Lehrstuhl für Technische Elektronik
Prof. Dr.-Ing. Dr.-Ing. habil. Robert Weigel
Prof. Dr.-Ing. Georg Fischer

Masterarbeit

im Studiengang
“Computational Engineering (MSc)”

von

Ahmed Shaaban

zum Thema

Explainable Artifical Intelligence in Automotive Manufacturing

Betreuer:	
Beginn:	01.03.2020
Abgabe:	06.10.2020

Contents

List of Abbreviations

ABS	Anti-lock Braking System
EOL	End of the Line
ESP	Electronic Stability Program
ICDATA	I See Data

1 Introduction

1.1 Motivation

1.2 Objective

2 Interpretability

2.1 Introduction About Interpretability and Explainability

3 Machine Learning Framework

3.1 ICDATA Framework

3.1.1 Why ICDATA Framework?

The specific need for the ICDATA framework reinforces the fact that the need for a framework built exclusively to be able to deal with large datasets, massive experiments, and to run them effectively in an appropriate time represent presently a key issue. Accordingly, the ICDATA framework supports us to quickly and effectively train our models on a large datasets and makes it even easier for us to interpret the results out of our models. Maintaining such a framework as a common framework and being used by all developers is considered a key advantage because all of the developers benefit from the framework. According to the authors, it helps standardize the code that is being used and facilitates the process of any necessary modifications to the framework.

Despite the fact that ICDATA deals with a very large datasets and complex experiments, it properly provides the user with the apparent ease of access, and the user does not have to deal with any complex coding challenges. Users need to access a few and basic commands to properly execute a complex and intensive operations. This undoubtedly comes as a direct result of how well the ICDATA framework has been built and interconnected.

The ICDATA framework in common is a framework entirely based on Python [1]. Unsurprisingly, Python was naturally chosen because it is compatible with all the various libraries typically used in Machine Learning and Data Science [2].

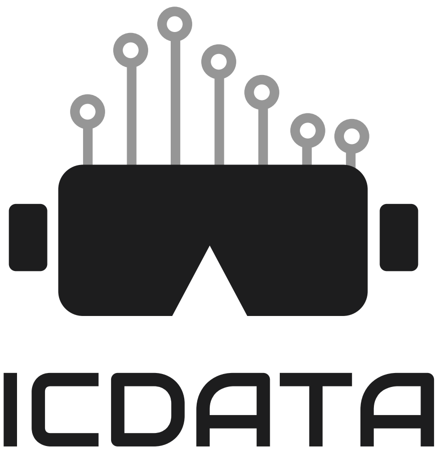

Figure 3.1: ICDATA [2]

3.1.2 Datasets Module in ICDATA

The ICDATA developers have developed the datasets module that comprises several Python classes for various dataset forms. Figure 3.2 display the different parts explicitly associated with the different datasets we have in ICDATA:

- Data source: This is something that can be retrieved by Python, like a NumPy [3] array or a Python list.
- Datasets: It is a Python object that maintains the data source, and it supports splitting and creating batches from it.
- Batches: It is just a piece of the dataset that could be generated as a result of random sampling out of the dataset, or even a precise splitting to produce certain specific instances out of the dataset in a batch form.

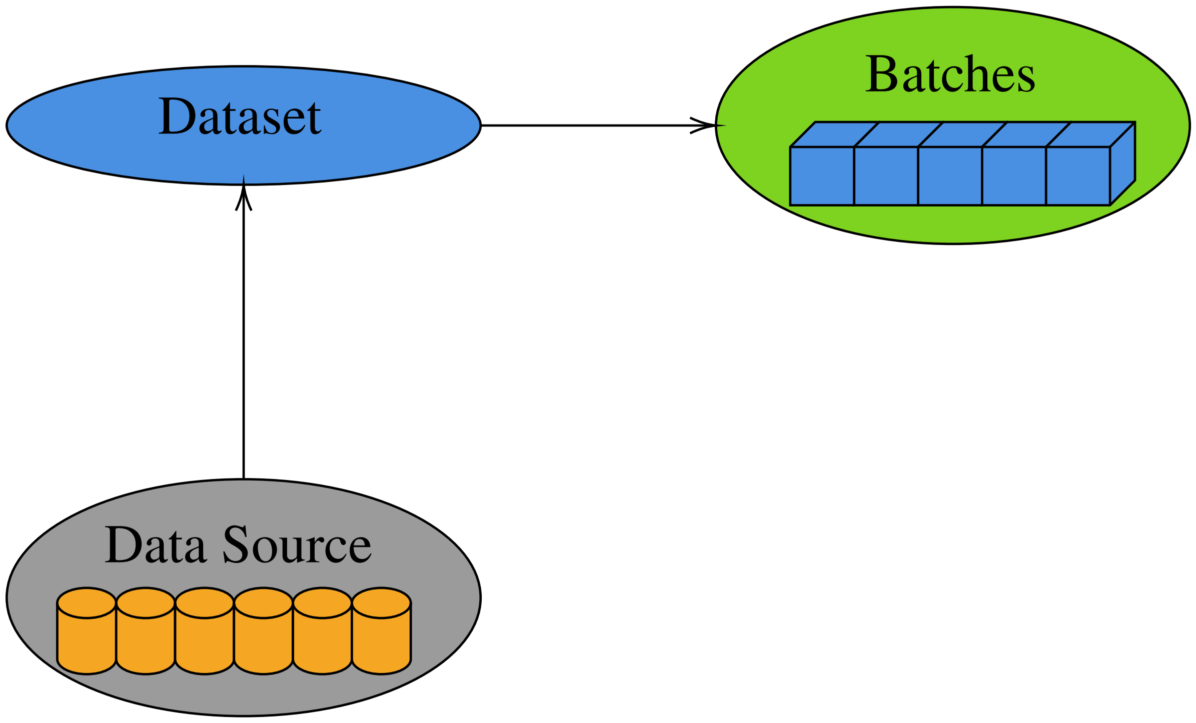

Figure 3.2: ICDATA Data Architecture [2]

In brief, these datasets are primarily focused on a data source that can be accessed in a standardized way, irrespective of whether this data source is PandasDataset or a collection of Images. Subsequently, datasets generate standardized batches that can be used in any model, e.g. for preparation, prediction, explanation.

The variant datasets that the ICDATA datasets module comprises are shown in Figure 3.3.

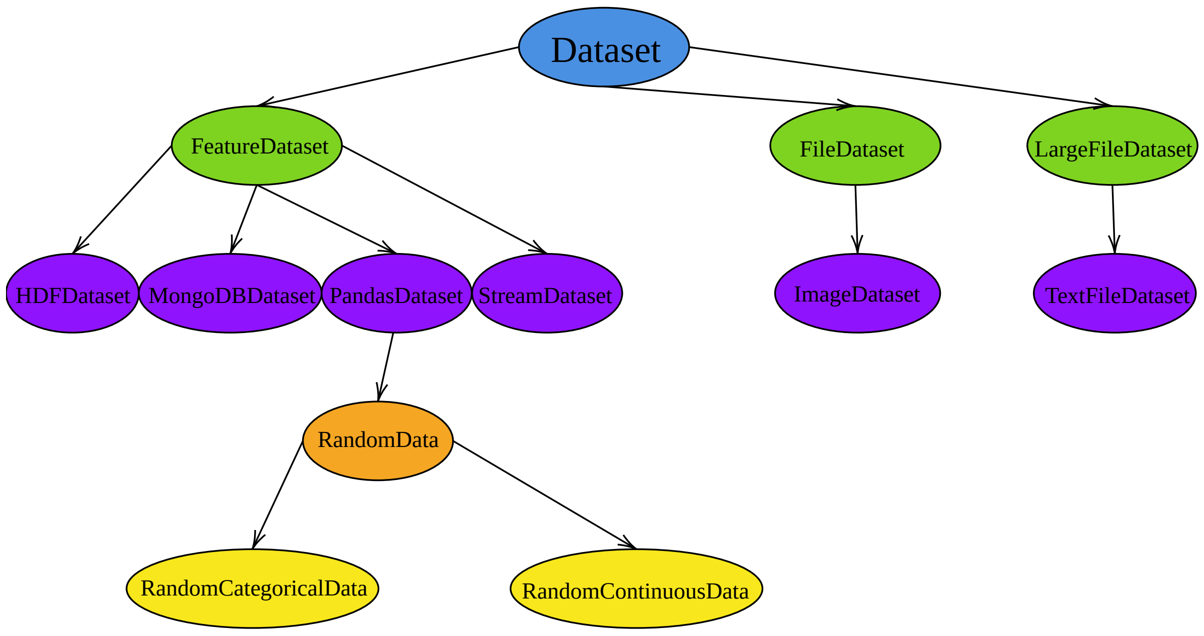

Figure 3.3: Dataset Overview [2]

This thesis concentrates primarily on the **FeatureDataset** and its descendants, the **HDFDataset** [4], and the **PandasDataset** [5]. Hence, let's go further and explain **FeatureDataset**.

FeatureDataset

FeatureDatasets are datasets characterized by having unique columns as their features. For these specific datasets, ICDATA introduces the **FeatureDataset** class, which we additionally manipulate the **FeatureBatch** object from ICDATA when iterating over them.

