anlearn

Release 0.1.2

Ondrej Kurák

May 10, 2021

CONTENT

In Gauss Algorithmic, we're working on many anomaly/fraud detection projects using open-source tools. We decided to put our two cents in and "tidy up" some of our code snippets, add documentation, examples, and release them as an open-source package. So let me introduce **anlearn**. It aims to offer multiple interesting anomaly detection methods in familiar scikit-learn API so you could quickly try some anomaly detection experiments yourself.

So far, this package is an alpha state and ready for your experiments.

Do you have any questions, suggestions, or want to chat? Feel free to contact us via Github, Gitter, or email.

CHAPTER

ONE

INSTALLATION

anlearn depends on scikit-learn and it's dependencies scipy and numpy.

Requirements:

- python >=3.6
- scikit-learn
- scipy
- numpy

Requirements for every supported python version with version and hashes could be found in requirements folder. We're using pip-tools for generating requirements files.

1.1 Intallation options

1.1.1 PyPI installation

```
pip install anlearn
```

1.1.2 Installation from source

```
git clone https://github.com/gaussalgo/anlearn
cd anlearn
```

Installing requirements.

```
# Generated requirements for all supported python versions
ls requirements/requirements-3.*.txt | grep -v dev
requirements/requirements-3.6.txt
requirements/requirements-3.7.txt
requirements/requirements-3.8.txt
pip install -r requirements/requirements-3.8.txt
```

or

```
pip install scikit-learn numpy scipy
```

Install *anlearn*.

```
pip install .
```

or

```
python setup.py install
```

CHAPTER

TWO

LODA: LIGHTWEIGHT ON-LINE DETECTOR OF ANOMALIES

The lightweight on-line detector of anomalies (LODA) is an outlier detection method proposed by Tomáš Pevný in 2015 article Loda: Lightweight on-line detector of anomalies[1].

LODA is a simple yet very sophisticated ensemble of weak estimators which results in a fast and robust anomaly detection model. The most significant advantages of this model are its simplicity, speed, ability to explain the cause of an anomaly, option for on-line training, and robustness to missing values.

Without going into too many technical details, the "hearth" of LODA consists of one-dimensional histograms constructed on sparse random projections. Random sparse projects allow LODA to use simple one-dimensional histograms, thus processing large datasets in relatively small time complexity. On top of that, with smart usage of their sparsity, it's possible not only to evaluate samples with missing features but also to use them in the training process. Because histograms are one of the simplest density estimators available, they have low construction/evaluation time complexity. Anomaly score is a negative average log probability estimated from histogram on projections. Also, it has a higher time complexity than scoring samples because we need to evaluate every feature separately.

```
>>> import numpy as np
>>> from anlearn.loda import LODA

>>> X = np.array([[0, 0], [0.1, -0.2], [0.3, 0.2], [0.2, 0.2], [-5, -5], [0.6, 0.7]])

>>> loda = LODA(n_estimators=10, bins=10, random_state=42)
>>> loda.fit(X)
LODA(bins=10, n_estimators=10, random_state=42)
>>> loda.predict(X)
array([ 1,  1,  1,  1, -1,  1])
```

Bellow, you can see a comparison of LODA and outlier detection algorithms included in scikit-learn (*Comparison of scikit-learn anomaly detection methods and LODA*).

2.1 LODA parameters

LODA is a quite simple outlier detection model. Our implementation so far has only three primary parameters: **n_estimators**, **bins**, and **random_seed**.

- **n_estimators**: number of projections and histograms.
- **bins**: number of bins for each histogram.
- **random_state**: random state for stochastic parts.

[1] Pevný, T. Loda: Lightweight on-line detector of anomalies. Mach Learn 102, 275–304 (2016). <https://doi.org/10.1007/s10994-015-5521-0>

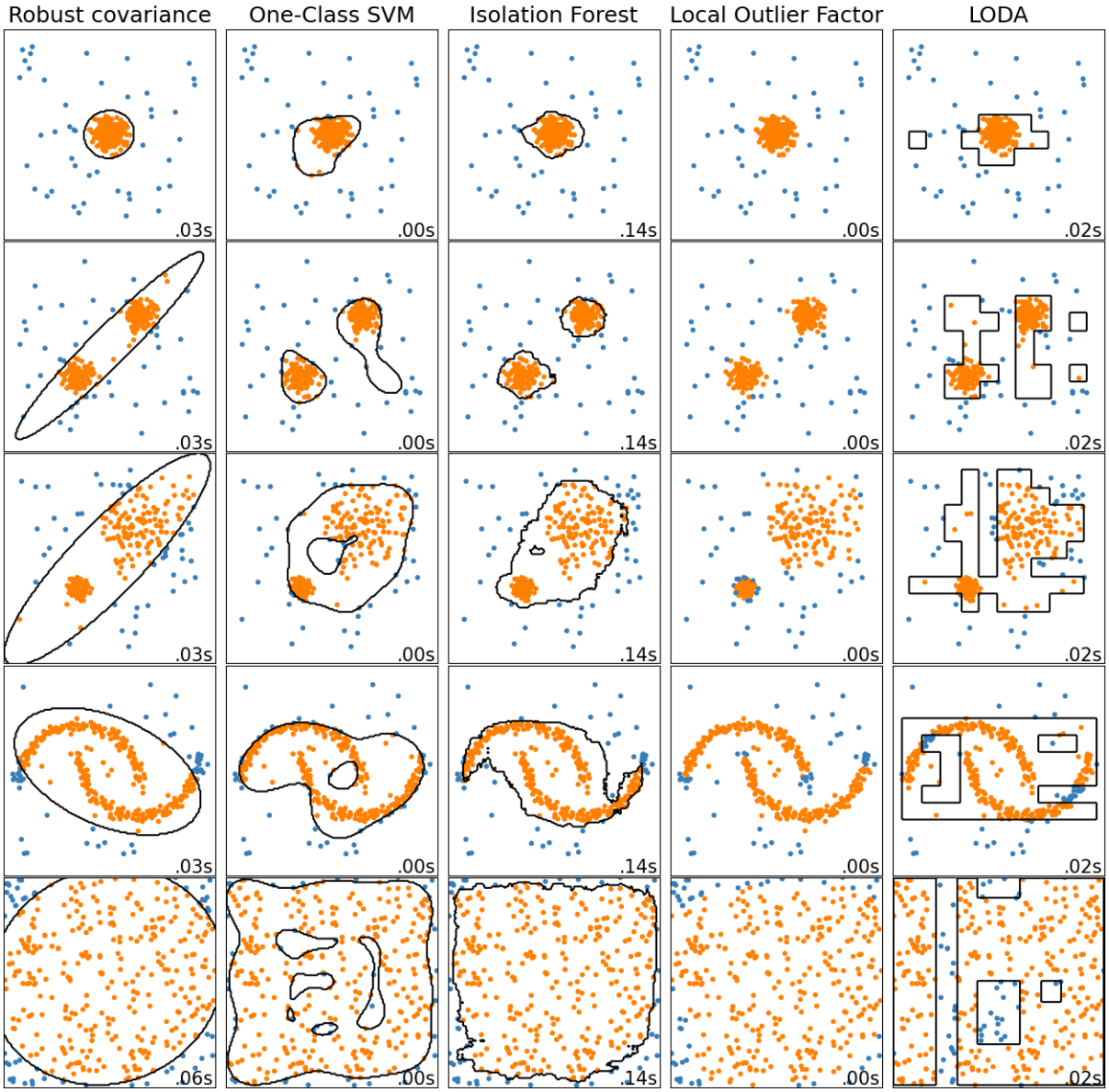


Fig. 1: *Comparison of scikit-learn anomaly detection methods and LODA*

- **q**: quantile for eveluating "anomalous" points during *predict* method. For detecting the "outliers" in predict method, we use the threshold evaluated from anomaly scores on training samples. For this purpose, we compute q quantile from training samples. For anomaly detection, we use the supposed percentage of abnormal points, for novelty detection 0.

See API docs for more details (`anlearn.loda.LODA`).

2.2 Projections and histograms

2.2.1 Random projections

As mentioned before, the "hearth" of LODA consists of random sparse projections and one-dimensional histograms.

Projections are sparse vectors with $\sqrt{d}$ non-zero features (d is number of dimensions of input data). Non-zero values are generated from $N(0; 1)$. This choice allows approximating the quantity of L_2 distances between points from input space in projected space[Page 5, 1]. Simply, we could take this as looking at the data from different angles. Sparsity also allows LODA to train and evaluate on data with missing values.

```
>>> loda.projections_
array([[-1.01283112,  0.        ],
       [ 0.        ,  0.31424733],
       [ 0.        , -0.90802408],
       [-1.4123037 ,  0.        ],
       [ 1.46564877,  0.        ],
       [-0.2257763 ,  0.        ],
       [ 0.        ,  0.0675282 ],
       [-1.42474819,  0.        ],
       [-0.54438272,  0.        ],
       [ 0.        ,  0.11092259]])
```

Right now, the number of projections is set on LODA initialization (**n_estimators** parameter) and initialized at the start of model fitting. In the future, we plan to implement an automatic selection for the number of projections.

See *LODA: projections & histograms* for full example.

2.2.2 Histograms

Histograms are the second essential part of the LODA model. In our implementation, we're using equi-width histograms. It's more or less for practical reasons. In the LODA article experiments, we could see that this type of histograms outperformed others (section **3.3 Histogram** and **4.Experiments**[Page 5, 1]). On top of that, it's straightforward to implement and fast. In future releases, we plan to introduce more flexibility in histograms (different types, online learning, etc.).

`anlearn.loda.Histogram` is implemented as a scikit-learn BaseEstimator (it shares similarities with `scipy.stats.rv_histogram`). For detecting bin width and intervals, we're using `numpy.histogram` function.

```
>>> loda.hists_
[Histogram(bins=10),
Histogram(bins=10),
Histogram(bins=10),
Histogram(bins=10),
Histogram(bins=10),
Histogram(bins=10),
```

(continues on next page)

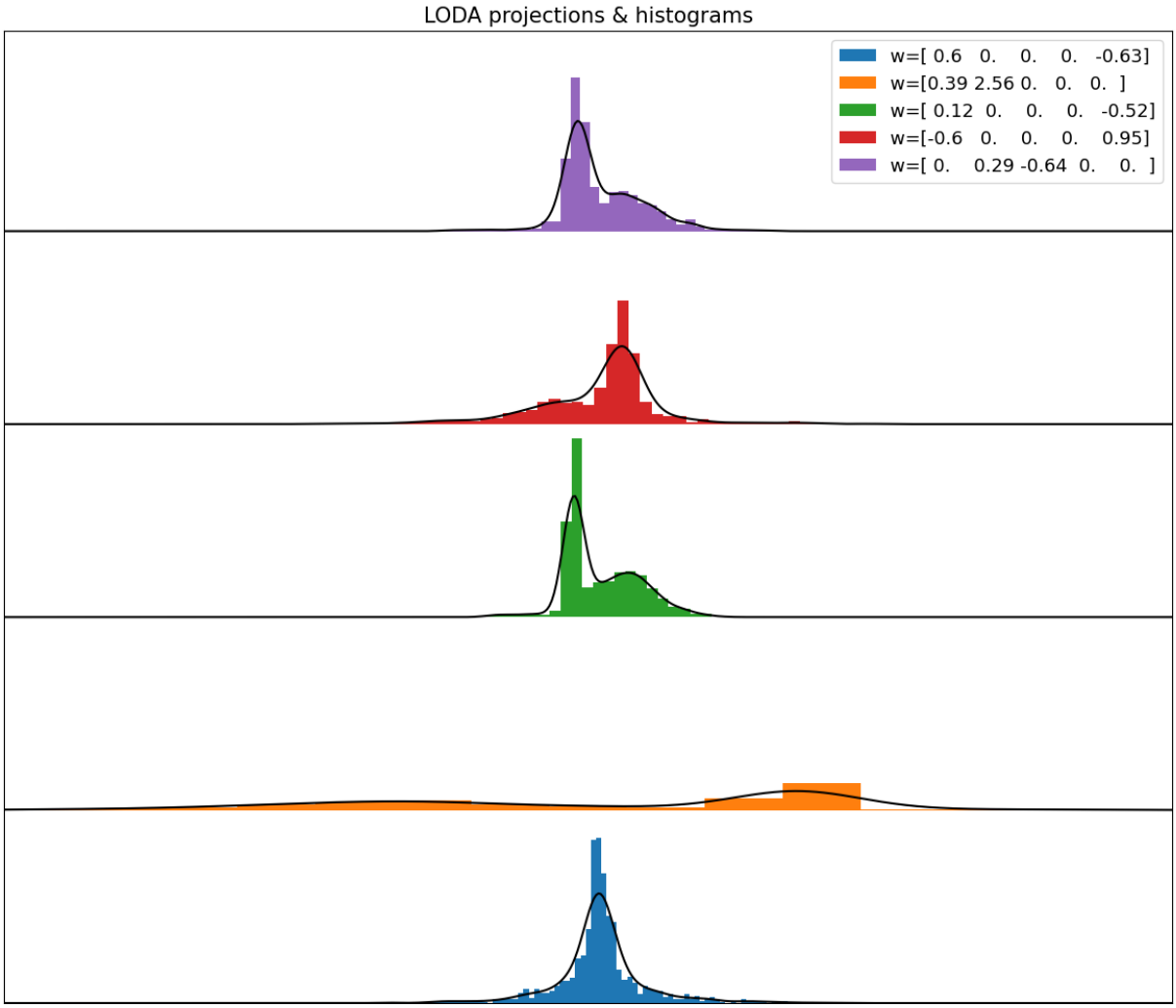


Fig. 2: *LODA: projections & histograms*

(continued from previous page)

```
Histogram(bins=10),
Histogram(bins=10),
Histogram(bins=10),
Histogram(bins=10)]
```

2.3 Explaining the cause of an anomaly

The knowledge that an example is anomalous just the first part of the whole anomaly detection pipeline. Without investigating further, I would consider this information almost useless. Lucky for us, LODA has a built-in way to get a little bit more information about why a particular example is viewed as an anomaly. With the smart usage of sparse projections, we could compute a one-tailed two-sample t-test between probabilities from histograms on projections with and without aspecific features. Casually speaking, if histograms using a particular feature have statistically higher anomaly scores than ones without it, we should have a closer look at it. Also, it has a higher time complexity than scoring samples because we need to evaluate every feature separately.

