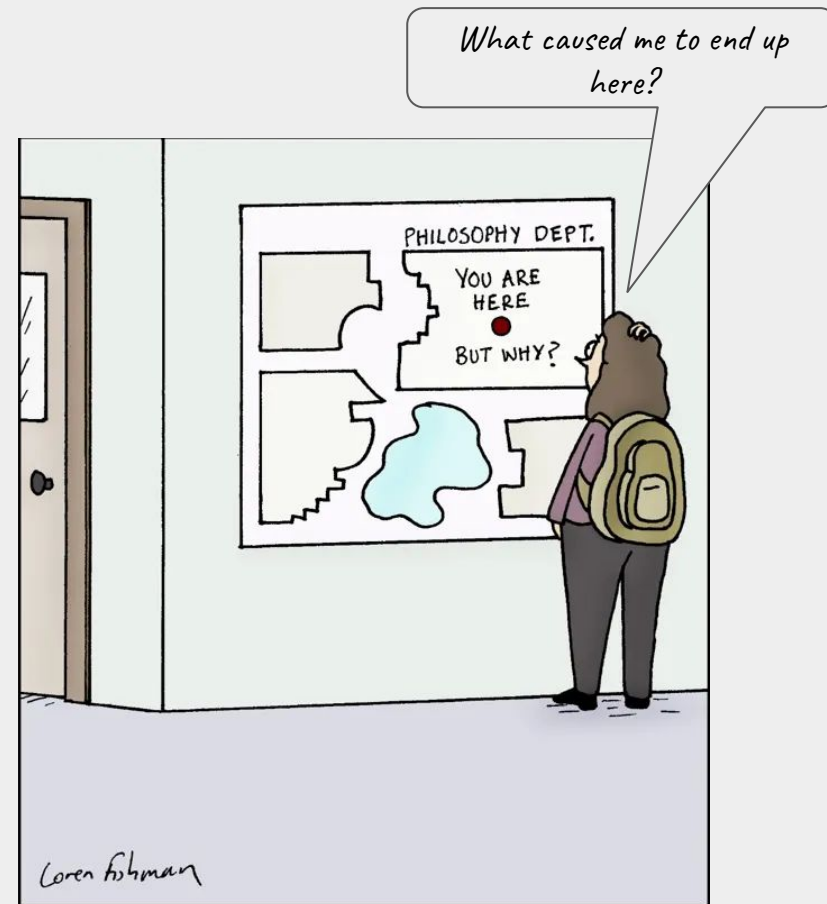


Explainable AI - Lecture 5

SHAP

Before we start...

How is the Shapley decomposition doing?



Recap: The Shapley decomposition in machine learning

The Shapley decomposition

- is a solution concept from cooperative game theory
- distributes the outcome of a game fairly among the players
- satisfies the four axioms of efficiency, symmetry, dummy and additivity.

We can go from **game theory** to **machine learning** by re-interpreting...

	Game theory	Machine learning
N	Grand coalition	All features
S	Coalitions	Sets of features
i	Player index	Feature index
v	Characterises the game	Characterises the model

Recap: The Shapley decomposition in machine learning

The Shapley decomposition:

- takes as input a set function $v : 2^N \rightarrow \mathbb{R}$ which maps a set of input features to a single real number,
- produces an attribution ϕ_i for each feature $i \in N$, that add up to $v(N)$.

This attribution, the Shapley value, of feature i is given by

$$\phi_i(v) = \sum_{S \subseteq N \setminus \{i\}} \frac{|S|!(N - |S| - 1)!}{N!} (v(S \cup \{i\}) - v(S))$$

Recap: The Shapley decomposition in machine learning

The Shapley decomposition:

- takes as input a set function $v : 2^N \rightarrow \mathbb{R}$ which maps a set of input features to a single real number,
- produces an attribution ϕ_i for each feature $i \in N$, that add up to $v(N)$.

This attribution, the Shapley value, of feature i is given by

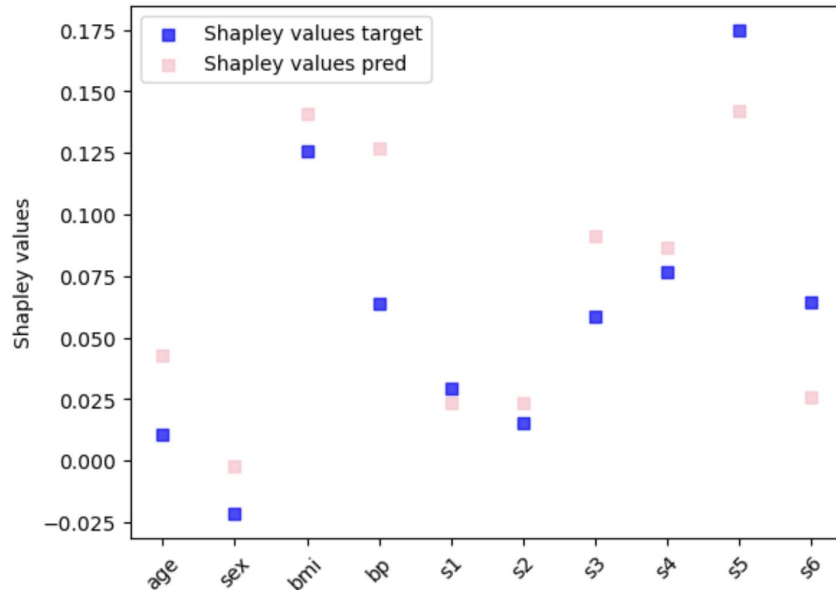
$$\phi_i(v) = \sum_{S \subseteq N \setminus \{i\}} \frac{|S|!(N - |S| - 1)!}{N!} (v(S \cup \{i\}) - v(S))$$

Central to the computation are the *marginal contributions*, which are calculated by adding and removing features from coalitions.

As each of the N features must be both included and excluded, the computation requires 2^N evaluations of v .

Recap: The Shapley values for a model

We calculated Shapley values for the features of a model by using as characteristic function v the correlation between the features and the targets / model predictions.



The marginal contribution for a machine learning model

Today, we want to use the model itself as characteristic function v .

In order to calculate the marginal contribution of a feature, we then need to be able to **remove features from the input** to the model.

$$\phi_i(v) = \sum_{S \subseteq N \setminus \{i\}} \frac{|S|!(N - |S| - 1)!}{N!} (v(S \cup \{i\}) - v(S))$$

How to remove features from a trained model?

The marginal contribution for a machine learning model

Today, we want to use the model itself as characteristic function v .

In order to calculate the marginal contribution of a feature, we then need to be able to **remove features from the input** to the model.

$$\phi_i(v) = \sum_{S \subseteq N \setminus \{i\}} \frac{|S|!(N - |S| - 1)!}{N!} (v(S \cup \{i\}) - v(S))$$

We can't remove features from a trained model!

The marginal contribution for a machine learning model

Model building:

1. Decide on input and output shape
2. Build the model architecture based on this
3. Adjust parameters to optimise the loss

$$\phi_i(v) = \sum_{S \subseteq N \setminus \{i\}} \frac{|S|!(N - |S| - 1)!}{N!} (v(S \cup \{i\}) - v(S))$$

Trying to evaluate a model with missing input yields a shape error (or something similar).

The marginal contribution for a machine learning model

There are ways to *estimate* the Shapley values for model features by simulating feature absence instead of actually excluding features.

Let's have a look at the formalism.

The marginal contribution for a machine learning model

Remember the human-interpretable feature space from LIME, representing feature presence or absence.

Define:

- the simplified input x' to represent interpretable inputs, usually binary vectors.
- the mapping $x = h_x(x')$ to convert between the full input x and the simplified input x' .

The marginal contribution for a machine learning model

Remember the human-interpretable feature space from LIME, representing feature presence or absence.

Define:

- the simplified input x' to represent interpretable inputs, meaning binary vectors.
- the mapping $x = h_x(x')$ to convert between the full input x and the simplified input x' .

We have different mappings h_x for different input spaces:

- For tabular data, h_x converts a vector of 1's and 0's to the tabular values.
- For images, h_x treats the image as a set of super pixels. An entry of 1 in x' leaves the super pixel at its original value and 0 “masks” the super pixel (commonly using an average pixel value).
- For language models, h_x can map missing tokens to a mask token if available, to an empty string, or some “neutral” baseline token intended to have minimal semantic content.

The marginal contribution for a machine learning model

Remember the human-interpretable feature space from LIME, representing feature presence or absence.

Define:

- the simplified input x' to represent interpretable inputs, meaning binary vectors.
- the mapping $x = h_x(x')$ to convert between the full input x and the simplified input x' .

The mapping h_x is specific for the point x (hence the subscript).

For example: if x is an image, then the mapping h_x from the value 1 is the original value in x .

The marginal contribution for a machine learning model

Using the feature / interpretable space notation, we go from the Shapley decomposition

$$\phi_i(v) = \sum_{S \subseteq N \setminus \{i\}} \frac{|S|!(N - |S| - 1)!}{N!} (v(S \cup \{i\}) - v(S))$$

to

$$\phi_i(f, x) = \sum_{z' \subseteq x'} \frac{|z'|!(M - |z'| - 1)!}{M!} [f_x(z') - f_x(z' \setminus i)]$$

where M is the number of simplified input features.

