Shapley Values & SHAP
ML Interpretability & Feature Importance

Written by: Jessica Kim; Summer 2022 INSPIRE Research Intern, Computational Biomedicine

Duration: June 27- Sept 2, 2022

Questions about SHAP Jupyter Notebook?

- Reach me at jessicaaaakim0@gmail.com or 818.923.4730
- Click on the link to access the SHAP notebook source code on my GitHub repository for the internship
- Click on the link to access the source code for SHAP

PREFACE

This document is a summarization/compilation of notes from papers and other resources found on Shapley Values and SHAP. There are links embedded throughout that will redirect you to the original source the information was found plus more details on how/why the methods are implemented. The content is general information for anyone getting into using SHAP for a project however, a more comprehensive look into various avenues of SHAP implementation can be found in the links. Note from the author: I do not have a full understanding (due to lack of experience in the field) of the published papers, hence, the simplified overview of Shapley Values and SHAP. I wrote this with the simple goal to provide a brief enough overview for the next person taking on this project …think of this as the 'starting point'. This is to help condense what anyone would generally need to know about implementing SHAP and shorten the time it would take to catch up.

I hope this is helpful for anyone starting to look into implementing SHAP to create more accurate global and local explanations for their ML models. The links and other resources can be accessed through this document, my GitHub repository for the project, or the INSPIRE presentation I put together that contains additional articles that provide examples on how to implement SHAP.

What are 'Shapley Values'?

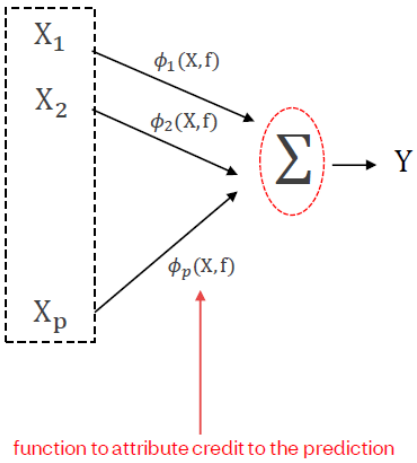
Shapley Values is a model-agnostic method from cooperative (also referred to as 'coalitional') game theory, coined by Lloyd Shapley in 1951. It tells us how to fairly distribute the 'payout' contribution among players. In the case of Machine Learning, we are attempting to fairly distribute a model's predicted Shapley Value score among all features (marginal contribution) used, including ones that were not used. Marginal contribution is another way of referring to Shapley Values, we are looking at how much each feature impacts (looking at magnitude) the final prediction of a model. Not only does Shapley Values use feature importance to show us which features are most important in a model's prediction but the bigger picture is helping us interpret and better understand ML model decision-making.

- **4 Axioms of Shapley Values**
  - **Efficiency**
    - Feature contributions must add up to the difference between the prediction for 'x' and the average
    - All Shapley Values calculated for each feature should be the sum of the predicted value of the model output/prediction
  - **Symmetry**
    - Contribution values of two interchangeable features should be equal regardless of the feature name
    - Simply…Feature names should not change/impact the contributions of the model's prediction
  - **Dummy**
    - Missing features added to any coalition should be given a Shapley Value of 0.0
    - Missing/unused features should not impact the model's output prediction
  - **Additivity**
    - Shapley value for each coalition is averaged over '*n*' coalitions in the model → averaged out to get the Shapley Value of the feature for that one model

- **To simplify...**
    - "The Shapley value is the average marginal contribution of a feature value across all possible coalitions." - Christophe M (kaggle)
    - "Shapley value is calculated by computing a weighted average payoff gain that player '*i*' provides when included in all coalitions that exclude '*i*'" - Divya Gopinath
- **Calculating Shapley Values**
    - Calculated by computing a weighted average of each feature across all coalitions (feature combinations)
    - Compares marginal contribution (Shapley value) of each feature to the model's overall predicted score
- **Understanding the forumla**

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

- ϕ **(Phi)**: measure of association of two binary variables (classification)
- ***S***: coalition/subset of players (combination of features with & without player ***i***)
- ***N***: total number of players
- ***i***: player (feature values of the instance contributing to the gain)
- ***v***: real-value payoff of the prediction from the features involved



- **Assumption**
    - Each feature is independent and will not provide accurate values if two or more features are highly-correlated (highly-correlated features must be removed when selecting features)
        - **Relationships between features are NOT CAUSAL, we look for the correlation between features**
- **Advantages of S.V**
    - Demonstrated to be considered an explanation method with a solid theory based on the 4 axioms previously mentioned
    - Allows contrastive explanations of the prediction for a single datapoint, subset of features (coalitions), and the average prediction of an entire dataset … rather than just a global explanation of the model (entire dataset)
    - Can be computed without having to know an ML's mechanics in making its prediction
- **Disadvantages of S.V.**
    - "Shapley value method suffers from the **inclusion of unrealistic data instances** when features are correlated."
    - Cannot be used to make predictions by changing the inputs
    - Not a good method to use to explain sparse explanations
    - Computational time exponentially increases for every feature used:

$$2^{K = total\ \#\ of\ features}$$

What is 'SHAP'?

SHAP, or **SH**apley **A**dditive ex**P**lanations, is software created by **Scott M. Lundberg** and **Su-In Lee** in 2017 adapted from Shapley Values that provides both global & local interpertability. SHAP is considered to be a model-agnostic 'unified framework' thats serves as a measure of feature importance using the ML model as the explanation object itself. In other words, using an ML model to explain its own predictions. The "unified" aspect of SHAP refers to the representation of Shapley Values as a collection of current model explanation methods used to understand additive feature attributions (learn more about method proof at NeurIPS). SHAP satisfies 3 properties in conjunction with the 4 axioms of Shapley Values as previously mentioned. The goals of SHAP are to address the long-standing issue of ML interpretabiity in ML models, especially for more complex models such as neural networks. SHAP was created with the intention of working towards 'correct interpretation' of a model's prediction which may help establish 1) trust 2) improve computational performance 3) support understanding of model predictions/behaviors.

