Explainability v2.0

Lewis Cole – 2023

Contents

[Outline 2](#_Toc124861275)

[Processing the Data Set 3](#_Toc124861276)

[Conformal Prediction 4](#_Toc124861277)

[Global Explainability 6](#_Toc124861278)

[Cross-Validation Results 6](#_Toc124861279)

[Rules Explainer 7](#_Toc124861280)

[Accumulated Local Effect 8](#_Toc124861281)

[Local Explainability 10](#_Toc124861282)

[Conformal Prediction Revisited 10](#_Toc124861283)

[Rules Explainer Revisited 11](#_Toc124861284)

[Anchor 12](#_Toc124861285)

[Counterfactual 14](#_Toc124861286)

[SHAP 16](#_Toc124861287)

[Nearest Neighbour 17](#_Toc124861288)

[ALE Revisited 18](#_Toc124861289)

# Outline

For a machine learning model to be accepted by a business, particularly within a regulated industry, its decisions need to be understood by the user. It is in some sense more important that the user feels confident in the models decisions rather than the model be the top performing. Regulation around such models is also increasing and it is no longer acceptable to accept a “black box” model.

This document outlines a selection of explainability methods, some of the problems we aim to address are:

* How does a user feel comfortable in using a model?
* How do we avoid ‘blind confidence’ in a model’s prediction?
* How can we understand ‘why’ a model has made a specific prediction?

This document outlines various methods aiming to address these problems. Working code has been produced on real world data, this code overcomes many of the complications of using these methods but is not in a “production ready” state.

# Processing the Data Set

In most datasets we encounter we have a combination of categorical, discrete, and continuous features. It is important when processing the data that we ensure these are formatted correctly.

In order for the methods to work correctly we require for categorical variables:

* Categorical features to be label encoded in a contiguous manner starting from 0
* The dtype of the dataframe column must be ‘category’

An example basic processing script can be seen below:

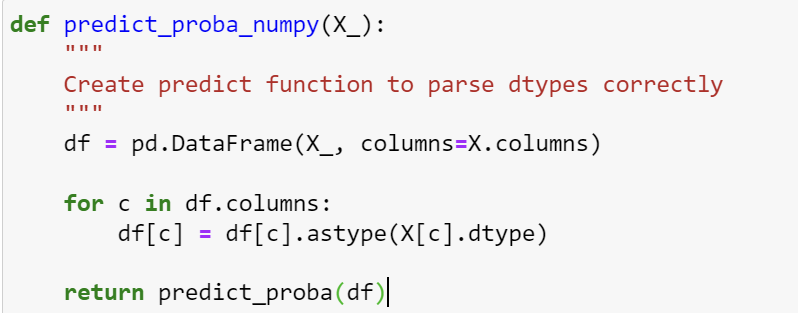
Graphical user interface, text, email

Description automatically generated

Here sklearn’s LabelEncoder is used.

Since we may be using a COPOD model later (which intrinsically constructs an empirical Copula) – the best approach would be a “frequency encoding”. Whereby the most frequent category gets encoded as “0”, the next most common “1” and so on (or equivalently the least frequent gets coded “0”, next least frequent gets coded “1” and so on).

Encoded in this way LightGBM (LGBM) is able to handle categorical features in this way with no other changes required. However, care must be taken when the LGBM is trained on a dataframe of this form but then predict\_proba is called on a numpy array – LGBM will not throw an error, but the resulting pseudo-probabilities no longer make sense. A simple wrapper needs to be made to overload the predict\_proba method to accept numpy arrays (the explainabiility methods require numpy arrays as inputs). A very basic example can be seen below:



Here the function takes input array “X\_” which is converted to a dataframe (“df”) – the columns are taken to be the columns in (global) dataframe “X”, and the dtypes of “df” are converted to those of “X”. The predict\_proba function is then applied to the dataframe.

