



# teex

## a Toolbox for the Evaluation of Explanations

*a bachelor's degree thesis by*  
JESÚS M. ANTOÑANZAS [1]

*under the supervision of*  
DR. YUNZHE JIA [2]

*tutored by*  
DR. MARTA ARIAS VICENTE [1]

## Data Science and Engineering degree

[1] *Universitat Politècnica de Catalunya · BarcelonaTech*

FACULTAT D'INFORMÀTICA DE BARCELONA  
ESCOLA TÈCNICA SUPERIOR D'ENGINYERIA DE TELECOMUNICACIÓ  
DE BARCELONA  
FACULTAT DE MATEMÀTIQUES I ESTADÍSTICA

[2] *University of Waikato*

AI INSTITUTE  
DEPARTMENT OF COMPUTER SCIENCE

September 2, 2021

## Abstract

In the machine learning (ML) community, models are developed, trained and deployed for many applications. Text-to-speech, product and media recommendation, medical aiding, environmental protection and many more are examples of current ML applications. But, more often than not, given the quality requirements for the applications, these models can become very complex. So complex, in fact, that the decisions they take are usually not understandable by humans. These are called black box models.

So, given the clear problem of not trusting models' decisions because of the relevance of their impact and their low transparency, explanation methods / explainers were born with the objective of distilling the factors that black box models take into account when making decisions into 'explanations', which humans can understand. There are many categorizations into which explanation methods fall. For example, the type of explanations they produce, on which models do they work, their mechanisms for extracting information or if they try to characterize a model's whole behaviour (global explanations) or individual predictions (local explanations).

Given the current rise of the field of Explainable AI (XAI), which is driven by necessity, researchers need a tool to easily and swiftly evaluate the performance of state-of-the-art explainer methods. On top of current evaluation techniques such as performing subjective human experiments or manually comparing the quality of explanations, we present a toolbox that will allow to add another layer of credibility to part of XAI research. The toolbox is aimed at the automatic evaluation of local explanations via comparison to ground-truth explanations. Version 1.0 contains several evaluation metrics for different explanation types: saliency maps, decision rules and feature and word importance vectors. Moreover, the library also provides real-world and artificial data with available ground truth explanations so that users can easily benchmark local explainer methods.

**Keywords**— Artificial Intelligence, Explainable Artificial Intelligence, Software

# Contents

<b><u>1 Introduction</u></b>	<b>4</b>
<u>1.1 Into the context</u>	4
<u>1.1.1 AI everything</u>	4
<u>1.1.2 Complexity everywhere</u>	5
<u>1.1.3 On Machine Learning Transparency</u>	5
<u>1.1.4 Explanation methods</u>	6
<u>1.1.5 Evaluation of explanation methods</u>	9
<u>1.2 Previous work on evaluation software</u>	10
<b><u>2 Problem definition: from context to details</u></b>	<b>11</b>
<u>2.1 Context</u>	11
<u>2.2 Motivation &amp; justification</u>	11
<u>2.3 Who teex is for</u>	12
<b><u>3 Work objectives</u></b>	<b>13</b>
<u>3.1 Architectural objectives</u>	13
<u>3.1.1 Availability (O1)</u>	13
<u>3.1.2 Plug &amp; play (O2)</u>	13
<u>3.1.3 Universal API (O3)</u>	13
<u>3.1.4 Explainer-agnostic (O4)</u>	13
<u>3.2 Functional objectives</u>	14
<u>3.2.1 Diverse explanation types (O5)</u>	14
<u>3.2.2 Diverse quality metrics (O6)</u>	14
<u>3.2.3 Data availability (O7)</u>	14
<u>3.3 Quality of life objectives</u>	15
<u>3.3.1 Test design (O8)</u>	15
<u>3.3.2 Complete API documentation (O9)</u>	15
<u>3.3.3 Example library (O10)</u>	16
<u>3.3.4 Open-sourced (O11)</u>	16
<b><u>4 teex: meeting the objectives</u></b>	<b>17</b>
<u>4.1 Architecture</u>	17
<u>4.1.1 Python as the language of choice (O1)</u>	17
<u>4.1.2 PyPI and Python versions (O2)</u>	18
<u>4.1.3 Towards a unified API (O3)</u>	19
<u>4.1.4 Explanations as a proxy for explainer evaluation (O4)</u>	20
<u>4.2 Functionalities</u>	21
<u>4.2.1 The 4 main explanation types (O5)</u>	21

4.2.2	<a href="#">Evaluation metrics &amp; how they can be shared (O6)</a>	23
4.2.3	<a href="#">A whole ecosystem: datasets in <code>teex</code> (O7)</a>	27
4.3	<a href="#">Quality of life measures</a>	28
4.3.1	<a href="#">Unit testing &amp; error handling (O8)</a>	28
4.3.2	<a href="#">Complete API documentation with Sphinx (O9)</a>	29
4.3.3	<a href="#">Basic example notebooks (O10)</a>	31
4.3.4	<a href="#">Making <code>teex</code> public (O11)</a>	31
<b>5</b>	<a href="#">Usage examples</a>	<b>32</b>
5.1	<a href="#">Data classes in <code>teex</code></a>	32
5.2	<a href="#">Evaluation in <code>teex</code></a>	32
5.3	<a href="#">A more extensive evaluation procedure with <code>teex</code></a>	33
<b>6</b>	<a href="#">Conclusion</a>	<b>35</b>
<b>7</b>	<a href="#">Appendix</a>	<b>39</b>

# 1 Introduction

## 1.1 Into the context

This work lies in the edge of current advances in machine learning. So, context is given in this section for the reader to catch up with the intent of the work and its relevance.

### 1.1.1 AI everything

These last few years, everything in the field of AI has gained a lot of traction. From companies trying to incorporate ML into many of their processes to new ones being born with the premise of solving common problems with these technologies. And, whether justified or not, this financial drive makes the field one of the most attractive amongst young people looking into advancing their career prospects. No doubt, AI has transformed industries and has the potential to do so to many others.

One prime example is the medical industry [9]. In the medical industry, many AI solutions are appearing for healthcare management, prognosis prediction or risk, amongst many others.



Figure 1: Examples of applications of AI in the healthcare industry. Source [Customer-Think](#)

Another good example is environmental science. Usually, many issues in this field arise from complex mechanics which depend on many factors at once and are usually constantly evolving, making analyses harder. The high complexity of these problems makes AI a good fit for solving them/some, particularly with the enormous computing power available today, which is one of the big drives behind the rise of AI / ML / DL. To put

into context this work, this project has been proposed as part of a larger project, TAI AO (Time-Evolving Data Science and Artificial Intelligence for Advanced Open Environmental Science, [taiao.ai](https://taiao.ai)), which focuses on increasing the attention of the community to this field by creating cutting edge technologies centered around helping the environment in different ways. From river overflow prediction or automatic critter detection to forest monitoring, plenty of open sourced projects are being born out of this initiative.

These are only two fields in which AI has application, but there are many. Biological sciences [21], the automotive industry (see [Tesla](https://tesla.com)), agriculture or financial systems are other examples of fields that are progressively changing towards being AI-centered.

### 1.1.2 Complexity everywhere

**Note** From this point forward, we are going to refer to Machine Learning (ML) instead of AI. ML is the subset of Artificial Intelligence methods that learn from data without being explicitly programmed to do so. That is, the goal of these methods is to extract knowledge from data. Ultimately, these methods are the ones addressed within this work, so referring only to them makes sense.

New ML methods are regularly being developed and adopted by the public. These new systems are increasingly more able to handle bigger data sets, which more commonly than not means their complexity also increases (Figure 2). That is, although some exceptions apply, simpler systems do not usually scale with the amount of data they are to be trained on. This is a trade-off that, currently, needs to happen if we are to truly take advantage of our connected world. But 'complexity' carries some hidden consequences: in many cases, these ML methods are being implemented as high-stake decision-makers as their capabilities advance. One might think that if a system has more than enough accuracy, it should be safe to apply it on any system, but this should not be the case, because we need to understand the decision-making process of a system if we are to trust it.

### 1.1.3 On Machine Learning Transparency

As we commented in the previous section [1.1.1](#), there has been and there is an increasing number of complex machine learning applications in the real world. The more complex these systems are, the less transparent they are for us humans. They are **black box models**, which means that their decision making process is complexly unknown to us. So, it only makes sense that, with the adoption of its technologies in production environments, more and more standards are being put into place to control the possible biases and inner-workings of these systems. In fact, there are already legal regulations on AI in the European Union (EUR-Lex - 52021PC0206 - EN - EUR-Lex (europa.eu)). These

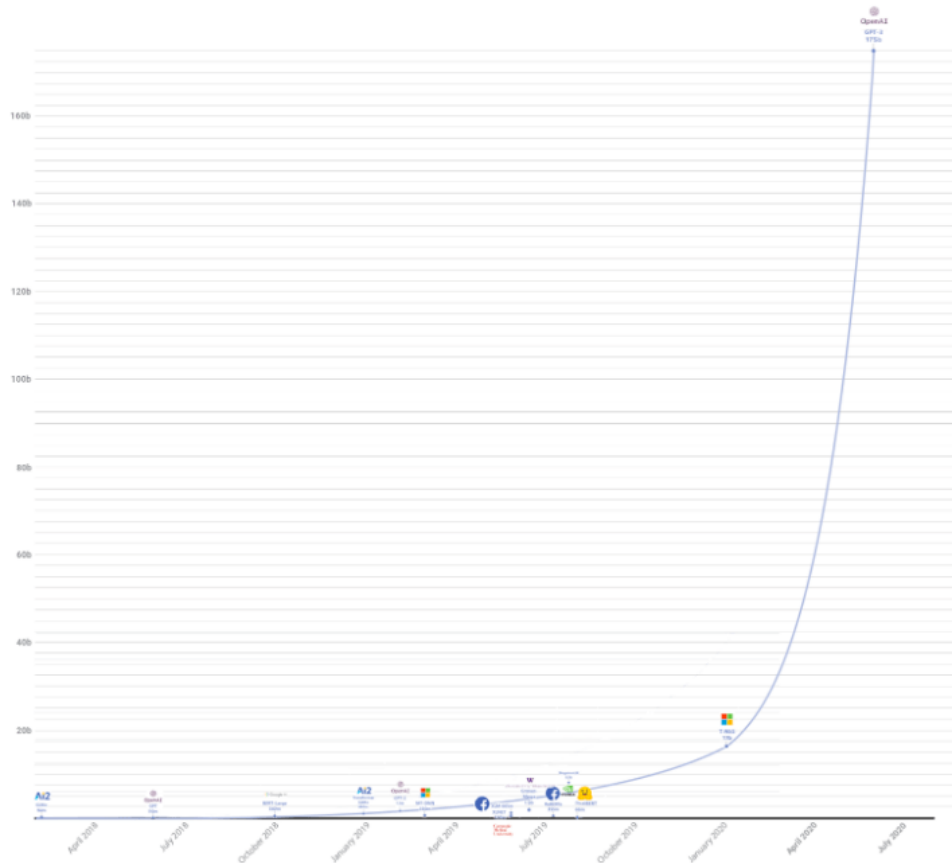


Figure 2: Number of parameters of state-of-the-art language models throughout the years. Note the exponential increase in complexity. [Source](#).

regulations are put into place with the intention of understanding how these models behave so that, ultimately, we can predict their real life performance and know whether they are taking reasonable evidence into account when performing predictions (see figure 3). If not put into place, widespread systems could be adopting decisions that we do not understand, or worse, violating ethical values. For many applications, this is of crucial importance. And, because ML models are becoming more and more complex, their predictions and functioning are usually not human-understandable. For this exact reason explainer/interpretability methods were born. They provide ways for condensing the ‘reasoning’ behind a ML model’s prediction, as well as their global (overall) behaviour in some cases.

#### 1.1.4 Explanation methods

Explanation methods / explainers (from eXplainable Artificial Intelligence or XAI) have been growing out of necessity. More and more methods are being proposed each month [2, 7, 8, 16, 22, 25], because with the rise in popularity of black box ML models, having the

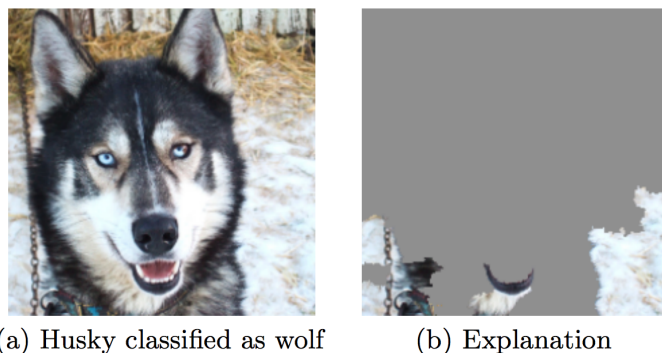


Figure 3: Imagine we have a ML model that tells whether they contain a husky or a wolf. It performs quite well on our test dataset, so we might think that it is ready for production. On a second look, we decide to double check its reasoning with an explainer method and see that, in reality, the classifier only identifies wolves by looking at the snow in an image, regardless of the animal in it. Clearly, there is a fundamental flaw in the classifier’s reasoning and we cannot trust it. Source [19].

system boiled down to its bare essentials is needed for humans to truly trust its predictions. Here we are going to explain what exactly is as an explanation method and their taxonomy.

An explanation method is any algorithm or system that makes accessible to humans the underlying logic of black-box ML / AI agents. An explainer can either distill the overall logic of a model (global interpretability) or the logic behind particular decisions (local interpretability). These boiled-down decision processes are referred to as **explanations**, and can exist in many different forms.

When we talk about explainers in this work, we are specifically referring to black-box explainers (there are some ML methods that are intrinsically interpretable, such as Linear or Logistic Regression), in which the functioning is completely opaque. We now present several high-level categorizations of explainer methods and explanations.

Explainer methods are usually classified as being:

- **Model specific**, when they only work on specific model architectures or family of architectures. For example, GRADCAM [20] only works on CNN-based models.
- **Model agnostic**, when they work on any machine learning algorithm. They basically only look at the input data and derive relations with the output of the model. Modern examples could be SHAP [14] or LIME [19].

The explanations that are generated by the explainer methods also receive their own categorisations. Mainly:



- **Global or model explanation:** where the overall logic of the model is trying to be conveyed to the user.
- **Local or outcome explanation:** where the reasons for a specific outcome are trying to be explained.
- **Black-box inspection:** where the aim is to retrieve a visual representation of the model inner-workings.

And the second one, which is based on the type of explanation that the method produce, as well as the kind of data they are based on, can be found in figure 4.