Of course, we should not consider this to be the ground truth for explaining the cause of an anomaly. That is a complicated process requiring more analysis with in-depth knowledge of data. LODA gives us only a good starting point to lead our investigation. If you want to see a full mathematical explanation read section **3.3 Explaining the cause of an anomaly**[Page 5, 1] in the original article.

```
>>> loda.score_features(X)
array([[ 3.57203657, -3.57203657],
       [ 1.15114953, -1.15114953],
       [ 1.8592136 , -1.8592136 ],
       [ 1.8592136 , -1.8592136 ],
       [ 2.29212856, -2.29212856],
       [-2.23606174,  2.23606174]])
```

To show this feature of LODA, we created a simple example using the Zoo dataset from the UCI Machine Learning Repository[3] (*LODA: Explaining the cause of an anomaly on Zoo dataset*). It contains different animal species and a summary of their characteristics (hair, feathers, eggs, milk, airborne, aquatic, etc.). We have chosen it because it's small, simple, and features are easily understandable (cat has for legs :) ...) First of all, we transform this dataset using UMAP (`umap.UMAP`)[2] to show in two dimensions.

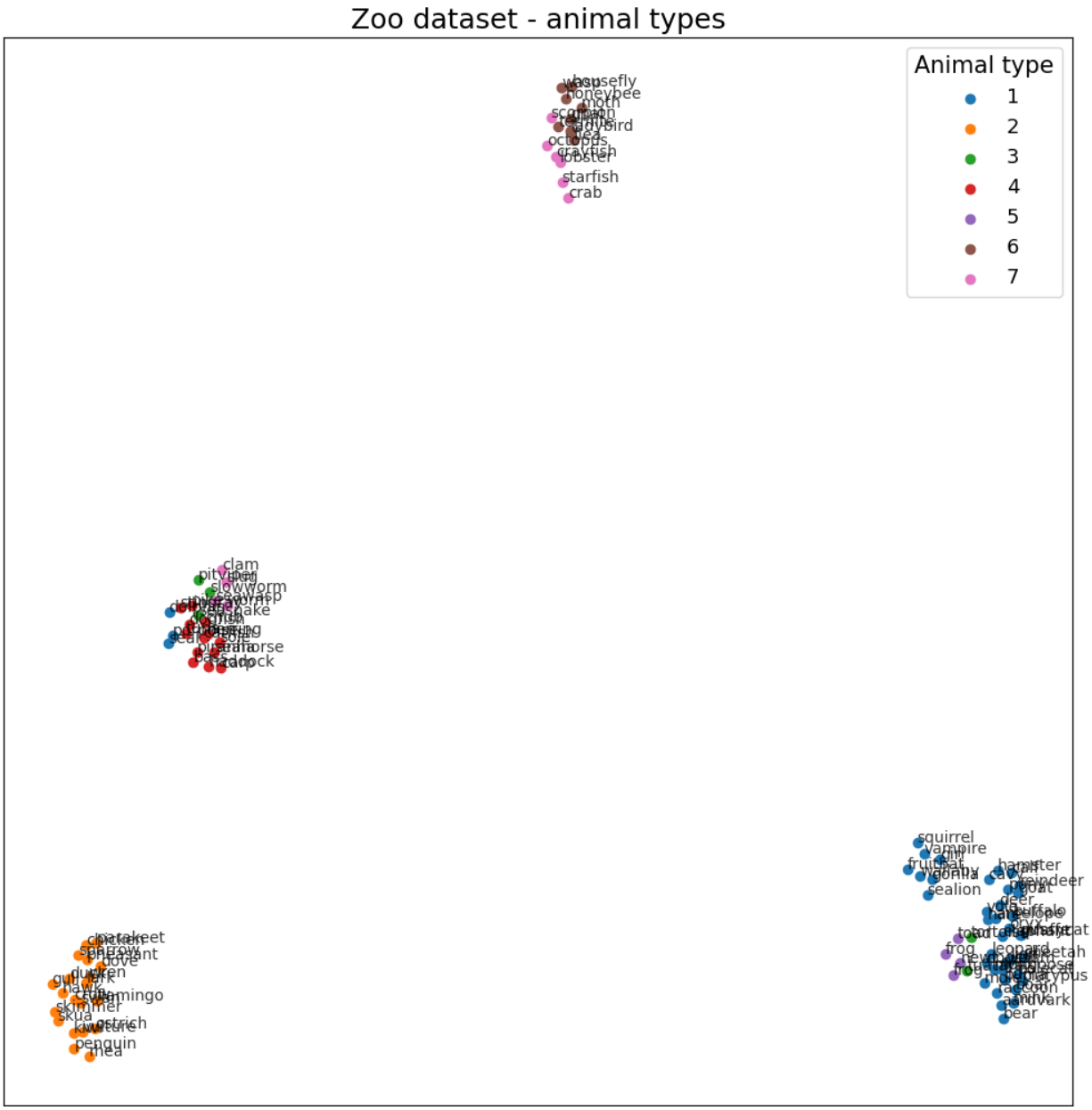
Once we get anomaly scores and importance of each feature, we could investigate further. We'll choose the five most anomalous animals. For example, we'll take a closer look at honeybee. It has a quite high score, and the most significant features are venomous (1.91), hair (1.55), breathes (1.28), and domestic (0.97). If we consider the composition of our dataset, there are no other venomous animals that are domestic, so it does seem right. We could find explanations like this for every other animal in the top five. Octopus has eight legs; sea wasp does have almost none of the features in the dataset, etc. So could we tell that these are the real reasons why these animals are unusual? Yes and no. Yes, this is why LODA sees them as anomalies considering our data, but without a review from a domain expert, we must be careful about such a statement. Also, consider the fact that this dataset is small, oversimplified, with just a limited number of features.

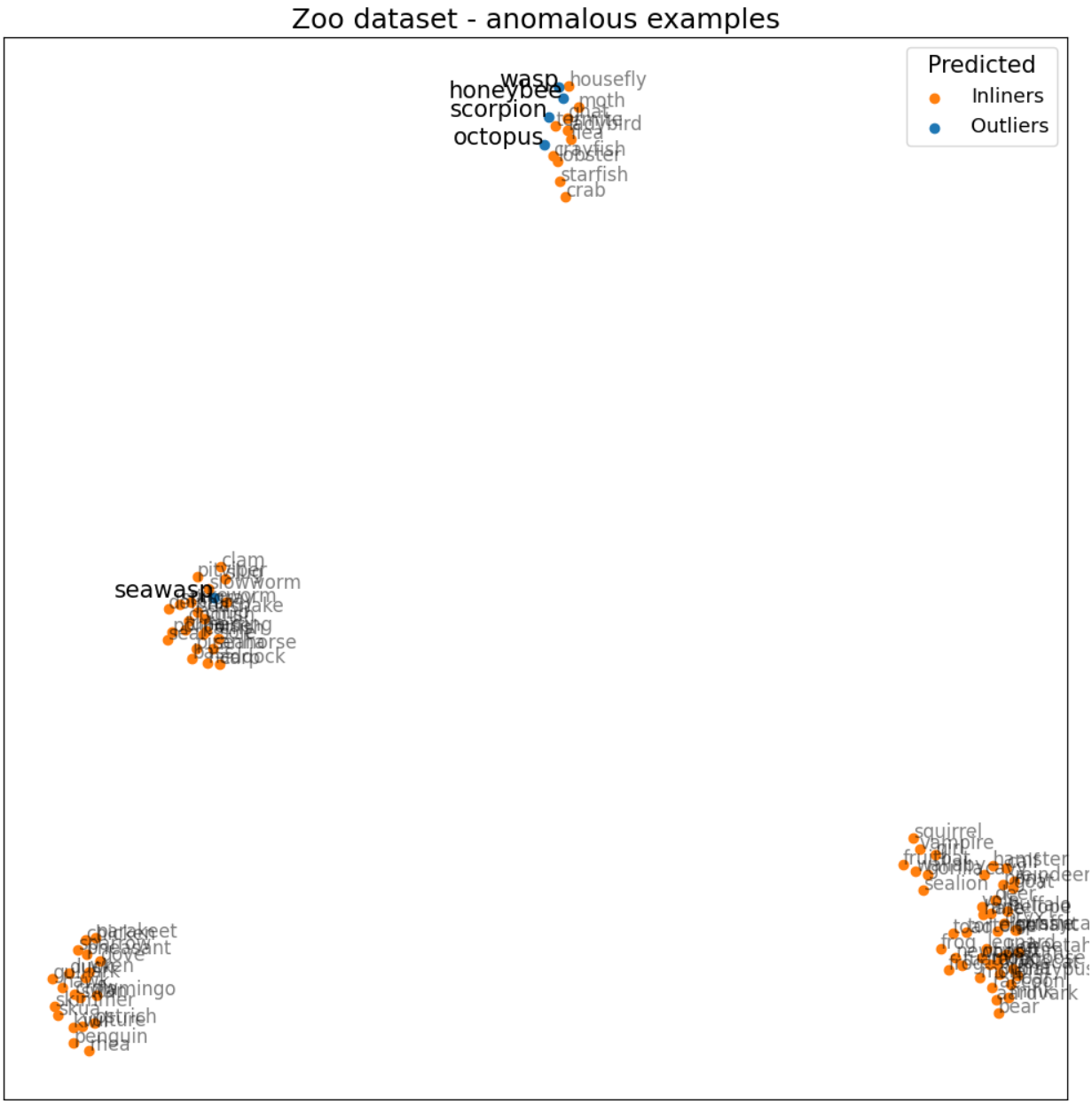
```
honeybee score: -4.576
        venomous 1 (1.91)
        hair 1 (1.55)
```

(continues on next page)

[3] Dua, D. and Graff, C. (2019). UCI Machine Learning Repository [http://archive.ics.uci.edu/ml]. Irvine, CA: University of California, School of Information and Computer Science. <https://archive.ics.uci.edu/ml/datasets/Zoo>

[2] McInnes, L., Healy, J., Saul, N., & Grossberger, L. (2018). UMAP: Uniform Manifold Approximation and Projection The Journal of Open Source Software, 3(29), 861. <https://github.com/lmcinnes/umap/>


Zoo dataset - animal types
Animal type
1
2
3
4
5
6
7
honeybee
starfish
crab
clam
slowworm
squirrel
vampire
sealion
bear
penguin


Zoo dataset - anomalous examples
Predicted
Inliners
Outliers
wasp
honeybee
scorpion
octopus
housefly
moth
flea
crayfish
lobster
starfish
crab
seawasp
clam
slowworm
squirrel
vampire
sealion
deer
bear
penguin
rhea

(continued from previous page)

```
        breathes 1 (1.28)
        domestic 1 (0.97)
octopus score: -5.763
        backbone 0 (3.06)
        legs 8 (1.79)
        feathers 0 (0.96)
        toothed 0 (0.80)
scorpion score: -5.007
        legs 8 (2.18)
        toothed 0 (1.23)
        domestic 0 (1.16)
        feathers 0 (0.82)
seawasp score: -4.898
        backbone 0 (1.78)
        milk 0 (1.06)
        toothed 0 (1.05)
        feathers 0 (0.80)
wasp score: -4.579
        feathers 0 (1.99)
        fins 0 (1.43)
        catsize 0 (1.41)
        breathes 1 (1.16)
```

To sum it up. LODA has a really powerful tool to explain the cause of an anomaly. It is more resource consuming than scoring samples. We should take a closer look at anomalies if we want to tell the real reason.

2.4 Using LODA on large datasets

In previous sections, we have seen that LODA is fully capable of getting similar results to more complex anomaly detection methods. Now we could take full advantage of LODA's low time and space complexity and use it on some more massive datasets.

We'll use Credit Card Fraud Detection dataset from the Machine Learning Group of Université Libre de Bruxelles[4] (it's available on Kaggle[5]). This dataset consists of credit card transactions with 492 frauds out of 284,807 transactions. Features are a byproduct of PCA transformation without any additional information due to confidentiality issues (*LODA: large data - Credit Card Fraud Detection dataset*).