# Conformal Prediction

The first step to explainability is reliability. We cannot expect the user to confident in the model predictions unless the outputs are shown to be reliable. In order to do this we can use conformal prediction techniques to produce the following as part of each “predict\_proba” call:

* Well calibrated probabilities
* An uncertainty metric
* An out of distribution metric

We take care of the first two bullet points by using a Venn-Abers Predictor: <https://arxiv.org/abs/1211.0025>

This produces two estimates: p0 and p1 – p0 is the true probability of a “positive” label given the example falls in the “negative” class. In contrast p1 is the true probability of a “positive” label given the example falls in the “positive” class. That is, for each prediction either p0 or p1 is the true probability of a positive label – however, at inference time we have no way of knowing which of these estimates is the correct one to use. We therefore combine p0 and p1 in such a way as to produce a “best estimate” probability – in practice we typically take: p1 / ( 1 – p0 + p1) – intuitively this should make sense, but it also minimizes the log-loss. This satisfies the first bullet above.

The second bullet point is covered by returning: p1 – p0 – by definition this is in the range [0,1] and is well defined. This gives a range on the true probability of a positive label. For very strong predictive models this range is small (the model is very certain) for weak models the range is very large (the model is uncertain) – this uncertainty varies by individual observation.

In order to simplify and speed up the inference time we can “pre-train” our conformal prediction adjustment and simply apply it as a function to our model outputs. We can generate p0 and p1 on a “grid” of points, and then calculate the equivalent “best estimate” (p\_score) and “uncertainty” (unc) estimates on those grid points:

Text

Description automatically generated

While this code is logically correct and easy to follow, it is not the most optimal from a run time perspective. The loop can be replaced by the functions in code base: <https://github.com/ptocca/VennABERS> - for significant speed up.

Given we have estimates of probability and uncertainty on our grid points we can then adjust the results of our classifier to become a conformal predictor using interpolation. Here I use basic linear interpolation in numpy:

Graphical user interface, text

Description automatically generated

This function calls some global numpy arrays defined in the previous step. A similar function can be defined to return the uncertainty.

In order to satisfy the third bullet: an out of distribution probability estimate – we can turn a basic COPOD anomaly detector into a conformal predictor. This converts the pseudo-probability of the anomaly detector into a “true” probability of being an outlier. This can be achieved neatly via a few calls of a quantile transformer:

Text

Description automatically generated

Following the codes above we are able to calculate (for each inference example):

* Probability of positive class
* Uncertainty in the probability of positive class
* Probability that the inference example is from outside the training set

Thanks to the theoretical guarantees of conformal prediction we can interpret these as “true” probabilities which is vastly superior to an arbitrary model score. The uncertainty and out of distribution estimates should also help alleviate “blind confidence” in model results.

# Global Explainability

A global explanation of a model aims to provide an insight of how the model performs in aggregate – that is: “how does the model perform overall?”. While this can’t help a user understand a particular prediction, a well presented global explanation should help the user feel confident that the model is “doing something sensible”.

Global explainability has a strong over-lap with “model governance” and so some visualizations and metrics will be shared between multiple UI screens.

## Cross-Validation Results

The first way we can communicate the model’s performance in the aggregate is to show the cross validated results These are generated at train time and so will not change through the lifetime of the model, but they should help explain why the model was considered suitable for use at the outset. The particular metrics and plots to use can be developed over time, some basic options can be seen below:

Chart, line chart

Description automatically generated

**Implementation Comments and Concerns**

Exhibits like these should add minimal additional workflow, as part of productionization we have to decide where best to store these metrics/arrays so that they are available for display.

## Rules Explainer

Another way in which we can explain the model in aggregate is to create a simple set of rules that explain how the model makes a binary decision on the positive class. One way to do this is to use the RIPPER method (Repeated Incremental Pruning to Produce Error Reduction). This is implemented in the Wittgenstein library ( <https://github.com/imoscovitz/wittgenstein> ) - amongst other options.

