# teex

*Release 1.1.2*

**Chus Antonanzas**

**Sep 10, 2023**

# CONTENTS

A Python Toolbox for the evaluation of machine learning explanations.

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

Visit our GitHub for source.

CHAPTER

ONE

# 1. USAGE OVERVIEW

`teex` is divided into subpackages, one for each explanation type. Each subpackage contains two modules, focused on two distinct functionalities:

- `eval`: contains *evaluation* methods for that particular explanation type. For every subpackage, there is one high-level function to easily compute all the available metrics for an arbitrary number of explanations.
- `data`: contains *data* classes with available g.t. explanations of that particular explanation type, both synthetic and real. All of them are objects that need to be instanced and, when sliced, will return the data, the target and the ground truth explanations, respectively.

## 1.1 1.1. Evaluation (with feature importance as an example)

**What are feature importance vectors?** They are vectors with one entry per feature. Each entry contains a weight that represents a feature's importance for the observation's outcome. Weights are usually in the range $[-1, 1]$.

Suppose that we have a dataset with available g.t. explanations (`gtExps`) and a model trained with it (`model`):

```
from teex.featureImportance.eval import feature_importance_scores

# get individual feature importance explanations with any method
predictedExps = get_explanations(model, X)

# evaluate predicted explanations against ground truths
feature_importance_scores(gtExps, predictedExps, metrics=['fscore', 'cs', 'auc'])
```

This basic syntax is followed by the main evaluation APIs of all 4 explanation types:

- **Feature Importance**: `feature_importance_scores`
- **Saliency Maps**: `saliency_map_scores`
- **Decision Rules**: `rule_scores`
- **Word Importance**: `word_importance_scores`

Other functionalities are included in each evaluation module.

## 1.2 1.2. Metrics supported

Metrics available as of `v1.0.0` are

- **Feature Importance**
  - **Cosine Similarity**: similarity between the two vectors is measured in an inner product space in terms of orientation.
  - **ROC AUC**: where the ground truth is binarized in order for it to represent a class and the predicted vector entries are interpreted as classification scores or likelihood.
  - **F1 Score**: where both ground truth and prediction are binarized according to a user-defined threshold.
  - **Precision**: g.t. and prediction treated as in F1 Score
  - **Recall**: g.t. and prediction treated as in F1 Score
- **Saliency Maps**
  - Same metrics as in feature importance. Each pixel in an image is considered to be a feature.
- **Decision Rules**
  - **Complete Rule Quality**: Proportion of lower and upper bounds in a rule explanation whose that are $\epsilon$-close to the respective lower and upper bounds (same feature) in the ground truth rule explanation amongst those that are not infinity.
  - All metrics in feature importance, where a transformation of the rule into feature importance vectors is performed first. See doc. for details.
- **Word Importance**:
  - All metrics in feature importance, where a vocabulary is considered the feature space and a word importance explanation may or may not contain words from the vocabulary.

Note how in **teex**, feature importance vectors are a universal representation: we 'translate' all other explanation types to feature importance vectors to allow a wider metric space.

## 1.3 1.3. Datasets

**teex** also provides an easy way to get and use data with available ground truth explanations. It contains real datasets and can generate synthetic ones. All of them are instanced as objects, and can be sliced as usual. For example:

```
from teex.saliencyMap.data import Kahikatea
X, y, exps = Kahikatea()[:]
```

downloads and assigns data from the Kahikatea dataset:

Other datasets, such as CUB-200-2011 and the Oxford-IIIT Pet Dataset, are available on **teex**, with over 19000 images and 230 distinct classes combined:

```
from teex.saliencyMap.data import CUB200
X, y, exps = CUB200()[:]
```

```
from teex.saliencyMap.data import OxfordIIIT
X, y, exps = OxfordIIIT()[:]
```

Synthetic datasets can also be easily generated:

```
from teex.saliencyMap.data import SenecaSM
X, y, exps = SenecaSM()[:]
```

Datasets for all other explanation types are available too.

CHAPTER

TWO

# 2. EXAMPLES

Here you will find hands-on examples of `teex`. For each explanation type, explanation evaluation and data generation are showcased. Advanced examples and experiments can also be found within the `saliency map` and `model selection` sections.

## 2.1 Notebook demos

### 2.1.1 Saliency map

Here you will find demos for the *saliency map* explanation type.

#### Generating image data with g.t. saliency maps

Let's explore the 'seneca' method for generating artificial images with available ground truth saliency maps. It was presented in Evaluating local explanation methods on ground truth, Riccardo Guidotti, 2021, and although the original intention of the paper was not to present an artificial data generator, it can serve our purpose.

[1]:
```
import matplotlib.pyplot as plt
import numpy as np

import teex

import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F

from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

from math import floor
```

### 1. Generating synthetic images

Let's generate artificial image data with the 'seneca' method. In this case, the generated images are composed of squared cells of a fixed size and randomly colored as (almost) Red, Green or Blue. A number of these images contain a randomly generated pattern such that the ones that do so are labeled as '1' and the ones that are not are labeled as '0'. If an image contains the pattern, then the ground truth explanation is a binary mask of the same dimensions where the pattern is highlighted. The user can control:

- Image width and height, in pixels
- Image cell width and height, in pixels and divisor of image width and height
- The proportion of the image that should be filled with cells (fillPct)
- Pattern height and width, in pixels. The number of pixels the randomly generated pattern will take (divisor of image width and height). The previous parameter 'fillPct' also specifies the number of cells filled in the pattern.
- The percentage of images that contain the pattern 'patternProp'
- colorDev: [0, 0.5] If 0, each cell will be completely red, green or blue. The greater (max 0.5), the more mixed will colored channels be. Adds complexity to the task of classification.

[3]:
```
from teex.saliencyMap.data import SenecaSM

nSamples = 100
randomState = 8
imageH, imageW = 32, 32
patternH, patternW = 16, 16
cellH, cellW = 4, 4
patternProp = 0.5
fillPct = 0.4
colorDev = 0.1

dataGen = SenecaSM(nSamples=nSamples, imageH=imageH, imageW=imageW,
                        patternH=patternH, patternW=patternW,
                        cellH=cellH, cellW=cellW, patternProp=patternProp,
                        fillPct=fillPct, colorDev=colorDev, randomState=randomState)

X, y, exps = dataGen[:]
pat = dataGen.pattern
```

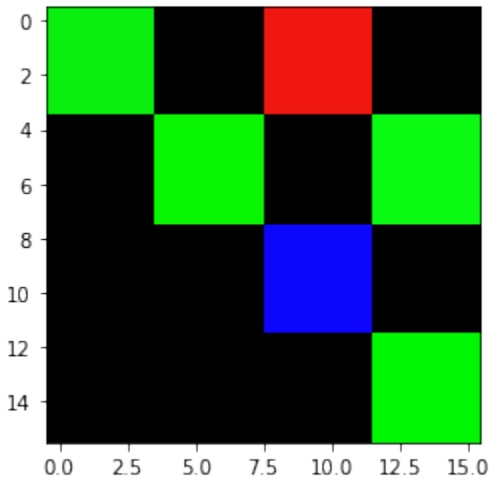
**X** contains the generated images, **y** the labels, **exps** the ground truth explanations and **pat** the exact pattern contained by the images.

### 2. Exploring the images

Some of the generated images contain the following pattern:

[4]:
```
plt.imshow(pat)
```

[4]:
```
<matplotlib.image.AxesImage at 0x142066cd0>
```

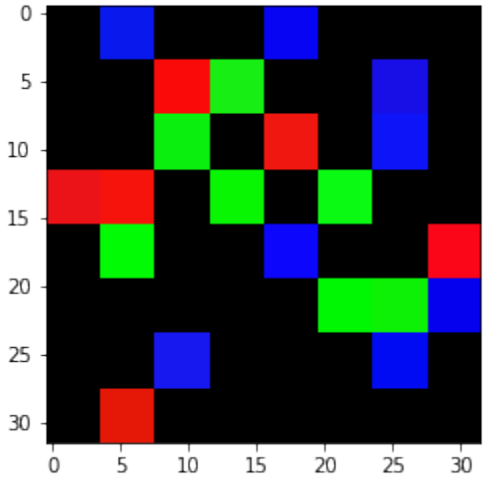



For example, the first one, which is labeled as

[5]:
```
print(y[0])
plt.imshow(X[0])
```

```
1
```

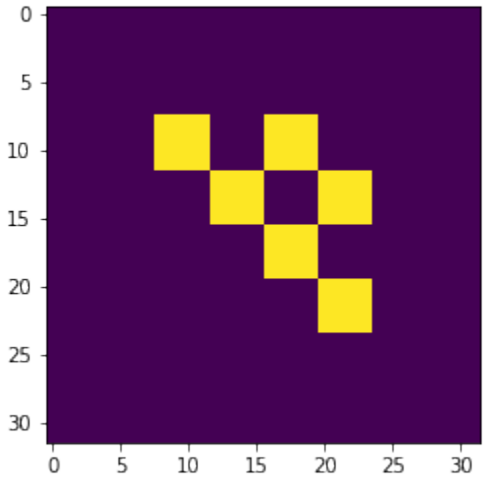
[5]:
```
<matplotlib.image.AxesImage at 0x1424ee7f0>
```




contains it

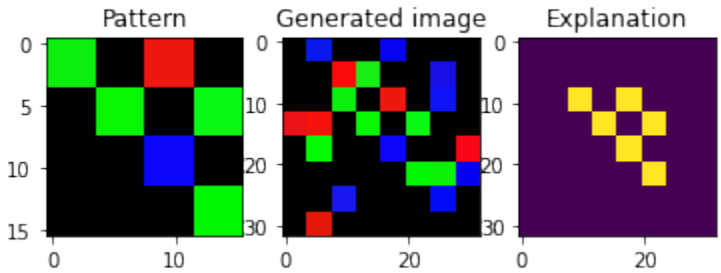
[6]:
```
plt.imshow(exps[0])
```

[6]:
```
<matplotlib.image.AxesImage at 0x142b78a60>
```



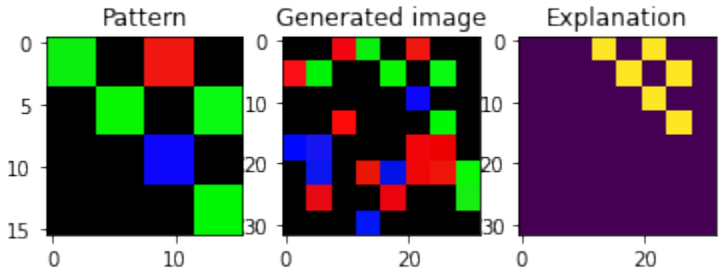
for a more clear view:

[7]:
```
def plt_imgs(p, img, exp):
    fig, axs = plt.subplots(1, 3)
    axs[0].imshow(p)
    axs[0].set_title('Pattern')
    axs[1].imshow(img)
    axs[1].set_title('Generated image')
    axs[2].imshow(exp)
    axs[2].set_title('Explanation')

plt_imgs(pat, X[0], exps[0])
```



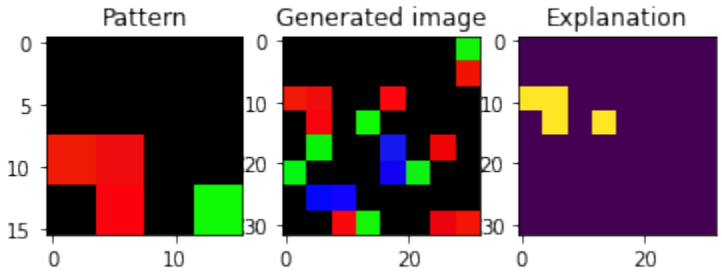

Not all images have the pattern in the same position:

[8]:
```
plt_imgs(pat, X[6], exps[6])
```




We can generate images with another pattern by changing the random state

[10]:
```
dataGen = SenecaSM(nSamples=100, imageH=imageH, imageW=imageW,
                   patternH=patternH, patternW=patternW,
                   cellH=cellH, cellW=cellW, patternProp=patternProp,
                   fillPct=fillPct, colorDev=colorDev, randomState=7)

X, y, exps = dataGen[:]
pat = dataGen.pattern

plt_imgs(pat, X[1], exps[1])
```



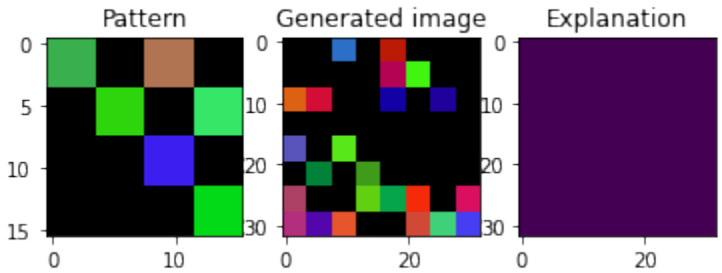

The images that do not contain a pattern have as explanation a black mask. Note that in order for the images to contain the pattern it must not only match the shape, but its colors too. We can check that we have the desired proportion of classes in the dataset:

[11]:
```
sum(y) / len(y) == patternProp
```

[11]:
```
True
```

We can also check how changing the parameter colorDev affects the coloring of the images

[12]:
```
dataGen = SenecaSM(nSamples=nSamples, imageH=imageH, imageW=imageW,
                   patternH=patternH, patternW=patternW,
                   cellH=cellH, cellW=cellW, patternProp=patternProp,
                   fillPct=fillPct, colorDev=0.5, randomState=randomState)

X, y, exps = dataGen[:]
pat = dataGen.pattern

plt_imgs(pat, X[1], exps[1])
```




Indeed, the pattern and the cells that are filled in the images are the same, but the colors are different.

### 3. Exploring a white-box model

The 'seneca' method used to generate the artificial data in TAIAOexp can also return an underlying white-box model. In the case of the image data, the model can recognize if the generated pattern is contained within an observation by performing a linear scan. The models implement .fit, .predict and .predict_proba methods in order for them to easily work with explainability frameworks. We can retrieve the whitebox model by setting the parameter 'returnModel' to **True** when generating the data.