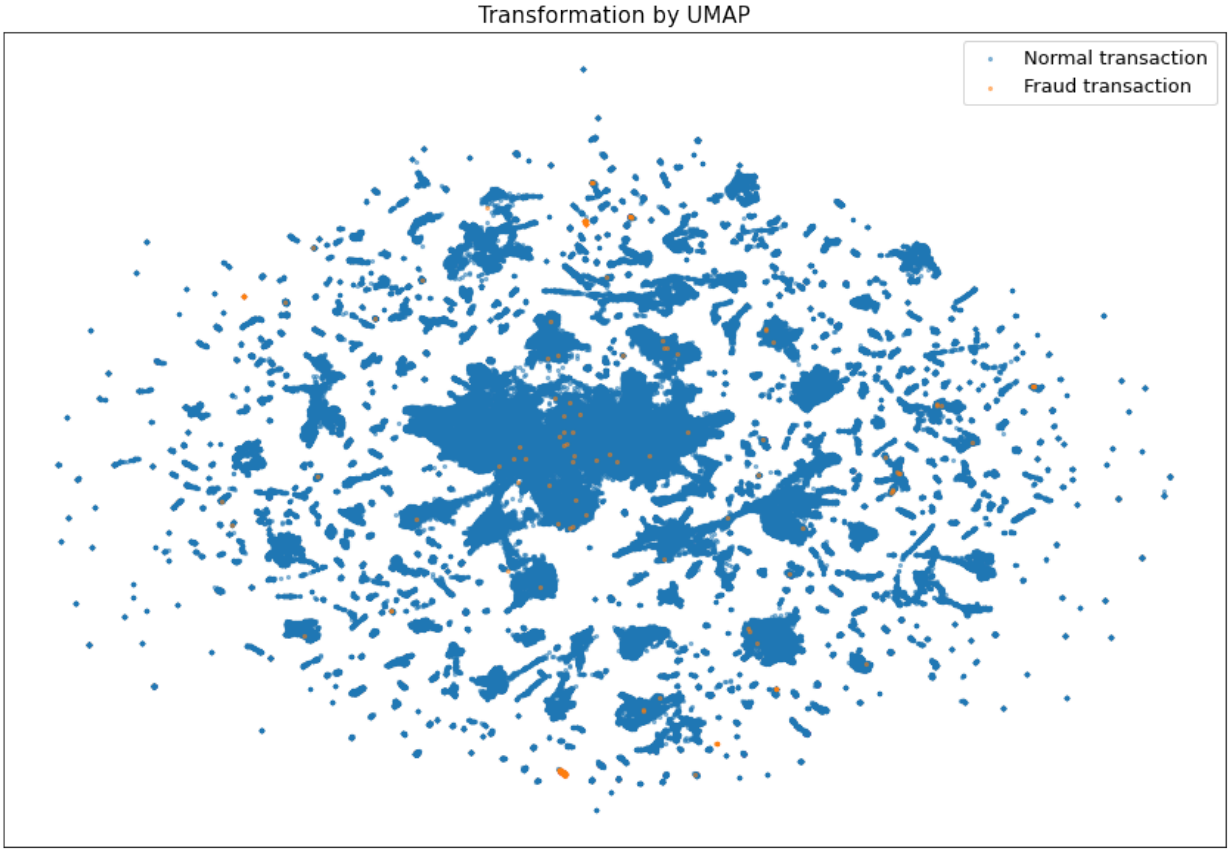
First of all, we'll visualize the entire dataset in low dimensional space to get an overview. We'll transform data using UMAP[?] and then plot results.

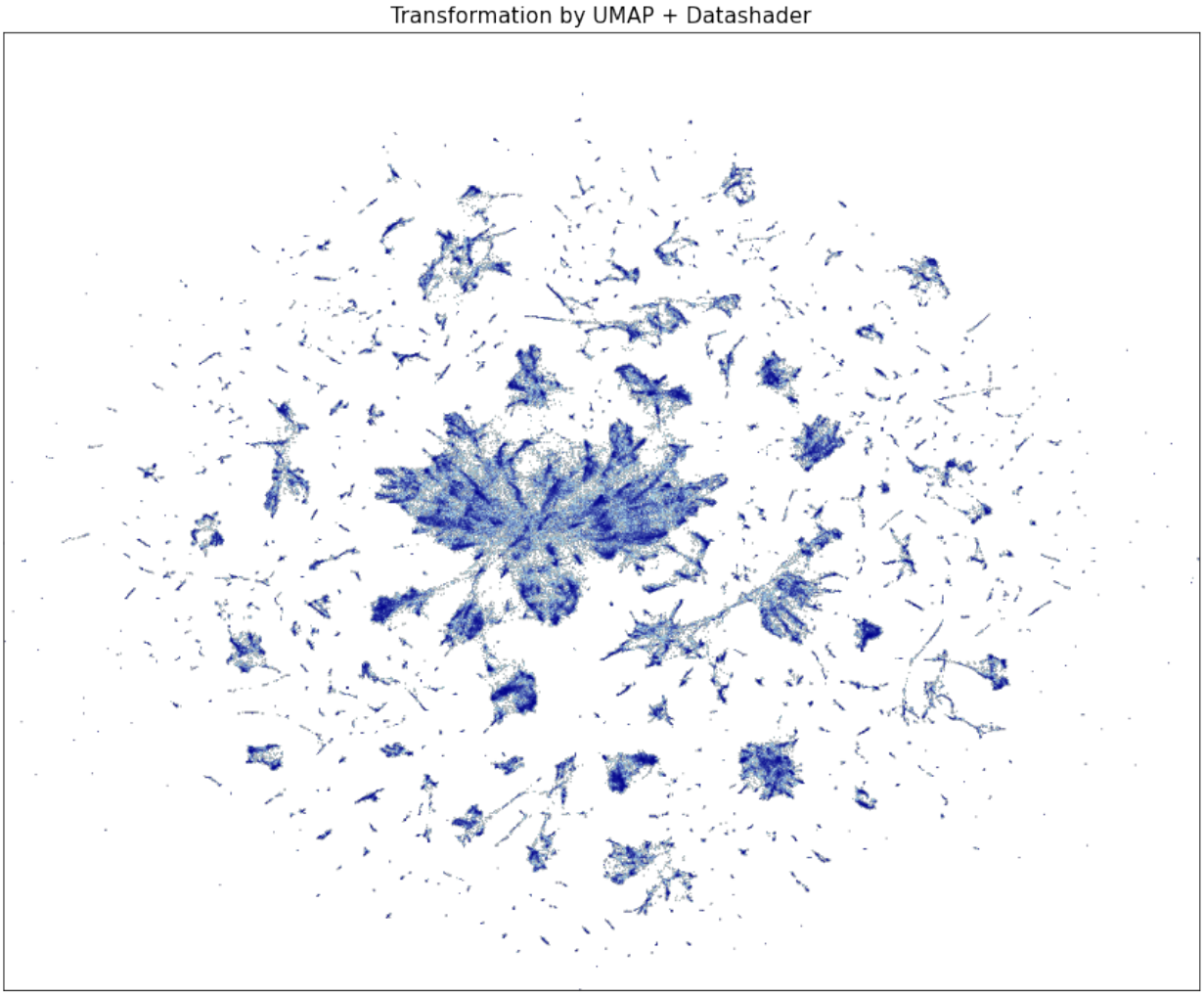
At first sight at this visualization, we could see some apparent clusters. Some of them even including a lot of fraud transactions. But this could be misleading due to significant overplotting. We'll try to solve this issue by using a more meaningful projection created by Datashader[6].

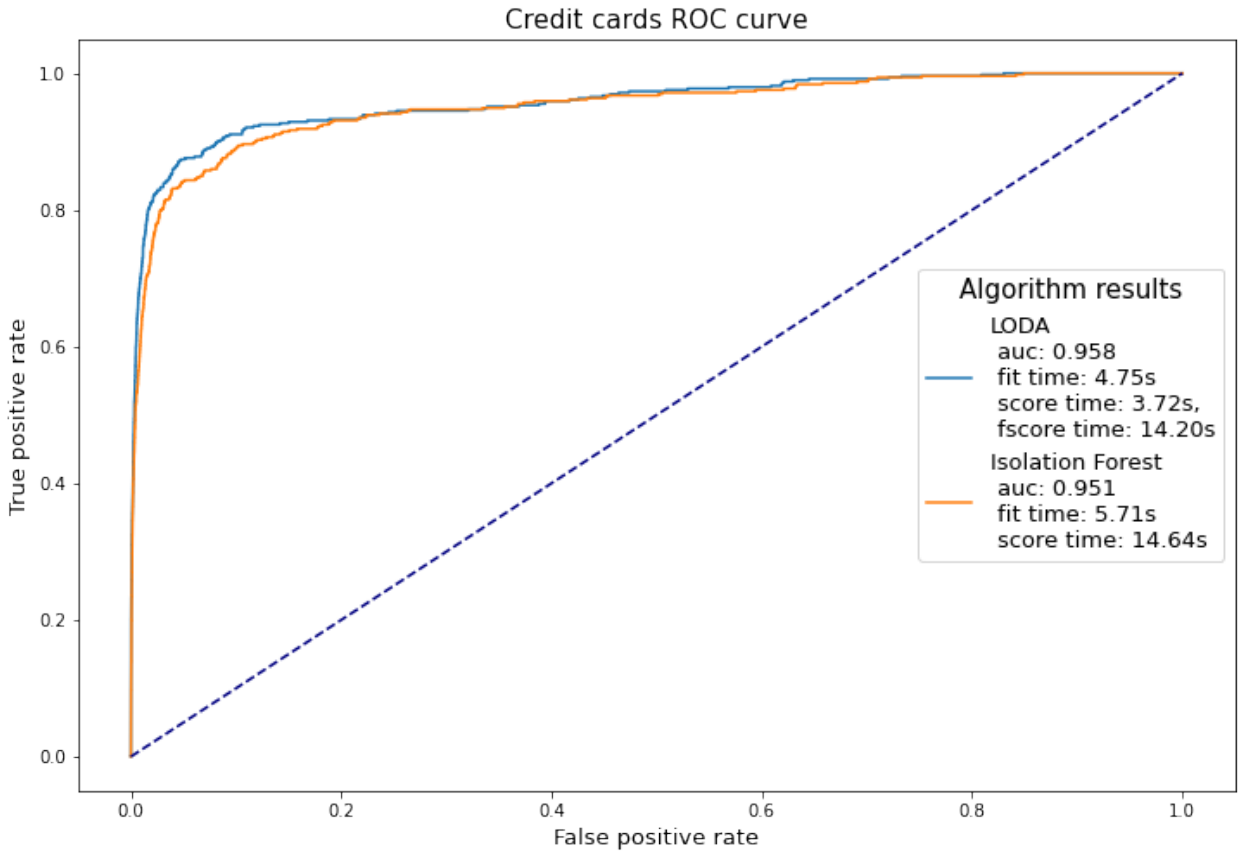
Once we have some clues about how the dataset looks, let's try to detect some fraud transactions. Because of its size, we'll use only LODA and isolation forest as anomaly detection methods. For comparing them, we'll use the area under the ROC curve.

As we can see, both methods performed very well (with the LODA slightly better). The low time complexity kicks in once we look at the training/predicting time for both detectors. It took LODA only 1/4 of the isolation forest's time to

[4] Machine Learning Group of Université Libre de Bruxelles <http://mlg.ulb.ac.be>
[5] Kaggle: Credit Card Fraud Detection <https://www.kaggle.com/mlg-ulb/creditcardfraud>
[6] HoloViz Datashader <https://datashader.org/>

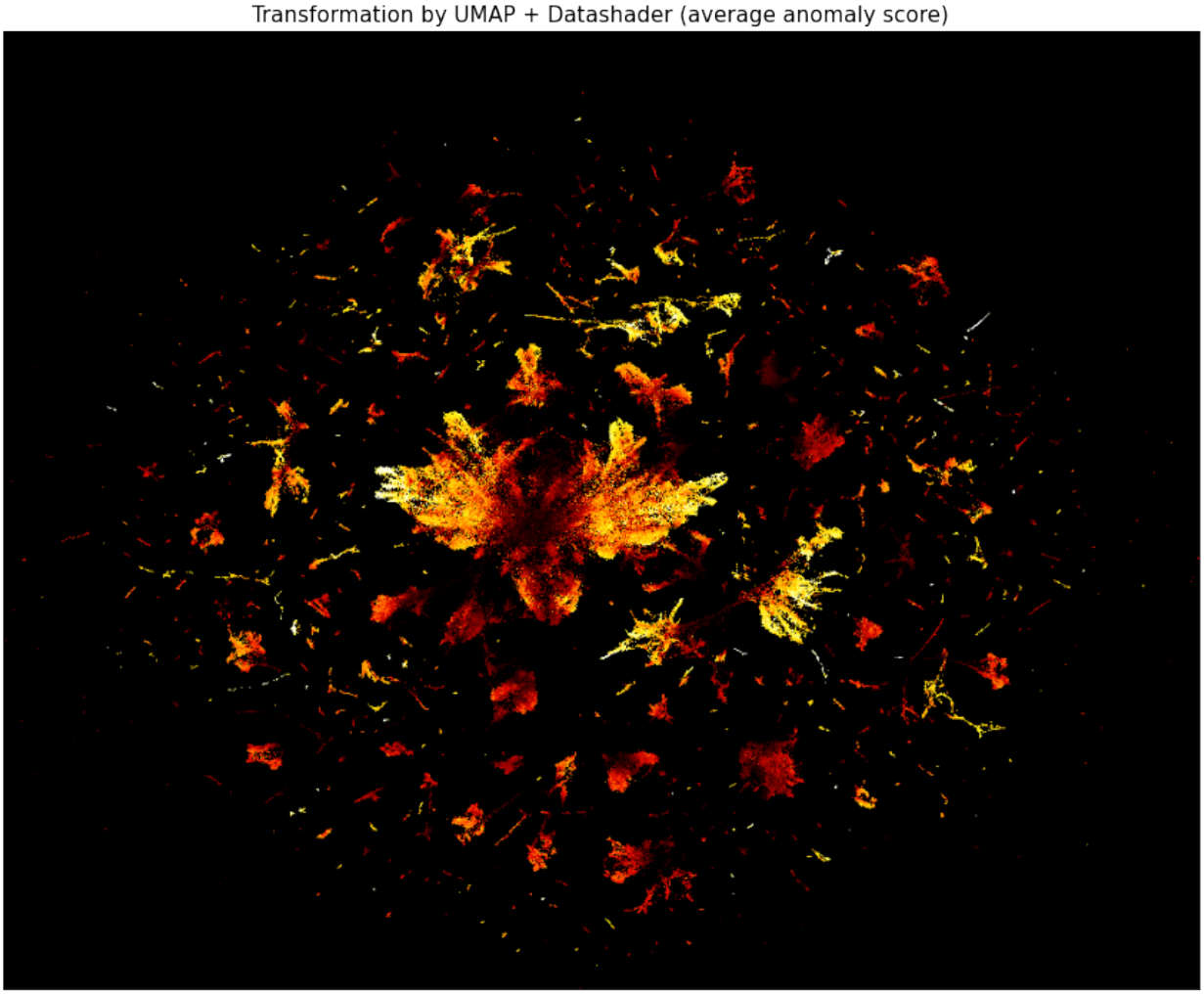

Transformation by UMAP
Normal transaction
Fraud transaction


Transformation by UMAP + Datashader


Credit cards ROC curve
True positive rate
False positive rate
0.0
0.2
0.4
0.6
0.8
1.0
Algorithm results
LODA
auc: 0.958
fit time: 4.75s
score time: 3.72s,
fscore time: 14.20s
Isolation Forest
auc: 0.951
fit time: 5.71s
score time: 14.64s

score 284,807 samples. It does not seem like such a big difference, but once we go up to millions of transactions, it could be a game-changer.