- **3 Properties of SHAP:**
  - **Local accuracy**

    Satisfies Efficiency axiom of Shapley Values
    Recall the 'Efficiency' axiom:

    - Feature contributions must add up to the difference between the prediction for 'x' and the average
    - All Shapley Values calculated for each feature should be the sum of the predicted value of the model output/prediction
  - **Missingness**

    Missing feature gets an **attribution of zero**
    In theory, the missing feature could have an arbitrary value without hurting the local accuracy but this is **ONLY** relevant to features that are **CONSTANT**
    Satisfies 'Dummy' axiom → missing features should not impact the model prediction
  - **Consistency**

When a feature has a higher impact in model A compared to model B, the **feature importance value** of model A should **ALWAYS** remain **LARGER** than model B regardless of changes in the models
Feature importance value of a model **increases or remains the same** when the contribution value of a feature also increases or remains the same

A Unified Approach to Interpreting Model Predictions: Section 4.1 - Model-Agnostic Approximations (Lee & Lundberg 2017)

SHAP calculates Shapley values based on the "conditional expectation function of the original model" and adheres to Properties 1 - 3. To approximate conditional expectations, there are two types of model agnostic approximation methods for SHAP that assume feature independence: Shapley sampling values and KernelSHAP. Shapley sampling values are based on the permutated version of Shapley values of the original Shapley Value equations. Ideally, this method is better for a smaller set of inputs as it calculates the values in separate sampling estimates for each feature. KernelSHAP is a regression approach that estimates all SHAP values using weighted linear regression rather than the original Shapley value equation. This method is based on both LIME and Shapley values (Shapley value is used to satisfy missing properties seen in LIME). The connection between linear regression and Shapley values is the difference of means to summarize a set of data points as a best least squares point estimate.

NOTE: As always, bias, estimation, and sampling number are key concerns when using these methods. The original KernelSHAP has been proven to be consistent but not unbiased. The suggested idea to improve KernelSHAP is an alternative called "unbiased KernelSHAP". The unbiased KernelSHAP method simply aims to eliminate biased estimators, reduce variance for convergence detection, and uncertainty estimates, and establish an appropriate dataset sampling technique. More can be read from the paper written by Ian Covert and Su-In Lee, "Improving KernelSHAP: Practical Shapley Value Estimation via Linear Regression".

Implementing SHAP

Implementing SHAP to create values and summary figures for a model normally requires an Explainer object, background datasets (typically train dataset X), test datasets, dataset feature names, and ML model type.

Recall that from a conceptual standpoint, the creation of an Explainer object is using the model itself to explain its own predictions. Considering the differing structures between various ML models, there are Explainers specifically designed to compute better approximations of Shapley values. The first two types of Explainers that can be used on any ML model are 1) .Explainer() 2) .KernelExplainer(). While both Explainers can be used to compute Shapley Values for any ML model, the difference between .Explainer() and .KernelExplainer() is that the Kernel explainer "uses a special weighted linear regression to compute the importance of each feature". Below is a list of commonly used Explainers (w/ default parameter setup) used for most ML models and can be customized to focus on specific constraints:

- **List of Commonly Used Explainers**
  - **Logit Link parameter is optional for most formats of the Explainers but can be used if the probability is not what you want to get out of your model's prediction & behavior**
  - **.Explainer()**

    Main primary explainer for the SHAP library of the SHAP package

    Can take any combination of model type and masker (normally background data matrix) to return an "Explainer object" of the model itself

    Can be thought of as a very generalized explainer to provide a simple look at the feature contribution to the prediction → other explainer types can be used other than this one if user requires faster computation or requires an explainer that best fits the behavior of a specific model type
  - **.KernelExplainer()**

    One of the original SHAP explainers that integrate both LIME and Shapley values

    Based on the model-agnostic approximation method, KernelSHAP

    Improves sample efficiency

    Mostly seen to be used on models such as Support-Vector Machine and K-Nearest Neighbors

Better used for smaller inputs to explain a model (uses smaller dataset sampling)

Declaring the Kernel Explainer

- General format → **shap.KernelExplainer(model, background dataset)**
- If model provides probability instead of log-odss ratio → **shap.KernelExplainer(model, background dataset, link="logit")**

Examples can be found here

- **Example 1: Census income classification with scikit-learn**
- **Example 2: Diabetes Regression w/ Scikit Learn**

It is important to note that fnlwgt (a statistical reweighting term) is the dominant feature in the 1000 predictions we explain. This is because it has larger variations in value than the other features and so it impacts the k-nearest neighbors calculations more.

```
[4]: f = lambda x: knn.predict_proba(x)[:,1]
med = X_train.median().values.reshape((1,X_train.shape[1]))
explainer = shap.KernelExplainer(f, med)
shap_values_single = explainer.shap_values(X.iloc[0,:], nsamples=1000)
shap.force_plot(explainer.expected_value, shap_values_single, X_display.iloc[0,:])
```

- **.TreeExplainer()**

Does not require a master when creating the Explainer object for tree-based models

Exact approximation for calculating Shapley Values in tree-based models such as Random Forest, Decision Tree, Extreme Gradient Boosting, and more

