AutoML for anomaly detection

Davide Schiaroli

August 9, 2022

Abstract

In this work we're gonna first discover what AutoMl is, why it is important in 2022 to automate the process and steps involved in the Machine Learning field and next what are the main tools used right now, analyzing and discussing the literature present at this time.
Regarding the framework, we're gonna analyze the distribution of each of them, in particular referring to the peculiarities and differences of each of them and the usability, that is the ease of installing and use of the tools.

1 Introduction

Since the birth of machine learning as we know it, some problems have slowed down the work of developers and researchers. The most common and insidious problems usually are:

- Tuning of many hyperparameters
- Traditional ML algorithms can require lot of computational power
- Feature selection and extraction
- Model discovery

1.1 AutoMl

To answer these problem, AutoML, or Automated Machine Learning has been proposed. For the moment, we can consider AutoML as a process that allows developers to simplify and speed up the creation of Machine Learning models, using processes, tools and techniques that allow some automation of otherwise manual processes. For example, we can note the most important ones:

- Feature selection and extraction
- Model selection
- Hyperparameter tuning

AutoML aims to evaluate performance automatically, based on user or stock predefined metrics, like accuracy for classification. In the next two figures, we can see what are the steps on a classic Machine Learning task that can be improved using AutoMl. In 1 we can see a no-AutoML scenario, while in 2 we can see where the AutoML processes are used. Nowadays AutoML research paper are mainly focused on supervised learning, but the study of this process for unsupervised learning methods is growing. This is especially true for anomaly detection.

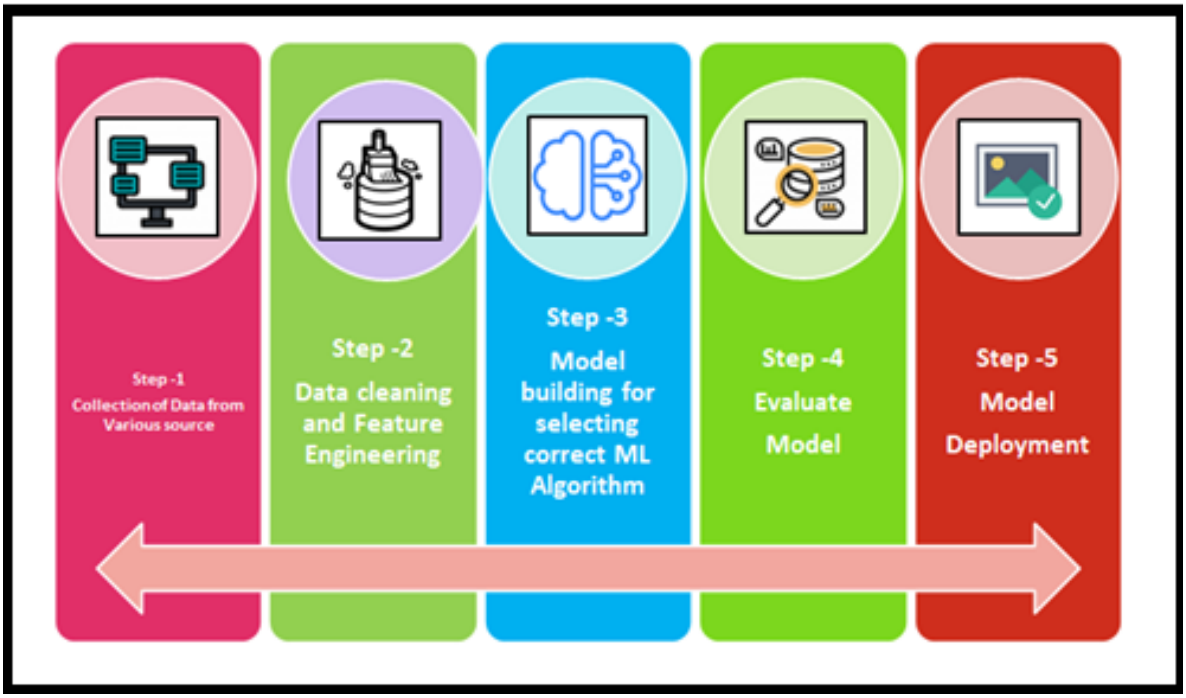

Figure 1: Machine Learning flow

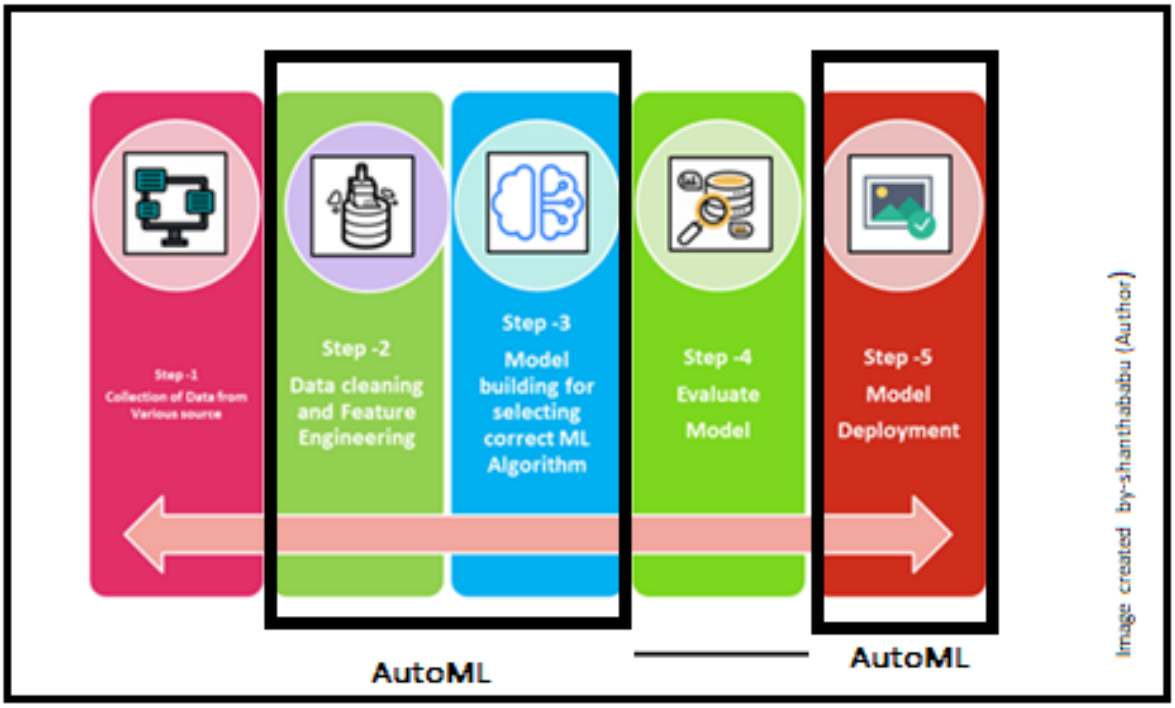

Figure 2: Machine Learning steps covered by AutoML

1.2 Anomaly detection

Anomaly detection is a common problem in machine learning, that consist in identifying outlier in a dataset 3. In the early days, anomaly detection was used for deleting noise in the training process, but now these types of algorithms are used in many heterogeneous fields, like in network security, data storage, medicine and finance. Three categories of anomaly detection exist:

- Supervised
- Semi-supervised
- Unsupervised

The first require a labelled dataset, which is not so common in anomaly detection field. The second assume that only a part of the data is labelled, while the last categories is the most used because it does not assume any labelling of data. It is important to note that an outlier is an example of data that is so different from other data in the same datasets that its generation method can be suspect. This is only one of the several definition of anomalies or outlier existing in the literature.