[13]:
```
dataGen = SenecaSM(nSamples=100, imageH=imageH, imageW=imageW,
                   patternH=patternH, patternW=patternW,
                   cellH=cellH, cellW=cellW, patternProp=patternProp,
                   fillPct=fillPct, colorDev=0.5, randomState=7)

X, y, exps = dataGen[:]
pat = dataGen.pattern
model = dataGen.transparentModel  # the underlying transparent model
```

[14]:
```
model
```

[14]:
```
<teex.saliencyMap.data.TransparentImageClassifier at 0x142e7fb50>
```

[15]:
```
model.predict(X[:5])
```

[15]:
```
[1, 1, 0, 0, 0]
```

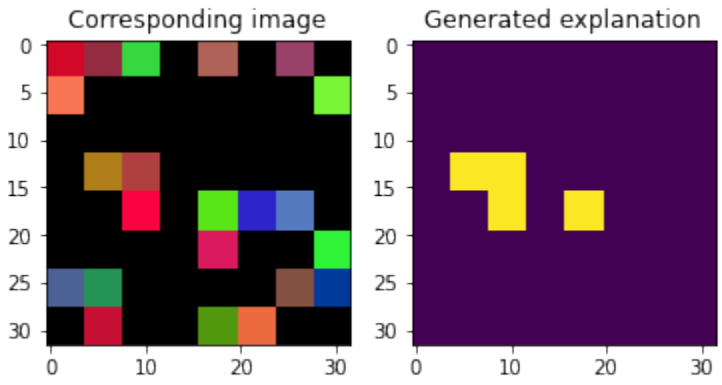
[16]:
```
model.predict_proba(X[:5])
```

[16]:
```
[[0.0, 1.0], [0.0, 1.0], [1.0, 0.0], [1.0, 0.0], [1.0, 0.0]]
```

The model can also 'explain' instances dynamically:

[17]:
```
explanations = model.explain(X[:2])

fig, axs = plt.subplots(1, 2)
axs[1].imshow(explanations[0])
axs[1].set_title('Generated explanation')
axs[0].imshow(X[0])
axs[0].set_title('Corresponding image')
```

[17]:
```
Text(0.5, 1.0, 'Corresponding image')
```



### 4. Loading Kahikata image data

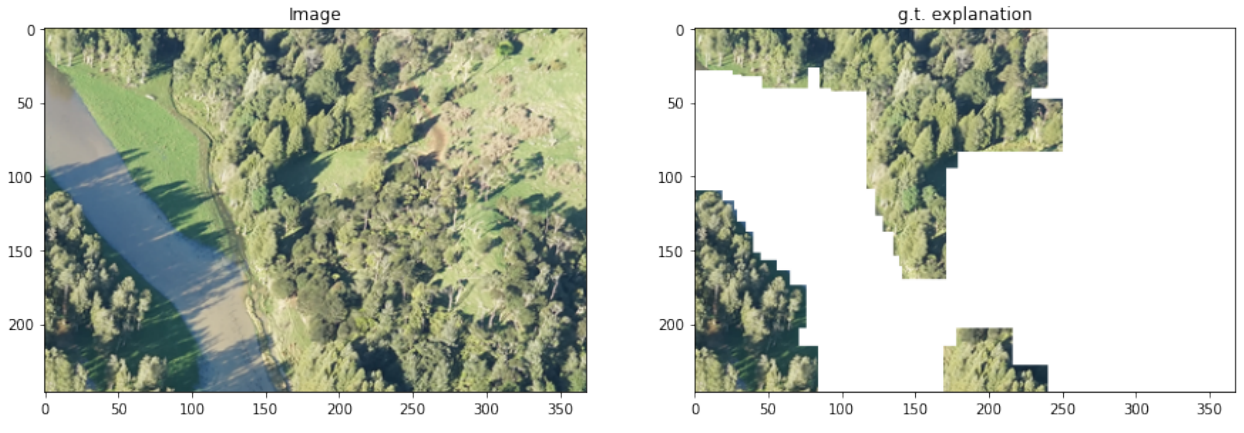
teex includes real datasets with available ground truth explanations. For example, the Kahikatea dataset contains 519 images, and the task is to tell whether each observation contains Kahikatea trees or not. There are 232 positive observations and 287 negative ones.

In teex, the non-artificial datasets are implemented as classes, similarly to PyTorch. After instancing the class, the data itself will be downloaded if it has not been used before. Once done, one can slice it to obtain observations. Each observation contains the data point, the label and the ground truth explanation.

[ ]:
```
from teex.saliencyMap.data import Kahikatea

kahikateaData = Kahikatea()
kData, kLabels, kExps = kahikateaData[:]
```

[21]:
```
i = 0
fig, axs = plt.subplots(1, 2, figsize=(15,15))
axs[0].imshow(kData[i])
axs[0].set_title('Image')
axs[1].imshow(kExps[i])
axs[1].set_title('g.t. explanation')
```

[21]:
```
Text(0.5, 1.0, 'g.t. explanation')
```




## Evaluation of explanation quality: saliency maps

In this notebook, we are going to explore how we can use **teex** to compare image explanation methods.

### 1. Evaluating synthetic image data explanations

In this section, we are going to 1. Generate image data with available g.t. explanations using the 'seneca' method. 2. Create and train a pytorch classifier that will learn to recognize the pattern in the images. 3. Generate explanations with some model agnostic methods and evaluate them w.r.t. the ground truth explanations.

[ ]:
```
%pip install teex
%pip install captum
%pip install torch==1.7.1+cu110 torchvision==0.8.2+cu110 -f https://download.pytorch.org/
↪whl/torch_stable.html
```

[2]:
```
import matplotlib.pyplot as plt

import numpy as np
import pandas as pd

import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F

import PIL
import pickle

from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, f1_score

from math import floor

from captum.attr import GradientShap, IntegratedGradients, Occlusion, DeepLift, Lime,␣
↪GuidedBackprop, GuidedGradCam

from teex.saliencyMap.data import SenecaSM
from teex.saliencyMap.eval import saliency_map_scores

from teex._utils._arrays import _minmax_normalize_array
```

[ ]:
```
from google.colab import drive
drive.mount('/content/drive')
```

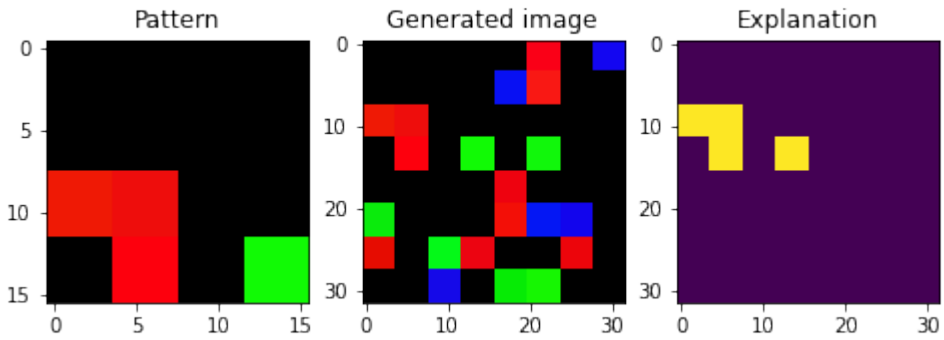
```
Mounted at /content/drive
```

### 1.1. Generating the data

[3]:
```
cellH, cellW = 4, 4
imH, imW = 32, 32

generator = SenecaSM(imageH=imH, imageW=imW,
                     cellH=cellH, cellW=cellW,
                     nSamples=5000, randomState=7)

X, y, exps = generator[:]
pattern = generator.pattern
```

[4]:
```
i = 2
fig, axs = plt.subplots(1, 3, figsize=(8, 8))
axs[0].imshow(pattern)
axs[0].set_title('Pattern')
axs[1].imshow(X[i])
axs[1].set_title('Generated image')
axs[2].imshow(exps[i])
axs[2].set_title('Explanation')
```

[4]:
```
Text(0.5, 1.0, 'Explanation')
```




### 1.2. Declaring and training the model

Declare a simple LeNet variant and its training routine.

[4]:
```
class FCNN(nn.Module):
    """ Basic NN for image classification. """

    def __init__(self, imH, imW, cellH, randomState=1):
        super(FCNN, self).__init__()
        stride = 1
        kSize = 5
        torch.manual_seed(randomState)
        self.conv1 = nn.Conv2d(3, 6, kernel_size=kSize, stride=stride)
        self.H1, self.W1 = floor(((imH - kSize) / stride) + 1), floor(((imW - kSize) /␣
↪stride) + 1)
        self.conv2 = nn.Conv2d(6, 3, kernel_size=kSize, stride=stride)
        self.H2, self.W2 = floor(((self.H1 - kSize) / stride) + 1), floor(((self.W1 -␣
↪kSize) / stride) + 1)
        self.conv3 = nn.Conv2d(3, 1, kernel_size=kSize, stride=stride)
        self.H3, self.W3 = floor(((self.H2 - kSize) / stride) + 1), floor(((self.W2 -␣
↪kSize) / stride) + 1)
        self.fc1 = nn.Linear(self.H3 * self.W3, 100)
        self.fc2 = nn.Linear(100, 2)

    def forward(self, x):
        if len(x.shape) == 3:
            # single instance, add the batch dimension
            x = x.unsqueeze(0)
        x = nn.ReLU()(self.conv1(x))
        x = nn.ReLU()(self.conv2(x))
        x = nn.ReLU()(self.conv3(x))
        x = nn.ReLU()(self.fc1(x.view(x.shape[0], -1)))
        x = self.fc2(x)
        return x

import copy
# sample training function for classification
def train_net(model, data, criterion, optimizer, device, batchSize, nEpochs,␣
↪randomState=888):
```

(continues on next page)

(continued from previous page)

```
    """ data: dict with 'train' and 'val' entries. Each entry is a list with X: FloatTensor,
↪ y: LongTensor """
    torch.manual_seed(randomState)
    bestValAcc = -np.inf
    bestModelWeights = copy.deepcopy(model.state_dict())
    for epoch in range(nEpochs):
        for phase in ['train', 'val']:
            if phase == 'train':
                model.train()
            else:
                model.eval()

            lossVal = .0
            corrects = 0

            for batch in range(int(len(data[phase][0]) / batchSize)):
                XBatch = data[phase][0][batch:batch + batchSize].to(device)
                yBatch = data[phase][1][batch:batch + batchSize].to(device)

                model.zero_grad()

                with torch.set_grad_enabled(phase == 'train'):

                    out = model(XBatch)
                    loss = criterion(out, yBatch)

                    if phase == 'train':
                        loss.backward()
                        optimizer.step()

                    _, preds = torch.max(out, 1)

                lossVal += loss.item() * XBatch.size(0)
                corrects += torch.sum(preds == yBatch.data)

            epochLoss = lossVal / len(data[phase][0])
            epochAcc = corrects.double() / len(data[phase][0])
            print(f'{phase} Loss: {round(epochLoss, 4)} Acc: {round(epochAcc.item(), 4)}
↪')

            if phase == 'val' and epochAcc > bestValAcc:
                bestValAcc = epochAcc
                bestModelWeights = copy.deepcopy(model.state_dict())

        model.load_state_dict(bestModelWeights)

    return model, bestValAcc
```

We cast the images to torch.Tensor type and get train, validation and test splits.

[6]:
```
XTrain, XTest, yTrain, yTest, expsTrain, expsTest = train_test_split(X, y, exps, train_
↪size=0.8, random_state=7)
```

(continues on next page)

(continued from previous page)

```
XTrain, XVal, yTrain, yVal, expsTrain, expsVal = train_test_split(XTrain, yTrain,␣
↪expsTrain, train_size=0.75, random_state=7)

XTrain = torch.FloatTensor(XTrain).permute(0, 3, 1, 2)
yTrain = torch.LongTensor(yTrain)
XVal = torch.FloatTensor(XVal).permute(0, 3, 1, 2)
yVal = torch.LongTensor(yVal)
XTest = torch.FloatTensor(XTest).permute(0, 3, 1, 2)
yTest = torch.LongTensor(yTest)
```

[7]:
```
print(f'Data proportions -> Train: {round(len(XTrain) / len(X), 3)}, Val:
↪{round(len(XVal) / len(X), 3)}, Test: {round(len(XTest) / len(X), 3)}')
print(f'Positive label proportions -> Train: {round((sum(yTrain) / len(yTrain)).item(),␣
↪3)}, \
Val: {round((sum(yVal) / len(yVal)).item(), 3)}, Test: {round((sum(yTest) / len(yTest)).
↪item(), 3)}')
```

```
Data proportions -> Train: 0.6, Val: 0.2, Test: 0.2
Positive label proportions -> Train: 0.489, Val: 0.526, Test: 0.507
```

and train the network

[ ]:
```
nFeatures = len(XTrain[0].flatten())
criterion = nn.CrossEntropyLoss()
model = FCNN(imH=32, imW=32, cellH=4)
optimizer = optim.Adam(model.parameters(), lr=1e-3)
data = {'train': [XTrain, yTrain], 'val': [XVal, yVal]}

if torch.cuda.is_available():
    device = torch.device("cuda:0")

    model.to(device)
    data["train"] = [XTrain.to(device), yTrain.to(device)]
    data["val"] = [XVal.to(device), yVal.to(device)]
else:
    device = torch.device("cpu")

model, valAcc = train_net(model, data, criterion, optimizer, device, batchSize=10,␣
↪nEpochs=10, randomState=7)
```

[9]:
```
if torch.cuda.is_available():
    print(f'Validation F1: {round(f1_score(yVal, F.softmax(model(XVal.cuda()), dim=-1).
↪cpu().argmax(dim=1).numpy()), 3)}')
    print(f'Test F1: {round(f1_score(yTest, F.softmax(model(XTest.cuda()), dim=-1).cpu().
↪argmax(dim=1).numpy()), 3)}')
else:
    print(f'Validation F1: {round(f1_score(yVal, F.softmax(model(torch.
↪FloatTensor(XVal)), dim=-1).argmax(dim=1).detach().numpy()), 3)}')
    print(f'Test F1: {round(f1_score(yTest, F.softmax(model(torch.FloatTensor(XTest)),␣
↪dim=-1).argmax(dim=1).detach().numpy()), 3)}')
```

```
Validation F1: 0.988
Test F1: 0.983
```

### 1.3. Generating and evaluating explanations

With the model trained on the synthetic images, we generate explanations (with Captum, but feel free to use other methods!). First, declare the explainers:

[10]:
```
layer = [layer for _, layer in model.named_modules()][-3]

gradShap = GradientShap(model)
intGrad = IntegratedGradients(model)
occlusion = Occlusion(model)
deepLift = DeepLift(model)
guidedBackProp = GuidedBackprop(model)
guidedGradCAM = GuidedGradCam(model, layer)
```

And define a function to obtain the explanations from different methods:

[5]:
```
def get_attributions(data, targets, explainer, params=None):
    """
    :param data: (Tensor) data to explain
    :param targets: (Tensor) class labels w.r.t which we want to compute the attributions
    :param explainer: (captum.attr method) initialised explainer
    :param params: (dict) parameters for the .attribute method of the explainer
    :return: ndarray of shape with attributions
    """

    if params is None:
        params = {}
    elif "baselines" in params and type(params["baselines"]) != int:
        params["baselines"] = params["baselines"].to(device)

    attributions = []
    for image, target in zip(data, targets):
        attr = explainer.attribute(image.unsqueeze(0), target=target, **params).cpu().
↪squeeze().detach().numpy()
        # mean pool channel attributions
        attr = np.mean(attr, axis=0)
        # viz._normalize_image_attr(tmp, 'absolute_value', 10)
        attributions.append(_minmax_normalize_array(attr))

    return np.array(attributions)
```

[ ]:
```
# use predicted labels
obsToExplain = torch.FloatTensor(XTest[:5]).to(device)
expsToCompare = expsTest[:5]
predTargets = F.softmax(model(obsToExplain), dim=-1).argmax(dim=1)[:5].to(device)
z = torch.LongTensor([1 if e == 0 else 0 for e in predTargets])
# z = torch.zeros(len(predTargets), dtype=torch.int)
```

[ ]:
```
# takes some minutes to run
gradShapExpsTest = get_attributions(obsToExplain, predTargets, gradShap, {'baselines':␣
↪torch.zeros((1, 3, imH, imW))})
intGradExpsTest = get_attributions(obsToExplain, predTargets, intGrad)
deepLiftExpsTest = get_attributions(obsToExplain, predTargets, deepLift)
```

(continues on next page)

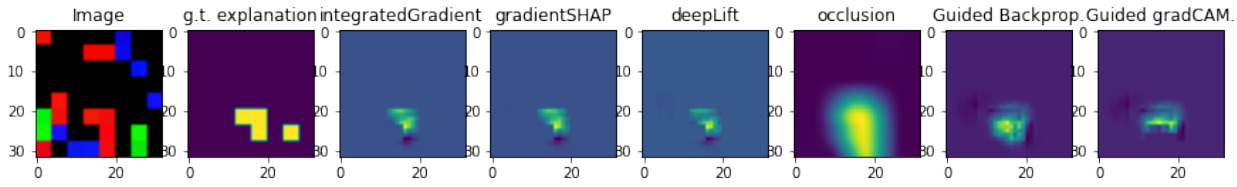
(continued from previous page)

```
occlusionExpsTest = get_attributions(obsToExplain, predTargets, occlusion, {'baselines':␣
↪0, 'sliding_window_shapes': (3, cellH*2, cellW*2)})
gBackPropExpsTest = get_attributions(obsToExplain, z, guidedBackProp)
gGradCAMExpsTest = get_attributions(obsToExplain, z, guidedGradCAM)

with open('expsSynth.pickle', 'wb') as handle:
    pickle.dump([gradShapExpsTest, intGradExpsTest, deepLiftExpsTest, occlusionExpsTest,␣
↪gBackPropExpsTest, gGradCAMExpsTest], handle)
```

[ ]:
```
with open('expsSynth.pickle', 'rb') as handle:
    gradShapExpsTest, intGradExpsTest, deepLiftExpsTest, occlusionExpsTest,\
        gBackPropExpsTest, gGradCAMExpsTest = pickle.load(handle)
```

[ ]:
```
i = 3

fig, axs = plt.subplots(1, 8, figsize=(15,15))
axs[0].imshow(obsToExplain[i].cpu().permute(1, 2, 0))
axs[0].set_title('Image')
axs[1].imshow(expsToCompare[i])
axs[1].set_title('g.t. explanation')
axs[2].imshow(intGradExpsTest[i])
axs[2].set_title('integratedGradient')
axs[3].imshow(gradShapExpsTest[i])
axs[3].set_title('gradientSHAP')
axs[4].imshow(deepLiftExpsTest[i])
axs[4].set_title('deepLift')
axs[5].imshow(occlusionExpsTest[i])
axs[5].set_title('occlusion')
axs[6].imshow(gBackPropExpsTest[i])
axs[6].set_title('Guided Backprop.')
axs[7].imshow(gGradCAMExpsTest[i])
axs[7].set_title('Guided gradCAM.')
```

```
Text(0.5, 1.0, 'Guided gradCAM.')
```




And we can evaluate the explanations. We set the binarization threshold for the generated explanations to 0.5 because we only want high values to count.

