Chapter13-Interpretability

2023 年 11 月 14 日

1 Interpretability

This chapter is dedicated to the techniques that help understand the way models process inputs into outputs. We recommend to read a recent book Molnar (2019) which is entirely devoted to this topic and a less technical reference Hall and Gill (2019). We will mainly be *factor-investing orientated* in this chapter and discuss examples with a financial dataset.

Quantitative tools that aim for interpretability of ML models are required to satisfy two simple conditions:

1. That they provide information about the model.
2. That they are highly comprehensible.

Often, these tools generate graphical outputs which are easy to read and yield immediate conclusions.

In attempts to white-box complex machine learning models, one dichotomy stands out:

- **Global models** seek to determine the relative role of features in the construction of the predictions once the model has been trained. This is done at the global level, so that the patterns that are shown in the interpretation hold on average over the *whole training set*.
- **Local models** aim to characterize how the model behaves around *one particular instance* by considering small variations around this instance. The way these variations are processed by the original model allows to simplify it by approximating it, e.g., in a linear fashion. This approximation can for example determine the sign and magnitude of the impact of each relevant feature in the vicinity of the original instance.

Molnar (2019) proposes another classification of interpretability solutions by splitting interpretations that depend on one particular model (e.g., linear regression or decision tree) versus the interpretations that can be obtained for any kind of model. In the sequel, we present the methods according to the global versus local dichotomy.

1.1 Global Interpretations

1.1.1 Simple Models as Surrogates

Let us start with the simplest example of all. In a linear model,

$$y_i = \alpha + \sum_{k=1}^{K} \beta_k x_i^k + \epsilon_i$$

The following elements are usually extracted from the estimation of the β_k:

- The R^2, which appreciates the **global fit** of the model (possibly penalized to prevent overfitting with many regressors). The R^2 is usually computed in-sample;
- The sign of the estimates $\hat{\beta}_k$, which indicates the direction of the impact of each **feature** x^k on y;
- The t-statistics $t_{\hat{\beta}_k}$, which evaluate the **magnitude** of this impact: large statistics in absolute value reveal prominent variables. Often, the t-statistics are translated into p-values which are computed under some suitable distributional assumptions.

The last two indicators are useful because they inform the user on which features matter the most and on the sign of the effect of each predictor. Most tools that aim to explain black boxes follow the same principles.

Decision trees, because they are easy to picture, are also great models for interpretability. Recently, Vidal et al. (2020) propose a method to reduce an ensemble of trees into a unique tree. The aim is to propose a simpler model that behaves exactly like the complex one.

More generally, it is an intuitive idea to resort to simple models to proxy more complex algorithms. One simple way to do so is to build so-called surrogate models. The process is simple:

1. Train the original model f on features $\mathbf{X}$ and labels $\mathbf{y}$;
2. Train a simpler model g to explain the predictions of the trained model $\hat{f}$ given the features $\mathbf{X}$:

$$\hat{f}(\mathbf{X}) = g(\mathbf{X}) + \textbf{error}$$

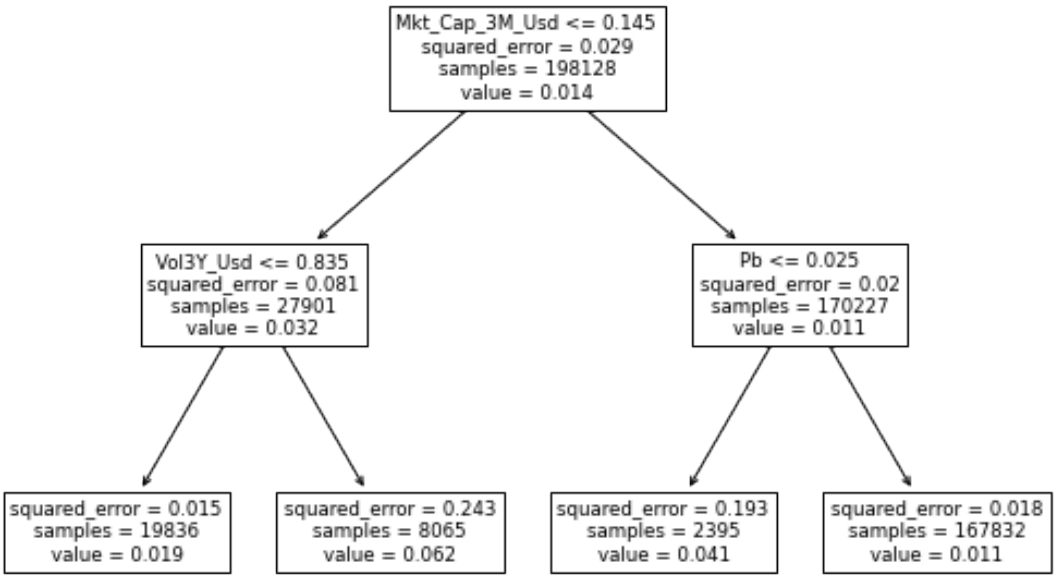
The estimated model $\hat{g}$ explains how the initial model $\hat{f}$ maps the features into the labels.

[ ]:
```
import pandas as pd
import numpy as np
import joblib

data_ml = pd.read_pickle('./data/data_ml.pkl')
separation_date = pd.to_datetime('2014-01-15')
```

```
training_sample = data_ml[data_ml['date'] < separation_date]
test_sample = data_ml[data_ml['date'] > separation_date]
features = data_ml.columns[2:95]

fit_RF = joblib.load('./models/fit_rf.pkl')
fit_RF
```

[ ]:
```
RandomForestRegressor(max_features=30, max_samples=10000, min_samples_split=250,
                      n_estimators=40)
```

[ ]:
```
from sklearn.tree import DecisionTreeRegressor, plot_tree
import matplotlib.pyplot as plt

dt = DecisionTreeRegressor(max_depth=2)
train_prediction = fit_RF.predict(training_sample[features])
dt.fit(training_sample[features], training_sample['R1M_Usd'])

plt.figure(figsize=(9, 6))
plot_tree(dt, feature_names=features)
plt.show()
```



The representation of the tree is quite different, compared to those seen in Chapter 6, but managed to do a proper job in capturing the main complexty of the model which it mimicks.