1.2.1 Time series anomalies

In the anomaly detection field, there is a interesting topic which has had numerous publications recently. First, a time series are observation that have been recorded in an orderly fashion and which are correlated in time. We can split anomalies into unwanted data, so we need to clean out the outliers from the datasets, and potentially interesting events that need to be analyzed. The unwanted outlier can be created by noise or fraudolent actions. In the past few years,many research is addressed to analyze these type of outliers, like in fraud detection. The outliers can be splitted also in Point and Subsequence. The point outlier is a datum that behaves differently in a specific time form the others in the same time series or to its neighbour points. Points can be univariate or multivariate, depending oon whether they affect one or more time-dependent variables. The subsequence outlier are consecutive point whose joint behaviour is unusual, but every single point, alone, is not an outlier.

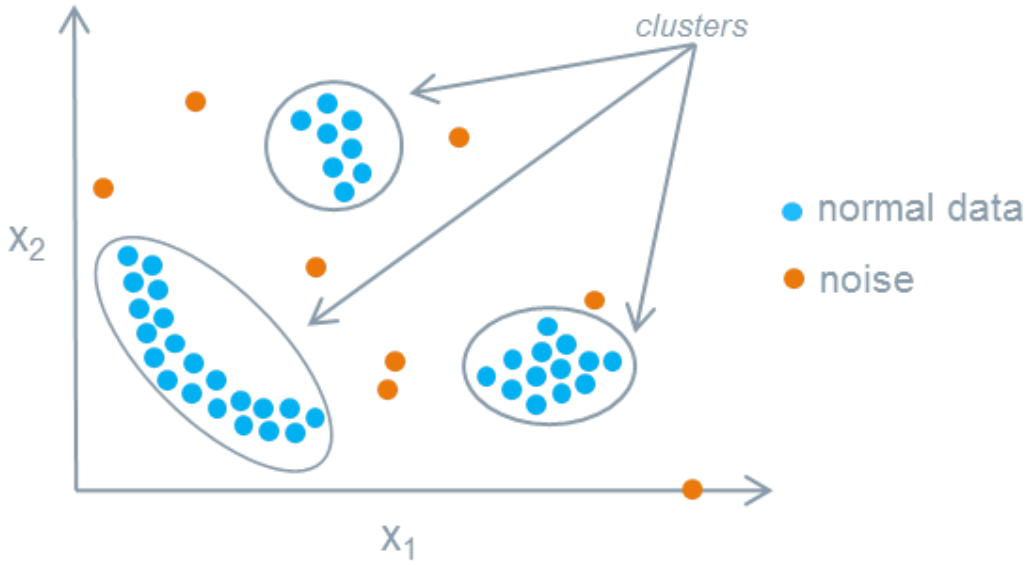

Figure 3: Anomaly detection example

1.3 Target and criteria of the survey

Like stated in [BSPS22], it is only in the last 5 years that the attention of researchers has poured into this branch of IA and Machine Learning, while the first true AutoML solution has been developed in 2004 in the medical research field. GEMS (gene expression model selector) [STDA05] is a system for the automated development and evaluation of high-quality cancer diagnostic models and biomarker discovery from microarray gene expression data. It therefore seems rational to take into consideration the technologies and frameworks developed in the last decade (t10). As for the audience of this SLR, we will assume that the reader already has a good knowledge of the subject, i.e. we will not explain the basic concepts of Machine Learning, but rather only some specific concepts of the AutoML or more specific topic.(a1000)

1.3.1 Research question and sources

The research questions that represent the goal of this SLR are these:

- Goal - What are the main tools and methods that use AutoML for anomaly detection?
- Q1) Which aspects of anomaly detection are improvable using AutoML?
- Q2) Which tools are actually used in nowadays applications?

The first question, or goal of the SLR, is a general question that addresses to summarise the tools, frameworks and technique used in 2022 for anomaly detection, focusing on systems that use autoML. The second question focuses on the aspects of anomaly detection that can or are actually improved by the use of autoML techniques.
Finally, we have one question that aims to separate those that are the theoretical tools, that is those that have not been developed and made into usable frameworks, from those that are public and therefore usable.
The criteria regarding the sources, given that the specific topic does not have a large literature, will be exclusively used to check the veracity of the sources, choosing it from the publication sites contained in [Omi22] and that have a valid number of citations.
For clarity, the sources used are the following

- Google Scholar http://scholar.google.com/
- Scopus http://www.scopus.com/

- Web of Knowledge http://www.webofknowledge.com/
- DBLP http://dblp.uni-trier.de/db/
- Publons http://publons.com
- IRIS http://cris.unibo.it/
- APICe http://pubs.apice.unibo.it/

No seminar or conference has been found with talks regarding this specific topic, so are not considered in this survey.

Then these specific keywords are used to queries the publisher databeses:

- (KW1) automl anomaly detection
- (KW2) automl outlier detection
- (KW3) automated machine learning anomaly detection
- (KW4) automated machine learning outlier detection

Then the first 20 result are taken as possible literature to be analyzed. As many publications did not add any useful content to the topic, they were excluded from the research. The papers has been categorized in *primary* paper, that is the ones that are introductory or "general purpose", and *specific* paper, the ones that talks about specific topic in anomaly detection, like network intrusion detection, fraud transaction detection, etc.. .

2 AutoML Methods

In this chapter we're gonna discover some of the problems that AutoML aims to solve and some of the methods that AutoML frameworks are exploiting.

2.1 Hyperparameter Optimization (HPO)

AutoML and the recent surge of Deep Learning, focused the researchers and the developers into the hyperparameter optimization field, also called HPO. Every single model in Machine Learning needs hyperparameters, and the most basic AutoML task is to automatically set these hyperparameter to increase performance of the model. Automated hyperparameter optimization has several important use cases:

- reduce the human effort necessary for applying machine learning. This is particularly important in the context of AutoML.
- improve the performance of machine learning algorithms (by tailoring them to the problem at hand); this has led to new state-of-the-art performances for important machine learning benchmarks in several studies
- improve the reproducibility and fairness of scientific studies. Automated HPO is clearly more reproducible than manual search. It facilitates fair comparisons since different methods can only be compared fairly if they all receive the same level of tuning for the problem at hand

The optimization problem dates back to the 1990s, when was also established that different hyperparameter combinations fits different datasets and nowadays it's well known that default hyperparameter set are somewhat inadequate.

Because of the increased usage of machine learning in companies, HPO is also of substantial commercial interest and plays an ever larger role there, be it in company-internal tools, as part of machine learning cloud services, or as a service by itself.

The main challenge of HPO are:

- Function evaluation for: large models, complex machine learning pipelines, or large datesets.
- Complex and high-dimensional configuration spaces.
- Impossibility to directly optimize for generilization due to the limited size of the training datasets.

2.1.1 Problems

The domain of a hyperparameter can be real-valued (e.g., learning rate), integer-valued (e.g., number of layers), binary (e.g., whether to use early stopping or not), or categorical (e.g., choice of optimizer). For integer and real-valued hyperparameters, the domains are mostly bounded for practical reasons.

The configuration space an contain **conditionality**, that can occur when certain hyperparameters values exclude some other values.

