SHAP and LIME Python Libraries: Part 1 – Great Explainers, with Pros and Cons to Both

blog.dominodatalab.com/shap-lime-python-libraries-part-1-great-explainers-pros-cons

Joshua Poduska

This blog post provides a brief technical introduction to the SHAP and LIME Python libraries, followed by code and output to highlight a few pros and cons of each.

Introduction

Model explainability is a priority in today's data science community. As data scientists, we want to prevent model bias and help decision makers understand how to use our models in the right way. Data science leaders and executives are mindful of existing and upcoming legislation that requires models to provide evidence of how they work and how they avoid mistakes (e.g., SR 11-7 and The FUTURE of AI Act).

Part 1 in this blog post provides a brief technical introduction to the SHAP and LIME Python libraries, followed by code and output to highlight a few pros and cons of each. Part 2 will explore these libraries in more detail by applying them to a variety of Python models. The goal of these posts is to familiarize readers with how to use these libraries in practice and how to interpret their output, helping you leverage model explanations in your own work.

SHAP and LIME

SHAP and LIME are both popular Python libraries for model explainability. SHAP (SHapley Additive exPlanation) leverages the idea of Shapley values for model feature influence scoring. The technical definition of a Shapley value is the "average marginal contribution of a feature value over all possible coalitions." In other words, Shapley values consider all possible predictions for an instance using all possible combinations of inputs. Because of this exhaustive approach, SHAP can guarantee properties like consistency and local accuracy. LIME (Local Interpretable Model-agnostic Explanations) builds sparse linear models around each prediction to explain how the black box model works in that local vicinity. In their NIPS paper, the authors of SHAP show that Shapley values provide the only guarantee of accuracy and consistency and that LIME is actually a subset of SHAP but lacks the same properties. For further study, I found the GitHub sites SHAP GitHub and LIME GitHub helpful resources:

So why would anyone ever use LIME? Simply put, LIME is fast, while Shapley values take a long time to compute. For you statisticians out there, this situation reminds me somewhat of Fisher's Exact Test versus a Chi-Squared Test on contingency tables. Fisher's Exact Test provides the highest accuracy possible because it considers all possible outcomes, but it takes forever to run on large tables. This makes the Chi-Squared Test, a distribution-based approximation, a nice alternative.

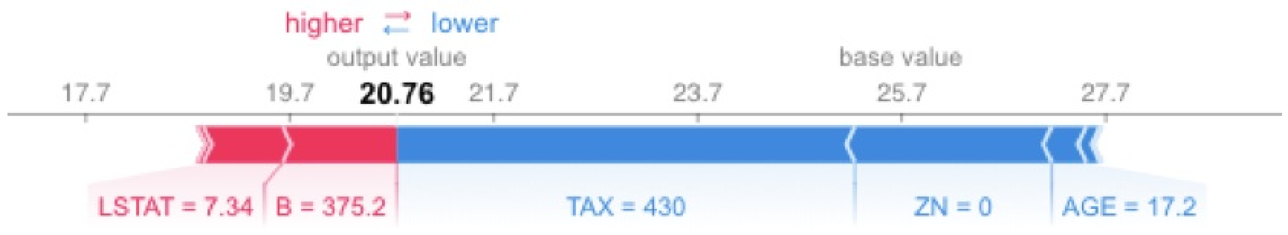
The SHAP Python library helps with this compute problem by using approximations and optimizations to greatly speed things up while seeking to keep the nice Shapley properties. When you use a model with a SHAP optimization, things run very fast and the output is accurate and reliable. Unfortunately, SHAP is not optimized for all model types yet.

For example, SHAP has a tree explainer that runs fast on trees, such as gradient boosted trees from XGBoost and scikit-learn and random forests from sci-kit learn, but for a model like k-nearest neighbor, even on a very small dataset, it is prohibitively slow. Part 2 of this post will review a complete list of SHAP explainers. The code and comments below document this deficiency of the SHAP library on the Boston Housing dataset. This code is a subset of a Jupyter notebook I created to walk through examples of SHAP and LIME. The notebook is hosted on Domino's trial site. Click here to view, download, or run the notebook.

```
import pandas as pd

import sklearn

from sklearn.model_selection import train_test_split

import sklearn.ensemble

import numpy as np

import lime

import lime.lime_tabular

import shap

import xgboost as xgb

import matplotlib

import matplotlib.pyplot as plt

from mpl_toolkits.mplot3d import axes3d, Axes3D

import seaborn as sns

import time
% matplotlib inline

X,y = shap.datasets.boston()

X_train,X_test,y_train,y_test = train_test_split(X, y, test_size = 0.2 ,
random_state = 0 )

X,y = shap.datasets.boston()

X_train,X_test,y_train,y_test = train_test_split(X, y, test_size = 0.2 ,
random_state = 0 )

knn = sklearn.neighbors.KNeighborsRegressor()
knn.fit(X_train, y_train)

X_train_summary = shap.kmeans(X_train, 10 )

t0 = time.time()

explainerKNN = shap.KernelExplainer(knn.predict,X_train_summary)

shap_values_KNN_test = explainerKNN.shap_values(X_test)

t1 = time.time()

timeit = t1 - t0
```

```
▯ timeit▯▯
```

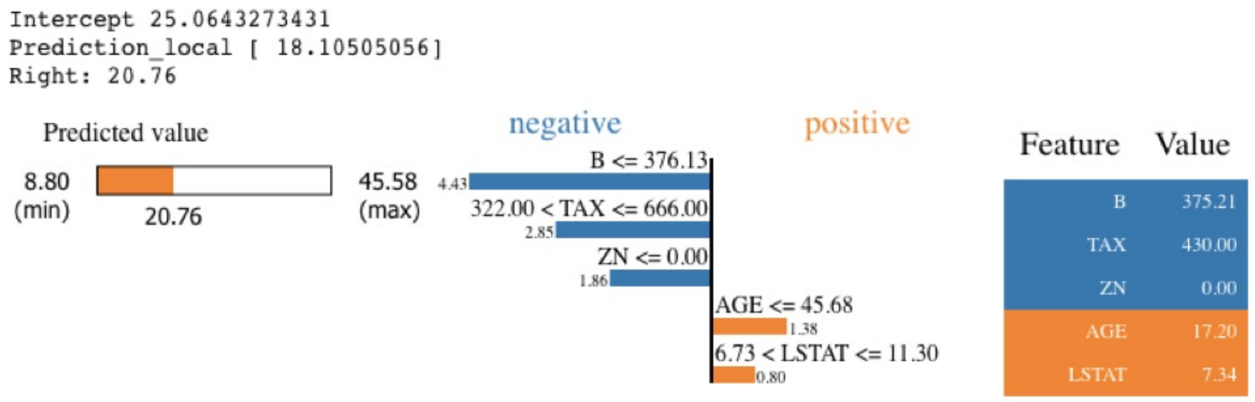
```
shap.force_plot(explainerKNN.expected_value, shap_values_KNN_test[j],
X_test.iloc[[j]])
```