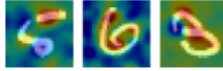




TABULAR	IMAGE	TEXT																																																									
<p><b>Rule-Based (RB)</b> A set of premises that the record must satisfy in order to meet the rule's consequence. <math>r = Education \leq College</math> <math>\rightarrow \leq 50k</math></p>	<p><b>Saliency Maps (SM)</b> A map which highlight the contribution of each pixel at the prediction.</p> 	<p><b>Sentence Highlighting (SH)</b> A map which highlight the contribution of each word at the prediction. the movie is not that bad</p>																																																									
<p><b>Feature Importance (FI)</b> A vector containing a value for each feature. Each value indicates the importance of the feature for the classification.</p> <table border="1"> <tr><td>capitalgain</td><td>0.00</td></tr> <tr><td>education-num</td><td>14.00</td></tr> <tr><td>relationship</td><td>1.00</td></tr> <tr><td>hoursperweek</td><td>3.00</td></tr> </table>	capitalgain	0.00	education-num	14.00	relationship	1.00	hoursperweek	3.00	<p><b>Concept Attribution (CA)</b> Compute attribution to a target "concept" given by the user. For example, how sensitive is the output (a prediction of zebra) to a concept (the presence of stripes)?</p> 	<p><b>Attention Based (AB)</b> This type of explanation gives a matrix of scores which reveal how the word in the sentence are related to each other.</p> <table border="1"> <tr><td></td><td>the</td><td>movie</td><td>is</td><td>not</td><td>that</td><td>bad</td></tr> <tr><td>the</td><td>0.8</td><td>0.2</td><td>0.1</td><td>0.1</td><td>0.1</td><td>0.1</td></tr> <tr><td>movie</td><td>0.2</td><td>0.8</td><td>0.1</td><td>0.1</td><td>0.1</td><td>0.1</td></tr> <tr><td>is</td><td>0.1</td><td>0.1</td><td>0.8</td><td>0.1</td><td>0.1</td><td>0.1</td></tr> <tr><td>not</td><td>0.1</td><td>0.1</td><td>0.1</td><td>0.8</td><td>0.1</td><td>0.1</td></tr> <tr><td>that</td><td>0.1</td><td>0.1</td><td>0.1</td><td>0.1</td><td>0.8</td><td>0.1</td></tr> <tr><td>bad</td><td>0.1</td><td>0.1</td><td>0.1</td><td>0.1</td><td>0.1</td><td>0.8</td></tr> </table>		the	movie	is	not	that	bad	the	0.8	0.2	0.1	0.1	0.1	0.1	movie	0.2	0.8	0.1	0.1	0.1	0.1	is	0.1	0.1	0.8	0.1	0.1	0.1	not	0.1	0.1	0.1	0.8	0.1	0.1	that	0.1	0.1	0.1	0.1	0.8	0.1	bad	0.1	0.1	0.1	0.1	0.1	0.8
capitalgain	0.00																																																										
education-num	14.00																																																										
relationship	1.00																																																										
hoursperweek	3.00																																																										
	the	movie	is	not	that	bad																																																					
the	0.8	0.2	0.1	0.1	0.1	0.1																																																					
movie	0.2	0.8	0.1	0.1	0.1	0.1																																																					
is	0.1	0.1	0.8	0.1	0.1	0.1																																																					
not	0.1	0.1	0.1	0.8	0.1	0.1																																																					
that	0.1	0.1	0.1	0.1	0.8	0.1																																																					
bad	0.1	0.1	0.1	0.1	0.1	0.8																																																					
<p><b>Prototypes (PR)</b> The user is provided with a series of examples that characterize a class of the black box <math>p = Age \in [35, 60], Education \in [College, Master] \rightarrow \geq 50k</math></p> <p><math>p = </math>  <math>\rightarrow</math> "cat"</p> <p><math>p = \dots not\ bad \dots \rightarrow</math> "positive"</p>																																																											
<p><b>Counterfactuals (CF)</b> The user is provided with a series of examples similar to the input query but with different class prediction</p> <p><math>q = Education \leq College \rightarrow \leq 50k</math></p> <p><math>c = Education \geq Master \rightarrow \geq 50k</math></p> <p><math>q = </math>  <math>\rightarrow</math> "3" <math>c = </math>  <math>\rightarrow</math> "8"</p> <p><math>q =</math> The movie is not that bad <math>\rightarrow</math> "positive"</p> <p><math>c =</math> The movie is that bad <math>\rightarrow</math> "negative"</p>																																																											

Figure 4: Explanation types based on data with which the model works. From [4].

In fact, explainer methods could also be categorized by the type of explanation they produce, which would be the same categories as in figure 4. Of course, these are the expla-

nation types that are popular at the moment. The popularity of the field of explainable machine learning makes it so that the list will be constructed upon.

### 1.1.5 Evaluation of explanation methods

**Why evaluate explanations?** Shortly, there are two main reasons for which one would want to evaluate explanations.

1. **Model quality.** Explainer methods exist with the sole purpose of interpreting the inner workings of black box models. With this in mind, if we assume that the logic returned to us by the explainer methods accurately represents the model, we can observe whether the model is making decisions based on good or bad evidence. If we can distinguish between these two states, then we might have an idea of the model performance in real life (i.e. its quality and predictive capabilities).
2. **Explainer quality.** If we are able to quantitatively evaluate the quality of a particular explainer, we can then compare it to other explainer methods. This provides a standard benchmark measure that researchers can use in order to show evidence of superiority between a new explainer method and the previous state-of-the-art, for example.

**Types of evaluation** The types of explainer evaluation procedures are usually categorised as being either quantitative or qualitative.

First, quantitative evaluation measures how close an explanation method can approximate the behaviour of the black box model it's trying to explain, and they are split into two *completeness* measures:

- Completeness with respect to the black box model. Where the evaluation criteria focuses on measuring how much the explainer method approximates the behaviour of the black box model. Some metrics that fall in this category are fidelity [28], stability [26] or faithfulness [3] which measure how much the explainer mimics the behaviour of the model, how similar are explanations for similar observations, and if importance scores of feature importance explanations really are important, respectively.
- Completeness with respect to the task at hand. Where the evaluation criteria is focused on one particular task, like an explanation model in a medical setting or a continual learning setting. Multiple methods can then be defined as valid for that specific task.

Secondly, there are the qualitative evaluation measures, which let us understand the real usability of the explanations. This usability is often bound to the end-user.

Another taxonomisation of the evaluation measures was proposed in [6], dividing the evaluation scene into three sections:

- Application-grounded: where the evaluation requires conducting human experiments in the context of a real task (i.e. doctors performing diagnosis thanks to the explanations provided).
- Human-grounded: where the evaluation requires conducting human experiments in the context of simplified tasks. These experiments can usually be performed with lay humans, as the task does not require domain experts. For example, in [19], the authors perform experiments of this type, amongst others, to evaluate the quality of their explanations
- Functionality-grounded: where a formal definition of interpretability is laid down and proxy tasks are created to evaluate the quality of the explanations, without involving humans. Experiments of this type are also performed in [19].

## 1.2 Previous work on evaluation software

Given the evaluation techniques presented in the previous section, we search for software that already meets **teex**'s objectives and has a proven community. We have manually analysed 54 individual open-sourced projects that are currently active and related to the field of XAI. Out of all 54, only **Captum** [13] (the interpretability library for PyTorch) contains some evaluation metrics. But, as it is not its real focus, only contains two of them: infidelity and sensitivity [29]. Moreover, their API only works on PyTorch models, and, as we will show in section 3, we want to take a broader approach. The other 53 projects usually compile existing evaluation metrics (like [15]), implement explainer methods (like [github.com/sicara/tf-explain](https://github.com/sicara/tf-explain), [12, 17, 27]) or collect XAI resources (like [github.com/EthicalML/awesome-production-machine-learning](https://github.com/EthicalML/awesome-production-machine-learning)). Moreover, we have performed a greedy search amongst hundreds of XAI projects in order to see if they contained keywords such as 'evaluation' and found none that seemed to match our requirements. With this investigation, we arrive at the conclusion that there has been no attempts at developing a general toolbox for the evaluation of explanations, as all of the efforts have been until now addressed to the creation of new methods.

## 2 Problem definition: from context to details

In this section of the work we are going to relate the context of the work with the work itself, all while giving motivation and justifying the work.

### 2.1 Context

The rise in machine learning usage by industries and the development of new techniques by the research community in combination with the huge increase in computational resources and data availability has created a rise in complexity of the systems implemented. These systems are increasingly being trusted for higher-stake decisions such as medical prognosis or social purposes, but their inherent complexity make them opaque to humans: we cannot interpret their decisions. So, explainability methods come into play by boiling down the logic of complex, black box models into explanations that humans can understand. The creation of explanation methods implies that we need to have streamlined and easy-to-use evaluation techniques, so that we can compare them. **teex** aims to be a part of the explanation evaluation scene, in particular of the evaluation of local explanations. The thought process behind **teex** can be visually inspected in figure [5](#).

### 2.2 Motivation & justification

As we have seen in section [1.1.5](#), there is a real need in the community for an explanation evaluation toolbox. Current implementations that can be found online all have the same issue: segmentation. The metrics and methods implemented in each framework are only a few. That is, in order to compute a good number of them one has to adapt to multiple frameworks, which can be very time consuming. Moreover, only a few number of implementations can be found, so the total number of usable methods is low. On top of that, usually, implementations are not model agnostic, which means that they are bound to a specific explainer architecture or explanation type.

All of these problems really come from the fact that if explanation methods are new, evaluation of explanation methods / explanations is an even more recent field, which implies immature implementations roaming the web and allows the opportunity for a unified, general framework that is the hub for evaluating explanations.

But, **why is the evaluation of explanations relevant in the first place?** Intuitively, a ML model that achieves competitive predictive performance and makes decisions based on reasonable evidence is better than one that achieves the same level of accuracy but makes decisions based on circumstantial evidence (figure [3](#)). Given a mechanism for extracting an explanation from a model, we can investigate what evidence the model uses for generating a particular prediction. We consider the explanation to be of high quality if

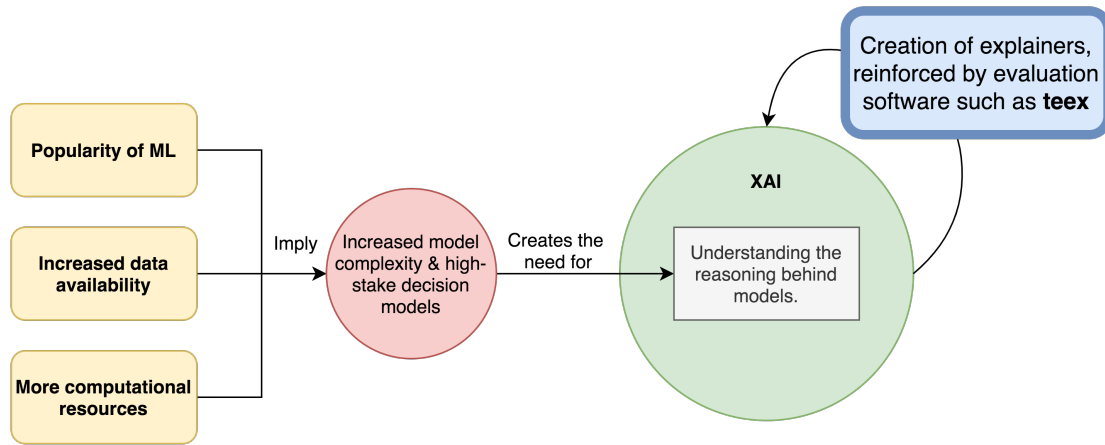


Figure 5: The context of **teex** and what it solves.

it is based on reasonable evidence and of low quality otherwise, and so, we can attempt to use explanation quality for many purposes: debugging, inform selection of an appropriate model, inference of patterns and more. On top of the information that the explanation quality can give, the evaluation of the explanations also provides a way to quantify the performance of explanation methods and compare them, thus allowing for direct benchmarking of new explainer architectures and algorithms.

### 2.3 Who **teex** is for

The general evaluation framework presented in this work is aimed primarily at researchers in the field, not because of difficulty of use, but because the public is not too involved in the task of evaluating explanations. Researchers are the ones that primarily need the tools, as they are the ones that would like to know if new explanation methods perform better than previous ones. This does not mean that a more casual public cannot use the tool, as it can, for example, be a way for users to pick an explainer amongst others for different purposes (i.e. for communicating the performance / justifying the results of models in the best way possible).

## 3 Work objectives

Given the context and the motivation behind the software presented in this work, we lay down the main objectives that needs to satisfy the first release. We believe that these were the bare requirements for the software to be functional, leaving lots of room to spare. We consider that the foundation that these objectives provide is strong enough for there to be interest in the community, which will bring more metrics and methods, in turn generating more interest and so on. We are going to assign each objective a code so that we can later refer to them.

### 3.1 Architectural objectives

First of all, we refer to the architectural objectives, as the first phase is the design of the software. This step is crucial, because, as they say, a good abstraction with mediocre implementation can be usable, but a bad abstraction well implemented is useless.

#### 3.1.1 Availability (O1)

We want **teex** to be available to as many users as possible. For this, we have to develop it into an already established ecosystem of ML software. If the software is published on a platform with many users, then the probability of success is simply greater.

#### 3.1.2 Plug & play (O2)

The installation process of **teex** and its usability to feel natural. If a framework has a steep learning curve, then users are simply not going to bother using it. Moreover, the system installing it should not need to meet a lot of requirements, and the few that exist must be common enough to be met by the majority of systems. These two requirements both contribute to a good user experience.

#### 3.1.3 Universal API (O3)

What we mean by universal API (Application Programming Interface) is that once a user learns the basics of **teex**, everything else should be common sense. That is, the way to make **teex** do all of the things that it can has to be similar enough. This way, **teex**'s API feels 'universal', because everything can be performed inside the same paradigm.

#### 3.1.4 Explainer-agnostic (O4)

Another architectural requirement of **teex** is its agnosticism. If we want the software to be general enough, with the vast amount of explainer methods that exist and that are being proposed each year, **teex** must not be bound to any particular method. That is, it has to be explainer-agnostic. If we can meet this requirement, then the software

will potentially be used in many more cases, and not just in a few ones that meet some explainer requirements.

## 3.2 Functional objectives

Now that the architectural objectives (the foundation of the software) are, we present the functional objectives, or what functionalities we want the software to have (limited to the version presented in this work).

### 3.2.1 Diverse explanation types (O5)

We have seen in section [1.1.4](#) that we can categorize explanation methods by the different types of explanations they create. We will later see how this point and the explainer-agnosticism relate to each other, but for now we want **teex** to be able to work with a diverse range of explanation types (not the same as explainer architectures). If we meet this requirement, then **teex** will be able to be used in the evaluation of explanations in different contexts (different types of models, such as language models or object detection models).

### 3.2.2 Diverse quality metrics (O6)

If the software is going to be used for the evaluation of explanation, then this evaluation has to be performed in a quantitative way. So, we need metrics. In fact, one could argue that the amount and importance of metrics implemented is one of the most important things for the software. Having a broad enough range of options to choose from makes the software more helpful in general.

### 3.2.3 Data availability (O7)

The last functional requirement is related with a fundamental problem for the XAI community. The issue exists because when one computes explanations via explainer methods, we cannot compare them to anything. Sure, our human brain knows what to expect and if something looks wrong in an explanation, but in order to implement automatic evaluation, ground truth explanations need to exist. A ground truth explanation ([figure 6](#)) represents how an explanation produced by an explainer should look like. The problem is that it is often up to manual labour to produce these kinds of explanation, although lately there have been some efforts to publish more and more data with available ground truth explanations. With this in mind, we want **teex** to help the user by providing datasets with available ground truth explanations. In a sense, **teex** can become a 'hub' for this kind of data, which further expands the user base that it could potentially have.



Figure 6: An image and its ground truth explanation: the task for the model is to recognize dogs, so in an ideal scenario the classifier would only look at the highlighted parts of the image on the right.

### 3.3 Quality of life objectives

Quality of life are measures put in place that will lengthen the lifespan of the system. This can be achieved by many means, so we need to be specific about it.

#### 3.3.1 Test design (O8)

We want all of the features added to the system to be stable. In fact, their design will be directly created with this objective in mind. But, sometimes, bugs arise in situations that one did not consider in the beginning. Because of this, we want to have a suite of tests implemented for every functionality of the software. These tests will allow:

1. faster development by allowing bug-checking to be automated.
2. new features to be added while quickly checking if any previous features were broke in the process.

