**All You Need to Know About SHAP for Explainable AI?**

[[Isha Choudhary](https://medium.com/@shahooda637?source=post_page-----8ad35a05e6ec--------------------------------)](https://medium.com/@shahooda637?source=post_page-----8ad35a05e6ec--------------------------------)

[Isha Choudhary](https://medium.com/@shahooda637?source=post_page-----8ad35a05e6ec--------------------------------)

·

[Follow](https://medium.com/m/signin?actionUrl=https%3A%2F%2Fmedium.com%2F_%2Fsubscribe%2Fuser%2Fa0449bd47c69&operation=register&redirect=https%3A%2F%2Fmedium.com%2F%40shahooda637%2Fall-you-need-to-know-about-shap-for-explainable-ai-8ad35a05e6ec&user=Isha+Choudhary&userId=a0449bd47c69&source=post_page-a0449bd47c69----8ad35a05e6ec---------------------post_header-----------)

18 min read

·

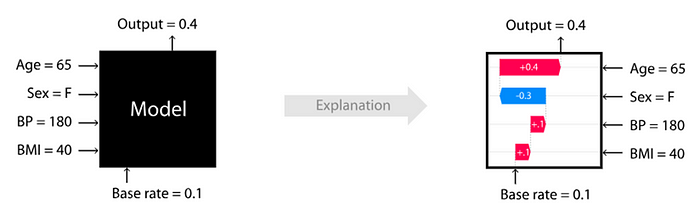
Aug 30

155

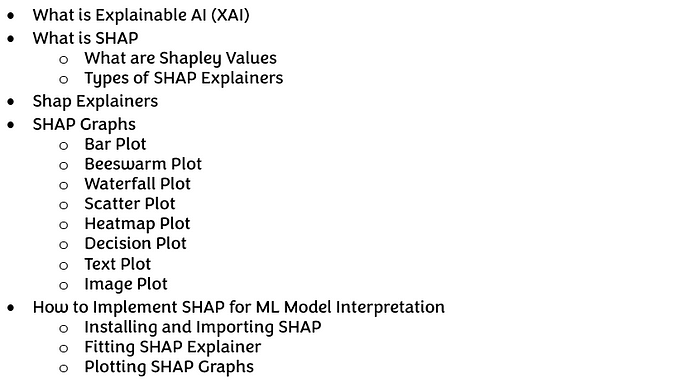
2

**A step-by-step guide to understand and learn SHAP (Shapley Additive exPlaination) and how to interpret ML models using the SHAP library**





**OUTLINE**



**WHAT IS EXPLAINABLE AI (XAI)**

Artificial Intelligence is a buzz for around a decade now. Every industry is working on incorporating AI into their workforce in one way or the other. Several AI based applications, models and solutions are being presented by researchers and scientists for automating different tasks in different industries. However, the way these models or applications process to yield the results is still a mystery to both the developer and the user. This lack of interpretability of AI models and applications make them hard to trust.

Explainable AI (XAI) refers to the ability of an AI model to deliver human understandable explanations for the decisions or predictions made by the machine. The objective of XAI is to bridge the gap between the human users and AI machines by making the decision-making process of the AI machine interpretable and transparent.

In traditional AI approaches like machine learning or deep learning, the models function as black box, which means completely un-understandable to humans about how a particular decision was made by the AI model and how did he reach to the suggested conclusions. XAI methods strives to provide an explanation of the internal processing or working of the AI model, enabling the users to gain a better and deeper understanding of the factors influencing the model’s output. XAI techniques vary depending on the type of AI model. They may include methods like feature importance analysis, rule extraction, surrogate models, visualizations, or natural language analysis.

This is a tutorial article which discusses the everything about an XAI library SHAP your Machine learning models. The following sections of this article would educate the reader about the very strong and popular XAI library ‘SHAP’ and how this library can be used for different types of machine learning algorithms. This article aims to deliver a complete end-to-end information about XAI library SHAP for anyone who is willing to learn and use it in their projects.

**WHAT IS SHAP**

SHAP or Shapley Additive Explanations, is a method used by people working on with machine learning models or developing AI for interpretation of predictions of ML models. It can be used for explaining the predictions of any model. This is what makes SHAP a really useful and popular XAI library for interpreting your ML models. SHAP is a contribution of various other XAI tools such as LIME, SHAPely, Sampling Values, DeepLift and more. The key idea of SHAP is to calculate the Shapley values for each feature of the dataset used to train and test the machine learning model which is to be interpreted. Each Shapley value represents the impact that the feature in generating the prediction delivered by the model.