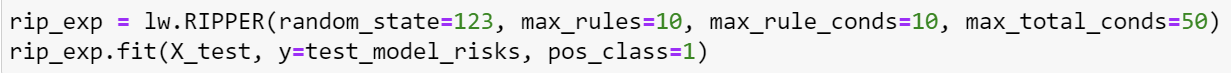
In order to implement RIPPER (as part of the Wittengstein library) we first have to apply a threshold to our model (i.e. create “predictions” from the probabilities):

A picture containing logo

Description automatically generated

We can use our calibration set (here labelled X\_test) from model training to do this.

We then simply instantiate the class and use a fit method:



In the instantiation we can control limits on the number of rules and clauses. Since we are applying this to the calibration set as part of model training as part of our standard model training workflow we can manipulate the parameters to get a suitably “nice” looking rule set explainer. We can then extract the ruleset:

Text

Description automatically generated

In this example we can see that a ruleset containing only a subset of categorical variables explains the model decisions. We can then calculate the accurachy of the ruleset in predicting the model and the precision on the positive and negative classes:

Text

Description automatically generated with low confidence

So we can see that in 97% of examples the ruleset matches the outcome of the trained LGBM model. This is a fairly good result given there are only 4 rules being applied, the user will then be able to verify the model by considering if this ruleset “makes sense” to them.

**Implementation Comments and Concerns**

Adds to workflow, may take some playing around to get a suitable ruleset.

Other libraries or methods may be more suitable to producing rule sets (e.g. IREP, SLIPPER, BRCG, etc.) The Wittgenstein library appeared to be the simplest and least error-prone library I could find.

## Accumulated Local Effect

The next question we may aim to answer is: “what happens if I vary feature values?” – that is, how does the predicted probabilities vary with feature value. To answer this question we can look at Accumulated Local Effect (ALE) plots - <https://christophm.github.io/interpretable-ml-book/ale.html> - ALE plots are used over partial dependence (PD) plots since in general we expect our features to not be independent from one another. The ALE plot shows, on average, how changing the value of a single feature impacts the predicted probability.

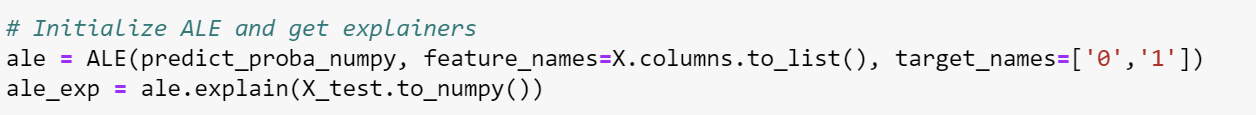
Example plots can be seen below:

Chart, line chart

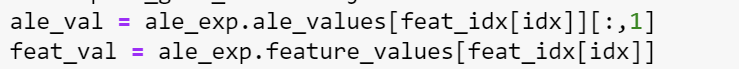
Description automatically generated

When the blue line is above the red line the predicted probability is increased and vice versa. We can see that there is a large increase in probability if the PAY\_0 categorical variable takes values 3 or above – this backs up the finding from the rules based explainer where PAY\_0 = 3, PAY\_0 = 4 and PAY\_0 = 5 all appear as rule conditions. This should help the user gain comfort that the model is acting sensibly.

Calculating the values for these plots is relatively simple, we instantiate an ALE class object (passing in the model’s predict\_proba function) and fit it to some feature set:



The ale\_exp object then contains arrays which can be accessed containing the X and Y values for the plots above:



**Implementation Comments and Concerns**

These plots can be generated at model train time.

The work to productionize will largely be around how/where to store the curves so that they can be surfaced within the UI.

The implementation used in the prototype uses the Alibi library from SeldonIO ( <https://docs.seldon.io/projects/alibi/en/stable/> )

# Local Explainability

We now turn our attention to local explainability. The goal of local explainability is to provide an explanation for an individual model score/decision.