#### 3.3.2 Complete API documentation (O9)

In order for users to have a complete view of what the software can do, we want **teex** to have a complete usage manual: the API documentation itself. This documentation will contain accurate descriptions of every method implemented in the software as well as examples, how they are used, parameter descriptions and data types supported. This API documentation will allow advanced users to extract the full potential of the software as well as make clear certain nuisances in its usage.



### 3.3.3 Example library (O10)

We do not want the API documentation to be the first experience users have of **teex**. Instead, another objective is to create a library of basic usage examples. This library will be simple enough so that new users can understand the purpose and basis of the software and allow them to start using it in their own projects. The library will not contain advanced usage and will not even provide examples of all the functionalities (as of the first released version). For that, users will have to go to the full API documentation.

### 3.3.4 Open-sourced (O11)

The last QOL measure is making the software open-sourced. This is not done arbitrarily: we want **teex** to grow with the community and users to be able to add their own functionalities if they like. We believe that the project might be too big for 1 person to manage, as implementing evaluation metrics, finding datasets, fixing issues and adding requested functionalities are only a few tasks that will need to be repeatedly done once the software is published and the community grows. Moreover, the community will decide in which direction **teex** should move in terms of purpose. Other advantage of open-sourcing the project is forking: versions can branch from the one presented in this work and maintained separately from this one, which is sort of a redundancy measure in case one does not work out. Moreover, another version might focus on other aspects and they all could become part of an ecosystem of explanation evaluation software packages.

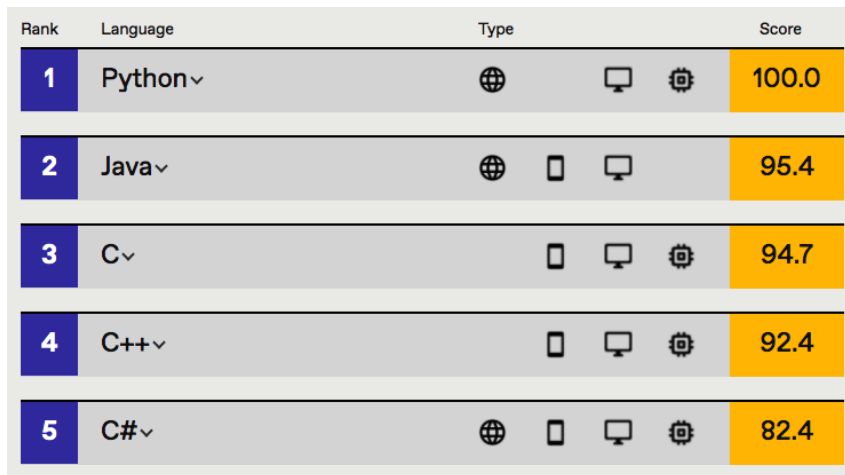
## 4 teex: meeting the objectives

Now that the initial requirements and objectives of the project have been laid out, we show how we have solved and implemented each. Before that, though, we would like to justify a design choice. In particular, we decided that it would not make sense for **teex** to have a graphical interface: in many cases, researchers use software pieces that are available in programming libraries and not as standalone programs. So, because we want **teex** to seamlessly work in the same ecosystem as these libraries, we have decided that **teex** needed to be a software library itself, in which methods are accessible as part of a programming language ecosystem. Also, note that we are not going to go into fine details because (1) it is not our purpose and (2) the document would be hundreds of pages long. Instead, we want the reader to have a general idea of how **teex** is built, its internal mechanism and some other things. With that said, let's move on.

### 4.1 Architecture

#### 4.1.1 Python as the language of choice (O1)

The popularity of Python is undeniable. It contains a plethora of libraries for scientific and industrial purposes. Although its speed is often questioned, its ease of use often trumps other disadvantages it may have. In particular, Python is huge in the field of AI / ML research, XAI included. One quick search shows us that Python is the preferred language for a big part of programmers, even more for researchers (figure 7). This has made us decide to use Python as the programming language of choice for the development of **teex**.



Rank	Language	Type	Score
1	Python	🌐 🖥️ ⚙️	100.0
2	Java	🌐 📱 🖥️	95.4
3	C	📱 🖥️ ⚙️	94.7
4	C++	📱 🖥️ ⚙️	92.4
5	C#	🌐 📱 🖥️ ⚙️	82.4

Figure 7: Top programming languages as of 2021, mainly for IEEE users: a representative sample of our target users. A global programming language ranking does not make sense in this context, as there are many applications unrelated to our purpose and many users that do not fit the target demographic. Source [IEEE language ranking](#)

### 4.1.2 PyPI and Python versions (O2)

In order to make **teex** available to as big of a part of the community as possible we have decided to publish it on the Python Package Index (PyPI). PyPI hosts the majority of available Python packages and provides a super simple command-line-interface for users to install available software. Building the package (transforming separate files into a usable library) so that it can be uploaded to PyPI needs a few steps. We use Python's own **setuptools** library to build the package. First and foremost, the package needs to be properly tested and error-free. Then, two files need to be created into the project:

1. **setup.cfg**. This file needs to contain with all the information for the Python builder to work. The setup file includes information like the package version, author, contact details, labels for the package, relevant URLs or Python version required in the systems. The builder will assign all of this metadata to the built package.
2. **pyproject.toml**. This file is first read when 'building' the package and it contains the requirements for the builder itself, not for the package that is building.

Once the package is locally built, the only thing left is to identify ourselves in PyPI and upload the project via CLI. Note that PyPI will host all of the software versions independently so that users can use whichever they like. Moreover, the name **teex** is forever reserved on the index, even if one happens to delete the project accidentally. A direct link to the PyPI **teex** entry is [pypi.org/project/teex/](https://pypi.org/project/teex/).

All in all, the huge advantages of having the software hosted on PyPI is that all Python users use it and its ease of installation (figure 8).

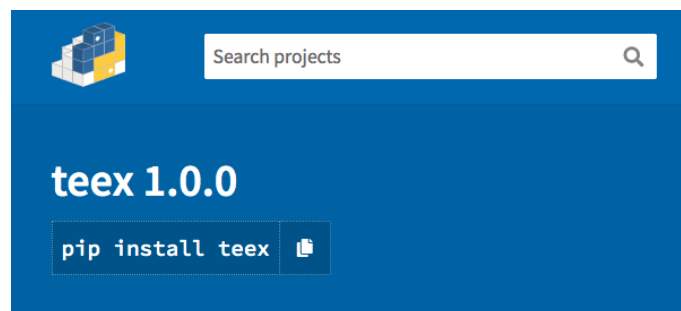


Figure 8: **teex** as portrayed in its PyPI entry. A single line of code in the terminal and the library is ready to import and use in any project (`pip install teex`).

In order to be more accessible to users, we have decided to support versions of Python  $\geq 3.6$ . This has been done via builtin tools in the IDE used to code the software: PyCharm. It automatically tests which versions are compatible with the software that one is writing, and we have made sure that all versions from 3.6 are good to go. The current version is

3.10, so we believe that almost all of our users will have a compatible version. Moreover, this is the version that the most popular AI packages, such as [Pytorch \[18\]](#), [Tensorflow \[1\]](#) or [Keras \[5\]](#) support.

### 4.1.3 Towards a unified API (O3)

We have designed the user API of **teex** so that it is simple to use and learn. On a high-level, **teex** is fundamentally split into modules. In particular, each module contains methods related to an explanation type (more about this in section [4.2.1](#)). Each module, though, contains two sub-modules: the evaluation **eval** and the **data** sub-modules. The reasoning behind each sub-modules is the following:

- **eval**: contains evaluation methods.
- **data**: contains data classes with available g.t. explanations of that particular explanation type, both synthetic and real.

This first segmentation of **teex** allows the user to grasp a common structure: for a particular explanation type, one needs to go to a specific module and use the sub-module **eval** or **data** depending on whether one wants to get data or evaluate explanations. Note that this segmentation also allows future collaborators to have a clearer view of where implementations for specific explanation types should go in the project: everything is clearly categorized. See figure [9](#) for a conceptual view of **teex**.

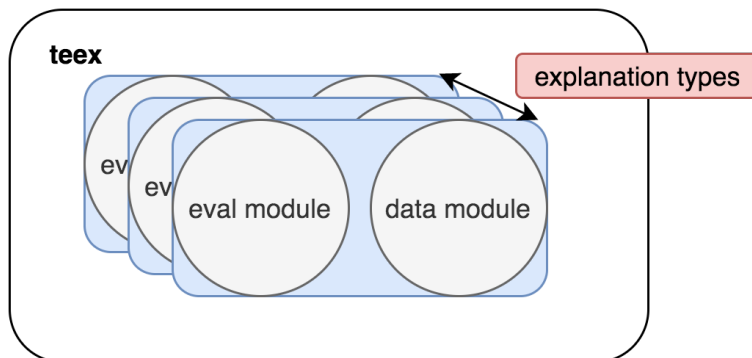


Figure 9: Highest abstract view of **teex**, subdivided into modules for each explanation type supported. Each module contains two sub-modules, one dedicated to evaluation and the other for data procurement.

Another design choice that allows for a better user experience in **teex** is the fact that in each **eval** sub-module there exists one method that is able to compute all of the evaluation metrics available for that particular explanation type. Moreover, the usage

syntax of this method is the same across the `eval` sub-modules, i.e. works the same way regardless of the explanation type that it is processing. These two facts allow for a better user experience (because all metrics can be computed with just a function call) and for software scalability, as metrics are all contained and not sparsely implemented. See figure [10](#) for a high level view of the intended architecture of the `eval` modules and [11](#) for the `data` modules.

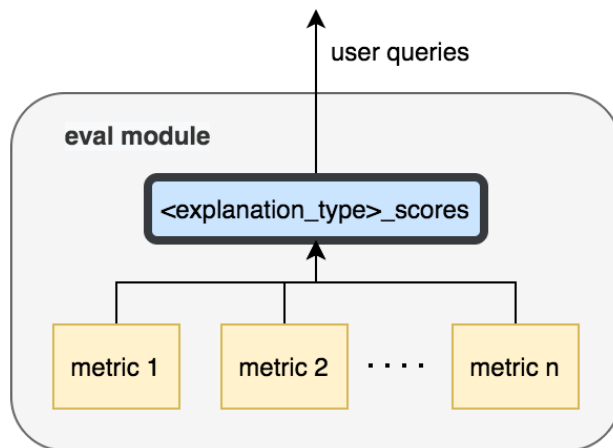


Figure 10: High level view of the intended architecture of the `eval` modules. Ideally, the user makes calls to a single method, which in turn calls individual evaluation metrics and combines the information in a nice package, which is lastly returned to the user.

Lastly, similar to the evaluation metrics, the methods in the `data` sub-modules also are implemented to work almost the same across explanation types. That is, they all are all implemented in the same way and designed to returned data in the same way. In particular, datasets are implemented as classes. For a user to retrieve data from a particular dataset, the object has to be instanced. If the dataset is to be retrieved, it will automatically download from the web if it does not exist in the machine, and when the user slices the instanced object it will return the desired data observations, labels and ground-truth explanations, respectively. This procedure is illustrated in figure [12](#).

#### 4.1.4 Explanations as a proxy for explainer evaluation (O4)

The last architectural objective is for teex to be explainer agnostic. The way we have achieved this objective is by focusing our attention on not the evaluation of explainer methods themselves, but on explanations. In particular, local explanations. As a reminder, local explanations represent the thought process behind a model’s prediction for a particular observation. The majority of local explanation methods generate explanations

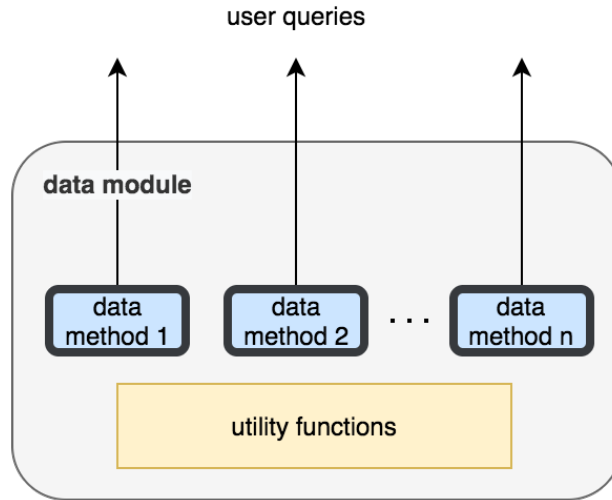


Figure 11: High level view of the intended architecture of the `data` modules. The user calls the desired data method, which in turn are supported by utility functions. We did not find appropriate for a single API call to return for any dataset, because each dataset has its own unique characteristics, and that may make things more complicated rather than easier for the user.

of similar types regardless of their architecture (more in section [4.2.1](#)), and the quality of the explanations produced by an explainer is directly correlated with the quality of the explainer method itself. So, by evaluating local explanations we have a system that is explainer agnostic and not only evaluates individual local explanations, it also can be used to evaluate explainer methods themselves via induction. As it is, **teex is based on this premise**: evaluating local explanations by comparing them to ground truth explanations.

## 4.2 Functionalities

### 4.2.1 The 4 main explanation types (O5)

So, if **teex** is to support specific local explanation types, which ones should they be? We have based our choosing criteria mainly on popularity and diversity of data from where these explanations can be derived. Looking at figure [4](#), we see that local explanations can be generated from different data types: tabular, image and language data. Based on this segmentation, we have chosen Feature Importance vectors, Saliency Maps, Decision Rules and Word Importance vectors as the explanations supported in the first version of **teex**. We are not going to go into implementation details because that would be too lengthy, but we are going to say how each explanation type is internally represented. Let's elaborate on each.

- **Feature Importance** vectors. They are vectors with one entry per feature. Each

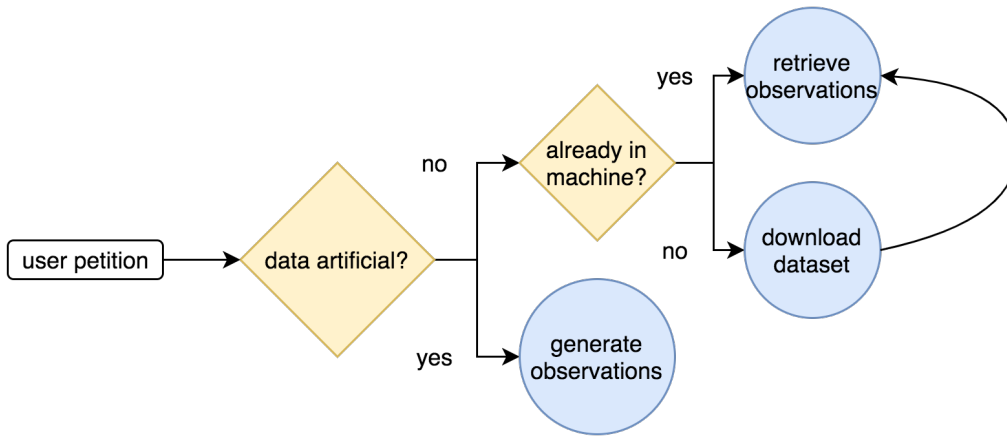


Figure 12: High level procedure of data retrieval for any data class in `teex`.

entry contains a weight that represents a feature’s importance for the observation’s outcome. Weights are usually in the range  $[-1, 1]$ . Popular feature importance model-agnostic explainers are, for example, SHAP [14] (figure 13) or LIME [19]. Because weights in each feature importance explanation method represent slightly different things, we make the assumption that they all mean roughly the same if they are in the same range (so that we want to compare methods). `teex` performs this mapping automatically if necessary.

Feature Importance vectors are treated as `float numpy` arrays by `teex`.