Such conditional spaces occur in the following cases:

- in the process of optimizing the architecture of a neural network, e.g. the number of layers can be an integer hyperparameter and the per-layer hyperparameters of layer i are only active if the network depth is at least i
- in the automated tuning of machine learning pipelines, where the choice between different preprocessing and machine learning algorithms is modeled as a categorical hyperparameter. This problem is known as Full Model Selection (FMS) or Combined Algorithm Selection and Hyperparameter optimization problem (CASH).

In order to reduce the evaluation time a set of strategies have been proposed:

- test machine learning algorithms on a subset of folds
- test machine learning algorithms on a subset of data

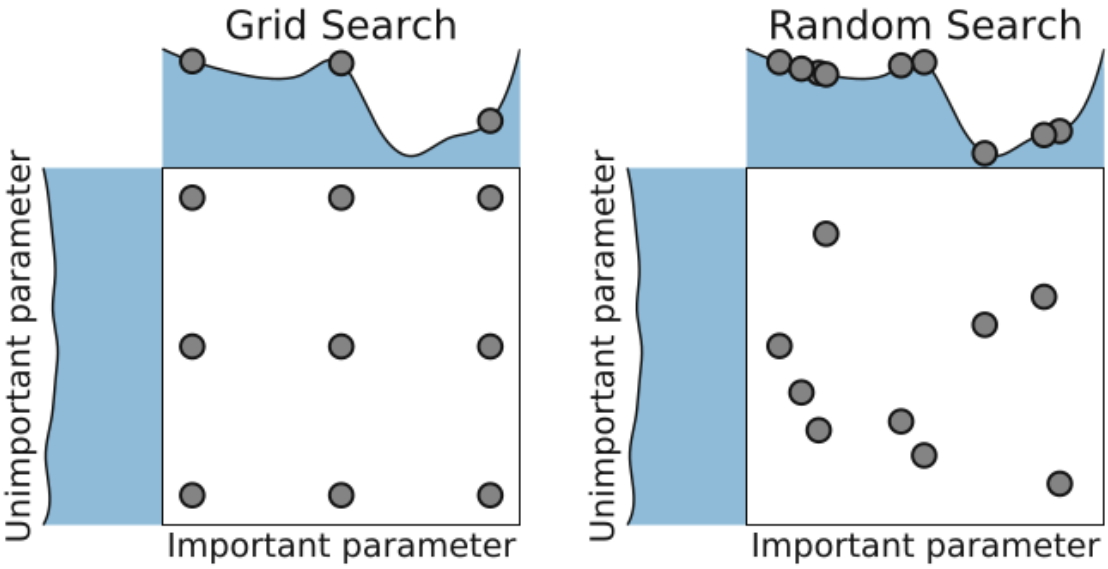

Figure 4: Comparison of grid search and random search

- test machine learning algorithms for a small amount of iterations

Recent work on multi-task and multi-source optimization can provide cheap information to help HPO, but do not necessarily train a machine learning model on the dataset of interest and therefore do not yield a usable model as a side product.

Only choosing a single hyperparameter configuration can be wasteful when many good configurations have been identified by HPO, and combining them in an ensemble can improve performance.This is particularly useful in AutoML systems with a large configuration space like, in FMS or CASH, where good configurations can be very diverse and it is also possible to directly optimize for models which would maximally improve an existing ensemble.

2.2 Blackbox Hyperparameter Optimization

In general, every blackbox optimization method can be applied to HPO. Due to the non-convex nature of the problem, global optimization algorithms are usually preferred, but some locality in the optimization process is useful in order to make progress within the few function evaluations that are usually available. We first discuss model-free blackbox HPO methods and then describe blackbox Bayesian optimization methods.

2.2.1 Model-Free Blackbox Optimization Methods

Grid search is the most basic HPO method, also known as full factorial design. The user specifies a finite set of values for each hyperparameter, and grid search evaluates the Cartesian product of these sets. Unfortunately this suffers from the curse of dimensionality (since the required number of function evaluations grows exponentially with the dimensionality of the configuration space) and another problem is that increasing the resolution of discretization substantially increases the required number of function evaluations. A simple alternative to grid search is **random search**. As the name suggests, random search samples configurations at random until a certain budget for the search is exhausted. This works better than grid search when some hyperparameters are much more important than others. To clearify this concept figure 4 can be used as a reference. Further advantages over grid search include easier parallelization (since workers do not need to communicate with each other) and flexible resource allocation. Random search can also be a useful method for initializing the search process, as it explores the entire configuration space and thus often finds settings with reasonable performance. However, it often takes far longer than guided search methods to identify one of the best performing. Population-based methods, such as genetic algorithms, evolutionary algorithms,

evolutionary strategies, and particle swarm optimization are optimization algorithms that maintain a population, i.e., a set of configurations, and improve this population by applying local perturbations (so-called mutations) and combinations of different members (so-called crossover) to obtain a new generation of better configurations. These methods are conceptually simple, can handle different data types, and are embarrassingly parallel since a population of N members can be evaluated in parallel on N machines. One of the best known population-based methods is the covariance matrix adaption evolutionary strategy (CMA-ES), which samples configurations from a multivariate Gaussian whose mean and covariance are updated in each generation based on the success of the population's individuals.

2.3 Meta-learning

Meta-learning is defined as the "the science of systematically observing how different machine learning approaches perform on a wide range of learning tasks, and then learning from this experience, or meta-data, to learn new tasks much faster than otherwise possible." [HKV19] In fact, tradtional learning, or base-learning has bias fixed a priori, or choosen by the user, while meta-learning studies how to choose the right bias. In this chapter, we provide an overview of the state of the art in this continuously evolving field.

When we learn new skills, we rarely start from scratch. We start from skills learned earlier in related tasks, reuse approaches that worked well before, and focus on what is likely worth trying based on experience. With every skill learned, learning new skills becomes easier, requiring fewer examples and less trial-and-error. In short, we learn how to learn across tasks. Likewise, when building machine learning models for a specific task, we often build on experience with related tasks, or use our (often implicit) understanding of the behavior of machine learning techniques to help make the right choices. The challenge in meta-learning is to learn from prior experience in a systematic, data-driven way.

First, we need to collect meta-data that describe prior learning tasks and previously learned models. They comprise the exact algorithm configurations used to train the models, including hyperparameter settings, pipeline compositions and/or network architectures, the resulting model evaluations, such as accuracy and training time, the learned model parameters, such as the trained weights of a neural net, as well as measurable properties of the task itself, also called *meta-features*.

Second, we need to learn from this prior meta-data, to extract and transfer knowledge that guides the search for optimal models for new tasks.

However, when a new task represents completely unrelated phenomena, or random noise, leveraging prior experience will not be effective. Luckily, in real-world tasks, there are plenty of opportunities to learn from prior experience.

2.4 Neural Architecture search (NAS)

Beacuse of the recent Deep Learning progress, there have been an increase in the reasearch filed of automated Neural Architecture Search. This can be extremely profitable for the developers that can save lot of time otherwise spent in the study of the perfect model for a domain. We can devide NAS methods in three categories:

- Search space: defines which architectures can be represented in principle. Incorporating prior knowledge about properties well-suited for a task can reduce the size of the search space and simplify the search.However, this also introduces a human bias, which may prevent finding novel architectural building blocks that go beyond the current human knowledge.

- Search strategy: details how to explore the search space. It encompasses the classical exploration-exploitation trade-off since, on the one hand, it is desirable to find well-performing architectures quickly, while on the other hand, premature convergence to a region of suboptimal architectures should be avoided.