To finalize this section, let's make another plot using Datashader and anomaly scores from LODA.



2.5 References

2.6 Examples using `anlearn.loda.LODA`

- *LODA: large data - Credit Card Fraud Detection dataset*

CHAPTER

THREE

GALLERY

Gallery of examples from anlearn.

3.1 LODA: projections & histograms

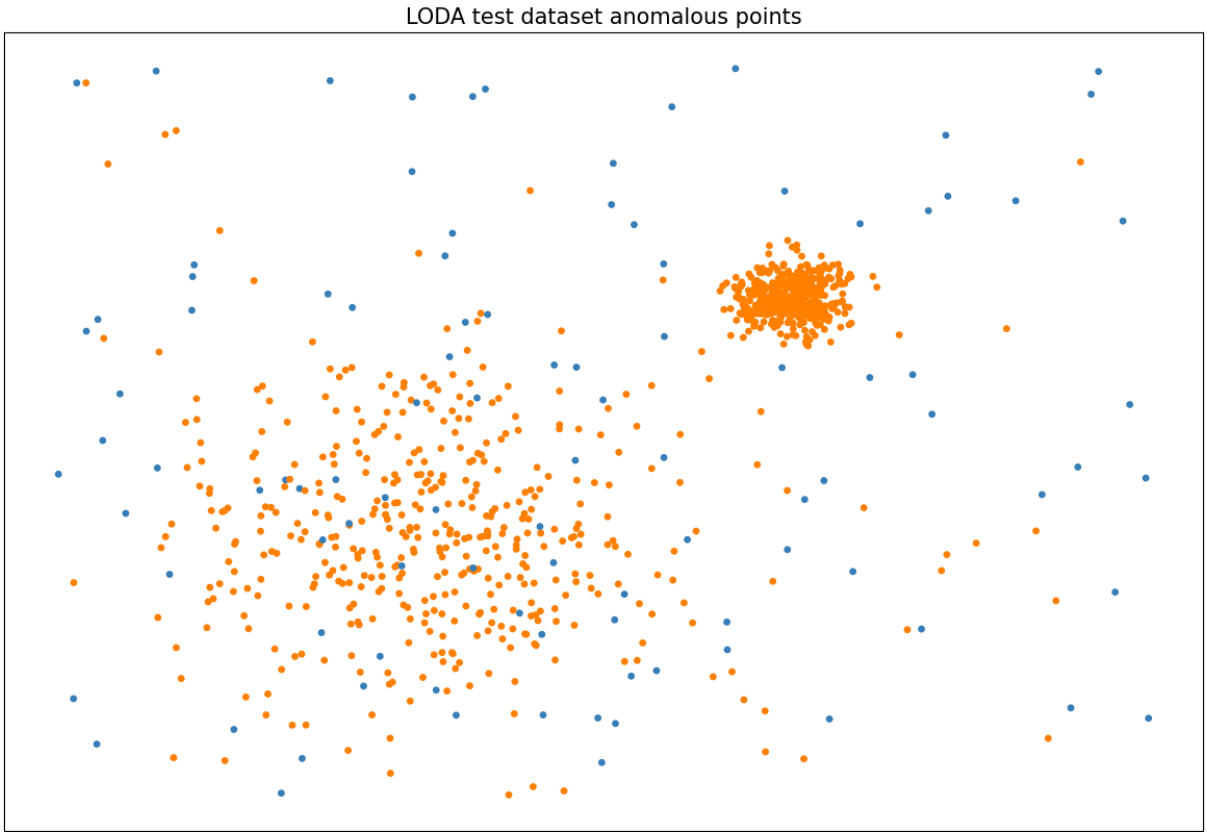
```
# Author: Ondrej Kurák kurak@gaussalgo.com
# License: LGPLv3+
import matplotlib.pyplot as plt
import numpy as np
from scipy.stats.kde import gaussian_kde
from sklearn.datasets import make_blobs

from anlearn.loda import LODA

rng = np.random.RandomState(42)

n_inliers = 900
n_outliers = 100
n_samples = n_inliers + n_outliers

n_features = 5

data = make_blobs(
    centers=[[-2] * n_features, [2] * n_features],
    cluster_std=[1.5, 0.3],
    random_state=42,
    n_samples=n_inliers,
    n_features=n_features,
)[0]

data = np.concatenate(
    [data, rng.uniform(low=-6, high=6, size=(n_outliers, n_features))]
)
```

```
loda = LODA(n_estimators=5, bins=100, random_state=42, q=0.1)
loda.fit(data)
predicted = loda.predict(data)

plt.figure(figsize=(12, 8))
```

(continues on next page)

(continued from previous page)

```
plt.subplot(111, aspect="auto")
plt.subplots_adjust(
    left=0.02, right=0.98, bottom=0.001, top=0.96, wspace=0.05, hspace=0.01
)
colors = np.array(["#377eb8", "#ff7f00"])
plt.scatter(data[:, 0], data[:, 1], s=15, color=colors[(predicted + 1) // 2])
plt.xticks(())
plt.yticks(())
plt.title("LODA test dataset anomalous points", fontsize=15)
plt.show()
```

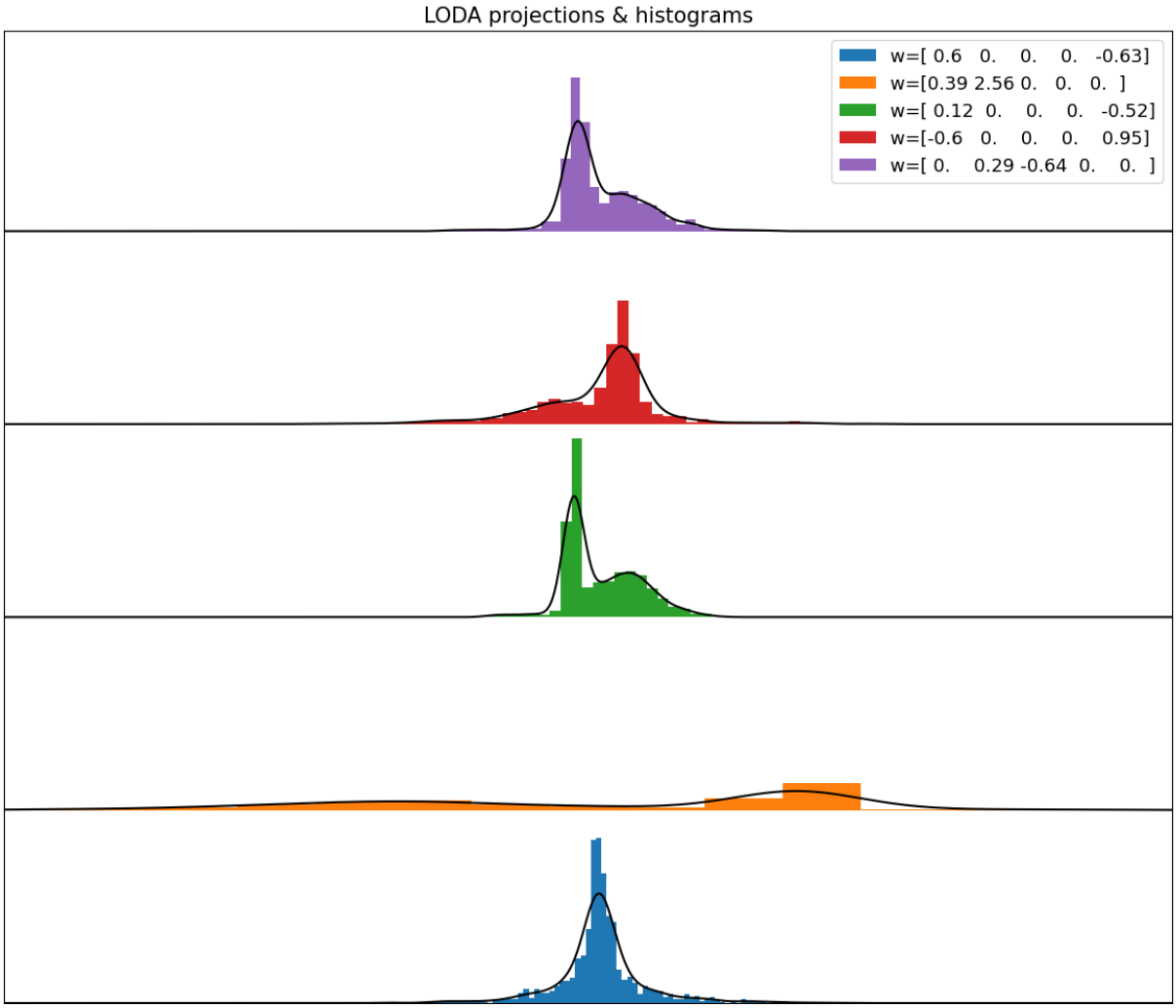



```
w_X = loda.projections_ @ data.T

labels = [f"w={x.round(2)}" for x in loda.projections_]
n_points = 500
bounds = (np.min(w_X), np.max(w_X))

plt.figure(figsize=(12, 10))
plt.subplot(111, aspect="auto")
plt.subplots_adjust(
    left=0.02, right=0.98, bottom=0.001, top=0.96, wspace=0.05, hspace=0.01
)
xx = np.linspace(*bounds, n_points)
yticks = []
for i, tmp in enumerate(zip(w_X, labels)):
```

(continues on next page)

(continued from previous page)

```
    points, label = tmp
    pdf = gaussian_kde(points)
    y = i + 0.1
    yticks.append(y)
    curve = pdf(xx)
    plt.hist(points, density=True, bottom=y, bins="auto", label=label)
    plt.plot(xx, curve + y, c="black")

plt.legend(fontsize=13)
plt.title("LODA projections & histograms", fontsize=15)
plt.xlim(bounds)
plt.yticks(())
plt.show()


# sphinx_gallery_thumbnail_number = 2
```



Total running time of the script: (0 minutes 0.494 seconds)

3.2 Comparison of scikit-learn anomaly detection methods and LODA

Comparison of LODA[1] and scikit-learn anomaly detection methods[2].

```
# Original Authors: Alexandre Gramfort <alexandre.gramfort@inria.fr>
#                   Albert Thomas <albert.thomas@telecom-paristech.fr>
# Edited by: Andrea Rozhoňoná <rozhonova@gaussalgo.com>
#            Ondrej Kurák <kurak@gaussalgo.com>
# License: BSD 3 clause
import time

import matplotlib
import matplotlib.pyplot as plt
import numpy as np
from sklearn import svm
from sklearn.covariance import EllipticEnvelope
from sklearn.datasets import make_blobs, make_moons
from sklearn.ensemble import IsolationForest
from sklearn.neighbors import LocalOutlierFactor

from anlearn.loda import LODA

print(__doc__)

matplotlib.rcParams["contour.negative_linestyle"] = "solid"

# Example settings
n_samples = 300
outliers_fraction = 0.15
n_outliers = int(outliers_fraction * n_samples)
n_inliers = n_samples - n_outliers

# define outlier/anomaly detection methods to be compared
anomaly_algorithms = [
    ("Robust covariance", EllipticEnvelope(contamination=outliers_fraction)),
    ("One-Class SVM", svm.OneClassSVM(nu=outliers_fraction, kernel="rbf", gamma=0.1)),
    (
        "Isolation Forest",
        IsolationForest(contamination=outliers_fraction, random_state=42),
    ),
    (
        "Local Outlier Factor",
        LocalOutlierFactor(n_neighbors=35, contamination=outliers_fraction),
    ),
    ("LODA", LODA(n_estimators=100, q=outliers_fraction, bins=10, random_state=42)),
]

# Define datasets
blobs_params = dict(random_state=0, n_samples=n_inliers, n_features=2)
datasets = [
```

(continues on next page)

[1] Pevný, T. Loda: Lightweight on-line detector of anomalies. Mach Learn 102, 275–304 (2016). <https://doi.org/10.1007/s10994-015-5521-0>

[2] Sckikit-learn Comparing anomaly detection algorithms for outlier detection on toy datasets https://scikit-learn.org/stable/auto_examples/miscellaneous/plot_anomaly_comparison.html#sphx-glr-auto-examples-miscellaneous-plot-anomaly-comparison-py