1.1.2 Variable Importance (Tree-based)

Decision trees are favorable for its interpretability and clear visual representation. They are easy to comprehend and is often associates with more sophisticated tools.

Indeed, both random forests and boosted trees fail to provide perfectly accurate accounts of what is happening inside the engine. In contrast, it is possible to compute the aggregate share (or importance) of each feature in the determination of the structure of the tree once it has been trained.

After training, it is possible to compute at each node n the gain $G(n)$ obtained by the subsequent split if there are any, i.e., if the node is not a terminal leaf. It is also easy to determine which variable is chosen to perform the split, hence we write $\mathcal{N}_k$ the set of nodes for which feature k is chosen for the partition. Then, the global importance of each feature is given by

$$I(k) = \sum_{n \in \mathcal{N}_k} G(n)$$

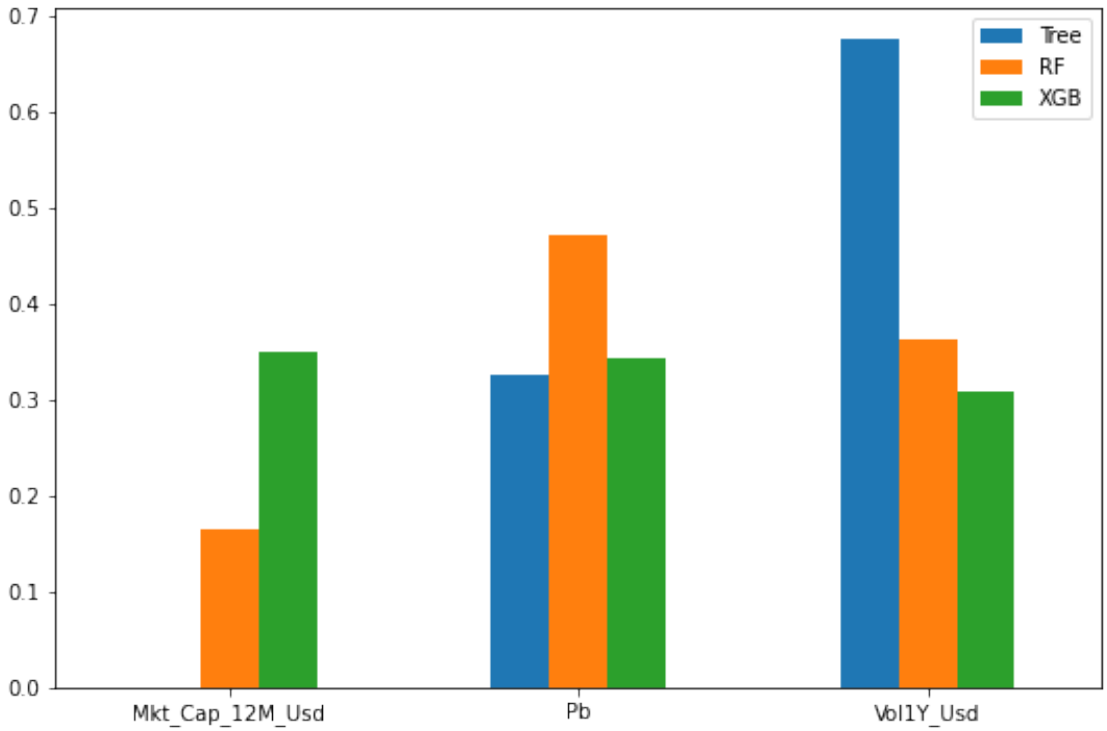
and it is often rescaled so that the sum of $I(k)$ across all k is equal to 1. In this case, $I(k)$ measures the relative contribution of feature k in the reduction of loss during the training. A variable with high importance will have a greater impact on predictions. Generally, these variables are those that are located close to the root of the tree.

Below we take a look at the results obtained from the tree-based models trained in Chapter 6. Notice that each fitted output has its own structure and importance vectors have different names.

[ ]:
```
fit_tree = joblib.load("./models/fit_tree.pkl")
fit_RF = joblib.load("./models/fit_rf.pkl")
fit_xgb = joblib.load("./models/fit_xgb.pkl")
```

[ ]:
```
tree_vi = pd.DataFrame(fit_tree.feature_importances_, index=features).T
rf_vi = pd.DataFrame(fit_RF.feature_importances_, index=features).T
xgb_vi = pd.DataFrame(fit_xgb.feature_importances_, index=features).T

vi_trees = pd.concat([tree_vi, rf_vi, xgb_vi], ignore_index=True)
vi_trees.index = ['Tree', 'RF', 'XGB']
```

```
# No global regularization needs since sklearn package has done this job
vi_trees_sub = vi_trees[['Mkt_Cap_12M_Usd','Pb','Vol1Y_Usd']].apply(lambda x: x␣
 ↪/ np.sum(x), axis=1).T

fig, ax = plt.subplots(figsize=(9, 6))
vi_trees_sub.plot.bar(ax=ax)
plt.xticks(rotation = 0)
plt.show()
```



Indeed, by construction, the simple tree model only has a small number of features with nonzero importance: we show 3 features above. In contrast, random forests and boosted trees are much more complex, and they give some importance to many predictors. For scale reasons, the normalization is performed *after* the subset of features is chosen. We preferred to limit the number of features shown on the graph for obvious readability concerns.

There are differences in the way the models rely on the features. For instance, the simple tree may give the most importance to volatility while random forests bet more on price-to-book ratio.

One defining property of random forests is that they give a chance to all features. Indeed, by

randomizing the choice of predictors, each individual exogenous variable has a shot at explaining the label. Along with boosted trees, the allocation of importance is more balanced across predictors, compared to the simple tree which puts most of its eggs in just a few baskets.

1.1.3 Variable Importance (agnostic)

We refer to the papers mentioned in the study by Fisher et al. (2019) to extend the feature importances to nontree-based methods. The aim is to quantify to what extent one feature contributes to the learning process.

One way to track the added value of one particular feature is to look at what happens if its values inside the training set are entirely *shuffled*. If the original feature plays an important role in the explanation of the dependent variable, then the shuffled version of the feature will lead to a much higher loss.