The marginal contribution for a machine learning model

Using the feature / interpretable space notation, we go from the Shapley decomposition

$$\phi_i(v) = \sum_{S \subseteq N \setminus \{i\}} \frac{|S|!(N - |S| - 1)!}{N!} (v(S \cup \{i\}) - v(S))$$

to

$$\phi_i(f, x) = \sum_{z' \subseteq x'} \frac{|z'|!(M - |z'| - 1)!}{M!} [f_x(z') - f_x(z' \setminus i)]$$

The sum is over all vectors z' of which the non-zero entries are a subset of the non-zero entries in x' .

The marginal contribution for a machine learning model

Using the feature / interpretable space notation, we go from the Shapley decomposition

$$\phi_i(v) = \sum_{S \subseteq N \setminus \{i\}} \frac{|S|!(N - |S| - 1)!}{N!} (v(S \cup \{i\}) - v(S))$$

to

$$\phi_i(f, x) = \sum_{z' \subseteq x'} \frac{|z'|!(M - |z'| - 1)!}{M!} [f_x(z') - f_x(z' \setminus i)]$$

The sum is over all vectors z' of which the non-zero entries are a subset of the non-zero entries in x' .

The size $|z'|$ is the number of non-zero entries in z .

The marginal contribution for a machine learning model

Using the feature / interpretable space notation, we go from the Shapley decomposition

$$\phi_i(v) = \sum_{S \subseteq N \setminus \{i\}} \frac{|S|!(N - |S| - 1)!}{N!} (v(S \cup \{i\}) - v(S))$$

to

$$\phi_i(f, x) = \sum_{z' \subseteq x'} \frac{|z'|!(M - |z'| - 1)!}{M!} [f_x(z') - f_x(z' \setminus i)]$$

The sum is over all vectors z' of which the non-zero entries are a subset of the non-zero entries in x' .

The size $|z'|$ is the number of non-zero entries in z .

$z' \setminus i$ denotes setting $z'_i = 0$.

The marginal contribution for a machine learning model

Using the feature / interpretable space notation, we go from the Shapley decomposition

$$\phi_i(v) = \sum_{S \subseteq N \setminus \{i\}} \frac{|S|!(N - |S| - 1)!}{N!} (v(S \cup \{i\}) - v(S))$$

to

$$\phi_i(f, x) = \sum_{z' \subseteq x'} \frac{|z'|!(M - |z'| - 1)!}{M!} [f_x(z') - f_x(z' \setminus i)]$$

The sum is over all vectors z' of which the non-zero entries are a subset of the non-zero entries in x' .

The size $|z'|$ is the number of non-zero entries in z .

$z' \setminus i$ denotes setting $z'_i = 0$.

The shorthand $f_x(z') = f(h_x(z'))$ means using h_x to evaluate the model in the feature space.

The marginal contribution for a machine learning model

$$\phi_i(f, x) = \sum_{z' \subseteq x'} \frac{|z'|!(M - |z'| - 1)!}{M!} [f_x(z') - f_x(z' \setminus i)]$$

Our problems are the $f_x(z')$ and $f_x(z' \setminus i)$, meaning all model evaluations with missing features.

We only know how to evaluate the model when all input features are present.

The marginal contribution for a machine learning model

$$\phi_i(f, x) = \sum_{z' \subseteq x'} \frac{|z'|!(M - |z'| - 1)!}{M!} [f_x(z') - f_x(z' \setminus i)]$$

Our problems are the $f_x(z')$ and $f_x(z' \setminus i)$, meaning all model evaluations with missing features.

We only know how to evaluate the model when all input features are present.

Define S : the present features, meaning the **set of non-zero** indices in z' .

We rewrite $f_x(z')$ using $f(z)$ and z_S as follows:

$$f_x(z') = f(h_x(z')) = E[f(z) | z_S]$$

As before, $f(z)$ means that we evaluate the model f on the input space version of z' .

The marginal contribution for a machine learning model

$$\phi_i(f, x) = \sum_{z' \subseteq x'} \frac{|z'|!(M - |z'| - 1)!}{M!} [f_x(z') - f_x(z' \setminus i)]$$

Our problems are the $f_x(z')$ and $f_x(z' \setminus i)$, meaning all model evaluations with missing features.

We only know how to evaluate the model when all input features are present.

Define S : the present features, meaning the set of non-zero indices in z' .

We rewrite $f_x(z')$ using $f(z)$ and z_S as follows:

$$f_x(z') = f(h_x(z')) = E[f(z) | z_S]$$

This represents the **expected model output**, in the absence of, given that the values of the present features are those in S .

So: we calculate the expected value of the model output *conditioned upon* the values of the present features.

Simulating feature absence

This idea, that we can **simulate feature absence** by calculating (or estimating) the **conditional expectation**,

$$f_x(z') = f(h_x(z')) = E[f(z) | z_S]$$

is the core of the widely used XAI library **SHAP**.

SHAP is short for SHapley Additive exPlanations, and we'll see what's up with the “additive” in just a moment.

Simulating feature absence

This idea, that we can **simulate feature absence** by calculating (or estimating) the **conditional expectation**,

$$f_x(z') = f(h_x(z')) = E[f(z) | z_S]$$

is the core of the widely used XAI library **SHAP**.

SHAP is short for SHapley Additive exPlanations, and we'll see what's up with the “additive” in just a moment.

First, let's keep working on how to estimate this expectation, to get a value for $f_x(z')$.

Simulating feature absence

S : the set of non-zero indices in z' , representing present features

\bar{S} : the set of zero indices in z' , representing absent features

Simulating feature absence

S : the set of non-zero indices in z' , representing present features

\bar{S} : the set of zero indices in z' , representing absent features

The expectation of the model output conditioned upon the values of the present features S can be written as

$$\begin{aligned} f(h_x(z')) &= E[f(z) \mid z_S] \\ &= E_{z_{\bar{S}} \mid z_S}[f(z)] \end{aligned}$$

Simulating feature absence

S : the set of non-zero indices in z' , representing present features

\bar{S} : the set of zero indices in z' , representing absent features

The expectation of the model output conditioned upon the values of the present features S can be written as

$$\begin{aligned} f(h_x(z')) &= E[f(z) \mid z_S] \\ &= E_{z_{\bar{S}} \mid z_S}[f(z)] \end{aligned}$$

Remember: we need values for the missing features in z when evaluating f (or we'll get an error).

$E_{z_{\bar{S}} \mid z_S}$ means that the values of these *absent features* \bar{S} should be conditional upon the values of the *present features* S .

Simulating feature absence: SHAP

At this point, different SHAP implementations diverge.

We will use the [python library SHAP](#). It is the most widely used SHAP library, and one of the most widely used XAI libraries. The alternative is the [R library shapr](#) (*fun fact: made in Norway*)

SHAP is installed using `pip install shap` or `conda install -c conda-forge shap`, and we will look at code later.

SHAP

SHAP is installed using `pip install shap` or `conda install -c conda-forge shap`, and we will look at code later.

It was introduced in a highly cited NeurIPS paper by Lundberg & Lee (2017)

A Unified Approach to Interpreting Model Predictions

December 2017

DOI:[10.48550/arXiv.1705.07874](https://doi.org/10.48550/arXiv.1705.07874)

Conference: NIPS

Authors:



Scott Lundberg
University of Washington



Su-In Lee

This is today's additional resource, and a bit of a challenging read...

Simulating feature absence as Lundberg & Lee (2017)

S : the set of non-zero indices in z' , representing present features

\bar{S} : the set of zero indices in z' , representing absent features

In SHAP, the expectation of the model output conditioned upon the values of S is *approximated* as

$$\begin{aligned} f(h_x(z')) &= E[f(z) \mid z_S] \\ &= E_{z_{\bar{S}} \mid z_S} [f(z)] \\ &\approx E_{z_{\bar{S}}} [f(z)] \end{aligned}$$

Simulating feature absence as Lundberg & Lee (2017)

S : the set of non-zero indices in z' , representing present features

\bar{S} : the set of zero indices in z' , representing absent features

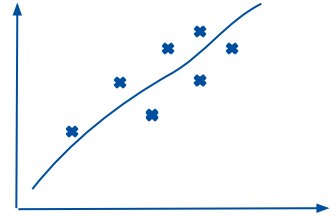
In SHAP, the expectation of the model output conditioned upon the values of S is approximated as

$$\begin{aligned} f(h_x(z')) &= E[f(z) \mid z_S] \\ &= E_{z_{\bar{S}} \mid z_S} [f(z)] \\ &\approx E_{z_{\bar{S}}} [f(z)] \end{aligned}$$

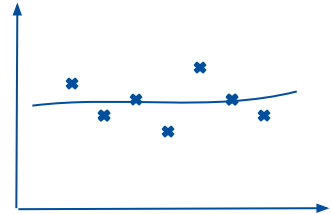
The expectation is calculated by sampling the “absent” features \bar{S} freely.

This implies the assumption that features are independent of each other (the values in S don't tell us anything about the values in \bar{S}).

Can you give an example of dependent features?

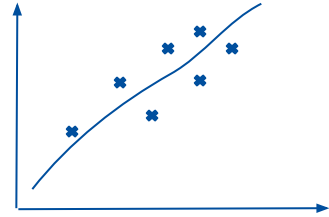


Can you give an example of independent features?