The intuition behind SHAP is simple, there is an associated Shapley value for each feature in your dataset. How is this done? For understanding this, let’s first understand the heart of SHAP, the Shapley Values.

**WHAT ARE SHAPLEY VALUES**

Shapley values are a concept of the cooperative game theory field. The objective of Shapley values is to measure each player’s contribution to the game. The concept behind the calculation of Shapley values is fundamentally based on the game theory where ’n’ players are participating in the game with an aim to achieve the reward ‘a’, and this reward is intended to be fairly distributed at each one of the ’n’ players according to the individual contribution, such as Shapley Value.

In short, Shapley value is the measure of the average marginal contribution of a feature for an instance among all possible bunch in the sample. Let’s understand this in detail.

Imagine a group of people (A, B, C, D, E) is working on a project to gain profit (P) for their company. Now, to distribute the company profit equally among the 5 based on their effort to gain that profit, we need to calculate their individual contribution to gain the profit (P). This contribution is the Shapley Value for each person in the group, or in case of data, each feature. To calculate the Shapley value of a person ‘A’ in the group, the difference between the profit generated in presence of that member and in the absence of that member is calculated.

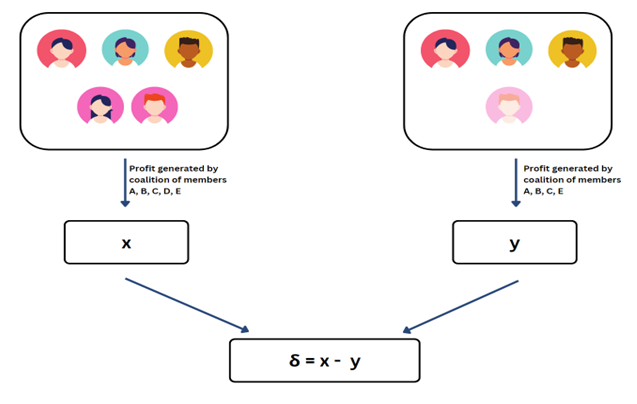


Figure 01: Calculating the Marginal Distribution for a feature (created by the Author)

This difference is what we call ‘*marginal contribution*’ of the member ‘A’ to the current coalition/group. All the different groups in which the member ‘A’ is present is the coalition for us here. Thus, we will calculate all the marginal contributions for this member (‘A’), based on all the possible coalitions for him/her. The mean of all the marginal distributions is the ‘*Shapley Value*’ for this member ‘A’.

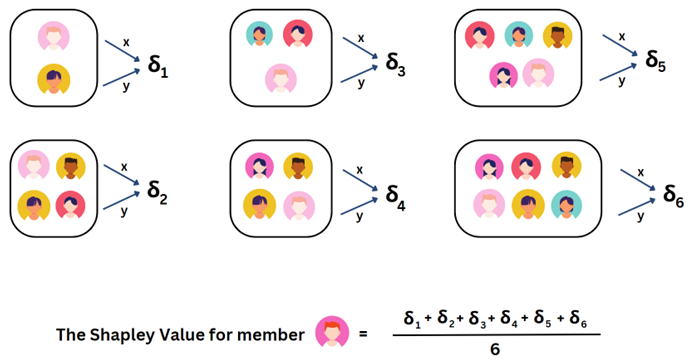


Figure 02: Calculating the Shapley Value for a feature (created by the Author)

So, this is how Shapley values are calculated. However, the machine learning models usually have a large number of features which are either discrete or continuous variables which cause the computation of the Shapley values to be very complicated for each feature for each instance. Therefore, to avoid this issue, there are many other shap explainers to calculate Shapley values for many different types of machine learning or deep learning models. The names and application of these are listed and discusses further in this article. One of the best things of SHAP is that it supports modelling procedures from all the libraries like SciKit-Learn, PySpark, TensorFlow, Keras or more.

**TYPES OF SHAP EXPLAINERS**

There are several types of SHAP explainer present in the SHAP library recently installed. Each of the following SHAP explainer types is designed for different types of models and data.