FeatureDataset is elegantly built in such a way that, through its defined arguments, we can define and provide features in the way we require them. Through it, we can differeniate between:

- **Features**: These are a set of features that should be considered, for example, for model training or batch processing.
- **meta-features**: Those represent features that should be discarded from the model training process, and can alternatively be used during the model evaluation process.
- **weight**: It is the name of the feature that handles the weighting information when such a feature exists.
- **target**: It retains real labels or targets in the case of supervised learning.

PandasDataset is primarily a wrapper for the PandasDataFrame and its two derivatives: **RandomCategoricalData** and **RandomContinousData** they have been extensively used during the test environment that I built to test my implementation of the **Explainer Module** mentioned in section 3.1.5.

A code snippets from [2] that demonstrate how we typically work with **FeatureDataset**.

```
data = MyFeatureData()
# A FeatureDataset, where the user properly identifies the features that the model
    should be trained on and distinguishes them from the meta-features and the
    target features in the case of a supervised Machine Learning task.

data.features
# The features that the model should be trained on

data.target
# The feature used as the correct label in the case of supervised learning tasks.

data.meta
# The meta-features that should be omitted from the model training process.
```

FeatureBatches in ICDATA

To put it concisely, we may claim that the **FeatureBatch** is of considerable value to ICDATA. As stated earlier, everything in ICDATA has been carefully developed to function on a batch basis, for example, all the different models can be trained, predicted, or explained on a batch basis.

In-depth, the **FeatureDataSet** is interconnected with the **FeatureBatch** in such a way that when we generate the **FeatureBatch**, the dependent features (features on which the model is trained) are segregated from the independent features (the label or target in our dataset), and both are differentiated from the meta-features, and the weighting features if they exist [2].

A code snippet from [2] that demonstrates how we deal with the **FeatureBatch** and how it can be produced from the **FeatureDataset**.

```
batch = data.make_iterator(batch_size=None)
# batch_size parameter of the make_iterator function is None, thus, a batch holding
     the whole dataset is returned.

batch.x
# A NumPy array holding the values of the dependent features.

batch.y
# A NumPy array holding the values of the independent features.
```

3.1.3 Processing Module In ICDATA

The processors module is one of ICDATA's most important modules. As ICDATA models are trained, produce predictions, explained correctly on a batch basis. For the general situation, not all of these fundamental processes are conducted on the original dataset, but typically on the processed version of the batches. And here you can reasonably see the notability of conveniently having the processors module. In essence, the module is responsible for properly processing the batches in an optimal way such that the user is capable of adjusting the dataset effectively, which, in turn, will result in models functioning accurately and generating more sensible and reliable results [2].

The processor module consists precisely of many **classes** that retain and identify a considerable variety of pre-processors and post-processors. Pre-processors are processors that utilize a feature as input, and post-processors are processors that support the prediction or probability as input. Figure 3.4 shows the various pre-processors and post-processors present in the ICDATA Framework.

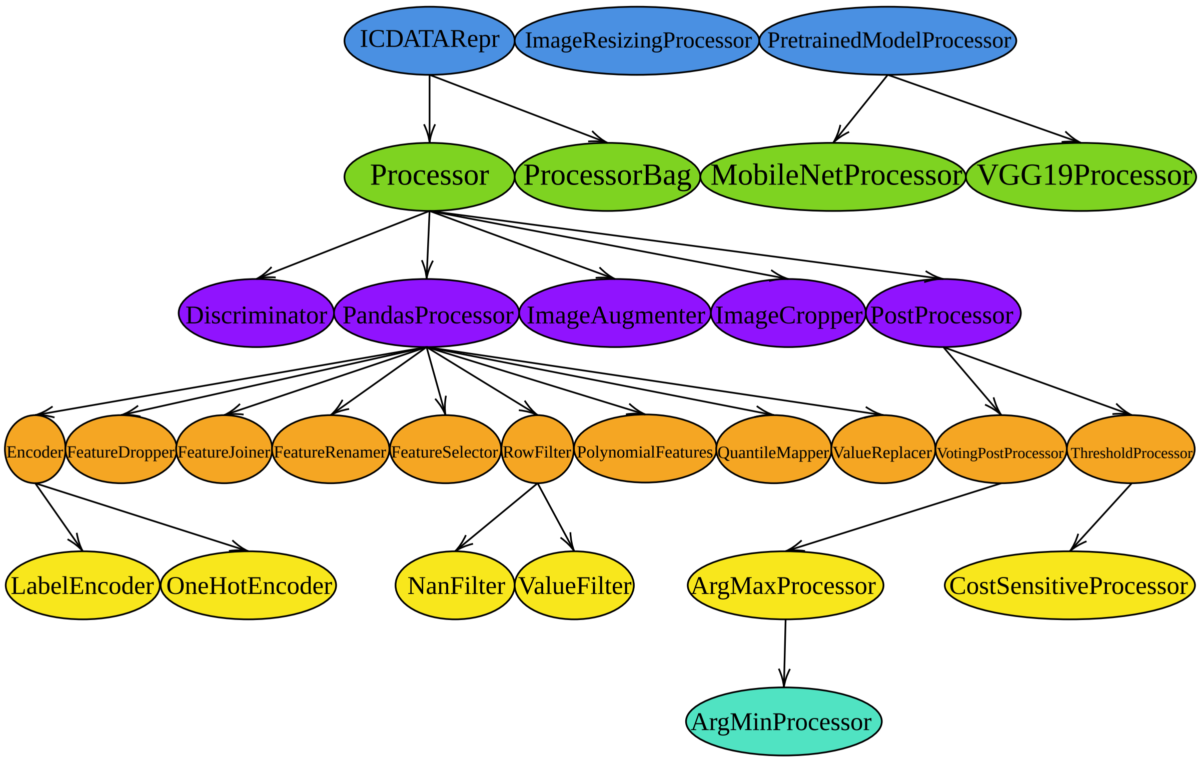

Figure 3.4: Processors Structure [2]

Some Pre-processors

- **OneHotEncoder**: Encode the categorical features as a one-hot encoded feature.
- **LabelEncoder**: Encode the target of the categorical features with a numeric value between 0 and the number of classes minus one.
- **FeatureDropper**: Processor used to remove some of the features from PandasDataFrame.
- **NanFilter**: Processor used to drop rows with nan values.
- **PolynomialFeatures**: Processors used to generate polynomial interactions between specified features.

Some Post-processors

- **ArgMaxProcessor**: Used in the case of multi-class predictions, where the highest probability class is usually returned.

- **ArgMinProcessor**: Used in the case of multi-class predictions, where the lowest probability class is usually returned.

- **ThresholdProcessor**: Transforms the probability to predictions based on a threshold, the predictions are returned if the relative probability is greater than the threshold selected.

Adding Processors to Models

ICDATA processors carry out a specific set of operations to a variety of features that should be precisely described by the user. Processors can be defined during model instantiation, where the user specifies which pre-processors and which post-processors should be used on which features. More than that, the user is allowed to specify the model hyper-parameters used during model instantiation [2].

The following code from [2] shows how processors can be attached to a model.

```python
from icdata.models import XGBoostModel
from icdata.processors import OneHotEncoder

my_preprocessors = [
OneHotEncoder(
feature='some_feature',
categories=['A', 'B', 'C', 'D', 'E']
)
FeatureDropper(
features=['feature_4', 'feature_7']
)
]
# Setup the preprocessors
# Define the preprocessors, and what features they should work on.

my_postprocessors = [
ThresholdProcessor(
label=0,
threshold=0.9,
label_true='ni0',
label_false='i0')
]
# Define the postprocessors, and its threshold.

model = XGBoostModel(
preprocessors=my_preprocessors,
postprocessors=my_postprocessors,
random_state=100)
# Attach them to the model
```

Reverting Processors

Some of the processors used in ICDATA are efficiently implemented in such a creative way that they can be quickly reversed. It would be of considerable help explicitly in the Explanation section mentioned in 3.1.5, where we must revert the OneHotEncoded dataset to the original dataset, and then LabelEncode the dataset to be appropriate for LIME Explainer [6].

The following ICDATA code [2] illustrates how the reverting process can be handled very quickly in ICDATA.

```
processor = MyProcessor()
# Call a pre-defined processor.

batch_processed = processor.apply(batch)
# Apply the called processor on a specified batch.

batch_original = processor.revert(batch_processed)
# Revert the batch to its original status.
```

3.1.4 Models Module In ICDATA

The Model module is a module accommodating the various types of models present in ICDATA. Almost all of the model types available in ICDATA and many of the different dataset types available in ICDATA are shown in Figure 3.5

The concrete manifestation behind this module is that ICDATA aims to provide the user with the ease of access, aside from the complexity of the functions needed to be executed. Consequently, all of the models should be trained, predicted, explained using the same commands, despite, the independent libraries or Machine Learning frameworks on which the models rely [2].

Model Training

As described above, the model training process is carried out on batches. Such batches are usually processed batches, which typically implies that after splitting the dataset into batches, the model pre-processors are applied to these batches, and the model is then trained on a preprocessed batch and not as on the initial dataset.

In the training process which can be summarized in Figure 3.5, the model could be submitted not only with the dataset needed for training, but we can also submit a validation dataset to check losses on both the training dataset and the validation dataset. With the aim of providing completeness and efficiency, the ICDATA training process [2] includes a variety of tasks that can be carried out independtely:

- We can submit the model during the training process, only with the initial dataset.
- Then internally during the training process, the dataset can be split into batches depending on the batch size that we require.
- Aside from the preceding two steps, we may also submit a model with already split train dataset and validation dataset.
- We can also decide about the number of training iterations that we need.

