Reusing XAI Techniques for Personalized AI Systems

Universidad Complutense de Madrid

Facultad de Informática

Trabajo de Fin de Grado del Grado en Ingeniería Informática

Curso 2021/22

Reutilización de Técnicas de XAI para Sistemas de IA Personalizados

Autor:
Jesús Miguel Darias Goitia

Directores:
Belén Díaz Agudo, Juan Antonio Recio García

Madrid, May 30, 2022

Abstract

Interpretability has become a crucial aspect in the field of Artificial Intelligence. Although there are many Explainable AI (XAI) techniques that provide explanations and improve the interpretability of AI systems, there are certain limitations. One of the most important limitations is that it is necessary to have a deep understanding of XAI to determine which techniques are better to explain a specific AI model to a specific user.

In this project, I reviewed some of the most well-known XAI libraries and methods. Then, I used these tools to build an API that unifies model-agnostic methods used to explain different types of models. Using the API as a basis, I built a case-based reasoning system that aims to recommend the best explanation techniques to users based on the feedback of previous users with similar background knowledge and AI models. The CBR system case base was populated by using real-user feedback on explanations for a set of use cases. Finally, I evaluate the case base and the performance of the system.

Keywords

XAI, Artificial Intelligence, Interpretability, Explainability, CBR, Case-based Reasoning, Machine Learning

Resumen

La interpretabilidad se ha vuelto un aspecto crucial en el campo de la Inteligencia Artificial. Aunque existen muchas técnicas de Inteligencia Artificial Explicable (XAI) que proporcionan explicaciones y mejoran la interpretabilidad de sistemas de IA, existen ciertas limitaciones. Una las limitaciones más importantes es la necesidad de tener un conocimiento profundo en XAI para determinar cuales son los mejores métodos para explicar un modelo de IA específico a un usuario específico.

En este trabajo hago una revisión de algunas de las librerías y métodos de XAI más conocidos. Luego, utilizando estas herramientas, desarrollo una API que unifica métodos agnósticos para explicar diferentes tipos de modelos. Utilizando dicha API como base, construyo un sistema de razonamiento basado en casos que busca recomendar las mejores técnicas de explicación a los usuarios, basándose en la evaluación previa de otros usuarios con conocimientos y modelos de IA similares. La base de casos del sistema CBR se generó utilizando la evaluación de usuarios reales acerca de explicaciones generadas sobre un conjunto de casos de uso. Finalmente, evalúo la base de casos y el rendimiento del sistema.

Palabras clave

XAI, Inteligencia Artificial, Interpretabilidad, Explicabilidad, CBR, Razonamiento Basado en Casos, Aprendizaje Automático

Contents

Chapter 1

Introduction

In recent years, the Artificial Intelligence and Machine Learning fields have been steadily growing. Particularly, the use of machine-learning predictive models that aim to provide insight into numerous fields has become more and more popular. As AI models gain more importance for human society and become ingrained in our day-to-day life, it is fundamental for people to trust them. In the case of machine-learning engineers, they must be able to understand the behavior of their models so they can improve their performance. They also must be able to identify any possible bias to guarantee the impartiality of the predictions. On the other hand, certain regulations, such as the General Data Protection Regulation (GDPR) in Europe [1], imply that users have the right to obtain an explanation of the decision reached. Furthermore, end-users should be confident in the results provided by the model. Naturally, the need for interpretability arises.

Miller (2017)[2] defines interpretability as the degree to which a human can understand the cause of a decision. The Explainable AI field, also known as **XAI**, refers to methods and techniques that are used to make AI models have higher interpretability. Models with high interpretability generate predictions whose origins are easier to understand, whereas understanding why certain predictions were made by lower-interpretability models is not a trivial task. An example of this is a linear regression model and a neural network. The linear regression model is highly interpretable, as every prediction value can be directly mapped to the defined linear function. On the other hand, the prediction of a neural network relies on the outcomes of its neuron layers, where each neuron is a single linear function by itself. If there is a large number of neurons and layers, it becomes practically impossible to map the final prediction to the combination of all their outcomes. Thus, linear models are naturally interpretable, as opposed to neural networks.

Interpretable models are often referred to as **white-box** or **glass-box** models because their functioning is transparent to the observer. Some examples of white-box models are linear and logistic regression, as well as decision trees. Less interpretable models are referred to as **black-box** models since their complexity obfuscates their inner-working, having the input data and output prediction as the only available information. Since white-box models are already interpretable due to their intrinsic simplicity, most of the existing explanation techniques are oriented to black-box models, in an attempt to make their functioning more transparent to both developers and users.

XAI techniques are classified according to different criteria. For example, the methods

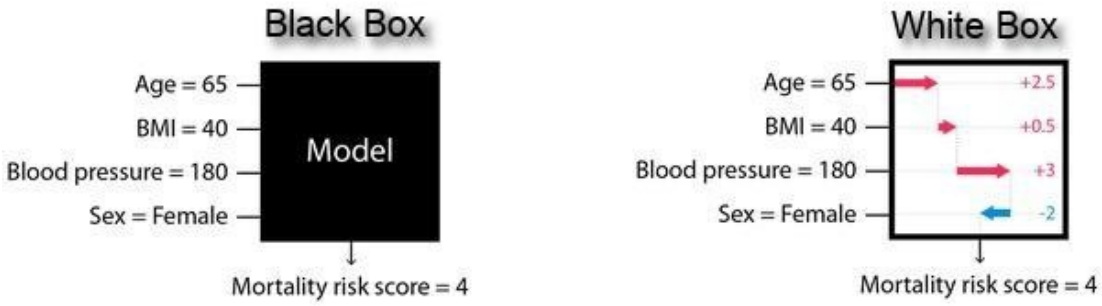


Figure 1.1: Example diagram of a black-box and a white-box model. The way a black-box works internally is unknown, while a white-box allows to map the prediction value to specific features.

that are applied after the model has been trained are referred to as **post-hoc**. For fully black-box models, post-hoc methods are generally used since no information can be derived from the model if it has not been trained. Interpretation techniques can also be classified as **model-specific** or **model-agnostic**. Model-specific methods can only be used to explain specific types of models. This implies knowing certain aspects of the models, such as their architecture. On the other hand, model-agnostic methods can be used to explain any type of model. They usually work by analyzing feature input and output pairs. By definition, these methods cannot have access to model internals such as weights or structural information [3]. For that reason, they are particularly useful for black-box-model interpretation. It is also possible to classify the interpretation technique depending on its scope: **global** or **local**. Global methods aim to explain the general behavior of the whole model, while local methods focus on explaining the result of an individual prediction.

In the last decade, many state-of-the-art XAI tools have been developed due to the growing popularity of machine learning. However, certain issues have also come to light. In particular, it is difficult to determine which methods are more effective to explain a specific model. Although model-agnostic methods can be applied to any kind of machine-learning model, their performance varies from case to case. The task of choosing the right methods becomes even more complicated when the background of the user receiving the explanation is considered.

One of the main purposes of this project is to provide the explanation techniques that are more useful to the users, based on the experiences of other users with similar models and background characteristics. The focus will be on using post-hoc, model-agnostic explanation methods. All of the code related to the project, as well as a set of use cases, is available at a GitHub repository[1]. This project has been developed within the context of *iSee: Intelligent Sharing of Explanation Experience by Users for Users* [4]. The iSee project aims to address the science and technology for capturing, sharing, and re-using explanation strategies based on similar user experiences while following the best XAI practices.

[1] https://github.com/jesusdarias/Reusing-XAI-Techniques-for-Personalized-AI-Systems

1.1 Objectives and Work Plan

In this section, the main objectives of the project are defined. The work plan for each objective has been outlined into specific tasks to be completed throughout the development cycle.

1. **Review the state-of-the-art XAI methods and libraries.**
   - Learn about the most popular model-agnostic XAI methods and libraries.
   - Define an evaluation criteria for the libraries.
   - Test the methods using different machine-learning use cases.
   - Identify advantages and limitations of the tested tools.
2. **Develop a unified API** that implements the reviewed XAI tools.
   - Design and develop the structure of the API by characterizing the requirements and parameters of each method.
   - Implement the methods from the libraries that will be used for the API.
   - Ease the use of the API to other users or developers by documenting the methods and dockerizing the implementation.
3. **Develop a case-based reasoning system** based on the previously built API that recommends the most suitable explanation methods based on previous user experience, according to the characteristics of the model and user background.
   - Define the structure of the cases.
   - Implement additional functionalities to the API for the retrieval process of similar cases, as well as retaining new cases.
   - Populate the case base by gathering user feedback on explanation methods applied to a series of use cases in different domains.
4. **Evaluate the case base and test the performance of the CBR system.**
   - Discuss the statistical description of the case base, trying to identify possible patterns.
   - Validate the CBR system and measure its performance.

Chapter 2

XAI Techniques and Libraries

In this section, I reviewed the most used XAI model agnostic techniques and libraries available. To illustrate the explanations that these tools provide, I applied them to two different use cases. The first use case is a random forest model used to predict cervical cancer based on tabular data, with attributes such as age, number of pregnancies, years that the individual has been smoking, etc. The second use case refers to a neural network that classifies color images of animals. Both of these use cases are explained in greater detail in Chapter 4. Throughout the review, I introduced figures that depict the explanation format provided by the methods. Also, as a side note to avoid any confusion, I used the terms explanation technique, explanation method, or simply "explainer" interchangeably.

2.1 Explainers

Local Interpretable Model-Agnostic Explanations (LIME): [5]

LIME is one of the most popular local explanation methods. It attempts to understand the model by perturbing the input of data samples and understanding how the predictions change. The intuition to local interpretability is to determine which feature changes will have the most impact on the prediction. According to its authors, the algorithm fulfils the desirable aspects of a model-agnostic explanation system regarding flexibility. LIME can work with any machine-learning model and is not limited to a particular form of explanation and representation. The implementation of this method offers different modules oriented to the data type of the model to be explained, supporting tabular data, color images, and text.

An essential requirement for LIME is to work with an interpretable representation of the input, like images or bags of words, that is understandable to humans. Although LIME is general and flexible, there are some scenarios where it is not possible to know the scope of instances that are also similarly affected by the same features. For that reason, there are other similar approaches like Anchors [6] that provide values to estimate this scope.

Anchors: [6]

Also known as scoped rules, Anchors attempt to explain individual predictions of a black-box model by finding a decision rule which allows the perturbation of other

Figure 2.1: Explanation using *LIME* for tabular data applied to the cancer classification model. The explanation contains the prediction made by the model for the provided individual, the weight of each attribute to the prediction value in a tree-like plot, and a table showing the values of each of the attributes of the individual appearing in the plot.

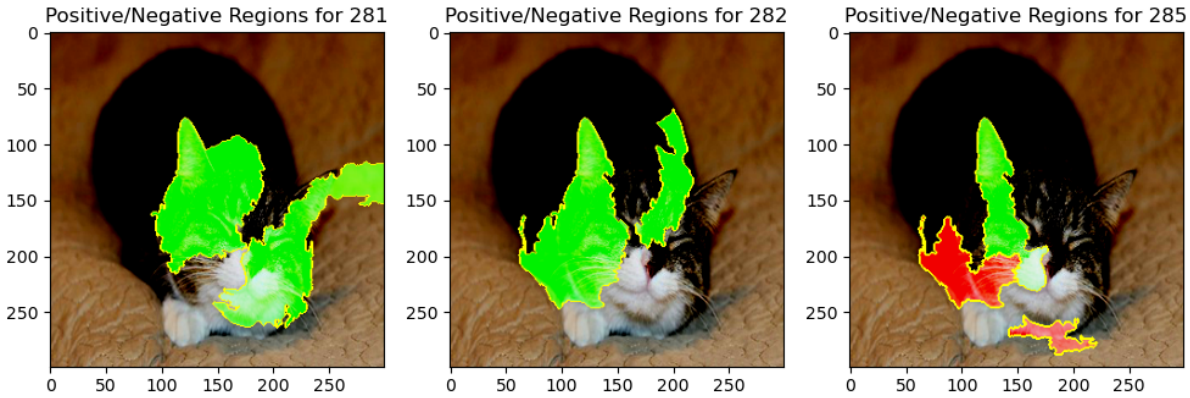


Figure 2.2: Explanation using *LIME* for images applied to the animal classification model. The group of pixels that contribute to the prediction of the image class is displayed in green, and the pixels that contribute negatively are highlighted in red. The numbers 281, 282, and 285 represent the classes "Tabby", "Tiger Cat", and "Egyptian Cat", respectively.

feature values without affecting the actual prediction. This algorithm was developed by the same researchers that proposed LIME. Similar to LIME, the approach taken by this algorithm is based on a perturbation strategy to generate local explanations, but instead of representing them through an approximated model, these explanations are expressed as IF-THEN rules. This makes anchors very easy to interpret.