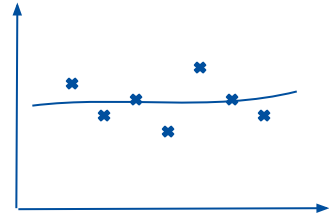
Examples of dependent features:

- Height and weight (*taller people tend to weigh more*).
- House size and number of rooms (*larger houses tend to have many rooms*)
- Age and price of a car (*newer cars tend to be more expensive*)



Examples of independent features:

- Blood pressure and shoe size
- Number of stripes on a zebra and the time of day
- Survey day of week and participant's height (*ask random people on the street, the day will not affect how tall people are*)



Summary, pat on the back

Summary so far

Good job! We've understood how we can “remove features” to calculate the marginal contribution in the Shapley decomposition for a machine learning model!

Notation summary:

- x : model input, in the feature space the model was trained on
- x' : simplified input, binary vectors representing feature absence or presence
- h_x : mapping between feature space input x and simplified input x'
- z' : perturbed simplified input, whose non-zero entries are a subset of the non-zero entries in x'
- $z = h_x(z')$: feature space version of z' , with missing features, also denoted z_S
- S : set of non-zero indices in z' , representing present features
- \bar{S} : set of zero indices in z' , representing the set of features not in S

Feature space F
with pixel values

x



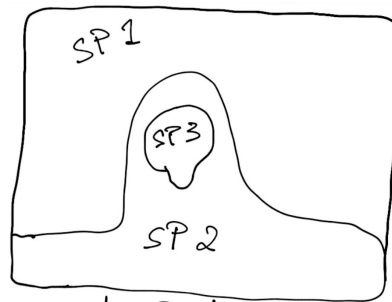
z

z_S



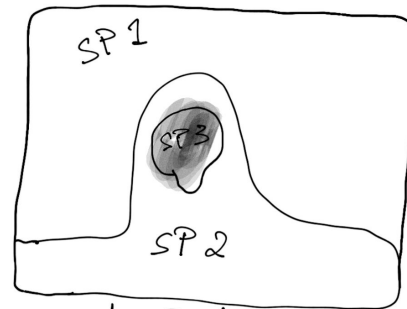
Human-interpretable space F'
with binary vectors

$h_x(x')$

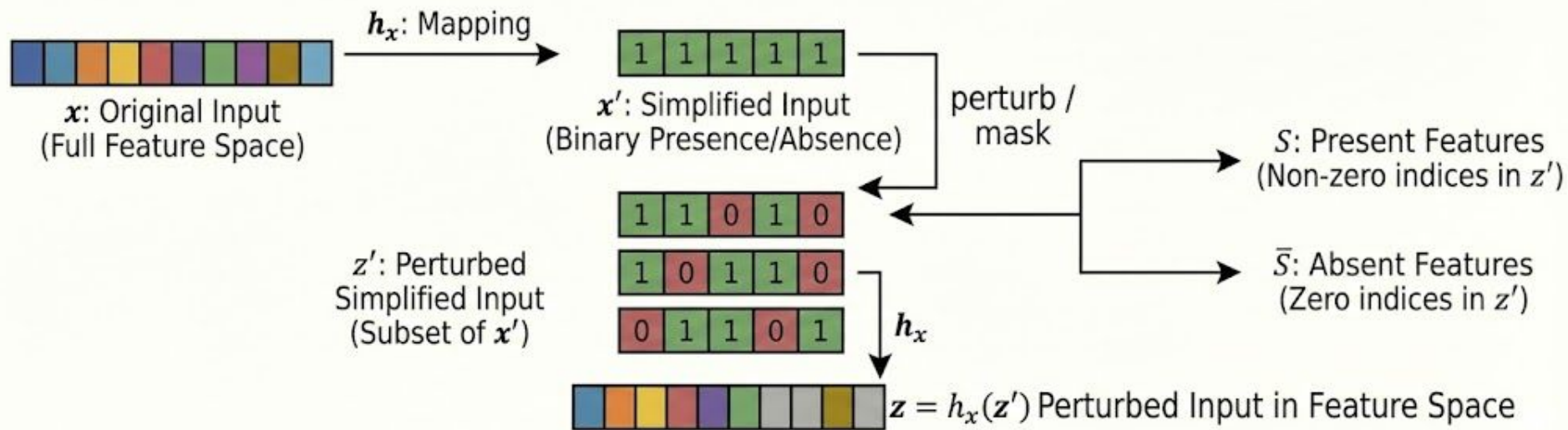


$$x' = \begin{array}{c|c|c} SP\ 1 & SP\ 2 & SP\ 3 \\ \hline 1 & 1 & 1 \end{array}$$

$h_x(z')$



$$z' = \begin{array}{c|c|c} SP\ 1 & SP\ 2 & SP\ 3 \\ \hline 1 & 1 & 0 \end{array}$$



Summary so far

The core idea in SHAP is that we can evaluate the model with missing input features as

$$f_x(z') = f(h_x(z')) = E[f(z) | z_S]$$

Summary so far

The core idea in SHAP is that we can evaluate the model with missing input features as

$$f_x(z') = f(h_x(z')) = E[f(z) | z_S]$$

SHAP as introduced in Lundberg & Lee (2017) further assumes independent features, to approximate

$$\begin{aligned} f(h_x(z')) &= E[f(z) | z_S] \\ &= E_{z_{\bar{S}} | z_S}[f(z)] \\ &\approx E_{z_{\bar{S}}}[f(z)] \end{aligned}$$

The interpretable model

or: How does this explain anything??

SHAP, continued

We have the interpretable feature space

(which kind of helped us solve one of the challenges of taking Shapley values from game theory to machine learning, since it operates with absent/present features. How lucky.)

but we still need a way to actually explain the model prediction. Here, SHAP makes use of an **interpretable model**, closely aligned with LIME.

Additive feature attribution

Lundberg & Lee (2017) define **additive feature attribution methods**: methods that present the model prediction as a linear combination of the interpretable features:

$$g(z') = \phi_0 + \sum_{i=1}^M \phi_i z'_i,$$

Additive feature attribution

Lundberg & Lee (2017) define **additive feature attribution methods**: methods that present the model prediction as a linear combination of the interpretable features:

$$g(z') = \phi_0 + \sum_{i=1}^M \phi_i z'_i,$$

You're used to this:

f is the original model,

g is the surrogate (explanation) model,

and local surrogate methods try to ensure that $g(z') \approx f(h_x(z'))$ whenever $z' \approx x'$, meaning **whenever the simplified input allows much of the information from the full feature space to be available.**

Additive feature attribution

For the surrogate model to be locally accurate, we need

$$f(x) = g(x') = \phi_0 + \sum_{i=1}^M \phi_i x'_i$$

Meaning that the surrogate model $g(x')$ matches the original model $f(x)$ when $x = h_x(x')$ (*as before, with LIME*)

Additive feature attribution

For the surrogate model to be locally accurate, we need

$$f(x) = g(x') = \phi_0 + \sum_{i=1}^M \phi_i x'_i$$

Meaning that the surrogate model $g(x')$ matches the original model $f(x)$ when $x = h_x(x')$ (as before, with LIME)

In this case (and now we're finally getting there, I promise), if our model can locally be written as a linear model, we can use the linearity of the expectation

$$E[X + Y] = E[X] + E[Y]$$

$$E[aX] = aE[X]$$

this is not some XAI / SHAP magic, this is just a property of the expectation. We cool?

Additive feature attribution

For the surrogate model to be locally accurate, we need

$$f(x) = g(x') = \phi_0 + \sum_{i=1}^M \phi_i x'_i$$

Meaning that the surrogate model $g(x')$ matches the original model $f(x)$ when $x = h_x(x')$ (as before, with LIME)

In this case (and now we're finally getting there, I promise), if our model can locally be written as a linear model, we can use the linearity of the expectation [to rewrite the expectation from earlier as](#)

$$E_{z_{\bar{S}}} [f(z)] \approx f([z_S, E[z_{\bar{S}}]])$$

Additive feature attribution

For the surrogate model to be locally accurate, we need

$$f(x) = g(x') = \phi_0 + \sum_{i=1}^M \phi_i x'_i$$

Meaning that the surrogate model $g(x')$ matches the original model $f(x)$ when $x = h_x(x')$ (as before, with LIME)

In this case (and now we're finally getting there, I promise), if our model can locally be written as a linear model, we can use the linearity of the expectation to rewrite the expectation from earlier as

$$E_{z_{\bar{S}}} [f(z)] \approx f([z_S, E[z_{\bar{S}}]])$$

The point is: As long as the model is linear, it doesn't matter if we

- first evaluate it for z_S and then calculate the expectation, or
- first calculate $E[z_{\bar{S}}]$ and then evaluate the model.

Additive feature attribution

For the surrogate model to be locally accurate, we need

$$f(x) = g(x') = \phi_0 + \sum_{i=1}^M \phi_i x'_i$$

Meaning that the surrogate model $g(x')$ matches the original model $f(x)$ when $x = h_x(x')$ (as before, with LIME)

In this case (and now we're finally getting there, I promise), if our model can locally be written as a linear model, we can use the linearity of the expectation to rewrite the expectation from earlier as

$$E_{z_{\bar{S}}} [f(z)] \approx f([z_S, E[z_{\bar{S}}]])$$

The point is: **As long as the model is linear**, it doesn't matter if we

- first evaluate it for $z_{\bar{S}}$ and then calculate the expectation, or
- first calculate $E[z_{\bar{S}}]$ and then evaluate the model.

SHAP all in all!