- Then the pre-processors and post-processers attached to the model are applied to the batches.
- Throughout the last step, the model is trained on the processed batches, the loss of the training dataset, and the validation dataset(if used), is retrieved such that we check the success of the model during training and whether or not the model has already been over-fitted.

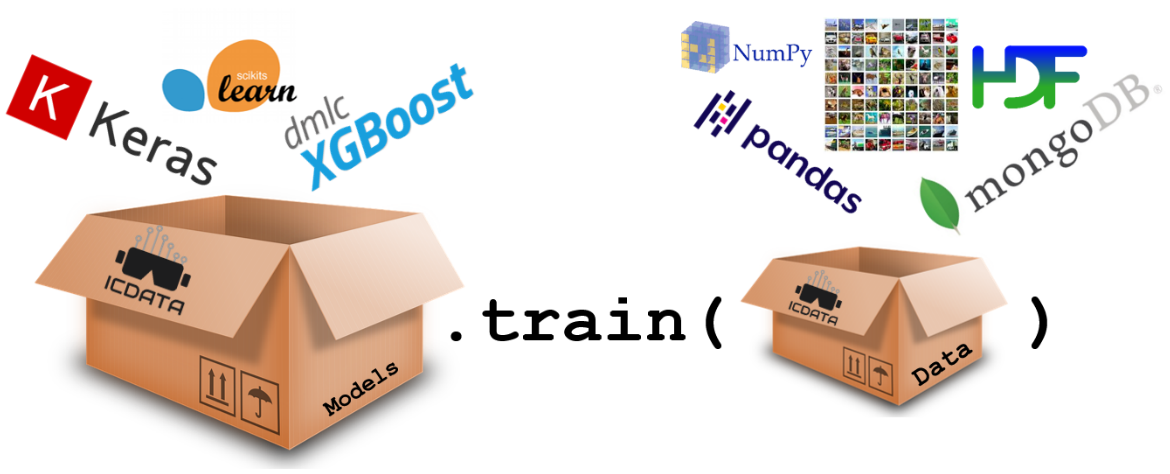

Figure 3.5: Model Training [2]

Model Prediction

The ICDATA prediction method has also been implemented to be used in a reasonable straight-forward manner. The prediction process in ICDATA can be summarized in the following steps [2]:

- Throughout the prediction process, the model can only be submitted with an initial dataset.
- Internally inside the prediction process , the dataset is returned into batches.
- Batches are processed.
- The probabilities and predictions are calculated on the batches processed.
- To adequately provide further flexibility and direct visualization to the user, the results of the prediction process are returned as a PandasDataFrame containing the estimated probabilities, the predictions and the true label of each instance.

Model Persistence

All the time ICDATA typically utilizes massive datasets. It was a crucial obstacle to seamlessly incorporate persistence in all of the models.

This way, all models can be conveniently saved and loaded. The benefit behind this is that the loaded models should not be re-trained and can be specifically used for evaluation and prediction [2].

3.1.5 Explain Module In ICDATA

The explain module was developed into ICDATA to effectively explain batches of data, the batch could be randomly split out of the dataset, or some specific parts collected in an ICDATA batch to be adequately explained by the explainers. **LimeExplainer** based on **LimeTabular-Explainer** [6] and **SHAPKernelExplainer** based on **SHAP KernelExplainer** [7] are the two explainers that can be used to explain any single instance in the used batch. These two particular explainers have been selected as they are model agnostic and can be used to explain any model, so they do not demand much from the models except a prediction function that returns the probability of the instances needed to be explained.

Dealing With Categorical Features

To get a more proper understanding of how and why I get developed the Explain module into the ICDATA framework in its current form. I would like to illustrate, firstly, how the categorical features are dealt with from our perspective, and how the explainers implemented can interact with the categorical features as well.

How We Deal With Categorical Features?

Most of the time the datasets we are dealing with within ICDATA have categorical features. Categorical features support various approaches to deal with them and how they should be properly handled.

In my specific case, the tabular dataset used in the use-case I received from Bosch has in common categorical features, and the proper way we worked with it, implied that we have transformed all of the categorical features into OneHotEncoded versions.

How LIME Deal With Categorical Features?

Having realized we are transforming all of our categorical features into OneHotEncoded form, typically the fundamental question that is actually blowing up in our heads, so how does LIME cope adequately with the categorical features as well?

Ordinarily having a tabular dataset, I have no possible choice except to naturally use **LimeTabularExplainer** to adequately explain our dataset. As a result, I looked at how **LimeTabular-**

Explainer deals with categorical features. I found that **LimeTabularExplainer** can only take a numerical dataset, it cannot accommodate dummies or strings. Consequently, all of the categorical features must be assuredly LabelEncoded in order to convert all of the strings we have into integers [6].

Logically, we may now wonder how the Explainer can keep up or faithfully relate to the categorical features in the explanations it provides?

The meaningful response to that comes from the magic parameter **categorical_names**, which is a dictionary containing the index of the categorical features in our dataset, and a list of the categories available in each of our current categorical features. This parameter properly preserves the direct relationship between the integer values in our dataset after it has been LabelEncoded and the initial strings. Such that, later on, the explainer can present that in its explanation [6].

It may as well rise in our minds now, all right, there is another key issue. The model, as described in the training section 3.1.4, is trained on processed batches, which will definitely have OneHotEncoded features. The model is therefore trained on a processed dataset, including OneHotEncoder. When the Explainer receives a LabelEncoded version of the dataset. Therefore, there is no correlation between the model and the explainer. Then the explanations from the explainer are unright?

Satisfactorily, the proper response to this is, of course, that the model needs to be adequately trained on the processed version of the dataset, and if we have the categorical features that are OneHotEncoded, then the model should be trained on this processed version of the dataset without any other choice, as not to treat the categorical features as continuous features. Also, after our **LimeTabularExplainer** is ready, the **explain instance** of LIME, which carefully provides explanations from the explainer, is provided with the model **prediction function**.The **prediction function** that determines the model's probability is built in a way such that it maintains the same version of the dataset that the model sees or is trained on. As a consistent result, there is no observed discrepancy between the probability that we will get out of the model and how the model is trained. Nevertheless, the explainer itself cannot include any OneHotEncoded features, because it must specifically verify that a categorical feature has just one value [6].

A lot more about **LIME**, its parameters, how it functions exactly, and how it has been implemented precisely, can be perceived in the subsequent section 3.1.5.

How SHAP Deal With Categorical Features?

No precise handling of the categorical features is given by the **SHAP KernelExplainer**. In line with this, we could also merely pass the **KernelExplainer** with the processed version of the dataset, even if it includes OneHotEncoding. The other consequence of that is at ease that each category out of the same feature is being typically handled as a separate feature after OneHotEncoding, of course, that increases the computation time as opposed to the case if we considered grouping of the features categories [7].

The model **prediction function** used by the **SHAPKernelExplainer** has been properly implemented in such a way that the processed version (e.g. OneHotEncoded) of the dataset can be accepted directly.

A lot more about the **SHAPKernelExplainer**, its parameters, how it was developed, how it works, how its explanations are calculated, is provided in section 3.1.5.

Sum Up

From the information already mentioned. It has been decided that we are going to implement a **preprocess** function that LabelEncode our dataset to be suitable for LIME, and a **postprocess** function that transforms our LabelEncoded dataset to OneHotEncoded version to be suitable for the model prediction function.

We should also be capable to note that the **SHAP KernelExplainer** does not need such two functions. They can additionally be utilized, nevertheless, because they represent a step and its inverse, and it was agreed to add both **preprocess** and **postprocess** functions to the parent class, so it is up to the user to employ them while using the **SHAPKernelExplainer**.

Structure Of the Explain Module

The ICDATA Explain module as show in Figure 3.6 is based on an **Explainer** parent class, with the parent class having certain functions required by the **LimeExplainer** or **SHAPKernelExplainer** child classes. These functions are init, preprocess, postprocess, build, and build_explainer.

- **init function**: The init function receives the trained ICDATA model used, and we also attach some of the necessary functions and variables from the model to the explainer.
- **preprocess function**: The preprocess function is of particular significance to the **LimeExplainer** class because **LimeTabularExplainer** can not accept the OneHotEncoded features that we may provide when we have categorical features in our dataset. Alternatively, we need to LabelEncode the dataset and sent this LabelEncoded version of the dataset to the **LimeExplainer** class, where the LavelEncoded dataset is then sent to the **LimeTabularExplainer**.
- **postprocess function**: Postprocess is considered to be the reverse of the preprocess function. Postprocess function is also of great importance to the **LimeExplainer**, as the model itself is trained on the processed version of the dataset, including the OneHotEncoded features, if available. As a result, the model's **prediction function** should be expected to have the same processed version of the dataset, which is why the postprocess function is used to convert the dataset back to the same coded version as the one the model uses.
- **build and build_explainer functions**: The **build_function** receives a processed version of the dataset in a batch, then only preprocesses it to create a **LabelEncoded** version of

the data available in the batch. It also receives kwargs which should be used as values for the different parameters of each of the explainers. Finally, it calls the **build_explainer function** that applies the kwargs to either of the **LimeExplainer** or the **SHAPKernel-Explainer** and returns the required explainer.