- **Decision Rule** explanations. A Decision Rule is a conjunction of statements that, if holds true, implies a result. For example, a decision rule could be: ”if it is white and it quacks, then it’s a duck”. There are two statements in this rule (1-”if it is white” and 2-”if it quacks”), which if they both hold true imply that what I am seeing is, indeed, a duck. In the context of tabular data, decision rules explanations contain statements regarding the values of the features for specific observations, and the result implied by the rule is usually a statement about the target’s value (i.e. target will be  $\geq 5$ ). In `teex`, Decision Rule explanations are implemented as standalone objects, and multiple routines for conversion of strings or other common decision rule representations to our proprietary decision rule objects have been provided. We have decided to implement our own Decision Rule class because it provides flexibility for the implementation of evaluation metrics and data generation routines.
- **Saliency Maps**. A saliency map is an image that shows each pixel’s unique quality. In our context, each pixel (feature) contains a score (in the ranges  $[-1, 1]$  or  $[0, 1]$  as with feature importance explanations) that represents a likelihood or probability of each pixel belonging to a particular class. See figure 15 for an example of a saliency map explanation. In `teex`, saliency map explanations are represented the

same way as feature importance explanations but in 2D, that is, as `float numpy` arrays. Saliency maps are one of the most relevant explanation types, as they are very human-ready and because of the popularity of ML image models across the industry.

- **Word Importance** explanations. These explanations are used to explain predictions in the context of sentences. For each word in a sentence / paragraph / text (observation in general), there is a weight associated that represents its importance in the context of the model issuing a particular prediction (figure 14). For example, in a language detection model, these weights would represent how important is each word for the prediction of that sentence being in a particular language. In `teex`, word importance vectors are represented as dictionaries, with each key being a word in an observation and each value being its associated weight.

Remember that, for each of these explanation types, `teex` has associated quality evaluation metrics and available datasets.

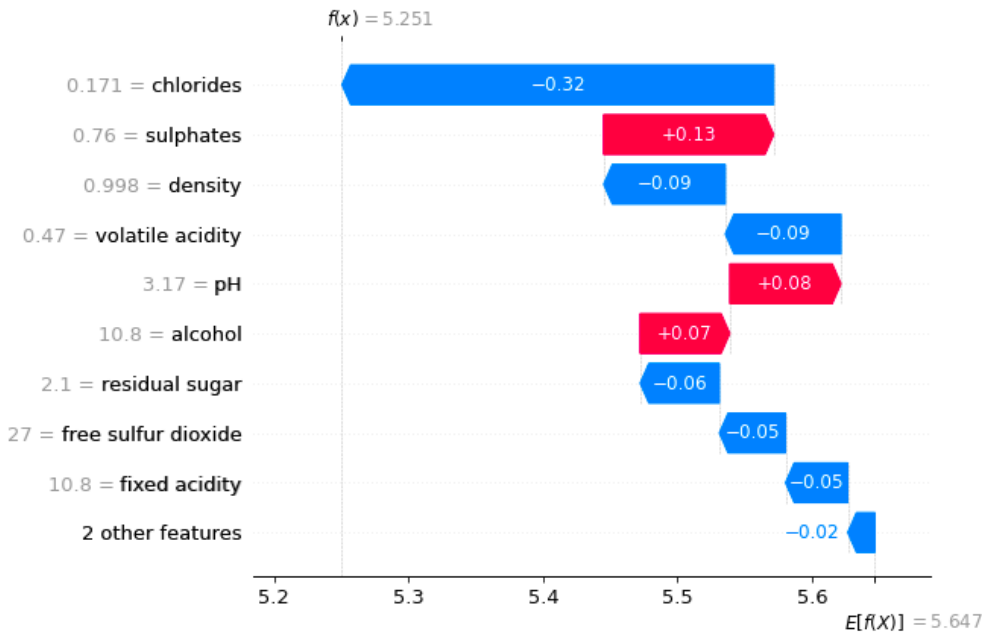


Figure 13: An example of a SHAP feature importance explanation. Each feature of an observation (names on the vertical axis) is assigned a weight (numbers on top of the colored blocks) that represents how much that feature contributed to the prediction of the model ( $f(x)$ ). Note the negative weights. Source: [Medium.com](https://medium.com).

#### 4.2.2 Evaluation metrics & how they can be shared (O6)

For each of the explanation type presented in section 4.2.1, we present the metrics that we have decided to include in the first version of `teex`. In table 1, one can find the ab-



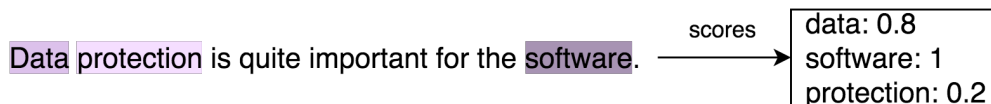


Figure 14: A word importance explanation. The text could be in the category "Computer Science", and the weights would represent how important those words are for the categorization of the text into that category. Words that do not appear in the explanation have 0 importance.



Figure 15: An image observation next to the saliency map explanation (overlaid on top of the original image). Warmer colors represent a higher score for that particular pixel, which means that the model looked more at that spot for the prediction of a dog being in the image.

breveation of each metric and the explanation type it is related to, which means that it is implemented in its respective `eval` module. Before going into further detail about the available metrics, we need to first explain one key concept `teex` takes advantage of.

**The universality of explanations in `teex`.** In `teex`, all explanation types share metrics. This is made possible by the fact that we can transform all explanation types into a common representation: a feature importance vector. This way, the metrics implemented for standard feature importance vectors are automatically made available to other explanation types, adding variety to the quality metrics that one can use to evaluate a system. But, how exactly is this translation performed for each explanation type? Lets see it.

- **Saliency Maps into Feature Importance vectors.** In saliency maps, each pixel has a weight associated to it. So, we can interpret each pixel as a feature. This way, the saliency map is just a 2D feature importance vector, which we can flatten and transform into a standard feature importance explanation.

Explanation Type	Metrics available
Feature Importance	F1, AUC, cs, prec, rec
Decision Rule	crq, F1, AUC, cs, prec, rec
Saliency Map	F1, AUC, cs, prec, rec
Word Importance	F1, AUC, cs, prec, rec

Table 1: Metrics available on the first release of **teex** for each explanation type. Notice how many metrics are shared amongst explanation types.

- Decision Rules into Feature Importance vectors.** Given the set of all features in a particular dataset, a decision rule associated with that dataset can be transformed into a feature importance vector. We create a vector with as many entries as the number of features, where each entry contains a 1 if the decision rule being translated contains a statement related to that particular feature and 0 otherwise. See figure [16](#) for a graphical example.
- Word Importance into Feature Importance vectors.** In a similar way to saliency maps, in word importance explanations, each word represents a feature. Moreover, similar to decision rules, imagine that there exists a vocabulary of words (may be all of the words in the ground truth explanation). A word importance vector will be translated into a feature importance vectors by putting 1s in the vector entries, which represent the fact the the word importance explanation contained that particular word, and 0s otherwise.

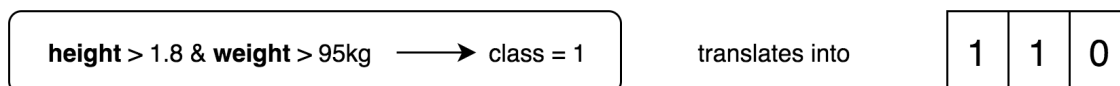


Figure 16: Translation of a decision rule to a feature importance explanation. In this case, the features in the dataset are "height", "weight" and "age". This particular explanation does not contain "age" in any statement, so its feature importance representation contains a 0 in its third entry.

Now that we know about the 'universal' explanation representation, we can go into further detail about the implemented metrics.

- AUC (ROC AUC).** A classification metric. Given prediction scores that represent the likelihood or probability score of a feature pertaining to a given class, we compute the ROC AUC with respect to the ground truth explanation values, which are binarized to represent classes. ROC AUC is a measure of the performance of a classification system. A value of 1 indicates a perfect classifier, while 0.5 corresponds to

a random classifier. In order to compute this metric, one computes or approximates the area under the Receiving Operating Curve. We do not describe the procedure as it is out of scope.

- **cs (Cosine Similarity)**. A metric that measures how similar two vectors that pertain to the same subspace are to each other in terms of orientation. A value of -1 means that they are perpendicular to each other, while a value of 1 means that the vectors are parallel. Note that this measure only takes into account orientation and not direction.

$$\text{similarity}(A, B) = \cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\| \|\mathbf{B}\|}$$

- **prec (Precision)**. In the classification setting, the precision is the percentage of correct positive predictions with respect to the total number of positive predictions made by a classifier. In a binary vector, this translates to the percentage of correct 1's predicted by the classifier with respect to the total number of 1's predicted.

$$\text{precision} = \frac{\text{True Positive}}{\text{True Positive} + \text{False Positive}}$$

- **rec (Recall)**. In the classification setting, the recall is a measure of how many correct positive predictions the classifier has issued with respect to the total number of true positive predictions. In a binary vector, this translates to the percentage of correct 1's predicted by the classifier with respect to the number of 1's in the ground truth vector.

$$\text{recall} = \frac{\text{True Positive}}{\text{True Positive} + \text{False Negative}}$$

- **F1 (F1-score)**. A harmonic average of precision and recall. This is a classification metric, so both the predicted and the ground truth explanations need to be binary.

$$\text{F1} = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$$

- **crq (Complete Rule Quality)** [10]. A metric exclusive to decision rules. It is defined as the proportion of lower and upper bounds in a rule explanation that are very close to the respective lower and upper bounds (same feature) in the ground truth rule explanation amongst those that are  $\neq \infty$ . This metric does not take the absence of a feature in the predicted decision rule into account: only the features that do appear are important. Mathematically, given two rules  $e, \tilde{e}$  and a similarity threshold  $\varepsilon$ , the quality of  $e$  with respect to  $\tilde{e}$  is:

$$q(e, \tilde{e}) = \frac{1}{N_\infty} \sum_{i=1}^{|e|} \delta_\varepsilon(e_i, \tilde{e}_i),$$

where

$$\delta_\varepsilon(e_i, \tilde{e}_i) = \begin{cases} 1 & \text{if } |e_i - \tilde{e}_i| \leq \varepsilon \wedge |e_i| \neq \infty \wedge |\tilde{e}_i| \neq \infty, \\ 0 & \text{otherwise} \end{cases}$$

Where  $N_\infty$  is the number of lower and upper bounds that are different from  $\infty$  in both  $e$  and  $\tilde{e}$ .

Now that an overview of the implemented metrics has been given, we'd like to take a moment to talk about binarization. Many of the metrics presented are classification metrics. So, when they are computed, **teex** automatically does all of the work. One thing that it cannot do, though, is decide a binarization threshold (values which  $>$  than it will be set to 1 and 0 otherwise). This is an important hyperparameter for the computation of metrics, then.

### 4.2.3 A whole ecosystem: datasets in teex (O7)

**The problem of data availability.** As we have seen, we need ground truth explanations in order to evaluate the ones created by explainer methods. Again, **this premise is what teex is ultimately based on**. The problem is that, with almost all data sets available today, ground truth explanations are very hard to get. For example, if we want Saliency Map ground truth explanations (where individual pixels have class importance, further explained in section [4.2.1](#)), an expert would have to manually label the areas where he/she thinks is of importance to the relevant class, for all of the images. Moreover, even when these datasets are created, there is no standard repository that categorizes whether a dataset has available ground truth explanations or not.

This is why we have decided for **teex** to include datasets, particularly all of them with available ground truth explanations. This will allow users to directly jump into using the evaluation metrics. All of the datasets are contained in their respective **data** sub-modules (depending on the type of the ground truth explanation). Now we explain the datasets available in this first version of **teex**.

**Feature Importance datasets.** An artificial binary classification dataset users can generate samples from. The procedure for generating the samples is the following: first, create observations from normal distributions and then add noise to those observations. Then, generate a random linear expression and classify the previous observations depending on whether they fall on one side or another of the hyperplane generated by the linear

expression. Finally, the ground truth explanation is generated as the gradient of linear expression evaluated at the point closes to the decision boundary (and of the same class as the observation we are computing the explanation for). Described in [10].

**Decision Rule datasets.** As with feature importance, the decision rule dataset included in this version of **teex** is artificially generated. The procedure of generation comprises three steps. First, generate data from a normal distribution. Then, train a decision tree on that data. Finally, extract the ground truth explanations as the decision paths learned by the decision tree. Again, this was described in [10].

**Saliency Map datasets.** **teex** provides two datasets with available ground truth explanations. The first one, presented in [11], contains aerial images of forests in New Zealand. The task is to tell whether the pictures contain a certain tree species (the Kahikatea) or not. The ground truth explanations have all been labeled by experts (figure 17). The second dataset is artificially generated. It again is a binary classification dataset, in which only an adjustable proportion of samples contain a randomly generated pattern. A sample is of class 1 if it contains the pattern and 0 otherwise. The ground truth explanations are images with the areas where the pattern appears highlighted (figure 18). This method was also presented in [10].

**Word Importance datasets.** **teex** provides one word importance dataset. It is a version of the famous "Newsgroup" dataset, which contains emails classified as pertaining to different themes. This version of the dataset has 187 observations, and each observation pertains to one of two classes. The ground truth explanations are presented to the users as dictionaries with words as keys and their importance for the prediction as values. This dataset has been extracted from [github.com/SinaMohseni/ML-Interpretability-Evaluation-Benchmark](https://github.com/SinaMohseni/ML-Interpretability-Evaluation-Benchmark).

Note that all artificial data generators can be easily customised by the end user, and they are seedable for reproducibility. All of these data set classes have methods so that the user can retrieve basic information about the dataset such as the classes it contains or the number of observations in it.

### 4.3 Quality of life measures

Quality of life objectives are meant to extend the lifespan of **teex**.

#### 4.3.1 Unit testing & error handling (O8)

We provide developers with a suite of unit tests. These can be run with a single script call and will test all of the described evaluation metrics and dataset classes, for all explanation

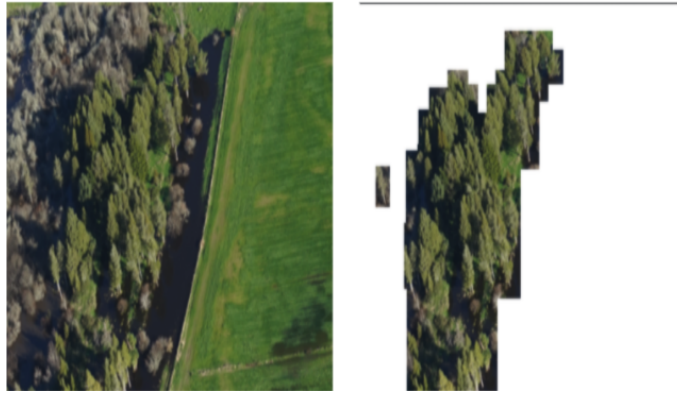


Figure 17: Sample of the Kahikatea dataset, containing a total of 519 labeled observations. To the left is the ground truth explanation, with the endemic Kahikatea trees highlighted.