[ ]:
```
metrics = ['auc', 'fscore', 'prec', 'rec', 'cs']
gradShapScores = saliency_map_scores(expsTest[yTest == 1], gradShapExpsTest,␣
↪metrics=metrics, binThreshold=0.5)
intGradScores = saliency_map_scores(expsTest[yTest == 1], intGradExpsTest,␣
↪metrics=metrics, binThreshold=0.5)
deepLiftScores = saliency_map_scores(expsTest[yTest == 1], deepLiftExpsTest,␣
↪metrics=metrics, binThreshold=0.5)
occlusionScores = saliency_map_scores(expsTest[yTest == 1], occlusionExpsTest,␣
↪metrics=metrics, binThreshold=0.5)
```

(continues on next page)

(continued from previous page)

```
gBackPropScores = saliency_map_scores(expsTest[yTest == 1], gBackPropExpsTest,␣
↪metrics=metrics, binThreshold=0.5)
gGradCAMScores = saliency_map_scores(expsTest[yTest == 1], gGradCAMExpsTest,␣
↪metrics=metrics, binThreshold=0.5)

scores = pd.DataFrame(data=[gradShapScores, intGradScores, deepLiftScores,
                            occlusionScores, gBackPropScores, gGradCAMScores],␣
↪columns=metrics)
scores['technique'] = ['gradSHAP', 'intGrad', 'deepLift', 'occlusion', 'guidedBackProp',
↪'guidedGradCAM']
scores
```

Note how the warnings tell us that the predictions from one of the techniques did not contain any relevant values.

From here, we can build a more complex explanation evaluation pipeline. Suppose that, given some explainer architecture and model, we want to measure how the influence of some explainer hyperparameters influence the quality of their generated explanations.

[6]:
```
def eval_explainers(
        explainers,
        explainerConfigs,
        data,
        targets,
        trueExps,
        metrics,
        binThreshold: float = .5) -> dict:
    """
    :param explainers: (dict of captum.attr explainers) explainers to use
    :param explainerConfigs: (dict of list of dict) hyperparameter values to use for␣
↪each explainer (see Captum docs)
    :param data: (Tensor) data to explain
    :param targets: (Tensor) labels w.r.t. which we compute the explanations
    :param trueExps: (ndarray) ground truth explanations
    :param metrics: (array-like of str) metrics to compute
    :param float binThreshold: threshold to use when binarizing for the computation of␣
↪classification metrics.
    """
    allScores = {explainer: {met: [] for met in metrics} for explainer in explainers.
↪keys()}
    for explainerName, explainer in explainers.items():
        for config in explainerConfigs[explainerName]:
            exps = get_attributions(data, targets, explainer, config)
            evals = saliency_map_scores(trueExps, exps, metrics=metrics,␣
↪binThreshold=binThreshold)
            for i, score in enumerate(evals):
                allScores[explainerName][metrics[i]].append(score)
    return allScores
```

for example, given these gradSHAP and guidedGradCAM configurations:

[ ]:
```
configs = {
    'gradSHAP': [{'baselines': torch.zeros((1, 3, imH, imW)), 'n_samples': 10, 'stdevs':␣
↪0.1},
```

(continues on next page)

(continued from previous page)

```
                {'baselines': torch.zeros((1, 3, imH, imW)), 'n_samples': 15, 'stdevs':␣
↪0.15}],
    'gradCAM': [{'interpolate_mode': 'nearest'},
                {'interpolate_mode': 'area'}]
}

explainers = {'gradSHAP': gradShap, 'gradCAM': guidedGradCAM}

metrics = ['auc', 'fscore', 'prec', 'rec', 'cs']
```

[ ]:
```
# takes some minutes to run
scores = eval_explainers(explainers, configs, XTest[yTest==1].to(device),␣
↪yTest[yTest==1], expsTest[yTest==1], metrics)

with open('synthScores.pickle', 'wb') as f:
    pickle.dump(scores, f)
```

[ ]:
```
with open('synthScores.pickle', 'rb') as f:
    scores = pickle.load(f)
```

For each explainer and configuration, we have a score:

[ ]:
```
scores
```

```
{'gradSHAP': {'auc': [0.7992577, 0.80201876],
  'fscore': [0.5050217, 0.5090714],
  'prec': [0.9954608, 0.9955106],
  'rec': [0.34479782, 0.3487118],
  'cs': [0.39023715, 0.3968119]},
 'gradCAM': {'auc': [0.34990147, 0.31682178],
  'fscore': [0.1094004, 0.11006599],
  'prec': [0.05815248, 0.058509283],
  'rec': [0.9214127, 0.92628205],
  'cs': [0.23264565, 0.23132235]}}
```

With these metrics, then, we can evaluate the performance of explainers.

### 2. Evaluating Kahikatea image explanations

teex includes real datasets with available ground truth explanations. For example, the Kahikatea dataset contains 519 images, and the task is to tell whether each observation contains Kahikatea trees or not. There are 232 positive observations and 287 negative ones.

In teex, the non-artificial datasets are implemented as classes, similarly to PyTorch. After instancing the class, the data itself will be downloaded if it has not been used before. Once done, one can slice it to obtain observations. Each observation contains the data point, the label and the ground truth explanation.

[7]:
```
from teex.saliencyMap.data import Kahikatea
```

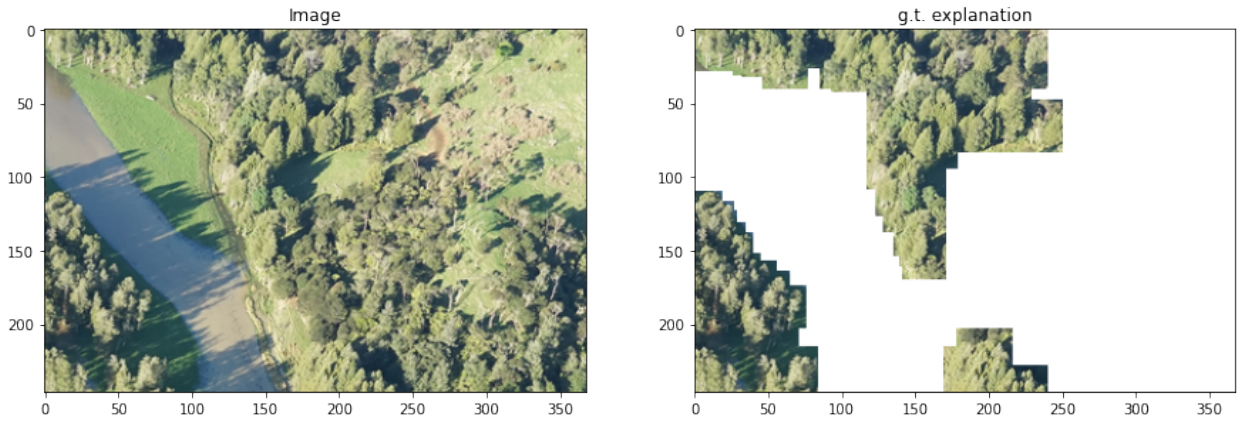
[8]:
```
nClasses = 2
kahikateaData = Kahikatea()
kData, kLabels, kExps = kahikateaData[:]
```

```
Files do not exist or are corrupted:
Downloading https://zenodo.org/record/5059769/files/kahikatea.zip?download=1 to /opt/
↪conda/lib/python3.7/site-packages/teex/_datasets/saliencyMap/kahikatea/rawKahikatea.zip
```

```
142303232it [00:43, 3241716.65it/s]
```

[9]:
```
i = 0
fig, axs = plt.subplots(1, 2, figsize=(15,15))
axs[0].imshow(kData[i])
axs[0].set_title('Image')
axs[1].imshow(kExps[i])
axs[1].set_title('g.t. explanation')
```

[9]:
```
Text(0.5, 1.0, 'g.t. explanation')
```




[10]:
```
kahikateaData.classMap
```

[10]:
```
{0: 'Not in image', 1: 'In image'}
```

Let's fine-tune a pre-trained squeezenet for our particular task.

[34]:
```
torch.hub._validate_not_a_forked_repo=lambda a,b,c: True  #
sqznet = torch.hub.load('pytorch/vision:v0.9.0', 'squeezenet1_0', pretrained=True,␣
↪progress=False)
torch.manual_seed(7)
```

```
Using cache found in /root/.cache/torch/hub/pytorch_vision_v0.9.0
```

[34]:
```
<torch._C.Generator at 0x7fd90d5742b0>
```

And modify its architecture so the shape of the output conforms to our 2-class problem instead of the 1000-class ImageNet.

[35]:
```
sqznet.classifier[1] = nn.Conv2d(512, nClasses, kernel_size=(1,1), stride=(1,1))
sqznet.num_classes = nClasses
inputSize = 224
```

Define the required transform for the input images

[13]:
```
from torchvision import transforms
```

(continues on next page)

(continued from previous page)

```
inputTransform = transforms.Compose([transforms.Resize((inputSize, inputSize)),
                                     transforms.ToTensor(),
                                     transforms.Normalize([0.485, 0.456, 0.406], [0.229,␣
↪0.224, 0.225])])
resizeTransform = transforms.Compose([transforms.Resize((inputSize, inputSize)),
                                     transforms.ToTensor()])
```

Transform the data to torch tensors and create the splits:

[14]:
```
kData = torch.stack([inputTransform(img) for img in kData])
kExps = torch.stack([resizeTransform(img) if isinstance(img, PIL.Image.Image) else torch.
↪zeros((3, inputSize, inputSize)) for img in kExps]).permute(0, 2, 3, 1)
kExps = kExps.numpy().astype(np.float32)  # we need g.t. explanations to be numpy arrays␣
↪for the evaluation
kLabels = torch.LongTensor(kLabels)

kTrain, kTest, kTrainLabels, kTestLabels, kExpsTrain, kExpsTest = train_test_split(kData,
↪ kLabels, kExps, train_size=0.8, random_state=7)
kTrain, kVal, kTrainLabels, kValLabels, kExpsTrain, kExpsVal = train_test_split(kTrain,␣
↪kTrainLabels, kExpsTrain, train_size=0.75, random_state=7)
```

[15]:
```
print(f'Data proportions -> Train: {round(len(kTrain) / len(kData), 3)}, Val:
↪{round(len(kVal) / len(kData), 3)}, Test: {round(len(kTest) / len(kData), 3)}')
print(f'Positive label proportions -> Train: {round((sum(kTrainLabels) /␣
↪len(kTrainLabels)).item(), 3)}, \
Val: {round((sum(kValLabels) / len(kValLabels)).item(), 3)}, Test:
↪{round((sum(kTestLabels) / len(kTestLabels)).item(), 3)}')
```

```
Data proportions -> Train: 0.599, Val: 0.2, Test: 0.2
Positive label proportions -> Train: 0.444, Val: 0.519, Test: 0.385
```

[ ]:
```
opt = optim.SGD(sqznet.parameters(), lr=1e-3)
crit = nn.CrossEntropyLoss()
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
batchSize = 10
nEpochs = 25
data = {'train': [kTrain.to(device), kTrainLabels.to(device)], 'val': [kVal.to(device),␣
↪kValLabels.to(device)]}

sqznet.to(device)

sqznet, valAcc = train_net(sqznet, data, crit, opt, device, batchSize, nEpochs,␣
↪randomState=7)

with open('sqznetTrained.pickle', 'wb') as f:
    pickle.dump(sqznet.state_dict(), f)
```

[38]:
```
with open('sqznetTrained.pickle', 'rb') as f:
    sqznet.load_state_dict(pickle.load(f))
```

[37]:
```
print(f'Validation F1: {round(f1_score(kValLabels, F.softmax(sqznet(kVal.to(device)),␣
↪dim=-1).cpu().argmax(dim=1).detach().numpy()), 3)}')
```

(continues on next page)

(continued from previous page)

```
print(f'Test F1: {round(f1_score(kTestLabels, F.softmax(sqznet(kTest.to(device)), dim=-
↪1).cpu().argmax(dim=1).detach().numpy()), 3)}')
```

```
Validation F1: 0.816
Test F1: 0.643
```

Let us get some explanations as samples

[77]:
```
back_hook = 'register_full_backward_hook' if torch.__version__ >= '1.8.0' else 'register_
↪backward_hook'

gradShap = GradientShap(sqznet)
intGrad = IntegratedGradients(sqznet)
occlusion = Occlusion(sqznet)
deepLift = DeepLift(sqznet)
guidedBackProp = GuidedBackprop(sqznet)
guidedGradCAM = GuidedGradCam(sqznet, sqznet.features[12])

# use predicted labels
obsToExplain = torch.FloatTensor(kTest[kTestLabels == 1][:2]).to(device)
expsToCompare = kExpsTest[kTestLabels == 1][:2]
predTargets = F.softmax(sqznet(obsToExplain), dim=-1).argmax(dim=1)

# takes some minutes to run
gradShapExpsTest = get_attributions(obsToExplain, predTargets, gradShap, {'baselines':␣
↪torch.zeros((1, 3, inputSize, inputSize))})
intGradExpsTest = get_attributions(obsToExplain, predTargets, intGrad)
deepLiftExpsTest = get_attributions(obsToExplain, predTargets, deepLift)
occlusionExpsTest = get_attributions(obsToExplain, predTargets, occlusion, {'baselines':␣
↪0, 'sliding_window_shapes': (3, inputSize, round(inputSize))})
gBackPropExpsTest = get_attributions(obsToExplain, predTargets, guidedBackProp)
gGradCAMExpsTest = get_attributions(obsToExplain, predTargets, guidedGradCAM)
```

```
/opt/conda/lib/python3.7/site-packages/captum/attr/_core/guided_backprop_deconvnet.py:65:
↪ UserWarning: Setting backward hooks on ReLU activations.The hooks will be removed␣
↪after the attribution is finished
  "Setting backward hooks on ReLU activations."
```