## Conformal Prediction Revisited

In a previous section we turned our classification model into a conformal predictor, we can now use these results to provide local reliability probabilities:

Text

Description automatically generated

Furthermore, we can look at these probabilities in context of the wider population to give additional insight:

Chart, line chart, histogram

Description automatically generated

Here the red line represents the value of the particular example, and we can convert that into a percentile of the population (96% in this example). The black curve represents the density of the population (that is: if you integrate the black curve to the left of the red you’ll get 96%). We can also view sub-populations: in this case the grey curve representing the “non-risky” population’s density and the pink curve the “risky” populations density. From this we can see (for this example) if this entity is “non-risky” it is an extreme outlier in the probability of risk. We can produce equivalent plots for uncertainty and out-of-distribution conformal predictions.

**Implementation Comments and Concerns**

In order to get the “risk” and “non-risk” curves we will need to join the scores table to the outcomes table. For simplicity we can take these to be the calibration set of the training data and so these shall remain static. For the “total” (black) curve and percentile we can either take the calibration set (and so the curve remains static) or we could take a latest “cohort” (say 3 months prior) and make the plot dynamic.

## Rules Explainer Revisited

Recall from above we created a rules based explainer for global model explainability. We can use this model locally – that is; what would this set of rules predict given this particular example.

Text

Description automatically generated with medium confidence

A potential outcome might be seen above. Here the rules based predictor has assigned “risk” as its outcome, and this is explained by the rule PAY\_0=4. If the rules based prediction matches the model’s prediction (as it does in this case) then this provides a reasonable proxy for explaining the model’s decision.

The rules based prediction will not match the model’s prediction 100% of the time. In the cases where the rules based prediction differs from the model this also provides additional information to the user and suggests that this might be a particularly “difficult” to classify example. Therefore some of the following more powerful methods may be of use.

Given a trained RIPPER instance from Wittgenstein (created in global explainability section) we can simply call the decision and explanation above via:



**Implementation Comments and Concerns**

This is relatively simple to implement, we need to pickle (or otherwise store) the rules based explainer and then call it. Since evaluating rules should not be computationally expensive this could be done at inference time and stored in a “rules explainer” table.

## Anchor

The next local explainability method we can look at is the “anchor”. In layman’s terms an anchor represents a subset of features for a given example, such that if they are kept constant (whilst the others are free to vary) the model will predict the same outcome. This is a very intuitive way of looking at explainability, it is in some sense the “main” reasons that the model has predicted the outcome it has. The original paper on anchors can be found: <https://homes.cs.washington.edu/~marcotcr/aaai18.pdf>

When an anchor is produced we can also provide 2 supporting statistics: the precision – which represents the proportion of examples in the neighbourhood of the anchor that produce the same model prediction, and the coverage – which represents the proportion of neighbours that share this anchor’s feature values. Ideally we would like an anchor to have a high precision (i.e. is very predictive of the model outcome) but low coverage (i.e. is somewhat “unique” to this example compared to its neighbourhood).

An example outcome of the anchor method could be:

Text

Description automatically generated

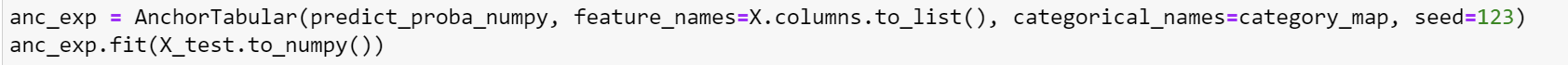
Here we have 3 categorical variable values that appear to be very predictive with relatively low coverage. In this particular example we can see this provides additional supporting evidence to the rules explainer, and the results of the ALE plots.