Since the perturbations are produced and evaluated for every specific instance that is being explained, the internal structure of the model is never referenced; thus, this algorithm is completely model-agnostic. Furthermore, the rules generated are reusable since through the coverage measure, they state to what other types of instances such rules apply in the perturbation space. However, one of the main drawbacks is that the generated boolean rules for tabular data models can be very complicated. This tends to happen when the target classes are imbalanced and there is a considerable number of attributes. As a result, the proposed anchor will have

very low coverage and consist of many boolean expressions. On a positive note, Anchors provide flexibility as they can be applied to models with tabular data, images, and text.


```
"anchor": "Smokes..years. > 0.00 AND Hormonal.Contraceptives..years. > 2.00
    AND 15.00 < First.sexual.intercourse <= 17.00 AND STDs..number. <= 0.00
    AND STDs..Number.of.diagnosis <= 0.00 AND Age > 31.00 AND Num.of.
    pregnancies > 2.00 AND Number.of.sexual.partners > 2.00",
"precision": 0.152,
"coverage": 0.002
```


Figure 2.3: Anchor example using the cancer prediction model. This anchor was generated from an individual that labeled as positive for cancer. Because of the class imbalance, the rule is complicated, and the precision and coverage values are very low. This makes the generated anchor less reliable and difficult to understand.

Shapley Additive Explanations (SHAP): [7]

SHAP is an explainer method based on Shapley values and local surrogates (LIME). Shapley values, a method from coalitional game theory, aims to fairly distribute the total value among the different features of an instance, working at a local level. The value that is distributed is the outcome given by the model, which in the case of a classification problem is the predicted probability. SHAP developers take this concept one step further by representing Shapley values as an additive linear model, bringing in aspects from LIME.

There are two different approaches: KernelSHAP and TreeSHAP. The first one refers to a kernel-based estimation for Shapley values calculation. However, computing Shapley values is computationally expensive, so the researchers developed TreeSHAP, which is a faster alternative to KernelSHAP but only applicable to tree-based models such as random forests. Nevertheless, the main advantage of the KernelSHAP explainer is that it is completely model-agnostic. SHAP can also provide a global explanation of the model by calculating all the Shapley values for each instance but depending on the approach and the given data, this calculation could be too slow.

Figure 2.4: Explanation using *Kernel SHAP* for an individual of the cancer prediction use case. The explanation is displayed as a force plot that confronts the features that positively contribute to the positive-for-cancer class (left in red) and those that contribute negatively to it (right in blue). It also displays the predicted probability and the base value for reference.

Partial Dependence Plots (PDPs): [8]

Partial Dependence Plots show the marginal effect one or two features have on the predicted outcome of a model. They explain the relationship between a feature and the target, which could be linear or more complex. One of the main advantages of these plots is their interpretability. However, this function represents how much such a feature influences the outcome only when there is no correlation with other features. This is one of the main downsides of PDPs since they may lead to erroneous interpretations. The plots should show the density of instances along the feature axis, as areas of the graph with lower density are not as reliable as other areas where the instances concentrate. While this is a simple and effective explanation method, it is highly limited by its applicability scope since it can not be used when correlation exists among features.

Accumulated Local Effects (ALE): [9]

ALE plots have the same intrinsic purpose as partial dependence plots: to describe how one or two specific features affect the prediction of the model on average. Nevertheless, the difference is that ALE plots are unbiased, meaning that they are still reliable when features are correlated. They are also faster to compute since they scale linearly.

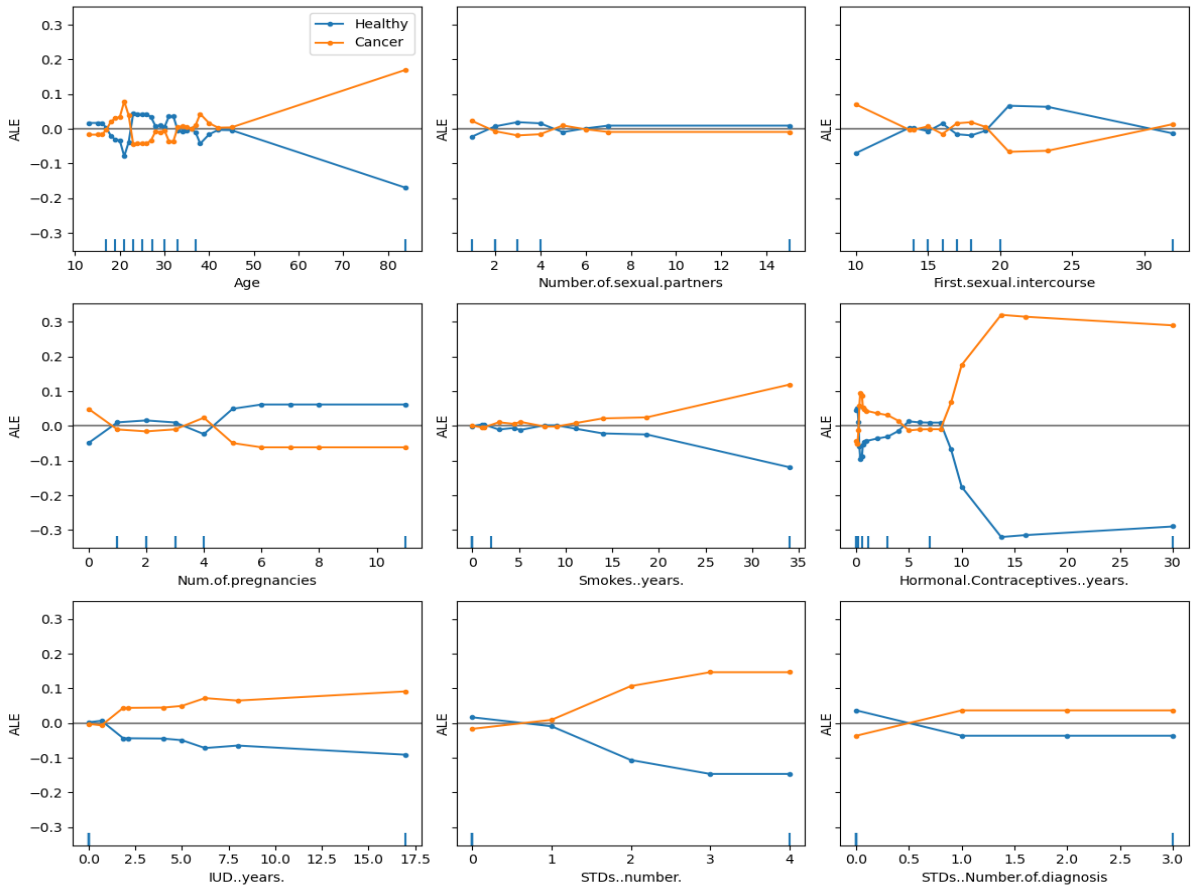


Figure 2.5: *ALE* plots of the attributes from the cancer classification model. The y-axis represents the global feature effect on the prediction probability for each feature. In this example, the probability of the cancer-positive class tends to increase as the age of individuals increase.

One of the key characteristics of ALE is that the calculation of the average effect is separated through intervals of that feature. The mathematical background and implementation of ALE plots are far more complicated than partial dependence, but the shortcomings faced by PD Plots are covered by using ALE plots when working with correlated features. For this reason, I decided to use ALE over PDPs for the implementation of the API described in chapter 3.

Counterfactual explanations: [10]

Counterfactuals are one of the best local methods when the target audience of the explanations is the customers or end-users that need to have a better understanding of a specific prediction to know how to alter the input to obtain a different result. The idea behind it is to provide a new instance that is as similar as possible to the original instance, but whose prediction is different. There are several aspects to be considered when choosing counterfactuals as the explanation method. First, counterfactual instances should vary as few features as possible. Second, a counterfactual should have feature values that are likely to be present in a real instance. Otherwise, the counterfactual should be discarded. Lastly, whenever possible, multiple counterfactuals that differ from each other should be provided.

Counterfactual explanations offer many advantages: they are easy to implement and do not require access to the training data. Moreover, their main advantage is that the explanation given is clear and easy to understand, even for people with little to no background in ML or statistics. However, one of the challenges of using this method is ensuring that the features of the counterfactual generated do not take unrealistic values that are not representative of real data. For example, referring to the cancer use case, a counterfactual could assign a value to the attribute "Years of Smoking" that is higher than the actual age of the person. Unfortunately, depending on the specific nature of the problem, this can be a difficult challenge to overcome.

Instance	5	4	37	3
Counterfactual	Sexual partners	Pregnancies	Smokes (years)	Contraceptives (years)
1	-	-	5.3	0.7
2	1	-	8.6	-
3	-	1	24.7	-

Table 2.1: Counterfactuals examples generated for an individual of the cancer classification use case that was labeled as positive for cancer. The attributes that did not change from the original instance are represented by a hyphen to increase the clarity of the explanation. All of the proposed counterfactuals involve the decrease of the years of smoking to obtain a classification of "Healthy".

Contrastive Explanation Method (CEM): [11]

CEM provides local explanations for black-box classification models in terms of Pertinent Positives (PP) and Pertinent Negatives (PN). Pertinent positives represent the features that should be minimally and sufficiently present to predict the same class as the original instance. On the other hand, Pertinent Negatives represent the features that should be minimally and sufficiently absent from the original instances to maintain the predicted class. In a certain way, Pertinent Positives can

be compared to anchors, while Pertinent Negatives are similar to counterfactuals. According to the authors, the explanations provided by CEM are clear and intuitive since they state that the instance is classified in the predicted class because some specific features are present and because some specific features are absent.

Feature Importance: [3]

Feature Importance is a global explanation method that measures the increase in the model error when random values are assigned to a specific feature. If the error of the model increases significantly when all of the values of a specific feature, that feature is considered to be important. This is a straightforward technique that can be applied to different types of black-box models. However, it is important to keep in mind that the importance of the features may split between them when they are correlated.

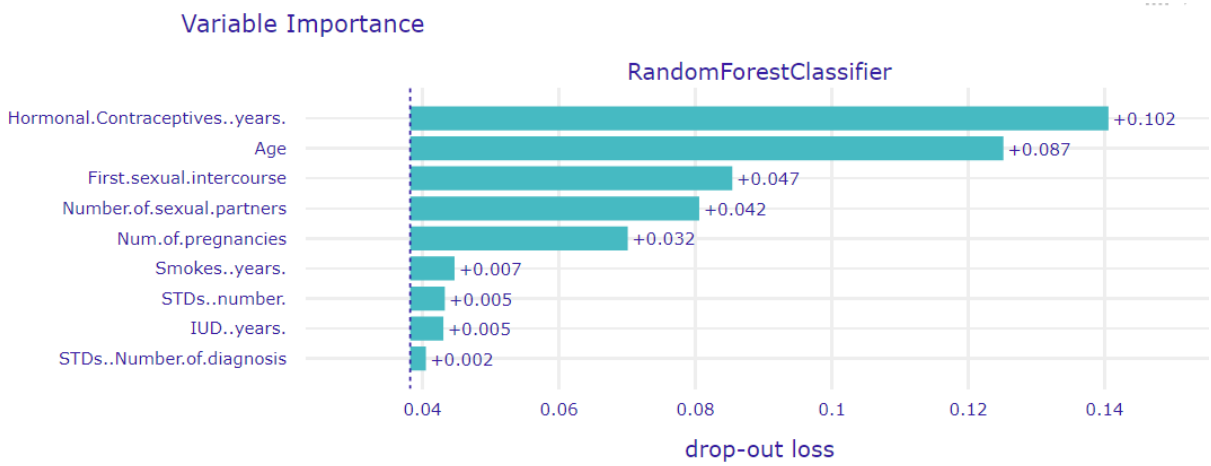


Figure 2.6: Feature Importance of the features from the cancer classification use case. The explanations was generated using Dalex [12]. From the explanation, we can infer that the most important features are the years of using contraceptives and the age of the individual.

2.2 XAI Libraries Review

The review and analysis of the XAI libraries are based on my work in *A Systematic Review on Model-agnostic XAI Libraries.* [13] presented at the 2020 XCBR Workshop.

2.2.1 Review Criteria

To evaluate the XAI libraries, I defined a series of attributes that allows the comparison of the characteristics of each library. The following aspects constitute the review criteria that was considered when evaluating the libraries:

- **Documentation and usability:** Is the documentation thorough and accurate? The documentation should also be complemented with examples on how to use the methods to ease the use of the library.
- **Performance metrics:** Refers to the availability of metrics such as accuracy, recall, ROC/AUC values, mean squared error, etc. These metrics allow users to evaluate

the performance of a model.