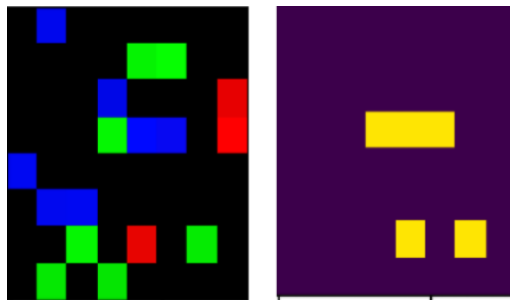


Figure 18: Sample of artificially generated saliency map data. The explanation on the right highlights where the hidden pattern is in the image. That pattern will also appear in a proportion of the samples.

types. The suite used to implement them is Python’s builtin `unittest`. Basically, each test is meant to test a functionality of the software (i.e. a dataset class), and is subdivided into atomic tests which are not dependant on each other (one tests the downloading part of the dataset, the other the unzipping, another the indexing and so on). This way, all functionalities are tested and the results do not depend on whether other things work or no. These tests are not presented as part of the library to the end user, as they are part of the developer’s toolkit.

Moreover, because we wanted debugging to be easy when developing for `teex`, we have implemented custom errors that are raised when anything goes wrong. Custom error classes allow us to have a more descriptive explanation of what caused the error and how it can be fixed.

### 4.3.2 Complete API documentation with Sphinx (O9)

Because we want `teex` to be built upon, there exists the need for extensive code documentation. We have decided to use [Sphinx](#), a documentation generator that is based on the

ReStructuredText (RST) grammar. If one documents the methods following RST grammar (in the code itself), then Sphinx can be applied almost off-the-shelf to automatically generate formatted code documentation.

Then, we have documented all methods following RST grammar. This markdown-like grammar allows to specify a method's description, input parameters (and their types and descriptions), as well as inline examples and returned objects and their type, all while supporting the insertion of inline code,  $\LaTeX$ math grammar and hyperlinks between methods and data classes. See figure 19 for an example.

```
word_to_feature_importance(wordImportances, vocabWords) -> list:
""" Maps words with importance weights into a feature importance vector.

:param wordImportances: (dict or array-like of dicts) words with feature importances as values with the same format
    as described in the method :func:`word_importance_scores`.
:param vocabWords: (array-like of str, 1D or 2D for multiple reference vocabularies) :math:`m` words that
    should be taken into account when transforming into vector representations. Their order will be followed.
:return: Word importances as feature importance vectors. Return types:

    - list of np.ndarray, if multiple vocabularies because of the possible difference in size of the reference
      vocabularies in each explanation.
    - np.ndarray, if only 1 vocabulary

:Example:

>>> word_to_feature_importance({'a': 1, 'b': .5}, ['a', 'b', 'c'])
>>> [1, .5, 0]
>>> word_to_feature_importance([{'a': 1, 'b': .5}, {'b': .5, 'c': .9}], ['a', 'b', 'c'])
>>> [[1, .5, 0. ], [0, .5, .9]]
"""
```

[teex.wordImportance.eval.word\\_to\\_feature\\_importance\(wordImportances, vocabWords\)→ list](#) [\[source\]](#)

Maps words with importance weights into a feature importance vector.

**Parameters:**

- wordImportances** – (dict or array-like of dicts) words with feature importances as values with the same format as described in the method `word_importance_scores()`.
- vocabWords** – (array-like of str, 1D or 2D for multiple reference vocabularies)  $m$  words that should be taken into account when transforming into vector representations. Their order will be followed.

**Returns:** Word importances as feature importance vectors. Return types:

- list of np.ndarray, if multiple vocabularies because of the possible difference in size of the reference vocabularies in each explanation.
- np.ndarray, if only 1 vocabulary

**Example:**

```
>>> word_to_feature_importance({'a': 1, 'b': .5}, ['a', 'b', 'c'])
>>> [1, .5, 0]
>>> word_to_feature_importance([{'a': 1, 'b': .5}, {'b': .5, 'c': .9}], ['a', 'b', 'c'])
>>> [[1, .5, 0. ], [0, .5, .9]]
```

Figure 19: On top, specification of a method's documentation string on the source files. On the bottom, the corresponding Sphinx representation.

The generated final documentation is not private, though. It is hosted on Read The Docs ([teex.readthedocs.io](https://teex.readthedocs.io)) for everyone to access. Moreover, for completeness, it can be found in the Appendix of this document.

Note that the project contains upwards of 3000 lines of code, many of which do not belong to public methods (they are utility functions or custom error classes, for example), which means that they do not appear on the API documentation published on Read The Docs. We encourage curious users to look at the code base (find it in section [4.3.4](#)).

### 4.3.3 Basic example notebooks (O10)

Because the API documentation is not a good place for first time users to start at when using **teex**, we have provided Python interactive notebooks with basic usage examples of the library. These comprise evaluation and generation of explanations of the different supported types. They can be found alongside with the code. See section [4.3.4](#) for more information on where to find it.

### 4.3.4 Making **teex** public (O11)

The last QOL objective was for **teex** to be open sourced. This would allow the community to more actively participate in the development of **teex**, as well as pointing out issues and adding functionalities. We have decided to host it on GitHub ([github.com/chus-chus/teex](https://github.com/chus-chus/teex)) for three reasons:

- **Popularity.** GitHub is arguably the best-known open source platform. More users imply more interaction.
- **Familiarity.** We simply know how to use GitHub to its fullest extent. We are not so familiar with other platforms.
- **Web hooks.** Web hooks are actions that are triggered in a particular website when an action on GitHub is triggered. In our case, the API documentation hosted on Read The Docs is automatically rebuilt when new code is pushed to **teex**'s GitHub repository, which saves us a lot of time.

The entirety of the source code is hosted on GitHub, including configuration files, utility functions, unit tests, requirements and example notebooks. This way, cloning (downloading) the repository will suffice for other developers to build and test their own functionalities, which can then be added on top of **teex** if desired.



## 5 Usage examples

In this section we provide basic usage examples of **teex**. Similar and more examples can be found the Github repository.

### 5.1 Data classes in teex

As we have previously said, all of the data, be it artificial or from the real world, is in its core a Python class. These classes have all been implemented to behave in the same way for the end user. An example of retrieving data with **teex** can be found in figure [20](#).

```
from teex.saliencyMap.data import Kahikatea

kData = Kahikatea() # instantiation
X, y, explanations = kData[30:130] # retrieval
```

Figure 20: Retrieving observations from the Kahikatea dataset. The user first imports the method from the appropriate sub-module. Then, an instance of the data is created. In this case, because it is not an artificial dataset, **teex** will take care of checking if the files exist (and are not corrupted) in the system already, downloading it and pre-processing so that they can be readily loaded into memory. Finally, the user slices the instanced object to obtain the data. The obtained observations and explanations in this example have been shown in figure [17](#).

All data classes in **teex** return, respectively, the data observations, the target values and the ground truth explanations when sliced. They all work the same way! The slicing syntax has been implemented so that it works as with any other sliceable object in Python. Another feature of data classes in **teex** is that when instanced, not all observations are loaded into memory at once. Only the ones requested by the user (in this case 100 observations from the 30th to the 130th) will be loaded into memory, which can save lots of memory depending on the magnitude of the dataset.

### 5.2 Evaluation in teex

Let us emulate an evaluation setting. Imagine that we have the data obtained from the Kahikatea dataset as in figure [20](#). The explanations, then, are saliency maps. Let us assume that we also have explanations created by a ML model trained on the data, and these happen to perfectly match the ground truth explanations. we can evaluate the metrics like in figure [21](#). The syntax is followed by all of **teex**'s **eval** sub modules:

- Feature Importance: `feature_importance_scores`

- Saliency Maps: `saliency_map_scores`
- Decision Rules: `decision_rule_scores`
- Word Importance: `word_importance_scores`

```
from teex.saliencyMap.eval import saliency_map_scores

saliency_map_scores(explanations, explanations,
                    metrics=['fscore', 'cs', 'auc'])
```

Figure 21: Computing evaluation metrics for the emulated scenario where predicted explanations equal ground truth explanations (follows the example on figure [20](#)).

Note that each high level evaluation method has many customization options. For example, if the user wants to average the metrics across all explanations or not, or what the binarization threshold (mentioned in section [4.2.2](#)) should be. For more details, please refer to the API documentation.

### 5.3 A more extensive evaluation procedure with teex

In this section, we go further from bare basic usage and provide a more complete example of the functionalities of `teex`. The source for this experiment can be found in the GitHub repository. The procedure for the experiment has been the following:

1. **Data generation.** First, we generate synthetic 5000 image samples with available saliency maps g.t. explanations. We generate them so that half of them contain a pattern and half of them not. We split the data into train, validation and test.
2. **Training the model.** We program and train a Deep Learning model so that it detects the hidden pattern in the images (with 0.99 F1 in validation and 0.98 F1 in test).
3. **Generating test explanations.** We generate local explanations from the test data via different explainer methods. In particular we use the following methods from `Captum`, a library that compiles interpretability techniques for Pytorch models: Gradient SHAP [\[14\]](#), Integrated Gradients [\[24\]](#), Occlusion [\[30\]](#) and DeepLift [\[23\]](#). See a sample for the generated explanations in figure [22](#). The explanations are computed only for the test set because the model has never seen it and will make the model behave the way it would on new, unseen data (i.e. its true performance, without overfitting).

technique	auc	fscore	prec	rec	cs
gradSHAP	0.489231	0.117196	0.062264	0.995251	<b>0.246058</b>
intGrad	0.481442	0.117204	0.062268	0.995319	0.245174
deepLift	0.454612	<b>0.117253</b>	<b>0.062294</b>	<b>0.995732</b>	0.242507
occlusion	<b>0.489906</b>	0.117245	0.062290	0.995664	0.238931

Table 2: Averaged evaluation metrics for the saliency map explanations generated in the experiment of section 5.3. We see how DeepLift seems to outperform the other techniques in this particular case. The scores, though, are very similar, probably due to the limited capabilities of the classifier itself.

4. **Evaluation of explanations.** Finally, we evaluate the quality of the explanations generated with respect to the available ground truth explanations. See table 2 for the results.

For more experiments and examples, we urge the reader to visit **teex**'s GitHub repository.

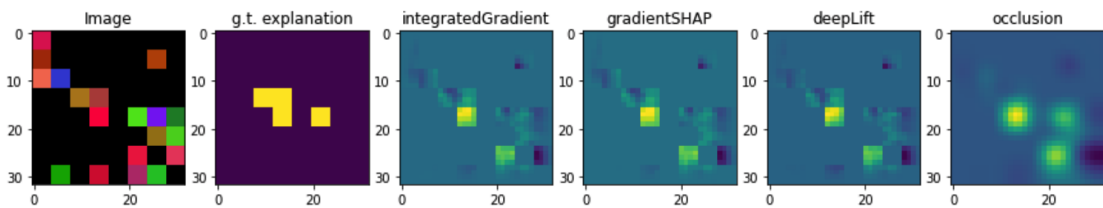


Figure 22: Generated explanations via different methods next to the observation they were based on and its associated ground truth explanation.

## 6 Conclusion

In this work, we have presented **teex**, a Python toolbox for the evaluation of local explanations. The evaluation procedure is performed via comparison to ground-truth explanations. We have described the quality metrics that **teex** is able to compute, as well as the types of explanation types it supports as of the first release: feature importance vectors, saliency maps, decision rules and word importance vectors. We have also described how **teex** has the potential to act like a 'hub' of datasets with available ground truth explanations. These kind of datasets are hard to find, and usually are not readily usable. With **teex**, users can start using these datasets with as little as 2 lines of code. The evaluation procedure is also extremely simple, as **teex** is responsible for all of the caveats (data format, data transformation, error handling or metrics recollection, amongst others) that an end user should not have to worry about. We have also presented background information, explained the high level architecture of **teex**, its functional components and the most important quality of life measures that we have put into place.

**teex** is the first piece of software that we know of with the described purposes. It is the first of its kind, and we really believe has the potential to expand into something bigger and be useful to XAI researchers because of (1) the novelty of the work and (2) its growth potential given the increasing importance the XAI field is gaining. Because of these reasons, the work has been made open-source, so that the community can participate in the development of **teex** if wanted. We have also created plenty of resources for first-time users to learn how **teex** works: an initial guide with installation instructions and an overview of the API, several Python notebooks with sample experiments and usage of the library, and a full API documentation guide. The software is available on [github.com/chus-chus/teex](https://github.com/chus-chus/teex) and the API documentation on [teex.readthedocs.io](https://teex.readthedocs.io).

We are very satisfied with the work and will continue working on features such as implementing more evaluation metrics, compiling new data sets, giving support to other explanation types. Moreover, at the time of writing this, we are preparing an article presenting the software for the Journal of Machine Learning Research, in particular the [Open Source Software Track](#). This work has been a process of, above all, learning. First, diving head-first into the field of XAI, which we were not familiar with in any way. Then, software development with Python, which is something that we would not have imagined doing, but has ended up being fun and challenging and will certainly be of use in the future. We are excited to continue working on this project so that it receives attention from the community.

## References

- [1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mane, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viegas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: Large-scale machine learning on heterogeneous distributed systems, 2016.
- [2] Rishabh Agarwal, Nicholas Frosst, Xuezhou Zhang, R. Caruana, and Geoffrey E. Hinton. Neural additive models: Interpretable machine learning with neural nets. *ArXiv*, abs/2004.13912, 2020.
- [3] David Alvarez Melis and Tommi Jaakkola. Towards Robust Interpretability with Self-Explaining Neural Networks. In S Bengio, H Wallach, H Larochelle, K Grauman, N Cesa-Bianchi, and R Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018.
- [4] Francesco Bodria, Fosca Giannotti, Riccardo Guidotti, Francesca Naretto, Dino Pedreschi, and Salvatore Rinzivillo. Benchmarking and survey of explanation methods for black box models, 2021.
- [5] François Chollet et al. Keras. <https://github.com/fchollet/keras>, 2015.
- [6] Finale Doshi-Velez and Been Kim. Towards a rigorous science of interpretable machine learning. *arXiv: Machine Learning*, 2017.
- [7] Jerome H Friedman and Bogdan E Popescu. Predictive learning via rule ensembles. *The Annals of Applied Statistics*, 2(3):916–954, 2008.
- [8] Kary Främling. Explainable ai without interpretable model. 09 2020.
- [9] Emilio Gómez-González, Emilia Gómez, Javier Márquez-Rivas, Manuel Guerrero-Claro, Isabel Fernández-Lizaranzu, María Isabel Relimpio-López, Manuel E. Dorado, María José Mayorga-Buiza, Guillermo Izquierdo-Ayuso, and Luis Capitán-Morales. Artificial intelligence in medicine and healthcare: a review and classification of current and near-future applications and their ethical and social impact. *CoRR*, abs/2001.09778, 2020.
- [10] Riccardo Guidotti. Evaluating local explanation methods on ground truth. *Artificial Intelligence*, 291:103428, 02 2021.

- [11] Yunzhe Jia, Eibe Frank, Bernhard Pfahringer, Albert Bifet, and Nick Lim. Studying and Exploiting the Relationship Between Model Accuracy and Explanation Quality. *Not Published*.
- [12] Janis Klaise, Arnaud Van Looveren, Giovanni Vacanti, and Alexandru Coca. Alibi explain: Algorithms for explaining machine learning models. *Journal of Machine Learning Research*, 22(181):1–7, 2021.
- [13] Narine Kokhlikyan, Vivek Miglani, Miguel Martin, Edward Wang, Bilal Alsallakh, Jonathan Reynolds, Alexander Melnikov, Natalia Kliushkina, Carlos Araya, Siqi Yan, and Orion Reblitz-Richardson. Captum: A unified and generic model interpretability library for pytorch. 09 2020.
- [14] Scott M Lundberg and Su-In Lee. A unified approach to interpreting model predictions. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 4765–4774. Curran Associates, Inc., 2017.
- [15] Sina Mohseni, Niloofar Zarei, and Eric D Ragan. A multidisciplinary survey and framework for design and evaluation of explainable ai systems. *arXiv preprint arXiv:1811.11839*, 2018.
- [16] Ramaravind K. Mothilal, Amit Sharma, and Chenhao Tan. Explaining machine learning classifiers through diverse counterfactual explanations. In *Proceedings of the 2020 Conference on Fairness, Accountability, and Transparency, FAT\* '20*, page 607–617, New York, NY, USA, 2020. Association for Computing Machinery.
- [17] Harsha Nori, Samuel Jenkins, Paul Koch, and Rich Caruana. Interpretml: A unified framework for machine learning interpretability. *arXiv preprint arXiv:1909.09223*, 2019.
- [18] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library, 2019.
- [19] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. "why should i trust you?": Explaining the predictions of any classifier, 2016.
- [20] Ramprasaath R. Selvaraju, Michael Cogswell, Abhishek Das, Ramakrishna Vedantam, Devi Parikh, and Dhruv Batra. Grad-cam: Visual explanations from deep networks via gradient-based localization. *International Journal of Computer Vision*, 128(2):336–359, Oct 2019.