1. shap.Explainer

2. shap.explainers.Tree

3. shap.explainers.Linear

4. shap.explainers.Permutation

5. shap.explainers.Deep

It is really important to choose the appropriate explainer based on the characteristics of our AI model and data. We will discuss these explainers in detail in the following sections.

**SHAP EXPLAINERS**

The Explainable AI techniques can only be used to understand the processing of the machine learning models, it is very crucial to have a proper dataset. The pre-requisite for using SHAP is to have a trained machine learning model and a proper dataset. The specific criteria depends on the type of SHAP explainer used and the nature of the dataset.

**shap.Explainer**

This is the primary explainer interface for the SHAP library. It takes any combination of a model and masker and returns a callable subclass object that implements the particular estimation algorithm that was chosen. The code for creating this explainer as given by SHAP is as follows:

shap.Explainer(mode, masker=None, link=CPUDispatcher(<function identity>), algorithm='auto', output\_names=None, feature\_names=None, linearize\_link=True, seed=None, \*\*kwargs)

This explainer use Shapley values to explain any machine learning model or python function. The parameters for this explainer are listed below. These are important to understand the type of data which is to be used as the input data for this explainer in order to get the shap plots. The parameters listed in the code are explained as:

**model : object or function**

User supplied function or model object that takes a dataset and computes the model for those samples. In simple words, this is the machine learning model, or algorithm we are trying to get explanation on.

**link : function**

Used to map between the output units of the model and the SHAP value units. By default, this parameter is kept auto as it uses ‘*shap.links.identity’*if not specified explicitly. Otherwise, ‘*shap.links.logit*’ can be useful so that expectations are computed in probability units while explanations remain in log-odds units.

**algorithm : “auto”, “permutation”, “partition”, “tree”, or “linear”**

This is the algorithm used to compute the Shapley values (remember the ‘What are Shapley Values’ of this article). By default, the ‘auto’ options attempts to make the best choice given the passed model, and the masker are also mentioned in the code, but this choice can always be overridden by passing the specific algorithm for computation.

**output\_names : None or list of strings**

The names of the model outputs. In case of an image classifier, it would be the name of all the classes, and similarly for other models. However, this parameter is optional, and if it is kept “*None*”, the Explanation object produced will not have any output\_names, which could affect the downstream plots.

**seed : None or int**

This is for reproducibility. By setting a seed, we can ensure that the random number generation used within SHAP remain consistent across multiple runs. It is particularly useful when we are comparing or reproducing the results in a consistent manner. Seed is also for debugging; it can help if we frequently trace back the issues.

**shap.explainers.Tree**

Tree SHAP is a fast and exact method to estimate SHAP values for tree models and ensembles of trees such as Decision Tree, XGBoost, or Random Forest, under several different possible assumptions about feature dependence. The code for creating this explainer as given by SHAP is as follows:

shap.explainers.Tree(model, data=None, model\_output='raw', feature\_perturbation='interventional', feature\_names=None, approximate=False, \*\*deprecated\_options)

This explainer uses Tree SHAP algorithms to explain the output of ensemble tree algorithms such as Random Forest, XGBoost and more.

The parameters of this explainer are explained below. Understanding of these parameters is important to use the code to fit this explainer on your model and data.

**model: model object**

This explainer takes a tree-based machine learning model only. These can be XGBoost, LightGBM, CatBoost, etc. Most of the tree-based SciKit-learn models are supported.

**data: numpy.array or pandas.DataFrame**

This is the background dataset used for integrating out features. This is an optional argument when feature\_perturbation = “tree\_path\_dependent”, as in that we can use the number of training samples that went down each tree path as our background dataset. If that is not the case, just input the dataset you used for training the model. One thing to note is that your dataset must be in proper format and compatible with the explainer.

**model\_output: “raw”, “probability”, “log\_loss”, or model method name**

This parameter notes, “what output of the model should be explained”. For regression models, this is “raw”, the standard output. For binary classification models like XGBoost, this is the log odds ratio. If the model output is the name of the supported prediction method on the model object, we use “model method name”.

For example, model\_output = “predict\_proba”, this explains the results of calling “*model.predict\_proba*”. Note that, the SHAP values will change on the basis of the type of model output specified. This argument is used or helpful for breaking down model performance by feature.