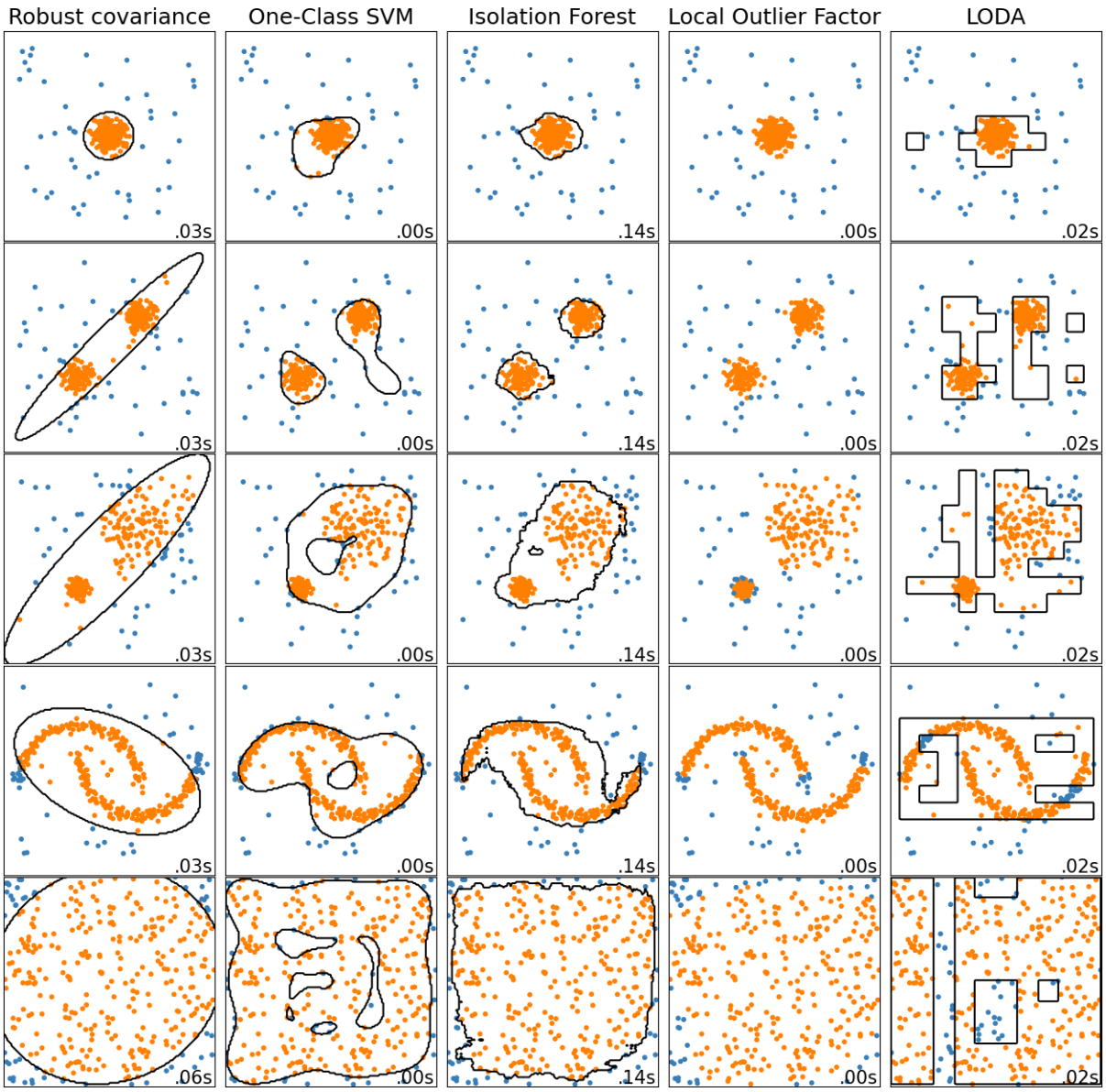
(continued from previous page)

```
    make_blobs(centers=[[0, 0], [0, 0]], cluster_std=0.5, **blobs_params)[0],
    make_blobs(centers=[[2, 2], [-2, -2]], cluster_std=[0.5, 0.5], **blobs_params)[0],
    make_blobs(centers=[[2, 2], [-2, -2]], cluster_std=[1.5, 0.3], **blobs_params)[0],
    4.0
    * (
        make_moons(n_samples=n_samples, noise=0.05, random_state=0)[0]
        - np.array([0.5, 0.25])
    ),
    14.0 * (np.random.RandomState(42).rand(n_samples, 2) - 0.5),
]

# Compare given classifiers under given settings
xx, yy = np.meshgrid(np.linspace(-7, 7, 150), np.linspace(-7, 7, 150))

plt.figure(figsize=(len(anomaly_algorithms) * 2 + 3, 12.5))
plt.subplots_adjust(
    left=0.02, right=0.98, bottom=0.001, top=0.96, wspace=0.05, hspace=0.01
)

plot_num = 1
rng = np.random.RandomState(42)

for i_dataset, X in enumerate(datasets):
    # Add outliers
    X = np.concatenate([X, rng.uniform(low=-6, high=6, size=(n_outliers, 2))], axis=0)

    for name, algorithm in anomaly_algorithms:
        t0 = time.time()
        algorithm.fit(X)
        t1 = time.time()
        plt.subplot(len(datasets), len(anomaly_algorithms), plot_num)
        if i_dataset == 0:
            plt.title(name, size=18)

        # fit the data and tag outliers
        if name == "Local Outlier Factor":
            y_pred = algorithm.fit_predict(X)
        else:
            y_pred = algorithm.fit(X).predict(X)

        # plot the levels lines and the points
        if name != "Local Outlier Factor":  # LOF does not implement predict
            Z = algorithm.predict(np.c_[xx.ravel(), yy.ravel()])
            Z = Z.reshape(xx.shape)
            plt.contour(xx, yy, Z, levels=[0], linewidths=1.5, colors="black")

        colors = np.array(["#377eb8", "#ff7f00"])
        plt.scatter(X[:, 0], X[:, 1], s=10, color=colors[(y_pred + 1) // 2])

        plt.xlim(-7, 7)
        plt.ylim(-7, 7)
        plt.xticks(())
```

(continues on next page)

(continued from previous page)

```
        plt.yticks(())
        plt.text(
            0.99,
            0.01,
            ("%.2fs" % (t1 - t0)).lstrip("0"),
            transform=plt.gca().transAxes,
            size=15,
            horizontalalignment="right",
        )
        plot_num += 1
```



3.2.1 References

Total running time of the script: (0 minutes 4.934 seconds)

3.3 LODA: large data - Credit Card Fraud Detection dataset

In previous sections, we have seen that LODA[1] is fully capable of getting similar results to more complex anomaly detection methods. Now we could take full advantage of LODA's low time and space complexity and use it on some more massive datasets.

We'll use Credit Card Fraud Detection dataset from the Machine Learning Group of Université Libre de Bruxelles[4] (it's available on Kaggle[5]). This dataset consists of credit card transactions with 492 frauds out of 284,807 transactions. Features are a byproduct of PCA transformation without any additional information due to confidentiality issues.

First of all, we'll visualize the entire dataset in low dimensional space to get an overview. We'll transform data using UMAP[2] and then plot results.

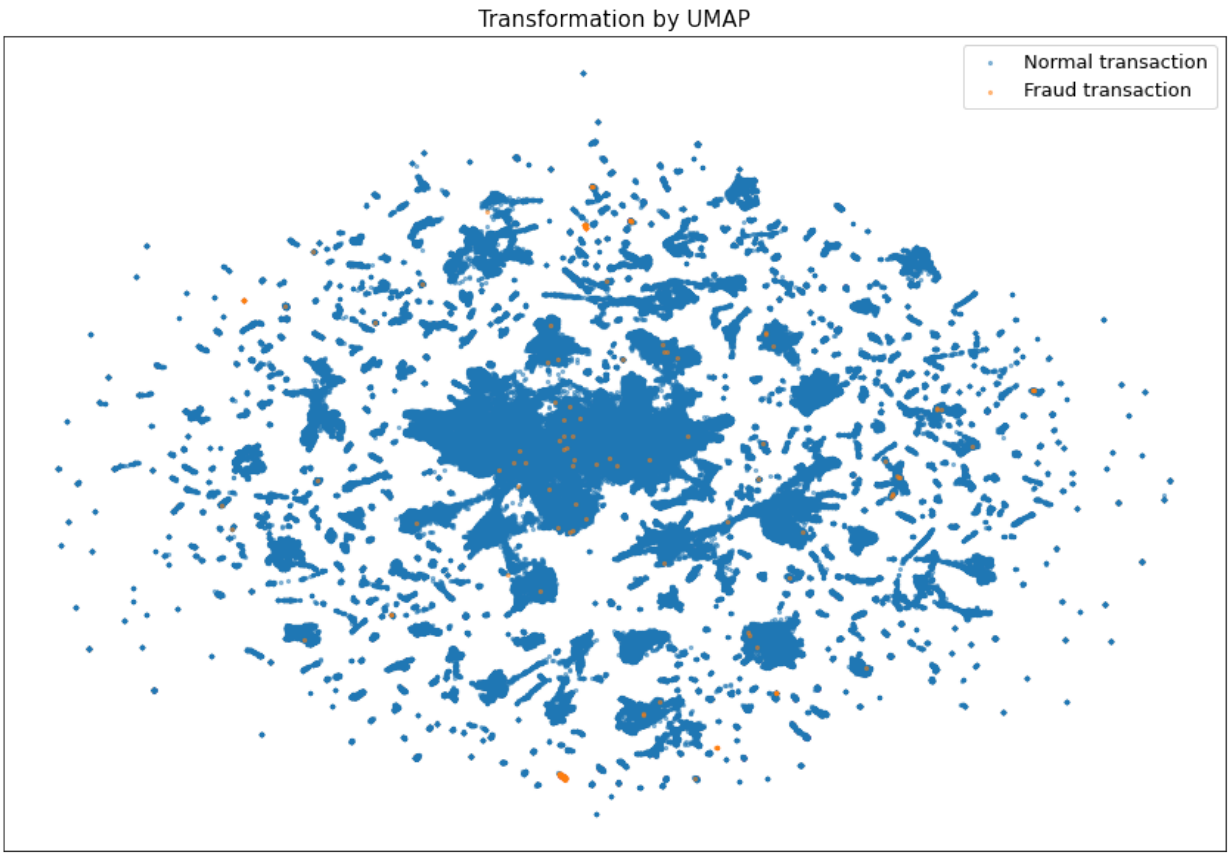
```
# Author: Ondrej Kurák kurak@gaussalgo.com
# License: LGPLv3+
import time

import datashader as ds
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from colorcet import fire
from datashader import transfer_functions as tf
from sklearn.ensemble import IsolationForest
from sklearn.metrics import auc, roc_curve
from umap import UMAP

from anlearn.loda import LODA

frame = pd.read_csv("../datasets/creditcard.csv")

X = np.arcsinh(frame.values[:, 1:-1])
y = frame["Class"].values


umap = UMAP(random_state=42)

# This could take ~30 min on Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz
# transformed = umap.fit_transform(X)
# with open("../datasets/transformed.npy", "wb") as out:
#     np.save(out, transformed)
transformed = np.load("../datasets/transformed.npy")
```

(continues on next page)

[1] Pevný, T. Loda: Lightweight on-line detector of anomalies. Mach Learn 102, 275–304 (2016). <https://doi.org/10.1007/s10994-015-5521-0>
[4] Machine Learning Group of Université Libre de Bruxelles <http://mlg.ulb.ac.be>
[5] Kaggle: Credit Card Fraud Detection <https://www.kaggle.com/mlg-ulb/creditcardfraud>
[2] McInnes, L., Healy, J., Saul, N., & Grossberger, L. (2018). UMAP: Uniform Manifold Approximation and Projection The Journal of Open Source Software, 3(29), 861. <https://github.com/lmcinnes/umap/>

(continued from previous page)