- **Available explainers:** mostly the ones described in the previous section.
- **Analysis and description capabilities:** refers to the availability of tools that allow a better interpretation of data itself such as marginal and scatter plots, data imbalances, etc.
- **Interactivity:** meaning the user is able to dive deeper into the explanation that is outputted by looking into certain features or other aspects more thoroughly.
- **Personalization:** Refers to the capability of providing different explanations according to the user profile and model characteristics.
- **Dependencies:** Development language/environment and requirements (if any). Use of other methods from libraries such as *TensorFlow* [14], *Scikit-learn* [15], *PyTorch* [16], and others. I also took into account the use of wrapper classes to implement some of the individual explanation methods previously described.

2.2.2 InterpretML

InterpretML [17] is one of the most popular XAI libraries. It offers both local and global model-agnostic explanations for black-box models oriented to tabular data. It includes methods such as *LIME*, *SHAP* and Partial Dependence plots. One of the main advantages of this library is that it implements a dashboard that allows users to interact with the explanations provided by looking at specific features or instances, thus allowing them to have a better understanding of the explanation.

Nevertheless, Interpret presents two important limitations. First, it does not cover a significant number of explanation methods to be considered a multi-purpose tool. And second, the original implementations of both LIME and SHAP are limited by the wrapper classes to only regression and binary classification models. The general aspects of this library according to the review criteria are comprised in Table 2.2.

Documentation and usability	The documentation is well-structured and explanatory. Usage examples are provided in a simple fashion so the user is able to begin using the library right away. This library is very intuitive and learning how to use it should not cause any issues for less-experienced developers.
Metrics	ROC/AUC values.
Explainers	LIME, SHAP, and PDPs.
Analysis	Yes. It provides marginal plots and class histograms.
Interactivity	It has a dashboard feature that allows the end-user to further inquire into different features and compare different explanations of the same instance.
Personalization	Not available
Dependencies	Python 3.6+. For the LIME and SHAP explainers, wrapper classes are used based on the original implementation.

Table 2.2: Review of InterpretML.

2.2.3 Alibi

Alibi [18] provides local and global explanation methods for classification and regression problems for black-box models. One of the main features of ALIBI is that some of its methods can be applied to tabular, image, or text data. Out of the reviewed libraries, is the one with the most explanation methods. One of the strengths of this library is that some explainers are optimized for Tensorflow models, such as CEM and counterfactuals, but any other model can be provided. The provided explainers are highly configurable which is good for experienced developers, but their functioning can be difficult to grasp for non-experienced users. Another relevant aspect is that the explanation is often given in a format difficult to comprehend, so it may be necessary to manually transform it to a more interpretable format. The review of Alibi is presented in Table 2.3.

Documentation and usability	The documentation is very extensive and educational. Not only does it explain how to use the methods, but gives a mathematical background for each explainer. However, the examples provided for some explainers only cover the explanation of models with a Tensorflow backend, which may cause difficulties to users who are not experienced in this environment.
Metrics	Linearity measure and trust scores.
Explainers	ALE, Anchors, SHAP, Counterfactuals, and CEM.
Analysis	Not available.
Interactivity	This library is not interactive. The process is finished once the explanation is outputted. In fact, most explanations are given in a low-level fashion as raw data that the user may need to convert to a more interpretable format.
Personalization	Not available.
Dependencies	Python 3.6+. This library is heavily based on tensorflow. For the SHAP explainer, it uses the original implementation of the author [7].

Table 2.3: Review of ALIBI.

2.2.4 Aix360

Aix360 [19] is a multi-purpose library that provides some of the most up-to-date explainers available. Besides implementing the widely accepted LIME and SHAP methods, algorithms like Protodash [20] and CEM with Monotonic Attribute Functions [21] show some of the latest local explainers available. Aix360 also offers global explainers such as Generalized Linear Rule Models and model performance metrics. Similar to the rest of the reviewed libraries, Aix360 does not cover the personalization aspect of the explanations. Despite this, the authors acknowledged that some explanations are more suitable than others depending on different factors, such as the target to be explained and the profile of the end-user [22]. Table 2.4 contains the review of this library.

Documentation and usability	The documentation is clear and extensive. It provides many usage examples with different data sets that make the library easy to use. The Aix360 website offers interactive tutorials as complementary guidance for its use.
Metrics	Faithfulness and monotonicity. Faithfulness refers to the correlation between the feature importance assigned by the interpretability algorithm and the effect of features on model accuracy. On the other hand, monotonicity tests whether model accuracy increases as features are added in order of their importance.
Explainers	LIME, SHAP, CEM, Protodash, GLRM, and Profweight
Analysis	Yes. Particularly, the Protodash algorithm is able to find prototypes that help summarize the data set.
Interactivity	This library does not offer interactivity features. Explanations are displayed to users in graphic format or plain data, and the is no further interaction between the user and the program.
Personalization	Not available. However, the importance of personalization of the explanations is referenced in the official website throughout the interactive demo [23]. It outlines that different users look for different kinds of explanations.
Dependencies	Python 3.6+. The implementation of the original authors is used for the LIME and SHAP explainers. [5][7]

Table 2.4: Review of Aix360.

2.2.5 Dalex

Dalex [12] is a multipurpose library that focuses on model-agnostic explanations for black-box models. The core methodology behind it is to create a wrapper object for the given model that can later be explained through a variety of local and global explainers. This library implements well-known explainers such as LIME, SHAP, and ALE, and also allows measuring the fairness of the model.

It provides plenty of different performance metrics according to the given model. Dalex is complemented by the Dalex *Arena* visual dashboard, which allows exploring different explanations interactively. Figure 2.7 shows an screenshot of Arena. One of the setbacks of this library is that it limits the scope of LIME and SHAP to binary classification problems, similar to InterpretML. Dalex is reviewed according to the criteria in Table 2.5.

Figure 2.7: Screenshot of the Arena dashboard. [18]

Documentation and usability	The documentation is good and plenty of examples are provided. Other complementary resources such as tutorials are provided as well. However, it may be hard to find the exact usage illustration for a specific explainer in a notebook since they are organized by data sets.
Metrics	There are many different metrics provided depending on the nature of the problem. For classification, F1 score, accuracy, recall, precision, specificity, and ROC/AUC are provided. For regression problems there is mean squared error, R squared, and median absolute deviation.
Explainers	LIME, SHAP, PDPs, ALE, and Feature Importance
Analysis	Not available.
Interactivity	The Dalex Arena allows the user to easily compare different explanations for the same problem and even for different models.
Personalization	Not available.
Dependencies	Python 3.6+. For the LIME and SHAP explainers, wrapper classes are used based on the original implementations [5, 7].

Table 2.5: Review of Dalex.

2.2.6 DiCE

DiCE [24], standing for Diverse Counterfactual Explanations, is a library that focuses on counterfactual generation for tabular data models. Three different approaches can be taken when using dice to find counterfactuals: using random sampling, k-d trees, or genetic algorithms. Its simplicity makes DiCE a great tool when only counterfactuals are needed as an explanation method. The counterfactuals can be generated using the training data to guide the search using the public data interface or by providing instead the description of the data using the private data interface. Table 2.6 contains the characteristics of DiCE according to the review criteria.

However, although the developers claim DiCE to be a model-agnostic tool, this is not completely accurate. In particular, counterfactual generation without the training data is currently available for *TensorFlow* models only. The same applies to the genetic algorithms approach to generating counterfactuals. Nevertheless, it is expected future development of this library to accept other model backends in the future.

Documentation and usability	The documentation is straightforward and provides various examples. It is very simple as this library uniquely relies on counterfactual generation.
Metrics	Not available.
Explainers	Counterfactuals
Analysis	Not available.
Interactivity	This library does not provide interactivity, but since the explanations are counterfactuasl, the data is presented in an easy-to-interpret format.
Personalization	Not available.
Dependencies	Python 3+. It does not use other external XAI libraries. However, depending on the generation method, it uses *TensorFlow* and *Pytorch*.

Table 2.6: Review of DiCE.

Chapter 3

Explainers API

In this chapter, I described the development of an API oriented to the explanation of machine-learning models through the use of model-agnostic explanation methods. The API was built using Python [25], as most of the available XAI tools are built on this programming language. For the development of the API, I followed the representational state transfer (REST) architectural style. Web APIs that follow REST conventions are based on HTTP methods to access different resources by URL-encoding the required parameters. One of the advantages of REST API is that they are stateless, meaning that no session state from previous interactions is stored.

The external libraries and applications that I used during the development and testing of the API are described in Section 3.1. The structure of the explanation methods and their description is covered in Section 3.2.

3.1 External Tools and Applications

For the development of the API, I chose to use *Flask* [26]. Flask is a micro web framework written in Python. Microframeworks facilitate the reception and routing of HTTP requests, as well as the return of the related response. Microframeworks are often used when building APIs for other services or applications. One of the strong points of Flask is its simplicity; it is possible to develop simple applications very easily without other additional tools or libraries. Particularly, I used Flask-RESTful, an extension of Flask that implements support for quickly building REST APIs.

To test the API methods, I used *Postman* [27]. Postman is an API platform that covers all of the steps in the API lifecycle. It has many different applications, but I mainly focused on testing through HTTP requests to the running Flask server. One of the best features of Postman is its user interface. It is way simpler to build the HTTP requests through the interface provided by Postman than through other command-line tools I tested like Curl [28]. Postman is also a great tool to manage different collections of requests.

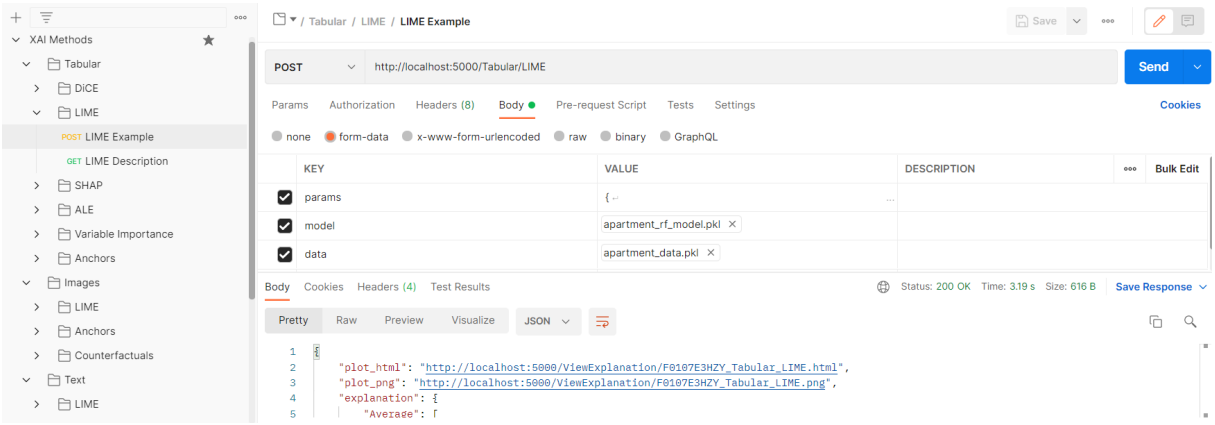


Figure 3.1: *Postman*'s user interface. The directory tree at the left shows the folders with the requests for the different explanation methods described in Section 3.2.

Once the API was tested, I created a Dockerfile for the installation of the dependencies and launching of the Flask server. *Docker* [29] is an open platform for developing, shipping, and running applications. The main advantage of using Docker is that it makes it possible to separate the applications from the infrastructure. This is achieved by running applications in an isolated environment known as a container. Containers are self-sufficient as they include all of the necessary dependencies needed to run the applications. Using containers eases the task of delivering software since the developer does not need to worry about the installation of additional software in the end-users system. The Dockerfile for the server is available at the GitHub repository associated with this project.

3.2 Explainers Catalogue

Following the REST schema, I implemented different explanation methods as uniquely identifiable resources of the API. Each of the resources contain function definitions for two HTTP requests types: *POST* and *GET*. The POST method is where an actual explanation is generated according to the parameters that are passed in the request. On the other hand, the GET method serves as the documentation for each one of the resources. It does not receive any parameters and returns a JSON object with the information about the specified resource. Although the different API resources may require different parameters for POST requests, there are 3 parameters that are necessary in most cases: *model*, *data*, and *params*.

The *model* parameter is a file representative of the machine-learning model that is to be explained. The supported files vary from explainer to explainer but the generally accepted file extensions are *.h5* for TensorFlow models, *.pt* for Pytorch models, and *.pkl* for Scikit-learn and other types of model. If the model was not built using any of the previous libraries, it must have a *predict* function signature, which is the prediction function of the model that will be used to generate the explanations. The .pkl files must be generated using the Joblib [30] library for the serializing of Python objects.

The *data* parameter is a .pkl file containing the training data used to train the model in a data frame format. The training data is required in almost all of the tabular data explainers. The target column must be the last column of the data frame passed. The data parameter is not necessary for the implemented image and text explainers.