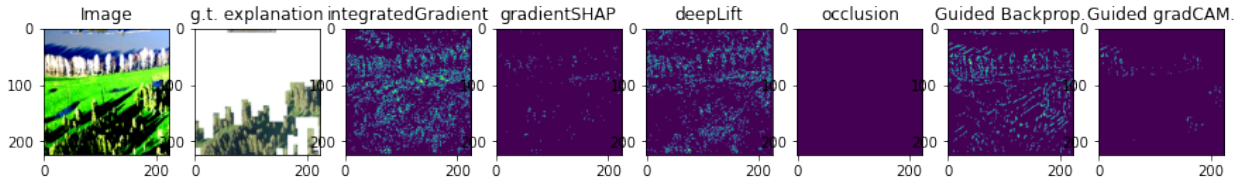
[43]:
```
# we binarize the results for easier visualization (this step is implicitly done by teex)
#   when computing metrics.
from teex._utils._arrays import _binarize_arrays

gsh = _binarize_arrays(gradShapExpsTest, method='abs', threshold=0.55)
ig = _binarize_arrays(intGradExpsTest, method='abs', threshold=0.55)
dl = _binarize_arrays(deepLiftExpsTest, method='abs', threshold=0.55)
oc = _binarize_arrays(occlusionExpsTest, method='abs', threshold=0.55)
gbp = _binarize_arrays(gBackPropExpsTest, method='abs', threshold=0.55)
ggc = _binarize_arrays(gGradCAMExpsTest, method='abs', threshold=0.55)

i = 0

fig, axs = plt.subplots(1, 8, figsize=(15,15))
axs[0].imshow(obsToExplain[i].cpu().permute(1, 2, 0))
```

(continues on next page)

(continued from previous page)

```
axs[0].set_title('Image')
axs[1].imshow(expsToCompare[i])
axs[1].set_title('g.t. explanation')
axs[2].imshow(ig[i])
axs[2].set_title('integratedGradient')
axs[3].imshow(gsh[i])
axs[3].set_title('gradientSHAP')
axs[4].imshow(dl[i])
axs[4].set_title('deepLift')
axs[5].imshow(oc[i])
axs[5].set_title('occlusion')
axs[6].imshow(gbp[i])
axs[6].set_title('Guided Backprop.')
axs[7].imshow(ggc[i])
axs[7].set_title('Guided gradCAM.')
```

[43]:
```
Text(0.5, 1.0, 'Guided gradCAM.')
```




From this visualization, it is clear that the threshold level is an important hyperparameter to consider.

Now, let's evaluate the quality of the explanations:

[66]:
```
kExplainers = {'gradSHAP': GradientShap(sqznet),
               'gradCAM': GuidedGradCam(sqznet, sqznet.features[12]),
               'deepLift': DeepLift(sqznet),
               'backProp': GuidedBackprop(sqznet),
               'occlusion': Occlusion(sqznet),
               'intGrad': IntegratedGradients(sqznet)}

kConfigs = {
    'gradSHAP': [{'baselines': torch.zeros((1, 3, inputSize, inputSize)), 'n_samples': 5,
↪ 'stdevs': 0.1}],
    'gradCAM': [{'interpolate_mode': 'nearest'}],
    'deepLift': [{}],
    'backProp': [{}],
    'occlusion': [{'baselines': 0, 'sliding_window_shapes': (3, round(inputSize),␣
↪round(inputSize))}],
    'intGrad': [{'method': 'riemann_trapezoid'}]
}

metrics = ['auc', 'fscore', 'prec', 'rec', 'cs']

binThresholds = [e / 100 for e in range(10, 70, 5)]
```

Evaluate positive test explanations. Note that teex implicitly handles the conversion of the RGB masks into 0-1 normalised grayscale masks (the shape of the g.t.s need to be (imH, imW, 3) for it to happen).

[67]:
```
# use predicted labels
obsToExplain = torch.FloatTensor(kTest[kTestLabels == 1]).to(device)
expsToCompare = kExpsTest[kTestLabels == 1]
predTargets = F.softmax(sqznet(obsToExplain), dim=-1).argmax(dim=1)
```

[ ]:
```
# comparison to ground truth explanations
resGT = {}
for binThres in binThresholds:
    # takes a few minutes to run
    scoresK = eval_explainers(kExplainers, kConfigs, obsToExplain, predTargets,
                              expsToCompare, metrics, binThreshold=binThres)
    resGT[f"{binThres}"] = scoresK
    # fileName = f'kahikateaScoresThres{binThres}.pickle'
    # with open(fileName, 'wb') as f:
    #     pickle.dump(scoresK, f)

with open("kahikateaScores", 'wb') as f:
    pickle.dump(resGT, f)
```

[ ]:
```
resGT
```

[ ]:
```
# comparison to integrated gradients explanations
obsToExplain = torch.FloatTensor(kTest[kTestLabels == 1]).to(device)
predTargets = F.softmax(sqznet(obsToExplain), dim=-1).argmax(dim=1)

expsToCompare = get_attributions(obsToExplain, predTargets, intGrad)

resIntGrad = {}
for binThres in binThresholds:
    # takes a few minutes to run
    scoresK = eval_explainers(kExplainers, kConfigs, obsToExplain, predTargets,
                              expsToCompare, metrics, binThreshold=binThres)
    resIntGrad[f"{binThres}"] = scoresK

with open("kahikateaScores", 'wb') as f:
    pickle.dump(resIntGrad, f)
```

[ ]:
```
resIntGrad
```

### 2.1.2 Decision rule

Here you will find demos for the *decision rule* explanation type.

#### Generating data with available g.t. decision rule explanations

We are going to see the available options for data generation with g.t. decision rule explanations and related methods.

[68]:
```
from teex.decisionRule.data import Statement, DecisionRule, SenecaDR, str_to_decision_
↪rule, rulefit_to_decision_rule

from rulefit import RuleFit
```

#### 1. DecisionRule objects in teex

To represent decision rules, *teex* provides a custom class. In short, we consider the atomic structure of a rule, a `Statement`, that represents an 'if' clause. Then, a `DecisionRule` object is comprised of a collection of `Statement` objects, which, if all held true, imply a result, also represented as a `Statement`.

For example, given the Statements:

- 'white_feathers' == true
- 'quacks' == true

we can build the decision rule that says:

- **if** (`white_feathers` == true) **and** (`quacks` == true) **then** (`is_duck` == true)

In code, we can build this exact example:

[69]:
```
s1 = Statement('white_feathers', True)
s2 = Statement('quacks', True)
s3 = Statement('is_duck', True)

dr = DecisionRule([s1, s2], s3)
print(dr)
```

```
IF 'white_feathers' = True, 'quacks' = True THEN 'is_duck' = True
```

or, more human-like:

[70]:
```
strRule = 'white_feathers = True & quacks = True -> is_duck = True'
dr = str_to_decision_rule(strRule, ruleType='unary')

print(repr(dr), '\n', dr)
```

```
<teex.decisionRule.data.DecisionRule object at 0x128cb9970>
 IF 'white_feathers' = True, 'quacks' = True THEN 'is_duck' = True
```

Statements are flexible and can represent multiple operators (`{'=', '!=', '>', '<', '>=', '<='}`) and be binary for numeric features (`0.5 < feature < 1`, for example). Both **teex** and the methods themselves provide methods for easy manipulation of `Statement` and `DecisionRule` objects, such as insertion, deletion or upsertion of new statements into a decision rule object. We urge the keen user to take a look at the API documentation for more on this.

The `DecisionRule` class provides a unified way of dealing with this kind of data, which allows for easier implementation of related methods, be it data generation or evaluation. So, all DecisionRule metrics work only with DecisionRule

objects. Not to worry, because **teex** provides methods for transforming from common decision rule representations to DecisionRule objects.

### 2. Generating artificial data with SenecaDR

**note** This method in particular was not originally conceived as a data generation procedure, but rather as a way to generate transparent classifiers (i.e. a classifier with available ground truth explanations). We use that generated classifier and some artificially generated data to return a dataset with observations, labels and ground truth explanations. The dataset generated contains numerical features with a binary classification.

As with all data generation procedures in **teex**, first an object needs to be instanced and then the data can be retrieved. We can adjust the number of samples we want, the number of features in the observations, the feature names and the random seed.

[71]:
```
dataGen = SenecaDR(nSamples=1000, nFeatures=3)
X, y, exps = dataGen[:]

print(f'Observation: {X[0]} \nLabel: {y[0]} \nExplanation: {exps[0]}')
```

```
Observation: [1.25824083 1.37756901 0.4123272 ]
Label: 0
Explanation: IF 0.111 < 'c', -0.015 < 'a', 0.901 < 'b' <= 2.31 THEN 'Class' = 0
```

[72]:
```
dataGen.featureNames
```

[72]:
```
['a', 'b', 'c']
```

See how the explanations generated are actually `DecisionRule` objects, with Statements for each class (not in all cases, though).

[73]:
```
exps[:5]
```

[73]:
```
[<teex.decisionRule.data.DecisionRule at 0x128cc7940>,
 <teex.decisionRule.data.DecisionRule at 0x128cc7ac0>,
 <teex.decisionRule.data.DecisionRule at 0x128cc7be0>,
 <teex.decisionRule.data.DecisionRule at 0x128cc7d00>,
 <teex.decisionRule.data.DecisionRule at 0x128cc7e20>]
```

See how the explanations generated are actually `DecisionRule` objects, with Statements for each class (not in all cases, though). Note that we can also specify the feature names instead of letting them be automatically generated. As with all of **teex**'s `Seneca` methods, the underlying data generated procedure is carried out by a transparent model that follows the `sklearn` API (has `.predict`, `.predict_proba` and `.fit` methods). In this case, the model is a Decision Tree classifier, and the explanations are the decision paths that the trained model takes when performing predictions. We believe this class can be of utility to users for easily extracting explanations.

[74]:
```
from teex.decisionRule.data import TransparentRuleClassifier

model = TransparentRuleClassifier()

# it can fit any binary classification data, not just this example
model.fit(X, y, featureNames=['f1', 'f2', 'f3'])
```

[75]:
```
print(model.predict(X[:5]))
```

```
[0 1 1 1 1]
```

[76]:
```
model.predict_proba(X[:5])
```

[76]:
```
array([[1., 0.],
       [0., 1.],
       [0., 1.],
       [0., 1.],
       [0., 1.]])
```

[77]:
```
model.explain(X[:5])
```

[77]:
```
[<teex.decisionRule.data.DecisionRule at 0x128cb90a0>,
 <teex.decisionRule.data.DecisionRule at 0x128cb92e0>,
 <teex.decisionRule.data.DecisionRule at 0x128cb9ee0>,
 <teex.decisionRule.data.DecisionRule at 0x128eaf9a0>,
 <teex.decisionRule.data.DecisionRule at 0x128eaffa0>]
```

[78]:
```
for dr in model.explain(X[:5]):
    print(dr)
```

```
IF 0.111 < 'f3', -0.015 < 'f1', 0.901 < 'f2' <= 2.31 THEN 'Class' = 0
IF 'f3' <= -0.324, 0.672 < 'f1', 'f2' <= -0.37 THEN 'Class' = 1
IF 'f3' <= -0.324, 0.672 < 'f1', 'f2' <= -0.37 THEN 'Class' = 1
IF -1.705 < 'f3' <= 0.111, 'f1' <= -0.041, 0.428 < 'f2' <= 0.63 THEN 'Class' = 1
IF -1.705 < 'f3' <= 0.111, 'f1' <= -0.041, 0.635 < 'f2' THEN 'Class' = 1
```

For more information on the transparent model, please see the notebook on Feature Importance data generation or visit **teex**'s API documentation.

### 3. Transforming common representations into `DecisionRule` objects

If we want to evaluate common decision rule explanation methods and the evaluation methods in **teex** work only with `DecisionRule` objects, we need methods for transforming representations. We have seen how we can transform string representations with `str_to_decision_rule`, but another useful method is `rulefit_to_decision_rule`. It transforms the rules computed by the RuleFit algorithm:

[79]:
```
# first, find some data
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
```

[99]:
```
# instance a rule fit object and get explanations
rf = RuleFit()
rf.fit(X, y, feature_names=features)
```

```
/Users/master/Google Drive/U/4t/TFG/teex/venv/lib/python3.8/site-packages/sklearn/linear_
↪model/_coordinate_descent.py:530: ConvergenceWarning: Objective did not converge. You␣
↪might want to increase the number of iterations. Duality gap: 2.20433295631139,␣
↪tolerance: 2.1169160949554895
  model = cd_fast.enet_coordinate_descent(
```

(continues on next page)

(continued from previous page)

```
/Users/master/Google Drive/U/4t/TFG/teex/venv/lib/python3.8/site-packages/sklearn/linear_
↪model/_coordinate_descent.py:530: ConvergenceWarning: Objective did not converge. You␣
↪might want to increase the number of iterations. Duality gap: 2.268052878131016,␣
↪tolerance: 2.1169160949554895
  model = cd_fast.enet_coordinate_descent(
```

[99]:
```
RuleFit(tree_generator=GradientBoostingRegressor(learning_rate=0.01,
                                                 max_depth=100,
                                                 max_leaf_nodes=5,
                                                 n_estimators=560,
                                                 random_state=559,
                                                 subsample=0.46436099318265595))
```

The rules from RuleFit can be extracted from here:

[102]:
```
rf.get_rules()
```

[102]:
```
                                                  rule    type      coef  \
0                                                 crim  linear -0.000000
1                                                   zn  linear  0.002153
2                                                indus  linear -0.000000
3                                                 chas  linear  0.000000
4                                                  nox  linear -0.000000
...                                                ...     ...       ...
1720         ptratio <= 18.75 & rm <= 7.452499866485596    rule -0.000000
1721                           dis > 6.341400146484375    rule -0.000000
1722  lstat > 5.184999942779541 & ptratio > 13.84999...    rule -0.000000
1723                                     tax <= 298.0    rule  0.000000
1724                          crim > 18.737899780273438    rule -0.000000

       support  importance
0     1.000000    0.000000
1     1.000000    0.048604
2     1.000000    0.000000
3     1.000000    0.000000
4     1.000000    0.000000
...        ...         ...
1720  0.401709    0.000000
1721  0.145299    0.000000
1722  0.829060    0.000000
1723  0.333333    0.000000
1724  0.029915    0.000000

[1725 rows x 5 columns]
```

and we can convert them into `DecisionRule` objects with a single line. Note that only the rules are transform, not the base coefficients (**type** = linear). Our method also provides parameters for the minimum support and importance for a rule to be transformed.