Running SHAP on a knn model built on the Boston Housing dataset took over an hour, which is a tough pill to swallow. We can get that down to three minutes if we sacrifice some accuracy and reliability by summarizing the data first with a k-means algorithm. As an alternative approach, we could use LIME. LIME runs instantaneously with the same knn model and does not require summarizing with k-means. See the code and output below. Note that LIME's output is different than the SHAP output, especially for features AGE and B. With LIME not having the same accuracy and consistency properties as Shapley Values, and with SHAP using a k-means summary before calculating influence scores, it's tough to tell which comes closer to the correct answer.

```
exp  =  explainer.explain_instance(X_test.values[j], knn.predict,
num_features = 5 )▯
```

```
exp.show_in_notebook(show_table = True )
```



```
Intercept 25.0643273431
Prediction_local [ 18.10505056]
Right: 20.76
```

Feature	Value
B	375.21
TAX	430.00
ZN	0.00
AGE	17.20
LSTAT	7.34

While LIME provided a nice alternative in the knn model example, LIME is unfortunately not always able to save the day. It doesn't work out-of-the-box on all models. For example, LIME cannot handle the requirement of XGBoost to use xgb.DMatrix() on the input data. See below for one attempt to call LIME with the XGBoost model. There are potential hacks that could get LIME to work on this model, including creating your own prediction function, but the point is LIME doesn't automatically work with the XGBoost library.

```
xgb_model = xgb.train({ 'objective' : 'reg:linear' }, xgb.DMatrix(X_train,
label = y_train))
```

```
max_features = 'auto' , max_leaf_nodes = None ,
```

```
min_impurity_decrease = 0.0 , min_impurity_split = None ,
```

```
min_samples_leaf = 1 , min_samples_split = 2 ,
```

```
min_weight_fraction_leaf = 0.0 , n_estimators = 10 , n_jobs = 1 ,
oob_score = False , random_state = None , verbose = 0 , warm_start = False )
```

```
explainer = lime.lime_tabular.LimeTabularExplainer(X_train.values,
```

```
feature_names = X_train.columns.values.tolist(),
```

```
class_names = [ 'price' ],
```

```
categorical_features = categorical_features,
```

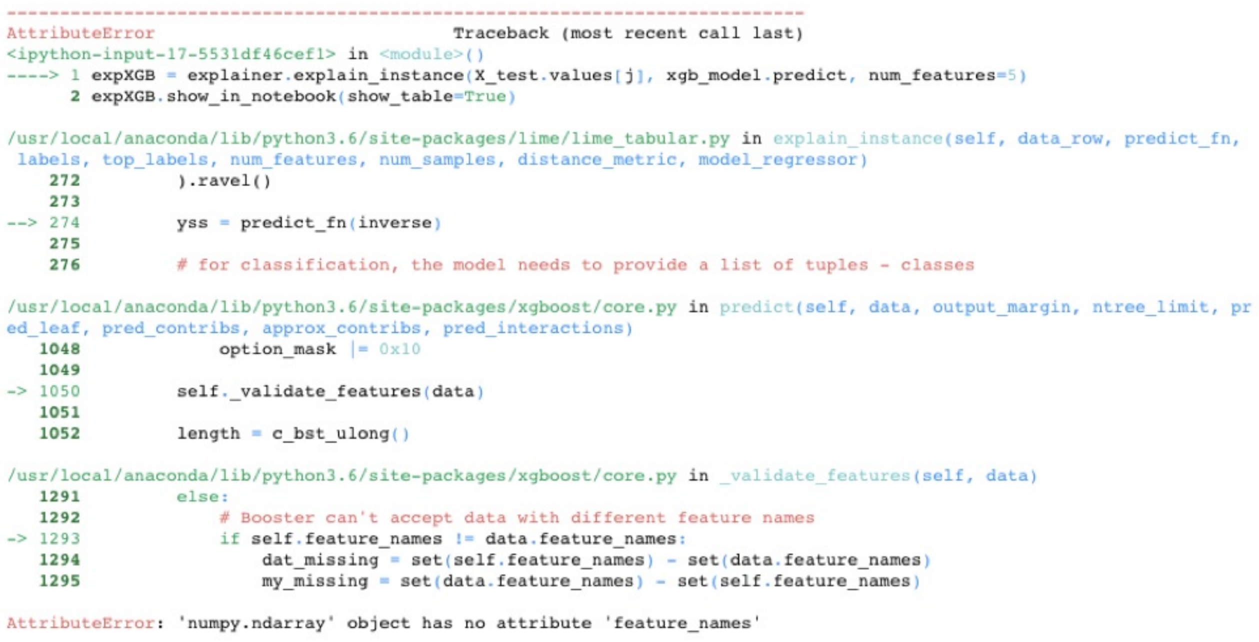
```
verbose = True ,
```

```
mode = 'regression' )
```

```
xgb_model.predict(xgb.DMatrix(X_test.iloc[[j]]))
```

```
expXGB = explainer.explain_instance(X_test.values[j], xgb_model.predict,
num_features = 5 )
```

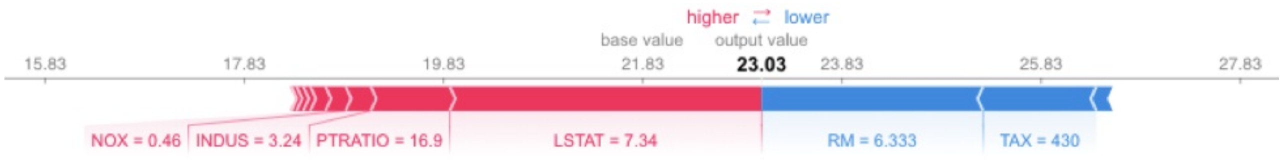
```
expXGB.show_in_notebook(show_table = True )
```



```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-17-5531df46cef1> in <module>()
----> 1 expXGB = explainer.explain_instance(X_test.values[j], xgb_model.predict, num_features=5)
      2 expXGB.show_in_notebook(show_table=True)

/usr/local/anaconda/lib/python3.6/site-packages/lime/lime_tabular.py in explain_instance(self, data_row, predict_fn,
 labels, top_labels, num_features, num_samples, distance_metric, model_regressor)
    272         ).ravel()
    273
--> 274         yss = predict_fn(inverse)
    275
    276         # for classification, the model needs to provide a list of tuples - classes

/usr/local/anaconda/lib/python3.6/site-packages/xgboost/core.py in predict(self, data, output_margin, ntree_limit, pr
ed_leaf, pred_contribs, approx_contribs, pred_interactions)
   1048             option_mask |= 0x10
   1049
-> 1050         self._validate_features(data)
   1051
   1052         length = c_bst_ulong()

/usr/local/anaconda/lib/python3.6/site-packages/xgboost/core.py in _validate_features(self, data)
   1291         else:
   1292             # Booster can't accept data with different feature names
-> 1293             if self.feature_names != data.feature_names:
   1294                 dat_missing = set(self.feature_names) - set(data.feature_names)
   1295                 my_missing = set(data.feature_names) - set(self.feature_names)

AttributeError: 'numpy.ndarray' object has no attribute 'feature_names'
```