The *params* parameter is a meta-parameter. It contains a JSON Object with the configuration parameters that are passed to the actual functions related to the XAI method being used. Many of these parameters are unique to the explainer that is being used, and they may be mandatory or optional. One of the most important parameters as it is present in all cases is the *backend* of the model being passed. This parameter allows identifying the type of model to extract its prediction function. The other parameter that must be passed when using local explanation methods is the *instance* (or *image*, for image explainers). This is the instance that will be used to generate the local explanation.

Following up, I list and describe the explainers that were implemented as resources of the API, along with their specific parameters. For this, I have classified the explainers according to their data type. Besides the resources related to the explainers, I created an auxiliary resource to display the explanations (*ViewExplanation*) in HTML and PNG format that are stored in the server when a method is executed.

3.2.1 Tabular Explainers

Tabular/Importance: calculates the feature importance of the specified features. It was implemented using the Dalex [12] library. It returns the explanation in JSON format, and URLs to view the bar-plot explanation in HTML and PNG formats. The POST request requires the *model*, *data* and *params* parameters. The *params* object contains the following:

- **backend:** The supported values are 'sklearn' (for Scikit-learn), 'TF1' (for TensorFlow 1.0), 'TF2' (for TensorFlow 2.0), 'PYT' (for PyTorch), and 'Other' for other model implementations.
- **variables:** (Optional) Array of strings with the names of the features for which the importance will be calculated. Defaults to all features.
- **model_task:** (Optional) A string containing 'classification' or 'regression' according to the AI task that the model performs. Defaults to 'classification'.

Tabular/ALE: computes the accumulated local effects (ALE) of a model for the specified features. It was implemented using a submodule from Alibi [18]. It returns the explanation in JSON format, as well as an URL to an explanation plot in PNG format. The POST request requires the *model*, *data* and *params* parameters. The *params* object contains the following:

- **backend:** The supported values are 'sklearn' (for Scikit-learn), 'TF1' (for TensorFlow 1.0), 'TF2' (for TensorFlow 2.0), 'PYT' (for PyTorch), and 'Other' for other model implementations.
- **feature_names:** (Optional) Array of strings corresponding to the columns in the training data.
- **target_names:** (Optional) Array of strings containing the names of the possible classes.
- **features_to_show:** (Optional) Array of integers representing the indices of the features to be explained. Defaults to all features.

Tabular/LIME: this method is based on the original implementation of LIME [5], using the tabular LIME module. It returns the estimated contribution of each feature to the final output. It returns the explanation in JSON format, as well as URLs to HTML and PNG formats. The POST request requires the *model*, *data* and *params* parameters. The *params* object contains the following:

- **instance:** Array representing the feature values of an instance, without including its prediction value.
- **backend:** The supported values are 'sklearn' (for Scikit-learn), 'TF1' (for TensorFlow 1.0), 'TF2' (for TensorFlow 2.0), 'PYT' (for PyTorch), and 'Other' for other model implementations.
- **model_task:** (Optional) A string containing 'classification' or 'regression' according to the AI task that the model performs. Defaults to 'classification'.
- **training_labels:** (Optional) Array of integers representing labels for training data.
- **feature_names:** (Optional) Array of strings corresponding to the columns in the training data.
- **categorical_features:** (Optional) Array of integers representing the indexes of the categorical columns. Columns not included here will be considered continuous.
- **categories_names:** (Optional) Dictionary with integer keys representing the indexes of the categorical columns, each key having as value an array of strings with the names of the different categories for that feature.
- **class_names:** (Optional) Array of strings containing the names of the possible classes.
- **output_classes:** (Optional) Array of integers representing the classes to be explained.
- **top_classes:** (Optional) integer representing the number of classes with the highest prediction probability to be explained.
- **num_features:** (Optional) integer representing the maximum number of features to be included in the explanation..

Tabular/SHAP: this resource uses the original implementation of SHAP [7] to estimate the contribution of each feature to the final prediction. The explanation is returned in JSON, HTML and PNG format. Both the HTML and PNG responses contain a force plot of the explanation. The POST request requires the *model*, *data* and *params* parameters. The *params* object contains the following:

- **instance:** Array representing the feature values of an instance, without including its prediction value.
- **backend:** The supported values are 'sklearn' (for Scikit-learn), 'TF1' (for TensorFlow 1.0), 'TF2' (for TensorFlow 2.0), 'PYT' (for PyTorch), and 'Other' for other model implementations.

- **feature_names:** (Optional) Array of strings corresponding to the columns in the training data.
- **output_names:** (Optional) Array of strings containing the names of the possible classes.
- **output_index:** (Optional) Integer representing the index of the class to be explained. This parameter is ignored for regression models. The default index is 0.

Tabular/DicePublic: generates counterfactuals using the public data interface from DiCE [24]. The explanation is returned in JSON, HTML and PNG format. The POST request requires the *model*, *data* and *params* parameters. The *params* object contains the following:

- **instances:** Array of arrays, where each one represents an instance given its feature values, **including** the target class.
- **backend:** The only supported values are 'sklearn' (for Scikit-learn), 'TF1' (for TensorFlow 1.0), 'TF2' (for TensorFlow 2.0), 'PYT' (for PyTorch).
- **method:** The method used for counterfactual generation. The supported methods are: 'random' (random sampling), 'genetic' (genetic algorithms),and 'kdtrees'.
- **cont_features:** Array of strings containing the name of the continuous features. Features not included here are considered categorical.
- **features_to_vary:** Either a string 'all' or a list of strings representing the feature names to vary.
- **desired_class:** Integer representing the index of the desired counterfactual class, or 'opposite' for the closest similar prediction.
- **num_cfs:** Number of counterfactuals to be generated for each instance.
- **permitted_range:** (Optional) JSON object with feature names as keys and permitted range in array as values.
- **continuous_features_precision** (Optional) JSON object with feature names as keys and decimal precision values as values.

Tabular/DicePrivate: this method generates counterfactuals using the private data interface from DiCE [24]. This means that is not necessary to pass the training data to explanation method, as opposed to DicePublic. The explanation is returned in JSON, HTML and PNG format. The POST request requires the *model* and *params* parameters. The *params* object contains the following:

- **instance:** JSON object representing the instance of interest with attribute names as keys, and feature values as values.
- **backend:** Currently, the only supported backend for the private data interface is 'TF2' (for TensorFlow 2.0). However, I decided to keep the backend parameters as the developers of DiCE expect to expand the scope of the interface to other models.

- **method:** The method used for counterfactual generation. The supported methods for private data are: 'random' (random sampling) and 'genetic' (genetic algorithms).
- **features:** JSON Object with feature names as keys and arrays containing the ranges of continuous features, or strings with the categories for categorical features.
- **outcome_name:** Name of the target column
- **type_and_precision** Integer representing the index of the desired counterfactual class, or 'opposite' for the closest similar prediction.
- **features_to_vary, desired_class, num_cfs:** Refer to DicePublic.

Tabular/Anchors: generates anchors by implementing Alibi's anchors submodule for tabular data. [18]. It returns the explanation in JSON format only. The POST request requires the *model*, *data* and *params* parameters. The *params* object contains the following:

- **instance:** Array representing the feature values of an instance, without including its prediction value.
- **backend:** The supported values are 'sklearn' (for Scikit-learn), 'TF1' (for TensorFlow 1.0), 'TF2' (for TensorFlow 2.0), 'PYT' (for PyTorch), and 'Other' for other model implementations.
- **feature_names:** (Optional) Array of strings corresponding to the columns in the training data.
- **categorical_names:** (Optional) Dictionary with indexes of categorical columns as keys and arrays of strings containing the categorical values as values.
- **ohe:** (Optional) Boolean value to indicate if the data is one-hot encoded.
- **threshold:** (Optional) Value from 0 to 1 that indicates the minimum level of precision required for the anchors. The default value is 0.95.

3.2.2 Image Explainers

For image explainer methods, it is not necessary to pass the training data for the generation of explanations. All of the implemented methods are local, therefore it is necessary to pass an individual image to be explained. The image file can be passed as an additional parameter body call. Passing a file is only possible when the model works with black and white or color images that are RGB-encoded using integers ranging from 0 to 255. If the model works with a different image encoding, it is necessary to pass the image as a matrix in the *params* JSON object.

Images/LIME: this method is based on the Image LIME [5] submodule. The POST request requires the *model*, *params*, and *image* (optional) parameters. For each of the classes specified in *params*, it displays the group of pixels that contribute positively or negatively to the prediction of the image class. The *params* object contains the following:

- **image:** Matrix representing the image. It is ignored if an image file was uploaded.
- **backend:** The supported values are 'sklearn' (for Scikit-learn), 'TF1' (for TensorFlow 1.0), 'TF2' (for TensorFlow 2.0), 'PYT' (for PyTorch), and 'Other' for other model implementations.
- **top_classes:** (Optional) integer representing the number of classes with the highest prediction probability to be explained.
- **segmentation_fn:** (Optional) A string with a segmentation algorithm to be used from the following:'quickshift', 'slic', or 'felzenszwalb'. The default algorithm is 'quickshift'.

Images/Counterfactuals: this method generates counterfactuals by implementing Alibi's submodule for counterfactuals for images [18]. The generated counterfactual is an image that is as similar as possible to the original but with a different prediction. The POST request requires the *model*, *params*, and *image* (optional) parameters. The *params* object contains the following:

- **image:** Matrix representing the image. It is gnored if an image file was uploaded.
- **backend:** The supported values are 'sklearn' (for Scikit-learn), 'TF1' (for TensorFlow 1.0), 'TF2' (for TensorFlow 2.0), 'PYT' (for PyTorch), and 'Other' for other model implementations.
- **target_class:** A string containing 'other' or 'same', or an integer denoting the desired class for the counterfactual instance.
- **target_proba:** Value from 0 to 1 representing the target probability for the counterfactual generated.

Images/Anchors: this method generates anchors by implementing Alibi's submodule for anchors for images [18]. The anchor displays the pixels that are sufficient to the model to justify the predicted class. The POST request requires the *model*, *params*, and *image* (optional) parameters. The *params* object contains the following:

- **image:** Matrix representing the image. It is gnored if an image file was uploaded.
- **backend:** The supported values are 'sklearn' (for Scikit-learn), 'TF1' (for TensorFlow 1.0), 'TF2' (for TensorFlow 2.0), 'PYT' (for PyTorch), and 'Other' for other model implementations.
- **threshold:** Value from 0 to 1 that indicates the minimum level of precision required for the anchors. The default value is 0.95.
- **segmentation_fn:** (Optional) A string with a segmentation algorithm to be used from the following: 'quickshift', 'slic', or 'felzenszwalb'. The default algorithm is 'quickshift'.

3.2.3 Text Explainers

Text/LIME: this method was built upon the Text LIME [5] module. The format is similar to the tabular version, showing the contribution of the most important words to the final classification. It returns the explanation in JSON, HTML, and PNG format. The POST request requires the *model* and *params* parameters. There is no need to pass any additional data. The *params* object is also similar to the tabular version:

- **instance:** A string containing the text to be explained.
- **backend:** The supported values are 'sklearn' (for Scikit-learn), 'TF1' (for TensorFlow 1.0), 'TF2' (for TensorFlow 2.0), 'PYT' (for PyTorch), and 'Other' for other model implementations.
- **class_names:** (Optional) Array of strings containing the names of the possible classes.
- **output_classes:** (Optional) Array of integers representing the classes to be explained.
- **top_classes:** (Optional) integer representing the number of classes with the highest prediction probability to be explained.
- **num_features:** (Optional) integer representing the maximum number of features to be included in the explanation..

Chapter 4

CBR System

In this chapter, I covered the design and implementation of a case-based reasoning system that relies on the XAI API developed in the previous chapter to add a layer of personalization to explanation experiences. This approach is covered in the paper *Using Case-based Reasoning for Capturing Expert Knowledge on Explanation Methods* [31]. The main idea behind this system is to recommend the most effective explanation techniques given a particular problem. In a simplified definition, case-based reasoning refers to the process of solving new problems or cases, based on successful solutions applied to similar cases in the past. Considering this definition, a CBR approach naturally fits the problem at hand.