- Performance estimation strategy: once an architecture capable to achieve high predictive performance on unseen data is found, we have to be able to estimate its performance. The simplest option is to perform a standard training and validation of the architecture on data, but this is unfortunately computationally expensive and limits the number of architectures that can be

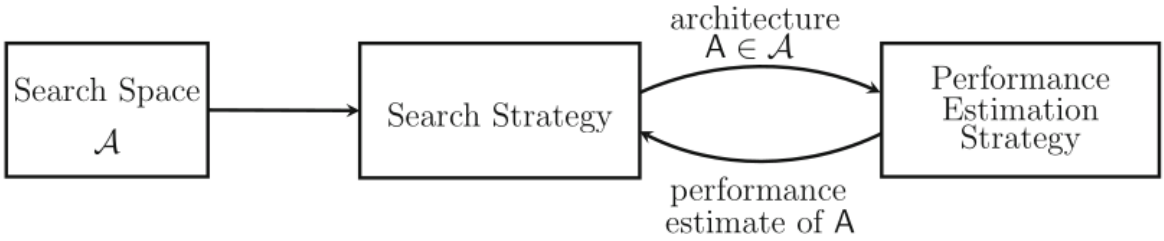

Figure 5: Phases of Neural Architecture search (NAS)

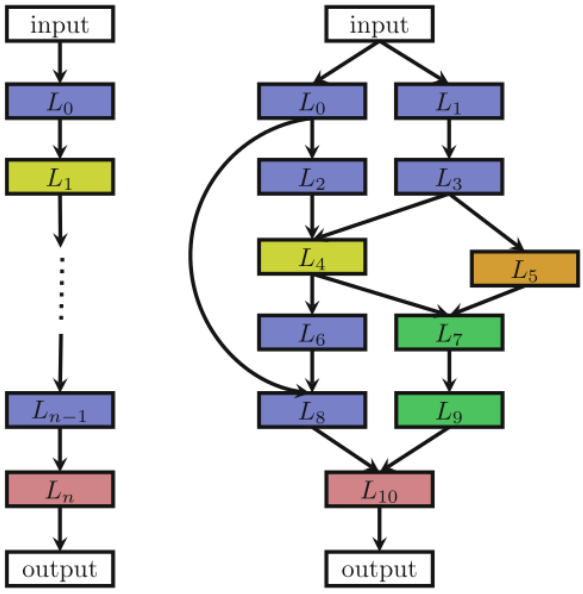

Figure 6: Search space of chain-structured neural networks

explored. Recent research therefore focuses on developing methods that reduce the cost of these performance estimations.

As an example of these three phases and to see how they are related one another we can take a look at the figure 5.

2.4.1 Search space

A simple search space is the space of *chain-structured neural networks*, and we can see an example in figure 6. A chain-structured neural network architecture A can be written as a sequence of n layers, where the i'th layer L_i receives its input from layer $i-1$ and its output serves as the input for layer $i+1$. The search space is then parametrized by:

1. the (maximum) number of layers n
2. the type of operation every layer can execute, e.g., pooling, convolution...
3. hyperparameters associated with the operation, e.g., number of filters, kernel size...

However we have to remember that the paramethers from (3) are conditioned on (2). Recent work on NAS incorporate modern design elements known from hand-crafted architectures such as skip connections, which allow to build complex, multi-branch networks.

2.4.2 Search strategy

Many different search strategies can be used to explore the space of neural architectures, including random search, Bayesian optimization, evolutionary methods, reinforcement learning (RL), and

gradient-based methods. Historically, evolutionary algorithms were already used by many researchers to evolve neural architectures (and often also their weights) decades ago.

To frame NAS as a reinforcement learning (RL) problem, the generation of a neural architecture can be considered to be the agent's action, with the action space identical to the search space. The agent's reward is based on an estimate of the performance of the trained architecture on unseen data. Different RL approaches differ in how they represent the agent's policy and how they optimize it. As an example recurrent neural network (RNN) policy can be used to sequentially sample a string that in turn encodes the neural architecture[ZL17].

Real et al.[RAHL19] conduct a case study comparing RL, evolution, and random search (RS), concluding that RL and evolution perform equally well in terms of final test accuracy, with evolution having better anytime performance and finding smaller models. Both approaches consistently perform better than RS in their experiments, but with a rather small margin: RS achieved test errors of approximately 4% on CIFAR-10, while RL and evolution reached approximately 3.5%.

Bayesian Optimization (BO) is one of the most popular methods for hyperparameter optimization (see also Chap. 1 of this book), but it has not been applied to NAS by many groups since typical BO toolboxes are based on Gaussian processes and focus on low-dimensional continuous optimization problems and even thpugh a full comparison is lacking, there is preliminary evidence that these approaches can also outperform evolutionary algorithms.

2.4.3 Performance estimation strategy

The simplest way to estimate the performance of a given architecture A is to train A on training data and evaluate its performance on validation data. However, training each architecture to be evaluated from scratch frequently yields computational demands in the order of thousands of GPU days for NAS. To reduce this computational burden, performance can be estimated based on lower fidelities of the actual performance after full training (also denoted as proxy metrics). Such lower fidelities include shorter training times, training on a subset of the data, on lower-resolution images, or with less filters per layer.

These low-fidelity approximations reduce the computational cost but they also introduce bias in the estimate as performance will typically be underestimated and this could be a problem, in fact, recent results indicate that relative ranking can change dramatically when the difference between the cheap approximations and the "full" evaluation is too big.

Another approach to speed up performance estimation is to initialize the weights of novel architectures based on weights of other architectures that have been trained before. One way of achieving this, dubbed network morphisms, allows modifying an architecture while leaving the function represented by the network unchanged.Continuing training for a few epochs can also make use of the additional capacity introduced by network morphisms. One-Shot Architecture Search is another promising approach for speeding up performance estimation, which treats all architectures as different subgraphs of a supergraph (the one-shot model) and shares weights between architectures that have edges of this supergraph in common.

Only the weights of a single one-shot model need to be trained and architectures (which are just subgraphs of the one-shot model) can just inherit trained weights from the one-shot model. This greatly speeds up performance estimation of architectures, since no training is required (only evaluating performance on validation data).

A general limitation of one-shot NAS is that the supergraph defined a-priori restricts the search space to its subgraphs. Moreover, approaches which require that the entire supergraph resides in GPU memory during architecture search will be restricted to relatively small supergraphs and search spaces accordingly and are thus typically used in combination with cell-based search spaces.

3 AutoMl frameworks for anomaly detection

In the literature there are several methods that uses the principle of automated machine learning in the anomaly detection field. They can be classified according to the type of evaluation (supervides or unsupervised). Furthermore, some of the methods are designed to be used for searching anomalies in time-series. In the next table it is possible to observe the framework or libraries considered for the examination.

Method name	Evaluation	Repository	Language
TODS	Supervised	https://github.com/datamllab/tods	Python
Pyodds	Supervised	https://github.com/datamllab/pyodds	Python
LSCP	Supervised	https://github.com/yzhao062/LSCP	Python
PyOd	Supervised	https://github.com/yzhao062/pyod	Python
MetaOd	Unsupervised	https://github.com/yzhao062/MetaOD	Python
Meta-AAD	Unsupervised	-	-
AutoOD	Supervised	-	-

3.1 Test environment

All the frameworks described before has been tested on the cloud, using Google Colab and on a machine with Manajaro, Kernel Linux 5.15.57-2-Manjaro.
The CPU of the machine is an Intel i7-7500u, 3.5 GHz. In the machine the latest major version released of Python and PiP has been used (3.10 for Python and 22.2 for PiP). Furthermore, in order to make the installation steps platform independent and more reproducible, all these steps are self-contained into a Docker container.