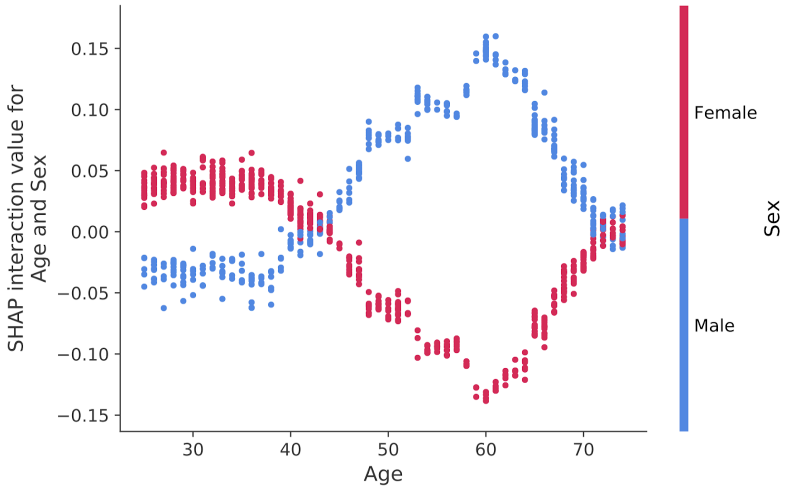
- Fast, local explanations that are **consistent**
- Reducing **time complexity** from exponential to polynomial time
- Sums calculation to **specific node** rather than a summation of all subsets of features
- Can measure **feature interaction effects** which can help uncover significant patterns that could easily be miss
- **Combines local explanations** to provide a consistent, **global** interpretation of a model's behavior
- Local explanation of feature dependence in TreeExplainer **can reveal patterns and individual variability of features**

Proven to work with XGB, CatBoost, and LightGBM)

Can use ".shap_values()" to generate SHAP values

Can use ".shap_interaction_values()"

- SHAP interaction values distribute credit to ALL PAIRS of players (default distribution of credit to each individual player)
- Captures local interaction effects as a matrix of features rather than as a vector (default)
- While the main effects of feature effects are apparent in global interpretations, interaction values can help us understand just exactly the **EXTENT** of the interaction a pair of features has on the prediction → simply provide **CONSISTENT** results



This figure (interaction values) corresponds to a dataset that looks at the increased risk of death in males versus females. The interaction values demonstrate that the risk of death INCREASES for males at age 60 compared to females around that same mark.

- **.LinearExplainer()**

  Used for linear models (e.g. Linear Regression, Logistic Regression)

  Accounts for correlations among input features

  Default parameter format → **shap.LinearExplainer(model, background dataset)**

  Can use ".shap_values()" to generate SHAP values

  Optional parameters include:

  - **"Nsamples"**

    Number of samples to estimate the transformation matrix

  - **"feature_perturbation"**

Interventional: default; "stays true to model" for features are given credit IF used by the model

Correlation_dependent: "stays true to data"; considers how the model would behave based on correlations in input data

slundberg / shap / tests / explainers / test_linear.py

```
from sklearn.linear_model import Ridge

    np.random.seed(0)

    coef = np.array([1, 2]).T

    # generate linear data
    X = np.random.normal(1, 10, size=(1000, len(coef)))
    y = np.dot(X, coef) + 1 + np.random.normal(scale=0.1, size=1000)

    # train linear model
    model = Ridge(0.1)
    model.fit(X, y)

    # explain the model's predictions using SHAP values
    explainer = shap.LinearExplainer(model, X)

    values = explainer.shap_values(X)

    assert values.shape == (1000, 2)

    expected = (X - X.mean(0)) * coef
    np.testing.assert_allclose(expected - values, 0, atol=0.01)
```

slundberg / shap / tests / explainers / test_linear.py

```
def test_perfect_colinear():
    import shap
    from sklearn.linear_model import LinearRegression
    import numpy as np

    X,y = shap.datasets.boston()
    X.iloc[:,0] = X.iloc[:,4] # test duplicated features
    X.iloc[:,5] = X.iloc[:,6] - X.iloc[:,6] # test multiple colinear features
    X.iloc[:,3] = 0 # test null features
    model = LinearRegression()
    model.fit(X, y)
    explainer = shap.LinearExplainer(model, X, feature_dependence="correlation")
    shap_values = explainer.shap_values(X)
    assert np.abs(shap_values.sum(1) - model.predict(X) + model.predict(X).mean()).sum() < 1e-7
```

[Click here for an example of using LinearExplainer() for Logistic Regression]

- **Refer to SHAP Explainers documentation for more information**

However, a small caveat with constant updates to the SHAP package is that the instantiation of the Explainer object to generate SHAP values is dependent on the ML model type. Some plots are better suited to display the interaction effects of features compared to others. When it comes to creating SHAP figures on trained models (based on this experience working with it), classification requires an 'extra' step when passing parameters for the test dataset and SHAP values matrix. Often, the figures will display two main colors, red & blue, to indicate a positive and negative effect of a feature on the final model's prediction. A greater length in bar size/density of dots indicates a greater impact on the prediction.