A classic representation of the CBR process is the 4R cycle: [32]

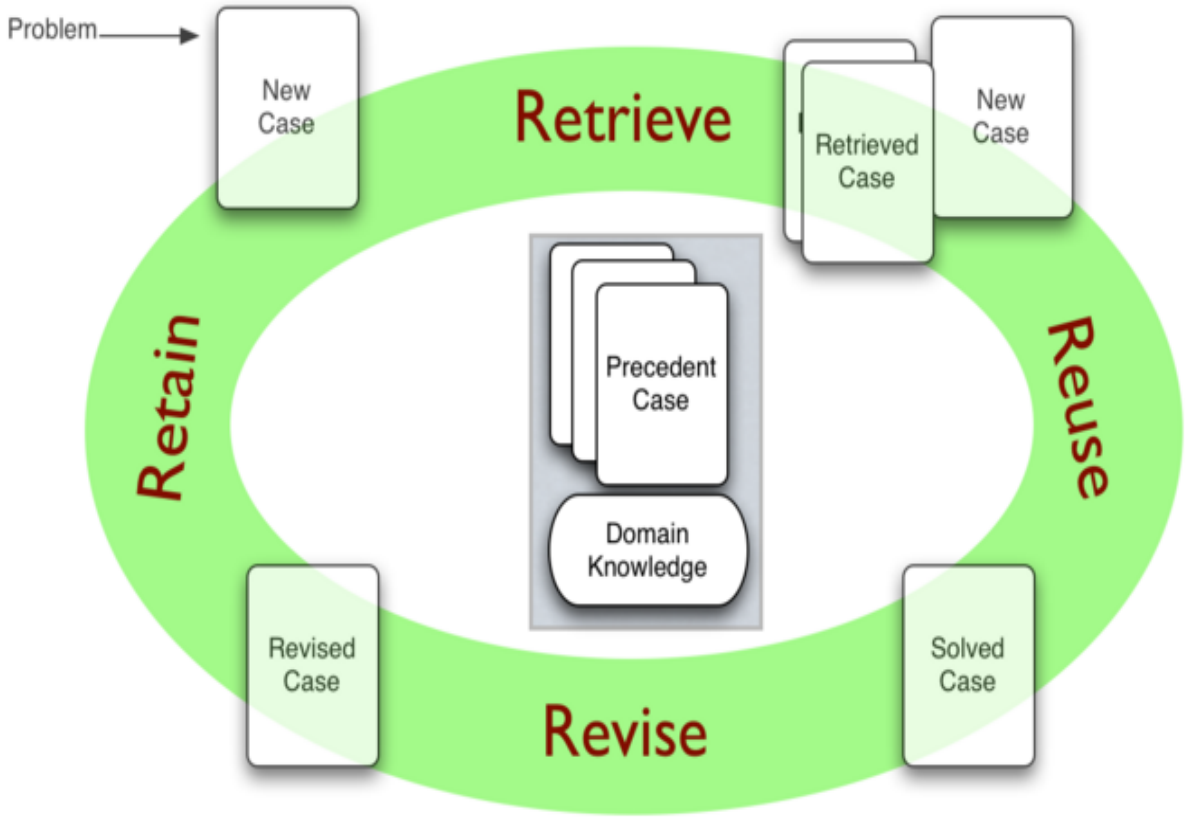


Figure 4.1: The 4R CBR cycle. [32]

1. **Retrieve:** retrieves cases from the case base that have similar characteristics to the target case. Typically, cases contain the description of a problem, the solution applied, and other relevant annotations.
2. **Reuse:** proposes a solution to the target case based on the retrieved cases.
3. **Revise:** test the effectiveness of the proposed solution applied to the target case.
4. **Retain:** capture the resulting experience and store it as a new case in the case base.

For the implementation of the case base, I created a collection in *MongoDB*[33], a NoSQL database system. I chose to use *MongoDB* as it follows a schema-free, document-oriented approach, that provides flexibility on the stored documents. The structure of the cases is defined in Section 4.1 Besides the cases, I created a separate collection that contained the descriptions of the explanation methods available in the API. The purpose of this collection is to ease the retrieval function to filter the explainers that have compatible characteristics with the target case.

To implement the functionalities of the CBR system, I added two new resources to the API: *Retriever* and *Retainer*. The *Retriever* receives a case description and returns a list of explainers sorted by the expected user satisfaction. On the other hand, the *Retainer* receives a new case to insert into the case base. The retrieval process is covered in greater detail in Section 4.2.

4.1 Case Structure

Each case is structured as a tuple $\langle D, S, R \rangle$ containing: (1) a description D of the ML model to be explained; (2) a solution S, that describes the explanation method (or explainer); and (3) a result R, which is the opinion (score) of the users about how good the solution is for this specific description.

The case description D includes the following attributes:

Domain: the domain refers to the field where the AI model is used. Some examples of domains are Medicine, Economics, Social, Security, Entertainment, and Image Recognition.

DataType: the type of data that the *Explainer* accepts as input. It can be text, images, or tabular data.

TrainingData: if the data that was used to train the model is available for the explainer methods to use.

AITask: the artificial intelligence task that the model completes. For this project, the only tasks considered were classification and regression, although realistically, this attribute is extensible to many other AI tasks such as information retrieval, natural language processing, and so on.

ModelBackend: the library or technology used to implement the AI model. These are the python libraries that were mentioned in the previous section: $Sklearn$, $Torch$, and $TensorFlow$.

Description	Domain	Economics
	DataType	Tabular
	TrainingData	Yes
	AITask	Classification
	ModelBackend	Sklearn
	ModelType	ANN
	DomainKnowledgeLevel	Expert
	MLKnowledge	Yes
	ExplanationScope	Local
Solution		Tabular/Anchors
UserScore		6

Table 4.1: Example of the structure of a case.

ModelType: The architecture of the AI model used to execute its *AITask*. Some examples of model types are artificial neural networks (ANN), random forests (RF), and support vector machines (SVM).

DomainKnowledgeLevel: the level of knowledge of the target user about the *AITask*, and the *Domain*. It can be low or expert.

MLKnowledge: if the user has some knowledge of machine learning. This is a yes or no attribute.

ExplanationScope: indicates if the target explanations are global, explaining the whole AI model, or local, explaining a single prediction.

An *explainer* is defined as a solution S that fits the problem description D. The third component in the case structure is the result R, which is a score that represents the overall satisfaction of the user with the explanation generated by the proposed solution through a 7-point Likert scale. An example depicting the case structure is shown in Table 4.1.

4.2 CBR Process

One of the most important aspects of a CBR system is how similar cases are retrieved. In this system, the proposed retrieval function was decomposed into two steps: filtering and sorting. Given a case description D, the filtering step takes into account certain attributes that allow identifying the explanation methods that are compatible with that case. Namely, these attributes are *DataType, TrainingData, AITask, ModelBackend, ModelType* and *ExplanationScope*. This filter guarantees that all the retrieved explainers are valid solutions. For example, consider a random forest regressor that works with tabular data, and that an explanation for an individual prediction is needed. Just by specifying the *DataType* attribute, only the tabular explainers are retrieved. Since the *AITask* is regression, explainers that only work with classification, such as counterfactuals,

will be discarded. Finally, since the target is to get explanations for a particular instance, the *ExplanationScope* will be local, and thus the final retrieved explainers proposed by the system will be *Tabular/LIME* and *Tabular/SHAP*.

During the initial filtering step, I use the case description attributes to guarantee that only the compatible explainers are returned. The output of this initial phase may contain different explainers to solve/explain the model of the case. Each of these explainers can be a solution to many different cases in the case base. I denoted the set of cases sharing the same explainer as a solution with $\mathcal{C}^S$. Logically, some solutions (explanation methods) may be more suitable than others for the case in point. For this reason, the sorting phase arranges the cases in ($\mathcal{C}^S$) according to the following similarity metric:

$$sim(q, c) = \frac{1}{W} \sum_{a \in SimAttr} w_a \cdot equal(q(a), c(a))$$

where q represents the target case, $c \in \mathcal{C}^S$, *SimAttr* represents the following attributes of the case description: *Domain*, *DomainKnowledgeLevel*, *MLKnowledge*, *AItask*, *ModelBackend*, and *ModelType*. The values $w_a \in [0..1]$ are weights that have been computed to obtain the minimum error using a greedy optimization method, and $W = \sum w_a$.

Once the similarity values are calculated for all the cases in $\mathcal{C}^S$, the score (R) of the k most similar cases to the query are averaged to obtain the *Mean Estimated Explanation Utility Score*. This score represents the expected user satisfaction for that explainer. The same process applies to the rest of the compatible solutions: all the cases with that explainer as *Solution* are retrieved creating the $\mathcal{C}^S$ set, their similarity values are calculated, and then the scores of the most similar cases are aggregated to determine the estimated utility value.

Finally, the explainers are ranked according to the utility score and the highest similarity from the $\mathcal{C}^S$ set before being proposed to the user. While this system does not consider the revision step from the traditional CBR approach, it is possible to manually retain new cases and store them in the case base by providing a valid case to the *Retainer* function.

ExplainerName	**ExpectedScore**	**Similarity**
Tabular/Anchors	5.27	0.9
Tabular/DicePublic	5.16	0.9
Tabular/SHAP	3.69	0.8
Tabular/LIME	3.22	0.9

Table 4.2: Example output of the CBR system using the *Retriever* function. The response is based on the target case described in Table 4.1. The proposed solutions are ranked based on the expected score. The similarity value provides an estimation of the reliability of the result.

.

4.3 Case Base Population

To populate the case base of the CBR system with real user input, I elaborated questionnaires related to a series of use cases where the user rates different explanations generated by several alternative explainers. One of the advantages of using questionnaires is that the users do not need to interact directly with the system and worry about aspects such as parameter configuration. In addition, the questionnaires provided users with the background needed about the case and the description of the explainers being applied. The purpose of each use case is to ask the user about their degree of satisfaction with the explanations proposed using a Likert scale (from 1 to 7).

Additionally, for each use case, two questions were included to do some basic profiling of the user that relate to their knowledge in the specific domain and their expertise in machine learning. With this information, it is possible to build a case with the feedback of a user about a specific explanation. In this way, the description of the use case is associated to the explanation method, which is the solution, and the scores given by the users are the result of the case. Next, I describe the use cases presented to the participants according to the case structure described in the previous section.

Use case 1: cervical cancer prediction[1]. The *Domain* of this use case is Medicine, a critical domain where trust and transparency are much needed. For the questionnaire, I proposed two different models: a random forest and a neural network *ModelType* built using sklearn (*ModelBackend*). Their task is to classify (*AITask*) individuals according to their risk of having cervical cancer. These models consider features, represented as tabular data (*DataType*), like their age, the number of sexual partners, and the number of pregnancies, among others. Many different explanation methods were applied (i.e., different *Solutions*). Some of these explanations methods were global (*ExplanationScope*), like Variable Importance and Accumulated Local Effects, and others were local (*ExplanationScope*) such as LIME, SHAP, Anchors, and DiCE.

Use case 2: depression screening[2]. This use case refers to a problem of the psychology field (Medicine *Domain*). In this case, a sklearn (*ModelBackend*) artificial neural network (ANN) (*ModelType*) aims to identify depression in students (classification *AItask*). The machine learning model was based on the study *An Artificial Neural Network for Depression Screening and Questionnaire Refinement in Undergraduate Students* [34]. The proposed model uses tabular data (*DataType*) that was collected through a questionnaire provided to students. The explanation methods (*Solutions*) that were applied in this cases were: Variable Importance, Accumulated Local Effects (global *ExplanationScope*), and LIME, SHAP, and Anchors (local *ExplanationScope*).

Use case 3: cost prediction[3]. In this problem, a random forest (*ModelType*) model predicts the price per square meter of apartments in Poland. This case is considered to belong to the Economics *Domain*. Since the output of the model is a continuous value (price/m^2), this is a regression problem (*AITask*). The model uses tabular data (*DataType*) that describes certain attributes of the apartments: their surface,

[1] https://forms.gle/ctJZx53wRhTb7hMf8
[2] https://forms.gle/2jYBkWgNcWjNKRLs6
[3] https://forms.gle/Kc91FWF9gKgg5yfS6

floor, location, year it was built, etc. The proposed explanation methods (*Solutions*) are the following: Variable Importance, Accumulated Local Effects (global *ExplanationScope*), and LIME, and SHAP (local *ExplanationScope*).

Use case 4: income prediction[4]. In this classification (*AITask*) problem, the aim is to predict whether a person earns more than 50K dollars a year. I used two different neural networks (*ModelType*), one being built with sklearn, and the other one with TensorFlow (*ModelBackend*). The models rely on the characteristics of the individuals in the *TrainingData*, such as their age, marital status, gender, and type of job, among others. To explain the sklearn model, I used variable importance, Accumulated Local Effects (global *ExplanationScope*), and LIME, SHAP, and DiCEPublic (local *ExplanationScope*). To explain a singular prediction simulating that the *TrainingData* was not available, I applied the DiCEPrivate method to the TensorFlow model. The *Domain* of this problem was labeled as Economics.

