06_explainability

June 21, 2025

[ ]:
```
%load_ext dotenv
%dotenv
import os
import sys
sys.path.append(os.getenv('SRC_DIR'))
import pandas as pd
import numpy as np
import pickle
ft_path = os.getenv("CREDIT_DATA")
df_raw = pd.read_csv(ft_path)
```

[ ]:
```
from credit.data import load_data

X, Y = load_data()
```

1 Loading Model Artifacts

Previously, we built a procedure for hyperparameter tuning. The process produces parametrized models that use the optimal parameters based on Grid Search or Hyperopt. The models are saved in the object store. We can access these objects via MLFlow's API using the uri: `models:/<model name>/<model version>`

Alternatively, the model could have been stored using the standard pickle format. We can load a model using `pickle.load()` within a context manager that handles the file (a `with(open(f, 'rb'))...` statement). The context manager will help close files in case of unexpected termination.

[ ]:
```
import mlflow.sklearn
mlflow.set_tracking_uri('http://localhost:5001')
model_name = 'CreditLogisticModel'
model_version = '6'
model_uri = f'models:/{model_name}/{model_version}'
pipe = mlflow.sklearn.load_model(model_uri)
pipe
```

In this case, we loaded a logistic regression model. The model, however, could have been from another family of models (Random Forest, Neural Net, etc.)

Below, we will explore some model-agnostic explainability methods.

2 Partial Dependence Plots

- Partial Dependence Plots (PDP) show the relationship between the target response and the input feature.
- They can be constructed for one or two inputs at a time.

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

X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size=0.2,␣
 ↪random_state=42)

# Why do we fit again?

pipe.fit(X_train, y_train)
```

2.1 Single-Feature PDP

[ ]:
```
PartialDependenceDisplay.from_estimator(pipe, X_train,
                                        features =␣
 ↪['revolving_unsecured_line_utilization'])
```

[ ]:
```
PartialDependenceDisplay.from_estimator(pipe, X_train,
                                        features =␣
 ↪['revolving_unsecured_line_utilization', 'debt_ratio'])
```

2.2 Two-Feature PDP

[ ]:
```
PartialDependenceDisplay.from_estimator(pipe, X_train,
                                        features =␣
 ↪[('revolving_unsecured_line_utilization', 'debt_ratio')])
```

2.3 Partial Dependence Values

You may require the underlying data of the plots above. To obtain it, use the function `partial_dependence()`.

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

partial_dependence(pipe, X_train, features =␣
 ↪['revolving_unsecured_line_utilization', 'high_debt_ratio'])
```

2.4 Permutation Feature Importance

- Permutation feature importance measures the contribution of each feature to a fitted model's performance.

- Randomly shuffles the values of a single feature and observing the result degradation of the model's score. If shuffling a feature greatly degrades performance, then we say the feature is important.
- Shuffling is involved, therefore it is convenient (and costly) to perform several repetitions.

Scikit's Documentation makes this warning:

> **Warning**: Features that are deemed of **low importance for a bad model** (low cross-validation score) could be **very important for a good model**. Therefore it is always important to evaluate the predictive power of a model using a held-out set (or better with cross-validation) prior to computing importances. Permutation importance does not reflect to the intrinsic predictive value of a feature by itself but how important this feature is for a particular model.

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

pi_res = permutation_importance(
    pipe, X_test, y_test,
    n_repeats=30,
    scoring = "neg_log_loss")
```

The function returns a dictionary with the following entries:

- `importances_mean`: mean of feature importance.
- `importances_std`: standard deviation of feature importance over n_repeats.
- `importances`: raw permutation importance scores (one feature per row, one reshuffle per column).

[ ]:
```
importances_dt = pd.DataFrame(pi_res.importances).T
importances_dt.columns = X_test.columns
```

[ ]:
```
import matplotlib.pyplot as plt
bp = plt.boxplot(importances_dt, vert = False,  labels = importances_dt.columns)
```

[ ]:
```
pd.concat(
    [
        pd.Series(pi_res.importances_mean, index = X_test.columns, name = 'Mean␣
 ↪Importance'),
        pd.Series(pi_res.importances_std, index = X_test.columns, name = "Std␣
 ↪Importance")
    ],
    axis = 1).sort_values("Mean Importance", ascending  = False)
```

2.5 SHAP Values

- SHAP is an advanced approach for providing explanation to model results.
- One library that implements this procedure is Shap.

2.5.1 Explainers

- SHAP values can be calculated for any model, however, the procedure can be computationally expensive.
- For certain models, some specific functions exist to speed the cauclations: they are contained in the `shap.explainers` module. A few noteable functions are:
  - `shap.Explainer()`: the primary explainer interface and chooses the explanation algorithm for you.
  - `shap.TreeExplainer()`: implements Tree Shap, a procedure optimized for tree-based ensemble methods (Random Forest, XGBoost, etc.)
  - `shap.LinearExplainer()`: computes SHAP Values for linear methods.

2.5.2 Obtaining explanations from shap

- Shap can obtain local and global explanations. The model produces additive explanations, therefore, obtaining global explanations is equivalent to obtaining individual explanations for all samples.
- Shap can obtain explanations for testing and training samples.
- Local explanations are obtained as shap_values, which reflect the contribution of each feature to the prediction made for each sample.

2.5.3 Using the shap package

- SHAP works on classifiers, but our pipelines contain preprocessing and classification steps.
- We can access each individual step through the `.named_steps` attirbute.
- Notice that we apply the transformation (`ColumnTransformer`) step to obtain transformed data and store the results in `data_transform`.
- Feature names are obtained from the preprocessor's `.get_feature_names_out()` which exposes the names of the features after they have been transformed by the `ColumnTransformer`.
- The explainer object is then use to provide all explanations in `data_transform`.

[ ]:
```
import shap
data_transform = pipe.named_steps['preproc'].transform(X_test)

explainer = shap.explainers.Linear(
    pipe.named_steps['clf'],
    data_transform,
    feature_names = pipe.named_steps['preproc'].get_feature_names_out())

shap_values = explainer(data_transform)
```

2.5.4 Waterfall Plots

From SHAP's documentation (emphasis added):

> Waterfall plots are designed to display explanations for **individual predictions**, so they expect a single row of an Explanation object as input. The bottom of a waterfall plot starts as the expected value of the model output, and then each row shows how

the positive (red) or negative (blue) contribution of each feature moves the value from the expected model output over the background dataset to the model output for this prediction.

- The waterfall plot below shows the contribution of each feature to an individual prediction.
- Only the most important features are shown, while the least important features are grouped together at the bottom of the chart (e.g., "4 other features").

[ ]:
```
shap.plots.waterfall(shap_values[1])
```

2.5.5 Beeswarm plot

- Beeswarm plots display the contributions of each feature to all cases in the sample. The feature values are color-coded when available.
- Beeswarm plots summarize the behaviour of the model across all items in the sample.

[ ]:
```
shap.plots.beeswarm(shap_values)
```