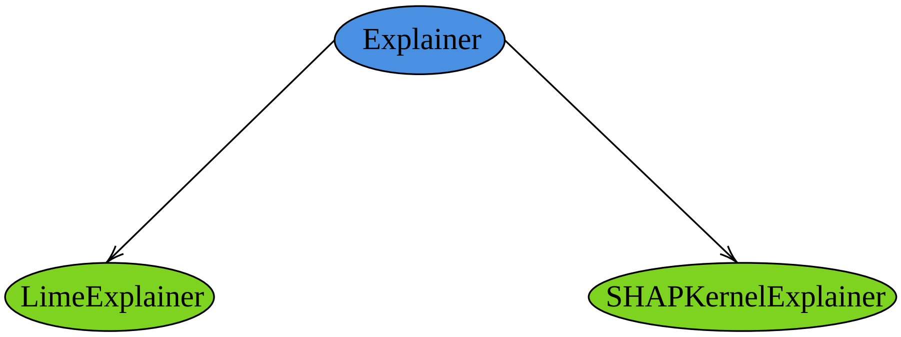

Figure 3.6: Explainer Module Classes

LimeExplainer Structure

Before discussing how the **LimeExplainer** has been implemented. Let's review some of the essential parameters of the **LimeTabularExplainer** and the LIME **explain_instance** shown in table 3.1 and table 3.2 respectively to make it simpler why these functions have been implemented in the **LimeExplainer** class.

LimeTabularExplainer	
training_data	This will be a NumPy 2d array, reflecting the data on which our Explainer is trained.
training_labels	This defines the exact labels of the instances sent to the training data parameter in the 2d array.
categorical_features	A list that holds the indices of the columns of the categorical features.
categorical_names	A dictionary containing the indices of the column's categorical features as keys, as well as the different categories present under each of these columns categorical features as values.
discretize_continous	If True, all non-categorical features would be discretized into quartiles.
discretizer	Let us pick the appropriate discretizers from the different discretizers available in LIME.

Table 3.1: Some of LimeTabularExplainer Parameters

explain_instance	
data_row	Represents a single row or a particular instance that needs to be explained.
predict_fn	The prediction function of the model used should take the NumPy array and return the prediction probabilities.

Table 3.2: Some of the Explainer explain_instance Parameters

Now let's continue explaining how **LimeExplainer** has been implemented, and we'll refer to the preceding parameters while demonstrating.

Figure 3.7 shows the various functions implemented in the development of the **LimeExplainer** class in the ICDATA framework. And it could be marked that, it uses four functions of the parent class Explainer, which is:**init**, **preprocess**, **postprocess**, and **build_explainer**.

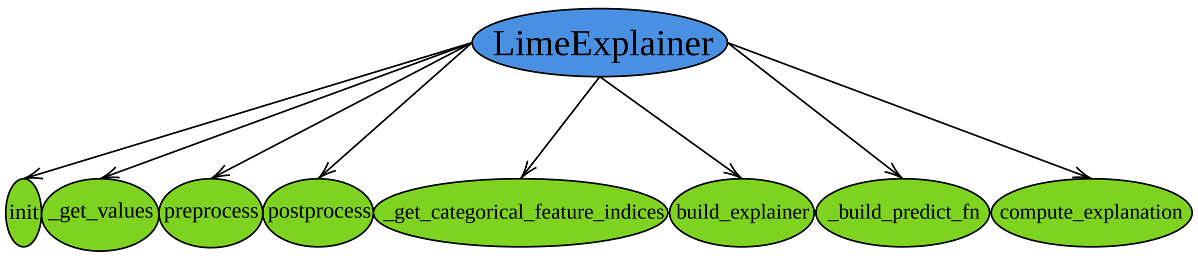

Figure 3.7: LimeExplainer Structure

init function

In addition to what is already stated in the **init** function 3.1.5 of the parent class **Explainer**, the init function in the **LimeExplainer** class examines how many OneHotEncoders we have added to our features, and equally prepares LabelEncoders to operate on those features and saves these encoders to variables attached to the Explainer. These variables containing the Encoders should be included in the preprocess 3.1.5 and postprocess 3.1.5 functions, that we will shortly highlight.

_get_values function

This function is used to return a list of valid categories in each of our categorical features. It receives a string that is the name of the categorical feature and returns all of the categories present in it. The output of this function is required to prepare the values to be submitted to the **LimeTabularExplainer categorical_names** parameter.

preprocess function

The general justification for the need for a **preprocess** function is explicitly mentioned in the 3.1.5. In-depth, since we are currently providing OneHotEncode for all of our categorical features, this function is explicitly used to convert all of our OneHotEncoders prepared from the init function, so if we obtain the initial unencoded version of our dataset, it will convert it back to the LabelEncoded version, so that it would be appropriate for what the **LimeTabularExplainer** intends. Also, turn to 3.1.3 to see how OneHotEncoding is reversed, and instead LabelEncoding is used.

The output of this function is a LabelEncoded batch and referring back to the FeatureBatch in 3.1.2, we will clearly see that the values to be added to the **LimeTabularExplainer training_data** parameter are **batch.x** (notice batch here for the LabelEncoded batch), and the value passed to the **training_labels** parameter is **batch.y**.

postprocess function

The general justification for why the **postprocess** function is required is also specified specifically in the 3.1.5. This function does exactly the inverse of the **preprocess** function. It converts all of the LabelEncoders that we have already prepared in the init function back to the original non-encoded version by using the inversion of processors mentioned in 3.1.3 and applies all of the OneHotEncoders that are already prepared from the init function so that we have the same version of the encoded dataset that the model is trained on.

The latest encoded version of the dataset is employed by the model's **prediction function**. Therefore, the model sees the same representation of the instances like the ones upon which probabilities are calculated.

_get_categorical_feature_indices function

Simply this function is used to return the index of the categorical features in the batch submitted to the **LimeExplainer** to be explained. Its output is used as an input to the **LimeTabularExplainer categorical_features** parameter.

build_explainer function

As stated briefly in the Explainer class section 3.1.5, the **build_explainer** function is used to return the **LimeExplainer**. To make it clearer, this function remains the one responsible for generating the Explainer in its final version. As a consequence, it receives output from all of the other functions, prepares all of the data in a suitable manner to what is required for each parameter in the **LimeTabularExplainer**. Latterly, it transmits the prepared values to each of the parameters in the **LimeTabularExplainer** and returns an operational version of the Explainer.

For example, this function takes the output of the **categorical_feature_indices** function 3.1.5 and the output from the **get_values** function 3.1.5 to process them together and returns the dictionary of the **categorical_names** in the way the **LimeTabularExplainer** expects this parameter.

_build_predict_fn function

The **prediction** function is the only function that LIME needs from the model. It will be employed by LIME **explain_instance** function to provide explanations for every instance needed to be explained. As stated earlier, in the **postprocess** function 3.1.5 the **prediction** function should see the final coded version of the features as the one, the model sees. And as LIME **explain_instance** function that uses the **prediction** function always has to be submitted by label encoded version of the dataset, and to preserve the constraint of consistency between the **model** and the **prediction** function, we use the **posprocess** function internally within the **prediction** function to transform the data (in the form of a batch) into the latest encoded version as the model.

compute_explanation function

Once we have the explainer ready from the **build_explainer** function, we can move forward and start explaining the instances we need to explain. It should be mentioned that the instances to be explained here are also LabelEncoded, and they are sent in the form of a batch, so we are going to iterate over each instance in the batch. LIME grants us this functionality of explaining the required instances which are submitted as batch.x (as in 3.1.2) to the **data_row** parameter of the **explain_instance** through the **explain_instance** function. That is why the **compute_explanation** function is designed to return the output of LIME's **explain_instance** function after allocating its parameters with the appropriate values.

SHAPKernelExplainer Strcture

It was decided to implement the **SHAPKernelExplainer**, since it is gladly a model agnostic explainer just like **LIME**, it can be utilized to properly explain any model, and it doesn't require anything like LIME from the model other than a prediction function. As seen from Figure 3.8, **SHAPKernelExplainer** does not use many of the functions of the **Explainer** parent class as LIME, explicitly, the **preprocess** and **postprocess** functions are no longer necessary. Theoretically, they can be employed as they are, indeed, a step, and it is inverse, but then they will be considered excessive or unnecessary steps.

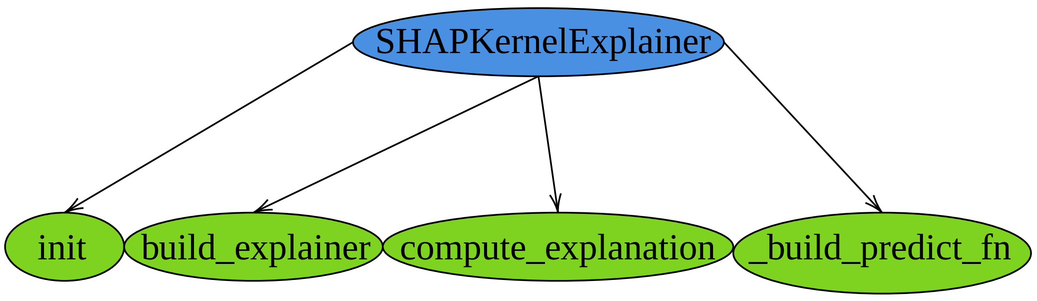

Figure 3.8: SHAPKernelExplainer Structure

init function

Same as stated in the **init** function of the parent class **Explainer** 3.1.5.