Use case 5: fraud detection[5]. In this questionnaire, I include the results of explanation methods applied to a random forest (*ModelType*) to classify (*AITask*) a credit-card transaction as fraudulent. The *Domain* of this case was defined as Security. Although the *TrainingData* is available, the meaning of the features is not provided in the dataset due to privacy issues. The model was trained using tabular data (*DataType*). The *Solutions* applied to this case were: variable importance, Accumulated Local Effects (global *ExplanationScope*), and LIME, SHAP, and Anchors (local *ExplanationScope*). The main goal of this domain is fighting against vulnerabilities in critical systems.

Use case 6: social problems identification[6]. This use case is related to the Social *Domain*. The explanation methods proposed *Solutions*) are variable importance, Accumulated Local Effects (global *ExplanationScope*), LIME and SHAP (local *ExplanationScope*). They try to explain the behavior of a Support Vector Machine (*ModelType*) applied to a regression problem (*AITask*) that tries to predict the final grades of Portuguese students. The data is given in a tabular *DataType* format and contains characteristics that define the social environment of the students, such as their age, the education level and job fields of their parents, their alcohol consumption, the grades obtained in previous terms, and many others.

Use case 7: text classification[7]. This model classifies (*AITask*) a newsgroup post on a specific topic. The machine learning model only uses the text (*DataType*) from the post to classify it in topics labeled as religion, autos, and baseball, among other topics. For that reason, the *Domain* of this problem has been identified as Entertainment. Although this domain may not be as critical as some of the previous ones, explanations in the entertainment industry offer many advantages. For example, it is possible to increase the satisfaction of the users or even persuade them to consume new products. The *Solution* that was applied to this use case was LIME (local *ExplanationScope*). The dataset for this problem was obtained from the Scikit-learn [15] documentation.

[4] https://forms.gle/KHXTGbJydXHAHH2p6
[5] https://forms.gle/mFe9ccVhZiLEjk4u6
[6] https://forms.gle/mFe9ccVhZiLEjk4u6
[7] https://forms.gle/KitNg2FnkTbuL3KR6

Use cases 8, 9, 10: image recognition[8] [9] [10]. These three questionnaires are included in the Image Recognition *Domain*. This domain is specific for understanding the prediction of the objects that appear in images. In these questionnaires, I propose toy examples related to the classification (*AITask*) of images (*DataType*) using artificial neural networks (*ModelType*) built on TensorFlow (*ModelBackend*). In questionnaire 8, LIME and Anchors are proposed as *Solutions* for classifications of images of animals. The dataset was extracted from the TensorFlow dataset repository [35]. In questionnaire 9, I propose Anchors and Counterfactuals to classify black and white images of clothing pieces. The data for this use case was obtained from the MNIST Fashion dataset [36]. Finally, in the last questionnaire, I used counterfactuals as the *Solution* to explain the classification of black and white images of handwritten digits, extracted from the MNIST handwritten digits dataset. [37].

[8] https://forms.gle/MCtagTCMB9jiFdgk6
[9] https://forms.gle/YHYga6d9eqLVFvsh7
[10] https://forms.gle/tZxzH8ZyY3VejhVv7

Chapter 5

Results Analysis

In this chapter, I analyze the statistical characteristics of the resulting case base obtained from the questionnaires. I also attempt to identify patterns in the preferences of users when rating explanation techniques. Lastly, I discuss the validation and performance of the optimized CBR system.

The case base included a total of 746 cases, where users evaluated the explanations for the domain models using the 11 different explanation methods from the XAI API. A total of 30 different users participated by filling in the questionnaires. Particularly, only 80 cases (10.7%) referred to users with no previous knowledge of machine learning.

The stratified analysis of the cases regarding the application domain and ML model is shown in Figure 5.1. The domains with the most cases are Medicine and Image recognition. There is a higher number of cases for these domains because there were more use cases associated with them. Regarding the distribution of models, there is a majority of cases applied to artificial neural networks (ANN).

The total of cases for each explainer and datatype is presented in Figure 5.2. Regarding the explainers, all the solutions are guaranteed to be valid methods for their case descriptions thanks to the filtering step of the retrieval phase that was applied when elaborating the questionnaires to evaluate the use cases. As expected, global methods, that can be applied in almost any case involving tabular data (ALE and Feature Importance), have more cases than most of the local explainers. It is important to highlight that the number of cases per

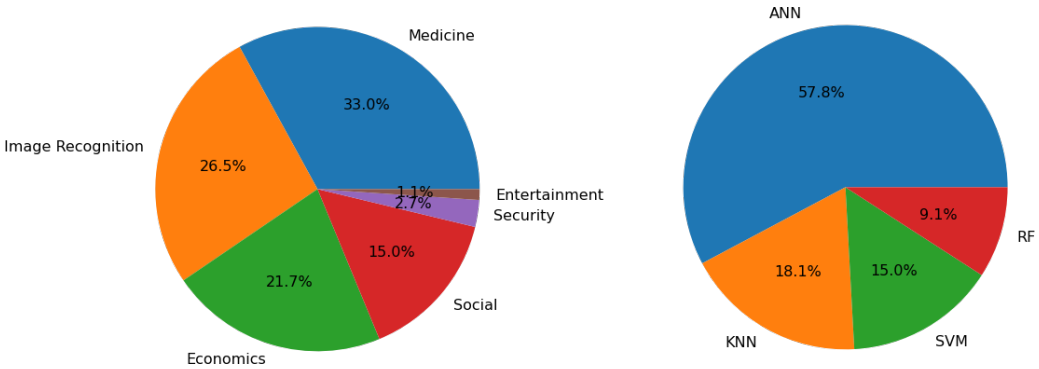


Figure 5.1: Stratified analysis of cases per domain (left) and ML model (right).

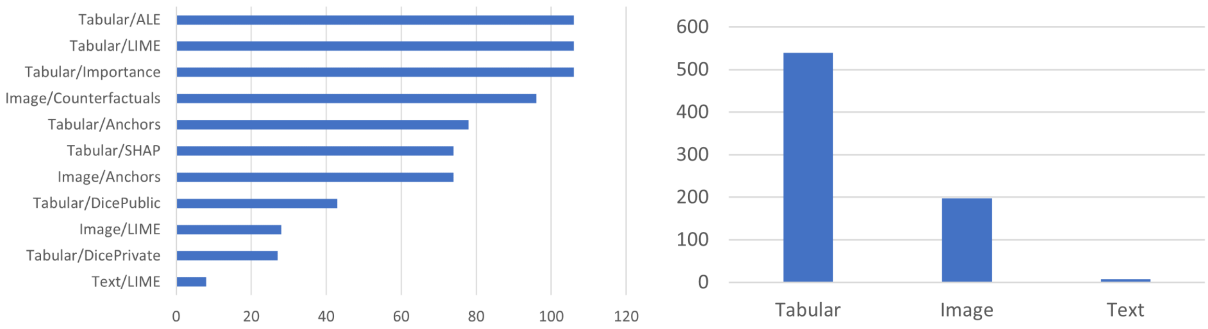


Figure 5.2: Stratified analysis of the number of cases per explainer (left) and data type (right).

explainer is directly proportional to the number of cases of the data type such explainer uses. It is also worth noting that there are few cases representative of the text data type, mainly because I only used one method to generate explanations for text-related models (LIME), but also because the Entertainment domain, where this type of explainer was used, did not count with enough participation.

For Figure 5.3, I obtained the average user score for each explainer method. As a reminder, the scores assigned by the user range from 1 to 7. Although LIME for text data has the higher score, this is the result with the least reliability since the number of cases where this explainer was used is not significant enough. However, the tabular global methods, ALE, and Variable Importance were the next best-rated explainers with an average score of 5.53 and 5.27, respectively. This does not mean that users disliked local explainers. In fact, most of the local explainers have an average score above the neutral mark of 4. However, explainers such as SHAP for tabular data, and Anchors for images, received the worst feedback with a mean score below 4.

The same pattern is identified upon analyzing the mean scores per explainer grouping by the different domains in Table 5.1. Again, the global methods ALE and Variable Importance are the best-rated ones by the users in most of the domains. However, there is an exception where ALE is the worst-rated explainer in the Security domain. One possible explanation for this is that in the only model explained in this domain the meaning of the features was not provided because of data privacy reasons.

Figure 5.3: Average user score by explainer.

Domain	Solution	UserScore
Social	Tabular/ALE	6.07
	Tabular/Importance	5.89
	Tabular/LIME	5.25
	Tabular/SHAP	4.50
Security	Tabular/Importance	4.80
	Tabular/Anchors	3.60
	Tabular/LIME	3.40
	Tabular/ALE	3.20
Medicine	Tabular/Importance	5.34
	Tabular/LIME	4.89
	Tabular/ALE	4.82
	Tabular/Anchors	4.56
	Tabular/DicePublic	3.93
	Tabular/SHAP	3.56
Image Recognition	Image/LIME	4.89
	Image/Counterfactuals	4.31
	Image/Anchors	3.82
Entertainment	Text/LIME	5.62
Economics	Tabular/ALE	5.59
	Tabular/Importance	5.59
	Tabular/Anchors	5.07
	Tabular/DicePublic	5.07
	Tabular/DicePrivate	4.55
	Tabular/LIME	3.59

Table 5.1: Mean score per explainer by domain.

The low score given by the users may imply that this method is not particularly helpful when the intrinsic meaning of the attributes is unknown. Nevertheless, this is one of the domains with the lowest number of cases in the case base, and the standard deviation is considerably higher. Regarding the local tabular methods, LIME, DiCE (counterfactuals), and Anchors were preferred over SHAP in all the domains. As for the image explainers, LIME was the better-rated explainer, followed by image counterfactuals. Anchors for images did not prove to be helpful for the users and its average score fell below the neutral mark of 4.

A similar outcome is obtained when grouping by the AI Task, as shown in Table 5.2. However, it is worth noting that the same explainers obtained a considerably higher score when used for regression tasks than for classification. One reason for this may be that the regression models proposed in the questionnaires are easier to interpret than the classification ones since the value of a feature is proportional to the predicted value, while classification models work with probabilities. However, it is worth pointing out that the sample size was not large enough as only two of the models presented in the questionnaires were regressors.

AITask	Solution	UserScore
Regression	Tabular/ALE	6.07
	Tabular/Importance	5.89
	Tabular/LIME	5.25
	Tabular/SHAP	4.50
Classification	Text/LIME	5.62
	Tabular/Importance	5.39
	Tabular/ALE	4.98
	Image/LIME	4.89
	Tabular/Anchors	4.67
	Tabular/DicePublic	4.65
	Tabular/DicePrivate	4.55
	Tabular/LIME	4.34
	Image/Counterfactuals	4.31
	Image/Anchors	3.82
	Tabular/SHAP	3.56

Table 5.2: Mean user score per explainer by AI task.

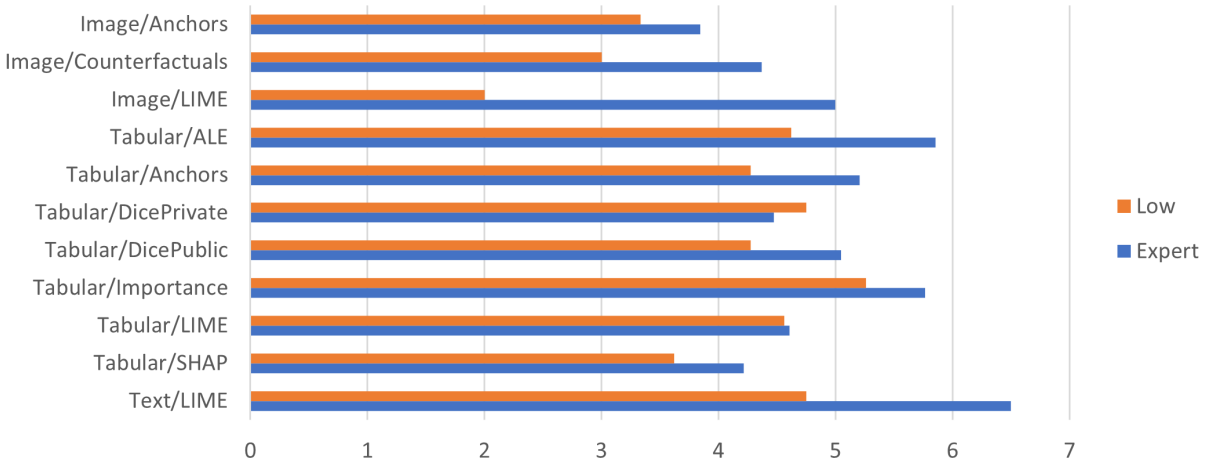


Figure 5.4: Mean user score per explainer according to the users' knowledge of the domain.