Locally, around x , our model can be represented by the linear surrogate model

$$f(x) = g(x') = \phi_0 + \sum_{i=1}^M \phi_i x'_i$$

And we estimate the model with absent features as follows

$$\begin{aligned} f(h_x(z')) &= E[f(z) \mid z_S] && \leftarrow \text{Mapping from simplified representation with missing features} \\ &= E_{z_{\bar{S}} \mid z_S} [f(z)] && \leftarrow \text{Rewriting the expectation of model value with missing features} \\ &\approx E_{z_{\bar{S}}} [f(z)] && \leftarrow \text{Assuming independent features } (z_{\bar{S}} \mid z_S = z_{\bar{S}}) \\ &\approx f([z_S, E[z_{\bar{S}}]]) && \leftarrow \text{Assuming } f \text{ can be represented linearly} \end{aligned}$$

When we do this, we get SHapley Additive exPlanation (SHAP) values.

SHAP all in all!

Locally, around x , our model can be represented by the linear surrogate model

$$f(x) = g(x') = \phi_0 + \sum_{i=1}^M \phi_i x'_i$$

ϕ_0 is the expected prediction of the model with empty input, and serves as a base value.

ϕ_i are the SHAP values of the M interpretable features with indices i .

SHAP all in all!

Locally, around x , our model can be represented by the linear surrogate model

$$f(x) = g(x') = \phi_0 + \sum_{i=1}^M \phi_i x'_i$$

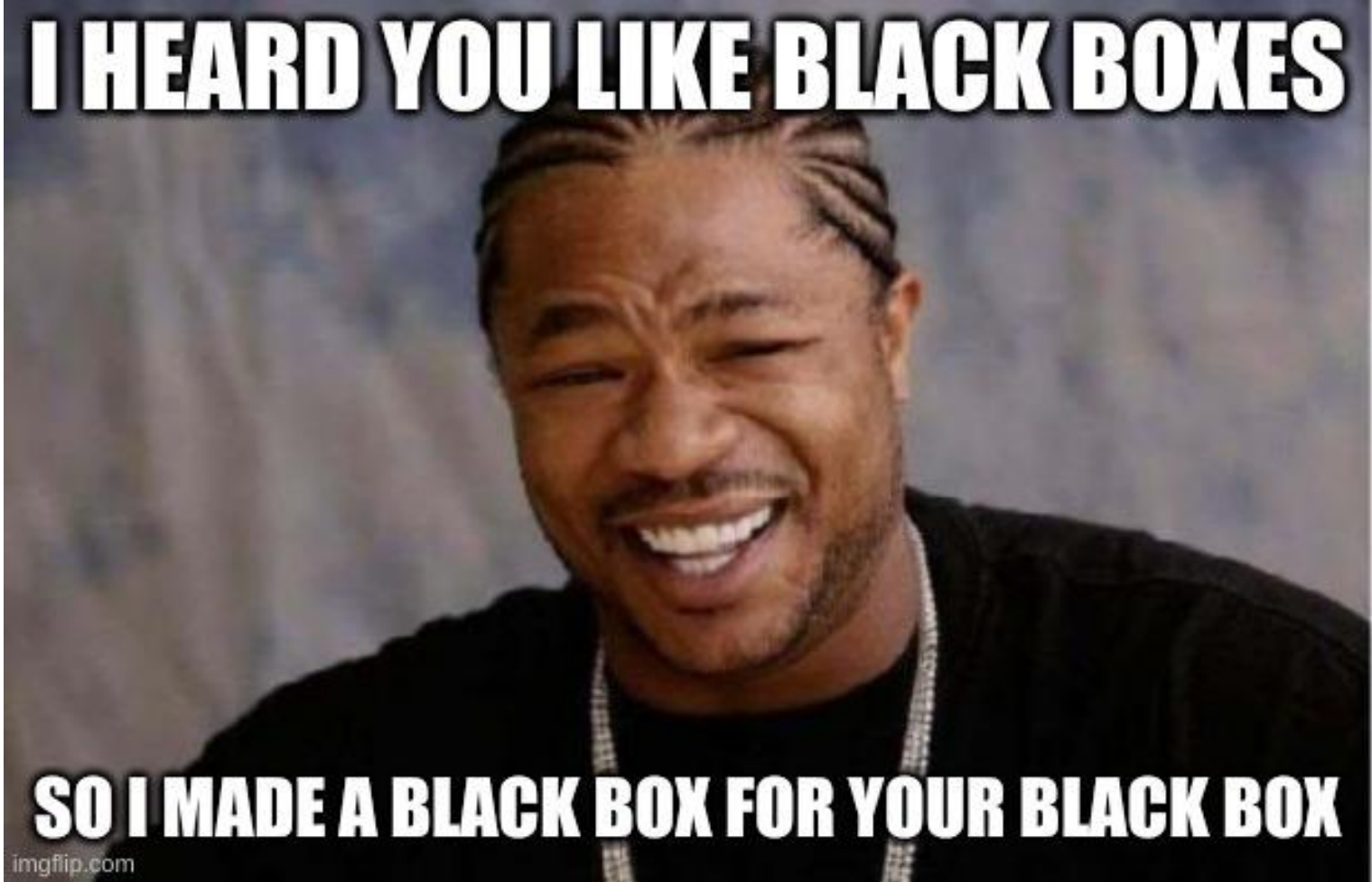
ϕ_0 is the expected prediction of the model with empty input, and serves as a base value.

ϕ_i are the SHAP values of the M features with index i .

The SHAP values **quantize how each feature contributes to moving the model prediction $f(x)$ away from the base value $E[f(z)]$.**

Put differently, the SHAP values **attribute to each feature the change in the expected model prediction when conditioning on that feature.**

I HEARD YOU LIKE BLACK BOXES



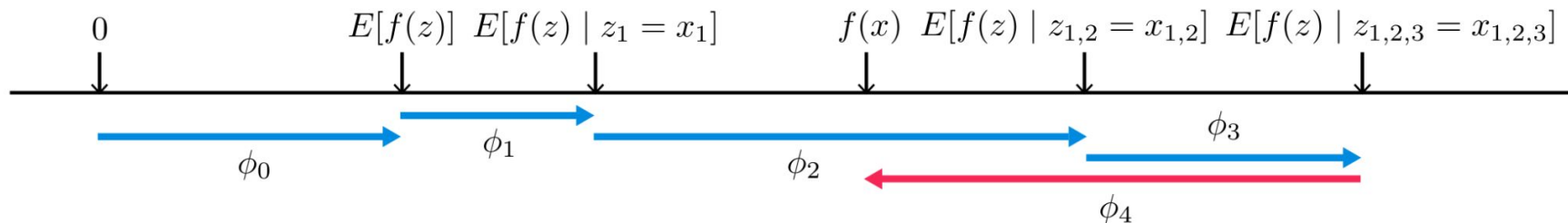
SO I MADE A BLACK BOX FOR YOUR BLACK BOX

imgflip.com

SHAP all in all!

$$f(x) = g(x') = \phi_0 + \sum_{i=1}^M \phi_i x'_i$$

The SHAP values ϕ_i quantize how each feature contributes to moving the model prediction $f(x)$ away from the base value ϕ_0 .



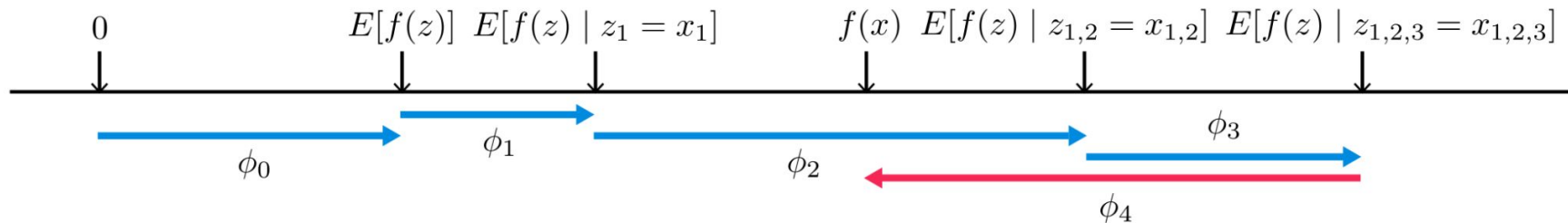
Visualisation showing how each feature $i=1,2,3,4$ drives the model prediction $f(x)$ away from the baseline prediction $\phi_0 = E[f(z)]$. In SHAP plots, **blue** is typically used to indicate **positive**, and **pink** to indicate **negative** contributions (from Lundberg & Lee (2017)).

One more thing before we look at code...

In the figure, features are added one by one, and the feature values in z conditioned upon their values, with the ordering $1 \rightarrow 1,2 \rightarrow 1,2,3 \rightarrow 1,2,3,4$.

This ordering does not matter as long as the surrogate model is linear and **the features are independent**.

We already made these assumptions, but please remember that the SHAP python package has these assumptions built in :)



Coding time!