The baseline method to assess feature importance in the general case is the following:

1. Train the model on the original data and compute the associated loss l^*;
2. For each feature k, create a new training dataset in which the feature's values are randomly *permuted*. Then evaluate the loss l_k of the model based on this altered sample.
3. Rank the variable important of each feature computed as a difference $VI_k = l_k - l^*$ or a ratio $VI_k = l_k/l^*$.

Whether to compute the losses on the training set or the testing set is an open question. The above procedure is of course random and can be *repeated* so that the importances are averaged over several trials which improves the stability of the results.

Below, we implement this algorithm manually so to speak for the features appearing in Figure 13.2. We test this approach on ridge regressions and recycle the variables used in Chapter 5. We start by the first step: computing the loss on the original training sample.

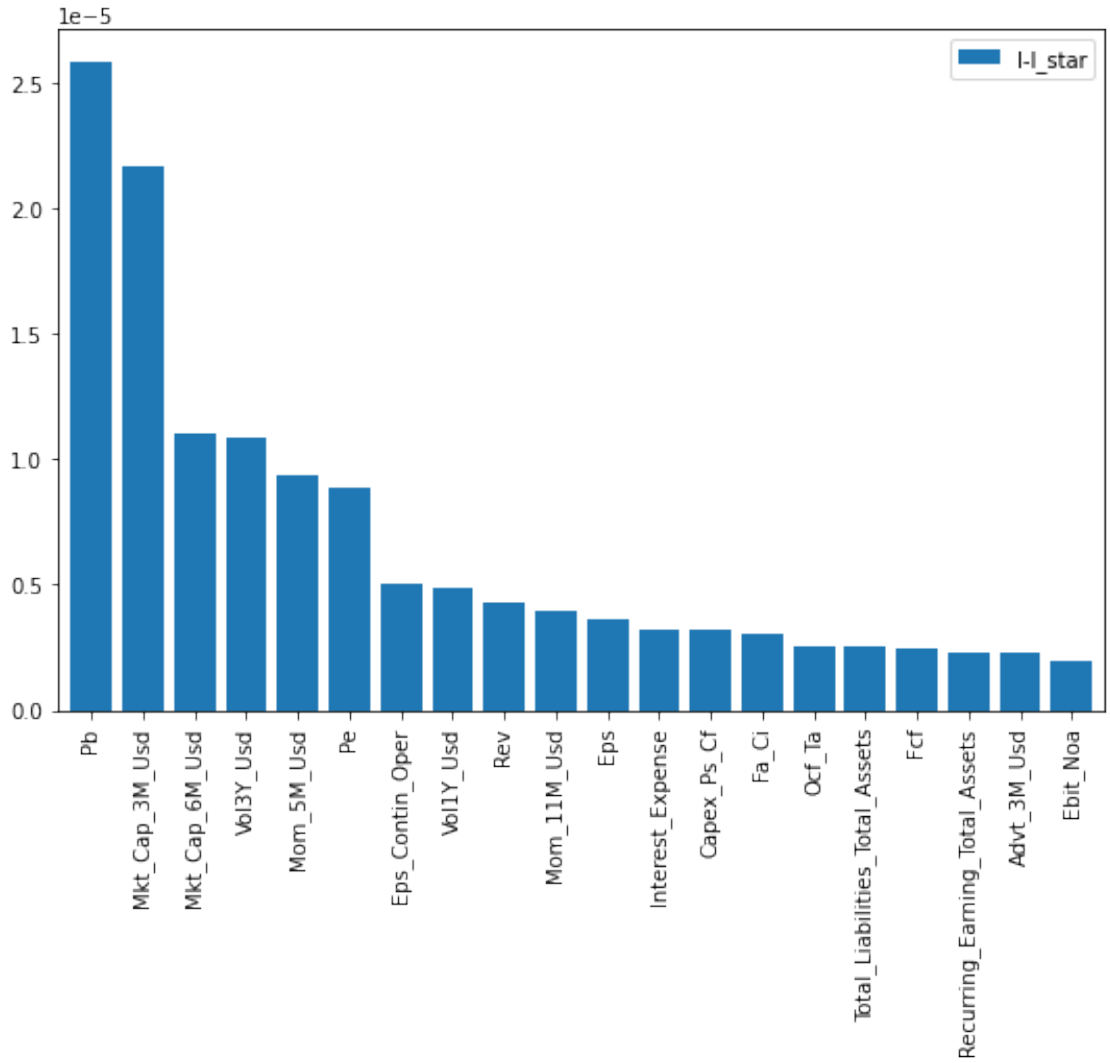
[ ]:
```
from sklearn.linear_model import ElasticNet
from sklearn.metrics import mean_squared_error

y_train = training_sample['R1M_Usd']
X_train = training_sample[features]

fit_ridge_0 = ElasticNet(alpha = 0.01, l1_ratio = 0.01).fit(X_train, y_train) #␣
 ↪For python, no penalization (i.e., alpha=0) is unrecommended!
l_star = mean_squared_error(fit_ridge_0.predict(X_train), y_train)
l_star
```

[ ]: 0.0289130000719163

Next, we evaluate the loss when each of the predictors has been sequentially shuffled. To reduce computation time, we only make one round of shuffling.

[ ]:
```
l = []
for feat_name in vi_trees.columns:
    tmp_data = X_train.copy()
    tmp_data[feat_name] = np.random.permutation(tmp_data[feat_name])
    # No re-training?
    tmp_l = mean_squared_error(fit_ridge_0.predict(tmp_data), y_train)
    l.append(tmp_l - l_star)
```

[ ]:
```
l_diff_df = pd.DataFrame(l, index=features, columns=['l-l_star']).
 ↪sort_values(by='l-l_star', ascending=False)
_, ax = plt.subplots(figsize=(9, 6))
l_diff_df[:20].plot.bar(ax=ax, width=0.8)
plt.show()
```



The resulting importances are in line with thoses of the tree-based models: the most prominent variables are volatility-based, market capitalization-based, and the price-to-book ratio; Note that in some cases the score can even be *negative*, which means that the predictions are more accurate than the baseline model when the values of the predictor are shuffled!

1.1.4 Partial Dependence Plot

Partial dependence plots (PDPs) aim at showing the relationship between the output of a model and the value of a feature.