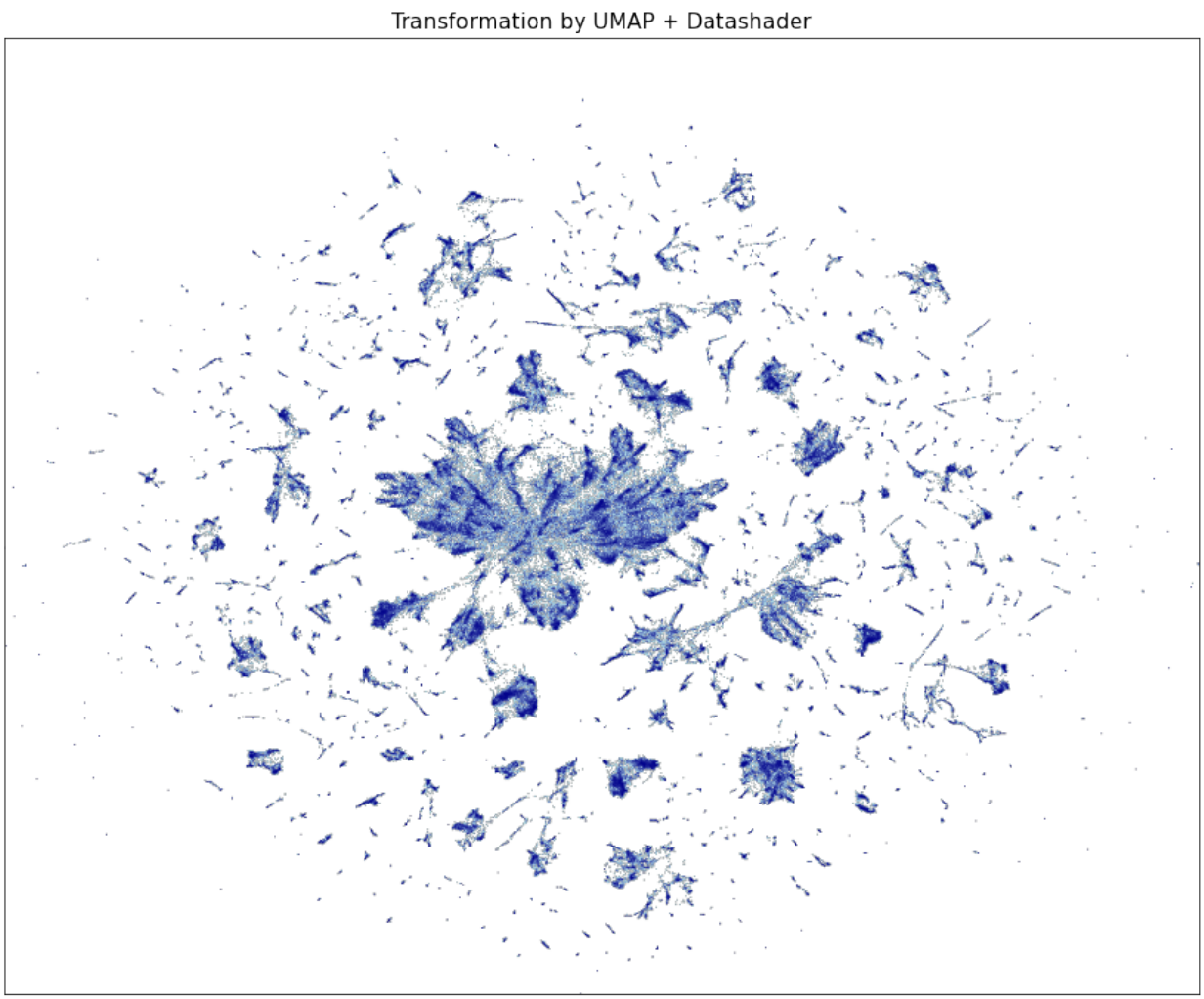
```
plt.figure(figsize=(12, 8))
plt.subplot(111, aspect="auto")
plt.subplots_adjust(
    left=0.02, right=0.98, bottom=0.001, top=0.96, wspace=0.05, hspace=0.01
)

for index, label in enumerate(("Normal transaction", "Fraud transaction")):
    plt.scatter(
        transformed[:, 0][y == index],
        transformed[:, 1][y == index],
        s=5,
        label=label,
        alpha=0.5,
    )

plt.legend(fontsize=13)
plt.xticks(())
plt.yticks(())

plt.title("Transformation by UMAP", fontsize=15)
plt.show()
```



At first sight at this visualization, we could see some apparent clusters. Some of them even including a lot of fraud transactions. But this could be misleading due to significant overplotting. We'll try to solve this issue by using a more

meaningful projection created by Datashader[6].

```
shader_data = pd.DataFrame(
    transformed,
    columns=["x", "y"],
)

agg = ds.Canvas(plot_width=1000, plot_height=800).points(shader_data, "x", "y")

img = tf.shade(agg, name="Transformation by UMAP + Datashader")

plt.figure(figsize=(15, 15))
plt.subplot(111, aspect="auto")
plt.subplots_adjust(top=0.96, wspace=0.05, hspace=0.01)
plt.imshow(img.to_pil())
plt.title("Transformation by UMAP + Datashader", fontsize=15)
plt.xticks(())
plt.yticks(())
plt.show()
```



[6] HoloViz Datashader <https://datashader.org/>

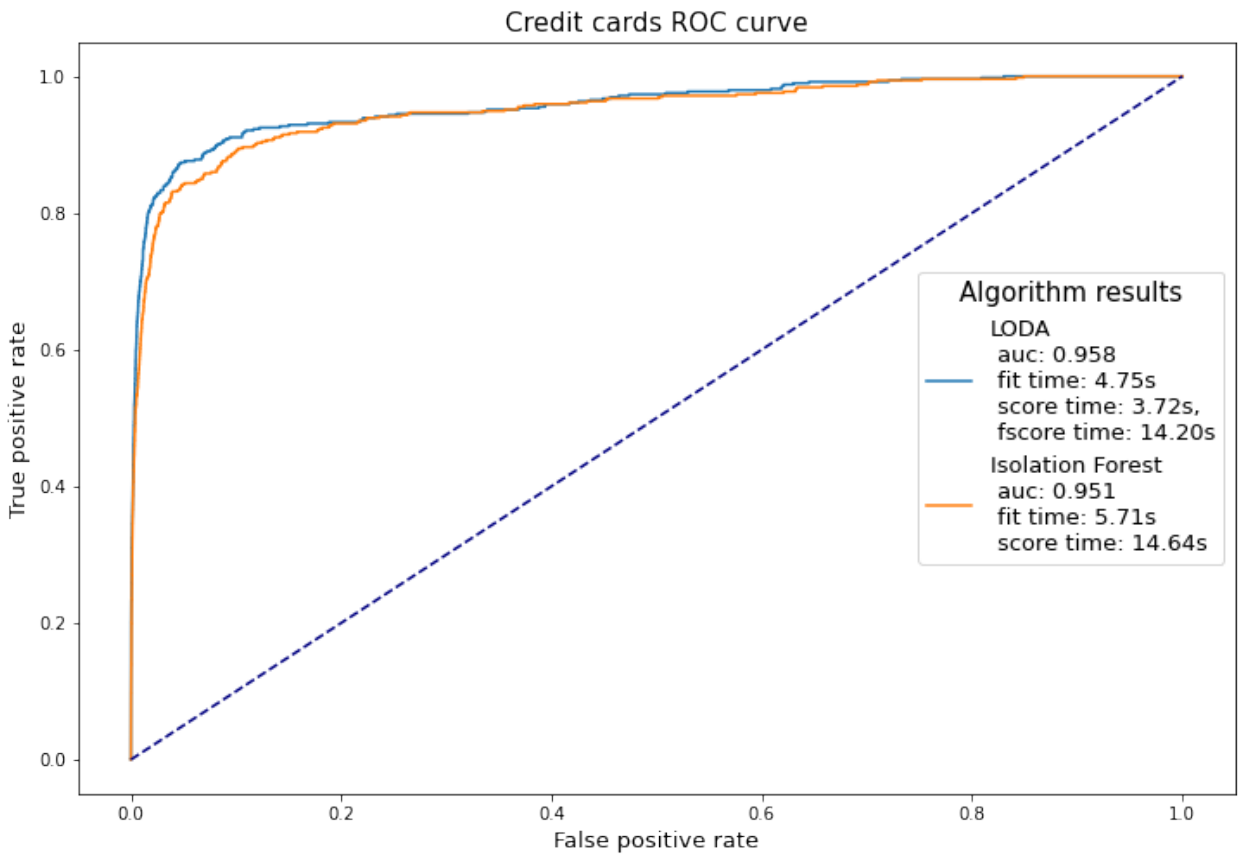
Once we have some clues about how the dataset looks, let's try to detect some fraud transactions. Because of its size, we'll use only LODA and isolation forest as anomaly detection methods. For comparing them, we'll use the area under the ROC curve.

```
times = {}

loda = LODA(n_estimators=100, random_state=42, bins=100)

start_time = time.monotonic()
loda.fit(X)
times["loda.fit"] = time.monotonic() - start_time

start_time = time.monotonic()
loda_scores = loda.score_samples(X)
times["loda.score_samples"] = time.monotonic() - start_time


start_time = time.monotonic()
feature_scores = loda.score_features(X)
times["loda.score_features"] = time.monotonic() - start_time

isoforest = IsolationForest(n_estimators=100, random_state=42)

start_time = time.monotonic()
isoforest.fit(X)
times["isoforest.fit"] = time.monotonic() - start_time

start_time = time.monotonic()
iso_scores = isoforest.score_samples(X)
times["isoforest.score_samples"] = time.monotonic() - start_time

loda_fpr, loda_tpr, _ = roc_curve(y, -loda_scores)
loda_auc = auc(loda_fpr, loda_tpr)

isof_fpr, isof_tpr, _ = roc_curve(y, -iso_scores)
isof_auc = auc(isof_fpr, isof_tpr)


plt.figure(figsize=(12, 8))
plt.subplot(111, aspect="auto")
plt.subplots_adjust(
    left=0.02, right=0.98, bottom=0.001, top=0.96, wspace=0.05, hspace=0.01
)

plt.plot(
    loda_fpr,
    loda_tpr,
    label=f"""LODA
 auc: {loda_auc:.3f}
 fit time: {times["loda.fit"]:.2f}s
 score time: {times["loda.score_samples"]:.2f}s,
 fscore time: {times["loda.score_features"]:.2f}s""",
)
```

(continues on next page)

(continued from previous page)

```
plt.plot(
    isof_fpr,
    isof_tpr,
    label=f"""Isolation Forest
 auc: {isof_auc:.3f}
 fit time: {times["isoforest.fit"]:.2f}s
 score time: {times["isoforest.score_samples"]:.2f}s""",
)

plt.plot([0, 1], [0, 1], color="navy", linestyle="--")

plt.title("Credit cards ROC curve", fontsize=15)
plt.legend(
    title="Algorithm results", title_fontsize=15, fontsize=13, loc="center right"
)
plt.xlabel("False positive rate", fontsize=13)
plt.ylabel("True positive rate", fontsize=13)

plt.show()
```



As we can see, both methods performed very well (with the LODA slightly better). The low time complexity kicks in once we look at the training/predicting time for both detectors. It took LODA only 1/4 of the isolation forest's time to score 284,807 samples. It does not seem like such a big difference, but once we go up to millions of transactions, it could be a game-changer.

To finalize this section, let's make another plot using Datashader and anomaly scores from LODA.

```
shader_data = pd.DataFrame(
    np.hstack([transformed, loda_scores[:, np.newaxis]]),
    columns=["x", "y", "anomaly_score"],
)

agg = ds.Canvas(plot_width=1000, plot_height=800).points(
    shader_data, "x", "y", ds.mean("anomaly_score")
)

img = tf.shade(
    agg, cmap=fire, name="Transformation by UMAP + Datashader (average anomaly score)"
)

img = tf.set_background(img, "black")

plt.figure(figsize=(15, 15))
plt.subplot(111, aspect="auto")
plt.subplots_adjust(
    left=0.02, right=0.98, bottom=0.001, top=0.96, wspace=0.05, hspace=0.01
)
plt.imshow(img.to_pil())
plt.title("Transformation by UMAP + Datashader (average anomaly score)", fontsize=15)
plt.xticks(())
plt.yticks(())
plt.show()
```

3.3.1 References

Total running time of the script: (0 minutes 0.000 seconds)

3.4 LODA: Explaining the cause of an anomaly on Zoo dataset

The knowledge that an example is anomalous just the first part of the whole anomaly detection pipeline. Without investigating further, I would consider this information almost useless. Lucky for us, LODA has a built-in way to get a little bit more information about why a particular example is viewed as an anomaly. With the smart usage of sparse projections, we could compute a one-tailed two-sample t-test between probabilities from histograms on projections with and without aspecific features. Casually speaking, if histograms using a particular feature have statistically higher anomaly scores than ones without it, we should have a closer look at it. Also, it has a higher time complexity than scoring samples because we need to evaluate every feature separately.

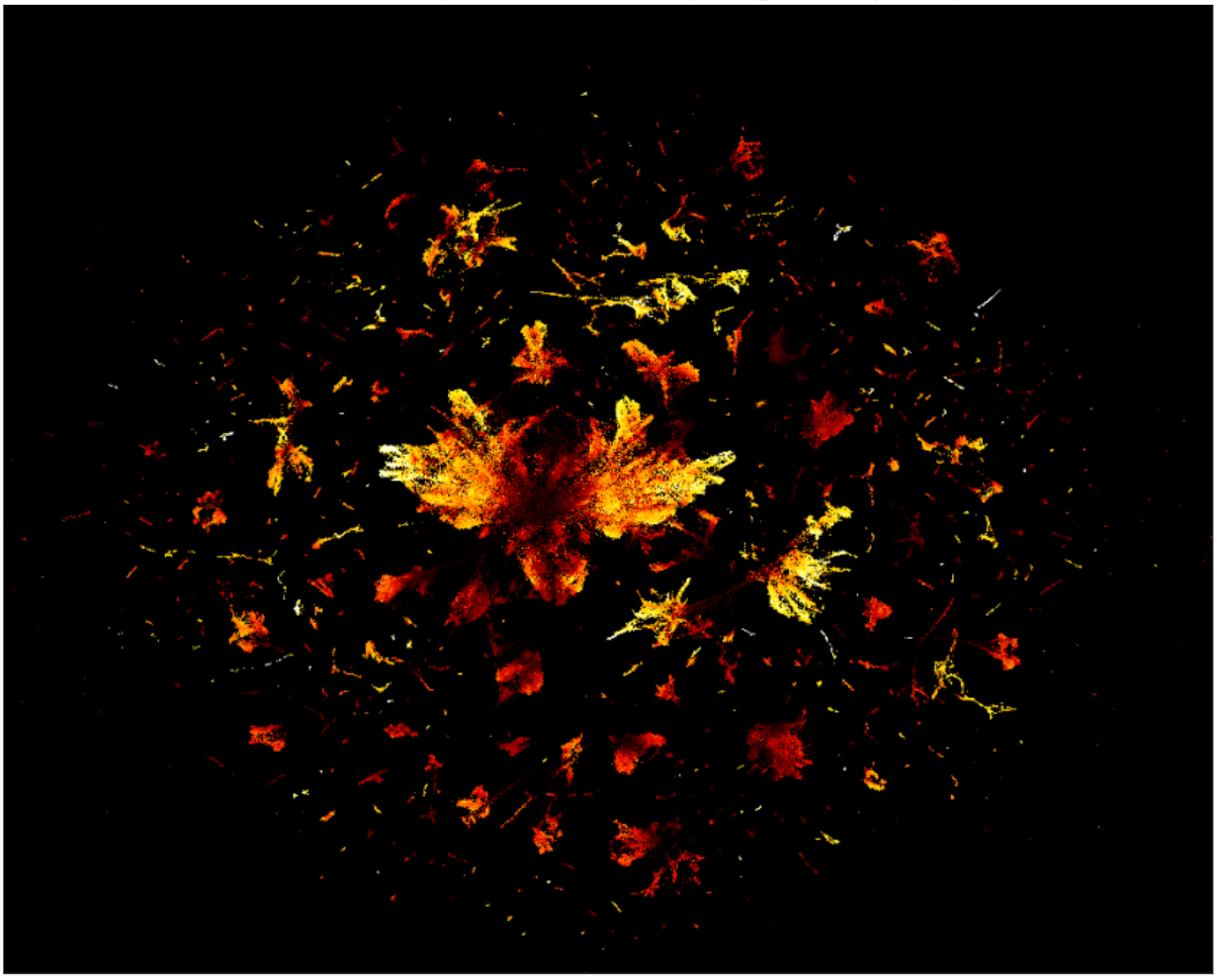
Of course, we should not consider this to be the ground truth for explaining the cause of an anomaly. That is a complicated process requiring more analysis with in-depth knowledge of data. LODA gives us only a good starting point to lead our investigation. If you want to see a full mathematical explanation read section **3.3 Explaining the cause of an anomaly**[1] in the original article.