3.1.1 Reproducibility

In order to make these experiment deployable in different system, a Docker container has been created. To manage different versions of python, pip and the necessary libraries instead, there were several choices:

- PyEnv
- Pipx
- Poetry

The most suitable choice was PyEnv, so it has been used for the deployment. As operative system a LTS Ubuntu version was the most suitable to be able to install all the various dependencies required.

3.1.2 Docker image

In the next snippet we can see a basic Docker image that use PyEnv on Ubuntu for installing two different versions of the Python interpeter.

```
FROM ubuntu:18.04

RUN apt update && apt install -y git curl make build-essential libssl-dev zlib1g-dev \
libbz2-dev libreadline-dev libsqlite3-dev wget curl llvm libncurses5-dev \
libncursesw5-dev xz-utils tk-dev libffi-dev liblzma-dev python-openssl
RUN curl https://pyenv.run | bash
RUN pyenv install -v 3.7.2
RUN pyenv install -v 3.6
```

3.2 PyOd

PyOD introduces himself to the public as "the most comprehensive and scalable Python library for detecting outlying objects in multivariate data". The paper about PyOd has been published in 2019 [ZNL19], since the creation of the Github repository 2018, PyOd has been downloaded more than 7 million times. PyOD includes more than 40 detection algorithms, and a benchmark has been developed and published counting 30 anomaly detection algorithms on 55 datasets. [HHH+22] Unlike the following frameworks, PyOd can be thought of as a library or set of tools that is used as a basic platform for the development of more complex frameworks.

3.2.1 Installation

Requirements:

- Python 3.6
- joblib
- matplotlib
- numpy>=1.19
- numba>=0.51
- scipy>=1.5.1
- scikit_learn>=0.20.0
- six
- statsmodels

There are several methods for installing PyOd: using pip, conda and cloning the project.

3.2.2 Use and comments

PyOD allows the creation of an outlier detection model in 5 lines of code:

```
# train an ECOD detector
from pyod.models.ecod import ECOD
clf = ECOD()
clf.fit(X_train)

# get outlier scores
y_train_scr = clf.decision_scores_ #outlier scores on the train
y_test_scr = clf.decision_function(X_test) #predict outlier on test
```

PyOD has some useful feature like the model save and load and the fast training. These feature also can be used with few lines of code.

```
from joblib import dump, load

# save the model
dump(clf, 'clf.joblib')
# load the model
clf = load('clf.joblib')
```

To test how PyOd works, an example given by the author of a RNN detector has been tested. We can see in the next figure what is displayed to the user.

The code that has been runned is the following:

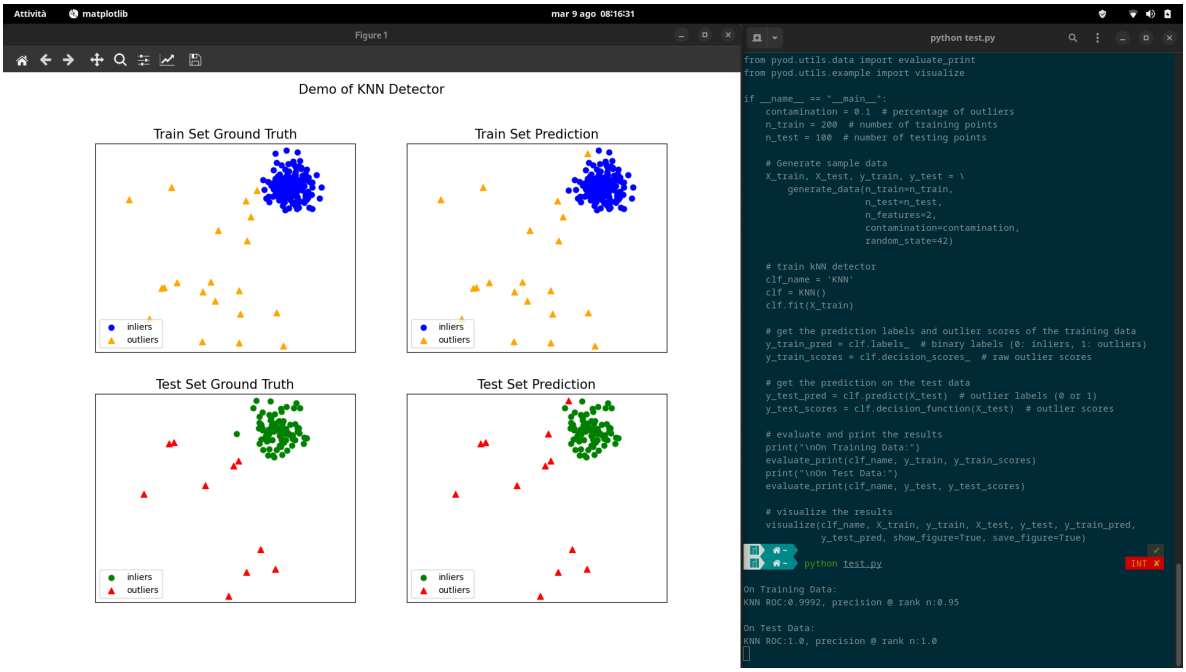

Figure 7: Run of a PyOd example

```
# -*- coding: utf-8 -*-
"""Example of using kNN for outlier detection
"""
# Author: Yue Zhao <zhaoy@cmu.edu>
# License: BSD 2 clause

from __future__ import division
from __future__ import print_function

import os
import sys

# temporary solution for relative imports in case pyod is not installed
# if pyod is installed, no need to use the following line
sys.path.append(
    os.path.abspath(os.path.join(os.path.dirname("__file__"), '..')))

from pyod.models.knn import KNN
from pyod.utils.data import generate_data
from pyod.utils.data import evaluate_print
from pyod.utils.example import visualize

if __name__ == "__main__":
    contamination = 0.1 # percentage of outliers
    n_train = 200 # number of training points
    n_test = 100 # number of testing points

    # Generate sample data
    X_train, X_test, y_train, y_test = \
        generate_data(n_train=n_train,
                      n_test=n_test,
                      n_features=2,
                      contamination=contamination,
                      random_state=42)
```

```
# train kNN detector
clf_name = 'KNN'
clf = KNN()
clf.fit(X_train)

# get the prediction labels and outlier scores of the training data
y_train_pred = clf.labels_ # binary labels (0: inliers, 1: outliers)
y_train_scores = clf.decision_scores_ # raw outlier scores

# get the prediction on the test data
y_test_pred = clf.predict(X_test) # outlier labels (0 or 1)
y_test_scores = clf.decision_function(X_test) # outlier scores

# evaluate and print the results
print("\nOn Training Data:")
evaluate_print(clf_name, y_train, y_train_scores)
print("\nOn Test Data:")
evaluate_print(clf_name, y_test, y_test_scores)

# visualize the results
visualize(clf_name, X_train, y_train, X_test, y_test, y_train_pred,
          y_test_pred, show_figure=True, save_figure=True)
```

We can see from the code how a training and test set is produced using the PyOd function "generate_data" in which a percentage of outliers is also specified. Then the type of detector is specified and the model is compiled.
Predictions are then made on the training set and then the results are evaluated on the test set. Finally, the results are printed on the console and through a graphics window, using the matplotlib library.
Using "generate_data", "normal data is generated by a multivariate Gaussian and outliers are generated by a uniform distribution".

