Machine Learning Explainability: 4th lesson – SHAP Values

You've seen (and used) techniques to extract general insights from a machine learning model. But what if you want to break down how the model works for an individual prediction?

SHAP is an acronym of SHapley Additive exPlantations, which described as break down methodology to show the impact of each feature. Where could you use this?

* A model says a bank should not loan someone money, and the bank is legally required to explain the basis for each loan rejection.
* A healthcare provider wants to identify what factors are driving each patient’s risk of some disease, so they can directly address those risk factors with targeted health interventions.

You'll use SHAP Values to explain individual predictions in this lesson. In the next lesson, you'll see how these can be aggregated into powerful model-level insights.

***How they work?***

SHAP values interpret the impact of having a certain value for a given feature in comparison to the prediction we'd make if that feature took some baseline value. An example is helpful, and we'll continue the soccer/football example from the **permutation importance** and **partial dependence plots** lessons.

In these tutorials, we predicted whether a team would have a player win the *Man of the Match* award:

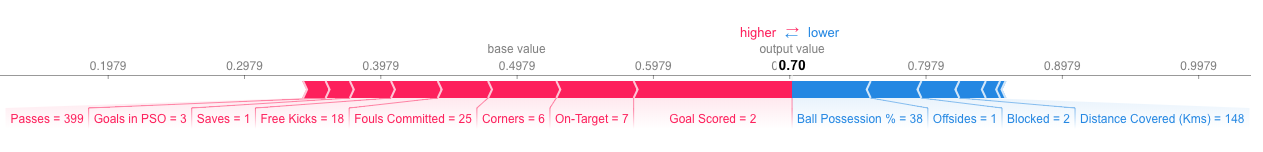
* We could ask how much was a prediction driven by the fact that the team scored 3 goals?
* But it is easier to give a concrete, numeric answer if we restate this as:

“*How much was a prediction driven by the fact that the team scored 3 goals, instead of some baseline number of goals?*”

Of course, each team has many features. So, if we answer this question for *number of goals*, we could repeat the process for all other features. SHAP values do this in a way that guarantees a nice property. Specifically, you decompose a prediction with the following equation:

sum(SHAP values for all features) = pred\_for\_team - pred\_for\_baseline\_values

That is, the SHAP values of all features sum up to explain why my prediction was different from the baseline. This allows us to decompose a prediction in a graph like this:



***How do you interpret this?***

We predicted 0.7, whereas the base\_value is 0.4979. Feature values causing increased predictions are in pink, and their visual size shows the magnitude of the feature’s effect. Feature values decreasing the prediction are shown in blue. The biggest impact comes from Goal Scored being 2. Though the ball possession value has a meaningful effect decreasing the prediction.

If you subtract the length of the blue bars from the length of the pink bars, it equals the distance from the base value to the output. There is some complexity to the technique, to ensure that the baseline plus the sum of individual effects adds up to the prediction (which isn't as straightforward as it sounds). We won't go into that detail here, since it isn't critical for using the technique.

Code to calculate SHAP values:

We calculate SHAP values using the wonderful Shap library. For this example, we'll reuse the model you've already seen with the Soccer data.

import numpy as np

import pandas as pd

from sklearn.model\_selection import train\_test\_split

from sklearn.ensemble import RandomForestClassifier

data = pd.read\_csv('../input/fifa-2018-match-statistics/FIFA 2018 Statistics.csv')

y = (data['Man of the Match'] == "Yes") *# Convert from string "Yes"/"No" to binary*

feature\_names = [i for i **in** data.columns if data[i].dtype **in** [np.int64, np.int64]]

X = data[feature\_names]

train\_X, val\_X, train\_y, val\_y = train\_test\_split(X, y, random\_state=1)

my\_model = RandomForestClassifier(random\_state=0).fit(train\_X, train\_y)