Figure 5.4 depicts the mean score given to the explainers depending on the previous domain knowledge of the users. One of the main aspects is that expert users in the proposed domains tend to evaluate the explainers more positively. Although there seems to be a greater disparity for image explainers, it is important to highlight that only one user claimed not to have knowledge in the Image Recognition domain, so it would be incorrect to make interpretations about the suitability of this explainer solely for expert users. However, in domains involving tabular data, the number of users with little knowledge about the domain is more similar to the number of expert users. Thus, the results obtained are more reliable and although the score distance between these types of users is lower, users with low domain knowledge give lower scores to the proposed explanations.

Regarding the CBR system, I validated and tested its performance. From the original case base, 15% of the cases were used as the test set, and the rest represented the case base used by the CBR system. Since each case from the test set was composed of the case description, solution, and user score (from 1 to 7), I calculated the predicted score of that explainer by feeding the case description from the test set to the retrieval function. Then, the expected score predicted by the system for each case was compared with the actual feedback given by that user to the explainer in point. This process was repeated with all the test cases to obtain the mean error.

Lastly, different weight configurations of the similarity functions were tested to minimize the mean error of the system. In Figure 5.5, the absolute error distribution of the optimized system is displayed. The mean absolute error was 1.03 with a standard deviation of 0.83.

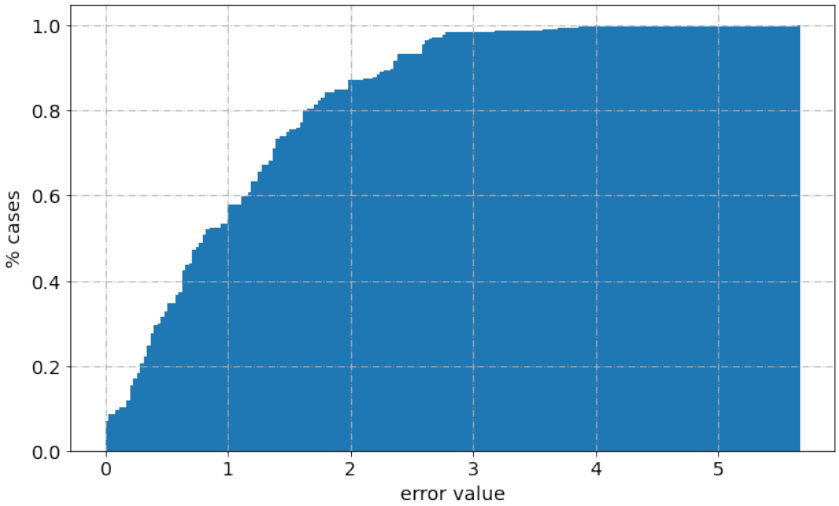


Figure 5.5: Cumulative histogram displaying the absolute error distribution of the CBR system. The y-axis represents the percentage of cases having the error value represented by the x-axis.

Chapter 6

Conclusions and Future Work

6.1 Objectives Review

At the beginning of this paper, I defined the main objectives that were to be accomplished throughout the development of this project. Also, I proposed a set of tasks to reach to achieve each of the objectives. As part of the conclusion to this work, I discuss in this section the accomplishment of the objectives and proposed tasks.

1. **Review the state-of-the-art XAI methods and libraries:**

   As described in chapter 2, I was able to review and test a wide variety of the most used XAI explanation methods. The majority of these methods were fully model-agnostic, and many of them applied to more than one type of data. Learning about their core functioning was a crucial task to the subsequent development of the API and CBR system, as these methods were the basic building units of both tools. Moreover, testing them with different use cases allowed me to identify the main limitations of some of them.

   Regarding the XAI libraries, reviewing them was a useful approach to knowing the state of the current tools in the field. Defining the evaluation criteria was a key aspect to identify some of the shortcomings of these tools. Besides the knowledge acquired thanks to this review, the most valuable insight gained is that, as of now, there is not a multi-purpose tool or library that focuses on the personalization of the explanations proposed. Furthermore, even the libraries with the most explainers left out explanation methods that were available in other libraries. When looking to implement XAI methods, having to install and test different libraries can be an important setback. This fact reinforces the need for a unifying tool that comprises most of the explanation methods in a single place, while also considering the personalization aspect of the explanations proposed.

2. **Develop a unified API that implements the reviewed XAI tools**

   In chapter 3, I covered the development of an API that implemented the XAI methods reviewed before. The main goal behind this API was to put together the greatest number of explanation methods in a single tool. I was able to build this API using Flask as a web framework. The final version of the API consisted of a total of 11 methods, that were classified according to the type of data that the

model used.

One of the key challenges of building this API was to define the attributes that should be passed in each of the calls. While the methods from the libraries already defined the parameters to be used, some of those parameters were not relevant in the schema of a web API. Also, some methods required the use of additional parameters that were not considered in their original implementation to allow greater flexibility in the explanations generated. Finally, all the methods were documented and a Docker image of the API was created to make the tool available to other users and developers.

3. **Develop a case-based reasoning system based on the previously built API that recommends the most suitable explanation methods:**

   One of the major goals of this project was to provide explanation methods to end-users and developers that were more appropriate for them. While this is a very complex task since many factors take place in the effectiveness of an explanation, I achieved this objective through a simple approximation by building a case-based reasoning system. In the proposed system, users with particular case descriptions rate their satisfaction after using a specific explanation method to explain their model. The case description contains the information used to filter the compatible methods and estimate the best one according to users with similar characteristics. The proposed CBR system was built upon the API from the previous section, to provide numerous explainers while also offering methods that are more likely to be helpful to the end-user, based on previous experiences from other users.

   Once the additional API resources for the retrieval of explanation methods were implemented, the last task was to populate the case base. To achieve this, I prepared a set of use cases that were described according to the case structure, and then organize a series of questionnaires to gather the feedback of users on the proposed explanations generated using the explainers from the API. Finally, the information given by the users was combined with the description of the use cases to populate the case base of the CBR system and conclude its development.

4. **Evaluate the case base and test the performance of the CBR system**

   The last of the proposed objectives consisted in analyzing the data contained in the case base and testing how the final CBR system performed. From the evaluation of the case base, the most notorious aspect was that most of the users that participated in the questionnaire preferred global explanation methods over local ones. The other pattern that I identified from this analysis was that users that were experienced in a domain gave more positive feedback to the explanations proposed. While these behaviors apply in this case, the limited number of participants with different profiles was one of the main setbacks I faced when interpreting the results. Because of this, along with other factors discussed in the (FUTURE WORK) section, the obtained results should not be extrapolated to all situations. Regardless, the collected cases allow me to grasp valuable insight that hints towards the explanation methods that are preferred by users.

   Regarding the performance of the model, I validated and tested the CBR system by partitioning the case base in test and train data. Using these data sets, I measured the performance of the system by calculating the mean error of the expected score for

the proposed explainer against the actual score that the user gave to that explainer. Lastly, I configured the similarity function weights to minimize the error of the model and thus improve its performance.

6.2 Future Work

Although I was able to accomplish all of the initially defined objectives, many other milestones could also be considered to expand the reach of the tools presented in this project. In this closing section, I covered some possible work paths that can be followed to improve the functionalities of the API and the CBR system.

One of the most straightforward ways to continue the development of this project consists in adding more explanation methods to the API. While I covered different model-agnostic methods, the only way to build a multi-purpose tool that comprises most of the available explanation methods is through the collaboration of the community of developers. Any developer can complement the functionalities of the current API by implementing new explanation techniques according to the general schema that I followed throughout the development phase.

A different perspective would be to expand the information contained in the case base. This task can be approached in numerous ways. For instance, new use cases can be added and create new questionnaires for users to fill in. Doing so would increase the reliability of the results derived from the case base. Another approach would be to look for a wider variety of users to gather their experiences. One of the limitations I experienced was that almost all users had previous experience with machine learning. Because of this, it was not possible to identify patterns related to users without a machine learning background.

Lastly, the case structure could be expanded to consider more attributes when calculating the similarity of cases in the retrieval phase of the CBR process. Some examples of attributes to consider are the number of features used in a tabular model or the dimensions of the images used by image-oriented models. Incorporating representative attributes into the case structure would increment the specificity of cases and could potentially improve the performance of the CBR system.

Appendices

Appendix A

Using the API with Postman

This quick guide illustrates how to launch the Flask server and make requests to any of the explanation methods in the API using Postman. The code, requirements, Dockerfile, and example use cases are available in the project's repository.

A.1 Launching the Server

A.1.1 Using Python

1. Clone the repository.
2. From the root folder, create a virtual environment for the installation of the required libraries with:

```
python -m venv .
```

3. Use pip to install the dependencies from the requirements file.

```
pip install -r requirements.txt
```

4. Once all the dependencies have been installed, access the XAI API folder and execute the script with:

```
python app.py
```

A.1.2 Using Docker

1. Clone the repository.
2. From the root folder, execute the following command to build a Docker Image:

```
docker build -t <tag_name> .
```

   The associated tag name you choose will be used to refer to the docker image that will be generated.

3. Run the container to launch the server. The -p option maps the port from the container to the real system, to allow making requests to the server.

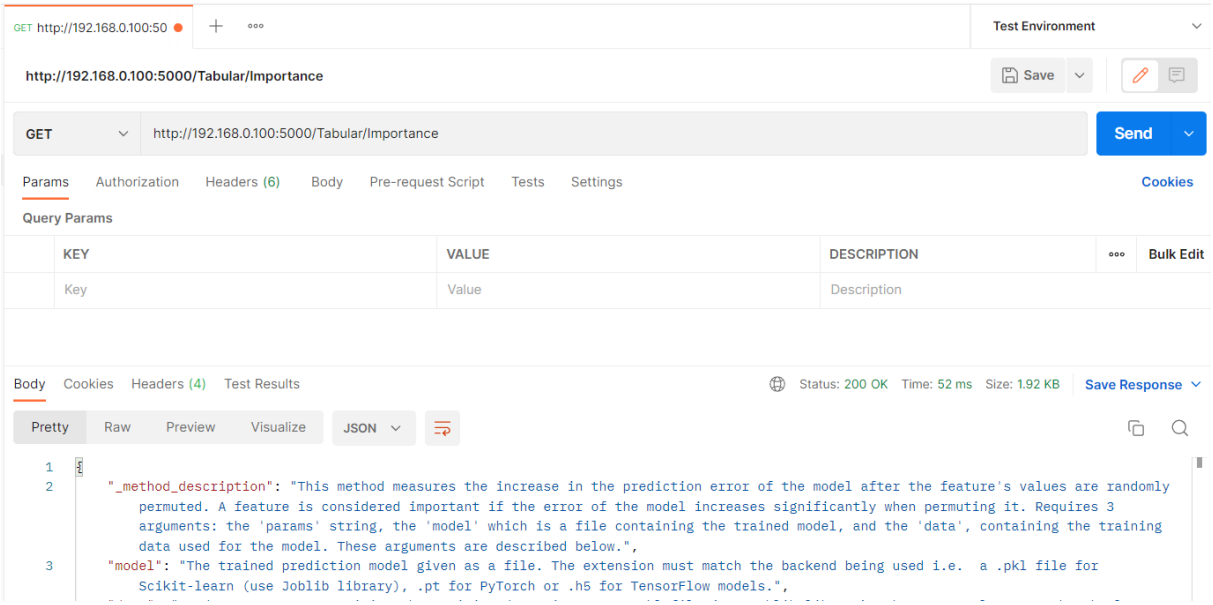
```
docker run -p 5000:5000 <tag_name>
```

A.2 Making Requests

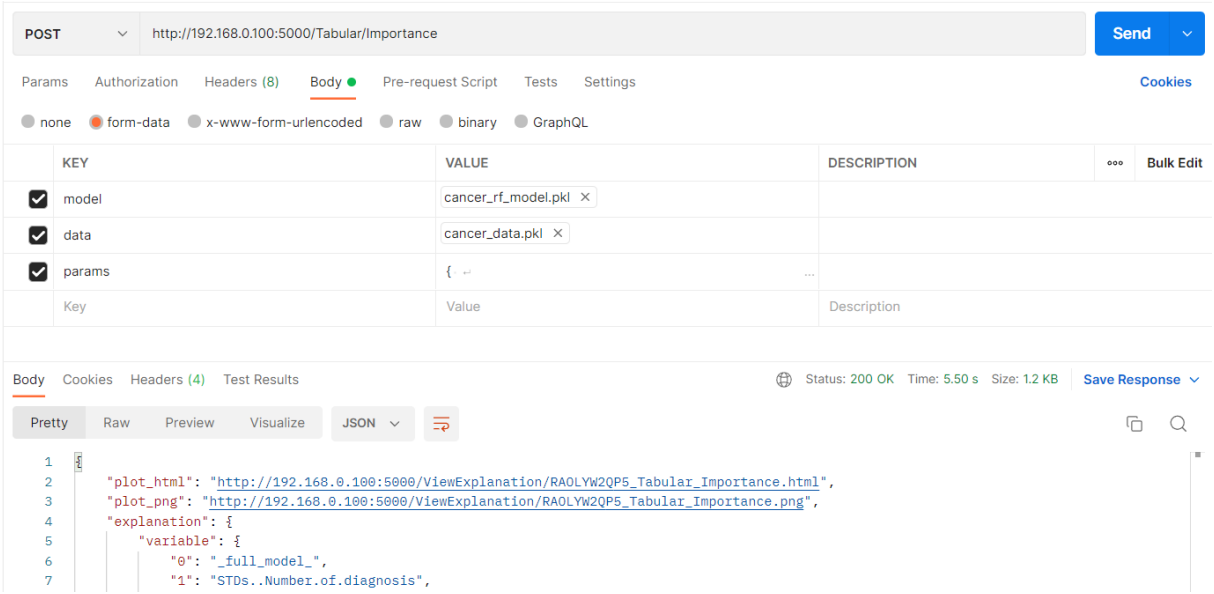
If the server was launched correctly, a similar message to the one in the image should appear, meaning that it is ready to receive requests to the specified address and port.

```
WARNING:werkzeug: * Running on all addresses.
   WARNING: This is a development server. Do not use it in a production deployment.
INFO:werkzeug: * Running on http://192.168.0.100:5000/ (Press CTRL+C to quit)
```