_build_predict_fn function

Same as in the case of LimeExplainer, the **predict** function is naturally used to return the model probabilities of the specific instances presented for the explanation. The key difference between the **predict** function used in **LimeExplainer** and the **prediction** function used in **SHAPKernelExplainer** is properly that the **prediction** function does not have a **postprocess** function step in **SHAPKernelExplainer**. Alternately, it directly receives instances in the same coded version of the model. To simplify, both the model and the model **prediction function** are provided with the same processed version of the dataset, for example, both of which already receive the OneHotEncoded version of the dataset.

build_explainer function

From a general perspective, the **build_explainer** function is conveniently used to retrieve a working version of the **SHAPKernelExplainer**.

In precise detail, it typically receives a processed batch of data. Take a subset out of this batch to be used as the background data needed by the **SHAPKernelExplainer**. As well as presenting the **SHAPKernelExplainer** with the already built model **prediction** function and forwarding the appropriate determined values to the valid Explainer parameters. Ultimately, the Explainer is returned to be used in further steps.

compute_explanation function

The **compute_explanation** function is used to return the shap values for each instance in the batch sent to it. As a result, the **shap_values** function from **SHAP KernelExplainer** is used internally to generate the corresponding shap values. Also, it passes the chosen values to certain parameters required for the estimation of shap values, such as the nsamples parameter, which specify the number of times the model will be re-evaluated during the explanation of each prediction.

Accessibility Of the Explainer Module

In the previous subsection, I intentionally tried providing a quick description of how the Explainers have been implemented in the ICDATA framework, and what functions they rely on, and how these functions interconnect with each other to properly provide a functional version of the Explainers, which gives an explanation of the specific instances in the batch presented to these Explainers.

As mentioned in the ICDATA section 3.1.1. The real objective behind ICDATA is to provide the user with remarkable ease of access. The user does not need to be aware of the functions and specific details I prominently mentioned in the previous subsection. Ideally, he can choose the Explainer with whom he is willing to explain the instances in the batch and obtain his results only by considering two functions that are directly linked from one side to the used model and from the other to the required Explainer class. These two functions as show in Figure 3.9 are: **create_explainer**, and **explain**.

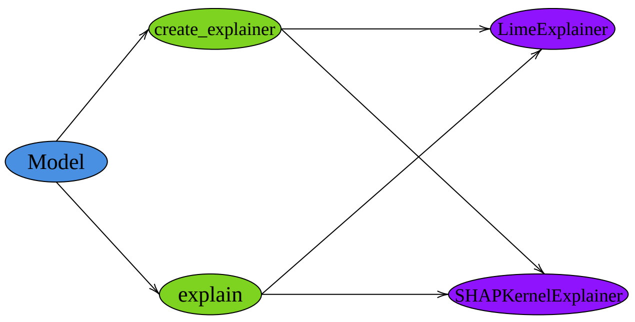

Figure 3.9: Accessibility in the Explainer Module

create_explainer function

The **create_explainer** function is the only general function the user has to access to create the required Explainer. As shown in Figure 3.10, the user can pick either **LimeExplainer**, which is the default Explainer or **SHAPKernelExplainer**, using the **”explainer”** parameter of the function. Additionally, the user can submit the kwargs required to be forwarded to either of the explainer using the **”kwargs”** parameter of the function. The **kwargs** will be sent automatically to the chosen Explainer afterwards. The function equally receives the **dataset** that we need to train our explainer on, **process** it and returns it in **batch** form to be adequate to the **build** function of the **Explainer** parent class, which in return calls the **build_explainer** function of the required Explainer and submits it with the processed batch and the kwargs.

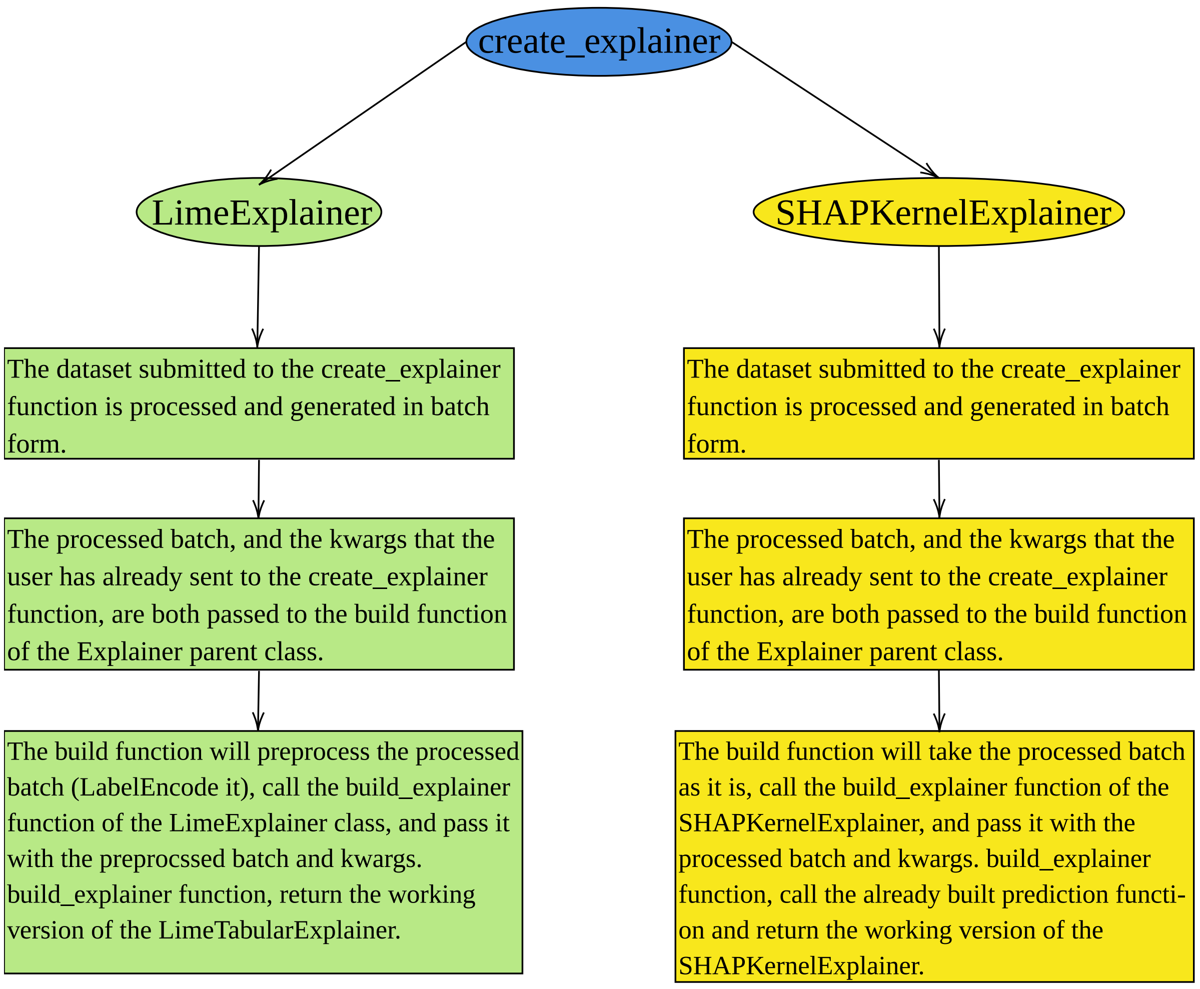
create_explainer
LimeExplainer
SHAPKernelExplainer
The dataset submitted to the create_explainer function is processed and generated in batch form.
The processed batch, and the kwargs that the user has already sent to the create_explainer function, are both passed to the build function of the Explainer parent class.
The build function will preprocess the processed batch (LabelEncode it), call the build_explainer function of the LimeExplainer class, and pass it with the preprocssed batch and kwargs. build_explainer function, return the working version of the LimeTabularExplainer.
The dataset submitted to the create_explainer function is processed and generated in batch form.
The processed batch, and the kwargs that the user has already sent to the create_explainer function, are both passed to the build function of the Explainer parent class.
The build function will take the processed batch as it is, call the build_explainer function of the SHAPKernelExplainer, and pass it with the processed batch and kwargs. build_explainer function, call the already built prediction functi-on and return the working version of the SHAPKernelExplainer.

Figure 3.10: create_explainer Function Workflow

Persistence of the Explainers

Based on the persistence criterion used in ICDATA 3.1.4. The Explainers have been implemented in ICDATA such that they can be saved after they have been called and trained. The Explainer save is accomplished after the **create_explainer** function is finished and our Explainer is returned. The saved Explainers will not have to be called again or trained again, we can just load it and have our Explainer ready to explain any instances we need.

explain function

As shown from Figure 3.11 the explain function is submitted with the dataset that we need to explain its instances, and the kwargs needed to be sent either to the **shap_values** function of SHAP, or to the **explain_instance** function of LIME. After receiving the dataset from the user, the function internally processes it, returns it as a batch. After that, the processed batch and kwargs are sent to the **compute_explanation** function of the explainer already selected in the **create_explainer** function step.