- [21] Andrew W Senior, Richard Evans, John Jumper, James Kirkpatrick, Laurent Sifre, Tim Green, Chongli Qin, Augustin Žídek, Alexander W R Nelson, Alex Bridgland, Hugo Penedones, Stig Petersen, Karen Simonyan, Steve Crossan, Pushmeet Kohli, David T Jones, David Silver, Koray Kavukcuoglu, and Demis Hassabis. Improved protein structure prediction using potentials from deep learning. *Nature*, 577(7792):706–710, 2020.
- [22] Mattia Setzu, Riccardo Guidotti, Anna Monreale, and Franco Turini. *Global Explanations with Local Scoring*, pages 159–171. 03 2020.
- [23] Avanti Shrikumar, Peyton Greenside, and Anshul Kundaje. Learning important features through propagating activation differences, 2019.
- [24] Mukund Sundararajan, Ankur Taly, and Qiqi Yan. Axiomatic attribution for deep networks, 2017.
- [25] Hui Fen Tan, G. Hooker, and M. T. Wells. Tree space prototypes: Another look at making tree ensembles interpretable. *Proceedings of the 2020 ACM-IMS on Foundations of Data Science Conference*, 2020.
- [26] Giorgio Visani, Enrico Bagli, Federico Chesani, Alessandro Poluzzi, and Davide Capuzzo. Statistical stability indices for lime: Obtaining reliable explanations for machine learning models. *Journal of the Operational Research Society*, pages 1–11, 02 2021.
- [27] Yulong Wang. Pytorch-visual-attribution. <https://github.com/yulongwang12/visual-attribution>, 2018.
- [28] Ziqi Yang. Fidelity: A property of deep neural networks to measure the trustworthiness of prediction results. In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*, Asia CCS ’19, page 676–678, New York, NY, USA, 2019. Association for Computing Machinery.
- [29] Chih-Kuan Yeh, Cheng-Yu Hsieh, Arun Sai Suggala, David I. Inouye, and Pradeep Ravikumar. On the (in)fidelity and sensitivity of explanations. In *NeurIPS*, 2019.
- [30] Matthew D Zeiler and Rob Fergus. Visualizing and understanding convolutional networks, 2013.

## 7 Appendix



---

**teex**

*Release 1.0.0*

**Jesus Antonanzas**

**Aug 04, 2021**



**CONTENTS:**

**1 teex** **3**  
    1.1 teex package ..... 3

**2 Indices and tables** **19**

**Python Module Index** **21**

**Index** **23**



A Python Toolbox for the Evaluation of machine learning Explanations.

This project aims to provide a simple way of evaluating all kinds of individual black box explanations. Moreover, it contains a collection of easy-to-access datasets with available ground truth explanations.

**teex** contains a subpackage for each explanation type, and each subpackage contains two modules:

- **eval**: with methods for explanation evaluation.
- **data**: with methods for data generation, loading and manipulation.

Visit our [GitHub](#) for source, tutorials and more.



## 1.1 teex package

### 1.1.1 Subpackages

**teex.decisionRule package**

**Submodules**

**teex.decisionRule.data module**

Module for synthetic and real datasets with available ground truth decision rule explanations. Also contains methods and classes for decisionRule data manipulation.

All of the datasets must be instantiated first. Then, when sliced, they all return the observations, labels and ground truth explanations, respectively.

**class** `teex.decisionRule.data.DecisionRule`(*statements=None, result=None*)

Bases: `object`

A conjunction of statements as conditions that imply a result. Internally, the rule is represented as a dictionary of `Statement` with the feature names as unique identifiers. A feature cannot have more than one `Statement` (Statements can be binary). This class is capable of adapting previous `Statement` objects depending on new Statements that are added to it with the `upsert_statement()` method (see `upsert_statement()` method).

#### Example

```
>>> c1 = Statement('a',lowB=2,upperB=3)      # 2 < a < 3
>>> r = DecisionRule([c1])
>>> # update the bounds for the feature 'a'
>>> c2 = Statement('a',lowB=3,upperB=5)
>>> r.upsert_statement(c2,updateOperators=False)
>>> # we can also insert new statements via upsert or insert
>>> c3 = Statement('b',lowOp='<=',lowB=3,upperOp='<',upperB=6)
>>> r.upsert_statement(c3)
>>> # a Statement cannot be updated if one of them is different class as the other
↳(binary / unary):
>>> c4 = Statement('b', 3, op='>')
>>> r.upsert_statement(c4) # THIS WILL RAISE AN ERROR!
```

#### Parameters

- **statements** – (list-like of Statement objects) Statements as conditions that make the result be True.
- **result** (*Statement*) – Logical implication of the Decision Rule when all of the Statements are True.

**delete\_statement** (*feature*) → None

Deletes a Statement in the rule.

**Parameters** **feature** (*str*) – name of the feature in the Statement to be deleted.

**get\_features**() → list

Gets features in the Rule.

**Return list** feature names as identifiers of the Statements in the rule.

**insert\_statement** (*statement: teex.decisionRule.data.Statement*) → None

Add Statement inplace to the conjunction.

**Parameters** **statement** – Statement object

**rename\_statement** (*oldFeature, newFeature*) → None

Changes the identifier of a Statement.

**Parameters**

- **oldFeature** (*str*) – id of the Statement to rename.
- **newFeature** (*str*) – new id of the Statement.

**replace\_statement** (*oldFeature, newStatement: teex.decisionRule.data.Statement*) → None

Replaces a Statement with another.

**Parameters**

- **oldFeature** (*str*) – identifier of the Statement to replace.
- **newStatement** (*Statement*) – new statement.

**set\_result** (*result*) → None

Sets the result for the Decision Rule.

**Parameters** **result** (*Statement*) – statement as logical implication.

**upsert\_statement** (*statement: teex.decisionRule.data.Statement, updateOperators: bool = True*) → None

If a statement already exists within the rule, updates its bounds (replacing or defining them) and its operators if specified. If not, inserts the statement as a new condition. If an existing condition is of different type (binary / non-binary) as the new condition, the update fails. A bound update is only performed if the new bound/s != np.inf or -np.inf.

**Parameters**

- **statement** – Statement object to upsert
- **updateOperators** – Should the operators be updated too?

**class** teex.decisionRule.data.**SenecaDR** (*nSamples: int = 1000, nFeatures: int = 3, featureNames=None, randomState: int = 888*)

Bases: teex.\_baseClasses.\_baseDatasets.\_SyntheticDataset

Generate synthetic binary classification data with ground truth decision rule explanations. The returned decision rule g.t. explanations are instances of the *DecisionRule* class.



Ground truth explanations are generated with the *TransparentRuleClassifier* class. The method was presented in [Evaluating local explanation methods on ground truth, Riccardo Guidotti, 2021]. From this class one can also obtain a trained transparent model (instance of *TransparentRuleClassifier*).

#### When sliced, this object will return

- X (ndarray) of shape (nSamples, nFeatures) or (nFeatures). Generated data.
- y (ndarray) of shape (nSamples,) or int. Binary data labels.
- explanations (list) of *DecisionRule* objects of length (nSamples) or *DecisionRule* object. Generated ground truth explanations.

#### Parameters

- **nSamples** (*int*) – number of samples to be generated.
- **nFeatures** (*int*) – total number of features in the generated data.
- **featureNames** – (array-like) names of the generated features. If not provided, a list with the generated feature names will be returned by the function (necessary because the g.t. decision rules use them).
- **randomState** (*int*) – random state seed.

```
class teex.decisionRule.data.Statement(feature, val=inf, op='=', lowOp=None, lowB=- inf,  
                                       upperOp=None, upperB=inf)
```

Bases: object

Class representing the atomic structure of a rule. A Statement follows the structure of ‘feature’ <operator> ‘value’. It can also be binary, like so: value1 <lowOp> feature <upperOp> value2. Valid operators are {‘=’, ‘!=’, ‘>’, ‘<’, ‘>=’, ‘<=’} or {‘<’, ‘<=’} in the case of a binary statement. The class will store upper and lower bound values if the lower and upper operators are specified (both, just 1 is not valid). If the upper and lower operators are not specified, a unary Statement will be created.

Although unary Statements (except ‘!=’) have translation into single binary Statements, they are separately represented for clarity. Moreover, unary Statements with operators ‘=’ and ‘!=’ are able to represent non-numeric values.

#### Example

```
>>> Statement('a', 1.5)                # a = 1.5
>>> Statement('a', 1.5, op='!=')       # a != 1.5
>>> Statement('a', lowOp='<', lowB=2, upperOp='<', upperB=5) # 2 < a < 5
>>> Statement('a', lowOp='<', lowB=2)  # 2 < a Wrong. Need to
↳ explicitly specify upper op
>>> Statement('a', lowOp='<', lowB=2, upperOp='<')          # 2 < a < np.inf
```

#### Parameters

- **feature** (*str*) – name of the feature for the Statement
- **val** – (float or str) Value for the statement (if not binary). Default `np.inf`.
- **op** (*str*) – Operator for the statement (if not binary)
- **lowOp** (*str*) – Operator for the lower bound (if binary)
- **lowB** (*float*) – Value of the upper bound (if binary). Default `-np.inf`.
- **upperOp** (*str*) – Operator for the upper bound (if binary)
- **upperB** (*float*) – Value of the lower bound (if binary). Default `np.inf`.

**class** teex.decisionRule.data.TransparentRuleClassifier(\*\*kwargs)

Bases: teex.\_baseClasses.\_baseClassifier.\_BaseClassifier

Used on the higher level data generation class *teex.featureImportance.data.SenecaFI* (use that and get it from there preferably).

Transparent, rule-based classifier with decision rules as explanations. For each prediction, the associated ground truth explanation is available with the *explain()* method. Follows the sklearn API. Presented in [Evaluating local explanation methods on ground truth, Riccardo Guidotti, 2021].

**explain**(obs)

Explain observations' predictions with decision rules.

**Parameters** obs – array of n observations with m features and shape (n, m)

**Returns** list with n *DecisionRule* objects

**fit**(data, target, featureNames=None)

Fits the classifier and automatically parses the learned tree structure into statements.

**Parameters**

- **data** – (array-like) of shape (n\_samples, n\_features) The training input samples. Internally, it will be converted to dtype=np.float32.
- **target** – (array-like of shape (n\_samples,) or (n\_samples, n\_outputs)) The target values (class labels) as integers or strings.
- **featureNames** – (array-like) names of the features in the data. If not specified, they will be created. Stored in self.featureNames.

**predict**(obs)

Predicts the class for each observation.

**Parameters** obs – (array-like) of n observations with m features and shape (n, m)

**Return** np.ndarray array of n predicted labels

**predict\_proba**(obs)

Predicts probability that each observation belongs to each of the c classes.

**Parameters** obs – array of n observations with m features and shape (n, m)

**Return** np.ndarray array of n probability tuples of length c

teex.decisionRule.data.rule\_to\_feature\_importance(rules, allFeatures) → numpy.ndarray

Converts one or more *DecisionRule* objects to feature importance vector/s. For each feature in *allFeatures*, the feature importance representation contains a 1 if there is a :class:'Statement' with that particular feature in the decision rule and 0 otherwise.

**Parameters**

- **rules** – (*DecisionRule* or (1, r) array-like of *DecisionRule*) Rule/s to convert to feature importance vectors.
- **allFeatures** – (array-like of str) List with m features (same as the rule features) whose order the returned array will follow. The features must match the ones used in the decision rules.

**Returns** (binary ndarray of shape (n\_features,) or shape (n\_rules, n\_features)).

teex.decisionRule.data.rulefit\_to\_decision\_rule(rules, minImportance: float = 0.0, minSupport: float = 0.0) → list

Transforms rules computed with the RuleFit algorithm (only from this implementation) into *DecisionRule* objects.

### Example

```

>>> import pandas as pd
>>> from rulefit import RuleFit
>>> from teex.decisionRule.eval import rule_scores
>>>
>>> boston_data = pd.read_csv('https://raw.githubusercontent.com/selva86/datasets/
↳master/BostonHousing.csv')
>>> y = boston_data.medv.values
>>> features = boston_data.columns
>>> X = boston_data.drop("medv", axis=1).values
>>>
>>> rf = RuleFit()
>>> rf.fit(X, y, feature_names=features)
>>> rf.predict(X)
>>>
>>> dRules, _ = rulefit_to_decision_rule(rf.get_rules())
>>> rule_scores(dRules, dRules, allFeatures=features, metrics=['crq', 'fscore'])

```

### Parameters

- **rules** (*pd.DataFrame*) – rules computed with the `.get_rules()` method of `RuleFit`. Default 0.
- **minImportance** (*float*) – minimum importance for a rule to have to be transformed. Default 0.
- **minSupport** (*float*) – minimum support for a rule to have to be transformed.

### Returns

- (list) parsed `DecisionRules`
- (list) indexes of skipped rows (because of exceptions such as ‘home < 1 & home > 3’).

`teex.decisionRule.data.str_to_decision_rule(strRule: str, ruleType: str = 'binary') → teex.decisionRule.data.DecisionRule`

Converts a string representing a rule into a `DecisionRule` object. The string must contain the individual feature bounds separated by ‘&’. For each feature bound, the feature must appear first. If `ruleType='binary'`, it is not necessary to explicitly specify both bounds: the missing one will be induced. To imply a result, use ‘->’ and follow it with a statement representation. This method is robust to situations like `feature > 3 & feature > 4` and missing whitespaces.

### Example

```

>>> r = 'a != 2.5 -> res > 3'
>>> print(str_to_decision_rule(r, 'unary'))
>>> r = 'a <= 2.5 & a > 1 -> res > 3'
>>> print(str_to_decision_rule(r, 'binary'))
>>> r = 'a <= 2.5 & a > 1 & b > 1 -> res > 3 & res <= 5'
>>> print(str_to_decision_rule(r, 'binary'))
>>> r = 'a <= 2.5 & a > 1 & b > 1 -> res = class0'
>>> print(str_to_decision_rule(r, 'binary'))
>>> print(str_to_decision_rule('d > 1 & d > 3 & d >= 4 & c < 4 & c < 3 & c <= 2->↳
↳home > 1 & home < 3')) # is robust

```

### Parameters

- **strRule** (*str*) – string to convert to rule.
- **ruleType** (*str*) – type of the Statement objects contained within the generated Decision-Rule object.

### teex.decisionRule.eval module

Module for evaluation of decision rule explanations.