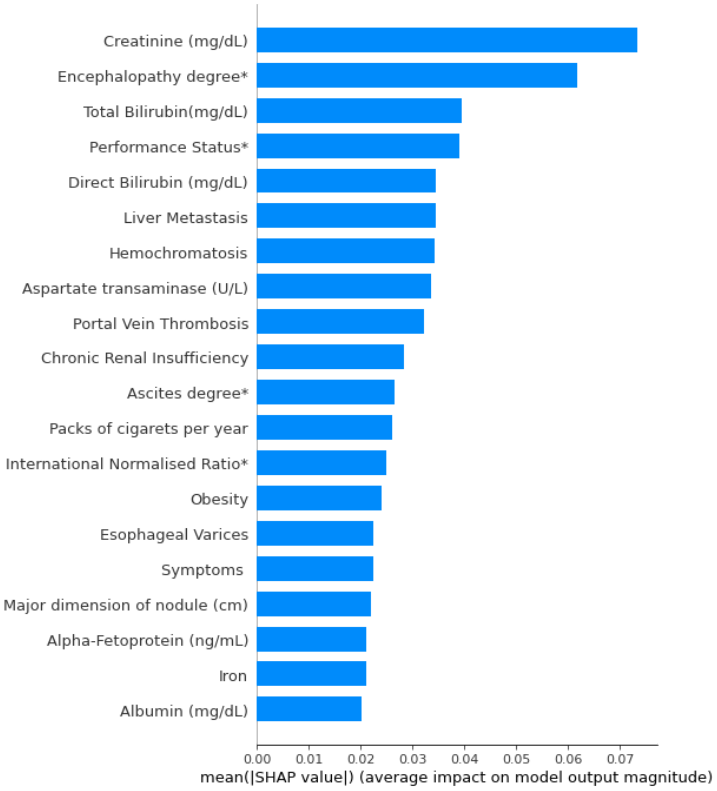
Because the models in STREAMLINE are focused on binary classification, the SHAP values that are returned (as a matrix) are split into Class 0 and Class 1. To look at a specific class (0 or 1), the parameter "shap_values" must indicate a class index → "shap_values[0]" or "shap_values[1]". If creating a figure requires passing the test dataset as well AND the user wants a specific class → the parameter must match "shap_values[0]" → "X_test.iloc[row]" or "X_test[row]". This can also take form as "shap_values[0][predict instance]" and "X_test[predict instance]". You can find examples of this in "shap_summary()" section of the SHAP notebook. Another thing to keep in mind is that some figures require using the model's explainer "expected_value" ("expected_value[0]" or "expected_value[1]") function which is the model's "expected" values over the whole dataset. It is what a model would predict without knowledge of the data where the expected values are close, but not equal exactly, to class frequencies. Binary classification classifies the outcome into 0 and 1 which in turn, models that produce two classifications (0 or 1) will result in having two expected values when using this call function. With the most recent updates to the SHAP package, using the waterfall plot and bar plot may require looking to the source code to fully understand how to implement these with STREAMLINE results because the most common way of applying doesn't necessarily work **(trust me, I've tried so it'd be great if the next person working on this can figure out why some SHAP figures don't display as it should if I were to train an ML model from scratch….had to do this for Decision Tree, Naive Bayes, Logistic Regression to work my way around the errors)**. The "expected_value" function is not applicable to

all models, especially for models that perform permutations on the dataset. Below is a list of commonly used SHAP figures and what they usually look like when implemented into a program:

- **List of Commonly Used SHAP Plots (global & local explanation)**
  - Summary plot
    - Summarizes shap_values for global and local interpretation
    - To call function → **example: shap.summary_plot(shap_values, X, feature_names, plot_type='violin', show=False)**
    - Can either take the matrix of shap_values or a single row for a single-prediction explanation of the model
    - Versatile plot type as it can display for single output or for multi-output
      - 'Dot'

        For single output
        Can look similar to a Beeswarm plot or a scatter plot
      - 'Bar'

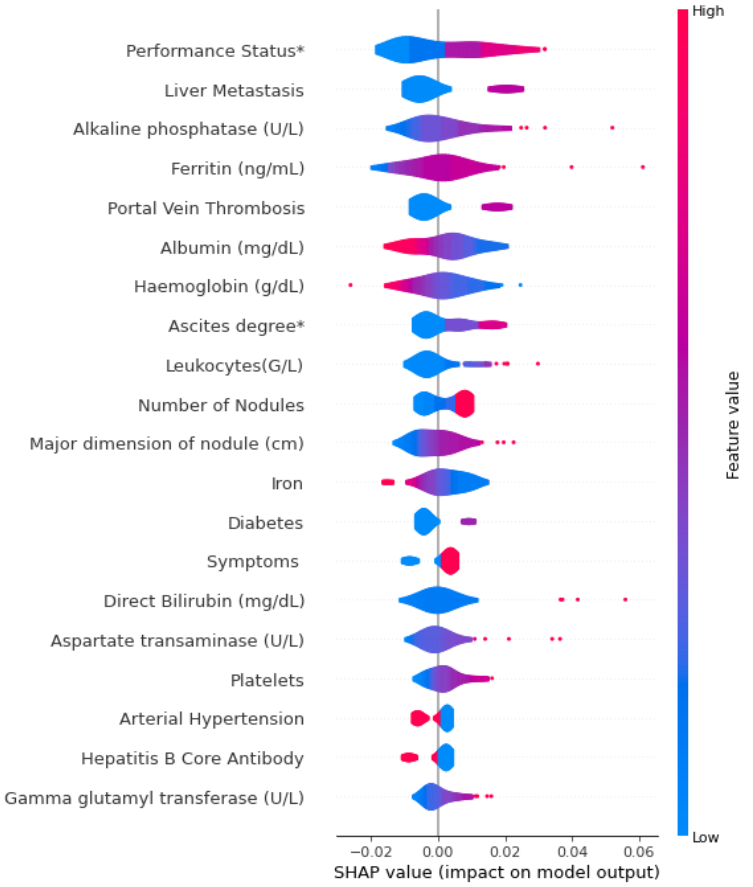
        The default format for multi-output
        NOT THE SAME as the "Bar Plot"
        Uses feature ordering based on the feature's SHAP value but doesn't display the actual SHAP value next to the feature



      - 'Violin'

        For multioutput

Will display if specified by the user to use “violin” as plot_type argument instead of the default argument “bar”
Looks like a wavelength of shap_values colored by the interaction effects among features and the contributions to the prediction