On the other hand, SHAP is optimized for XGBoost and provides fast, reliable results. The following code runs very fast. It uses the TreeExplainer from the SHAP library, which is optimized to trace through the XGBoost tree to find the Shapley value estimates.

```
explainerXGB = shap.TreeExplainer(xgb_model)

shap_values_XGB_test = explainerXGB.shap_values(X_test)

shap.force_plot(explainerXGB.expected_value, shap_values_XGB_test[j],
X_test.iloc[[j]])
```



Conclusion

Hopefully, this post has given you a few pointers on how to choose between SHAP and LIME and brought to light some of the limitations of each. While both approaches have their strengths and limitations, I personally prefer to use SHAP when I can and rely on LIME when SHAP's compute costs are too high. Stay tuned for my next post on this topic, which will provide multiple examples of how to use these libraries on a variety of models and also show how to interpret their output.

SHAP and LIME Python Libraries: Part 2 – Using SHAP and LIME

blog.dominodatalab.com/shap-lime-python-libraries-part-2-using-shap-lime

Joshua Poduska

This blog post provides insights on how to use the SHAP and LIME Python libraries in practice and how to interpret their output, helping readers prepare to produce model explanations in their own work.

Introduction

Part 1 of this blog post provides a brief technical introduction to the SHAP and LIME Python libraries, including code and output to highlight a few pros and cons of each library. In Part 2 we explore these libraries in more detail by applying them to a variety of Python models. The goal of Part 2 is to familiarize readers with how to use the libraries in practice and how to interpret their output, helping them prepare to produce model explanations in their own work. We do this with side-by-side code comparisons of SHAP and LIME for four common Python models.

The code below is a subset of a Jupyter notebook I created to walk through examples of SHAP and LIME. The notebook is hosted on Domino's trial site. Click here to view, download or run the notebook on an environment with all the required dependencies already installed and on AWS hardware provided free of charge by Domino. The easiest way to get started is to click the green Launch Notebook button after clicking on the link above.

SHAP and LIME Individual Prediction Explainers

First, we load the required Python libraries.

```
import pandas as pd

import numpy as np

import sklearn

import xgboost as xgb

import sklearn.ensemble

from sklearn.model_selection  import train_test_split

import lime

import lime.lime_tabular

import shap

import time

import os

import matplotlib.pyplot as plt

import seaborn as sns
```

Next, we load the Boston Housing data, the same dataset we used in Part 1.

```
X,y  = shap.datasets.boston() X_train,X_test,y_train,y_test  =
train_test_split(X, y, test_size = 0.2 , random_state = 0 )
```

Let’s build the models that we’ll use to test SHAP and LIME. We are going to use four models: two gradient boosted tree models, a random forest model and a nearest neighbor model.

```
xgb_model  = xgb.train({ 'objective' : 'reg:linear' }, xgb.DMatrix(X_train,
label = y_train))

sk_xgb  = sklearn.ensemble.GradientBoostingRegressor()

sk_xgb.fit(X_train, y_train)

 rf  = sklearn.ensemble.RandomForestRegressor() rf.fit(X_train, y_train)

knn  = sklearn.neighbors.KNeighborsRegressor() knn.fit(X_train, y_train)
```

The SHAP Python library has the following explainers available: deep (a fast, but approximate, algorithm to compute SHAP values for deep learning models based on the DeepLIFT algorithm); gradient (combines ideas from Integrated Gradients, SHAP and SmoothGrad into a single expected value equation for deep learning models); kernel (a specially weighted local linear regression to estimate SHAP values for any model); linear (compute the exact SHAP values for a linear model with independent features); tree (a fast and exact algorithm to compute SHAP values for trees and ensembles of trees) and sampling (computes SHAP values under the assumption of feature independence — a good alternative to kernel when you want to use a large background set). The first three of our models can use the tree explainer.

```
explainerXGB = shap.TreeExplainer(xgb_model)
shap_values_XGB_test = explainerXGB.shap_values(X_test)
shap_values_XGB_train = explainerXGB.shap_values(X_train)
explainerSKGBT = shap.TreeExplainer(sk_xgb)
shap_values_SKGBT_test = explainerSKGBT.shap_values(X_test)
shap_values_SKGBT_train = explainerSKGBT.shap_values(X_train)
explainer
RF = shap.TreeExplainer(rf)
shap_values_RF_test = explainerRF.shap_values(X_test)
shap_values_RF_train = explainerRF.shap_values(X_train)
```

As explained in Part 1, the nearest neighbor model does not have an optimized SHAP explainer so we must use the kernel explainer, SHAP's catch-all that works on any type of model. However, doing that takes over an hour, even on the small Boston Housing dataset. The authors of SHAP recommend summarizing the data first with a K-Means procedure, as shown below.

```
X_train_summary = shap.kmeans(X_train, 10 )
t0 = time.time()
explainerKNN = shap.KernelExplainer(knn.predict, X_train_summary)
shap_values_KNN_test = explainerKNN.shap_values(X_test)
shap_values_KNN_train = explainerKNN.shap_values(X_train)
t1 = time.time()
timeit = t1 - t0
timeit
```

Now that we have the models and the SHAP explainers built, I find it helpful to put all the SHAP values into dataframes for later use.

```
df_shap_XGB_test  =  pd.DataFrame(shap_values_XGB_test,
columns = X_test.columns.values)

df_shap_XGB_train  =  pd.DataFrame(shap_values_XGB_train,
columns = X_train.columns.values)

df_shap_SKGBT_test  =  pd.DataFrame(shap_values_SKGBT_test,
columns = X_test.columns.values)

df_shap_SKGBT_train  =  pd.DataFrame(shap_values_SKGBT_train,
columns = X_train.columns.values)

df_shap_RF_test  =  pd.DataFrame(shap_values_RF_test,
columns = X_test.columns.values)

df_shap_RF_train  =  pd.DataFrame(shap_values_RF_train,
columns = X_train.columns.values)

df_shap_KNN_test  =  pd.DataFrame(shap_values_KNN_test,
columns = X_test.columns.values)

df_shap_KNN_train  =  pd.DataFrame(shap_values_KNN_train,
columns = X_train.columns.values)
```

That concludes the necessary setup for the SHAP explainers. Setting up LIME explainers is quite a bit easier, with only one explainer that is then applied to each model individually.

```
categorical_features  =  np.argwhere(np.array([ len ( set (X_train.values[:,x]))
for  x  in  range (X_train.values.shape[ 1 ])]) < = 10 ).flatten()
explainer  =  lime.lime_tabular.LimeTabularExplainer(X_train.values,
feature_names = X_train.columns.values.tolist(),
class_names = [ 'price' ],
categorical_features = categorical_features,
verbose = True ,  mode = 'regression' )
```