`teex.decisionRule.eval.complete_rule_quality`(*gts*: `teex.decisionRule.data.DecisionRule`, *rules*: `teex.decisionRule.data.DecisionRule`, *eps*: `float = 0.1`) → `float`

Computes the complete rule quality (crq) between two decision rules. All ‘Statements’ in both rules must be binary (have upper and lower bounds). The metric is defined as the proportion of lower and upper bounds in a rule explanation whose that are eps-close to the respective lower and upper bounds (same feature) in the ground truth rule explanation amongst those that are  $\neq \infty$ . Mathematically, given two rules  $e, \tilde{e}$  and a similarity threshold  $\varepsilon$ , the quality of  $e$  with respect to  $\tilde{e}$  is:

$$q(e, \tilde{e}) = \frac{1}{N_{\infty}} \sum_{i=1}^{|e|} \delta_{\varepsilon}(e_i, \tilde{e}_i),$$

where

$$\delta_{\varepsilon}(e_i, \tilde{e}_i) = \begin{cases} 1 & \text{if } |e_i - \tilde{e}_i| \leq \varepsilon \wedge |e_i| \neq \infty \wedge |\tilde{e}_i| \neq \infty, 0 \\ \text{otherwise} & \end{cases}$$

Where  $N_{\infty}$  is the number of lower and upper bounds that are different from  $\infty$  in both  $e$  and  $\tilde{e}$ . More about this metric can be found in [Evaluating local explanation methods on ground truth, Riccardo Guidotti, 2021].

#### Example

```
>>> c1 = Statement('a', lowB=2, upperB=1)
>>> c2 = Statement('a', lowB=2.2, upperB=1.1)
>>> r1 = DecisionRule([c1])
>>> r2 = DecisionRule([c2]) # both rules contain the feature 'a'
```

```
>>> print(complete_rule_quality(r1, r2, eps=0.01))
>>> print(complete_rule_quality(r1, r2, eps=0.1))
>>> print(complete_rule_quality(r1, r2, eps=0.2))
```

```
>>> c3 = Statement('b', lowB=2.2, upperB=1.1)
>>> r3 = DecisionRule([c3])
```

```
>>> print(complete_rule_quality(r1, r3, eps=0.2))
```

```
>>> # The metric does not take the absence of a feature into account
>>> r3 = DecisionRule([c3, c2])
>>> print(complete_rule_quality(r1, r3, eps=0.2))
```

#### Parameters

- **gts** – (DecisionRule or array-like of DecisionRules) ground truth rule w.r.t. which to compute the quality

- **rules** – (DecisionRule or array-like of DecisionRules) rule to compute the quality for
- **eps** – (float) threshold  $\varepsilon$  for the bounds to be taken into account in the metric, with precision up to 3 decimal places.

**Returns** (float or ndarray of shape (n\_samples,)) Complete rule quality.

`teex.decisionRule.eval.rule_scores(gts: teex.decisionRule.data.DecisionRule, rules: teex.decisionRule.data.DecisionRule, allFeatures, metrics=None, average=True, crqParams=None) → float`

Quality metrics for `teex.decisionRule.data.DecisionRule` objects.

### Parameters

- **gts** – (DecisionRule or array-like of DecisionRules) ground truth decision rule/s.
- **rules** – (DecisionRule or array-like of DecisionRules) approximated decision rule/s.
- **allFeatures** – (array-like) names of all of the relevant features (i.e. `featureNames` of `teex.decisionRule.data.SenecaDR` object.)
- **metrics** – (array-like of str, default ['fscore']) metrics to compute. Available:
  - 'fscore': Computes the F1 Score between the ground truths and the predicted vectors.
  - 'prec': Computes the Precision Score between the ground truths and the predicted vectors.
  - 'rec': Computes the Recall Score between the ground truths and the predicted vectors.
  - 'crq': Computes the Complete Rule Quality of rule w.r.t. gt.
  - 'auc': Computes the ROC AUC Score between the two rules.
  - 'cs': Computes the Cosine Similarity between the two rules.

Note that for 'fscore', 'prec', 'rec', 'auc' and 'cs' the rules are transformed to binary vectors where there is one entry per possible feature and that entry contains a 1 if the feature is present in the rule, otherwise 0.

- **average** – (bool, default True) Used only if `gts` and `rule` are array-like. Should the computed metrics be averaged across all of the samples?
- **crqParams** (*dict*) – Extra parameters complete rule quality.

### Returns

(ndarray) specified metric/s in the original order. Can be of shape

- (n\_metrics,) if only one DecisionRule has been provided in both `gts` and `rules` or when both are array-like and `average=True`.
- (n\_metrics, n\_samples) if `gts` and `rules` are array-like and `average=False`.

## Module contents

### teex.featureImportance package

### Submodules

## teex.featureImportance.data module

Module for synthetic and real datasets with available ground truth feature importance explanations. Also contains methods and classes for decisionRule data manipulation.

All of the datasets must be instanced first. Then, when sliced, they all return the observations, labels and ground truth explanations, respectively.

```
class teex.featureImportance.data.SenecaFI(nSamples: int = 200, nFeatures: int = 3,  
                                           featureNames=None, randomState: int = 888)
```

Bases: teex.\_baseClasses.\_baseDatasets.\_SyntheticDataset

Generate synthetic binary classification tabular data with ground truth feature importance explanations. This method was presented in [Evaluating local explanation methods on ground truth, Riccardo Guidotti, 2021].

From this class one can also obtain a trained transparent model (instance of *TransparentLinearClassifier*). When sliced, this object will return

- X (ndarray) of shape (nSamples, nFeatures) or (nFeatures). Generated data.
- y (ndarray) of shape (nSamples,) or int. Generated binary data labels.
- explanations (ndarray) of shape (nSamples, nFeatures) or (nFeatures). Generated g.t. feature importance explanations. For each explanation, the values are normalised to the [-1, 1] range.

### Parameters

- **nSamples** – (int) number of samples to be generated.
- **nFeatures** – (int) total number of features in the generated data.
- **featureNames** – (array-like) names of the generated features. If not provided, a list with the generated feature names will be returned by the function.
- **randomState** – (int) random state seed.

```
class teex.featureImportance.data.TransparentLinearClassifier(randomState: int = 888)
```

Bases: teex.\_baseClasses.\_baseClassifier.\_BaseClassifier

Used on the higher level data generation class *SenecaFI* (use that and get it from there preferably).

Transparent, linear classifier with feature importances as explanations. This class also generates labeled data according to the generated random linear expression. Presented in [Evaluating local explanation methods on ground truth, Riccardo Guidotti, 2021].

```
explain(data, newLabels=None)
```

Get feature importance explanation as the gradient of the expression evaluated at the point (from the n ‘training’ observations) with the same class as ‘obs’ and closest to the decision boundary  $f = 0$ .

The procedure is as follows: for each data observation  $x$  to explain, get the observation  $z$  from the ‘training’ data that is closer to the decision boundary and is of different class than  $x$ . Then, get the observation  $t$  from the ‘training’ data that is closer to  $z$  but of the same class as  $x$ . Finally, return the explanation for  $x$  as the gradient vector of  $f$  evaluated at  $t$ .

### Parameters

- **data** – (ndarray) array of  $k$  observations and  $m$  features, shape  $(k, m)$ .
- **newLabels** – (ndarray, optional) precomputed data labels (binary ints) for ‘data’. Shape  $(k)$ .

**Returns** (ndarray)  $(k, m)$  array of feature importance explanations.

**fit**(*nFeatures=None, featureNames=None, nSamples=100*) → None

Generates a random linear expression and random data labeled by the linear expression as a binary dataset.

**Parameters**

- **nFeatures** – (int) number of features in the data.
- **featureNames** – (array-like) names of the features in the data.
- **nSamples** – (int) number of samples for the generated data.

**Returns** (ndarray, ndarray) data of shape (n, m) and their respective labels of shape (n)

**predict**(*data*)

Predicts label for observations. Class 1 if  $f(x) > 0$  and 0 otherwise where  $x$  is a point to label and  $f()$  is the generated classification expression.

**Parameters** **data** – (ndarray) observations to label, shape (k, m).

**Returns** (ndarray) array of length n with binary labels.

**predict\_proba**(*data*)

Get class probabilities by evaluating the expression  $f$  at ‘data’, normalizing the result and setting the probabilities as  $1 - \text{norm}(f(\text{data}))$ ,  $\text{norm}(f(\text{data}))$ .

**Parameters** **data** – (ndarray) observations for which to obtain probabilities, shape (k, m).

**Returns** (ndarray) array of shape (n, 2) with predicted class probabilities.

## teex.featureImportance.eval module

Module for evaluation of feature importance explanations.

`teex.featureImportance.eval.cosine_similarity`(*u, v, bounding: str = 'abs'*) → float

Computes cosine similarity between two real valued arrays. If negative, returns 0.

**Parameters**

- **u** – (array-like), real valued array of dimension n.
- **v** – (array-like), real valued array of dimension n.
- **bounding** (*str*) – if the CS is  $< 0$ , bound it in  $[0, 1]$  via absolute val (‘abs’) or  $\max(0, \text{val})$  (‘max’)

**Return float** (0, 1) cosine similarity.

`teex.featureImportance.eval.feature_importance_scores`(*gts, preds, metrics=None, average=True, thresholdType='abs', binThreshold=0.5, verbose=1*)

Computes quality metrics between one or more feature importance vectors. The values in the vectors must be bounded in  $[0, 1]$  or  $[-1, 1]$  (to indicate negative importances in the second case). If they are not, the values will be mapped.

For the computation of the precision, recall and FScore, the vectors are binarized to simulate a classification setting depending on the param. `thresholdType`. In the case of ROC AUC, the ground truth feature importance vector will be binarized as in the case of ‘precision’, ‘recall’ and ‘FScore’ and the predicted feature importance vector entries will be considered as prediction scores. If the predicted vectors contain negative values, these will be either mapped to 0 or taken their absolute val (depending on the chosen option in the param. `thresholdType`).

**Edge cases:** Edge cases for when metrics are not defined have been accounted for:

- When computing classification scores ('fscore', 'prec', 'rec'), if there is only one class in the ground truth and / or the prediction, one random feature will be flipped (same feature in both). Note that some metrics such as 'auc' may still be undefined in this case if there is only 1 feature per data observation.
- For 'auc', although the ground truth is binarized, the prediction vector represents scores, and so, if both contain only one value, only in the ground truth a feature will be flipped. In the prediction, a small amount ( $1^{-4}$ ) will be summed to a random feature if no value is != 0.
- When computing cosine similarity, if there is no value != 0 in the ground truth and / or prediction, one random feature will be summed  $1e-4$ .

**On vector ranges:** If the ground truth array or the predicted array have values that are not bounded in  $[-1, 1]$  or  $[0, 1]$ , they will be mapped accordingly. Note that if the values lie within  $[-1, 1]$  or  $[0, 1]$  no mapping will be performed, so it is assumed that the scores represent feature importances in those ranges. These are the cases considered for the mapping:

- if values in the  $[0, \infty]$  range: map to  $[0, 1]$
- if values in the  $[-\infty, 0]$  range: map to  $[-1, 1]$
- if values in the  $[-\infty, \infty]$  range: map to  $[-1, 1]$

### Parameters

- **gts** (*np.ndarray*) – (1d *np.ndarray* or 2d *np.ndarray* of shape (n\_features, n\_samples)) ground truth feature importance vectors.
- **preds** (*np.ndarray*) – (1d *np.ndarray* or 2d *np.ndarray* of shape (n\_features, n\_samples)) predicted feature importance vectors.
- **metrics** – (str or array-like of str) metric/s to be computed. Available metrics are
  - 'fscore': Computes the F1 Score between the ground truths and the predicted vectors.
  - 'prec': Computes the Precision Score between the ground truths and the predicted vectors.
  - 'rec': Computes the Recall Score between the ground truths and the predicted vectors.
  - 'auc': Computes the ROC AUC Score between the ground truths and the predicted vectors.
  - 'cs': Computes the Cosine Similarity between the ground truths and the predicted vectors.The vectors are automatically binarized for computing recall, precision and fscore.
- **average** (*bool*) – (default True) Used only if **gt** and **rule** contain multiple observations. Should the computed metrics be averaged across all the samples?
- **thresholdType** (*str*) – Options for the binarization of the features for the computation of 'fscore', 'prec', 'rec' and 'auc'.
  - 'abs': features with absolute val  $\leq$  **binThreshold** will be set to 0 and 1 otherwise. For the predicted feature importances in the case of 'auc', their absolute val will be taken.
  - 'thres': features  $\leq$  **binThreshold** will be set to 0, 1 otherwise. For the *predicted* feature importances in the case of 'auc', negative values will be cast to 0 and the others left *as-is*.
- **binThreshold** (*float*) – (in  $[-1, 1]$ ) Threshold for the binarization of the features for the computation of 'fscore', 'prec', 'rec' and 'auc'. The binarization depends on both this parameter and **thresholdType**. If **thresholdType** = 'abs', **binThreshold** cannot be negative.
- **verbose** (*int*) – Verbosity level of warnings. 1 will report warnings, else will not.

**Returns** (*ndarray* of shape (n\_metrics,) or (n\_samples, n\_metrics)) specified metric/s in the indicated order.



## Module contents

### teex.saliencyMap package

#### Submodules

#### teex.saliencyMap.data module

Module for synthetic and real datasets with available ground truth saliency map explanations. Also contains methods and classes for saliency map data manipulation.

All of the datasets must be instanced first. Then, when sliced, they all return the observations, labels and ground truth explanations, respectively.

#### **class** teex.saliencyMap.data.Kahikatea

Bases: teex.\_baseClasses.\_baseDatasets.\_ClassificationDataset

Binary classification dataset from [Y. Jia et al. (2021) Studying and Exploiting the Relationship Between Model Accuracy and Explanation Quality, ECML-PKDD 2021].

This dataset contains images for Kahikatea (an endemic tree in New Zealand) classification. Positive examples (in which Kahikatea trees can be identified) are annotated with true explanations such that the Kahikatea trees are highlighted. If an image belongs to the negative class, None is provided as an explanation.

#### Example

```
>>> kDataset = Kahikatea()
>>> img, label, exp = kDataset[1]
```

where `img` is a PIL Image, `label` is an int and `exp` is a PIL Image. When a slice is performed, `obs`, `label` and `exp` are lists of the objects described above.

**class** teex.saliencyMap.data.SenecaSM(*nSamples=1000, imageH=32, imageW=32, patternH=16, patternW=16, cellH=4, cellW=4, patternProp=0.5, fillPct=0.4, colorDev=0.1, randomState=888*)

Bases: teex.\_baseClasses.\_baseDatasets.\_SyntheticDataset

Synthetic dataset with available saliency map explanations.

Images and g.t. explanations generated following the procedure presented in [Evaluating local explanation methods on ground truth, Riccardo Guidotti, 2021]. The g.t. explanations are binary ndarray masks of shape (imageH, imageW) that indicate the position of the pattern in an image (zero array if the pattern is not present) and are generated. The generated RGB images belong to one class if they contain a certain generated pattern and to the other if not. The images are composed of homogeneous cells of size (cellH, cellW), which in turn compose a certain pattern of shape (patternH, patternW) that is inserted on some of the generated images.

From this class one can also obtain a trained transparent model (instance of *TransparentImageClassifier*).

When sliced, this object will return

- X (ndarray) of shape (nSamples, imageH, imageW, 3) or (imageH, imageW, 3). Generated image data.
- y (ndarray) of shape (nSamples,) or int. Image labels. 1 if an image contains the pattern and 0 otherwise.
- explanations (ndarray) of shape (nSamples, imageH, imageW) or (imageH, imageW). Ground truth explanations.

#### Parameters

- **nSamples** (*int*) – number of images to generate.