Let us fix a feature k. We want to understand the **average impact** of k on the predictions of the trained model $\hat{f}$. In order to do so, we assume that the feature space is random and we split it in

two: k versus $-k$, which stands for all features except for k. The partial dependence plot is defined as

$$\bar{f}_k(x_k) = \mathbb{E}[\hat{f}(\mathbf{x}_{-k}, x_k)] = \int \hat{f}(\mathbf{x}_{-k}, x_k) d\mathbb{P}_{-k}(\mathbf{x}_{-k})$$

where $d\mathbb{P}_{-k}(\cdot)$ is the (multivariate) distribution of the non-k features $\mathbf{x}_{-k}$. The above function takes the feature values x_k as argument and keeps all other features frozen via their sample distributions: this shows the impact of feature k solely. In practice, the average is evaluated using Monte-Carlo simulations:

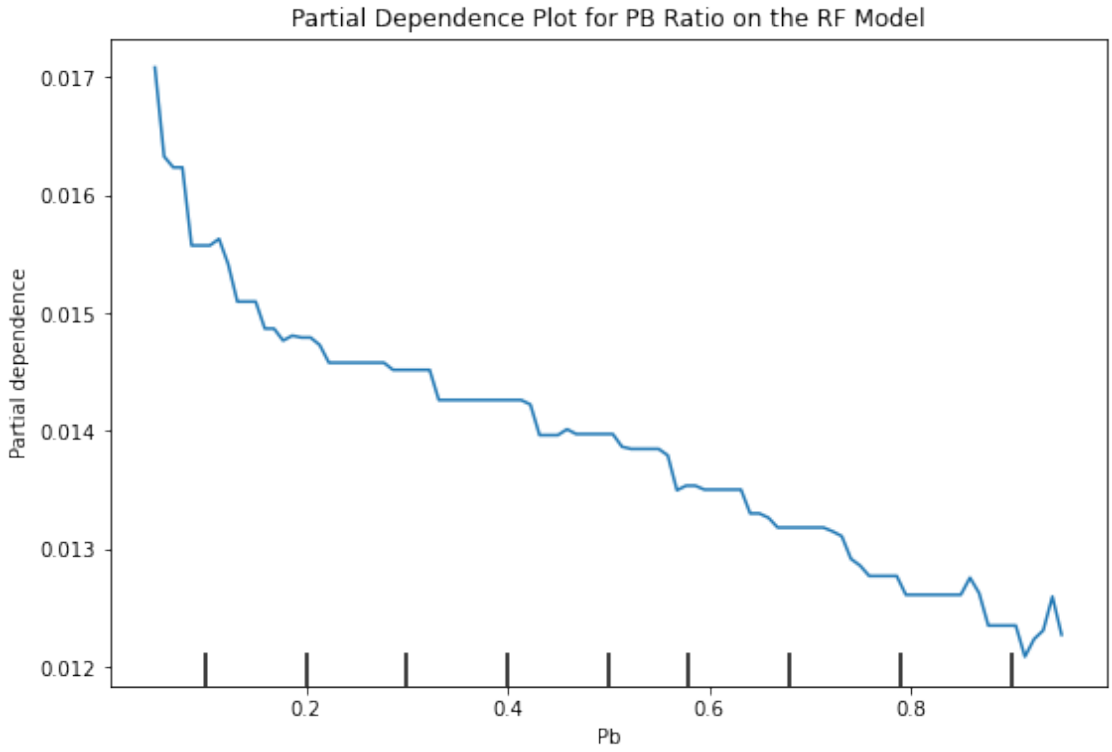
$$\bar{f}_k(x_k) \approx \frac{1}{M} \sum_{m=1}^{M} \hat{f}\left(x_k, \mathbf{x}_{-k}^{(m)}\right)$$

where $\mathbf{x}_{-k}^{(m)}$ are independent samples of the non-k features.

Theoretically, PDPs could be computed for more than one feature at a time. In practice, this is only possible for two features (yielding a 3D surface) and is more computationally intense.

We illustrate this concept below, with the help of the `sklearn` package in Python. The model we seek to explain is the random forest built in Section 6.2. We recycle some variables used therein. We choose to test the impact of the *price-to-book ratio* on the outcome of the model.

[ ]:
```
from sklearn.inspection import PartialDependenceDisplay

fig, ax = plt.subplots(figsize=(9, 6))
pdp_PB = PartialDependenceDisplay.from_estimator(fit_RF,␣
 ↪training_sample[features], ["Pb"], ax=ax)
plt.title("Partial Dependence Plot for PB Ratio on the RF Model")
plt.show()
```



The average impact of the price-to-book ratio on the predictions is decreasing. This was somewhat expected, given the conditional average of the dependent variable given the price-to-book ratio. This latter function is depicted in Figure 6.3 and shows a behavior comparable to the above curve: strongly decreasing for small value of P/B and then relatively flat. When the price-to-book ratio is low, firms are undervalued. Hence, their higher returns are in line with the *value* premium.

Finally, we refer to Zhao and Hastie (2020) for a theoretical discussion on the *causality* property of PDPs. Indeed, a deep look at the construction of the PDPs suggests that they could be interpreted as a *causal representation* of the feature on the model's output.

1.2 Local Interpretations

Whereas global interpretations seek to assess the impact of features on the output *overall*, local methods try to quantify the behavior of the model on particular instances or the neighborhood thereof. Local interpretability has recently gained traction and many papers have been published on this topic. Below, we outline the most widespread methods.

1.2.1 LIME

LIME (Local Interpretable Model-Agnostic Explanations) is a methodology originally proposed by Ribeiro, Singh, and Guestrin (2016). Their aim is to provide a faithful account of the model under two constraints:

- **Simple interpretability**, which implies a limited number of variables with visual or textual representation. This makes sure that any human can easily understand the outcome;
- **Local Faithfulness**: the explanation holds for the vicinity of the instance.