**feature\_perturbation: “interventional” (default) or “tree\_path\_dependent” (default when data = None)**

The SHAP values rely on conditional expectations, therefore, we need to decide how to handle correlated (or dependent) input features. The “interventional” approach breaks the dependencies between the features according to the rules dictated by casual inference.

The ”tree\_path\_dependent” approach is to just follow the trees and use the number of training examples that went down each leaf to represent the background distribution. This approach does not require a background dataset.

**shap.explainers.Linear**

This explainer computes the SHAP values for a linear model and can account for the correlations among the input features. Assuming that the features are independent, the computation of interventional SHAP values for the linear model for the ith features is

coef[i] \* (x[i] — X.mean(0)[i])

The code for this explainer as given by SHAP is as follows:

shap.explainers.Linear(model, masker, link=CPUDispatcher(<function identity>), nsamples=1000, feature\_perturbation=None, \*\*kwargs)

This explainer computer SHAP values for a linear model, optionally accounting for inter-feature correlations. The arguments used in the code above are explained in following points.

**model: (coef, intercept) or sklearn.linear\_model**

The model hers is a user supplied model either as a parameter pair or sklearn object.

**data: (mean, cov), numpy.array, pandas.DataFrame, iml.DenseData or scipy.csr\_matrix**

This is the background dataset used for computing conditional expectations. Here, passing the raw data matrix is just a convenient alternative to the mean and covariance.

**nsamples: int**

This is the number of samples to use when estimating the transformation matrix used to account for feature correlations.

**feature\_perturbation: “interventional” (default) or “correlation\_dependent”**

For this explainer, there are two ways to compute the SHAP values, either the full conditional SHAP values or the interventional SHAP values. For interventional, we break any dependence structure between features, while in full conditional, correlations among features is respected, and both the features (correlated features) get some credit for the model’s behaviour. In other words, intervention is “true to the model”, while full conditional is “true to the data”.

**shap.explainers.Permutation**

This Shap explainer is a model agnostic explainer that guarantees local accuracy (additivity) by iterating completely through an entire permutation of the features in both forward and reverse directions. This gives us the exact SHAP values for models with up to second order interaction effects. To get better SHAP values with higher order interactions, we can iterate this many times by random permutations. With this explainer, we can also account for hierarchical data structures with partition trees. The code for this explainer as given by SHAP is as follows:

shap.explainers.Permutation(model, masker, link=CPUDispatcher(<function identity>), feature\_names=None, linearize\_link=True, seed=None, \*\*call\_args)

This shap explainer method approximated the Shapley values by iterating through permutations of the inputs. The explanation for the arguments used in the code above are discussed below.

**model: function**

This should be a callable python object that executed the model given a set of input data samples.

**masker: function or numpy.array or pandas.DataFrame**

This should be a callable python object which is used to “mask” out hidden features of the form *masker(binary\_mask, x)*. It takes a single input sample and a binary mask and return a matrix of masked samples. As a shortcut for the standard masking using by SHAP you can pass a background data matrix instead of a function and that matrix will be used for masking. To use a clustering game structure, we can pass a shap.maskers.Tabular(data, clustering = “correlation”) object.

**seed: None or int**

seed for reproducibility

**\*\*call\_args: valid argument to the \_\_call\_\_ method**

These arguments are saved and passes to the \_\_call\_\_ method as the new default values for these arguments.

**shap.explainers.Deep**

This is the enhanced version of the DeepLIFT algorithm (Deep SHAP), mean to approximate SHAP values for deep learning models. Here, we approximate the conditional expectations of SHAP values using a selection of background samples. By integrating over many background samples Deep estimates approximate SHAP values such that they sum up to the difference between the expected model output on the passed background samples and the current model output

*(f(x) — E[f(x)])*

The code for this explainer as given by SHAP is as follows:

shap.explainers.Deep(model, data, session=None, learning\_phase\_flags=None)

The explanation for the parameters used in the code above are given in points below.

**model: if framework == ‘tensorflow’, (input: [tf.Tensor], output: tf.Tensor)**

A pair of TensorFlow tensors (or a list and a tensor) that specifies the input and output of the model to be explained. Note that SHAP values are specific to a single output value, so the output tf.Tensor should be a single dimensional output.

**: if framework == ‘pytorch’, an nn.Module object (model), or a tuple (model, layer)**

The model is an nn.Module object which takes a tensor as input, and the shape of this tensor is the shape of data and returns a single dimensional output. If the input is a tuple, the returned shap values will be for the input of the layer argument.