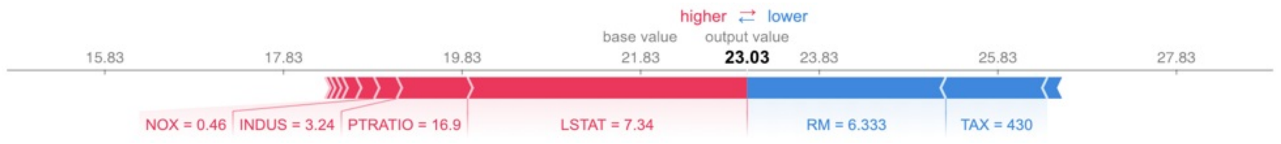
OK, now it’s time to start explaining predictions from these models. To keep it simple, I choose to explain the first record in the test set for each model using SHAP and LIME.

```
j  =  0
shap.initjs()
```

XGBoost SHAP

Notice the use of the dataframes we created earlier. The plot below is called a force plot. It shows features contributing to push the prediction from the base value. The base value is the average model output over the training dataset we passed. Features pushing the prediction higher are shown in red. Features pushing it lower appear in blue. The record we are testing from the test set has a higher than average predicted value at 23.03 compared to 21.83. LSTAT (percent lower status of the population) is 7.34 for this record. This pushes the predicted value higher. Unfortunately, the force plot does not tell us exactly how much higher, nor does it tell us how 7.34 compares to the other values of LSTAT. You can get this information from the dataframe of SHAP values, but it is not displayed in the standard output.

```
shap.force_plot(explainerXGB.expected_value, shap_values_XGB_test[j],
X_test.iloc[[j]]) 
```



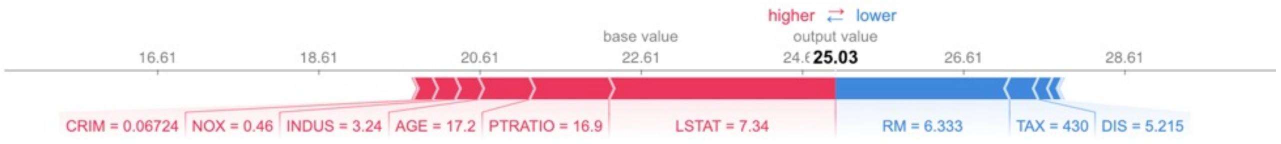
XGBoost LIME

Out-of-the-box LIME cannot handle the requirement of XGBoost to use xgb.DMatrix() on the input data, so the following code throws an error, and we will only use SHAP for the XGBoost library. Potential hacks, including creating your own prediction function, could get LIME to work on this model, but the point is that LIME doesn't automatically work with the XGBoost library.

```
expXGB  = explainer.explain_instance(X_test.values[j], xgb_model.predict,
num_features = 5 ) expXGB.show_in_notebook(show_table = True )
```

Scikit-learn GBT SHAP

```
shap.force_plot(explainerSKGBT.expected_value, shap_values_SKGBT_test[j],
X_test.iloc[[j]])
```

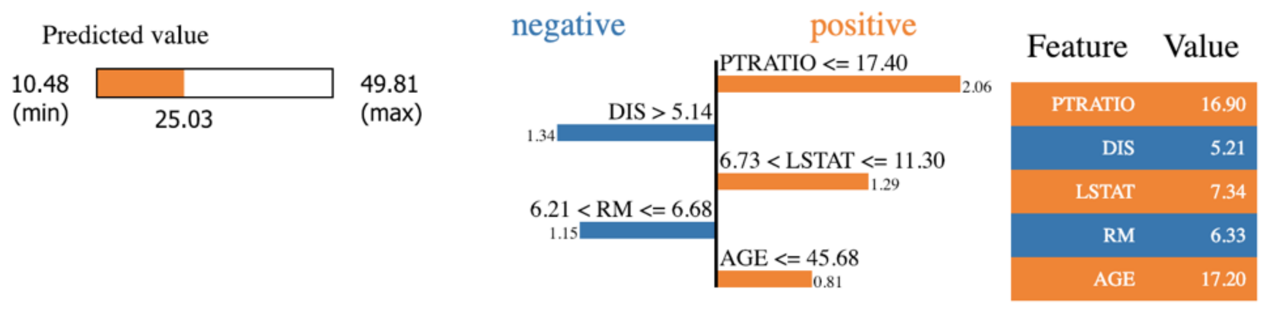


Scikit-learn GBT LIME

LIME works on the Scikit-learn implementation of GBTs. LIME's output provides a bit more detail than that of SHAP as it specifies a range of feature values that are causing that feature to have its influence. For example, we know that PTRATIO had a positive influence on this predicted house price because its value was below 17.4. SHAP does not provide this information. However, LIME's feature importance differs from SHAP's. Since SHAP has a more solid theoretical foundation (see Part 1 of this blog), most people tend to trust SHAP if LIME and SHAP disagree, especially with the tree and linear SHAP explainers.

```
expSKGBT  = explainer.explain_instance(X_test.values[j], sk_xgb.predict,
num_features = 5 )

expSKGBT.show_in_notebook(show_table = True )
```

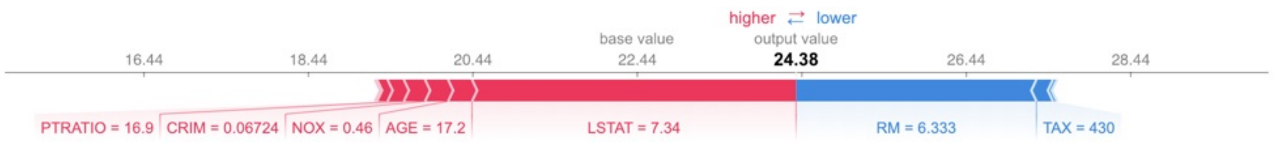
```
Intercept 22.9447425102
Prediction_local [ 24.61004276]
Right: 25.029662837
```



Feature	Value
PTRATIO	16.90
DIS	5.21
LSTAT	7.34
RM	6.33
AGE	17.20

Random Forest SHAP

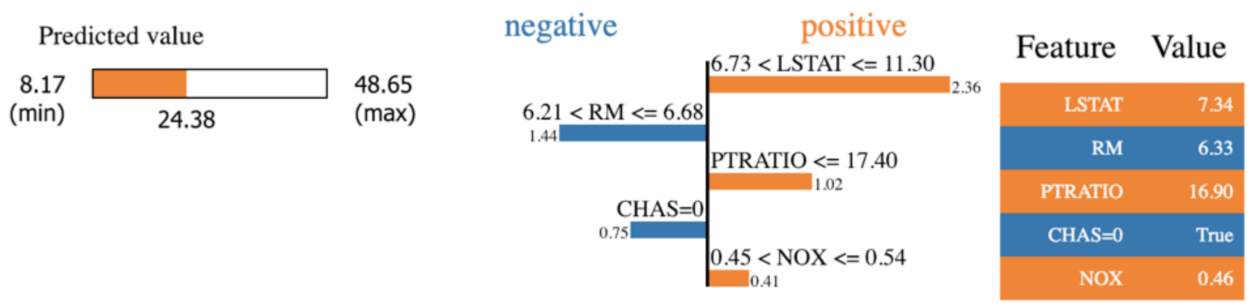
```
shap.force_plot(explainerRF.expected_value, shap_values_RF_test[j],
X_test.iloc[[j]])
```