To show this feature of LODA, we created a simple example using the Zoo dataset from the UCI Machine Learning Repository[3]. It contains different animal species and a summary of their characteristics (hair, feathers, eggs, milk,

[1] Pevný, T. Loda: Lightweight on-line detector of anomalies. Mach Learn 102, 275–304 (2016). <https://doi.org/10.1007/s10994-015-5521-0>

[3] Dua, D. and Graff, C. (2019). UCI Machine Learning Repository [http://archive.ics.uci.edu/ml]. Irvine, CA: University of California, School of Information and Computer Science. <https://archive.ics.uci.edu/ml/datasets/Zoo>

Transformation by UMAP + Datashader (average anomaly score)



airborne, aquatic, etc.). We have chosen it because it's small, simple, and features are easily understandable (cat has for legs :) ...) First of all, we transform this dataset using UMAP (`umap.UMAP`)[2] to show in two dimensions.

```
# Author: Ondrej Kurák kurak@gaussalgo.com
# License: LGPLv3+

import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from umap import UMAP

from anlearn.loda import LODA

frame = pd.read_csv(
    "https://raw.githubusercontent.com/sharmaroshan/Zoo-Dataset/master/zoo.csv",
)

frame.set_index("animal_name", inplace=True)

print(frame)
```

Out:

```
             hair  feathers  eggs  milk  ...  tail  domestic  catsize  class_type
animal_name                              ...
aardvark        1         0     0     1  ...     0         0        1           1
antelope        1         0     0     1  ...     1         0        1           1
bass            0         0     1     0  ...     1         0        0           4
bear            1         0     0     1  ...     0         0        1           1
boar            1         0     0     1  ...     1         0        1           1
...           ...       ...   ...   ...  ...   ...       ...      ...         ...
wallaby         1         0     0     1  ...     1         0        1           1
wasp            1         0     1     0  ...     0         0        0           6
wolf            1         0     0     1  ...     1         0        1           1
worm            0         0     1     0  ...     0         0        0           7
wren            0         1     1     0  ...     1         0        0           2

[101 rows x 17 columns]
```

```
# !cat ../datasets/zoo.names

# 1. Title: Zoo database

# 2. Source Information
#    -- Creator: Richard Forsyth
#    -- Donor: Richard S. Forsyth
#              8 Grosvenor Avenue
#              Mapperley Park
#              Nottingham NG3 5DX
#              0602-621676
#    -- Date: 5/15/1990
```

(continues on next page)

[2] McInnes, L., Healy, J., Saul, N., & Grossberger, L. (2018). UMAP: Uniform Manifold Approximation and Projection The Journal of Open Source Software, 3(29), 861. <https://github.com/lmcinnes/umap/>

(continued from previous page)

```
# 3. Past Usage:
#    -- None known other than what is shown in Forsyth's PC/BEAGLE User's Guide.

# 4. Relevant Information:
#    -- A simple database containing 17 Boolean-valued attributes.  The "type"
#       attribute appears to be the class attribute.  Here is a breakdown of
#       which animals are in which type: (I find it unusual that there are
#       2 instances of "frog" and one of "girl"!)

#       Class# Set of animals:
#       ====== ===================================================================
#            1 (41) aardvark, antelope, bear, boar, buffalo, calf,
#                   cavy, cheetah, deer, dolphin, elephant,
#                   fruitbat, giraffe, girl, goat, gorilla, hamster,
#                   hare, leopard, lion, lynx, mink, mole, mongoose,
#                   opossum, oryx, platypus, polecat, pony,
#                   porpoise, puma, pussycat, raccoon, reindeer,
#                   seal, sealion, squirrel, vampire, vole, wallaby,wolf
#            2 (20) chicken, crow, dove, duck, flamingo, gull, hawk,
#                   kiwi, lark, ostrich, parakeet, penguin, pheasant,
#                   rhea, skimmer, skua, sparrow, swan, vulture, wren
#            3 (5)  pitviper, seasnake, slowworm, tortoise, tuatara
#            4 (13) bass, carp, catfish, chub, dogfish, haddock,
#                   herring, pike, piranha, seahorse, sole, stingray, tuna
#            5 (4)  frog, frog, newt, toad
#            6 (8)  flea, gnat, honeybee, housefly, ladybird, moth, termite, wasp
#            7 (10) clam, crab, crayfish, lobster, octopus,
#                   scorpion, seawasp, slug, starfish, worm

# 5. Number of Instances: 101

# 6. Number of Attributes: 18 (animal_name, 15 Boolean attributes, 2 numerics)

# 7. Attribute Information: (name of attribute and type of value domain)
#    1. animal_name:      Unique for each instance
#    2. hair                Boolean
#    3. feathers            Boolean
#    4. eggs                Boolean
#    5. milk                Boolean
#    6. airborne            Boolean
#    7. aquatic             Boolean
#    8. predator            Boolean
#    9. toothed             Boolean
#   10. backbone            Boolean
#   11. breathes            Boolean
#   12. venomous            Boolean
#   13. fins                Boolean
#   14. legs                Numeric (set of values: {0,2,4,5,6,8})
#   15. tail                Boolean
#   16. domestic            Boolean
#   17. catsize             Boolean
```

(continues on next page)

(continued from previous page)

```
#   18. class_type          Numeric (integer values in range [1,7])

# 8. Missing Attribute Values: None

# 9. Class Distribution: Given above
```

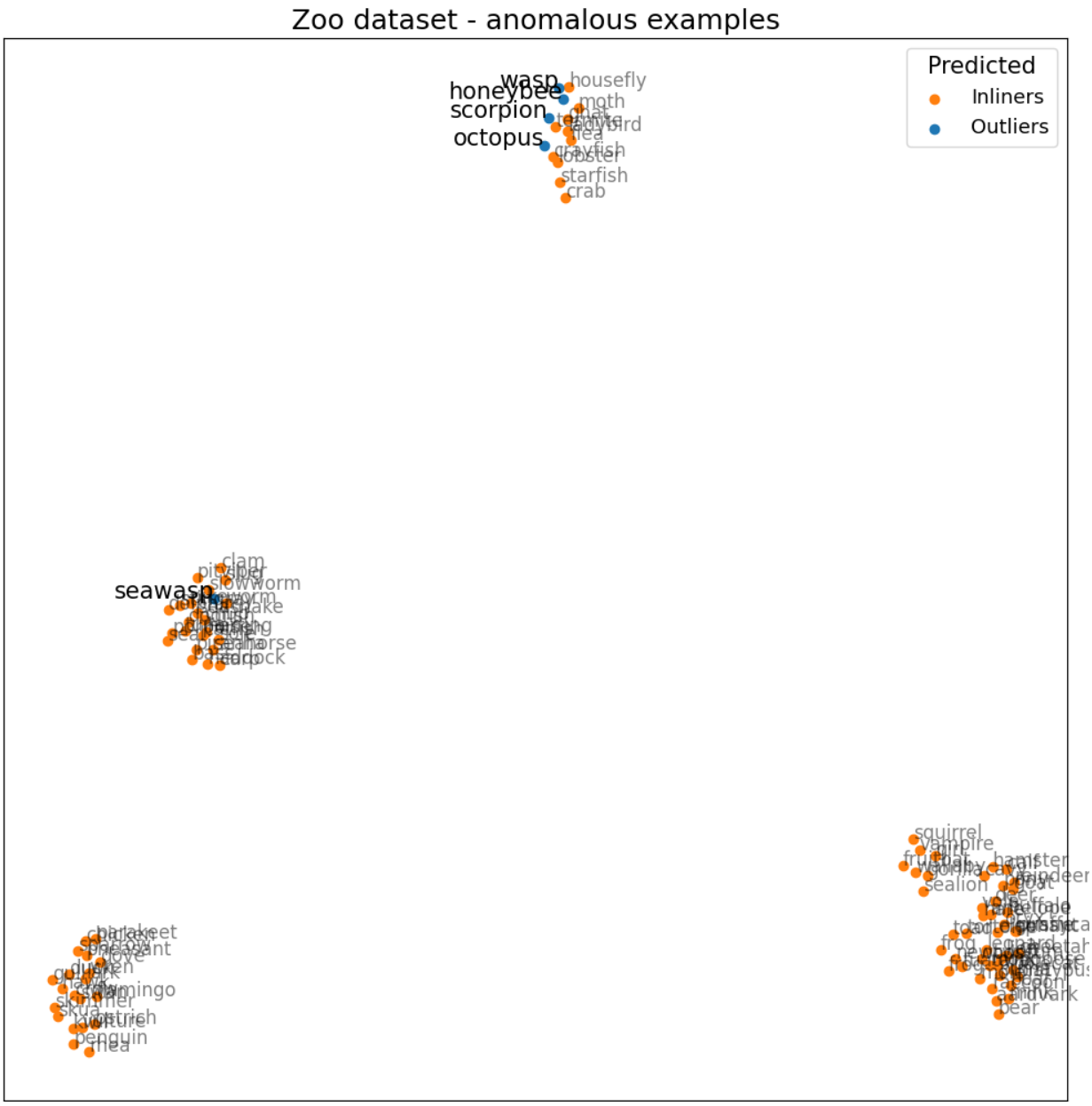
3.4.1 Data visualization

```
X = frame.values[:, :-1]

# Prepare data for visualization using UMAP
umap = UMAP(n_neighbors=15, min_dist=0.9, random_state=42)
transformed = umap.fit_transform(X)

plt.figure(figsize=(10, 10))
plt.subplot(111, aspect="auto")
plt.subplots_adjust(
    left=0.02, right=0.98, bottom=0.001, top=0.96, wspace=0.05, hspace=0.01
)


for type in np.unique(frame["class_type"]):
    selected = transformed[frame["class_type"] == type]
    plt.scatter(selected[:, 0], selected[:, 1], label=type)

for name, x, y in zip(frame.index, transformed[:, 0], transformed[:, 1]):
    plt.annotate(name, (x, y), alpha=0.8, fontsize=10)

plt.title("Zoo dataset - animal types", fontsize=18)
plt.xticks(())
plt.yticks(())
plt.legend(title="Animal type", title_fontsize=15, fontsize=13)
plt.show()
```

3.4.2 Explaining the cause of an anomaly

Once we get anomaly scores and importance of each feature, we could investigate further. We'll choose the five most anomalous animals. For example, we'll take a closer look at honeybee. It has a quite high score, and the most significant features are venomous (1.91), hair (1.55), breathes (1.28), and domestic (0.97). If we consider the composition of our dataset, there are no other venomous animals that are domestic, so it does seem right. We could find explanations like this for every other animal in the top five. Octopus has eight legs; sea wasp does have almost none of the features in the dataset, etc. So could we tell that these are the real reasons why these animals are unusual? Yes and no. Yes, this is why LODA sees them as anomalies considering our data, but without a review from a domain expert, we must be careful about such a statement. Also, consider the fact that this dataset is small, oversimplified, with just a limited number of features.

```
loda = LODA(n_estimators=100, bins=100, random_state=42)
loda.fit(X)

scores = loda.score_samples(X)
predicted = loda.predict(X)


plt.figure(figsize=(10, 10))
plt.subplot(111, aspect="auto")
plt.subplots_adjust(
    left=0.02, right=0.98, bottom=0.001, top=0.96, wspace=0.05, hspace=0.01
)

X_n = transformed[predicted == 1]
X_a = transformed[predicted == -1]

plt.scatter(X_n[:, 0], X_n[:, 1], color="tab:orange", label="Inliners")
plt.scatter(X_a[:, 0], X_a[:, 1], color="tab:blue", label="Outliers")

for name, x, y in zip(frame.index[predicted == 1], X_n[:, 0], X_n[:, 1]):
    plt.annotate(name, (x, y), alpha=0.5, fontsize=12)

for name, x, y in zip(frame.index[predicted == -1], X_a[:, 0], X_a[:, 1]):
    plt.annotate(name, (x, y), fontsize=15, ha="right")

plt.title("Zoo dataset - anomalous examples", fontsize=18)
plt.legend(title="Predicted", title_fontsize=15, fontsize=13)
plt.xticks(())
plt.yticks(())
plt.show()

feature_scores = loda.score_features(X)

for animal, score, feature_score in zip(
    frame[predicted == -1].itertuples(),
    scores[predicted == -1],
    feature_scores[predicted == -1],
):
    name = animal[0]
    srt = np.argsort(feature_score)[::-1]

    print(f"{name} score: {score:.3f}")

    for feature, value, importance in zip(
        frame.columns[srt][:4], np.array(animal[1:])[srt], feature_score[srt]
    ):
        print(f"\t{feature} {value} ({importance:.2f})")
```



Out:

```
honeybee score: -4.576
        venomous 1 (1.91)
        hair 1 (1.55)
        breathes 1 (1.28)
        domestic 1 (0.97)
octopus score: -5.763
        backbone 0 (3.06)
        legs 8 (1.79)
        feathers 0 (0.96)
        toothed 0 (0.80)
scorpion score: -5.007
        legs 8 (2.18)
```

(continues on next page)

(continued from previous page)

```
        toothed 0 (1.23)
        domestic 0 (1.16)
        feathers 0 (0.82)
seawasp score: -4.898
        backbone 0 (1.78)
        milk 0 (1.06)
        toothed 0 (1.05)
        feathers 0 (0.80)
wasp score: -4.579
        feathers 0 (1.99)
        fins 0 (1.43)
        catsize 0 (1.41)
        breathes 1 (1.16)
```

3.4.3 Summary

To sum it up. LODA has a really powerful tool to explain the cause of an anomaly. It is more resource consuming than scoring samples. We should take a closer look at anomalies if we want to tell the real reason.