**data**

if framework == ‘tensorflow’: [numpy.array] or [pandas.DataFrame],

if framework == ‘pytorch’: [torch.tensor]

**session: if framework == ‘tensorflow’, None or tensorflow.Session**

The TensorFlow session that has the model we are explaining. If none is passes then it tries to find the right session by first looking for a Keras session, then back to the default TensorFlow session.

**learning\_phase\_flags: None or list of tensors**

If we have our own custom learning phase flags pass them here. When explaining a prediction, we need to ensure that we are not in training mode, since this changes the behavior of ops like batch norm or dropout.

If None is passed, then it looks for tensors in the graph that look like learning flags (this works with Keras models). Note, that we assume all the flags should have a value of False during predictions (and hence explanations).

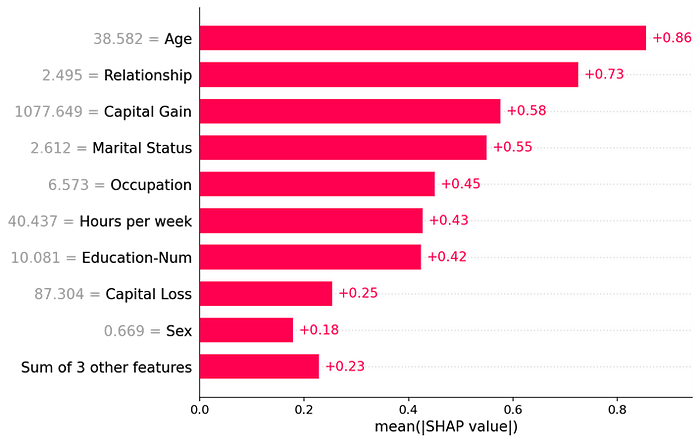
**SHAP GRAPHS**

This section of the article discusses about the different SHAP graphs. In this section, we will look at the code for plotting SHAP graphs in python and also the output for that code.

**BAR PLOT**

This SHAP graph create a bar plot of a set of SHAP values and helps us understand the feature importance in formulating the model output. The code for plotting this graph in python is given below.

shap.plots.bar(shap\_values)



By default, the bar plot only shows a maximum of ten bars, the following code shows the way we can display selective number of features on shap bar plot. Also, the second line of code helps us print the shap bar plot for a particular instance ‘x’.

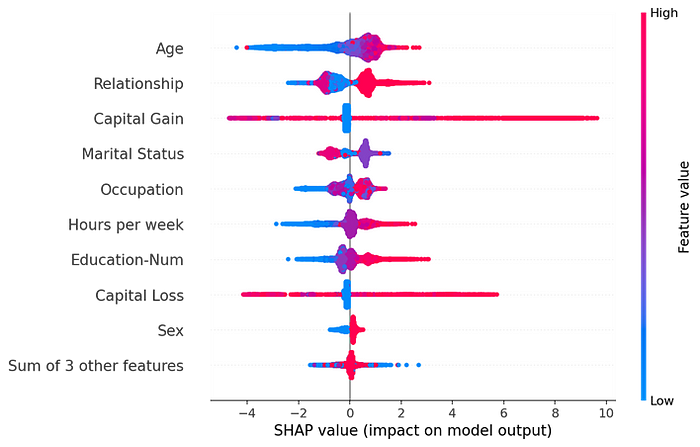
shap.plots.bar(shap\_values, max\_display = n) # code for printing n features on bar plot   
  
shap.plots.bar(shap\_values[x]) # code for printing bar plot for one instance ‘x’

For binary classification problems, the bar plot is printed in two colours in single bar, as for positive and negative class, red for positive and blue for negative.

**BEESWARM PLOT**

The Beeswarm plot is designed to display an information-dense summary of how the top features in a dataset impact the model’s output. Each instance is represented by a dot for each different feature. The position of the dot is decided by the shap values and the color by the original value of the feature. The code for plotting this shap plot is given below.