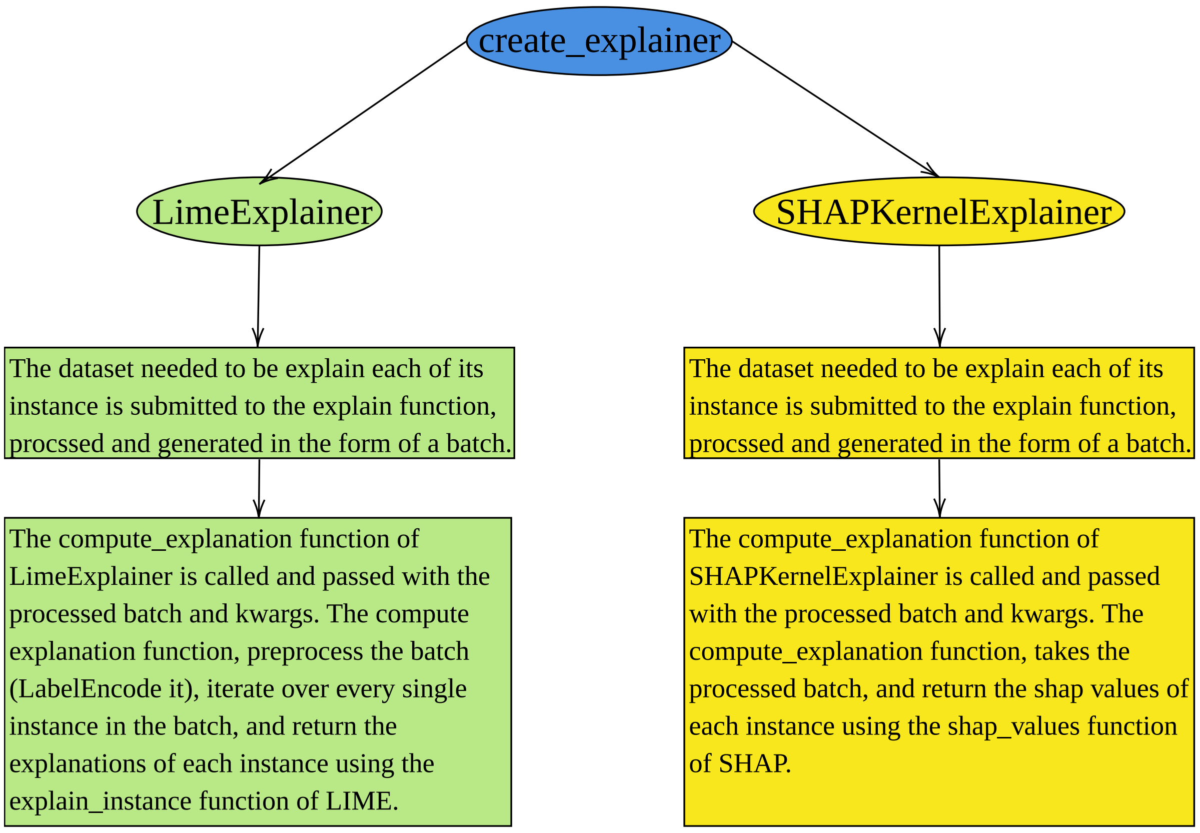
create_explainer
LimeExplainer
SHAPKernelExplainer
The dataset needed to be explain each of its instance is submitted to the explain function, procssed and generated in the form of a batch.
The dataset needed to be explain each of its instance is submitted to the explain function, procssed and generated in the form of a batch.
The compute_explanation function of LimeExplainer is called and passed with the processed batch and kwargs. The compute explanation function, preprocess the batch (LabelEncode it), iterate over every single instance in the batch, and return the explanations of each instance using the explain_instance function of LIME.
The compute_explanation function of SHAPKernelExplainer is called and passed with the processed batch and kwargs. The compute_explanation function, takes the processed batch, and return the shap values of each instance using the shap_values function of SHAP.

Figure 3.11: explain Function Workflow

3.1.6 Full Example Using XGBoost Model

I used the XGBoost [8] model for my entire job. XGBoost in common is the most widely used model in ICDATA and was the model used in the use-case [9] I worked on, the use-case is discussed in depth in section 3.2.

It is not shocking to see that XGBoost is the most commonly used model in ICDATA, because XGBoost is extraordinarily efficient, scalable, and portable. It is well-recognized for its speed and reliability as it performs professionally Machine Learning tasks using the gradient boosting framework, and this preciously what naturally makes it the state-of-the-art in many of the Machine Learning tasks. Gradient boosting [10] is essentially a supervised learning algorithm that determines the target variable by summing up the results of a collection or a set of simplified and weaker models.

XGBoost is well known to be used for massive datasets and specifically tabular datasets, while it is more known that deep learni ng methods are used for image and text datasets. It is therefore selected as the most adequate optimal model to deal with the use-case and tabular dataset that I have worked on.

In this specific section, I will include some ICDATA code snippets [2] that highlight how we deal with our data and how we deal with our model during training, prediction, processing, and explanation.

Dataset Initialization

Introduce how to adequately deal with dataset and its various functions as generating batches from it, and how splitting is done. The dataset used here is a random dataset where all the features are categorical.

```
from icdata.datasets import RandomCategoricalData
data = RandomCategoricalData(
  n_features=20,
  n_points=1000,
  random_state=10)
# Initialize our Dataset
```

Generate Batches Out Of the Dataset

```
iterator = data.make_iterator(batch_size=200)
# Split the dataset into batches, with each batch size of 200 samples. The last
    batch may be smaller than the prior batches.
```

Dataset Splitting

```
data_train, data_test = data.split(
  test_size=0.2,
  random_state=12,
  stratified=True
)
# Split the dataset into the train portion and test portion.
```

```
data_train, data_val = data_train.split(
  test_size=0.1,
  random_state=12,
  stratified=True
)
# We can further divide the train portion into portions of the train and the
    validation.
```

Model Instantiation

We will introduce our model and attach it with the necessary **pre-processors** and **post-processors**.

```
from icdata.models import XGBoostModel
from icdata.processors import OneHotEncoder
from icdata.processors import NanFilter


model = XGBoostModel(
  preprocessors=[
    OneHotEncoder(
      name='Encoder',
      feature="AAA",
      categories=[0, 1]),
    NanFilter()
  ],
  random_state=30,
  )
# The OneHotEncoder is used for the specific "AAA" feature, with its categories
    defined.
# NanFilter is attached to the model to remove all rows with nan values.
# The random state needed to be used with the model can also be attached to the
    model during model instantiation.
```

Model Training

In the last step of model instantiation, and immediately before starting the training of our model, we will transmit the hyper-parameters required to be used by the model.

The following train the model for 100 iterations, each of which has 10 boosts for 30 randomly sampled data points. After every 5 iterations (i.e. 300 boosting rounds) the area under the curve is computed using the validation dataset [2].

```
from icdata.scoring.metrics import AUC
from icdata.parameters import HyperParameters
from icdata.monitoring import Monitor

params=HyperParameters(
  alpha=0.0,
  max_depth=5,
  eta=0.1,
  num_boost_round=10,
  objective='binary:logistic',
  sampling=True,
  batch_size=30,
  n_iterations=100
  )

# Define the hyper-parameters of the model.

model.set_params(params)

# Set the hyper-parameters to the model.

monitor = model.train(
  data=data_train,
  data_val=data_val,
  validation_steps=5,
  metrics=[AUC()],
  monitor=Monitor()
  )

# Begin the training process of the model by simply calling the train function and
    passing it with the appropriate parameters.

# Be aware that even though the model has been passed with data_train. The model
    train function itself internally adjusts the data_train into a batch, then
    processes it properly, then the model begins to be trained on this processed
    batch, and the loss results are calculated on batch basis as well.
```

The monitor returned instance contains the **loss_train** and **loss_val** dictionaries mentioned eariler in section 3.1.4, to help check the peformance of our model during the training process.

The **monitor.loss_val** presents results as shown here:

```
{
5:  {'AUC': {0: 0.4609130706691682}},
10: {'AUC': {0: 0.4996873045653534}},
15: {'AUC': {0: 0.5015634771732334}},
20: {'AUC': {0: 0.6041275797373358}},
25: {'AUC': {0: 0.5165728580362727}},
30: {'AUC': {0: 0.5428392745465916}},
35: {'AUC': {0: 0.4959349593495935}},
40: {'AUC': {0: 0.5628517823639775}},
45: {'AUC': {0: 0.5209505941213259}},
50: {'AUC': {0: 0.5040650406504065}},
55: {'AUC': {0: 0.5084427767354597}},
60: {'AUC': {0: 0.5528455284552846}},
65: {'AUC': {0: 0.5384615384615385}},
70: {'AUC': {0: 0.5453408380237648}},
75: {'AUC': {0: 0.5240775484677924}},
80: {'AUC': {0: 0.5572232645403377}},
85: {'AUC': {0: 0.49530956848030017}},
90: {'AUC': {0: 0.5146966854283928}},
95: {'AUC': {0: 0.47842401500938087}}
}
```

The model started to overfit in iteration 20, almost after 200 boosting rounds.