We start with the same data set as yesterday and again a simple XGBoost regressor

In the diabetes dataset, each y value reflects how much a patient's condition progressed after a year

```
X_data, y_data = load_diabetes(scaled=False, as_frame=True, return_X_y=True)
features = X_data.columns.to_numpy()
features
```

```
array(['age', 'sex', 'bmi', 'bp', 's1', 's2', 's3', 's4', 's5', 's6'],
      dtype=object)
```

```
# I also love np arrays, but keep the dataframe structure if you want feature names on your SHAP plot
x_train, x_test, y_train, y_test = train_test_split(X_data, y_data, test_size=0.33, random_state=42)
```

```
regressor = xgboost.XGBRegressor()
regressor.fit(x_train, y_train);
```

SHAP for tree based models

XGBoost is a tree based model, and for these, we can use the TreeExplainer (more details to follow)

We first initialise the explainer, giving it the model, and we specify a background dataset.

What does SHAP do with the background dataset?

SHAP for tree based models

XGBoost is a tree based model, and for these, we can use the TreeExplainer (more details to follow)

We first initialise the explainer, giving it the model, and we specify a background dataset.

SHAP uses the background dataset to estimate the expectation by marginalising missing features independently (the assumption we have been discussing).

Next, we calculate the SHAP values and store them in a numpy array.

```
explainer = shap.TreeExplainer(regressor, data=x_train)
shap_values = explainer.shap_values(x_test)
```

SHAP values for a single datapoint

Let's pick out a single data point

```
: y_pred = regressor.predict(x_test)
  index = 6
  print("Mean BMI in test data:", np.mean(x_test.bmi))
  print("Mean prediction on test data:", np.mean(y_pred))
  print("Prediction on our data point:", y_pred[index])
  print("Data point:")
  print(x_test.iloc[index])

shap.force_plot(explainer.expected_value, shap_values[index,:], x_test.iloc[index,:])
```

Mean BMI in test data: 26.043150684931508

Mean prediction on test data: 149.2698

Prediction on our data point: 207.50145

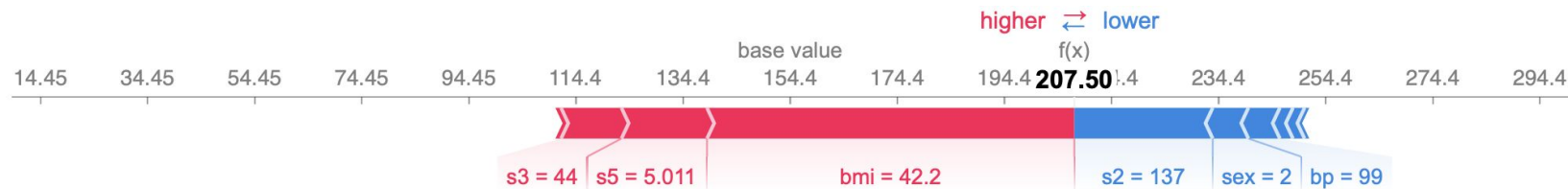
Data point:

age	46.0000
sex	2.0000
bmi	42.2000
bp	99.0000
s1	211.0000
s2	137.0000
s3	44.0000
s4	5.0000
s5	5.0106
s6	99.0000

Name: 367, dtype: float64

SHAP values for a single datapoint: force plot

Let's pick out a single data point, and visualize the SHAP values.



According to these, what is the single most important feature causing this patient to have a higher prediction than the mean test prediction?

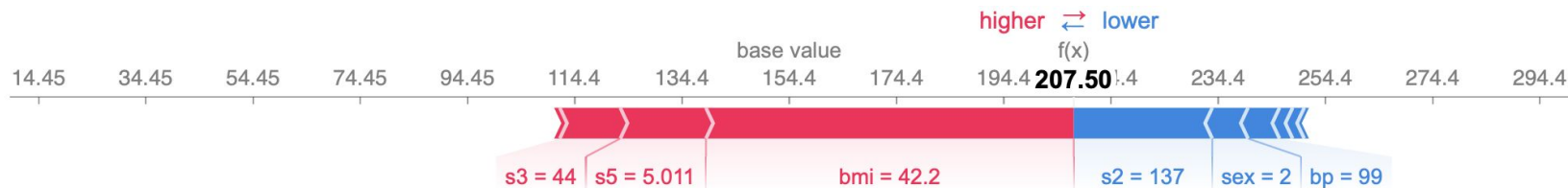
SHAP values for a single datapoint: force plot

Remember the linear model SHAP formula

$$f(x) = g(x') = \phi_0 + \sum_{i=1}^M \phi_i x'_i$$

What do the SHAP values explain?

What is the meaning of the SHAP values?

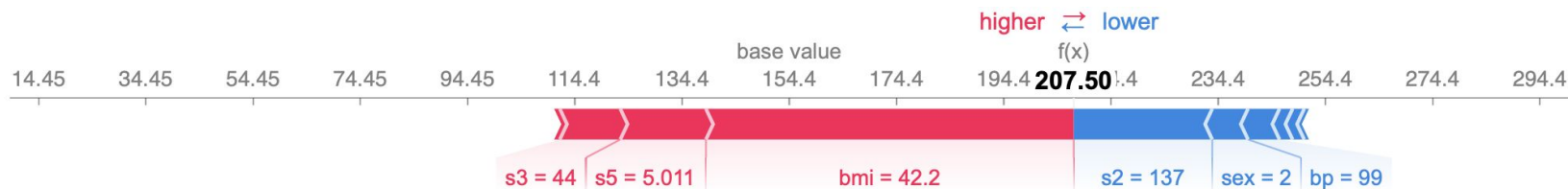


SHAP values for a single datapoint: force plot

Remember the linear model SHAP formula

$$f(x) = g(x') = \phi_0 + \sum_{i=1}^M \phi_i x'_i$$

Each feature's SHAP value quantizes how much the feature drives the model prediction away from the baseline prediction $\phi_0 = E[f(z)]$

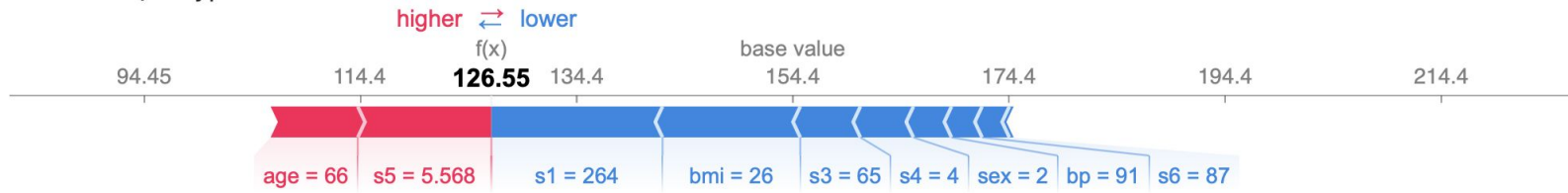


```
print("Base value + sum of the SHAP values=", explainer.expected_value + sum(shap_values[index,:]))
```

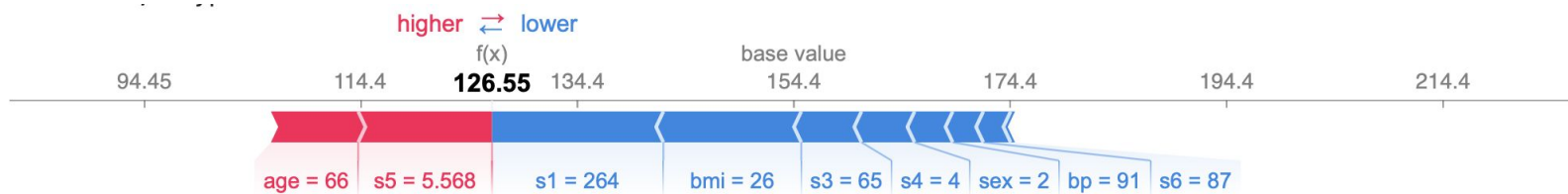
Base value + sum of the SHAP values= 207.50137

SHAP values for a single datapoint: force plot

Explain the prediction for this datapoint :)



SHAP values for a single datapoint: force plot



```
print(explainer.expected_value)
print(sum(shap_values[index, :]))
print(regressor.predict(x_test)[index])
print(explainer.expected_value + sum(shap_values[index, :]))
```

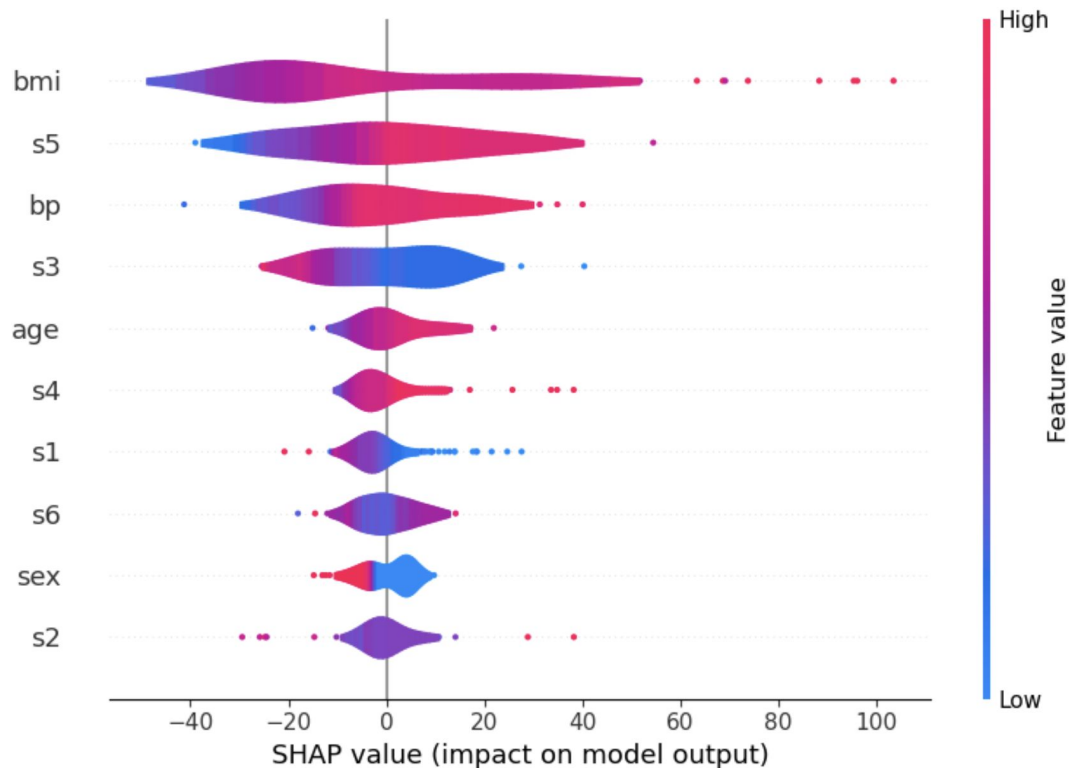
```
154.44574596805717
-27.891726560560564
126.554016
126.55401940749661
```