- **imageH** (*int*) – height in pixels of the images. Must be multiple of `cellH`.
- **imageW** (*int*) – width in pixels of the images. Must be multiple of `cellW`.
- **patternH** (*int*) – height in pixels of the pattern. Must be  $\leq$  `imageH` and multiple of `cellH`.
- **patternW** (*int*) – width in pixels of the pattern. Must be  $\leq$  `imageW` and multiple of `cellW`.
- **cellH** (*int*) – height in pixels of each cell.
- **cellW** (*int*) – width in pixels of each cell.
- **patternProp** (*float*) – ([0, 1]) percentage of appearance of the pattern in the dataset.
- **fillPct** (*float*) – ([0, 1]) percentage of cells filled (not black) in each image.
- **colorDev** (*float*) – ([0, 0.5]) maximum val summed to 0 valued channels and minimum val subtracted to 1 valued channels of filled cells. If 0, each cell will be completely red, green or blue. If  $> 0$ , colors may be a mix of the three channels (one  $\sim 1$ , the other two  $\sim 0$ ).
- **randomState** (*int*) – random seed.

**class** `teex.saliencyMap.data.TransparentImageClassifier`

Bases: `teex._baseClasses._baseClassifier._BaseClassifier`

Used on the higher level data generation class `SenecaSM` (use that and get it from there preferably).

Transparent, pixel-based classifier with pixel (features) importances as explanations. Predicts the class of the images based on whether they contain a certain specified pattern or not. Class 1 if they contain the pattern, 0 otherwise. To be trained only a pattern needs to be fed. Follows the sklean API. Presented in [Evaluating local explanation methods on ground truth, Riccardo Guidotti, 2021].

**explain**(*obs: numpy.ndarray*)  $\rightarrow$  `numpy.ndarray`

Explain observations' predictions with binary masks (pixel importance arrays).

**Parameters** *obs* (`np.ndarray`) – array of n images as ndarrays.

**Returns** list with n binary masks as explanations.

**fit**(*pattern: numpy.ndarray, cellH: int = 1, cellW: int = 1*)  $\rightarrow$  `None`

Fits the model.

**predict**(*obs: numpy.ndarray*)  $\rightarrow$  `numpy.ndarray`

Predicts the class for each observation.

**Parameters** *obs* (`np.ndarray`) – array of n images as ndarrays of `np.float32` type.

**Returns** array of n predicted labels.

**predict\_proba**(*obs: numpy.ndarray*)  $\rightarrow$  `numpy.ndarray`

Predicts probability that each observation belongs to class 1 or 0. Probability of class 1 will be 1 if the image contains the pattern and 0 otherwise.

**Parameters** *obs* (`np.ndarray`) – array of n images as ndarrays.

**Returns** array of n probability tuples of length 2.

`teex.saliencyMap.data.binarize_rgb_mask`(*img, bgValue='high'*)  $\rightarrow$  `numpy.array`

Binarizes a RGB binary mask, letting the background (negative class) be 0. Use this function when the image to binarize has a very defined background.

**Parameters**

- **img** – (`ndarray`) of shape (imageH, imageW, 3), RGB mask to binarize.
- **bgValue** – (`str`) Intensity of the negative class of the image to binarize: { 'high', 'low' }

**Returns** (ndarray) a binary mask.

`teex.saliencyMap.data.rgb_to_grayscale(img)`

Transforms a 3 channel RGB image into a grayscale image (1 channel).

**Parameters** `img` (*np.ndarray*) – of shape (imageH, imageW, 3)

**Return** `np.ndarray` of shape (imageH, imageW)

## teex.saliencyMap.eval module

Module for evaluation of saliency map explanations.

`teex.saliencyMap.eval.saliency_map_scores(gts, sMaps, metrics=None, binThreshold=0.01, gtBackgroundVals='high', average=True)`

Quality metrics for saliency map explanations, where each pixel is considered as a feature. Computes different scores of a saliency map explanation w.r.t. its ground truth explanation (a binary mask).

### Parameters

- **gts** (*np.ndarray*) – ground truth RGB or binary mask/s. Accepted shapes are
  - (*imageH, imageW*) A single grayscale mask, where each pixel should be 1 if it is part of the salient class and 0 otherwise.
  - (*imageH, imageW, 3*) A single RGB mask, where pixels that **do not** contain the salient class are all either black (all channels set to 0) or white (all channels set to max.).
  - (*nSamples, imageH, imageW*) Multiple grayscale masks, where for each where, in each image, each pixel should be 1 if it is part of the salient class and 0 otherwise.
  - (*nSamples, imageH, imageW, 3*) Multiple RGB masks, where for each image, pixels that *do not* contain the salient class are all either black (all channels set to 0) or white (all channels set to max.).

If the g.t. masks are RGB they will be binarized (see param `gtBackground` to specify the color of the pixels that pertain to the non-salient class).

- **sMaps** (*np.ndarray*) – grayscale saliency map explanation/s ([0, 1] or [-1, 1] normalised). Supported shapes are
  - (*imageH, imageW*) A single explanation
  - (*nSamples, imageH, imageW*) Multiple explanations
- **metrics** – (str / array-like of str, default=['auc']) Quality metric/s to compute. Available:
  - 'auc': ROC AUC score. The val of each pixel of each saliency map in `sMaps` is considered as a prediction probability of the pixel pertaining to the salient class.
  - 'fscore': F1 Score.
  - 'prec': Precision Score.
  - 'rec': Recall score.
  - 'cs': Cosine Similarity.

For 'fscore', 'prec', 'rec' and 'cs', the saliency maps in `sMaps` are binarized (see param `binThreshold`).

- **binThreshold** (*float*) – (in [0, 1]) pixels of images in `sMaps` with a val bigger than this will be set to 1 and 0 otherwise when binarizing for the computation of 'fscore', 'prec', 'rec' and 'auc'.

- **gtBackgroundVals** (*str*) – Only used when provided ground truth explanations are RGB. Color of the background of the g.t. masks ‘low’ if pixels in the mask representing the non-salient class are dark, ‘high’ otherwise).
- **average** (*bool*) – (default True) Used only if `gts` and `sMaps` contain multiple observations. Should the computed metrics be averaged across all of the samples?

**Returns**

specified metric/s in the original order. Can be of shape

- (*n\_metrics*,) if only one image has been provided in both `gts` and `sMaps` or when both are contain multiple observations and `average=True`.
- (*n\_metrics*, *n\_samples*) if `gts` and `sMaps` contain multiple observations and `average=False`.

**Return type** `np.ndarray`

**Module contents****teex.wordImportance package****Submodules****teex.wordImportance.data module**

Module for real datasets with available ground truth word importance explanations. Also contains methods and classes for word importance data manipulation.

**class** `teex.wordImportance.data.Newsgroup`

Bases: `teex._baseClasses._baseDatasets._ClassificationDataset`

20 Newsgroup dataset from <https://github.com/SinaMohseni/ML-Interpretability-Evaluation-Benchmark>

Contains 188 human annotated newsgroup texts belonging to two categories.

**Example**

```
>>> nDataset = Newsgroup()
>>> obs, label, exp = nDataset[1]
```

where `obs` is a `str`, `label` is an `int` and `exp` is a `dict`. containing a score for each important word in `obs`. When a slice is performed, `obs`, `label` and `exp` are lists of the objects described above.

**teex.wordImportance.eval module**

Module for evaluation of word importance explanations.

`teex.wordImportance.eval.word_importance_scores`(*gts*: `Union[Dict[str, float], List[Dict[str, float]]]`,  
*preds*: `Union[Dict[str, float], List[Dict[str, float]]]`,  
*vocabWords*: `Optional[Union[List[str], List[List[str]]]] = None`, *metrics*: `Optional[Union[str, List[str]]] = None`, *binThreshold*: `float = 0.5`,  
*average*: `bool = True`) → `numpy.ndarray`

Quality metrics for word importance explanations, where each word is considered as a feature. An example of an explanation:

```
>>> {'skate': 0.7, 'to': 0.2, 'me': 0.5}
```

### Parameters

- **gts** – (dict, array-like of dicts) ground truth word importance/s, where each BOW is represented as a dictionary with words as keys and floats as importances. Importances must be in  $[0, 1]$  or  $[-1, 1]$ .
- **preds** – (dict, array-like of dicts) predicted word importance/s, where each BOW is represented as a dictionary with words as keys and floats as importances. Importances must be in the same scale as param. `gts`.
- **vocabWords** – (array-like of str 1D or 2D for multiple reference vocabularies, default None) Vocabulary words. If None, the union of the words in each ground truth and predicted explanation will be interpreted as the vocabulary words. This is needed for when explanations are converted to feature importance vectors. If this parameter is provided as a 1D list, the vocabulary words will be the same for all explanations, but if not provided or given as a 2D array-like (same number of reference vocabularies as there are explanations), different vocabulary words will be considered for each explanation.
- **metrics** – (str / array-like of str, default=['prec']) Quality metric/s to compute. Available:
  - All metrics in `teex.featureImportance.eval.feature_importance_scores()`.
- **binThreshold** (*float*) – (in  $[0, 1]$ , default .5) pixels of images in `sMaps` with a val bigger than this will be set to 1 and 0 otherwise when binarizing for the computation of 'fscore', 'prec', 'rec' and 'auc'.
- **average** (*bool*) – (default True) Used only if `gts` and `preds` contain multiple observations. Should the computed metrics be averaged across all samples?

### Returns

specified metric/s in the original order. Can be of shape:

- (n\_metrics,) if only one image has been provided in both `gts` and `preds` or when both are contain multiple observations and `average=True`.
- (n\_metrics, n\_samples) if `gts` and `preds` contain multiple observations and `average=False`.

**Return type** np.ndarray

`teex.wordImportance.eval.word_to_feature_importance(wordImportances, vocabWords) → list`  
Maps words with importance weights into a feature importance vector.

### Parameters

- **wordImportances** – (dict or array-like of dicts) words with feature importances as values with the same format as described in the method `word_importance_scores()`.
- **vocabWords** – (array-like of str, 1D or 2D for multiple reference vocabularies) *m* words that should be taken into account when transforming into vector representations. Their order will be followed.

### Returns

Word importances as feature importance vectors. Return types:

- list of np.ndarray, if multiple vocabularies because of the possible difference in size of the reference vocabularies in each explanation.

- np.ndarray, if only 1 vocabulary

### Example

```
>>> word_to_feature_importance({'a': 1, 'b': .5},['a', 'b', 'c'])
>>> [1, .5, 0]
>>> word_to_feature_importance([{'a': 1, 'b': .5}, {'b': .5, 'c': .9}],['a', 'b', 'c
↪'])
>>> [[1, .5, 0. ], [0, .5, .9]]
```

## Module contents

### 1.1.2 Module contents

## INDICES AND TABLES

- genindex
- modindex
- search





## PYTHON MODULE INDEX

### t

- teex, 18
- teex.decisionRule, 9
- teex.decisionRule.data, 3
- teex.decisionRule.eval, 8
- teex.featureImportance, 13
- teex.featureImportance.data, 10
- teex.featureImportance.eval, 11
- teex.saliencyMap, 16
- teex.saliencyMap.data, 13
- teex.saliencyMap.eval, 15
- teex.wordImportance, 18
- teex.wordImportance.data, 16
- teex.wordImportance.eval, 16



## INDEX

### B

`binarize_rgb_mask()` (in module `teex.saliencyMap.data`), 14

### C

`complete_rule_quality()` (in module `teex.decisionRule.eval`), 8

`cosine_similarity()` (in module `teex.featureImportance.eval`), 11

### D

`DecisionRule` (class in `teex.decisionRule.data`), 3

`delete_statement()` (`teex.decisionRule.data.DecisionRule` method), 4

### E

`explain()` (`teex.decisionRule.data.TransparentRuleClassifier` method), 6

`explain()` (`teex.featureImportance.data.TransparentLinearClassifier` method), 10

`explain()` (`teex.saliencyMap.data.TransparentImageClassifier` method), 14

### F

`feature_importance_scores()` (in module `teex.featureImportance.eval`), 11

`fit()` (`teex.decisionRule.data.TransparentRuleClassifier` method), 6

`fit()` (`teex.featureImportance.data.TransparentLinearClassifier` method), 10

`fit()` (`teex.saliencyMap.data.TransparentImageClassifier` method), 14

### G

`get_features()` (`teex.decisionRule.data.DecisionRule` method), 4

### I

`insert_statement()` (`teex.decisionRule.data.DecisionRule` method), 4

### K

`Kahikatea` (class in `teex.saliencyMap.data`), 13

### M

module

`teex`, 18

`teex.decisionRule`, 9

`teex.decisionRule.data`, 3

`teex.decisionRule.eval`, 8

`teex.featureImportance`, 13

`teex.featureImportance.data`, 10

`teex.featureImportance.eval`, 11

`teex.saliencyMap`, 16

`teex.saliencyMap.data`, 13

`teex.saliencyMap.eval`, 15

`teex.wordImportance`, 18

`teex.wordImportance.data`, 16

`teex.wordImportance.eval`, 16

### N

`NewsGroup` (class in `teex.wordImportance.data`), 16

### P

`predict()` (`teex.decisionRule.data.TransparentRuleClassifier` method), 6

`predict()` (`teex.featureImportance.data.TransparentLinearClassifier` method), 11

`predict()` (`teex.saliencyMap.data.TransparentImageClassifier` method), 14

`predict_proba()` (`teex.decisionRule.data.TransparentRuleClassifier` method), 6

`predict_proba()` (`teex.featureImportance.data.TransparentLinearClassifier` method), 11

`predict_proba()` (`teex.saliencyMap.data.TransparentImageClassifier` method), 14

### R

`rename_statement()` (`teex.decisionRule.data.DecisionRule` method), 4

`replace_statement()` (`teex.decisionRule.data.DecisionRule` method), 4

rgb\_to\_grayscale() (in module  
*teex.saliencyMap.data*), 15

rule\_scores() (in module *teex.decisionRule.eval*), 9

rule\_to\_feature\_importance() (in module  
*teex.decisionRule.data*), 6

rulefit\_to\_decision\_rule() (in module  
*teex.decisionRule.data*), 6

## S

saliency\_map\_scores() (in module  
*teex.saliencyMap.eval*), 15

SenecaDR (class in *teex.decisionRule.data*), 4

SenecaFI (class in *teex.featureImportance.data*), 10

SenecaSM (class in *teex.saliencyMap.data*), 13

set\_result() (*teex.decisionRule.data.DecisionRule*  
method), 4

Statement (class in *teex.decisionRule.data*), 5

str\_to\_decision\_rule() (in module  
*teex.decisionRule.data*), 7

## T

teex

module, 18

teex.decisionRule

module, 9

teex.decisionRule.data

module, 3

teex.decisionRule.eval

module, 8

teex.featureImportance

module, 13

teex.featureImportance.data

module, 10

teex.featureImportance.eval

module, 11

teex.saliencyMap

module, 16

teex.saliencyMap.data

module, 13

teex.saliencyMap.eval

module, 15

teex.wordImportance

module, 18

teex.wordImportance.data

module, 16

teex.wordImportance.eval

module, 16

TransparentImageClassifier (class in  
*teex.saliencyMap.data*), 14

TransparentLinearClassifier (class in  
*teex.featureImportance.data*), 10

TransparentRuleClassifier (class in  
*teex.decisionRule.data*), 6

## U

upsert\_statement() (*teex.decisionRule.data.DecisionRule*  
method), 4

## W

word\_importance\_scores() (in module  
*teex.wordImportance.eval*), 16

word\_to\_feature\_importance() (in module  
*teex.wordImportance.eval*), 17