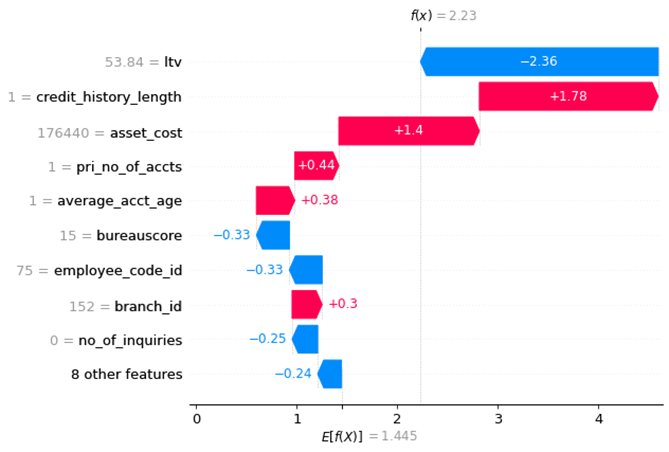
shap.plots.beeswarm(shap\_values)   
  
shap.plots.beeswarm(shap\_values, max\_display = n) # code for printing n features on the plot   
  
shap.plots.beeswarm(shap\_values[x]) # code for printing the plot for one instance ‘x’



**WATERFALL PLOT**

Shap Waterfall plot is used to get an explanation of a single prediction by the model, and thus, they expect a single row of an explanation object as an input. The code for printing the waterfall plot as given below prints the waterfall plot for an instance ‘x’.

shap.plots.waterfall(shap\_values[x])   
  
shap.plots.waterfall(shap\_values, max\_display = n) # code for printing n features on the plot

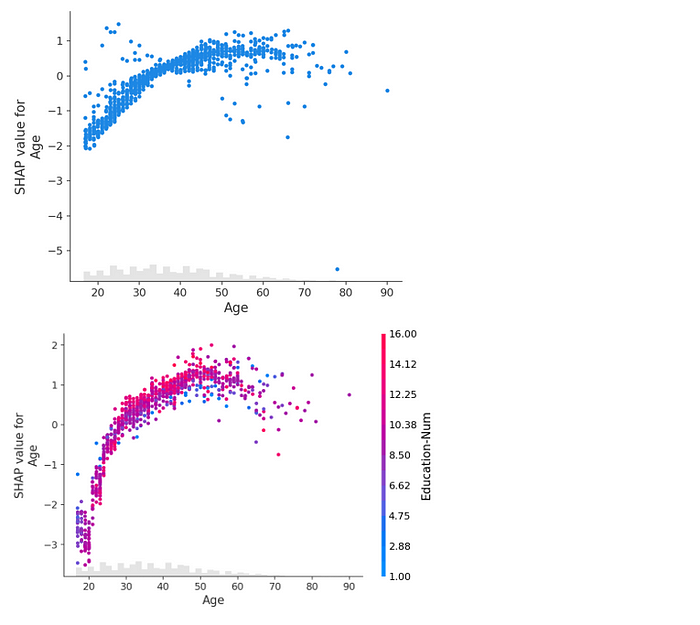


Here, the red bar shown positive contribution and blue bar for negative contribution.

**SCATTER PLOT**

As waterfall plot only shows a single sample of the data, we are not able to understand the impact of changing value of a feature. For this, we can use a scatter plot. In a scatter plot, each dot is a single prediction (row) from the dataset. The x-axis is the value of the feature from the dataset, and the y-axis is the SHAP value for that feature. The light grey area at the bottom of the plot is a histogram got the distribution of the data values. The code for plotting the shap scatter plot is given below.

shap.plots.scatter(shap\_values[ :, “a”]) # scatter plot for feature “a”  
  
shap.plots.scatter(shap\_values[ :, “a”]), color = shap\_values) # code for adding color to the plot

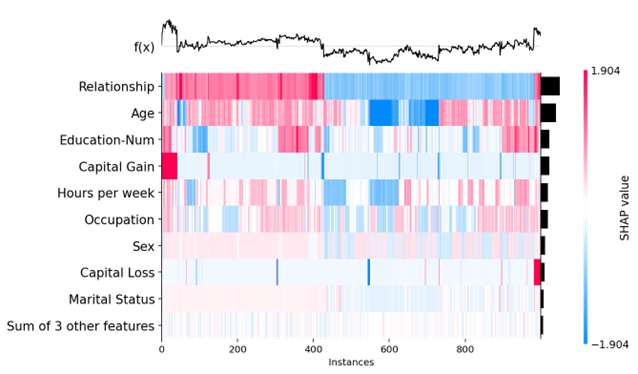


We can also control/decide the feature to be used for colouring the scatterplot by specifying the feature column to the “*color*” parameter in the code above.