[103]:
```
# and transform into decision rule objects
dRules, skippedRows = rulefit_to_decision_rule(rules)
```

[104]:
```
dRules[:5]
```

[104]:
```
[<teex.decisionRule.data.DecisionRule at 0x12efcbfa0>,
 <teex.decisionRule.data.DecisionRule at 0x12ef1b100>,
 <teex.decisionRule.data.DecisionRule at 0x12a32ad00>,
 <teex.decisionRule.data.DecisionRule at 0x12a32a970>,
 <teex.decisionRule.data.DecisionRule at 0x12a32a940>]
```

[105]:
```
for rule in dRules[:5]:
    print(rule)
```

```
IF 'nox' <= 0.6694999933242798, 'dis' <= 1.3980499505996704 THEN None
IF 'ptratio' <= 18.65000057220459, 7.423499822616577 < 'rm' THEN None
IF 1.1736000180244446 < 'dis', 21.489999771118164 < 'lstat', 'rm' <= 7.423500061035156␣
↪THEN None
IF 7.433000087738037 < 'rm', 'lstat' <= 14.805000305175781 THEN None
IF 20.19499969482422 < 'lstat' THEN None
```

### Evaluation of explanation quality: decision rules

In this notebook, we are going to explore how we can use **teex** to evaluate decision rule explanations

[14]:
```
from teex.decisionRule.data import SenecaDR
from teex.decisionRule.eval import rule_scores

# these three imports fix an issue with imports in SkopeRules
import six
import sys
sys.modules['sklearn.externals.six'] = six

from skrules import SkopeRules
```

The first step is to gather data with available ground truth decison rule explanations. **teex** makes it simples:

[33]:
```
dataGen = SenecaDR(nSamples=500, nFeatures=5, randomState=88)
X, y, exps = dataGen[:]
```

[34]:
```
X[:5]
```

[34]:
```
array([[-2.02711717, -1.32958987, -0.77103092,  0.99843625, -2.27314715],
       [-0.64076447,  1.62339205,  1.75445611, -1.00969545,  1.83765661],
       [ 1.50354713, -1.27483644, -2.19842768, -1.05181378,  1.07449273],
       [ 0.06917376, -0.45268848, -1.05498443,  0.00318232, -0.65430449],
       [ 1.04850317,  2.69542922,  2.05851293, -0.06200245, -1.50837284]])
```

[35]:
```
y[:5]
```

[35]:
```
array([1, 1, 0, 1, 0])
```

[36]:
```
for e in exps[:5]:
    print(e)
```

```
IF 'a' <= -0.648, 'e' <= 0.125, 'c' <= -0.638, -1.473 < 'd' THEN 'Class' = 1
IF 'a' <= 0.962, 0.278 < 'e', -1.018 < 'b', -1.441 < 'c', 'd' <= 1.025 THEN 'Class' = 1
IF 0.962 < 'a', -2.876 < 'b' <= -0.656, 'd' <= -0.766, 'c' <= -2.147, -0.739 < 'e' THEN
↪'Class' = 0
```

(continues on next page)

(continued from previous page)

```
IF -0.467 < 'a' <= 0.962, 'e' <= 0.125, 'c' <= -0.638, -1.473 < 'd' THEN 'Class' = 1
IF 0.962 < 'a', -0.095 < 'b', -1.843 < 'd', -2.64 < 'e' THEN 'Class' = 0
```

The second step is training an estimator and predicting explanations. We could use any system for training and generating the explanations. We are going to skip this step, as its independent to **teex** and up to the user to decide in which way to generate the explanations. Instead, we are going to use the ground truth explanations as if they were the predicted ones.

So, we compare the predicted explanations with the ground truth ones.

[39]:
```
metrics = ['crq', 'prec', 'rec', 'fscore']
scores = rule_scores(exps, exps, dataGen.featureNames, metrics)
```

```
/usr/local/lib/python3.8/site-packages/teex/featureImportance/eval.py:77: UserWarning: A␣
↪binary ground truth contains uniform values, so one entry has been randomly flipped␣
↪for the metrics to be defined.
  warnings.warn('A binary ground truth contains uniform values, so one entry has been␣
↪randomly flipped '
/usr/local/lib/python3.8/site-packages/teex/featureImportance/eval.py:80: UserWarning: A␣
↪binary prediction contains uniform values, so one entry has been randomly flipped for␣
↪the metrics to be defined.
  warnings.warn('A binary prediction contains uniform values, so one entry has been␣
↪randomly flipped '
```

[40]:
```
for i, metric in enumerate(metrics):
    print(f'{metric}: {scores[i]}')
```

```
crq: 1.0
prec: 1.0
rec: 1.0
fscore: 1.0
```

We obtain perfect scores, as the ground truths are exactly the same as the predictions.

### 2.1.3 Feature importance

Here you will find demos for the *feature importance* explanation type.

#### Generating data with available g.t. feature importance explanations

We are going to see the available options for data generation with g.t. feature importance explanations.

#### 1. Generating artificial data with SenecaFI

[1]:
```
from teex.featureImportance.data import SenecaFI
```

We are going to explore `SenecaFI`, a method from [Evaluating local explanation methods on ground truth, Riccardo Guidotti, 2021].

**note** This method was not originally conceived as a data generation procedure, but rather as a way to generate transparent classifiers (i.e. a classifier with available ground truth explanations). We use that generated classifier and some

artificially generated data to return a dataset with observations, labels and ground truth explanations. The dataset generated contains numerical features with a binary classification.

[4]:
```
# instance the data generator
dataGen = SenecaFI(nSamples=100, nFeatures=4, randomState=1)

# retrieve the generated observations
X, y, exps = dataGen[:]
```

[5]:
```
print(f'Observation: {X[0]} \nLabel: {y[0]} \nExplanation: {exps[0]}')
```

```
Observation: [ 0.34558419 -0.65128101  1.82843024 -0.59277453]
Label: 1
Explanation: [ 1.      1.      0.1897 -0.3012]
```

The ground truth FI explanations are scales to the range (-1, 1) by feature. That is, if a feature contains a 1 in a particular observation, that means that it is the observation where that feature is most important in the dataset. Inversely, if an observation contains a -1, it means that the specific feature contributes the most negatively in the dataset.

One can specify the number of points to be generated (`nSamples`), the number of features (`nFeatures`), the names of the features (`featureNames`) and the random state (`randomState`).

[6]:
```
dataGen.featureNames  # automatically generated
```

[6]:
```
['a', 'b', 'c', 'd']
```

The explanations are generated by first creating a random collection of points. Then, creating a random linear expression and finally evaluating its derivative at the points closest to the original observations. The underlying model can be accessed:

[7]:
```
model = dataGen.transparentModel
model
```

[7]:
```
<teex.featureImportance.data.TransparentLinearClassifier at 0x12bfd0be0>
```

This structure follows the sklearn API (`.fit`, `.predict`, `.predict_proba`) and can be used to test explainer methods, for example. An important method that it contains is the `.explain`, which given an observation, explains the prediction. All of the observations that the object receive must be of shape (nObservations, nFeatures).

Compute predictions:

[8]:
```
print(f'Single observation: {model.predict(X[0].reshape(1, -1))} \nMultiple observations:
↪ {model.predict(X[:10])}')
```

```
Single observation: [1]
Multiple observations: [1 1 1 0 0 1 0 1 0 0]
```

Compute class probabilities:

[9]:
```
print(f'Single observation: \n{model.predict_proba(X[0].reshape(1, -1))} \n\nMultiple␣
↪observations: \n{model.predict_proba(X[:10])}')
```

```
Single observation:
[[0. 1.]]

Multiple observations:
[[0.        1.       ]
```

(continues on next page)

(continued from previous page)

```
 [0.05753863 0.94246137]
 [0.25390828 0.74609172]
 [1.         0.        ]
 [0.80108987 0.19891013]
 [0.         1.        ]
 [1.         0.        ]
 [0.         1.        ]
 [1.         0.        ]
 [1.         0.        ]]
```

Compute explanations:

```
[10]: print(f'Single observation: \n{model.explain(X[0].reshape(1, -1))} \n\nMultiple␣
      ↪observations: \n{model.explain(X[:10])}')
```

```
Single observation:
[[1. 1. 1. 1.]]

Multiple observations:
[[ 1.      1.      1.     -1.    ]
 [ 1.      1.      0.0879 -0.4518]
 [ 1.      1.      0.0706 -0.6003]
 [ 1.      1.      0.2038 -0.237 ]
 [ 1.      1.      0.1008 -0.7226]
 [ 1.      1.     -1.      1.    ]
 [ 1.      1.      0.1038 -0.6127]
 [ 1.      1.      0.1553 -0.4214]
 [ 1.      1.      0.1394 -0.7511]
 [ 1.      1.     -0.3793 -0.749 ]]
```

Note that the scaler will work with the observations that it is explaining.

### Evaluation of explanation quality: feature importance vectors

In this notebook, we are going to explore how we can use **teex** to evaluate feature importance explanations

```
[1]: from teex.featureImportance.data import SenecaFI, lime_to_feature_importance, scale_fi_
     ↪bounds
     from teex.featureImportance.eval import feature_importance_scores

     from lime.lime_tabular import LimeTabularExplainer

     from sklearn.ensemble import RandomForestClassifier
     from sklearn.metrics import fbeta_score

     import numpy as np
```

We are going to

1. Generate synthetic data with feature importance explanations
2. Train a black box model on the data
3. Generate LIME explanations of the data

4. Evaluate the LIME explanations agains the ground truth

### 1. Generating the data

[8]:
```
dataGen = SenecaFI(nSamples=300, nFeatures=4, randomState=0)

X, y, exps = dataGen[:]
```

[9]:
```
exps[0]
```

[9]:
```
array([0.9735, 0.2541, 0.8937, 0.4596], dtype=float32)
```

### 2. Training a black box model on the data

[10]:
```
# split data
Xtr, Xte = X[:200], X[200:]
ytr, yte = y[:200], y[200:]
etr, ete = exps[:200], exps[200:]
```

[11]:
```
model = RandomForestClassifier()
model.fit(Xtr, ytr)
```

[11]:
```
RandomForestClassifier()
```

[12]:
```
# F1 Score
print('Train F1: ', fbeta_score(model.predict(Xtr), ytr, beta=1))
print('Test F1: ', fbeta_score(model.predict(Xte), yte, beta=1))
```

```
Train F1:  1.0
Test F1:  0.8378378378378379
```

The classifier fits the data quite nicely.

### 3. Generating LIME explanations

We first instance the explainer

[13]:
```
explainer = LimeTabularExplainer(Xtr, feature_names=dataGen.featureNames, mode=
↪'classification')

# sample explanation
explainer.explain_instance(Xte[0], model.predict_proba).show_in_notebook()
```

```
<IPython.core.display.HTML object>
```

and generate the explanations for the test set. Unfortunately, with LIME we have to generate explanations 1 by 1. We use a function for transforming LIME explanations into feature importance vectors. This function does not transform the data.

[14]:
```
limeExps = []

# takes a few moments to run
for testObs in Xte:
    exp = explainer.explain_instance(testObs, model.predict_proba)
    limeExps.append(lime_to_feature_importance(exp, nFeatures=4))

limeExps = np.array(limeExps)
```

[15]:
```
limeExps[:5]
```

[15]:
```
array([[-0.07230919, -0.0801119 ,  0.09253925,  0.10824497],
       [-0.18176845, -0.03213522, -0.01890396, -0.13922554],
       [-0.07199723,  0.03484774, -0.06955719, -0.11750789],
       [ 0.12277604,  0.02374334, -0.06775021,  0.08342091],
       [ 0.10764252, -0.07811511, -0.01083719,  0.09821577]])
```

### 4. Evaluating LIME explanations

Now that the explanations are computed, we can evaluate them agains the ground truth explanations. See how LIME explanations are not bounded in the (-1, 1) range, so comparison would not valid. For this reason, **teex**'s FI evaluation method scales each feature to the (-1, 1) or (0, 1) range if they are not already in the range.

[16]:
```
metrics = ['fscore', 'cs', 'auc', 'prec', 'rec']
avgMetrics = feature_importance_scores(ete, limeExps, metrics)
```

```
/Users/master/Google Drive/U/4t/TFG/teex/venv/lib/python3.8/site-packages/teex/
↪featureImportance/eval.py:80: UserWarning: A binary prediction contains uniform values,
↪ so one entry has been randomly flipped for the metrics to be defined.
  warnings.warn('A binary prediction contains uniform values, so one entry has been␣
↪randomly flipped '
```

[17]:
```
avgMetrics
```

[17]:
```
array([0.34333327, 0.50356585, 0.51      , 0.58      , 0.24666668],
      dtype=float32)
```

We can also not average the metrics across the observations:

[19]:
```
allMetrics = feature_importance_scores(ete, limeExps, metrics, average=False)
```

```
/Users/master/Google Drive/U/4t/TFG/teex/venv/lib/python3.8/site-packages/teex/
↪featureImportance/eval.py:80: UserWarning: A binary prediction contains uniform values,
↪ so one entry has been randomly flipped for the metrics to be defined.
  warnings.warn('A binary prediction contains uniform values, so one entry has been␣
↪randomly flipped '
```

[20]:
```
allMetrics[:5]
```

[20]:
```
array([[0.6666667 , 0.34652364, 0.5       , 1.        , 0.5       ],
       [0.        , 0.7850784 , 0.5       , 0.        , 0.        ],
       [0.        , 0.85806304, 1.        , 0.        , 0.        ],
       [0.6666667 , 0.44639328, 0.5       , 1.        , 0.5       ],
```

(continues on next page)

(continued from previous page)

```
       [0.        , 0.37971714, 0.33333334, 0.        , 0.        ]],
      dtype=float32)
```

### 2.1.4 Word importance

Here you will find demos for the *word importance* explanation type.

#### Generating data with available g.t. word importance explanations

We are going to see an example of data generation with g.t. word importance explanations.