The original (black-box) model is f and we assume we want to approximate its behavior around instance x with the interpretable model g. The simple function g belongs to a larger class G. The vicinity of x is denoted π_x and the complexity of g is written $\Omega(g)$. LIME seeks an interpretation of the form

$$\xi(x) = \underset{g \in G}{\operatorname{argmin}} \, \mathcal{L}(f, g, \pi_x) + \Omega(g)$$

where $\mathcal{L}(f, g, \pi_x)$ is the loss function (error/imprecision) induced induced by g in the vicinity π_x of x. The penalization $\Omega(g)$ is for instance the number of leaves or depth of a tree, or the number of predictors in a linear regression.

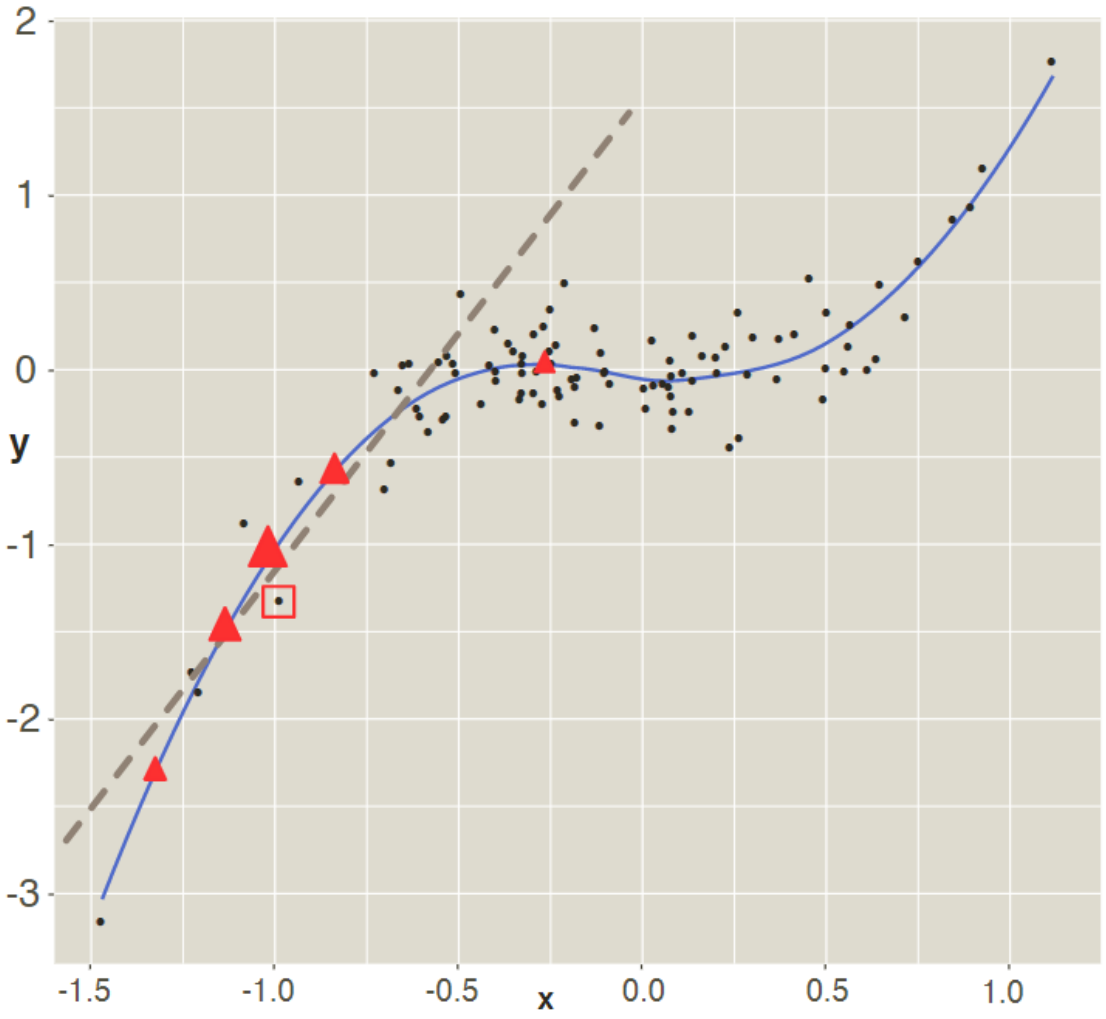
It now remains to define some of the above terms. The vicinity of x is defined by $\pi_x(z) = e^{-D(x,z)^2/\sigma^2}$, where D is some distance measure and σ^2 some scaling constant. We underline that this function decreases when z shifts away from x.

The tricky part is the loss function. In order to minimize it, LIME generates artificial samples close to x and averages/sums the error on the label that the simple representation makes. For simplicity, we assume a scalar output for f, hence the formulation is the following:

$$\mathcal{L}(f, g, \pi_x) = \sum_z \pi_x(z)(f(z) - g(z))^2$$

and the errors are weighted according to their distance from the initial instance x: the closest points get the largest weights. In its most basic implementation, the set of models G consists of all linear models.

In the following Figure 13.5, we provide a simplified diagram of how LIME works.



For expositional clarity, we work with only one dependent variable.

- The original training sample is shown with the black points.
- The fitted (trained) model is represented with the blue line (smoothed conditional average). Wanted: how the model works around one particular instance (the red circle).
- Sampling: 5 new points around the instance (5 red triangles).
- Each triangle lies on the blue line (model predictions) and has a weight proportional to its size: the triangle closest to the instance has a bigger weight.
- A linear model is built with these 5 points with a WLS - two parameters (the intercept and the slope) can be evaluated with standard statistical tests.

The sign of the slope is important. It is fairly clear that if the instance had been taken closer to $x = 0$, the slope would have probably been almost flat and hence the predictor could be locally discarded.

Another important detail is the number of sample points. In our explanation, we take only five, but in practice, a robust estimation usually requires one thousand points or more. Indeed, when too few neighbors are sampled, the estimation risk is high and the approximation may be rough.

We proceed with an example of implementation. There are several steps:

1. Fit a model on some training data.
2. Wrap everything with the lime function.
3. Focus on a few predictors and see their impact over a few particular instances.

We start with the first step. This time, we work with a boosted tree model.