Model Persistence

Once the model has been trained, we can currently save our model and load it to be used for prediction or evaluation.

```
model.save('/path/to/model/')
```

The model can be loaded again as follows.

```
model = model.load('/path/to/my_model/')
```

Model Prediciton

Once we obtain a model that we can access and have already been trained, we can employ it to retrieve the probabilities out of a required dataset. It should be noted that the predict function operates in the same way as the training process. It is then sent with a dataset, adjust it into a batch, process the batch, and return the probability for each instance in the batch.

```
for prediction in model.predict(data, batch_size=100, max_batches=200):
  prediction.probabilities
# We call the predict function, send it with the required data to be predicted, and
    the required parameters.

# After that, we access the probabilities for each instance in each of the returned
    batches.
```

Creating the Explainer

After the model has been trained, and we are satisfied with its results. The subsequent step is to start the process of explaining our model. The very first step in the explanation should be begun by calling the **model.create_explainer** method. The **create_explainer** returns the necessary Explainer, which could be **LimeExplainer** or **SHAPKernelExplainer**. The user only submits the **create_explainer** function with a dataset or a dataset split on which the Explainer should be trained. The user is free as well to submit the function with the kwargs that he would like to set to the **Explainer**, otherwise the default values will be set to the parameters of the required **Explainer**.

```
model.create_explainer(data_train)
```

Getting the Explanations

We have got the Explainer ready, so we can right now get explanations for the required instances or a specific submitted dataset. This can be granted to us by the **model.explain** function.

Briefly, the **model.explain** function returns the results from the **explain_instance** function in case the **LimeExplainer** is selected or returns the results from the **shap_values** function in case the Explainer is **SHAPKernelExplainer**.

As with the **create_explainer** function, the user is free to set values to the parameters of either the **explain_instance** function or the **shap_values** function by sending their kwargs through the **model.explain** function, otherwise, the default values will be set.

```
model.explain(data_test)
```

3.2 HU9 Use-case

3.2.1 Origin of the Use-Case

The use-case on which I am working is implemented out of the production lines of the Electronic Stability Program (ESP), the data are tabulated from three separate production lines.

The ESP shown in Figure 3.12 is a scalable product concept for full skidding protection. The marketable ESP product from Bosch is recognized for the modular design of its hardware and software. As a result, distinct variants can comfortably accommodate diverse types of cars, typically ranging from small cars to premium cars, to light trucks. This shows up in the interests of the consumers because they will select the program that better fits their cars, as well as what unique functionality and features they expect to be included and incorporated in their vehicles [11].

The ESP is well known for assisting the driver in all crucial driving conditions. It is attributable not only to the ability to sense and prevent vehicle skidding movements but also to the fact that it encompasses the functionality of the Anti-lock Braking System (ABS) and the traction control systems. Addtionally, the ESP control unit also compares the realistic movement of the vehicle 25 times per second with the desired travel direction. Throughout the desired end, this direct sum in progressively improving driving safety [12].

It is observed in practice that up to 80 percent of all skidding accidents can be avoided by the ESP [12].

Product Benefits [11]

- Scalable product design with the highest flexibility.
- Low weight and small box size.
- Very good pedal feeling and very low noise.
- Provides valuable help in the critical conditions.

Figure 3.12: ESP [11]

3.2.2 Testing in the Production Lines

At the end of each production line, there is always a test done for all the pieces produced from this line. This is usually referred to as the End of the Line (EOL) test.

The purpose of the EOL test is to check whether the parts are OK and can be sold or NOT OK, so we can send them to the trash. In particular, for the ESP unit, it is necessary to check that its valves are tight enough and that no leakage is observed.

The EOL test is not a single-step test. However, it is a four-step process to be accurate and effective. A part that fails the test once is required to be checked again for three more times, and if the part passes any of these checks, it will be sold normally. There is no doubt that further tests on the production lines require more time, and that is where the role of the model comes in. After the first failure of any part on the production lines, the model is used to predict whether this particular part is likely to be turned into an OK part in any of the remaining test attempts if there is no hope, we simply have to throw it away [9]. A visualized overview of the test process on the production line is seen in the Figure below. 3.13

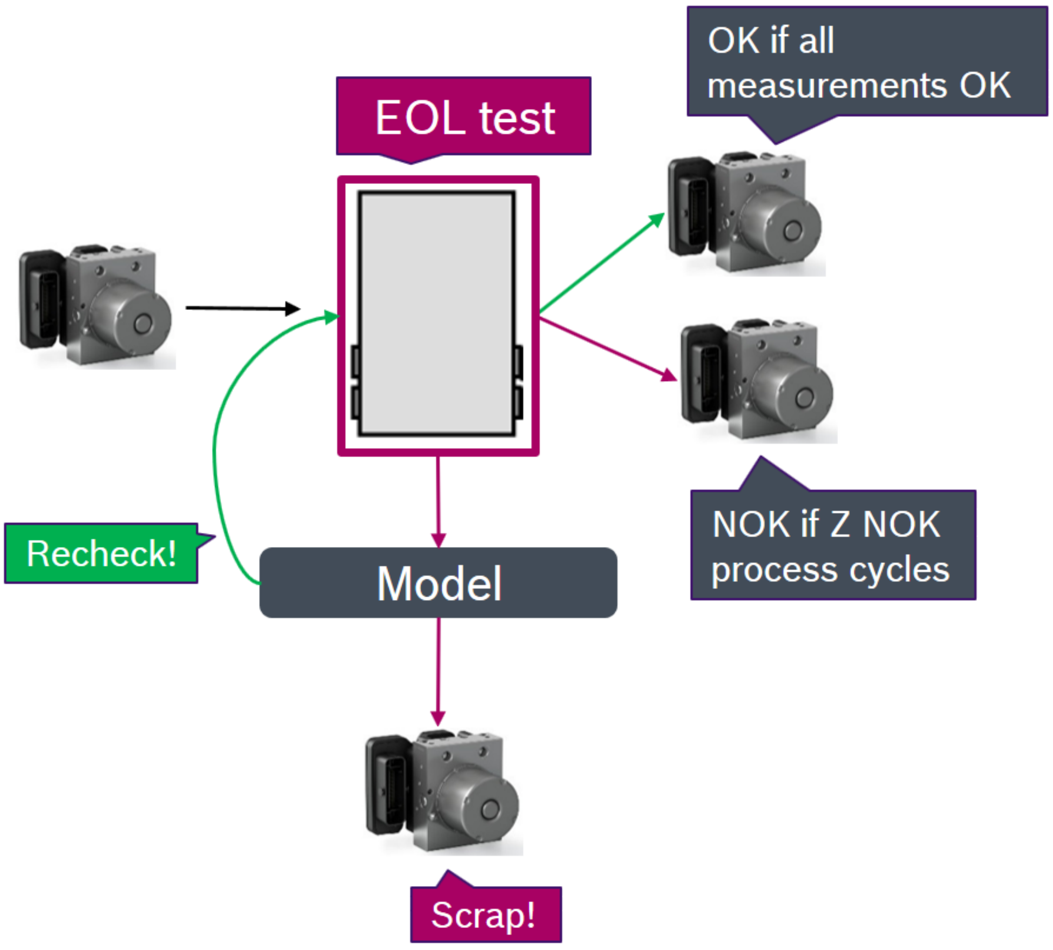

Figure 3.13: Testing Process [9]

3.2.3 Use-Case Overview

As mentioned above, our ESP units may be OK or NOT OK. Therefore, our use-case is a classification process for two classes that are OK class and NOT OK class.

Dataset Overview

The initial version of the dataset used in the use-case comes with a total of 322 features. These features are split into different categories. 311 features will be used as explicit features of our model, such that they can be used in the training of our model. We have one feature as our target feature, which basically holds the true label for each of our parts, and shows whether this particular part is OK or NOT OK. In addition, we have 10 meta-features that will be exculded from the training of our model and used for the model evaluation or debugging [9].

In short, the explicit features of the use-case may be further divided into different groups, some of which relate to:

- Pressure.
- Volume.
- Voltage.
- Timing.
- Leakage.

The initial version of the dataset is not the version of the dataset used by the model, the model only relates to the processed version of the dataset. As a consequence, we must initialize our dataset as already specified in 3.1.2 and then process it. All about dataset processing is stated precisely in the following subsection 3.2.4.

Model Overview

The XGBoost model is the model used in the use-case [9]. The model comes with a set of hyperparameters and a set of pre-processors and post-processors. These pre-processors and post-processors operate on a batch from the dataset, when, for example, we present the model with a dataset to be trained on.

The hyperparameter set used for the model is as follows:

- tree_method: exact.
- sampling: False.
- n_iterations: 1.
- batch_size: None.
- seed: 42.
- silent: 1.
- nthread: 8.
- num_boost_round: 320.

- objective: multi:softprob.
- reg_alpha: 0.8.
- reg_lambda: 0.2.
- colsample_bytree: 0.8.
- eta: 0.05.
- gamma: 0.1.
- max_delta_step: 0.5.
- min_child_weight: 1.
- max_depth: 10.
- scale_pos_weight: 1.
- subsample: 0.5.
- num_class: 2.

More information about the XGBoost model parameters can be found at [13].

Take a look at the following subsection 3.2.4 to Figure out what pre-processors are attached to the model.

We have a valid post-processor that can be attached to the model which is the Threshold Processor. However, specifically for my work, It was not critical to attach it to the model, as the ThresholdProcessor is used to set thresholds during model evaluation.