[25]:
```
from teex.wordImportance.data import Newsgroup

import numpy as np
```

Word importance representations in **teex** are presented as dictionaries. These dictionaries contain as keys all of the words (or at least the relevant ones / the ones that have been scored) in a text, and as values the scores. Let's see an example:

[4]:
```
dataGen = Newsgroup()
X, y, exps = dataGen[:]
```

The Newsgroup dataset contains texts from emails that correspond to either a medical or an electronic class:

[5]:
```
dataGen.classMap
```

[5]:
```
{0: 'electronics', 1: 'medicine'}
```

Each text is represented as a string:

[9]:
```
X[3]
```

[9]:
```
b"From: cfb@fc.hp.com (Charlie Brett)\nSubject: Re: Hi Volt from battery\nNntp-Posting-
↪Host: hpfcmgw.fc.hp.com\nOrganization: Hewlett-Packard Fort Collins Site\nX-Newsreader:
↪ TIN [version 1.1 PL8.5]\nLines: 7\n\nYou might want to get a disposible flash camera,␣
↪shoot the roll of film,\nthen take it apart (they're snapped together). We used a␣
↪bunch of them\nat my wedding, but instead of sending the whole camera in, I just took\
↪nthe film out (it's a standard 35mm canister), and kept the batteries\n(they use one␣
↪AA battery). Sorry, I didn't keep any of the flash electronics.\n\n          Charlie␣
↪Brett - Ft. Collins, CO\n"
```

corresponds to a specific class:

[10]:
```
dataGen.classMap[y[3]]
```

[10]:
```
'electronics'
```

and has a ground truth explanation with the format explained above:

[11]:
```
exps[3]
```

[11]:
```
{'volt': 1.0,
 'battery': 1.0,
 'batteries': 0.5,
 'electronics': 1.0,
 'flash': 0.5}
```

In this instance, the words in the explanation are the ones that characterize the text as pertaining to the "electronics" class. A medical example could be:

[38]:
```
X[23]
```

[38]:
```
b"From: oldman@coos.dartmouth.edu (Prakash Das)\nSubject: Re: Is MSG sensitivity␣
↪superstition?\nArticle-I.D.: dartvax.C60KrL.59t\nOrganization: Dartmouth College,␣
↪Hanover, NH\nLines: 19\n\nIn article <1993Apr20.173019.11903@llyene.jpl.nasa.gov>␣
↪julie@eddie.jpl.nasa.gov (Julie Kangas) writes:\n>\n>As for how foods taste:  If I'm␣
↪not allergic to MSG and I like\n>the taste of it, why shouldn't I use it?  Saying I␣
↪shouldn't use\n>it is like saying I shouldn't eat spicy food because my neighbor\n>has␣
↪an ulcer.\n\nJulie, it doesn't necessarily follow that you should use it (MSG or\
↪nsomething else for that matter) simply because you are not allergic\nto it. For␣
↪example you might not be allergic to (animal) fats, and\nlike their taste, yet it doesn
↪'t follow that you should be using them\n(regularly). MSG might have other bad (or␣
↪good, I am not up on \nknowledge of MSG) effects on your body in the long run, maybe␣
↪that's\nreason enough not to use it. \n\nAltho' your example of the ulcer is funny, it␣
↪isn't an\nappropriate comparison at all.\n\n-Prakash Das\n"
```

[39]:
```
dataGen.classMap[y[23]]
```

[39]:
```
'medicine'
```

[40]:
```
exps[23]
```

[40]:
```
{'msg': 0.5, 'ulcer': 0.5, 'allergic': 1.0, 'sensitivity': 0.5}
```

### Evaluation of explanation quality: word importance vectors

In this notebook, we are going to explore how we can use **teex** to evaluate word importance explanations

[12]:
```
from teex.wordImportance.data import Newsgroup
from teex.wordImportance.eval import word_importance_scores
```

The first step is to gather data with available word importance explanations. **teex** makes it simples:

[13]:
```
dataGen = Newsgroup()
X, y, exps = dataGen[:]
```

[14]:
```
X[1]
```

[14]:
```
b'From: aj2a@galen.med.Virginia.EDU (Amir Anthony Jazaeri)\nSubject: Re: Heat Shock␣
↪Proteins\nOrganization: University of Virginia\nLines: 8\n\nby the way ms. olmstead␣
↪dna is not degraded in the stomach nor\nunder pH of 2.  its degraded in the duodenum␣
↪under approx.\nneutral pH by DNAase enzymes secreted by the pancreas.  my\npoint: ␣
↪check your facts before yelling at other people for not\ndoing so.  just a friendly␣
↪suggestion.\n\n\naaj 4/26/93\n'
```

[15]:
```
print(y[1], dataGen.classMap[y[1]])
```

```
1 medicine
```

[16]:
```
exps[1]
```

[16]:
```
{'shock': 0.5,
 'heat': 0.5,
 'proteins': 0.5,
 'dna': 0.5,
 'stomach': 0.5,
 'duodenum': 0.5,
 'dnaase': 0.5,
 'enzymes': 0.5,
 'pancreas': 0.5}
```

The second step is training an estimator and predicting explanations. We could use any system for training and generating the explanations. We are going to skip this step, as its independent to **teex** and up to the user to decide in which way to generate the explanations. Instead, we are going to use the ground truth explanations as if they were the predicted ones.

So, we compare the predicted explanations with the ground truth ones.

[19]:
```
metrics = ['prec', 'rec', 'fscore']
scores = word_importance_scores(exps, exps, metrics=metrics)
```

```
/usr/local/lib/python3.8/site-packages/sklearn/metrics/_classification.py:1248:␣
↪UndefinedMetricWarning: Precision is ill-defined and being set to 0.0 due to no␣
↪predicted samples. Use `zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
/usr/local/lib/python3.8/site-packages/sklearn/metrics/_classification.py:1248:␣
↪UndefinedMetricWarning: Recall is ill-defined and being set to 0.0 due to no true␣
↪samples. Use `zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
/usr/local/lib/python3.8/site-packages/sklearn/metrics/_classification.py:1495:␣
↪UndefinedMetricWarning: F-score is ill-defined and being set to 0.0 due to no true nor␣
↪predicted samples. Use `zero_division` parameter to control this behavior.
  _warn_prf(
/usr/local/lib/python3.8/site-packages/sklearn/metrics/_classification.py:1248:␣
↪UndefinedMetricWarning: Precision is ill-defined and being set to 0.0 due to no␣
↪predicted samples. Use `zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
/usr/local/lib/python3.8/site-packages/sklearn/metrics/_classification.py:1248:␣
↪UndefinedMetricWarning: Recall is ill-defined and being set to 0.0 due to no true␣
↪samples. Use `zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
/usr/local/lib/python3.8/site-packages/sklearn/metrics/_classification.py:1495:␣
↪UndefinedMetricWarning: F-score is ill-defined and being set to 0.0 due to no true nor␣
↪predicted samples. Use `zero_division` parameter to control this behavior.
  _warn_prf(
/usr/local/lib/python3.8/site-packages/sklearn/metrics/_classification.py:1248:␣
↪UndefinedMetricWarning: Precision is ill-defined and being set to 0.0 due to no␣
↪predicted samples. Use `zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
/usr/local/lib/python3.8/site-packages/sklearn/metrics/_classification.py:1248:␣
↪UndefinedMetricWarning: Recall is ill-defined and being set to 0.0 due to no true␣
↪samples. Use `zero_division` parameter to control this behavior.
```

(continues on next page)

(continued from previous page)

```
  _warn_prf(average, modifier, msg_start, len(result))
/usr/local/lib/python3.8/site-packages/sklearn/metrics/_classification.py:1495:␣
↪UndefinedMetricWarning: F-score is ill-defined and being set to 0.0 due to no true nor␣
↪predicted samples. Use `zero_division` parameter to control this behavior.
  _warn_prf(
```

[20]:
```
for i, metric in enumerate(metrics):
    print(f'{metric}: {scores[i]}')
```

```
prec: 0.9839572310447693
rec: 0.9839572310447693
fscore: 0.9839572310447693
```

We obtain quasi perfect scores, as the ground truths are exactly the same as the predictions. The fact that they are not 1 is due to instances when only 1 feature is available and thus, metrics are not well defined.

### 2.1.5 Model selection

In the following notebook, we demostrate how evaluation of explanations can be leveraged for model selection.

#### Leveraging explanation quality for model selection

In this notebook we are going to explore how, using **teex**, we can improve our model selection procedures. We follow the procedure described in Jia et al. (2021).

#### 0. Approach

Intuitively, a model that has a good 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. So, given an explanation model, we can investigate which evidence a model is basing its decisions on: a set of explanations will be of quality if it's based on reasonable evidence and of low quality otherwise. Then, given two models with similar predictive performance, we can leverage whether or not it is basing its decisions on good or bad evidence. With this intuition, we define a model scoring mechanism:

$$score(f) = \alpha \cdot \text{score}_{acc}(f) + (1 - \alpha) \cdot \text{score}_{\text{explanation}}(f)$$

where $\alpha \in [0, 1]$ is a hyperparameter, $f$ is the model being assessed and $\text{score}_{acc}(f)$ and $\text{score}_{\text{explanation}}(f)$ are accuracy and explanation scores, respectively. All models $f_1, ..., f_n$ will be assigned a score and we will choose based on it. **teex** will help us compute $\text{score}_{\text{explanation}}(f)$. From here, we will perform some experiments to see how correlated is the performance from models picked with this custom score with their performance on the test set.

[104]:
```
random_state = 888

import teex.saliencyMap.data as sm_data
import teex.saliencyMap.eval as sm_eval

import matplotlib.pyplot as plt
import numpy as np
import torch
import torch.nn as nn
```

(continues on next page)

(continued from previous page)

```
import copy
import random
import pickle
import altair as alt
import pandas as pd

random.seed(random_state)
torch.manual_seed(random_state)

from sklearn import metrics, model_selection
from captum import attr
from captum.attr import visualization as viz
from torch import optim
from torchvision import models, transforms
from torch.utils import data
from scipy import stats
```

### 1. Getting the data

We are going to work with a subset of the Oxford-IIIT Pet dataset, included in **teex**. It contains roughly 7000 images from 37 categories.

[105]:
```
oxford_data = sm_data.OxfordIIIT()
```

We have the following classes available

[106]:
```
oxford_data.classMap
```

[106]:
```
{0: 'cat_Abyssinian',
 1: 'dog_american_bulldog',
 2: 'dog_american_pit_bull_terrier',
 3: 'dog_basset_hound',
 4: 'dog_beagle',
 5: 'cat_Bengal',
 6: 'cat_Birman',
 7: 'cat_Bombay',
 8: 'dog_boxer',
 9: 'cat_British_Shorthair',
 10: 'dog_chihuahua',
 11: 'cat_Egyptian_Mau',
 12: 'dog_english_cocker_spaniel',
 13: 'dog_english_setter',
 14: 'dog_german_shorthaired',
 15: 'dog_great_pyrenees',
 16: 'dog_havanese',
 17: 'dog_japanese_chin',
 18: 'dog_keeshond',
 19: 'dog_leonberger',
 20: 'cat_Maine_Coon',
 21: 'dog_miniature_pinscher',
 22: 'dog_newfoundland',
```

(continues on next page)

(continued from previous page)

```
 23: 'cat_Persian',
 24: 'dog_pomeranian',
 25: 'dog_pug',
 26: 'cat_Ragdoll',
 27: 'cat_Russian_Blue',
 28: 'dog_saint_bernard',
 29: 'dog_samoyed',
 30: 'dog_scottish_terrier',
 31: 'dog_shiba_inu',
 32: 'cat_Siamese',
 33: 'cat_Sphynx',
 34: 'dog_staffordshire_bull_terrier',
 35: 'dog_wheaten_terrier',
 36: 'dog_yorkshire_terrier'}
```

We are going to work in a 4 class setting: let us choose some animals!

[107]:
```
num_classes = 4
batch_size = 10
input_size = 224

iSi, lSi, eSi = oxford_data.get_class_observations(32) # Siamese cats
iMaine, lMaine, eMaine = oxford_data.get_class_observations(20) # Maine Coon cats
iJapChin, lJapChin, eJapChin = oxford_data.get_class_observations(17) # Japanese Chin␣
↪dogs
iPinscher, lPinscher, ePinscher = oxford_data.get_class_observations(21) # Mini Pinscher␣
↪dogs

classDict = {
    32: 0,
    20: 1,
    17: 2,
    21: 3
}
```

[37]:
```
f, axarr = plt.subplots(2,4, figsize=(10, 3))

axarr[0,0].imshow(np.array(iSi[5]))
axarr[0,0].axis("off")
axarr[0,1].imshow(np.array(eSi[5]))
axarr[0,1].axis("off")

axarr[0,2].imshow(np.array(iMaine[0]))
axarr[0,2].axis("off")
axarr[0,3].imshow(np.array(eMaine[0]))
axarr[0,3].axis("off")

axarr[1,0].imshow(np.array(iJapChin[10]))
axarr[1,0].axis("off")
axarr[1,1].imshow(np.array(eJapChin[10]))
axarr[1,1].axis("off")
```

(continues on next page)

(continued from previous page)

```
axarr[1,2].imshow(np.array(iPinscher[3]))
axarr[1,2].axis("off")
axarr[1,3].imshow(np.array(ePinscher[3]))
axarr[1,3].axis("off")
```

[37]:
```
(-0.5, 276.5, 199.5, -0.5)
```

### 1.1. Utils

Here are some utils that will be used later on. This notebook implements classes and methods that can be reused and are robust, so that later experimentation is

- easy
- replicable

[38]:
```
def swap_to_original(im):
    """ Swap np.array from C*H*W to H*W*C or from W*H to H*W"""
    if len(im.shape) > 2:
        return np.swapaxes(np.swapaxes(im, 0, 2), 0, 1)
    else:
        return np.swapaxes(im, 0, 1)

def create_data(demo_data, exps, random_state, returnIndexes = False):
    """ Returns a dataloaders dict for training and two dicts:
        1. All data
        2. Explanations

        These dicts are indexed by the split (train / val / test). """

    data_length = len(demo_data)
    train_idx, test_idx = model_selection.train_test_split(list(range(data_length)),␣
↪train_size=0.3, random_state=random_state)
    train_idx, val_idx = model_selection.train_test_split(train_idx, test_size=0.3,␣
↪random_state=random_state)

    all_data = {}
    all_data['train'] = data.Subset(demo_data, train_idx)
    all_data['val'] = data.Subset(demo_data, val_idx)
```

(continues on next page)

(continued from previous page)

```
    all_data['test'] = data.Subset(demo_data, test_idx)

    all_exps = {}
    all_exps['train'] = exps[train_idx]
    all_exps['val'] = exps[val_idx]
    all_exps['test'] = exps[test_idx]

    dataloaders_dict = {split: data.DataLoader(all_data[split],
                                               batch_size=batch_size,
                                               shuffle=True,
                                               num_workers=0) for split in ['train', 'val',
↪'test']}

    if returnIndexes is True:
        return dataloaders_dict, all_data, all_exps, train_idx, val_idx, test_idx
    else:
        return dataloaders_dict, all_data, all_exps

def set_parameter_requires_grad(model):
    for param in model.parameters():
        param.requires_grad = False


def replace_layers(model, old, new):
    """ Many Captum attribution methods require
    ReLU operations to NOT be inplace. We will use this
    method to modify the pretrained model's behaviour. """

    for n, module in model.named_children():
        if len(list(module.children())) > 0:
            ## compound module, go inside it
            replace_layers(module, old, new)
        if isinstance(module, old):
            ## simple module
            try:
                n = int(n)
                model[n] = new
            except:
                setattr(model, n, new)

def highlight_max(s):
    '''
    highlight the maximum in a Series yellow.
    '''
    is_max = s == s.max()
    return ['background-color: yellow' if v else '' for v in is_max]
```

### 2. Declaring our classifiers

We create a common model class that will help us perform the experiments and implement its logic. See how we use `teex` in the explanation evaluation function!