[ ]:
```
import lime
from xgboost import XGBRegressor

xgb_model = XGBRegressor(max_depth=5, learning_rate=0.5,
                         gamma=0.1, colsample_bytree=1,
                         min_child_weight=10, subsample=1.0)
xgb_model.fit(training_sample[features], training_sample['R1M_Usd'])
```

[ ]:
```
XGBRegressor(base_score=0.5, booster='gbtree', callbacks=None,
             colsample_bylevel=1, colsample_bynode=1, colsample_bytree=1,
             early_stopping_rounds=None, enable_categorical=False,
             eval_metric=None, feature_types=None, gamma=0.1, gpu_id=-1,
             grow_policy='depthwise', importance_type=None,
             interaction_constraints='', learning_rate=0.5, max_bin=256,
             max_cat_threshold=64, max_cat_to_onehot=4, max_delta_step=0,
             max_depth=5, max_leaves=0, min_child_weight=10, missing=nan,
             monotone_constraints='()', n_estimators=100, n_jobs=0,
             num_parallel_tree=1, predictor='auto', random_state=0, …)
```

Then, we head on to steps two and three. We will use the package `lime` with the submodule `lime_tabular`.

[ ]:
```
from lime.lime_tabular import LimeTabularExplainer

explainer = LimeTabularExplainer(training_sample[features].values,␣
 ↪mode='regression',
                                 feature_names=features,
                                 verbose=True)
```

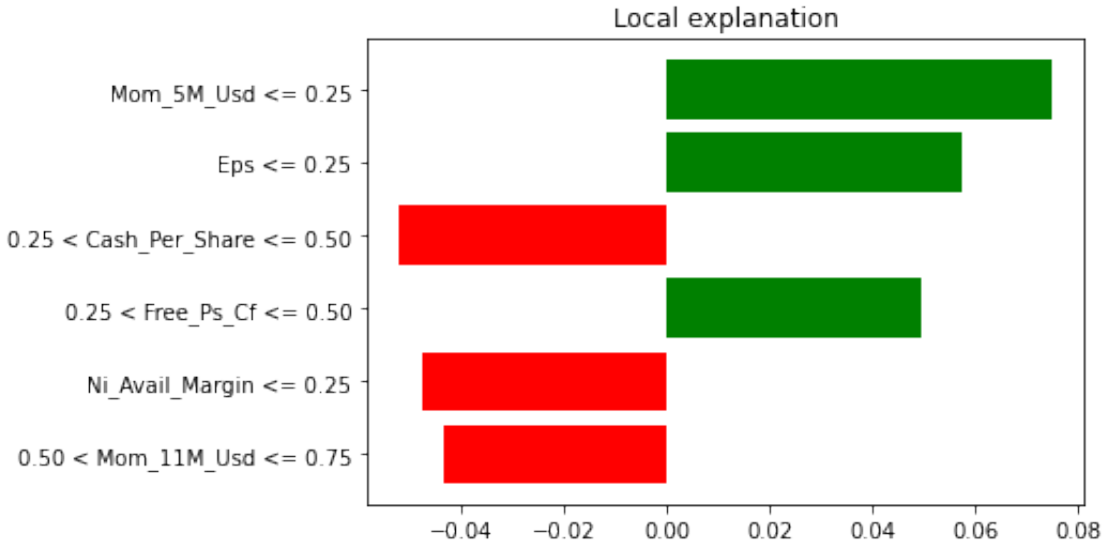
```
explanation = explainer.explain_instance(data_row=training_sample[features].
 ↪iloc[0].values,
                                         predict_fn=xgb_model.predict,
                                         labels=(training_sample['R1M_Usd'].
 ↪iloc[0], ),
                                         num_features=6,
                                         distance_metric='euclidean',
                                         num_samples=900)
```

```
Intercept 0.09639734584246004
Prediction_local [0.13510569]
Right: 0.2636147
```

[ ]:
```
explanation.show_in_notebook()
```

```
<IPython.core.display.HTML object>
```

[ ]:
```
fig = explanation.as_pyplot_figure()
```



There are two types of information: the sign of the impact and the magnitude of the impact. The sign is revealed with the color (positive in green or yellow, negative in red or blue) and the magnitude is shown with the size of the rectangles.

The values to the left of the graphs show the ranges of the features with which the local approxi-

mations were computed.

Lastly, we briefly discuss the choice of distance function chosen in the code. It is used to evaluate the discrepancy between the true instance and a simulated one to give more or less weight to the prediction of the sampled instance. Our dataset comprises only numerical data; hence, the Euclidean distance is a natural choice:

$$\text{Euclidean}(\mathbf{x}, \mathbf{y}) = \sqrt{\sum_{n=1}^{N} (x_i - y_i)^2}.$$

Another possible choice would be the Manhattan distance:

$$\text{Manhattan}(\mathbf{x}, \mathbf{y}) = \sum_{n=1}^{N} |x_i - y_i|.$$

The problem with these two distances is that they fail to handle categorical variables. This is where the *Gower distance* steps in (Gower (1971)). The distance imposes a different treatment on features of different types (classes versus numbers essentially, but it can also handle missing data!). For categorical features, the Gower distance applies a binary treatment: the value is 1 if the features are equal and 0 if not (i.e., $1_{\{x_n = y_n\}}$). For numerical features, the spread is quantified as $1 - \frac{|x_n - y_n|}{R_n}$, where R_n is the maximum absolute value the feature can take. All similarity measurements are then aggregated to yield the final score. Note that in this case, the logic is reversed, $\mathbf{x}$ and $\mathbf{y}$ are very close if the Gower distance is close to 1, and they are far away if the distance is close to 0.

1.2.2 Shapley Values

The approach of Shapley values is somewhat different compared to LIME and closer in spirit to PDPs. It originates from cooperative game theory (Shapley, 1953). One way to assess the impact (or usefulness) of a variable is to look at what happens if we *remove* this variable from the dataset. If this is very detrimental to the quality of the model (i.e., to the accuracy of its predictions), then it means that the variable is substantially valuable.