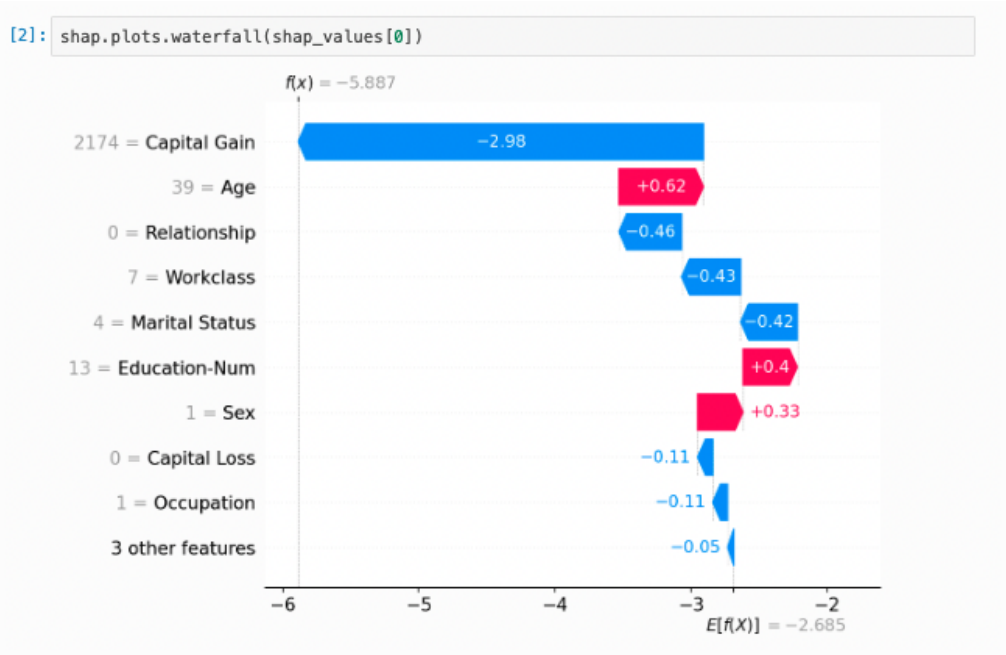


Both images of the summary bar plot and summary violin plot are taken from the actual output from running the SHAP notebook. The violin plot type is preferred if the user is looking for a global explanation of the interaction effects of each feature on the mode’s prediction.

- Waterfall
  - This type of plot is designed to display explanations for **individual predictions**
  - Looks similar to the structure of a local Bar plot
  - User can specify max number of features to display in plot (“max_display = 10”)
  - **Issue →** updated version of SHAP may require looking at source code for proper implementation of waterfall plot by calling **waterfall legacy**
    - Example of this implementation (from my SHAP notebook):

shap.plots._waterfall.waterfall_legacy(explainer.expected_value[1], shap_values[1][random_single_predict], testX[random_single_predict], feature_names, show=False)

- Parameters require for waterfall legacy need the single row of the Explainer object
- Notice that the first parameter requires the expected_value which means this plot type may only work for certain models



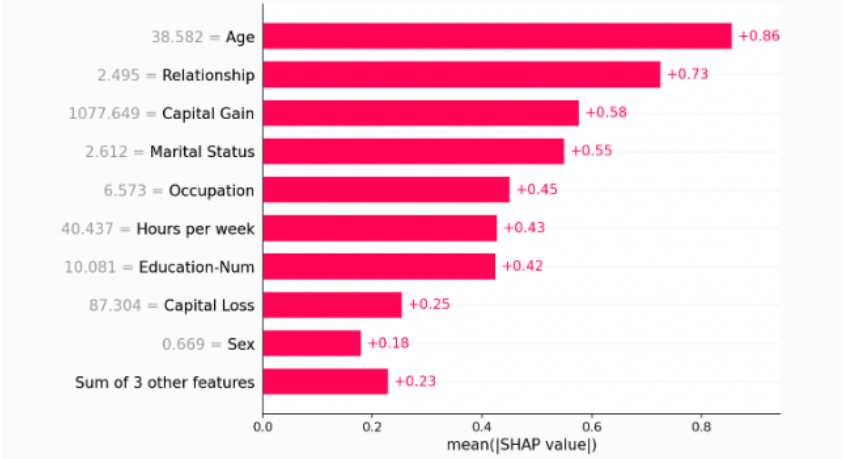
This is what a typical waterfall plot would look like when using the default function call to shap.plots.waterfall(shap_values[single-predict]). Results may vary depending on which model and Explanation object is passed. Please refer to the source code for this.

- Bar plot
  - Displays SHAP value for each bar on the graph for each feature used in the prediction
  - Can generate global interpretation and local explanations of a single instance
  - Each feature is taken from the mean absolute value over all samples of the dataset
  - Not the same as the summary plot in the shape of a bar graph
    - The difference is that the Bar plot is able to display the SHAP feature value adjacent to the bar of the features
    - Summary bar plot simply shows the features based on importance to the prediction and DOES NOT display the feature value

Global bar plot

Passing a matrix of SHAP values to the bar plot function creates a global feature importance plot, where the global importance of each feature is taken to be the mean absolute value for that feature over all the given samples.

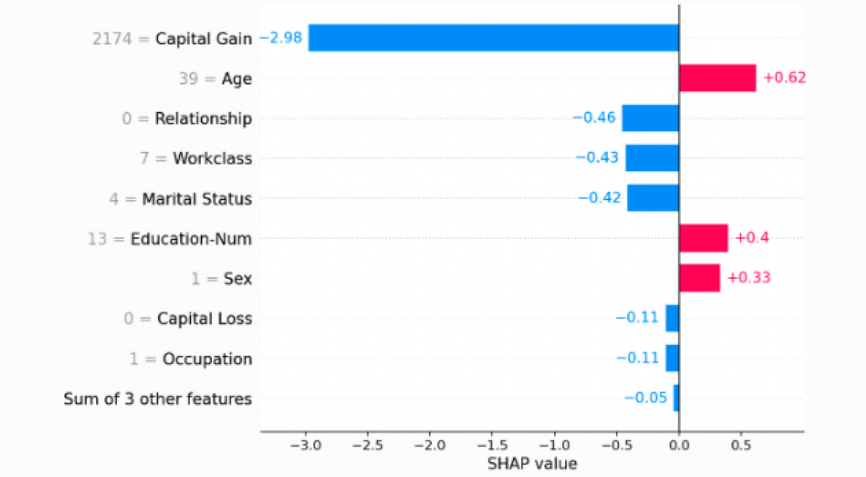
```
[5]: shap.plots.bar(shap_values)
```