[39]:
```
class Model:
    """ Our Model class. Training will be performed with
    learning rate of 0.001 and momentum 0.9, using a standard gradient descent
    optimizer and cross entropy loss. """

    def __init__(self, modelName: str, model: torch.nn.Module,
                 input_size: int, random_state: int = 888) -> None:
        self.modelName = modelName
        self.model = model
        self.random_state = random_state
        self.input_size = input_size

        self._trained = False
        self._valid_explainers = ("ggcam", "sal", "gshap", "bp")
        self._latent_explainers = {
            "ggcam": attr.GuidedGradCam,
            "sal": attr.Saliency,
            "gshap": attr.GradientShap,
            "bp": attr.GuidedBackprop
        }
        self._lr = 0.001
        self._device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
        self._optim_momentum = 0.9
        self._optimizer = optim.SGD(self.model.parameters(), lr=self._lr, momentum=self._
↪optim_momentum)
        self._criterion = torch.nn.CrossEntropyLoss()

    def train(self, dataloaders: dict, epochs: int = 10,
              log_models: bool = False, verbose: bool = True) -> None:

        if self._trained:
            raise Exception("Model is already trained.")

        model_log = [] # contains deep copies of self at each training epoch

        for epoch in range(epochs):
            if verbose:
                print('Epoch {} / {}'.format(epoch + 1, epochs))
                print('-' * 10)

            for phase in ['train', 'val']:
                if phase == 'train':
                    self.model.train()
                else:
                    self.model.eval()

                running_loss = 0.0
                running_corrects = 0
```

(continues on next page)

(continued from previous page)

```
                for inputs, labels in dataloaders[phase]:
                    inputs = inputs.to(self._device)
                    labels = labels.to(self._device)

                    # zero the parameter gradients
                    self._optimizer.zero_grad()

                    # forward
                    # track history if only in train
                    with torch.set_grad_enabled(phase == 'train'):
                        # Get model outputs and calculate loss
                        outputs = self.model(inputs)
                        loss = self._criterion(outputs, labels)

                        _, preds = torch.max(outputs, 1)

                        if phase == 'train':
                            loss.backward()
                            self._optimizer.step()

                    running_loss += loss.item() / inputs.size(0)
                    running_corrects += torch.sum(preds == labels.data)

                epoch_loss = running_loss / len(dataloaders[phase].dataset)
                epoch_acc = running_corrects.double() / len(dataloaders[phase].dataset)

                if verbose:
                    print(f"{self.modelName} {phase} Loss {epoch_loss:.4f} Acc {epoch_
↪acc:.4f}")

            if log_models:
                model_copy = copy.deepcopy(self)
                model_copy._trained = True
                model_log.append(model_copy)

        self._trained = True

        if log_models:
            return model_log

    def get_explanations(self,
                         X: np.ndarray,
                         method: str = "sal",
                         expParams: dict = None,
                         attrParams: dict = None) -> np.ndarray:

        if self._check_is_trained() and self._check_explainer(method):
            if expParams is None:
                expParams = {}
            if attrParams is None:
                attrParams = {}
```

(continues on next page)

(continued from previous page)

```
            explainer = self._get_explainer(method, **expParams)
            pred_labels = self.get_predictions(X)
            exps = explainer.attribute(inputs=X, target=pred_labels, **attrParams)
            retExps = []
            for exp in exps:
                swapped = swap_to_original(exp.detach().numpy())
                retExps.append(np.abs(viz._normalize_image_attr(swapped, 'positive', 2)))
            return np.stack(retExps)

    def get_predictions(self, X):
        if self._check_is_trained():
            _ , preds = torch.max(self.model(X), dim=1)
            return preds

    def compute_acc(self, gtY, y) -> float:
        scores = metrics.accuracy_score(y, gtY)
        return np.mean(scores)

    def compute_exp_score(self, gtExps, predExps) -> float:
        return sm_eval.saliency_map_scores(gtExps, predExps, metrics="auc")

    def compute_total_score(self,
                            gtY: np.ndarray,
                            y: np.ndarray,
                            gtExps: np.ndarray,
                            exps: np.ndarray,
                            alpha: float) -> float:

        return [alpha * self.compute_acc(gtY, y) + (1 - alpha) * self.compute_exp_
↪score(gtExps, exps)][0][0]

    def _check_is_trained(self) -> bool:
        if self._trained:
            return True
        else:
            raise Exception("Model is not trained yet!")

    def _check_explainer(self, exp: str) -> bool:
        if exp not in self._valid_explainers:
            raise ValueError(f"Explainer {exp} is not valid. ({self._valid_explainers})")
        else:
            return True

    def _get_explainer(self, exp: str, **kwargs) -> bool:
        if kwargs is not None:
            explainer = self._latent_explainers[exp](self.model, **kwargs)
        else:
            explainer = self._latent_explainers[exp](self.model)

        return explainer
```

We then get 3 models, modify them to our use case and initialize them. 1 of them is not pre-trained, the others are.

[40]:
```
def generate_models(num_classes, input_size, modelNames):

    """ Instance a SqueezeNet and a ResNet18, both pretrained, and modify them to our
    particular problem. """

    # SqueezeNet
    sqz = models.squeezenet1_0(weights="DEFAULT")
    set_parameter_requires_grad(sqz)
    replace_layers(sqz, nn.ReLU, nn.ReLU(inplace=False))
    sqz.classifier[1] = torch.nn.Conv2d(512, num_classes, kernel_size=(1,1), stride=(1,
↪1))
    sqz.num_classes = num_classes

    # ResNet18 (pre-trained)
    rnetPre = models.resnet18(weights="DEFAULT")
    set_parameter_requires_grad(rnetPre)
    replace_layers(rnetPre, nn.ReLU, nn.ReLU(inplace=False))
    num_ftrs = rnetPre.fc.in_features
    rnetPre.fc = nn.Linear(num_ftrs, num_classes)

    # ResNet18 (random init.)
    rnetEmpty = models.resnet18(weights=None)
    set_parameter_requires_grad(rnetEmpty)
    replace_layers(rnetEmpty, nn.ReLU, nn.ReLU(inplace=False))
    num_ftrs = rnetEmpty.fc.in_features
    rnetEmpty.fc = nn.Linear(num_ftrs, num_classes)

    squeezeNet = Model(modelNames[0], sqz, input_size)
    resNetPre = Model(modelNames[1], rnetPre, input_size)
    resNetEmpty = Model(modelNames[2], rnetEmpty, input_size)

    return squeezeNet, resNetPre, resNetEmpty
```

### 3. Preparing our data

Now, we are going to incorporate our data into a `torch.Dataset`, for easy batching and loading. We also create the data splits and transform the observations to the model's required input size.

[108]:
```
class DemoData(data.Dataset):
    """ A base Dataset class for compatibility with PyTorch """

    def __init__(self, data, labels) -> None:
        self.data = data
        self.labels = labels

    def __len__(self):
        return len(self.data)

    def __getitem__(self, slice):
        return self.data[slice], self.labels[slice]
```

(continues on next page)

(continued from previous page)

```
def transform(images, input_size, normalize=True):

    """ Transformation of input images so that they work with our models.
    A resize, centering and a normalisation (only if the
    images are not explanations) are performed. """

    transform = transforms.Compose([
        transforms.ToTensor(),
        transforms.Resize(input_size),
        transforms.CenterCrop(input_size)
    ])

    if normalize:
        norm = transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
        d = [norm(transform(im)) for im in images]
    else:
        d = [transform(im) for im in images]
    return torch.stack(d, axis=0)


indexes = random.sample(list(range(len(iSi) + len(iPinscher) + len(iJapChin) +␣
↪len(iMaine))),
                        len(iSi) + len(iPinscher) + len(iJapChin) + len(iMaine))
ims, labs, exps = (iSi + iPinscher + iJapChin + iMaine,
                   lSi + lPinscher + lJapChin + lMaine,
                   eSi + ePinscher + eJapChin + eMaine)

original_images = transform([ims[idx] for idx in indexes], input_size, normalize=False)
ims = transform([ims[idx] for idx in indexes], input_size, normalize=True)
# transform labels from their class ID to [0, 1, 2, 3]
labs = torch.LongTensor([classDict[labs[idx]] for idx in indexes])
exps = np.squeeze(transform([exps[idx] for idx in indexes], input_size, normalize=False).
↪numpy())

demo_data = DemoData(ims, labs)
```

now, using a previously declared utility function, creating the dataloaders is easy. By default, we are using a 20 / 10 / 70 split and a batch size of 10 on all experiments.

[109]:
```
dataloaders_dict, all_data, all_exps, _, _, test_idx = create_data(demo_data, exps,␣
↪random_state, returnIndexes = True)
```

We can easily train the models:

[110]:
```
modelNames = ["SqueezeNet", "ResNet18 (pre-trained)", "ResNet18 (random init)"]
squeezeNet, resNet, resNetEmpty = generate_models(num_classes, input_size, modelNames)

squeezeNet.train(dataloaders_dict, epochs=2)
resNet.train(dataloaders_dict, epochs=2)
resNetEmpty.train(dataloaders_dict, epochs=2)
```

```
Epoch 1 / 2
```

(continues on next page)

(continued from previous page)

```
----------
SqueezeNet train Loss 0.0093 Acc 0.6429
SqueezeNet val Loss 0.0084 Acc 0.9028
Epoch 2 / 2
----------
SqueezeNet train Loss 0.0023 Acc 0.9107
SqueezeNet val Loss 0.0014 Acc 0.9583
Epoch 1 / 2
----------
ResNet18 (pre-trained) train Loss 0.0108 Acc 0.5952
ResNet18 (pre-trained) val Loss 0.0104 Acc 0.8472
Epoch 2 / 2
----------
ResNet18 (pre-trained) train Loss 0.0047 Acc 0.8869
ResNet18 (pre-trained) val Loss 0.0039 Acc 1.0000
Epoch 1 / 2
----------
ResNet18 (random init) train Loss 0.0155 Acc 0.2024
ResNet18 (random init) val Loss 0.0226 Acc 0.2361
Epoch 2 / 2
----------
ResNet18 (random init) train Loss 0.0150 Acc 0.2798
ResNet18 (random init) val Loss 0.0258 Acc 0.2361
```

With the models trained, we can also perform predictions and evaluate them in a straightforward manner (note that here they only have been trained for 2 epochs):

[26]:
```
test_preds = squeezeNet.get_predictions(all_data['test'][:][0])
test_acc = squeezeNet.compute_acc(test_preds, all_data['test'][:][1])
print(f"SqueezeNet: {round(test_acc * 100, 2)}% of test accuracy.")

test_preds = resNet.get_predictions(all_data['test'][:][0])
test_acc = resNet.compute_acc(test_preds, all_data['test'][:][1])
print(f"ResNet (pre-trained): {round(test_acc * 100, 2)}% of test accuracy.")

test_preds = resNetEmpty.get_predictions(all_data['test'][:][0])
test_acc = resNetEmpty.compute_acc(test_preds, all_data['test'][:][1])
print(f"ResNet (random init): {round(test_acc * 100, 2)}% of test accuracy.")
```

```
SqueezeNet: 93.39% of test accuracy.
ResNet (pre-trained): 98.04% of test accuracy.
ResNet (random init): 25.54% of test accuracy.
```

We can also compute explanations with different explanation methods:

[113]:
```
pred_exps_sal = squeezeNet.get_explanations(all_data['test'][:5][0], method="sal")
pred_exps_bp = squeezeNet.get_explanations(all_data['test'][:5][0], method="bp")
pred_exps_ggcam = squeezeNet.get_explanations(all_data['test'][:5][0],
                                              method="ggcam",
                                              expParams={"layer": squeezeNet.model.
↪features[12]})
pred_exps_gshap = squeezeNet.get_explanations(all_data['test'][:5][0],
                                              method="gshap",
```

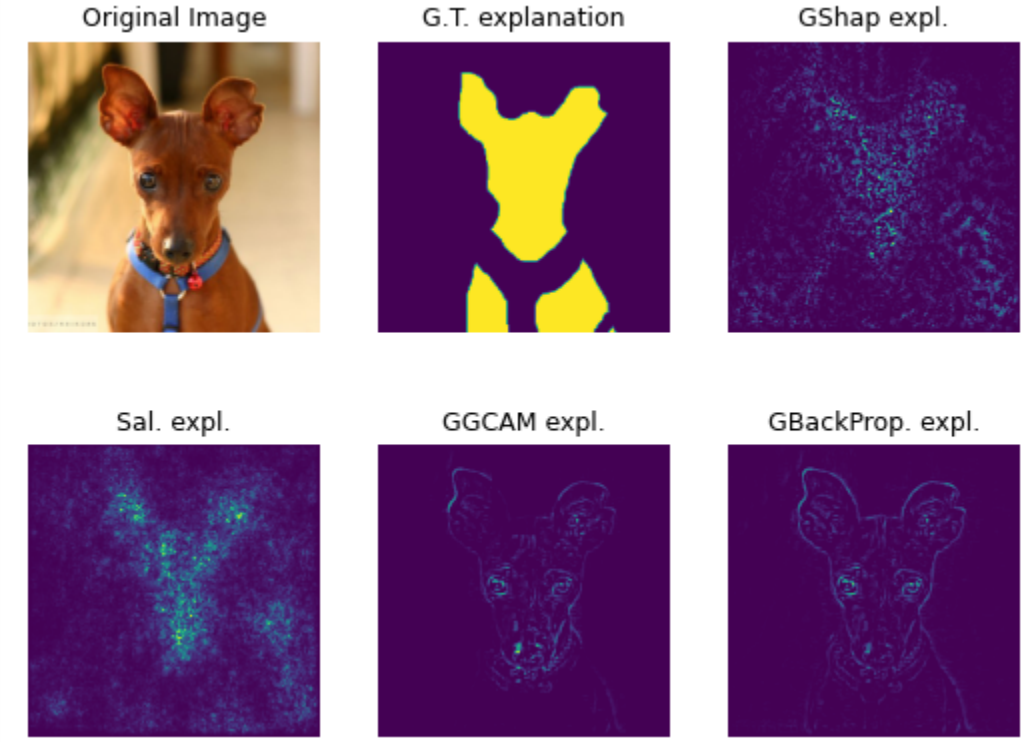
(continues on next page)

(continued from previous page)

```
                                                attrParams={'baselines': torch.zeros((1, 3,
↪ input_size, input_size))})
```

[114]:
```
i = 4
f, axarr = plt.subplots(2, 3)
axarr[0,0].imshow(np.array(swap_to_original(original_images[test_idx][i])))
axarr[0,0].axis("off")
axarr[0,0].set_title('Original Image', fontsize=9)

axarr[0,1].imshow(np.array(all_exps['test'][i]))
axarr[0,1].axis("off")
axarr[0,1].set_title('G.T. explanation', fontsize=9)

axarr[0,2].imshow(pred_exps_gshap[i])
axarr[0,2].axis("off")
axarr[0,2].set_title('GShap expl.', fontsize=9)

axarr[1,0].imshow(pred_exps_sal[i])
axarr[1,0].axis("off")
axarr[1,0].set_title('Sal. expl.', fontsize=9)

axarr[1,1].imshow(pred_exps_ggcam[i])
axarr[1,1].axis("off")
axarr[1,1].set_title('GGCAM expl.', fontsize=9)

axarr[1,2].imshow(pred_exps_bp[i])
axarr[1,2].axis("off")
axarr[1,2].set_title('GBackProp. expl.', fontsize=9)
```

[114]:
```
Text(0.5, 1.0, 'GBackProp. expl.')
```




Now we are ready to experiment.

### 4. Performing the experiments

In this section we are going to put everything together. These are the steps to be followed in order to obtain our conclusions:

1. Randomly split our data into 20 / 10 / 70 train, validation and test sets.
2. Train our models for 10 epochs: we keep each epoch as a model configuration, leaving us with $N = 2 \cdot 10$ models $f_1, ..., f_N$.
3. Evaluate, using our custom score function, each model $f_i$ with $\alpha \in \{0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1\}$ on the validation set. Note that we use GuidedGradCAM for the generation of explanations.
4. Compute the test accuracy for all models $f_i$.
5. Compute the Pearson correlation between the custom validation scores and test accuracy.
6. Repeat steps 1 to 5 (5 times) and average the correlations.