We will look at SHAP values for a single row of the dataset (we arbitrarily chose row 5). For context, we'll look at the raw predictions before looking at the SHAP values.

row\_to\_show = 5

data\_for\_prediction = val\_X.iloc[row\_to\_show] *# use 1 row of data here. Could use multiple rows if desired*

data\_for\_prediction\_array = data\_for\_prediction.values.reshape(1, -1)

my\_model.predict\_proba(data\_for\_prediction\_array)

/opt/conda/lib/python3.7/site-packages/sklearn/base.py:451: UserWarning: X does not have valid feature names, but RandomForestClassifier was fitted with feature names

"X does not have valid feature names, but"

array([[0.29, 0.71]])

The team is 70% likely to have a player win the award. Now, we’ll move onto the code to get SHAP values for that single prediction.

import shap *# package used to calculate Shap values*

*# Create object that can calculate shap values*

explainer = shap.TreeExplainer(my\_model)

*# Calculate Shap values*

shap\_values = explainer.shap\_values(data\_for\_prediction)

The shap\_values object above is a list with two arrays. The first array is the SHAP values for a negative outcome (unable to win the award), and the second array is the list of SHAP values for the positive outcome (wins the award). We typically think about predictions in terms of the prediction of a positive outcome, so we'll pull out SHAP values for positive outcomes (pulling out shap\_values[1]).

It is cumbersome to review raw arrays, but the SHAP package has a nice way to visualize the results.

shap.initjs()

shap.force\_plot(explainer.expected\_value[1], shap\_values[1], data\_for\_prediction)



If you look carefully at the code where we created the SHAP values, you'll notice we reference Trees in shap.TreeExplainer(my\_model). But the SHAP package has explainers for every type of model:

* shap.DeepExplainer works with deep learning models.
* shap.KernelExplainer works with all models, though it is slower than other Explainers and it offers an approximation rather than exact SHAP values.

Here is an example using KernelExplainer to get similar results. The results aren't identical because KernelExplainer gives an approximate result. But the results tell the same story.

*# use Kernel SHAP to explain test set predictions*

k\_explainer = shap.KernelExplainer(my\_model.predict\_proba, train\_X)

k\_shap\_values = k\_explainer.shap\_values(data\_for\_prediction)

shap.force\_plot(k\_explainer.expected\_value[1], k\_shap\_values[1], data\_for\_prediction)

X does not have valid feature names, but RandomForestClassifier was fitted with feature names

X does not have valid feature names, but RandomForestClassifier was fitted with feature names

X does not have valid feature names, but RandomForestClassifier was fitted with feature names

The default of 'normalize' will be set to False in version 1.2 and deprecated in version 1.4.

If you wish to scale the data, use Pipeline with a StandardScaler in a preprocessing stage. To reproduce the previous behavior:

from sklearn.pipeline import make\_pipeline

model = make\_pipeline(StandardScaler(with\_mean=False), LassoLarsIC())

If you wish to pass a sample\_weight parameter, you need to pass it as a fit parameter to each step of the pipeline as follows:

kwargs = {s[0] + '\_\_sample\_weight': sample\_weight for s in model.steps}

model.fit(X, y, \*\*kwargs)

Set parameter alpha to: original\_alpha \* np.sqrt(n\_samples).

The default of 'normalize' will be set to False in version 1.2 and deprecated in version 1.4.

If you wish to scale the data, use Pipeline with a StandardScaler in a preprocessing stage. To reproduce the previous behavior:

from sklearn.pipeline import make\_pipeline

model = make\_pipeline(StandardScaler(with\_mean=False), LassoLarsIC())

If you wish to pass a sample\_weight parameter, you need to pass it as a fit parameter to each step of the pipeline as follows:

kwargs = {s[0] + '\_\_sample\_weight': sample\_weight for s in model.steps}

model.fit(X, y, \*\*kwargs)

Set parameter alpha to: original\_alpha \* np.sqrt(n\_samples).