3.3 LSCP

LSCP, or "Locally Selective Combination in Parallel Outlier Ensembles" is an unsupervised framework to selectively combine base detectors by emphasizing data locality[ZNHL19]. It has been developed and published from the author of PyOd, as we can see from the Github repository proprietary. We can see a high-level description of how LSCP works in 8

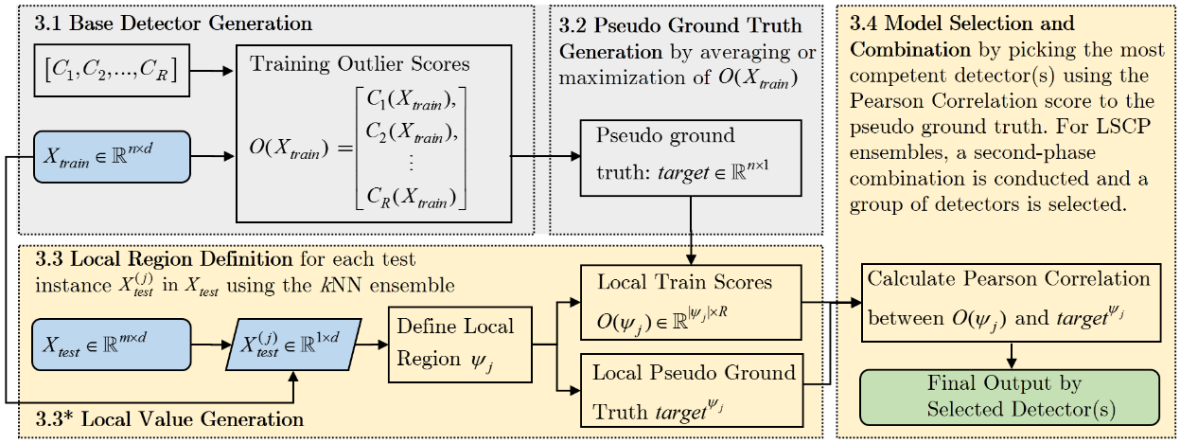

Figure 8: Flowchart of LSCP

3.3.1 Installation

Requirements:

- Python 3.6
- numpy>=1.13
- numba>=0.35
- scipy>=0.19
- scikit_learn>=0.19

And the installation is as simple as:

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

3.3.2 Use and comments

Regarding the installation of LSCP, a version higher than 0.21 of scikit_learn causes the framework not to work as a deprecated function is used (funcsign). Then the requirements ask to install numba with a version number greater than 0.35, that requires pip greater than 19, otherwise LLVM, a dependency of numba can't be installed. The version of python, pip and the modules used in this project makes the use incredibly difficult to perform even in a clean environment from any installation. For this reason, a online google Colab notebook has been used to analyze the framework. The colab can be found at this link: LSCP Colab Demo. This colab has been devolped from Saurav Singla, a data scientis and ML expert for educational purpose and publicly available. The repository include 20 public datasets, and a demo is executable using this command:

```
python demo_lof.py
```

3.4 TODS

TODS, that stands for automated Time-series Outlier Detection System, is a full-stack machine learning system for outlier detection on multivariate time-series data[LZW+20]. It has been developed by DATA Lab, a research hub of the Rice University, in Houston. It provides modules for all the basis functionality like data processing, time series processing, feature analysis and extraction. Tods gave access to different algorithms based on the type of detection scenarios.

3.4.1 Installation

Requirements: Python 3.7+, pip 19+

Since the installation guide is provided only for Debian based machine, a custom installation has been made. Some package has been not installed, some has been installed using the Aur repository name, like "openblas" instead of "libopenblas". After installing the necessary packages in the container, the modules needed by the python interpreter were installed with pip

```
pip install -e .
```

3.4.2 Use and comments

```
import pandas as pd

from axolotl.backend.simple import SimpleRunner

from tods import generate_dataset, generate_problem
from tods.searcher import BruteForceSearch

# Some information
table_path = 'datasets/yahoo_sub_5.csv'
```

```
target_index = 6 # what column is the target
time_limit = 30 # How many seconds you wanna search
metric = 'F1_MACRO' # F1 on both label 0 and 1

# Read data and generate dataset and problem
df = pd.read_csv(table_path)
dataset = generate_dataset(df, target_index=target_index)
problem_description = generate_problem(dataset, metric)

# Start backend
backend = SimpleRunner(random_seed=0)

# Start search algorithm
search = BruteForceSearch(problem_description=problem_description,
                          backend=backend)

# Find the best pipeline
best_runtime, best_pipeline_result = search.search_fit(input_data=[dataset],
    time_limit=time_limit)
best_pipeline = best_runtime.pipeline
best_output = best_pipeline_result.output

# Evaluate the best pipeline
best_scores = search.evaluate(best_pipeline).scores
```

The author give also three different demo, very useful in order to try the different scenarios of time series anomaly detection. General demo. Fraud detection demo. Blockchain demo. The first is a general example where different algorithms and dataset are used to demonstrate how TODS can be used. The second is an example of a fraud payment scenario and different algorithms are used to analyze the differencies. The last notebook is a blockchain outlier detection example, the Google BigQuery Bitcoin Blockchain Dataset is used for retrieving a real blockchain to be analyzed.

3.5 PyOdds

PyOdds is an end-to end Python system for outlier detection with database support. PyOdds provides outlier detection algorithms that can be used both from ML experts and by IT freshman. It is developed by DATA Lab at Texas A&M University This system allows to use 14 algorithms, like autoencoder and KNN or Linkedin Luminol.

3.5.1 Installation

Requirements: Python 3.6+

The installation steps are very easy, but several modules are necessary, like:

- pandas>=0.25.0
- taos==1.4.15
- tensorflow==2.0.0b1
- numpy>=1.16.4
- seaborn>=0.9.0
- torch>=1.1.0
- luminol==0.4
- tqdm>=4.35.0
- matplotlib>=3.1.1
- scikit_learn>=0.21.3

After the installation of these modules, with:

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

We can install the actual pyodds, with:

```
pip install pyodds
pip install git+git@github.com:datamllab/PyODDS.git
```

3.5.2 Use and comments

A simple demo of the Api is given in the repository:

```
from utils.import_algorithm import algorithm_selection
from utils.utilities import output_performance,connect_server,query_data

# connect to the database
conn,cursor=connect_server(host, user, password)

# query data from specific time range
data = query_data(database_name,table_name,start_time,end_time)

# train the anomaly detection algorithm
clf = algorithm_selection(algorithm_name)
clf.fit(X_train)

# get outlier result and scores
prediction_result = clf.predict(X_test)
outlierness_score = clf.decision_function(test)

#visualize the prediction_result
visualize_distribution(X_test,prediction_result,outlierness_score)
```

3.6 MetaOd

MetaOd is an application that performs "Automated Outlier Detection via Meta-Learning". Simply, MetaOd tries to, given a dataset, find the most performing outlier detection models for it. The publisher is the same of PyOd and LSCP.[ZRA20] MetaOd paper claims that his effectiveness in selecting a detection model that significantly outperforms the most popular outlier detection algorithms like LOF and iForest. Furthermore, it is proved that this method does not suffer from speed problems. A description of how MetaOd works can be seen in 9

3.6.1 Installation

- Python 3.5, 3.6, or 3.7
- joblib>=0.14.1
- liac-arff
- numpy>=1.18.1
- scipy>=0.20
- scikit_learn==0.22.1
- pandas>=0.20
- pyod>=0.8

In order to install MetaOd, is possible to use pip or to clone the repository

```
pip install metaod # normal install
```