**HEATMAP PLOT**

The heatmap plot is designed to show the population structure of a dataset using supervised clustering and a heatmap. Supervised clustering involves clustering the data points by their explanations and not by their original feature values. In a heatmap plot from shap, the x-axis is the instances, and the y-axis is the model inputs while the shap values are encoded on the color scale. The code for plotting this plot in python is given below.

shap.plots.heatmap(shap\_values) # heatmap plot  
  
# controlling the order of features on the heatmap  
shap.plots.heatmap(shap\_values, feature\_values = shap\_values.abs.max(0))

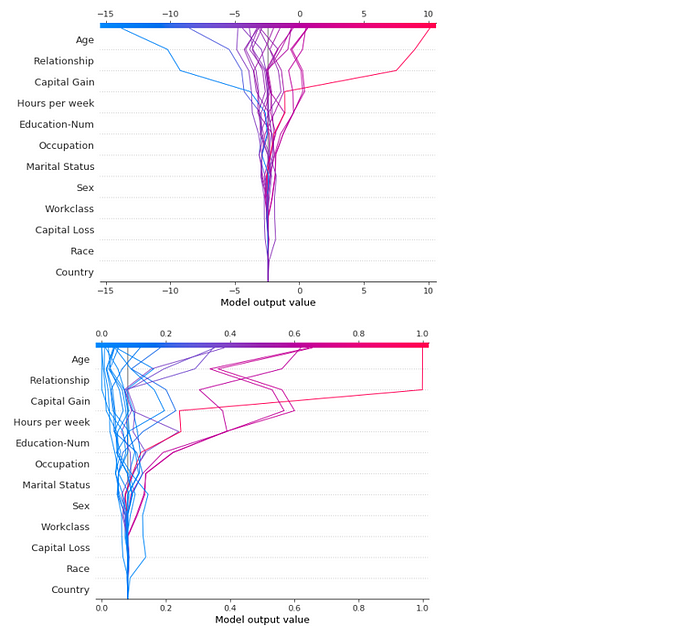


We can also sort the ordering of features as well as instances by using “*feature\_values*” and “*instance\_order*” parameters.

**DECISION PLOT**

Decision plot is the visualization of the model decisions, where the explanation is plotted as line which explains a single model prediction. However, SHAP says that this plot is not informative by itself, we use it only to illustrate the primary concepts. The x-axis of this plot represents the model’s output, and the y-axis lists the model’s features. Here, the importance is calculated over the observations plotted. Each observation prediction is represented by a coloured line. At the top, each line strikes the x-axis at its corresponding observation’s predicted value. This value determines the color of the graph. The bottom of the graph is the base value, which is calculated by adding the shap values of each feature. The code for plotting a decision plot is given below.

shap.decision\_plot(expected\_value, shap\_values, features\_display) # decision plot  
  
# using ‘link = logit’ to transform log odds to probabilities   
shap.decision\_plot(expected\_value, shap\_values, features\_display, link = ‘logit’)



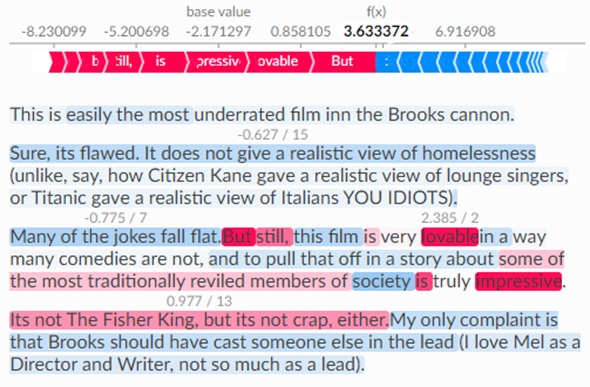
We can also plot the decision plot for any particular instance or for a particular classification class. The code for these are as follows.

y\_pred = (shap\_values.sum(1) + expected\_values) > 0  
  
misclassified = y\_pred!=y\_test[select] # defining ‘misclassified’  
  
shap.decision\_plot(expected\_value, shap\_values, features\_display, link = ‘logit’, highlight = misclassified)

**TEXT PLOT**

The Text plot is an explanation of a string of text using colouring and interactive labels. This plot is specifically designed for language models. The text plot is a visualization of model output explanation for a single instance. The code for plotting the text plot for your language model is as follows.