[47]:
```
def experiment_epoch(data, explanations, alphas, modelNames, random_state):
    """ Steps 1 to 5 of our experiment guide. Return correlations for each model f_i and␣
↪alpha value. """

    # 1. split data
    dataloaders_dict, all_data, all_exps = create_data(data, explanations, random_state)
    correlations = {}

    for alpha_value in alphas:
```

(continues on next page)

(continued from previous page)

```
        squeezeNet, resNetPre, resNetEmpty = generate_models(num_classes, input_size,␣
↪modelNames)
        evals = {}
        correlations[alpha_value] = {}

        # 2. train models
        sqzModels = squeezeNet.train(epochs=1, dataloaders=dataloaders_dict, log_
↪models=True, verbose=False)
        resPreModels = resNetPre.train(epochs=1, dataloaders=dataloaders_dict, log_
↪models=True, verbose=False)
        resEmptyModels = resNetEmpty.train(epochs=10, dataloaders=dataloaders_dict, log_
↪models=True, verbose=False)

        for model in sqzModels + resPreModels + resEmptyModels:
            if model.modelName not in evals:
                evals[model.modelName] = {}
                evals[model.modelName]['score'] = []
                evals[model.modelName]['acc'] = []
            # 3. get custom score on validation
            explainerLayer = model.model.layer4 if "ResNet18" in model.modelName else␣
↪model.model.features[12]
            PredValExps = model.get_explanations(all_data['val'][:][0],
                                                 method="ggcam",
                                                 expParams={"layer": explainerLayer})
            valPreds = model.get_predictions(all_data['val'][:][0])
            score = model.compute_total_score(all_data['val'][:][1], valPreds,
                                              all_exps['val'][:], PredValExps, alpha_
↪value)
            # 4. get accuracy on test
            testPreds = model.get_predictions(all_data['test'][:][0])
            testAcc = model.compute_acc(all_data['test'][:][1], testPreds)

            evals[model.modelName]['score'].append(score)
            evals[model.modelName]['acc'].append(testAcc)

        # 5. compute correlation
        for modelName in modelNames:
            corr, _ = stats.pearsonr(evals[modelName]['score'],
                                     evals[modelName]['acc'])
            correlations[alpha_value][modelName] = corr

    return correlations
```

We can now average the correlations from all 5 experiment epochs (Warning: long execution time – ~10h in a M1 Pro machine):

[ ]:
```
experiment_epochs = 5
final_corrs = {}
alphas = np.arange(0, 1.1, 0.1)

for i in range(experiment_epochs):
    corrs = experiment_epoch(demo_data, exps, alphas, modelNames, random_state+i)
```

(continues on next page)

(continued from previous page)

```
    for alpha, corrsDict in corrs.items():
        if alpha not in final_corrs:
            final_corrs[alpha] = {}
        for model, corr in corrsDict.items():
            if model not in final_corrs[alpha]:
                final_corrs[alpha][model] = [corr]
            else:
                final_corrs[alpha][model].append(corr)

# average results
for alpha, corrsDict in corrs.items():
    for model, corr in corrsDict.items():
        final_corrs[alpha][model] = sum(final_corrs[alpha][model]) / experiment_epochs
```

Optionally save and load the results:

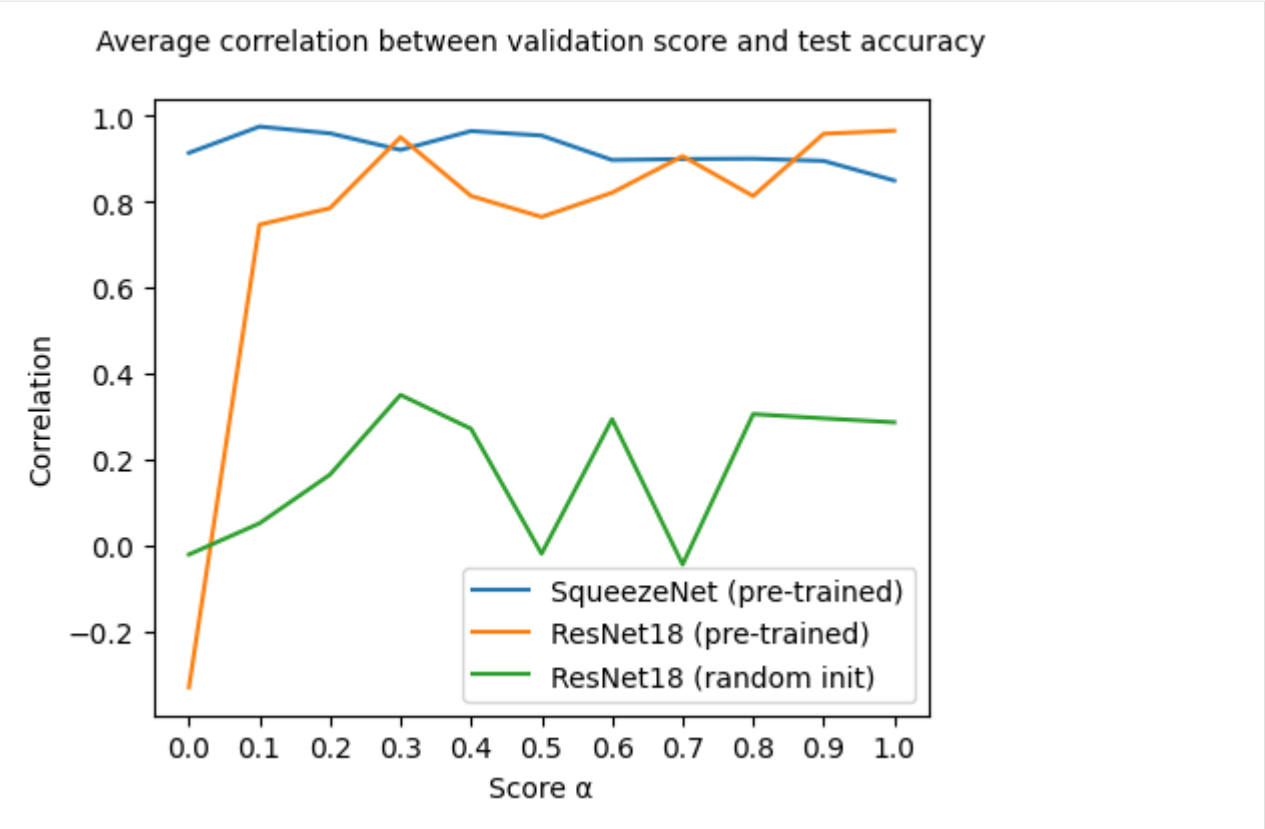
[ ]:
```
# with open("final_corrs", "wb") as f:
#     pickle.dump(final_corrs, f)
```

[100]:
```
with open("final_corrs", "rb") as f:
    final_corrs = pickle.load(f)
```

[101]:
```
resultsDf = pd.DataFrame.from_dict(final_corrs)
alphas_beautyfied = [f" = {round(alpha, 1)}" for alpha in alphas]
resultsDf.set_axis(alphas_beautyfied, axis=1).style.apply(highlight_max, axis=1)
```

[101]:
```
<pandas.io.formats.style.Styler at 0x2aefe33d0>
```

[103]:
```
plt.figure(figsize=(5, 4))
plt.plot(resultsDf.transpose())
plt.xticks(alphas)
plt.xlabel("Score ")
plt.ylabel("Correlation")
plt.legend(["SqueezeNet (pre-trained)", "ResNet18 (pre-trained)", "ResNet18 (random init)
↪"])
plt.title("Average correlation between validation score and test accuracy\n",␣
↪fontsize=10)
plt.show()
```




Here a few observations from our results:

1. SqueezeNet (pre-trained). Performance on test seems to be more correlated to validation explanation quality than to validation accuracy, as can be seen by the steady drop for bigger $\alpha$ values. Having in mind that $\alpha = 1$ would mean that we only take into account accuracy and that $\alpha = 0$ would mean that we only consider explanation quality, we see that, on average, this model perform best with some, but not too much, explanation quality in the mix.

2. ResNet18 (pre-trained). Opposite to SqueezeNet, correlation between only explanation quality ($\alpha = 0$) and test performance is negative on the ResNet model and sharply goes up incorporating accuracy information. Looking into test accuracies, we notice that models in all training epochs have almost perfect test accuracy (~1). In this scenario, and for this particular architecture, it's possible that explanation quality cannot add a lot of information. This, in turn, can mean that the ordering based on scores is not relevant (explanation scores are all almost equal due to all model performances being ~1).

3. ResNet18 (random init). In this case, where model performances are not perfect, we note how mixing explanation quality and validation accuracy makes correlation increase steadily, reaching a global maximum at $\alpha = 0.3$. At this value, a bigger fraction of explanation quality is taken into account for computing the score than of validation accuracy.

### 5. Next steps: the real world

Now that we have an intuition on how explanation quality can be leveraged for model selection, how do we proceed? In a real environment, one would have to consider how $\alpha$ is impacting results. In particular, as we cannot decide on a value ad-hoc, we could, for example, choose a range of values. With that range of values, one can:

- Use as definitive the most (best) frequent model whithin them
- Choose the top-k most frequent models and analyse their real-world performance on the go

Even better, we could find a way to better optimise $\alpha$ without the need to pick multiple models or sacrifing other models that could perform well. This is an interesting research question, and we look forward to seeing it evolve!

### 6. Conclusion: what we've seen

This was quite a lot! Thank you for your time. Wrapping up, we want to briefly go over the points that this notebook addresses. In particular, this showcase mainly serves the purpose of

- Being a complex example of how `teex` can be used
- Showing how evaluating explanations in the way that `teex` proposes can be useful
- Showcasing the ease of use of both loading datasets and evaluating explanations with `teex`: Although everything surrounding it is complex, see how short the explanation evaluation function is on the class `Model`! We provide robust evaluation API's so that you can build whatever you like on top of them.

On a final note: the results reported here are not intended to be a representation of the ones from the paper in which the experiments were based. For more extensive results, check the paper out!

CHAPTER

THREE

# 3. API REFERENCE

## 3.1 API reference

### 3.1.1 teex package

#### Subpackages

#### teex.decisionRule package

#### 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 instanced 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 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:* 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:* 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:* 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: _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: `_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.**clean_binary_statement**(*bounds: list*)

Parses binary statement edge cases from a list of operators and values. Checks if the edge cases occur for any pair of operator and value. Does not fix errors with bounds != or =.

f > 3 & f > 4 TRANSFORMS INTO f > 4 f > 3 & f >= 4 TRANSFORMS INTO f >= 4

f < 3 & f < 4 TRANSFORMS INTO f < 3 f <= 3 & f < 4 TRANSFORMS INTO f <= 3

**Parameters**

**bounds** (*list*) – list with bounds i.e. [(op, val), ..., (op, val)]

**Returns**

op1, val1, op2, val2

teex.decisionRule.data.**rule_to_feature_importance**(*rules*, *allFeatures*) → 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*) → Tuple[List[*DecisionRule*], list]

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

**Example**

```
>>> import pandas as pd
>>> from rulefit import RuleFit
>>>
>>> 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)
>>>
>>> dRules, _ = rulefit_to_decision_rule(rf.get_rules(), rf.predict(X))
```

**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'*) → *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 DecisionRule object.

### teex.decisionRule.eval module

Module for evaluation of decision rule explanations.

teex.decisionRule.eval.**complete_rule_quality**(*gts: DecisionRule*, *rules: 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 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:

$$q(e, \tilde{e}) = \frac{1}{N_{\not\infty}} \sum_{i=1}^{|e|} \delta_\varepsilon(e_i, \tilde{e}_i),$$

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 $N_{\not\infty}$ 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:* DecisionRule, *rules:* 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 sets of decision rules.
  - 'cs': Computes the Cosine Similarity between the two sets of decision 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`.

### teex.featureImportance package

### 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: `_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: `_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 - norm(f(data)), norm(f(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.data.**lime_to_feature_importance**(*exp*, *nFeatures*, *label=1*)

Convert from a `lime.explanation.Explanation` object to a np.array feature importance vector.

**Parameters**

- **exp** (*lime.explanation.Explanation*) – explanation to convert to vector.

- **label** – (int, str) label of lime explanation. If lime explanations are generated by default, then it will be 1.
- **nFeatures** (*int*) – number of features in the explanation

**Returns**
feature importance vector

**Return type**
np.ndarray

teex.featureImportance.data.**scale_fi_bounds**(*x: ndarray*, *verbose: bool = False*)

Map values of an 1D or 2D np.ndarray on certain conditions. The mapping is on a by-column basis. That is, each column will be separately scaled.:

```
(for each column in ``x``)
if values in the range [-1, 1] or [0, 1]       -> do nothing
else:
    case 1: if values in the [0, inf] range    -> map to [0, 1]
    case 2: if values in the [-inf, 0] range   -> map to [-1, 1]
    case 3: if values in the [-inf, inf] range -> map to [-1, 1]
```

### 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, val) ('max')

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

teex.featureImportance.eval.**feature_importance_scores**(*gts*, *preds*, *metrics=None*, *average: bool = True*, *thresholdType: str = 'abs'*, *binThreshold: float = 0.5*, *verbose: bool = True*)

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 <= `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 <= `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** (*bool*) – Verbosity of warnings. `True` will report warnings, `` `False` `` will not.

**Returns**

(ndarray of shape (n_metrics,) or (n_samples, n_metrics)) specified metric/s in the indicated order.

### teex.saliencyMap package

### 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. Note that all real-world datasets implement the `delete_data` method, which allows to delete all of their downloaded internal data. In this module, images and explanations are represented by the `PIL.Image.Image` class.

**class** teex.saliencyMap.data.**CUB200**

Bases: `_ClassificationDataset`

The CUB-200-2011 Classification Dataset. 11788 observations with 200 different classes. From

Wah, Branson, Welinder, Perona, & Belongie. (2022). CUB-200-2011 (1.0) [Data set]. CaltechDATA. https://doi.org/10.22002/D1.20098

**get_class_observations**(*classId: int*)

Get all observations from a particular class given its index.

**Args:**

**classId (int): Class index. It can be consulted from the attribute** `CUB200.classMap`

**Returns:**

imgs (list): Images pertaining to the specified class. labels (list): Int labels pertaining to the specified class. exps (list): Explanations pertaining to the specified class.

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

Bases: `_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.

**get_class_observations**(*classId: int*) → list

Get observations of a particular class.

**Parameters**

**classId** (*int*) – Class ID. See attribute `classMap`.

**Returns**

Observations of the specified type.

**Return type**

list

**class** teex.saliencyMap.data.**OxfordIIIT**

Bases: _ClassificationDataset

The Oxford-IIIT Pet Dataset. 7347 images from 37 categories with approximately 200 images per class. From

O. M. Parkhi, A. Vedaldi, A. Zisserman and C. V. Jawahar, "Cats and dogs," 2012 IEEE Conference on Computer Vision and Pattern Recognition, 2012, pp. 3498-3505, doi: 10.1109/CVPR.2012.6248092.

**get_class_observations**(*classId: int*) → list

Get all observations from a particular class given its index.

**Args:**

**classId (int): Class index. It can be consulted from the attribute** classMap.

**Returns:**

imgs (np.ndarray): Images pertaining to the specified class. labels (np.ndarray): Int labels pertaining to the specified class. exps (np.ndarray): Explanations pertaining to the specified class.

**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: _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 <= imageH and multiple of cellH.
- **patternW** (*int*) – width in pixels of the pattern. Must be <= 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 substracted 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 ~1, the other two ~0).
- **randomState** (*int*) – random seed.

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

Bases: _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: ndarray*) → 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: ndarray*, *cellH: int = 1*, *cellW: int = 1*) → None

Fits the model.

**predict**(*obs: ndarray*) → 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: ndarray*) → 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'*) → 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.**delete_sm_data**() → None

Removes from internal storage all downloaded Saliency Map datasets. See the `delete_data` method of all Saliency Map datasets to delete only their corresponding data.

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

## teex.wordImportance package

### 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: `_ClassificationDataset`

20 Newsgroup dataset. Contains 188 human annotaded newsgroup texts belonging to two categories. From

Sina Mohseni, Jeremy E Block, and Eric Ragan. 2021. Quantitative Evaluation of Machine Learning Explanations: A Human-Grounded Benchmark. https://doi.org/10.1145/3397481.3450689

**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*, *verbose: bool = False*) → 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?
- **verbose** (*bool*) – Will the call be verbose?

**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]]
```

# PYTHON MODULE INDEX

## t

# INDEX

## R



## S



## T



## U



## W