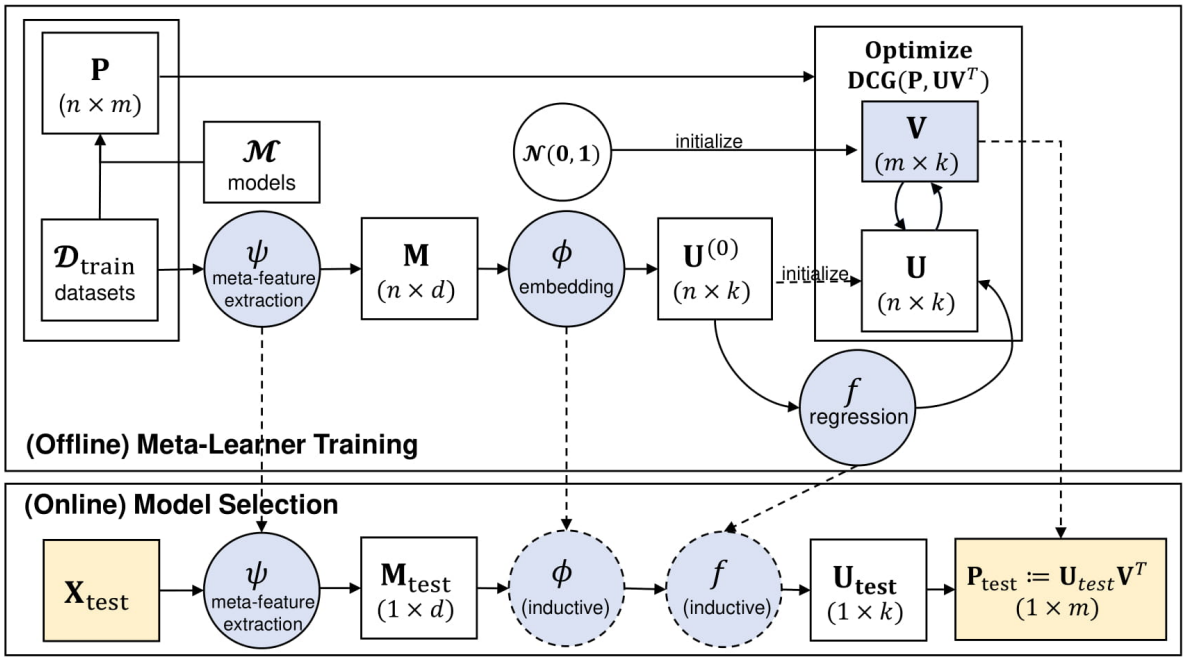

Figure 9: Flowchart of MetaOd

or

```
git clone https://github.com/yzhao062/metaod.git
cd metaod
pip install .
```

3.6.2 Use and comments

An api for selecting the outlier detection model can be written in only three lines of code is provided in the repo:

```
from metaod.models.utility import prepare_trained_model
from metaod.models.predict_metaod import select_model

# load pretrained MetaOD model
prepare_trained_model()

# use MetaOD to recommend models. It returns the top n model for new data X_train
selected_models = select_model(X_train, n_selection=100)
```

3.7 Stats

Method name	Fork	Stars	License
TODS	91	644	Apache-2.0
Pyodds	34	199	MIT License
LSCP	12	23	BSD-2-Clause
PyOd	1152	6060	BSD-2-Clause
MetaOd	17	104	BSD-2-Clause

In the next 2 flowchart we can see how many publications has been written by year 10 in order to analyze the search trend of AutoML and anomaly detection and the type of the model of every publication 11

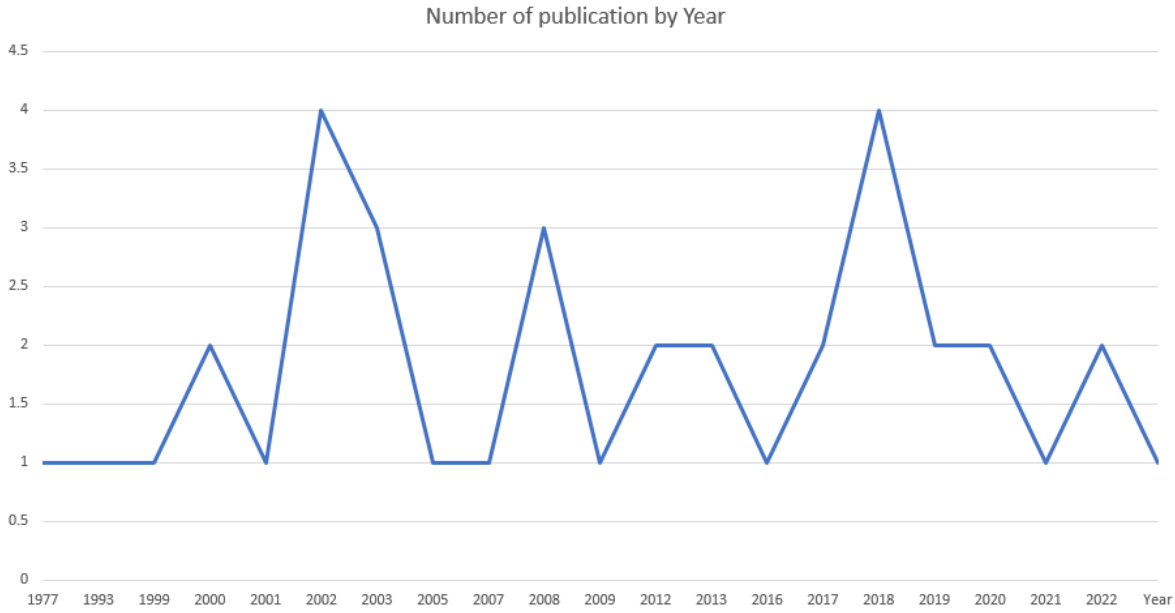

Figure 10: Publications per year

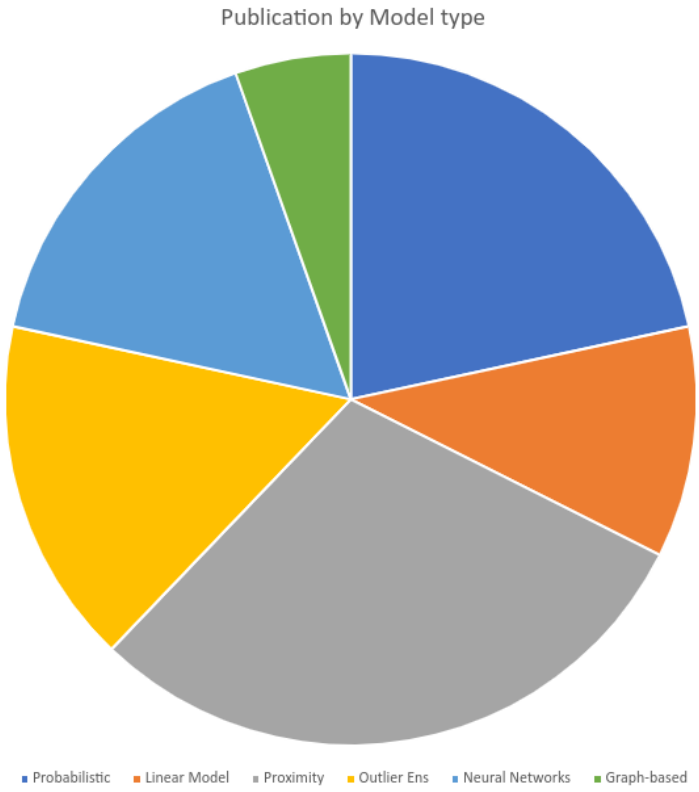

Figure 11: Publication by model type

All the frameworks make no assumption on the target machine, only TODS require some additional steps that are more easily made on a Linux machine. All the frameworks are developed in Python, so they can run on Windows machine as well as on Linux or Mac. All the framework are proved to run on local machine and on Google Colab (except where specified), using the provided Demo as example.