shap.plots.text(shap\_values[x])

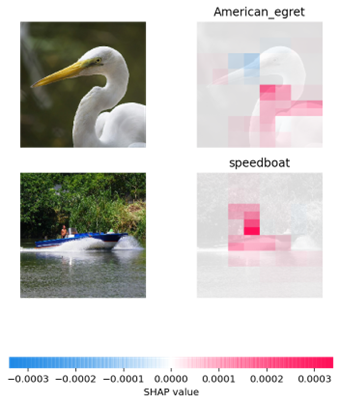


This is an explanation for a sentiment analysis model, and the text plot shows the contribution of different words in model’s output. The contribution of different words and phrases are highlighted by different colours based on their corresponding shap values.

**IMAGE PLOT**

The Image plot is an explanation for the deep learning models used for image classification of object detection. This plot shows the contribution of different pixels of the image for formulating the final output of the model. The code for this plot is given below.

shap.image\_plot(shap\_values)



**HOW TO IMPLEMENT SHAP FOR ML MODEL INTERPRETATION**

This section of the article explains the steps following which you can plot the shap plot discussed earlier in this article.

**INSTALLING AND IMPORTING SHAP**

To use the SHAP library to understand your black box machine learning models, the first step is to install the SHAP library. We can install SHAP by using the following pip command:

!pip install shap



After installing the shap library in your python notebook, the next thing is to import the shap library. Without importing, you cannot use shap. The code for importing the shap library is given below.

import shap

**FITTING SHAP EXPLAINER**

After importing the shap library, we now need to get the shap\_values and expected\_values in order to plot the different shap plots. Here, we are assuming that your machine learning model is already trained and tested and now we are trying to explain those models using shap. The following lines of code can be used to fit the shap explainer on your machine learning model or algorithm you want to get explanation for.

# computing SHAP values for bar, beeswarm, and waterfall plot   
background = shap.maskers.Independent(X, max\_samples=1000) #read ‘Shap Explainers’  
explainer = shap.Explainer(model, background)  
shap\_values = explainer(X)  
  
# or  
explainer = shap.Explainer(model, X)  
shap\_values = explainer(X)

These lines of code can be used to get shap values for plotting the shap bar plot, beeswarm, and waterfall plot. The code for getting the shap values for decision plot, heatmap plot, and scatter plot are as follows.

**Shap Values and Expected Values for Decision Plot**

# computing SHAP values for decision plot  
feature\_names = ["List of Features"]   
  
# Create the explainer and get the shap values  
explainer = shap.Explainer(model)  
shap\_values = explainer.shap\_values(X)  
  
# Select the SHAP values for a specific sample  
sample\_index = 0 # Index of the sample you want to explain  
sample\_shap\_values = shap\_values[sample\_index]

**Shap Values for Heatmap and Scatter Plot**

# Create the explainer and get the shap values  
explainer = shap.Explainer(model, X)  
  
# slicing the dataset to 1000 instances for lesser computation time  
shap\_values = explainer(X [ : 1000])

**Shap Values for Text Plot**

# Build an explainer  
# ‘m’ is either the model or the prediction function, ‘tokenizer’ is the tokenizer for the language model  
explainer = shap.Explainer(m, tokenizer)  
  
# Explain the model's predictions on IMDB reviews (for example, change the model as per your need)  
imdb\_train = nlp.load\_dataset("imdb")["train"]  
shap\_values = explainer(imdb\_train[:10], fixed\_context=1)

**Shap Values for Image Plot**

# define a masker that is used to mask out partitions of the input image, this one uses a blurred background  
masker = shap.maskers.Image("inpaint\_telea", X[0].shape)  
  
# By default, the Partition explainer is used for all partition explainer, f is model or the prediction function  
explainer = shap.Explainer(f, masker, output\_names=class\_names)  
  
# Here we use 500 evaluations, you can use as per your need  
shap\_values = explainer(X[1:3], max\_evals=500, batch\_size=50, outputs=shap.Explanation.argsort.flip[:1])

**PLOTTING SHAP GRAPHS**

After getting the shap values and expected values for the machine learning or deep learning models created, the next step to get the explanation for the models is to plot the shap graphs. The code for all different types of shap graphs are already explained in detail in section “Shap Graphs”.