SHAP values for a dataset: violin plot

What happens if we plot all the calculated SHAP values?

SHAP values for a dataset: violin plot

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



There is quite a lot of information on this plot.

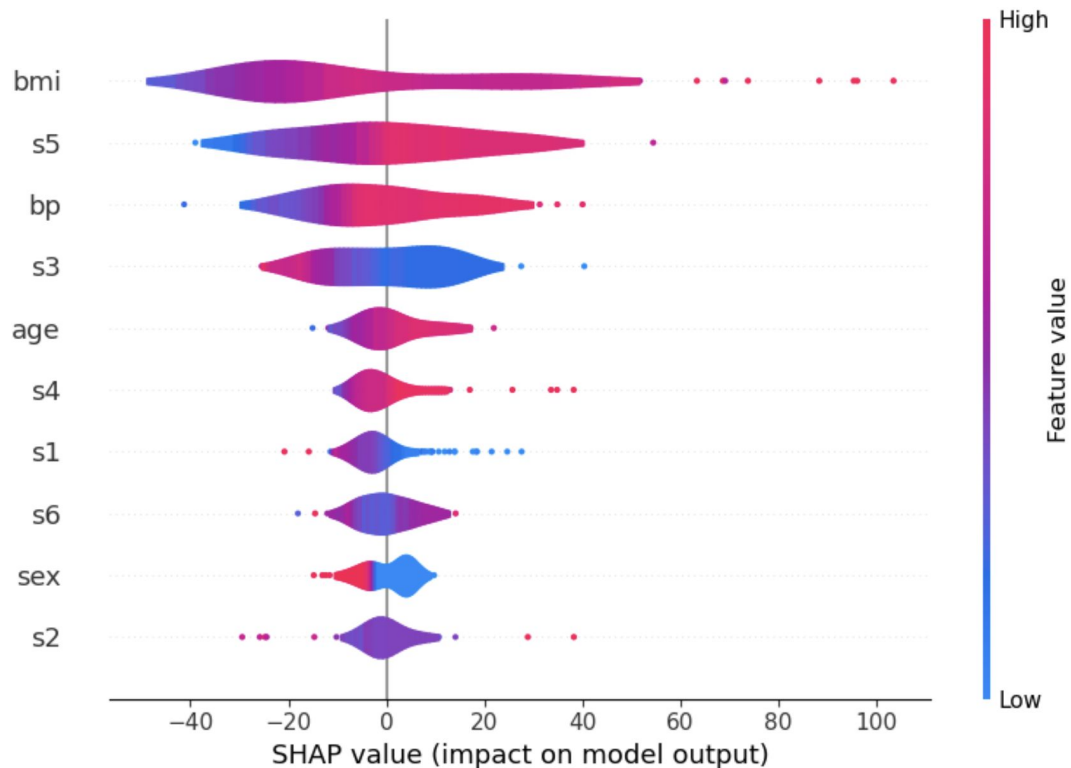
What is the y axis?

What is the x axis?

What is the color bar?

SHAP values for a dataset: violin plot

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



Yes, this plot is effectively three dimensional

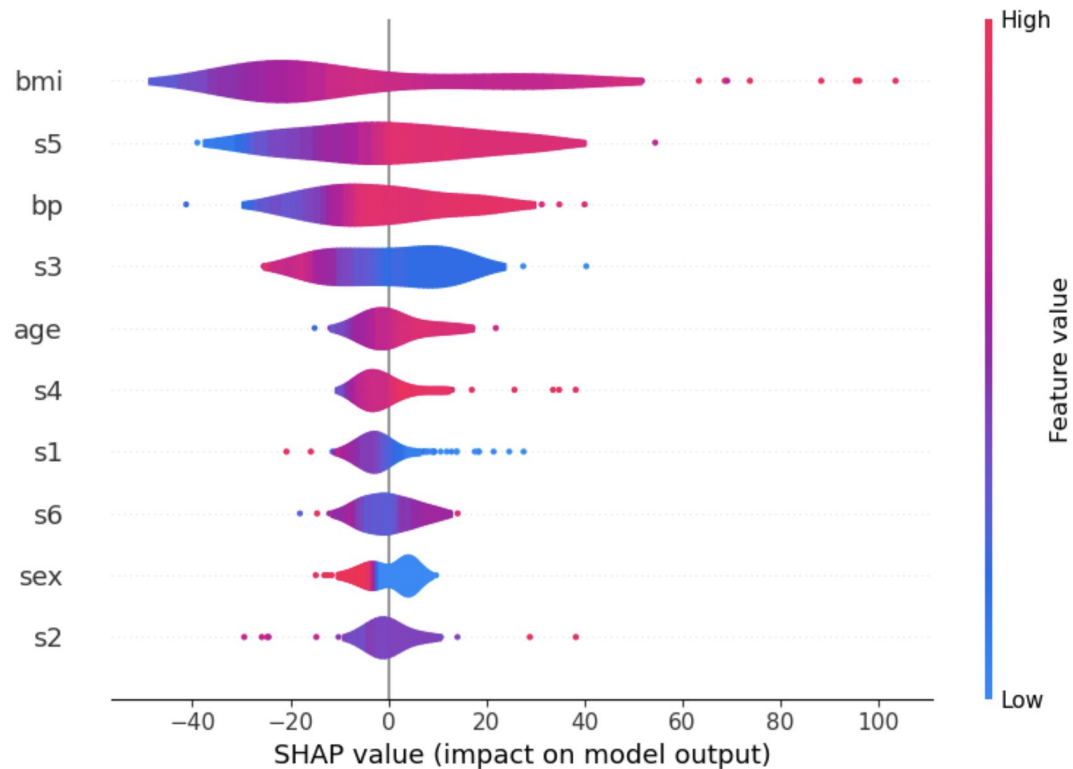
What do we see?

What would you say is the most important feature overall?

How important is age?

SHAP values for a dataset: violin plot

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



... is this some kind of *global* feature importance?

The background dataset...

What is the base value? What should it equal?

The background dataset...

The base value represents the expected model prediction without any feature values, so it should equal the mean prediction on the background data (in our case the training data)

```
base = explainer.expected_value
mean_pred = regressor.predict(x_train).mean()
mean_y = y_train.mean()
print("Base value:", base, "\nMean prediction on training data:", mean_pred)
```

The background dataset...

The base value represents the expected model prediction without any feature values, so it should equal the mean prediction on the background data (in our case the training data)

```
base = explainer.expected_value
mean_pred = regressor.predict(x_train).mean()
mean_y = y_train.mean()
print("Base value:", base, "\nMean prediction on training data:", mean_pred)
```

Base value: 154.44574596805717
Mean prediction on training data: 151.98286

One time I ran the code

```
base = explainer.expected_value
mean_pred = regressor.predict(x_train).mean()
mean_y = y_train.mean()
print("Base value:", base, "\nMean prediction on training data:", mean_pred)
```

Base value: 151.98288
Mean prediction on training data: 151.98286

Another time I ran the code

The background dataset...

It doesn't always! :-O

Remember that the base value is estimated assuming feature independence - which is not always the case. It might be completely wrong, or vary depending on the random split to test/train.

```
base = explainer.expected_value
mean_pred = regressor.predict(x_train).mean()
mean_y = y_train.mean()
print("Base value:", base, "\nMean prediction on training data:", mean_pred)
```

Base value: 154.44574596805717
Mean prediction on training data: 151.98286

```
base = explainer.expected_value
mean_pred = regressor.predict(x_train).mean()
mean_y = y_train.mean()
print("Base value:", base, "\nMean prediction on training data:", mean_pred)
```

Base value: 151.98288
Mean prediction on training data: 151.98286

SHAP for tree based models

Instead of using background data to estimate the base value, we can let the SHAP TreeExplainer use the tree to find the base value:

```
explainer = shap.TreeExplainer(regressor, feature_perturbation="tree_path_dependent")
shap_values = explainer.shap_values(x_test)
base = explainer.expected_value
mean_pred = regressor.predict(x_train).mean()
print("Base value:", base, "\nMean prediction on training data:", mean_pred)
```

Base value: 151.98288

Mean prediction on training data: 151.98286

In this case, SHAP uses the tree structure to simulate feature absence.