Local bar plot

Passing a row of SHAP values to the bar plot function creates a local feature importance plot, where the bars are the SHAP values for each feature. Note that the feature values are show in gray to the left of the feature names.

```
[7]: shap.plots.bar(shap_values[0])
```

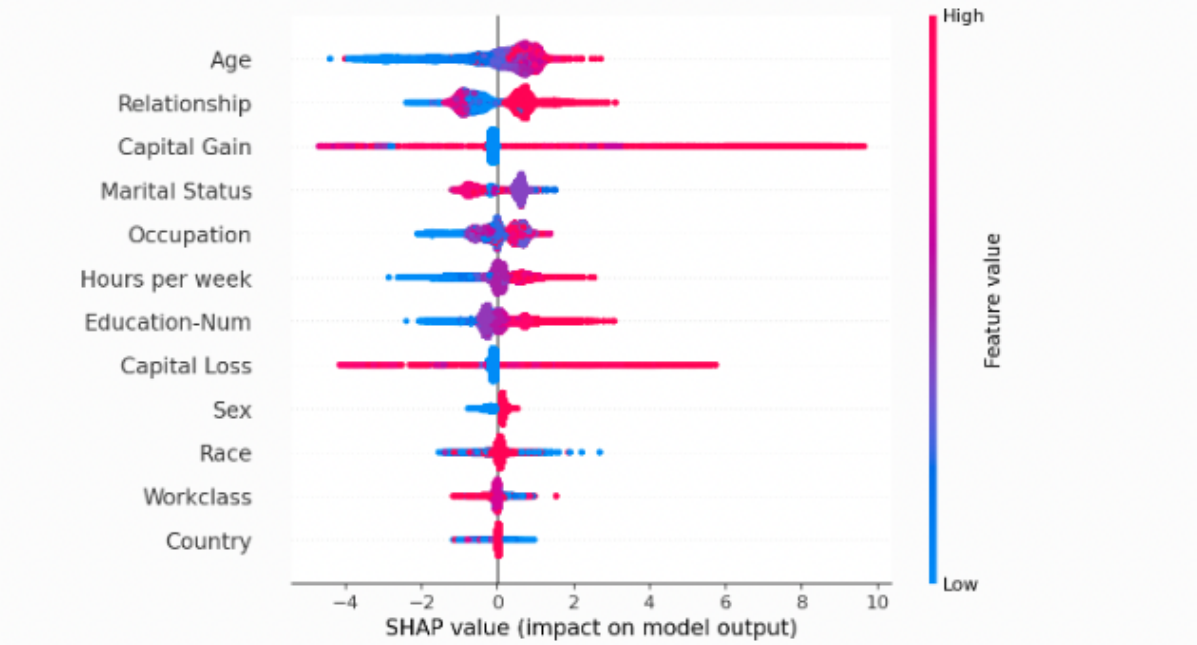


Execution should work for any given model but in the case it doesn't, consider: the model type, how models were trained, dataset size, data structure of SHAP values returned

- Beeswarm plot
  - Displays an information-dense summary of how the top features in a dataset impact the model's output

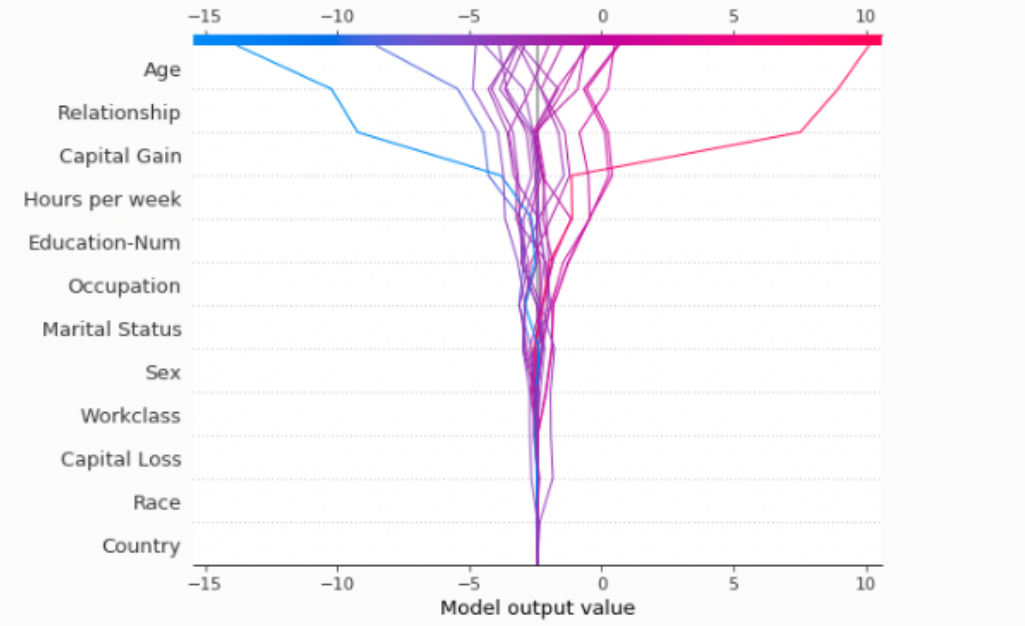
- Features are ORDERED BY DEFAULT using the mean absolute value of the feature's SHAP value
- Can choose to display however many features in the figure → **max_display = 10 or max_display = 20**
- **Applicable to most models**
  - **For global explanation → pass parameter "shap_values"**
  - **For local explanation → pass parameter "shap_values[0]"**

```
[3]: shap.plots.beeswarm(shap_values, max_display=20)
```