In order to use the Anchor generating library we first have to pre-process the categorical variables to a dictionary that has as a key the column number of the categorical variable, and the corresponding values being the ordered integer values the variable cant take stored as strings. An example of how to do this is:

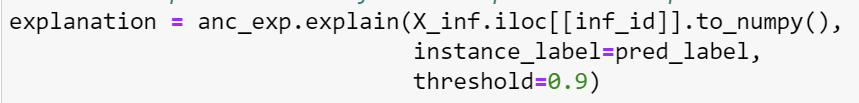
A picture containing calendar

Description automatically generated

We can then pass the predict\_proba function, a list of feature names and the categorical map into the anchor object’s initialization and fit it to some feature set:



We can then perform an inference prediction on a specific example via:



The threshold parameter controls the “precision” of the resulting anchor. The returned anchor must have a precision of at least 90% in this example. Once fit we can return the explanation, precision and coverage:

Graphical user interface, text, application, email

Description automatically generated

**Implementation Comments and Concerns**

The anchor method is relatively slow at inference time. It is unlikely that we can generate these explanations in advance. We will likely have to expose this via a UI button: “generate anchor”, or only generate these explanations for a small subset (e.g. the alerted population only).

In practice we will also have to convert categorical variables to their associated “true” name (e.g. “teacher” for the occupation variable).

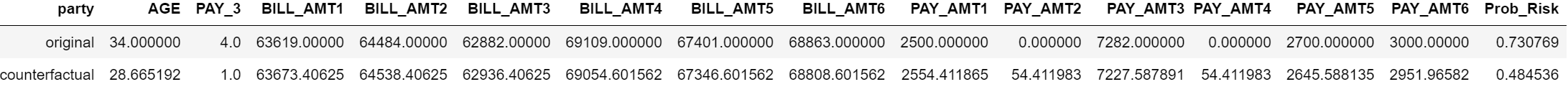
The implementation used comes from the Alibi library.

## Counterfactual

A counterfactual is in some sense the “reverse” of the anchor – while the anchor is the minimal set of features that keeps the prediction the same, the counterfactual is the minimal variation to feature values required to “reverse” the model’s prediction (in a binary classification task).

A challenge with counterfactual analysis is generating “sensible” counterfactuals – that is producing counterfactuals that look “realistic”. It does not aid explanation to (for example) say that a customer should be “half way between an accountant and a bricklayer” – while the model may be able to interpret such a concept and produce a probability it does not reflect the real world. In order to overcome this difficult it was found that using the concept of a “prototype” works well, this was originally presented in the paper: <https://arxiv.org/abs/1907.02584> .

An example output from the counterfactual may look like:



Here we can see that changing a few features (most notably AGE, PAY\_3, PAY\_AMT2, PAY\_AMT4) the probability of risk reduces from 73% to 48%. This can help the user contextualize the model’s prediction, if these changes are considered spurious (say it just so happened that this month the customer’s PAY\_AMT2, PAY\_AMT4 are zero – but ordinarily they’re around the 50 mark) – then the user can temper their expectations on the customer being risky.

In order for the counterfactuals to be generated there are a few pre-processing steps. Firstly eager-execution must be enabled in TensorFlow or the method will not work currently. Second, we have to create a category dictionary. This is slightly different to the category map used for anchors – it takes as values the column number and as the values the number of categories within that column. We also need to specify a range for each feature, this is typically the entire range observed for a feature (to ensure it does not go outside the observed range in calculating the counterfactual) but it can be specified to be smaller than this (e.g. interquartile range) to force counterfactuals to be nearer the body of the distribution and not in the tails. An example of how to do this can be seen:

Text

Description automatically generated

The counterfactual can then be generated via:



The counterfactual’s feature values can then be extracted via:



**Implementation Comments and Concerns**

At present time counterfactuals are not ready for production.

Generating counterfactuals is extremely expensive computationally. Pre-calculating counterfactuals is entirely out of the question. In testing the quickest counterfactual calculation for this example was approx.. 7 minutes – in comparison the anchor calculated in a couple of seconds. With additional work we may be able to speed this up (through distribution on Ray, parameter tuning, models with quicker predict\_proba methods, using a different library for counterfactual generation, etc.)