SHAP for tree based models

Instead of using background data to estimate the base value, we can let the SHAP TreeExplainer use the tree to find the base value:

```
explainer = shap.TreeExplainer(regressor, feature_perturbation="tree_path_dependent")
shap_values = explainer.shap_values(x_test)
base = explainer.expected_value
mean_pred = regressor.predict(x_train).mean()
print("Base value:", base, "\nMean prediction on training data:", mean_pred)
```

Base value: 151.98288

Mean prediction on training data: 151.98286

In this case, SHAP uses the tree structure to simulate feature absence.

When SHAP encounters a split in the tree which depends on a missing feature, it goes both ways down the tree, and recursively sums up the values, before averaging the final result.

To avoid traversing the same path several times, it also keeps track of the tree traversal.

SHAP for tree based models

Instead of using background data to estimate the base value, we can let the SHAP TreeExplainer use the tree to find the base value:

```
explainer = shap.TreeExplainer(regressor, feature_perturbation="tree_path_dependent")
shap_values = explainer.shap_values(x_test)
base = explainer.expected_value
mean_pred = regressor.predict(x_train).mean()
print("Base value:", base, "\nMean prediction on training data:", mean_pred)
```

Base value: 151.98288

Mean prediction on training data: 151.98286

In this case, SHAP uses the tree structure to simulate feature absence.

When SHAP encounters a split in the tree which depends on a missing feature, it goes both ways down the tree, and recursively sums up the values, before averaging the final result.

To avoid traversing the same path several times, it also keeps track of the tree traversal.

Because of this, the TreeExplainer runs in polynomial time instead of 2^N .

Image data

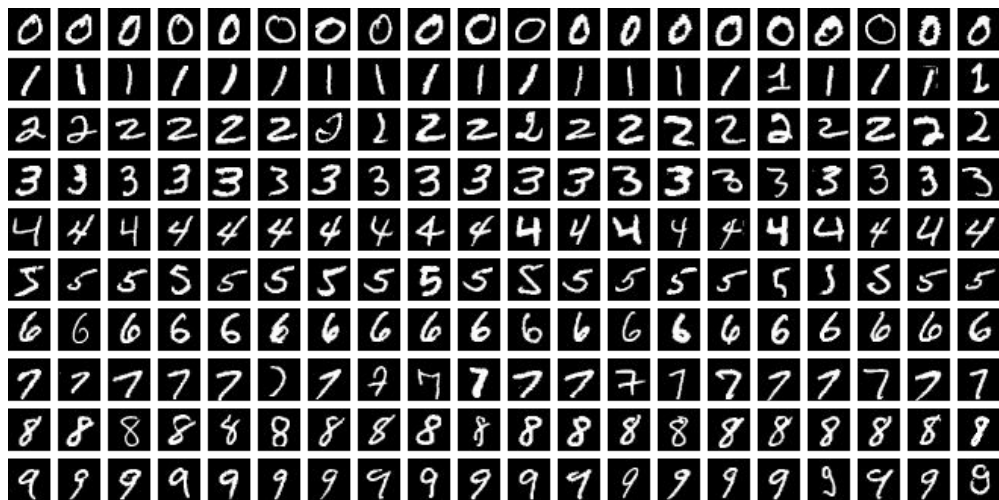
SHAP for neural networks

Let's use SHAP to explain a CNN trained to classify handwritten digits.

MNIST is a very popular dataset in the field of image processing, and often used for benchmarking.

It contains 70,000 handwritten digits from 0 to 9, with 28 x 28 pixels and 1 channel (black-white).

The dataset is already divided into training and testing sets.



MNIST loading and plotting

We will make our model in pytorch, so we use torch loaders

```
batch_size = 64

train_loader = torch.utils.data.DataLoader(
    datasets.MNIST("mnist_data",
        train=True,
        download=True,
        transform=transforms.Compose([transforms.ToTensor()])),
    batch_size=batch_size,
    shuffle=True,
)

test_loader = torch.utils.data.DataLoader(
    datasets.MNIST("mnist_data",
        train=False,
        transform=transforms.Compose([transforms.ToTensor()])),
    batch_size=batch_size,
    shuffle=True,
)
```

MNIST loading and plotting

```
imgs, lbls = next(iter(test_loader))  
print(imgs.shape) # torch.Size([B, C, H, W])  
print(lbls.shape) # torch.Size([B])
```

```
torch.Size([64, 1, 28, 28])  
torch.Size([64])
```

```
# This cell is just for looking at some of the images  
from torchvision.utils import make_grid  
imgs5 = imgs[:5]
```

```
# Make a grid and plot  
grid = make_grid(imgs5, nrow=5, padding=2)  
plt.figure(figsize=(8, 2))  
plt.imshow(grid.permute(1, 2, 0).squeeze(), cmap="gray")  
plt.axis("off")  
plt.title("Five MNIST samples")  
plt.show()
```

Five MNIST samples



Pytorch CNN model for MNIST classification

To classify the handwritten digits [0,9], we will use a small CNN.

The model weights and code for loading the model are provided (mnist_cnn.pt)

If you want, you can also modify and train the model yourself.

```
num_epochs = 5
device = torch.device("cpu")

class Net(nn.Module):
    def __init__(self):
        super().__init__()

        self.conv_layers = nn.Sequential(
            nn.Conv2d(1, 10, kernel_size=5),
            nn.MaxPool2d(2),
            nn.ReLU(),
            nn.Conv2d(10, 20, kernel_size=5),
            nn.Dropout(),
            nn.MaxPool2d(2),
            nn.ReLU(),
        )
        self.fc_layers = nn.Sequential(
            nn.Linear(320, 50),
            nn.ReLU(),
            nn.Dropout(),
            nn.Linear(50, 10),
            nn.Softmax(dim=1),
        )

    def forward(self, x):
        x = self.conv_layers(x)
        x = x.view(-1, 320)
        x = self.fc_layers(x)
        return x
```

Pytorch CNN model for MNIST classification

The provided model has a high accuracy despite being small, and you can test it if you like

Found model file mnist_cnn.pt. Loading...

Test set: Average loss: 0.0015, Accuracy: 9751/10000 (98%)

Use the model to predict on test data

```
index = 2
model.eval()
with torch.no_grad():
    images, labels = next(iter(test_loader))
    x = images[index-1:index].to(device)
    y = labels[index-1:index].to(device)

    output = model(x)
    pred = output.argmax(dim=1).item()

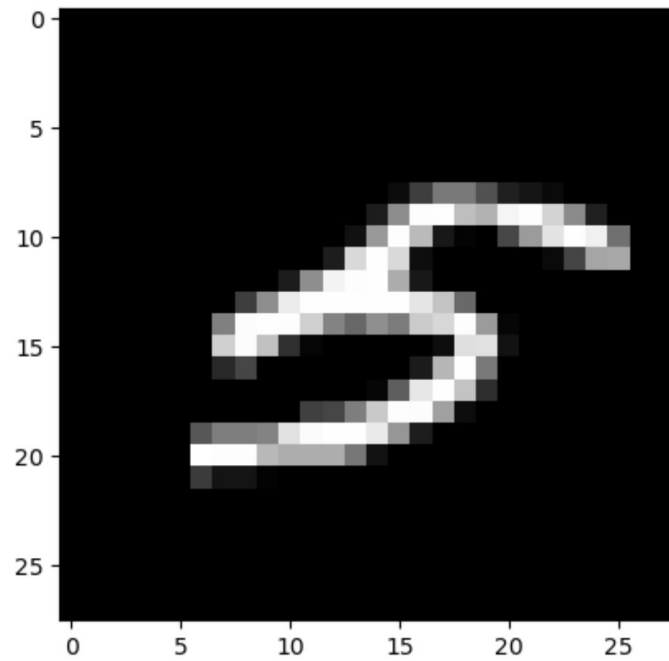
print("Label:", y.item())
print("Prediction:", pred)
print(f"Probabilities: {np.round(output.numpy()[0],2)}")

plt.imshow(x.squeeze(), cmap="gray")
plt.show()
```

Label: 5

Prediction: 5

Probabilities: [0.02 0.02 0.05 0.19 0.06 0.5 0.05 0.04 0.03 0.03]



SHAP on CNN model for MNIST classification

First, we need background data. We can take this from the train loader. The more samples we include, the slower the computation.

Build a small background set (reference distribution)

```
# Use ~50–200 images to keep it fast
background, _ = next(iter(train_loader))
background = background[:50].to(device)    # [50, 1, 28, 28]
background.shape
```

```
torch.Size([50, 1, 28, 28])
```

SHAP on CNN model for MNIST classification

Next, we call `shap.DeepExplainer` (which we will talk more about in a minute)

Create the explainer

```
explainer = shap.DeepExplainer(model, background)
```

Pick a few samples to explain

```
test_samples, test_labels = next(iter(test_loader))
N = test_samples.size(0)
idx = torch.randperm(N)[:5]
test_samples = test_samples[idx]          # [5, 1, 28, 28]
test_labels[idx]
test_samples.shape
```

```
torch.Size([5, 1, 28, 28])
```

and sample (or hand-pick, if you want) a few samples to explain

SHAP on CNN model for MNIST classification

Finally, we get SHAP values for the samples from the explainer, which returns a list of arrays per data point.