Random Forest LIME

```
exp = explainer.explain_instance(X_test.values[j], rf.predict,
num_features = 5 )
exp.show_in_notebook(show_table = True )
```

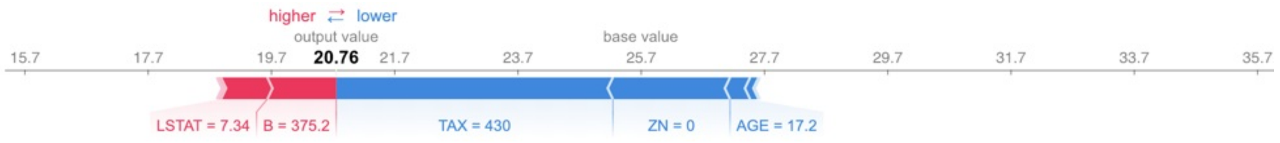
```
Intercept 23.527734324
Prediction_local [ 25.11970367]
Right: 24.38
```



Feature	Value
LSTAT	7.34
RM	6.33
PTRATIO	16.90
CHAS=0	True
NOX	0.46

KNN SHAP

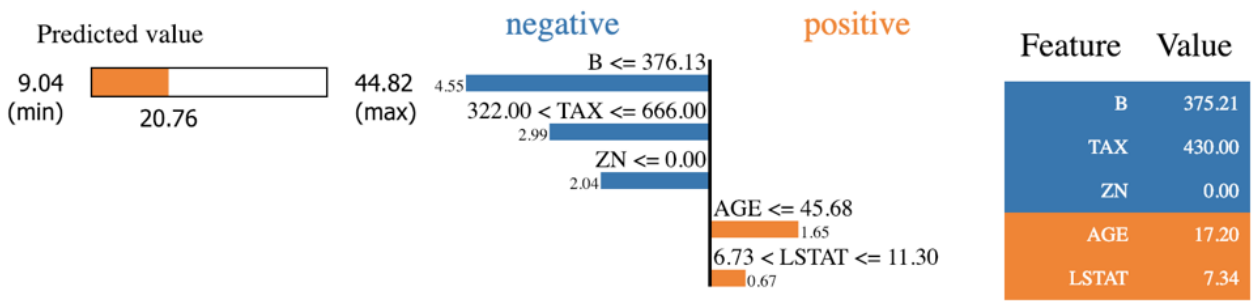
```
shap.force_plot(explainerKNN.expected_value, shap_values_KNN_test[j],
X_test.iloc[[j]])
```



KNN LIME

```
exp  =  explainer.explain_instance(X_test.values[j], knn.predict,
num_features  =  5  ) exp.show_in_notebook(show_table  =  True  )
```

```
Intercept 25.212409696
Prediction_local [ 17.94694087]
Right: 20.76
```



Feature	Value
B	375.21
TAX	430.00
ZN	0.00
AGE	17.20
LSTAT	7.34

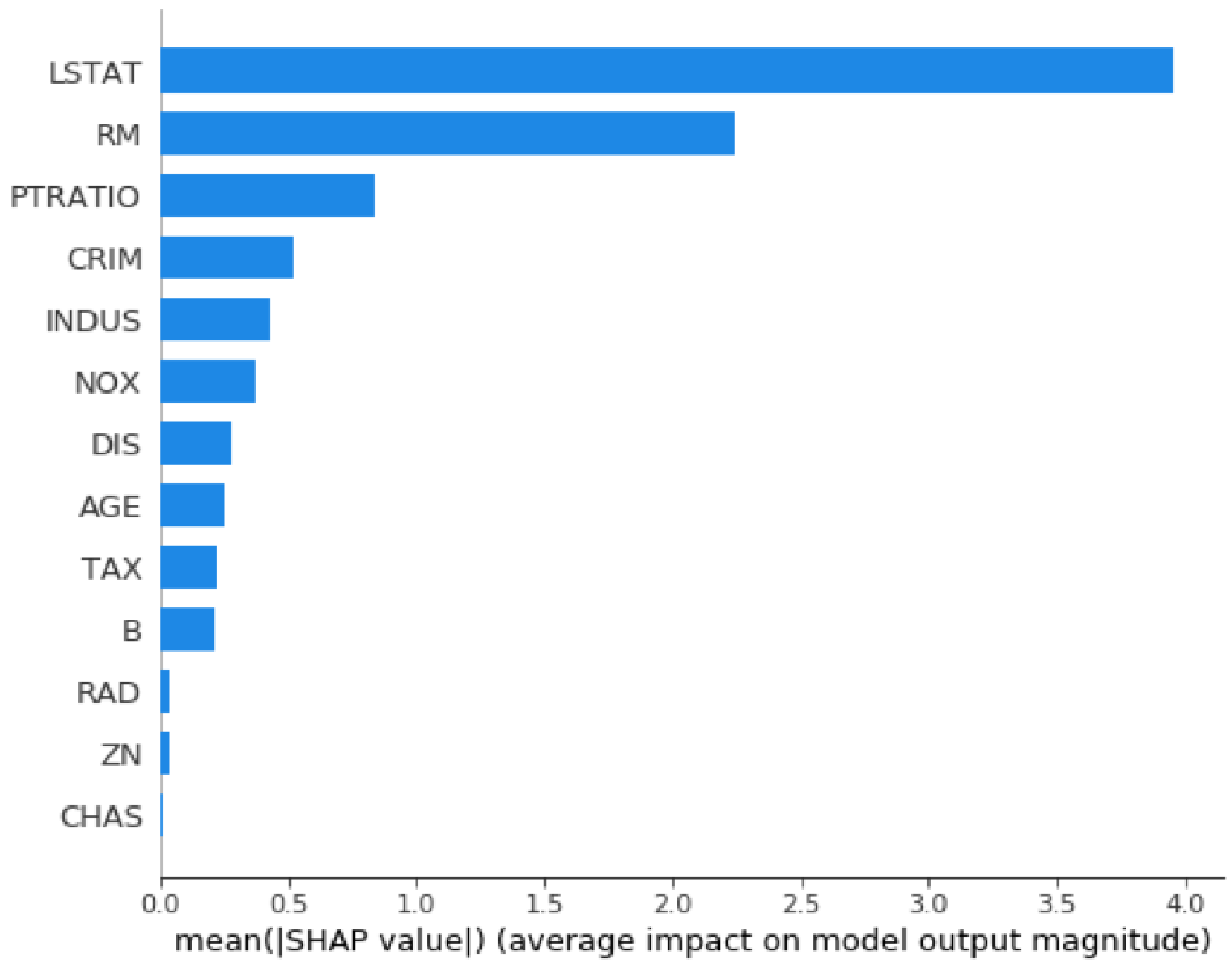
Explainability on a Macro Level with SHAP