The simplest way to proceed is to take all variables and remove one to evaluate its predictive ability. Shapley values are computed on a larger scale because they consider all possible combinations of variables to which they add the target predictor. Formally, this gives:

$$\phi_k = \sum_{S \subseteq \{x_1, \ldots, x_K\} \setminus x_k} \underbrace{\frac{\text{Card}(S)!(K - \text{Card}(S) - 1)!}{K!}}_{\text{weight of coalition}} \underbrace{\left(\hat{f}_{S \cup \{x_k\}}(S \cup \{x_k\}) - \hat{f}_S(S) \right)}_{\text{gain when adding } x_k}$$

S is any subset of the **coalition** that doesn't include feature k and its size is $\text{Card}(S)$.

In the equation above, the model f must be altered because it's impossible to evaluate f when features are missing. In this case, there are several possible options:

- Set the missing value to its average or median value (in the whole sample) so that its effect is some 'average' effect;
- Directly compute an average value $\int_{\mathbb{R}} f(x_1, \dots, x_k, \dots, x_K) d\mathbb{P}_{x_k}$, where $d\mathbb{P}_{x_k}$ is the empirical distribution of x_k in the sample.

Obviously, Shapley values can take a lot of time to compute if the number of predictors is large. We refer to Chen et al. (2018) for a discussion on a simplifying method that reduces computation times in this case. Extensions of Shapley values for interpretability are studied in Lundberg and Lee (2017).

In python, we will use the `shap` package to implement the Shapley values. We start by fitting a random forest model.

[ ]:
```
from sklearn.ensemble import RandomForestRegressor

features_short = ["Div_Yld", "Eps", "Mkt_Cap_12M_Usd", "Mom_11M_Usd", "Ocf",␣
 ↪"Pb", "Vol1Y_Usd"]
fit_RF_short = RandomForestRegressor(n_estimators=40,
                                     max_samples=10000,
                                     bootstrap=True,
                                     max_features=4,
                                     min_samples_leaf=250)
fit_RF_short.fit(training_sample[features_short], training_sample['R1M_Usd'])
```

[ ]:
```
RandomForestRegressor(max_features=4, max_samples=10000, min_samples_leaf=250,
                      n_estimators=40)
```

We can then analyze the behavior of the model around the first instance of the training sample.

[ ]:
```
import shap

explainer = shap.explainers.Exact(model=fit_RF_short.predict,
                                  masker=training_sample[features_short].values,
                                  feature_names=features_short)
shap_values = explainer(training_sample[features_short].iloc[:1].values) # on␣
 ↪the first instance
shap_values
```

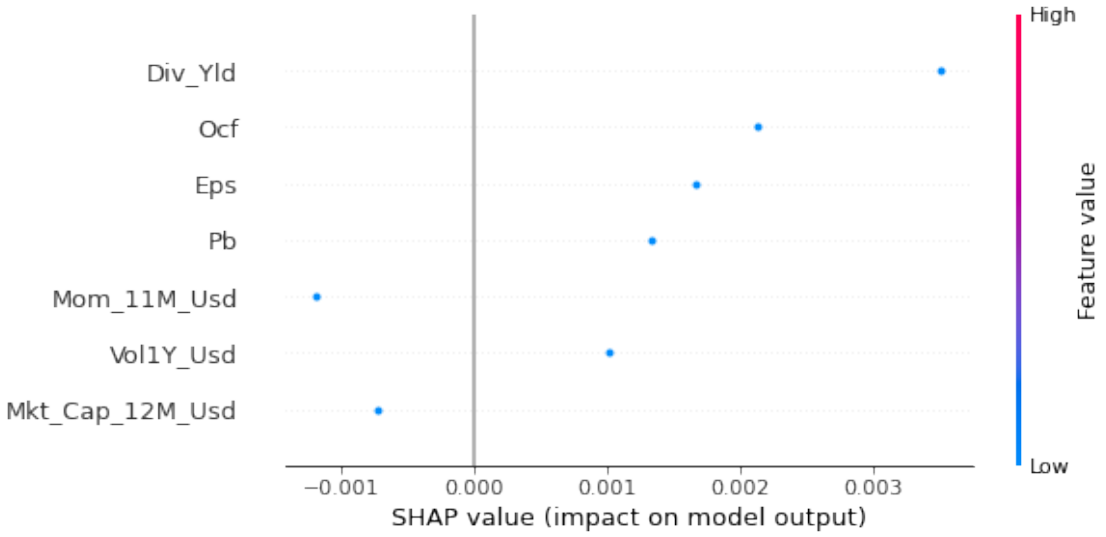
```
X does not have valid feature names, but RandomForestRegressor was fitted with
feature names
```

[ ]: .values =
array([[0.00351467, 0.00166742, -0.00071621, -0.00118842, 0.00213526,
         0.00133457, 0.00101373]])

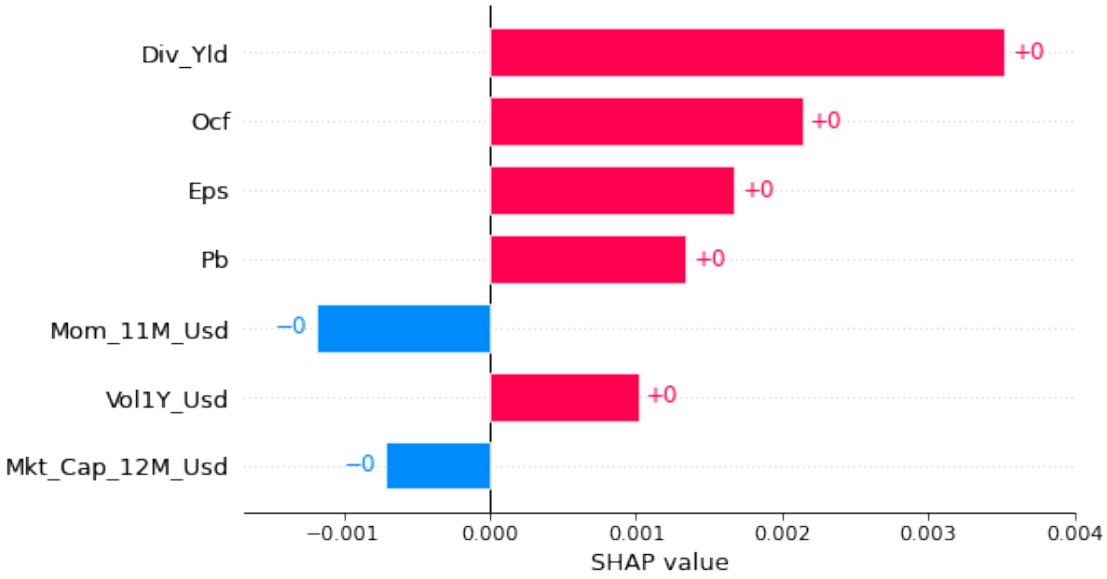
.base_values =
array([0.0143331])

.data =
array([[0.01, 0.01, 0.52, 0.59, 0.7 , 0.21, 0.77]])

[ ]:
```
import matplotlib.pyplot as plt
shap.plots.beeswarm(shap_values)
plt.show()
```



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



In the output shown, we again obtain the two crucial insights: **sign** of the impact of the feature and **relative importance** (compared to other features).