The array contains an image per class: [channels, height, width, classes]

```
shap_values = explainer.shap_values(test_samples)
shap_values[0].shape
```

```
(1, 28, 28, 10)
```

Always: `shap_values[data_index].shape == [1, 28, 28, 10]`

Plotting the SHAP values

We could print the SHAP values and inspect them. *What would be a better way to visualize them?*

(look at the shape)

```
shap_values = explainer.shap_values(test_samples)
shap_values[0].shape
```

(1, 28, 28, 10)

Plotting the SHAP values

We could print the SHAP values and inspect them. Since we have one SHAP value per pixel of the image, we can just plot the SHAP values on the input image.

```
shap_values = explainer.shap_values(test_samples)
shap_values[0].shape
```

```
(1, 28, 28, 10)
```

In general: when the structure of the explanation is the same as the structure of the input data, the two can be visualized together to make the explanation more intuitive.

Plotting the SHAP values

We could print the SHAP values and inspect them. Since we have one SHAP value per pixel of the image, we can just plot the SHAP values on the input image.

This is done using the `shap.image_plot` function. This function is incredibly annoying, since it needs very specific input shapes.

First input: `shap_values: [numpy.array:]`: List of numpy arrays of SHAP values. Each array has the shape `[samples, W, H, C]`, and length equal to the number of model outputs.

Second input: `pixel_values: numpy.array`: Matrix of pixel values `[samples, W, H, C]` for each image. It should be the same shape as each array in the `shap_values` list of arrays.

Plotting the SHAP values

```
shap_numpy = list(np.transpose(shap_values, (4, 0, 2, 3, 1)))
test_numpy = np.swapaxes(np.swapaxes(test_samples.numpy(), 1, -1), 1, 2)
print(len(shap_numpy))      # 10
print(shap_numpy[0].shape)  # (5, 28, 28, 1)
print(test_numpy.shape)     # (5, 28, 28, 1)
```

Reshape the inputs to image_plot

```
10
(5, 28, 28, 1)
(5, 28, 28, 1)
```

```
probs = model(test_samples).detach().numpy()      # [N,10]
pred_cls = probs.argmax(axis=1)
row_labels = [f"pred={int(pred_cls[i])} (p={probs[i, pred_cls[i]]:.2f})" for i in range(len(pred_cls))]

shap.image_plot(shap_numpy, -test_numpy, true_labels=row_labels)
```

Make a list of predicted labels to put on top of the plot (you'll see :)

Plotting the SHAP values

0 1 2 3 4 5 6 7 8 9

```
shap.image_plot(shap_numpy, -test_numpy, true_labels=row_labels)
```

pred=1 (p=0.99)

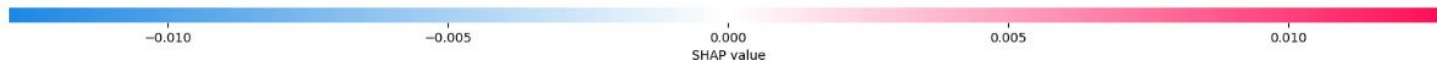
pred=9 (p=0.99)

pred=0 (p=1.00)

pred=4 (p=1.00)

pred=3 (p=1.00)

What does the SHAP values represent?



Plotting the SHAP values

0 1 2 3 4 5 6 7 8 9

```
shap.image_plot(shap_numpy, -test_numpy, true_labels=row_labels)
```

pred=1 (p=0.99)

pred=9 (p=0.99)

pred=0 (p=1.00)

pred=4 (p=1.00)

pred=3 (p=1.00)



The SHAP values represent how much each feature (pixel) drives the prediction away from the base value.

What do we see?

Plotting the SHAP values for a new set of predictions

0 1 2 3 4 5 6 7 8 9

```
shap.image_plot(shap_numpy, -test_numpy, true_labels=row_labels)
```

pred=8 (p=0.99)



pred=6 (p=1.00)



pred=6 (p=0.92)



pred=6 (p=0.94)



pred=7 (p=0.97)



Let's hear it :)

-0.010

-0.005

0.000
SHAP value

0.005

0.010

TreeExplainer, DeepExplainer,..., *Explainer

There are many different SHAP explainers, and we won't study them all.

Remember, for tree based models, TreeExplainer is the best.

For neural networks, we have several options, including [DeepExplainer](#).

Put simply, DeepExplainer finds feature attributions by backpropagating the difference between node activations for the data point x and the background data points.

The full story is a bit longer. In this course, you only have to understand the SHAP basics, *not how the different explainer objects are created*.

DeepExplainer - if you're interested

DeepExplainer estimates SHAP values for neural networks based on node activations. It is based on the idea of DeepLIFT (Deep Learning Important FeaTures) [1]. DeepLIFT calculates feature attributions by:

- comparing the activation of each neuron x to its 'reference activation' x^*
- backpropagating these differences (not gradients) to the input
- assigning contribution scores according to the resulting difference in the input space.

Contributions are calculated so that they sum to $f(x) - f(x^*)$. The implementation assumes independent input features and linear activations.

Per node attribution in DeepLIFT can be used to approximate Shapley values. DeepExplainer computes feature attributions that sum up to $f(x) - E[f(x)]$ by integrating over many background samples (instead of using one reference value).

Intuitively: a DeepLIFT baseline $x^* \approx$ **one** SHAP background point; SHAP's baseline is the **expectation over many** such points, which aligns with the Shapley value definition.

DeepExplainer approximates Shapley values for deep neural networks by backpropagating contribution multipliers through (piecewise-linear) layers (like ReLU).

SHAP - Shapley Additive exPlanations

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		

SHAP - Shapley Additive exPlanations

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		

SHAP - Shapley Additive exPlanations

Where do we place this in the taxonomy?

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

Model-agnostic: *We systematically exclude input features and use only the model's prediction.*

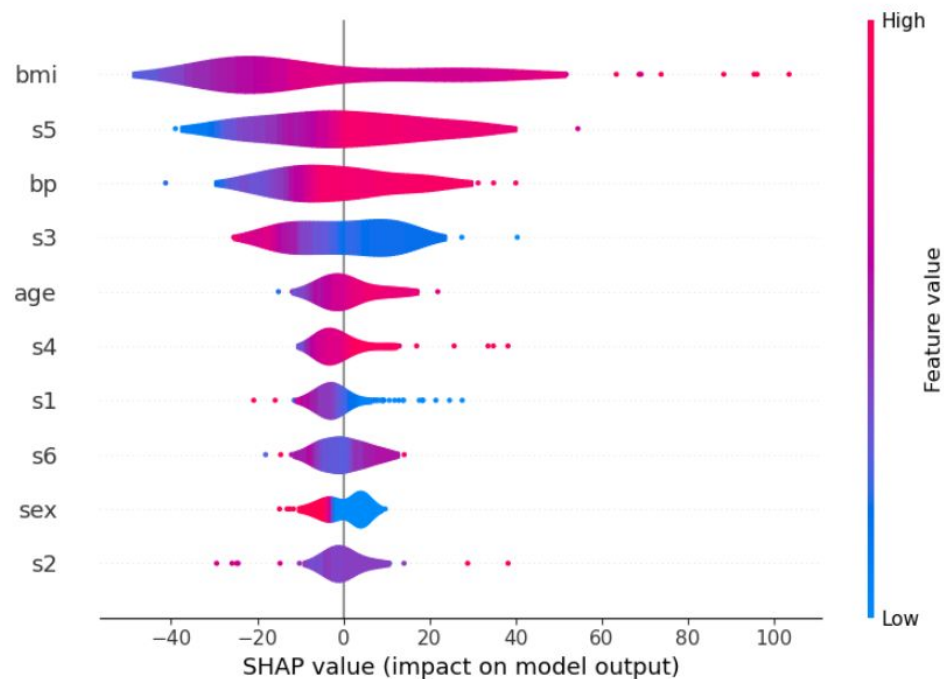
Local or global?

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

The SHAP plot from before

This is probably the most widely used SHAP plot out there

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



What is plotted here?

What are the dots?

SHAP - Shapley Additive exPlanations

Where do we place this in the taxonomy?

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

Model-agnostic: *We systematically exclude input features and use only the model's prediction.*

Local: *We explain single predictions - although these are often plotted across a whole dataset!*

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

SHAP - Shapley Additive exPlanations

Where do we place this in the taxonomy?

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

Model-agnostic: *We systematically exclude input features and use only the model's prediction*

Local: *We explain single predictions - although these are often plotted across a whole dataset!*

Post-hoc methods	Model-agnostic	Model-specific
Local	<i>SHAP</i>	
Global	<i>(sometimes used here)</i>	

Homework

Understand the interpretable space, and study the notation until you know it! What is z' ? What is \bar{S} ? etc.

Work your way through the notebook and get comfortable with the code.

Make SHAP explainer plots and understand what the plots tell us.

You should be able to go from a SHAP plot to a full sentence about what the baseline represents and how the attributed feature importances contribute to the model prediction.

You should know the two main underlying assumptions in the Python version of SHAP.

Grab the interpretable feature space notation and SHAP library by the whatever and learn to love it.
See you tomorrow!

SHAP violin feature importance plot

What happens if we plot all the calculated SHAP values?

First, do you remember what was the most important feature according to yesterday's Shapley values?

How important was age?

SHAP for tree based models

What was the most important feature according to yesterday's Shapley values? How important was age?