The whole idea behind both SHAP and LIME is to provide model interpretability. I find it useful to think of model interpretability in two classes — local and global. Local interpretability of models consists of providing detailed explanations for why *an individual prediction* was made. This helps decision makers trust the model and know how to integrate its recommendations with other decision factors. Global interpretability of models entails seeking to understand the *overall structure* of the model. This is much bigger (and much harder) than explaining a single prediction since it involves making statements about how the model works in general, not just on one prediction. Global interpretability is generally more important to executive sponsors needing to understand the model at a high level, auditors looking to validate model decisions in aggregate, and scientists wanting to verify that the model matches their theoretical understanding of the system being studied.

The graphs in the previous section are examples of local interpretability. While LIME does not offer any graphs for global interpretability, SHAP does. Let's explore a few of these graphs. I have chosen to use the first model, the one from the XGBoost library, for these graphical examples.

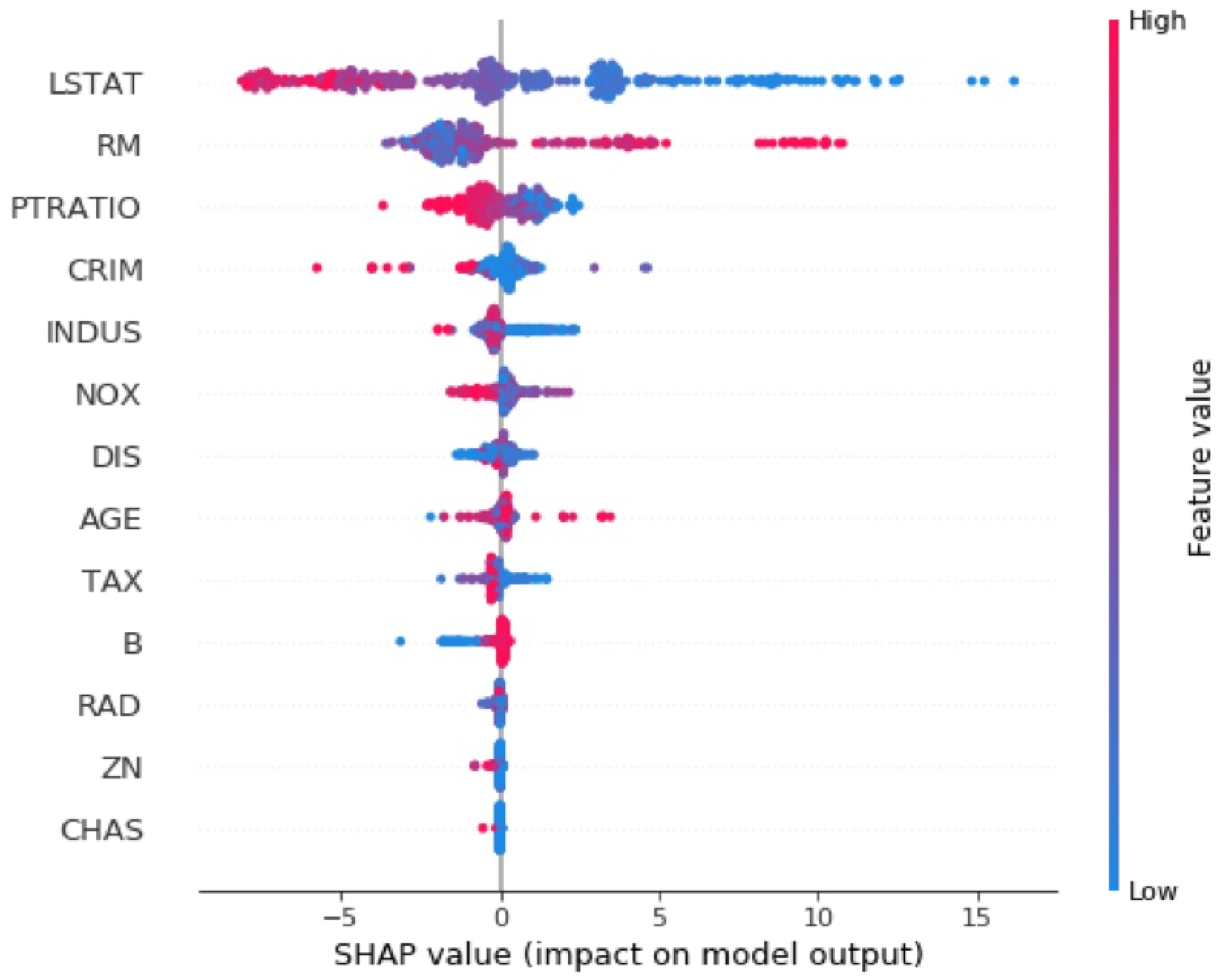
Variable importance graphs are useful tools for understanding the model in a global sense. SHAP provides a theoretically sound method for evaluating variable importance. This is important, given the debate over which of the traditional methods of calculating variable importance is correct and that those methods do not always agree.

```
shap.summary_plot(shap_values_XGB_train, X_train, plot_type  =  "bar"  )  
```