1.2.3 Breakdown

Breakdown (see, e.g., Staniak and Biecek (2018)) is a mixture of ideas from PDPs and Shapley values. The core of breakdown is the so-called **relaxed model prediction** defined in the following equation:

$$\tilde{f}_{\boldsymbol{k}}(x^*) = \frac{1}{M}\sum_{m=1}^{M} \hat{f}\left(x_{-\boldsymbol{k}}^{(m)}, x_{\boldsymbol{k}}^*\right)$$

It is close in spirit to Equation (13.1) for PDP. The difference is that we are working at the local level, i.e., on one particular observation, say x^*. We want to measure the impact of a set predictors on the prediction associated to x^*; hence, we fix two sets $\boldsymbol{k}$ (fixed features) and $-\boldsymbol{k}$ (free features) and evaluate a **proxy** for the average prediction of the estimated model $\hat{f}$ when the set $\boldsymbol{k}$ of predictors is fixed at the values of x^* (equal to $x_{\boldsymbol{k}}^*$).

The $x^{(m)}$ in the above expression are either simulated values of instances or simple sampled values from the dataset. The notation implies that the instance has some values replaced by those of x^*, namely those that correspond to the indices $\boldsymbol{k}$. When $\boldsymbol{k}$ consists of all features, then $\tilde{f}_{\boldsymbol{k}}(x^*)$ is equal to the raw model prediction $\hat{f}(x^*)$ and when $\boldsymbol{k}$ is empty, it is equal to the average sample value of the model (constant prediction).

The quantity of interest is the so-called contribution of feature $j \notin \boldsymbol{k}$ with respect to data points x^* and set $\boldsymbol{k}$:

$$\phi^j_{\boldsymbol{k}}(x^*) = \tilde{f}_{\boldsymbol{k}\cup j}(x^*) - \tilde{f}_{\boldsymbol{k}}(x^*)$$

Just as for Shapley values, the above indicator computes an average impact when augmenting the set of predictors with feature j. By definition, it depends on the set $\boldsymbol{k}$, so this is one notable difference with Shapley values (that span *all* permutations).

In Staniak and Biecek (2018), the authors devise a procedure that incrementally increases or decreases the set $\boldsymbol{k}$. This greedy idea helps alleviate the burden of computing all possible combinations of features. Moreover, a very convenient property of their algorithm is that the sum of all contributions is equal to the predicted value:

$$\sum_j \phi^j_{\boldsymbol{k}}(x^*) = f(x^*).$$

The visualization makes that very easy to see. In order to illustrate one implementation of breakdown, we train a random forest on a limited number of features, as shown below. This will increase the readability of the output of the breakdown.

[ ]:
```
fit_RF_short = RandomForestRegressor(n_estimators=12,
                                     max_samples=10000,
                                     bootstrap=True,
                                     max_features=5,
                                     min_samples_leaf=250)
fit_RF_short.fit(training_sample[features_short], training_sample['R1M_Usd'])
```

[ ]:
```
RandomForestRegressor(max_features=5, max_samples=10000, min_samples_leaf=250,
                      n_estimators=12)
```

[ ]:
```
import dalex as dx

explainer = dx.Explainer(fit_RF_short, training_sample[features_short],␣
 ↪training_sample['R1M_Usd'])
explain_break = explainer.predict_parts(pd.DataFrame(training_sample.loc[0,␣
 ↪features_short]).T, type='break_down')
explain_break.plot()
```

```
Preparation of a new explainer is initiated
```

```
  -> data               : 198128 rows 7 cols
  -> target variable    : Parameter 'y' was a pandas.Series. Converted to a
numpy.ndarray.
  -> target variable    : 198128 values
  -> model_class        : sklearn.ensemble._forest.RandomForestRegressor
(default)
  -> label              : Not specified, model's class short name will be used.
(default)
  -> predict function   : <function yhat_default at 0x0000028187C168C0> will be
used (default)
  -> predict function   : Accepts pandas.DataFrame and numpy.ndarray.
  -> predicted values   : min = -0.0182, mean = 0.0129, max = 0.0589
  -> model type         : regression will be used (default)
  -> residual function  : difference between y and yhat (default)
  -> residuals          : min = -0.932, mean = 0.00106, max = 27.2
  -> model_info         : package sklearn

A new explainer has been created!

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

The graphical output is intuitively interpreted.

- The purple bar is the prediction of the model at the chosen instance.
- Green bars signal a positive contribution and the red rectangles show the variables with negative impact.
- The relative sizes indicate the importance of each feature.

1.3 Takeaways

Interpretability (tools): informative and comprehensible.

- **Global models** and global interpretations: *average* over the training set.
  - Simple models as surrogates: simple models as a proxy to complex algorithms
  - Variable importance
    - Tree-based: frequency of a feature used to perform the split
    - Agnostic: randomly permute a feature's values and compare the loss difference
  - Partial dependence plot (PDPs)
    - Fix a feature k and predict the **average impact** of k on the predictions of the

trained model

- **Local interpretations**
  - **LIME** (Local Interpretable Model-Agnostic Explanations): an interpretation to minimize a specific loss function with the vicinity of x
    - Distance metrics choices: Euclidean, Manhattan, Gower (to handle classification tasks)
  - Shapley Values: loss differences when removing a particular variable considering all possible combinations
  - Breakdown: a mixture from PDPs and Shapley values but depending on only one set $\boldsymbol{k}$ instead of all permutations