3.4.4 References

Total running time of the script: (0 minutes 5.948 seconds)

CHAPTER

FOUR

DEVELOPERS GUIDE

4.1 Tools

As anlearn developers, we're using these tools.

- Code formating: black, isort
- Linting: flake8, mypy
- Requirements: poetry
- Testing: pytest, tox
- Documentation: sphinx
- Other: pre-commit, nix-shell, direnv

4.2 Setting-up the developers' environment

4.2.1 Nix-shell & direnv

For easy environment management, we're using nix-shell in combination with direnv. Using these two tools reduces the time and effort required to create and maintain a deterministic environment. We highly recommend using them. Nix configuration is in the *shell.nix* files + *nix* folder and the direnv configuration is in the *.envrc* file.

4.2.2 Python tools

As for python versions currently, support is for python 3.6, 3.7, and 3.8. To ensure a similar code style choice for formating is black and isort. As a linters we use mypy and flake8.

For easier code check before committing any changes, there is an option to use the pre-commit tool. As you can see in *.pre-commit-config.yaml* it is using only currently installed versions of black, isort, and flake8.

4.2.3 Installation using poetry

We are using the poetry for packaging and dependencies managemet. To installing all developement dependencies including ones for generating documentation simply use:

```
poetry install -E docs
```

- For pre-commit (after installing tools)

```
pre-commit install
```

4.3 Tests

All tests are in *test* folder. We're using a combinaiton of tox and pytest for testing. You can run tests by using the *tox* command directly or by using *make pytest* or *make check* commands.

4.3.1 Configuration files

- Configurations for flake8 and mypy are in the *setup.cfg* file.
- Configurations for isort, black, and pytest are in the *pyproject.toml* file.
- Configuration for pre-commit is in the *.pre-commit-config.yaml* file.
- Configuration for tox is in the *tox.ini* file.

4.4 Documentation

We're using sphinx in combination with Read the Docs Sphinx Theme. For generating the documentation, you have to have *anlearn[docs]* installed (*poetry install -E docs*). You can create the documentation by using the *make docs* command.

CHAPTER

FIVE

API REFERENCE

anlearn:

loda
stats

5.1 anlearn.loda

5.1.1 anlearn.loda.Histogram

class anlearn.loda.**Histogram**(*bins: Union[int, str] = 'auto', return_min: bool = True*)

Histogram model

Histogram model based on `scipy.stats.rv_histogram`.

Parameters

- **bins** (*Union[int, str], optional*) –
  - `int` - number of equal-width bins in the given range.
  - `str` - method used to calculate bin width (`numpy.histogram_bin_edges`).

  See `numpy.histogram_bin_edges` bins for more details, by default "auto"
- **return_min** (*bool, optional*) – Return minimal float value instead of 0, by default True

hist

Value of histogram

Type numpy.ndarray

bin_edges

Edges of histogram

Type numpy.ndarray

pdf

Probability density function

Type numpy.ndarray

fit(*X: numpy.ndarray*) → *anlearn.loda.Histogram*

Fit estimator

Parameters **X** (*numpy.ndarray*) – Input data, shape (n_samples,)

Returns Fitted estimator

Return type *Histogram*

get_params(*deep=True*)

Get parameters for this estimator.

Parameters **deep** (*bool, default=True*) – If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns **params** – Parameter names mapped to their values.

Return type dict

predict_proba(*X: numpy.ndarray*) → numpy.ndarray

Predict probability

Predict probability of input data X.

Parameters **X** (*numpy.ndarray*) – Input data, shape (n_samples,)

Returns Probability estimated from histogram, shape (n_samples,)

Return type numpy.ndarray

set_params(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters ****params** (*dict*) – Estimator parameters.

Returns **self** – Estimator instance.

Return type estimator instance

5.1.2 anlearn.loda.LODA

class anlearn.loda.**LODA**(*n_estimators: int = 1000, bins: Union[int, str] = 'auto', q: float = 0.05, random_state: Optional[int] = None, n_jobs: Optional[int] = None, verbose: int = 0*)

LODA: Lightweight on-line detector of anomalies[1]

LODA is an ensemble of histograms on random projections. See Pevný, T. Loda[1] for more details.

Parameters

- **n_estimators** (*int, optional*) – number of histograms, by default 1000
- **bins** (*Union[int, str], optional*) –
  - `int` - number of equal-width bins in the given range.
  - `str` - method used to calculate bin width (`numpy.histogram_bin_edges`).

  See `numpy.histogram_bin_edges` bins for more details, by default "auto"
- **q** (*float, optional*) – Quantile for compution threshold from training data scores. This threshold is used for *predict* method, by default 0.05

[1] Pevný, T. Loda: Lightweight on-line detector of anomalies. Mach Learn 102, 275–304 (2016). <https://doi.org/10.1007/s10994-015-5521-0>

- **random_state** (*Optional[int], optional*) – Random seed used for stochastic parts., by default None
- **n_jobs** (*Optional[int], optional*) – Not implemented yet, by default None
- **verbose** (*int, optional*) – Verbosity of logging, by default 0

projections_
Random projections, shape (n_estimators, n_features)

Type numpy.ndarray

hists_
Histograms on random projections, shape (n_estimators,)

Type List[*Histogram*]

anomaly_threshold_
Treshold for *predict()* function

Type float

Examples

```
>>> import numpy as np
>>> from anlearn.loda import LODA
>>> X = np.array([[0, 0], [0.1, -0.2], [0.3, 0.2], [0.2, 0.2], [-5, -5], [0.6, 0.
↪7]])
>>> loda = LODA(n_estimators=10, bins=10, random_state=42)
>>> loda.fit(X)
LODA(bins=10, n_estimators=10, random_state=42)
>>> loda.predict(X)
array([ 1,  1,  1,  1, -1,  1])
```

References

fit(*X: anlearn._typing.ArrayLike, y: Optional[anlearn._typing.ArrayLike] = None*) → *anlearn.loda.LODA*
Fit estimator

Parameters

- **X** (*ArrayLike*) – Input data, shape (n_samples, n_features)
- **y** (*Optional[ArrayLike], optional*) – Present for API consistency by convention, by default None

Returns Fitted estimator

Return type *LODA*

fit_predict(*X, y=None*)
Perform fit on X and returns labels for X.

Returns -1 for outliers and 1 for inliers.

Parameters

- **X** (*{array-like, sparse matrix, dataframe} of shape (n_samples, n_features)*) –
- **y** (*Ignored*) – Not used, present for API consistency by convention.

Returns **y** – 1 for inliers, -1 for outliers.

Return type ndarray of shape (n_samples,)

get_params(*deep=True*)

Get parameters for this estimator.

Parameters **deep** (*bool, default=True*) – If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns **params** – Parameter names mapped to their values.

Return type dict

predict(*X: anlearn._typing.ArrayLike*) → numpy.ndarray

Predict if samples are outliers or not

Samples with a score lower than *anomaly_threshold_* are considered to be outliers.

Parameters **X** (*ArrayLike*) – Input data, shape (n_samples, n_features)

Returns 1 for inlineres, -1 for outliers, shape (n_samples,)

Return type numpy.ndarray

score_features(*X: anlearn._typing.ArrayLike*) → numpy.ndarray

Feature importance

Feature importance is computed as a one-tailed two-sample t-test between $-log(\hat{p}_i)$ from histograms on projections with and without a specific feature. The higher the value is, the more important feature is.

See full description in **3.3 Explaining the cause of an anomaly**[Page 40, 1] for more details.

Parameters **X** (*ArrayLike*) – input data, shape (n_samples, n_features)

Returns Feature importance in anomaly detection.

Return type numpy.ndarray

Notes

$$t_j = \frac{\mu_j - \bar{\mu}_j}{\sqrt{\frac{s_j^2}{|I_j|} + \frac{\bar{s}_j^2}{|\bar{I}_j|}}}$$

score_samples(*X: anlearn._typing.ArrayLike*) → numpy.ndarray

Anomaly scores for samples

Average of the logarithm probabilities estimated of individual projections. Output is proportional to the negative log-likelihood of the sample, that means the less likely a sample is, the higher the anomaly value it receives[Page 40, 1]. This score is reversed for scikit-learn compatibility.

Parameters **X** (*ArrayLike*) – Input data, shape (n_samples, n_features)

Returns The anomaly score of the input samples. The lower, the more abnormal. Shape (n_samples,)

Return type numpy.ndarray

set_params(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters ****params** (*dict*) – Estimator parameters.

Returns **self** – Estimator instance.

Return type estimator instance

- *LODA: large data - Credit Card Fraud Detection dataset*

5.2 anlearn.stats

5.2.1 anlearn.stats.IQR

class `anlearn.stats.IQR`(*k: float = 1.5, lower_quantile: float = 0.25, upper_quantile: float = 0.75, ensure_2d: bool = True*)

Interquartile range

Outlier deteciton method using Tukey's fences. If lower quantile is 0.25 (Q_1 lower quartile) and upper quantile is 0.75 (Q_3 upper quartile), then outlier is any observation outside the range:

$$[Q_1 - k(Q_3 - Q_1); Q_3 + k(Q_3 - Q_1)]$$

John Tukey proposed $k = 1.5$ is an outlier, and $k = 3$ is far out.

Parameters

- **k** (*float, optional*) – Outlier threshold, by default 1.5
- **lower_quantile** (*float, optional*) – Lower quantile, from (0; 1), by default 0.25
- **upper_quantile** (*float, optional*) – Upper quantile, from (0; 1), by default 0.75
- **ensure_2d** (*bool, optional*) – Frobid input 1D arrays, by default True

lqv_

Lower quantile value estimated from the input data

Type float

uqv_

Upper quantile value estimated from the input data

Type float

iqr_

Interquartile range, *uqv_* - *lqv_*

Type float

Example

```
>>> import numpy as np
>>> from anlearn.stats import IQR
>>> X = np.hstack([[-7,-4], np.arange(5), [10, 15]])
>>> iqr = IQR(ensure_2d=False)
>>> iqr.fit(X)
IQR(ensure_2d=False)
>>> iqr.predict(X)
array([-1,  1,  1,  1,  1,  1,  1,  1, -1])
>>> iqr.score_samples(X)
array([-1.75, -1.  , -0.  , -0.  , -0.  , -0.  , -0.  , -1.5 , -2.75])
```

Raises `ValueError` – Lower quantile must be lower than upper quantile.

`fit`(*X: anlearn._typing.ArrayLike*, *y: Optional[anlearn._typing.ArrayLike] = None*) → *anlearn.stats.IQR*

Fit estimator

Parameters

- **X** (*ArrayLike*) – Input data of shape (n_samples, 1) or (n_samples,) if *ensure_2d* is False
- **y** (*Optional[ArrayLike], optional*) – Ignored, present for API consistency by convention, by default None

Returns Fitted estimator

Return type *IQR*

`fit_predict`(*X*, *y=None*)

Perform fit on X and returns labels for X.

Returns -1 for outliers and 1 for inliers.

Parameters

- **X** (*{array-like, sparse matrix, dataframe} of shape (n_samples, n_features)*) –
- **y** (*Ignored*) – Not used, present for API consistency by convention.

Returns **y** – 1 for inliers, -1 for outliers.

Return type ndarray of shape (n_samples,)

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters **deep** (*bool, default=True*) – If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns **params** – Parameter names mapped to their values.

Return type dict

`predict`(*X: anlearn._typing.ArrayLike*) → numpy.ndarray

Predict if samples are outliers or not

Samples with a score lower than **k** are considered to be outliers.

Parameters **X** (*ArrayLike*) – Input data, shape (n_samples, n_features)

Returns Shape (n_samples,) 1 for inlineres, -1 for outliers

Return type numpy.ndarray

score_samples(*X: anlearn._typing.ArrayLike*) → numpy.ndarray

Score samples

Score is comuputed as distance from interval $[Q_{lower}; Q_{upper}]$ divided by interquartile range. $score = distance(data, (lqv, uqv))/iqr$. Score is inverted for scikit-learn compatibility

Parameters **X** (*ArrayLike*) – Input data of shape (n_samples, 1) or (n_samples,) if `ensure_2d` is False

Returns Shape (n_samples,). The outlier score of the input samples. The lower, the more abnormal.

Return type numpy.ndarray

set_params(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters ****params** (*dict*) – Estimator parameters.

Returns **self** – Estimator instance.

Return type estimator instance

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

CHAPTER

SEVEN

LICENSE

PYTHON MODULE INDEX

a

INDEX