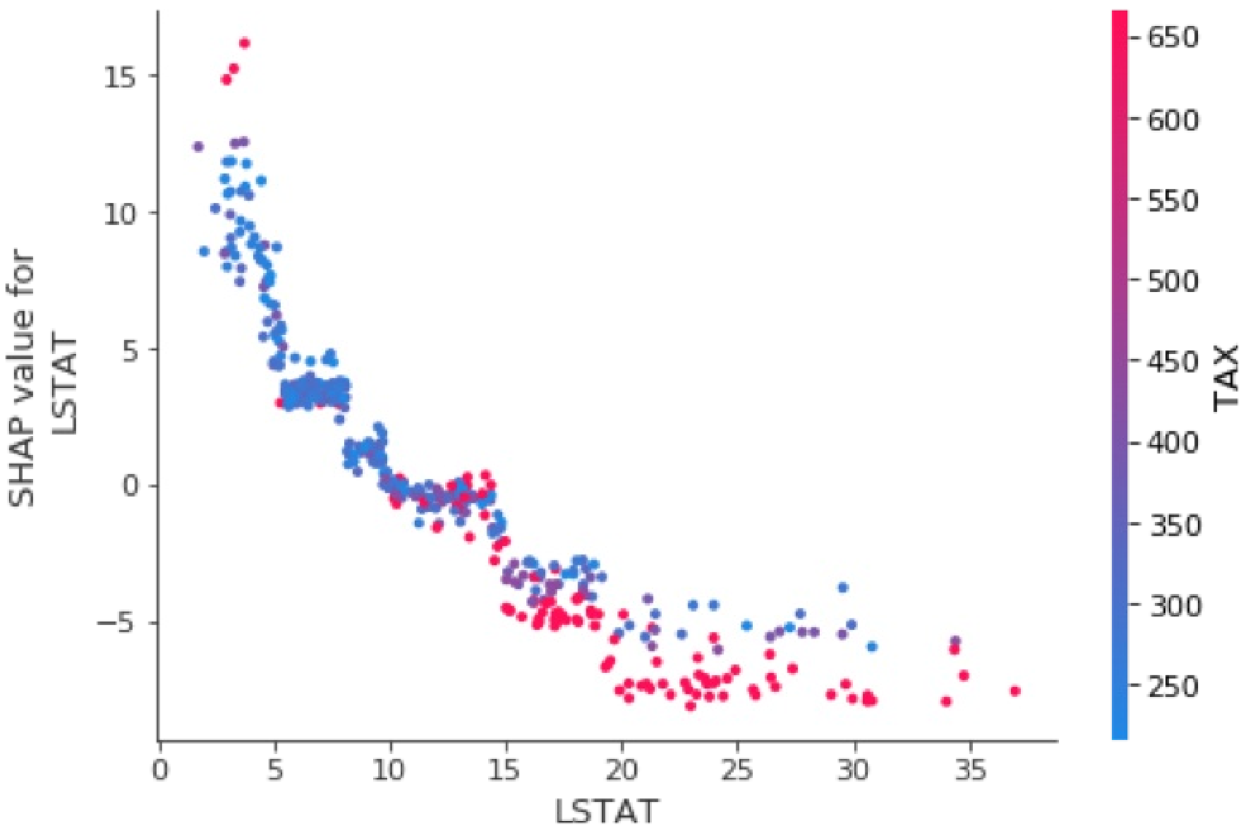
Similar to a variable importance plot, SHAP also offers a summary plot showing the SHAP values for every instance from the training dataset. This can lead to a better understanding of overall patterns and allow discovery of pockets of prediction outliers.

```
shap.summary_plot(shap_values_XGB_train, X_train)
```



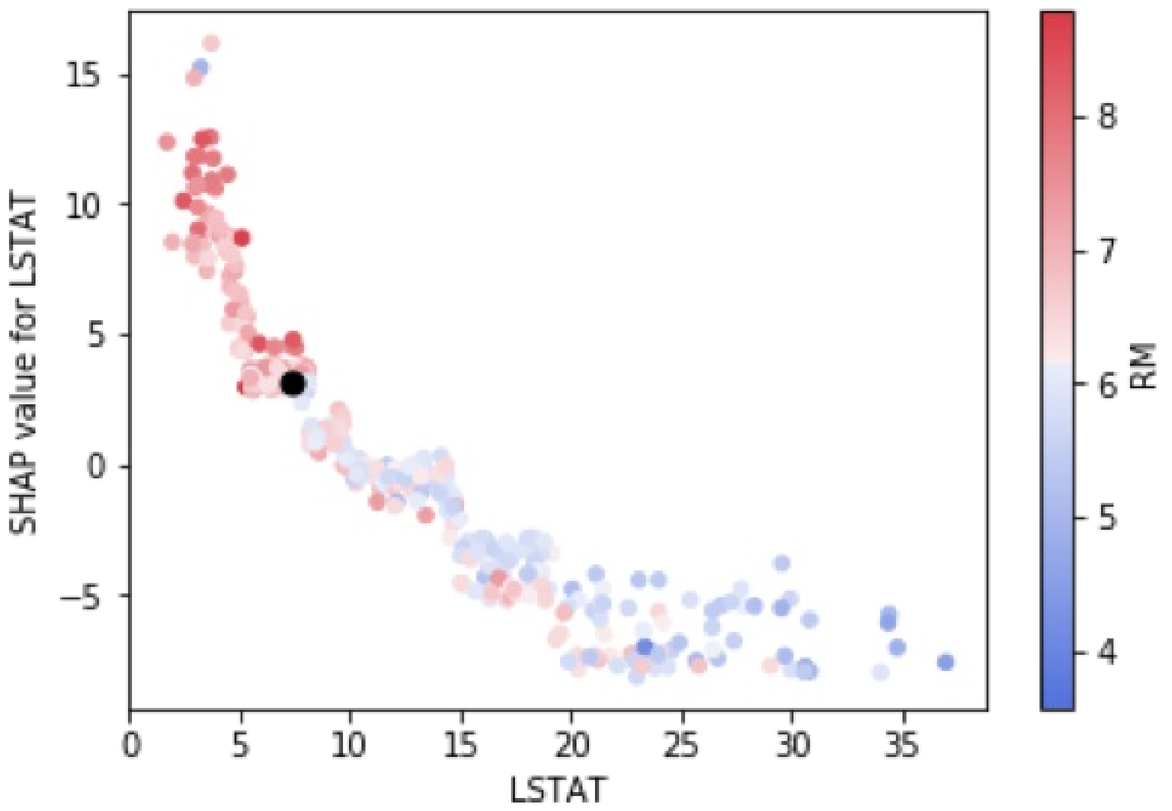
Variable influence or dependency plots have long been a favorite of statisticians for model interpretability. SHAP provides these as well, and I find them quite useful.

```
shp_plt = shap.dependence_plot( "LSTAT" , shap_values_XGB_train, X_train)
```