- [Decision plot]
  - Slightly more challenging to interpret visually
  - Helps users understand how the COMPLEX models reached the predictions
  - **Supports hierarchical cluster feature ordering and user-defined feature ordering.**
  - **Only useful when:**
    - Display multioutput decisions
    - Show a large number of feature effects
    - Display cumulative interactions of features
    - Explore feature effects for a RANGE of feature values
    - Compare/contrast predictions between different models
    - Identify feature outliers
    - Identify typical prediction paths

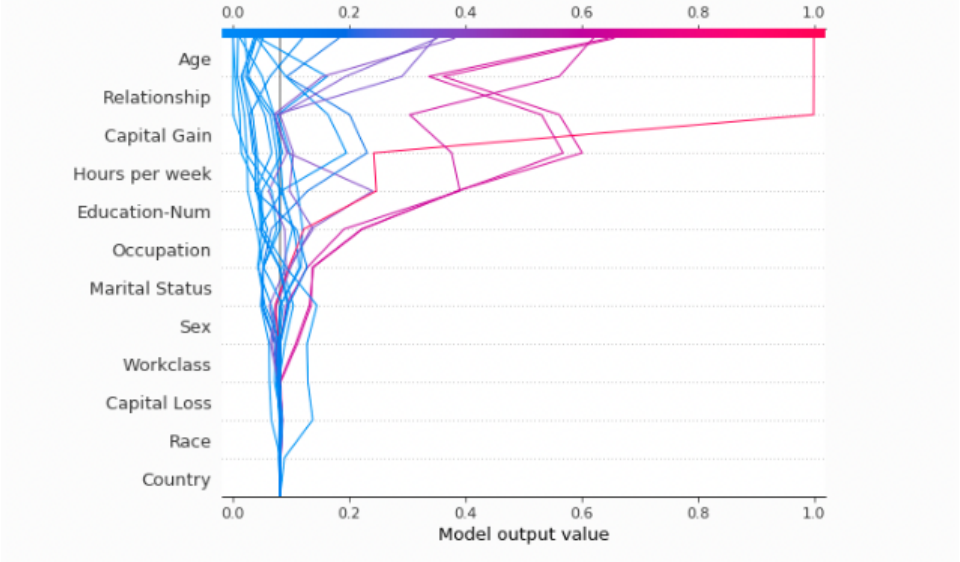
```
shap.decision_plot(expected_value, shap_values, features_display)
```



"This is a global explanation of a dataset where the x-axis uses the model's expected_value, each line strikes the x-axis at its corresponding observation's predicted value. Moving from the bottom of the plot to the top, SHAP values for each feature are added to the model's base value."

Like the force plot, the decision plot supports `link='logit'` to transform log odds to probabilities.

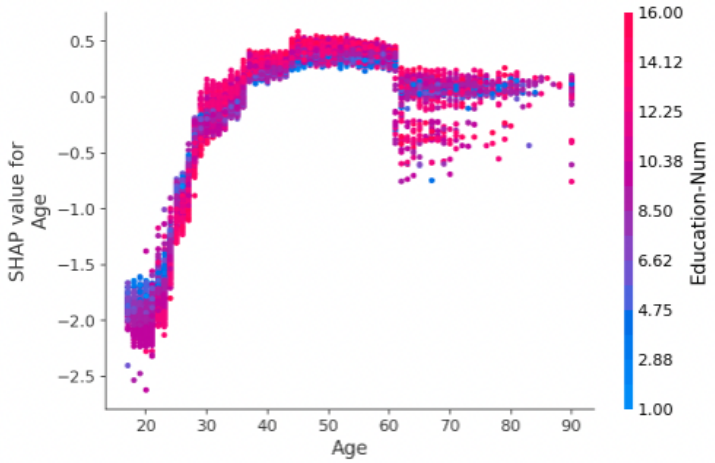
```
[4]: shap.decision_plot(expected_value, shap_values, features_display, link='logit')
```



Another example of how decision plots can be transformed to display another "interpretation" of features to the model's prediction (log-odds transformation). Other types of functions to transform decision plots can be found in the link attached to the section titled "Decision plot".

- Scatter plot
  - Call function for scatter plot → **shap.plots.scatter(shap_values[:, "Feature"], color=shap_values)**
  - **The most recent version of the function call can be found in the source code (revised since June 14, 2022)**
  - Dependence plots (scatter plot)
    - **NOTE: This function for the dependence plot may or may not work considering the SHAP package update**
    - Type of scatter plot that can be called as → **shap.dependence_plot(index of feature we want to plot, shap_values, X aka data matrix that is either a NumPy array or dataframe)**
    - "Shows the effect a single feature has on the predictions made by the model."
    - Finds and identifies the feature column with the strongest interaction with the feature being examined
    - Plots the dataset value of the feature (taken from the X dataset matrix) on the x-axis while the feature's SHAP value is plotted on the y-axis
    - Can be colored based on interaction value among features
    - Vertical dispersion refers to the interaction effects of the feature
    - **For examples of how to use the dependence plot in various ways, click here**

```
# The first argument is the index of the feature we want to plot
# The second argument is the matrix of SHAP values (it is the same shape as the data matrix)
# The third argument is the data matrix (a pandas dataframe or numpy array)
shap.dependence_plot(0, shap_values, X)
```