The method is also fairly sensitive to hyperparameter choice and if these are not correctly specified it may not be possible to generate the counterfactual.

Another challenge with the prototype approach to counterfactuals specifically is that (currently) there is no way change the “threshold” applied to the model. That is: it considers anything with probability > 50% to be “risky” and <50% “non-risky”. This means that if you calculate a counterfactual for a customer with a score of 51% the counterfactual instance only needs to look for a counterfactual with score 49% - for all intents and purposes not a material change in risk.

## SHAP

In explainability v1.0 we calculated SHAP values for each prediction. These are retained as a local explanation in v2.0 since it is deemed that they add value (or at least are not detrimental). Compared to other local explanation (e.g. LIME) SHAP offers a number of useful theoretical guarantees, at the expense of inference time. More details can be found at: <https://github.com/slundberg/shap>

In v2.0 there is a suggested amendment to how SHAP values are displayed however. As with the reliability metrics there is the suggestion of presenting the percentiles within the population, since this provides additional context. An example output may look like:

Table

Description automatically generated

In this example the usual suspect of PAY\_0 = 4 represents the largest SHAP value (most important predictor). By comparing this to the population we see that it is the 66th percentile of the risk population and the 96th percentile of the non-risk population. This suggests that if this example is a “non-risky” example it is somewhat of an outlier in this class.

Note the display above is just meant to be representative. In practice we will need to deal with categorical variables more carefully. This was not performed when prototyping since this is a minor update to an existing method.

**Implementation Comments and Concerns**

As in previous examples the “reference” population for calculating the percentiles needs to be chosen carefully. For simplicity this can be the percentile within the calibration (and or training) set of data used by the model.

As noted above, when productionizing more care needs to be placed on treating non-ordinal categorical variables correctly.

## Nearest Neighbour

Another way to explain a model’s decision is to look at the “closest” labelled instances to the example you are trying to explain. We can do this using a nearest neighbour argument.

The challenge in a method such as this is defining the sensible metric to use to define what we mean by “closest”. This is typically not a trivial task, and changing the metric will result in vastly different neighbours being chosen. This is particularly problematic in our case where we have vastly different scales of features: for example a counting feature might take values between 0 and 10 and a transaction sum feature might take values 0 through to millions. Moreover, even if 2 features have a relatively equal range they might not be scaled similarly: a change of $5000 in one feature might represent a monumental shift in one feature but be inconsequential for another.

The method suggested for explainability v2.0 is to apply a Quantile Transform (Copula Transform) on the feature data. Since this produces an empirical ranking of each feature the “changes” between each feature should be equalized, moreover this is intuitively how we may think about transaction data. For example, moving from the 99th to 99.5th percentile of wealth may be a huge jump in monetary terms (say 100s of millions) we consider these wealths somewhat equivalent to each other. Given this transform we can then apply a simple Manhattan (L1) metric to represent the “distance” between data points. This should produce somewhat sensible neighbours being produced.

An example output of this process may look like:

Graphical user interface, text, application, email

Description automatically generated

Here we can see that the non-risky nearest neighbour is itself a very high probability of risk according to the model. This suggests that if the example we’re looking at is in fact “non-risky” itself it is something of a “difficult” or perhaps “borderline” case.

**Implementation Comments and Concerns**

This should be an easy method to implement. It uses standard sklearn functions and can be evaluated at scale during inference time. For ease of deployment we could fit this method using the calibration set used during model training rather than update the method at each inference batch.

## ALE Revisited

Given we have already calculated the ALE plots as part of the global explainability step – we can repurpose these plots for local explainability and simply overlay the current examples values. This can give additional insight to the user on how the prediction probabilities might change with feature values.

This could look like:

Chart, line chart

Description automatically generated

**Implementation Comments and Concerns**

If the ALE plots are already built for global explainability overlaying a particular example’s feature values should be trivial.