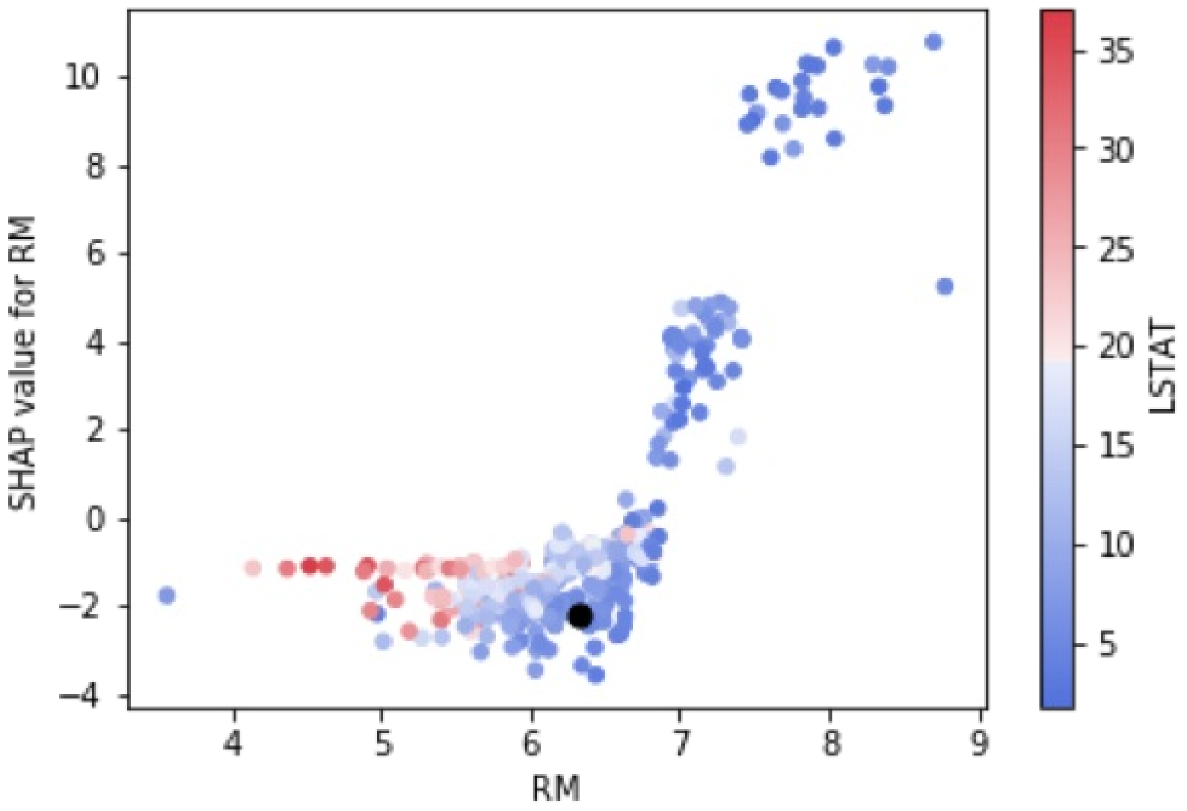


I like these so much, I decided to customize them a bit using matplotlib and seaborn to allow two improvements. First, I highlighted the jth instance with a black dot so we can combine the best of global and local interpretability into one graph. Second, I allowed flexibility with the choice of color-by-variable.

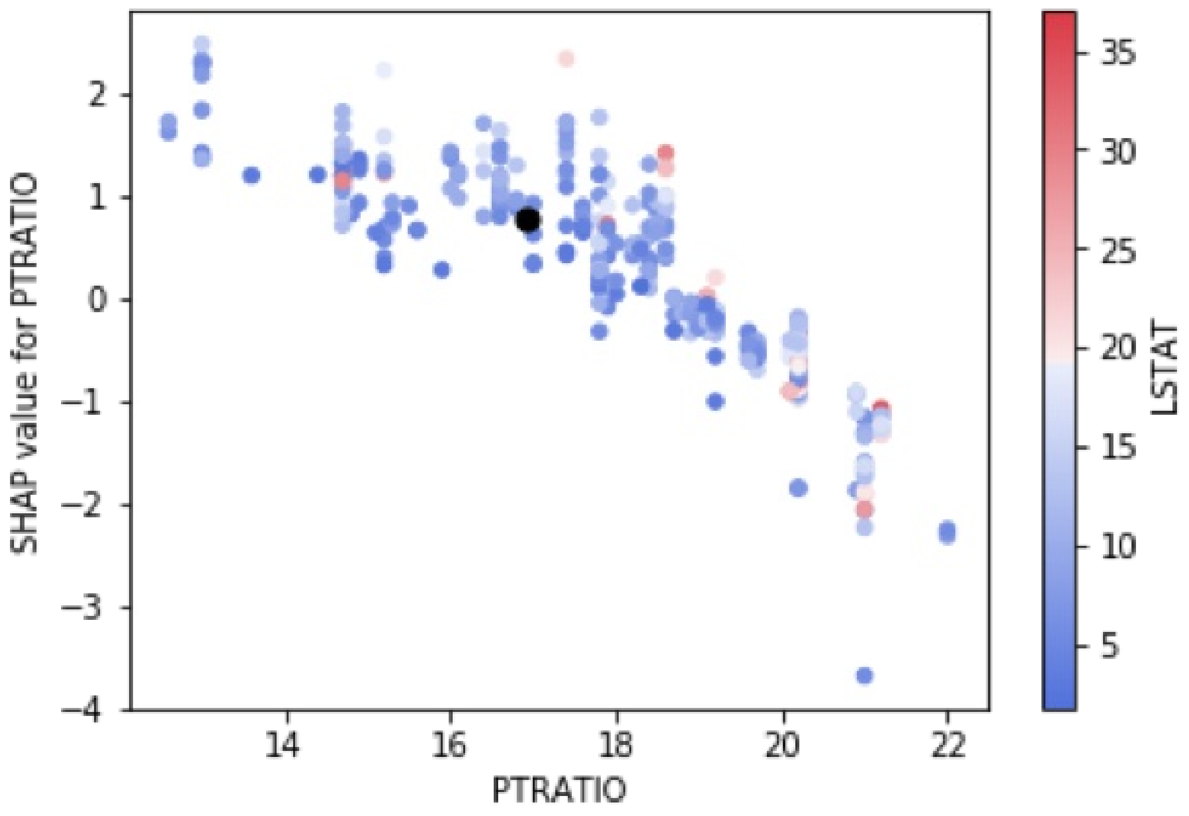
```
def dep_plt(col, color_by, base_actual_df, base_shap_df, overlay_x, overlay_y):

cmap = sns.diverging_palette( 260 , 10 , sep = 1 , as_cmap = True )

f, ax = plt.subplots()

points = ax.scatter(base_actual_df[col], base_shap_df[col],
c = base_actual_df[color_by], s = 20 , cmap = cmap)

f.colorbar(points).set_label(color_by)

ax.scatter(overlay_x, overlay_y, color = 'black' , s = 50 )

plt.xlabel(col)

plt.ylabel( "SHAP value for " + col)

plt.show()

imp_cols =
df_shap_XGB_train. abs ().mean().sort_values(ascending = False ).index.tolist()


for i in range ( 0 , len (imp_cols)):

if i = = 0 : 

dep_plt(imp_cols[i], 

imp_cols[i + 1 ], 

X_train, 

df_shap_XGB_train,

X_test.iloc[j,:][imp_cols[i]], 

df_shap_XGB_test.iloc[j,:][imp_cols[i]])

if (i &gt;; 0 ) and (i &lt;; 3 ) : 

dep_plt(imp_cols[i], 

imp_cols[ 0 ],  X_train, 

df_shap_XGB_train,

X_test.iloc[j,:][imp_cols[i]], 

df_shap_XGB_test.iloc[j,:][imp_cols[i]])
```



SHAP value for LSTAT
LSTAT
RM



SHAP value for RM
RM
LSTAT



Model explainability remains top of mind for many data scientists and data science leaders today. SHAP and LIME are solid libraries for helping provide these explanations, both on a local and a global level. The need to explain black box models will only increase as time goes on. I believe that in the not too distant future we will find that model explainability combined with model sensitivity/stress testing will become a standard part of data science work and that it will end up owning its own step in most data science life cycles.