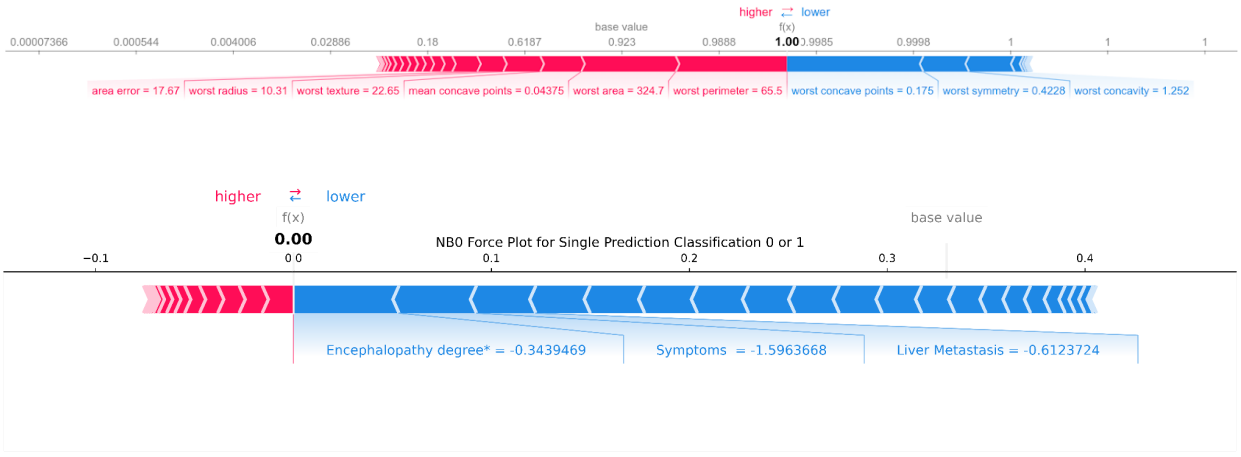
The dependence scatters plot shown above demonstrates the interaction effects between features, Age, and Education-Num. An interaction effect is present if this other feature and the feature we are plotting display as a distinct vertical pattern of coloring.

- Force plot
  - Visualizes SHAP values with an additive layout
  - May take **"shap_values"** as a parameter by typically prefers using **"explainer.expected_value"** as the parameter to generate the plot
  - If implementation does not work for model and Explainer object → refer to **source code**
  - **Issue:** using this plot type for multiple samples for a given model must be displayed in the Jupyter Notebook output cell for the SHAP code. In other cases, figures can be saved using matplotlib
    - **shap.force_plot(explainer.expected_value, shap_values, feature_names, show=True)**
    - **shap.force_plot(explainer.expected_value[0], shap_values[0], feature_names,show=True)**
    - **To save figure:**
      - **shap.force_plot(shap_values[random_single_predict], X, feature_names=feature_names, matplotlib=True, show=False)**

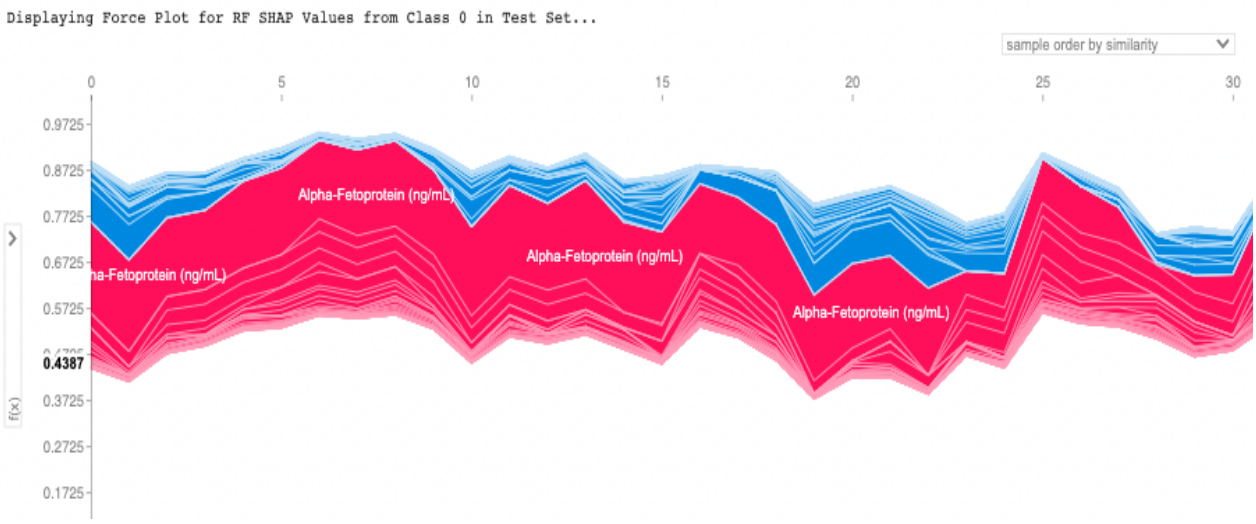
        plt.title(f'{abbrev}{cvCount} Force Plot for Single Prediction Classification 0 or 1')

        plt.savefig(f'{save_path}/ForcePlots/{abbrev}{str(cvCount)}_singlePredictFP.png', dpi=600, bbox_inches='tight')

        plt.close()
  - **Further examples of color customization can be found at this link**



The output of the force plot depends on whether the user is looking at binary classification (probability) or log-odds using the actual model's predicted value. Models dealing with binary classification will output the model prediction as either Class 0 or Class 1 and display each feature's impact. Positive (red) with a longer 'force' is considered to have a greater, positive influence on the model's prediction. Negative (blue) with longer 'force' is considered to have a greater, negative effect on the prediction (and vice versa).



The image above is taken from the actual output cell of the SHAP Notebook. This is a global explanation for the dataset and shap_values in Class 0 (this is also performed for Class 1 as well). Numbers at the top indicate the sample number (ex: Row 10) whereas the y-axis represents the SHAP values of the feature. This is an interactive plot and only can be accessed when displayed in the output cell of the notebook.