3.2.4 Processing in the Use-Case

It is worth mentioning that the initial dataset referred to in paragraph 3.2.3 is not the dataset used by our model in its training or prediction processes, but a pre-processing step is needed to modify the features of our model in an appropriate manner and to obtain more insightful results from our model.

Processing in the use-case is predominantly focused on processors already implemented in our ICDATA framework, the full description of the various processors in the ICDATA framework is already stated in the 3.1.3 subsection. Apart from those processors, however, we also have some processors that are only specific to our use-case and specifically implemented to deal with its features.

Use-Case Specific Processors

Figure 3.14 addresses the pre-processors that have been specifically added in order to conform with the features of the use-case, but they are still typically focused on the ICDATA pre-processors [9].

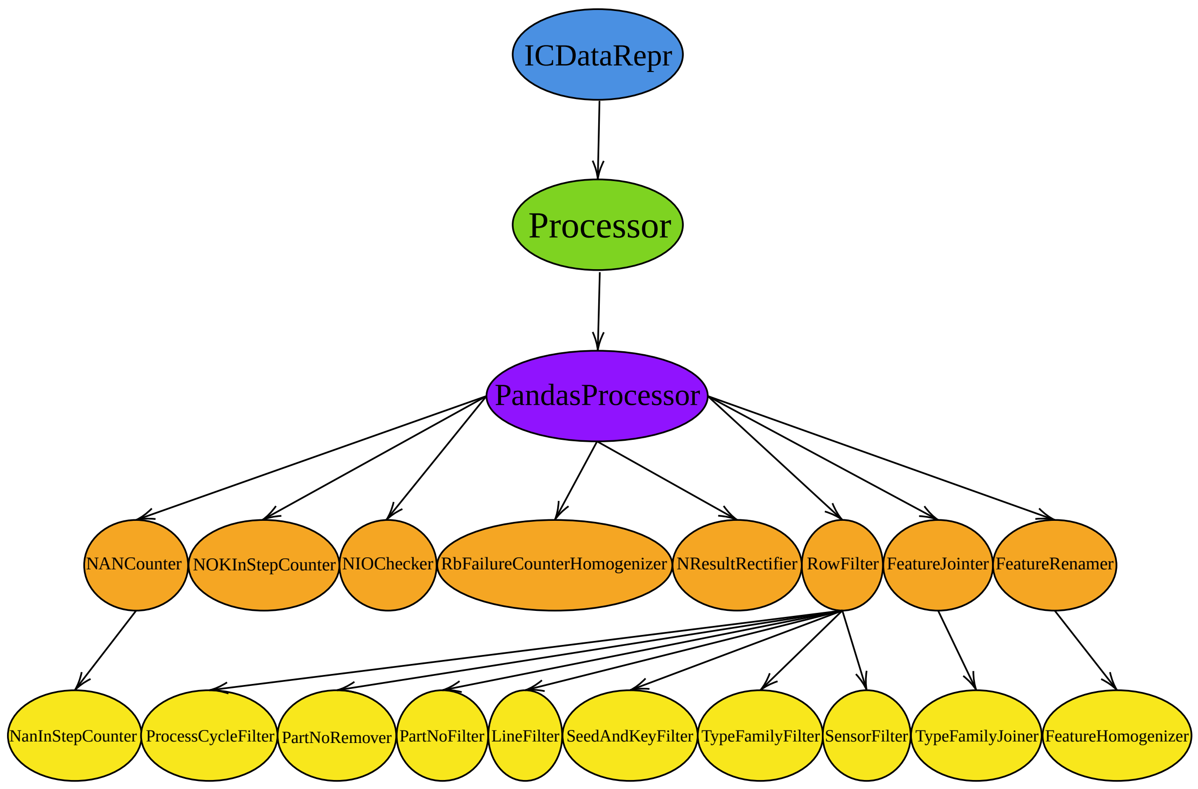

Figure 3.14: Use-Case-specific processors

- NANCounter: Generate a new feature that represents the number of null entries inside the specified features set.
- PartNoRemover: A processor that removes the part number from the feature name.
- PartNoFilter: A processor that filters the parts with the part number in their feature name.
- LineFilter: A processor that filters the parts according to the production line on which they were checked.
- FeatureHomogenizer: Processor that removes the name of the testing cell from the feature name as attached by the Machine Learning interface.

Processors From ICDATA

Apart from the specific pre-processors built for the use-case, we, in addition, utilize some of the ICDATA pre-processors and post-processors that are already shown in the Figure 3.4.

Some of the ICDATA processors [2] required for our use-case are:

- HyperParameters.

- ThresholdProcessor.
- FeatureSelector.
- OneHotEncoder.
- PolynomialFeatures.
- QuantileMapper.
- NanFilter.

Note that the only processor that could be used as a postprocessor in our use-case is the ThresoldProcessor.

3.2.5 How the Use-Case Is Used?

In the usual way, we deal with the use-case the same as the flow that I described in the full example section 3.1.6, although nothing can be mentioned except that the dataset and the model are referred to differently from the typical case scenario.

The code snippets shown here are based on data from [9].

Importing the Dataset:

In the use-case, we use the internally defined function **get_dataset** to import the dataset flawlessly from the specified HDF file.

```
from testing_cells.models.data import get_dataset
data = get_dataset('./dataset_file.hdf')
```

Calling the model:

After calling our dataset, we need to assemble our model. To achieve that, we typically use an internally defined use-case function called **get_model**, which returns a ready-to-use version of the model. It returns the model with the full set of hyperparameters, preprocessor, and postprocessors attached to it.

```
from testing_cells.models.recheck_prediction import get_model
model = get_model(
typetree_file='./typetree.hdf',
quantiles_file='./quantiles_file.pkl')
```

4 State of the art

4.1 LIME

4.1.1 Theoretical and Mathmatical Explanation

4.1.2 Working Principle

4.2 SHAP

4.2.1 Theoretical and Mathmatical Explanation

4.2.2 Working Principle

4.3 Microsoft Interpret

4.3.1 Theoretical and Mathmatical Explanation

4.3.2 Working Principle

5 Experiments and Results

5.1 LIME Results

5.1.1 Performance

5.1.2 Advantages

5.1.3 Findings

5.1.4 Disadvantages

5.2 SHAP Results

5.2.1 Performance

5.2.2 Advantages

5.2.3 Findings

5.2.4 Disadvantages

5.3 Microsoft Interpret Results

5.3.1 Performance

5.3.2 Advantages

5.3.3 Findings

5.3.4 Disadvantages

5.4 Comparison

6 Conclusion and Outlook

6.1 Conclusion

6.2 Outlook and Future Scope

Bibliography

[1] G. Van Rossum and F. L. Drake, “Python 3 Reference Manual”. CreateSpace, 2009 (cit. on p. 5).

[2] *ICDATA*. Developers, “ICDATA, a Python framework for reproducable and large scale machine learning experiments”, version 1.22.1, Robert Bosch GmbH, 2020 (cit. on pp. 5–13, 24, 25, 32).

[3] T. E. Oliphant, “A guide to NumPy”. Trelgol Publishing USA, 2006, vol. 1 (cit. on p. 6).

[4] B. Fortner, “HDF: The hierarchical data format”, *Dr Dobb’s J Software Tools Prof Program*, vol. 23 (5), p. 42, 1998 (cit. on p. 7).

[5] W. McKinney, “Data structures for statistical computing in python”, in *Proceedings of the 9th Python in Science Conference*, Austin, TX, vol. 445, 2010, pp. 51–56 (cit. on p. 7).

[6] M. T. Ribeiro, S. Singh, and C. Guestrin, “” Why should I trust you?” Explaining the predictions of any classifier”, in *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*, 2016, pp. 1135–1144 (cit. on pp. 10, 13, 14).

[7] S. M. Lundberg and S.-I. Lee, “A unified approach to interpreting model predictions”, in *Advances in neural information processing systems*, 2017, pp. 4765–4774 (cit. on pp. 13, 14).

[8] T. Chen and C. Guestrin, “Xgboost: A scalable tree boosting system”, in *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, 2016, pp. 785–794 (cit. on p. 23).

[9] T. Windisch and B. Geisselmann, “TestingCells, a package with models for Re-Check prediction of testing cells”, version 0.8.4, Robert Bosch GmbH, 2020 (cit. on pp. 23, 29–31, 33).

[10] J. H. Friedman, “Greedy function approximation: a gradient boosting machine”, *Annals of statistics*, JSTOR, pp. 1189–1232, 2001 (cit. on p. 23).

[11] R. Bosch, “Electronic stability program (ESP®)”, [Online]. Available: `https://www.bosch-mobility-solutions.com/en/products-and-services/passenger-cars-and-light-commercial-vehicles/driving-safety-systems/electronic-stability-program/esp-generation-9/`, Accessed: 2020-06-23 (cit. on pp. 27, 28).

[12] R. Bosch, “Electronic stability program (ESP®)”, [Online]. Available: `https://www.bosch-mobility-solutions.com/en/products-and-services/passenger-cars-and-light-commercial-vehicles/driving-safety-systems/electronic-stability-program/`, Accessed: 2020-06-23 (cit. on pp. 27, 28).

[13] *XGBoost*. Developers, “XGBoost Parameters”, [Online]. Available: `https://xgboost.readthedocs.io/en/latest/parameter.html`, Accessed: 2020-06-23 (cit. on p. 31).

List of Figures

List of Tables