1. To make requests, open Postman and go to *My Workspace* > *File* > *New Tab*.
2. To get information about how to use a specific method, we can send a GET request. In the URL bar, specify the address and port of the server, followed by the name of the method, and send the request. The response is displayed in the bottom part of the console. For example, for Tabular/Importance:



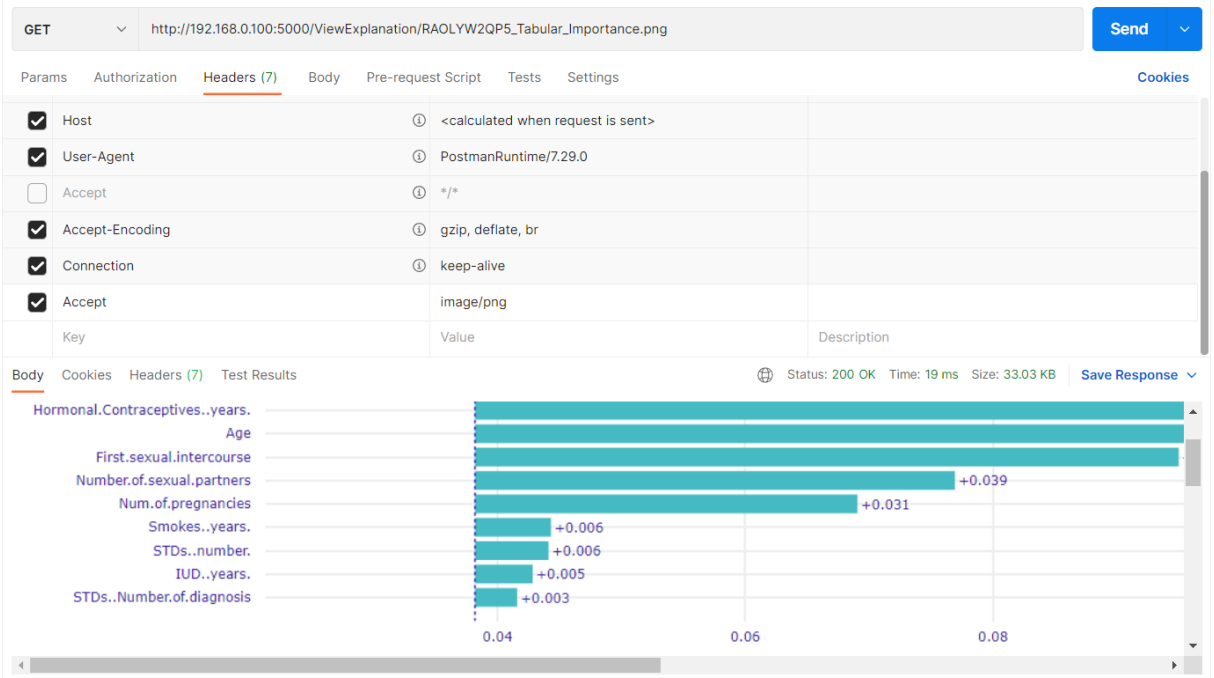
3. To execute the methods, we have to make a POST request. To do so, change the request type to POST and go to *Body* > *form-data*. Here is were we specify the required parameters, such as the *model* and *data* files, and the *params* object. In this example, I am using the cancer use case model and data. The only parameters included in the *params* object were the *backend* (sklearn) and the *model_task* (classification).



A.2.1 Visualizing Explanations

The responses to HTTP requests are given in JSON format. However, most of the methods return responses that also contain the URLs to plots or graphs of the explanations in HTML or PNG format. Before accessing the explanations, it is necessary to change the default JSON mime-type.

1. To visualize these explanations, click on the URL in the response. It will open a new request tab with the specified URL.
2. Go to *Headers* and disable the *Accept* attribute.
3. Add a new header with the same name, *Accept*, as a key, and specify the value according to the type of file you are trying to access. For .png files, specify *image/png*. For .html files, specify *text/html*. Finally, send the request.



Bibliography

[1] European Parliament. *Artificial intelligence: questions of interpretation and application of international law European Parliament resolution of 20 January 2021 on artificial intelligence: questions of interpretation and application of international law in so far as the EU is affected in the areas of civil and military uses and of state authority outside the scope of criminal justice (2020/2013(INI)).* https://www.europarl.europa.eu/doceo/document/TA-9-2021-0009_EN.pdf. 2021.

[2] T. Miller. "Explanation in Artificial Intelligence: Insights from the Social Sciences". In: *CoRR* abs/1706.07269 (2017). arXiv: 1706.07269. URL: http://arxiv.org/abs/1706.07269.

[3] C. Molnar. *Interpretable Machine Learning. A Guide for Making Black Box Models Explainable*. 2nd ed. 2022. URL: https://christophm.github.io/interpretable-ml-book.

[4] *iSee: Intelligent Sharing of Explanation Experience by Users for Users*. https://isee4xai.com/. Coordinator: Belén Díaz Agudo.

[5] 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*. New York, NY, USA: Association for Computing Machinery, 2016, pp. 1135–1144. ISBN: 9781450342322. DOI: 10.1145/2939672.2939778.

[6] M. T. Ribeiro, S. Singh, and C. Guestrin. "Anchors: High-Precision Model-Agnostic Explanations". In: *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, AAAI-18*. Ed. by S. A. McIlraith and K. Q. Weinberger. AAAI Press, 2018, pp. 1527–1535.

[7] S. Lundberg and S.-I. Lee. *A Unified Approach to Interpreting Model Predictions*. 2017. arXiv: 1705.07874 [cs.AI].

[8] J. H. Friedman. "Greedy function approximation: A gradient boosting machine." In: *The Annals of Statistics* 29.5 (2001), pp. 1189–1232. DOI: 10.1214/aos/1013203451.

[9] D. W. Apley and J. Zhu. *Visualizing the Effects of Predictor Variables in Black Box Supervised Learning Models*. 2019. arXiv: 1612.08468 [stat.ME].

[10] S. Verma, J. Dickerson, and K. Hines. *Counterfactual Explanations for Machine Learning: A Review*. 2020. arXiv: 2010.10596 [cs.LG].

[11] A. Dhurandhar, P.-Y. Chen, R. Luss, *et al. Explanations based on the Missing: Towards Contrastive Explanations with Pertinent Negatives*. 2018. arXiv: 1802.07623 [cs.AI].

[12] H. Baniecki, W. Kretowicz, P. Piatyszek, *et al.* "dalex: Responsible Machine Learning with Interactive Explainability and Fairness in Python". In: *Journal of Machine*

Learning Research 22.214 (2021), pp. 1–7. URL: http://jmlr.org/papers/v22/20-1473.html.

[13] J. M. Darias, B. Dıaz-Agudo, and J. A. Recio-Garcıa. "A Systematic Review on Model-agnostic XAI Libraries". In: *Workshops Proceedings for the 29th International Conference on Case-Based Reasoning co-located with the 29th International Conference on Case-Based Reasoning (ICCBR 2021), Salamanca (Spain) / Online, September 13-16, 2021*. Ed. by H. Borck, V. Eisenstadt, A. A. Sánchez-Ruiz, *et al.* Vol. 3017. CEUR Workshop Proceedings. CEUR-WS.org, 2021, pp. 28–39. URL: http://ceur-ws.org/Vol-3017/96.pdf.

[14] Martín Abadi, Ashish Agarwal, Paul Barham, *et al. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/.

[15] F. Pedregosa, G. Varoquaux, A. Gramfort, *et al.* "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[16] A. Paszke, S. Gross, F. Massa, *et al.* "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach, H. Larochelle, A. Beygelzimer, *et al.* Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[17] H. Nori, S. Jenkins, P. Koch, *et al.* "InterpretML: A Unified Framework for Machine Learning Interpretability". In: *arXiv preprint arXiv:1909.09223* (2019).

[18] J. Klaise, A. V. Looveren, G. Vacanti, *et al.* "Alibi Explain: Algorithms for Explaining Machine Learning Models". In: *Journal of Machine Learning Research* 22.181 (2021), pp. 1–7. URL: http://jmlr.org/papers/v22/21-0017.html.

[19] V. Arya, R. K. E. Bellamy, P.-Y. Chen, *et al. One Explanation Does Not Fit All: A Toolkit and Taxonomy of AI Explainability Techniques*. 2019. URL: https://arxiv.org/abs/1909.03012.

[20] K. S. Gurumoorthy, A. Dhurandhar, and G. Cecchi. "ProtoDash: Fast Interpretable Prototype Selection". In: *ArXiv* abs/1707.01212 (2017).

[21] R. Luss, P. Chen, A. Dhurandhar, *et al.* "Generating Contrastive Explanations with Monotonic Attribute Functions". In: *CoRR* abs/1905.12698 (2019). arXiv: 1905.12698. URL: http://arxiv.org/abs/1905.12698.

[22] M. Hind. "Explaining Explainable AI". In: *XRDS* 25.3 (Apr. 2019), pp. 16–19. ISSN: 1528-4972. DOI: 10.1145/3313096. URL: https://doi.org/10.1145/3313096.

[23] *AI Explainability 360 - Demo*. https://aix360.mybluemix.net/data.

[24] R. K. Mothilal, A. Sharma, and C. Tan. "Explaining machine learning classifiers through diverse counterfactual explanations". In: *Proceedings of the 2020 Conference on Fairness, Accountability, and Transparency*. 2020, pp. 607–617.

[25] G. Van Rossum and F. L. Drake. *Python 3 Reference Manual*. Scotts Valley, CA: CreateSpace, 2009. ISBN: 1441412697.

[26] M. Grinberg. *Flask web development: developing web applications with python*. " O'Reilly Media, Inc.", 2018.

[27] *Postman*. https://www.postman.com/.

[28] M. Hostetter, D. A. Kranz, C. Seed, *et al.* "Curl: a gentle slope language for the Web." In: *World wide web journal* 2.2 (1997), pp. 121–134.

[29] D. Merkel. "Docker: lightweight linux containers for consistent development and deployment". In: *Linux journal* 2014.239 (2014), p. 2.

[30] *Joblib*. https://joblib.readthedocs.io/en/latest/.

[31] J. M. Darias, B. Dıaz-Agudo, J. A. Recio-Garcıa, *et al.* "Using Case-based Reasoning for Capturing Expert Knowledge on Explanation Methods". In: 2022.

[32] A. Aamodt and E. Plaza. "Case-based reasoning: Foundational issues, methodological variations, and system approaches". In: *AI communications* 7.1 (1994), pp. 39–59.

[33] K. Chodorow and M. Dirolf. *MongoDB: The Definitive Guide*. 1st. O'Reilly Media, Inc., 2010. ISBN: 1449381561.

[34] M. Orozco-del-Castillo, E. Orozco-del-Castillo, E. Brito-Borges, *et al.* "An Artificial Neural Network for Depression Screening and Questionnaire Refinement in Undergraduate Students". In: Nov. 2021, pp. 1–13. ISBN: 978-3-030-89585-3. DOI: 10.1007/978-3-030-89586-0_1.

[35] *TensorFlow Datasets, A collection of ready-to-use datasets*. https://www.tensorflow.org/datasets.

[36] H. Xiao, K. Rasul, and R. Vollgraf. *Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms*. cite arxiv:1708.07747Comment: Dataset is freely available at https://github.com/zalandoresearch/fashion-mnist Benchmark is available at http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/. 2017. URL: http://arxiv.org/abs/1708.07747.

[37] Y. LeCun and C. Cortes. "MNIST handwritten digit database". In: (2010). URL: http://yann.lecun.com/exdb/mnist/.