3.8 Algorithms literature

Many algorithms have been developed and published, but have not achieved popularity and no publicly functioning versions or prototypes have been released. Among the various algorithms, which are used by PyOd, we can highlight the main categories which are probabilistic, linear, proximity-based, outlier ensembles, neural networks and graph-based models.

Method type	Abbr	Algorithm	Year
Probabilistic	ECOD [LZH+22]	Empirical Cumulative Distribution Functions	2022
Probabilistic	ABOD [KSZ08]	Angle-Based Outlier Detection	2008
Probabilistic	FastABOD [KSZ08]	Fast Angle-Based Outlier Detection using approximation	2008
Probabilistic	COPOD [LZB+20]	COPOD: Copula-Based Outlier Detection	2020
Probabilistic	MAD [Hoa13]	Median Absolute Deviation	1993
Probabilistic	SOS [Jan13]	Stochastic Outlier Selection	2012
Probabilistic	KDE [LLP07]	Outlier Detection with Kernel Density Functions	2007
Probabilistic	Sampling [SB13]	Rapid distance-based outlier detection via sampling	2013
Probabilistic	GMM	Probabilistic Mixture Modeling for Outlier Analysis	-
Linear Model	PCA [SCSC03]	Principal Component Analysis	2003
Linear Model	MCD [HR04] [RD99]	Minimum Covariance Determinant	1999
Linear Model	CD [Coo77]	Use Cook's distance for outlier detection	1977
Linear Model	OCSVM [SPST+01]	One-Class Support Vector Machines	2001
Linear Model	LMDD [AAR96]	Deviation-based Outlier Detection	1996
Proximity	LOF [BKNS00]	Local Outlier Factor	2000
Proximity	COF [TCFC02]	Connectivity Outlier Factor	2002
Proximity	(Inc.) COF [TCFC02]	Memory Efficient Connectivity Outlier Factor	2002
Proximity	CBLOF [HXD03]	Clustering Local Outlier Factor	2003
Proximity	LOCI [KGPF03]	LOCI: Fast outlier detection using the local correlation integral	2003
Proximity	HBOS [GD12]	Histogram Outlier Score	2012
Proximity	kNN [RRS00]	k Nearest Neighbors	2000
Proximity	AvgKNN [AP02]	Average kNN	2002
Proximity	MedKNN [AP02]	Median kNN	2002
Proximity	SOD [KKSZ09]	Subspace Outlier Detection	2009
Proximity	ROD [ABC22]	Rotation Outlier Detection	2020
Outlier Ens	IForest [LTZ08]	Isolation Forest	2008
Outlier Ens	INNE [BTA+18]	Isolation-based Anomaly Detection Using NN Ensembles	2018
Outlier Ens	FB [LK05]	Feature Bagging	2005
Outlier Ens	XGBOD [ZH18]	Extreme Boosting Based Outlier Detection	2018
Outlier Ens	LODA [Pev15]	Lightweight On-line Detector of Anomalies	2016
Outlier Ens	SUOD [ZHC+21]	Accelerating Large-scale Unsupervised Heterogeneous Outlier Det	2021
Neural Net	AutoEncoder	Fully connected AutoEncoder	-
Neural Net	VAE [KW13]	Variational AutoEncoder	2013
Neural Net	Beta-VAE	Variational AutoEncoder	2018
Neural Net	SO_GAAL [LLZ+18]	Single-Objective Generative Adversarial Active Learning	2019
Neural Net	MO_GAAL [LLZ+18]	Multiple-Objective Generative Adversarial Active Learning	2019
Neural Net	DeepSVDD [RGD+18]	Deep One-Class Classification	2018
Neural Net	AnoGAN [SSW+17]	Anomaly Detection with Generative Adversarial Networks	2017
Graph-based	R-Graph	Outlier detection by R-graph	2017
Graph-based	LUNAR	LUNAR: Unifying Local Outlier Detection Methods	2022

4 Present and future challenge

Since AutoML is still in its discovery phase by the ML and Deep Learning researchers, there are lot of open question and challenges. The same goes for the anomaly detection, although this field has been studied for many more years. Certainly, using automated approaches in this field of machine learning can lead to better results, helping researchers in the choice of models, in the automatic tuning of hyperparameters and much more.
Furthermore, AutoML, by simplifying many aspects that are often difficult for less experienced developers to evaluate and understand, can lead to what is called democratization. This means that lot of person with different background and knowledge, can bring improvements in these topics often reserved only for the elite of researchers.
the field of anomaly detection is constantly growing like almost all other fields of Machine Learning, but there are some problems that are slowing down its progress. Among others we can note the management of unbalanced datasets, the rarity of labeled datasets to be able to test multiple supervised algorithms on different datasets.
This is visible for example by looking at the services that cloud providers make available on their platforms. Google, Microsoft, Amazon and others developed and commercialized several ML tools that is completely based on the cloud. Some of these Saas are so simple to use that often you don't even need lines of code. Many of them can be used dragging and dropping elements of the UI, or using dropdown menu. Many other instead gives the developer API for using the tools.
Given and considered what the current scenario is, it is reasonable to think that the AutoML and the field of anomaly detection can make significant progress in the coming years, both considering each as independent and as joint systems.

References

[AAR96] Andreas Arning, Rakesh Agrawal, and Prabhakar Raghavan. A linear method for deviation detection in large databases. In *KDD*, 1996.

[ABC22] Yahya Almardeny, Noureddine Boujnah, and Frances Cleary. A novel outlier detection method for multivariate data. *IEEE Transactions on Knowledge and Data Engineering*, 34(9):4052–4062, 2022.

[AP02] Fabrizio Angiulli and Clara Pizzuti. Fast outlier detection in high dimensional spaces. In *PKDD*, 2002.

[BKNS00] Markus Breunig, Hans-Peter Kriegel, Raymond Ng, and Joerg Sander. Lof: Identifying density-based local outliers. volume 29, pages 93–104, 06 2000.

[BSPS22] Maroua Bahri, Flavia Salutari, Andrian Putina, and Mauro Sozio. AutoML: state of the art with a focus on anomaly detection, challenges, and research directions. *International Journal of Data Science and Analytics*, February 2022.

[BTA+18] Tharindu R. Bandaragoda, Kai Ming Ting, David Albrecht, Fei Tony Liu, Ye Zhu, and Jonathan R. Wells. Isolation-based anomaly detection using nearest-neighbor ensembles. *Computational Intelligence*, 34(4):968–998, November 2018.

[Coo77] R. Dennis Cook. Detection of influential observation in linear regression. *Technometrics*, 19(1):15–18, 1977.

[GD12] Markus Goldstein and Andreas R. Dengel. Histogram-based outlier score (hbos): A fast unsupervised anomaly detection algorithm. 2012.

[HHH+22] Songqiao Han, Xiyang Hu, Hailiang Huang, Mingqi Jiang, and Yue Zhao. Adbench: Anomaly detection benchmark, 2022.

[HKV19] Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren, editors. *Meta-Learning*, pages 35–61. Springer International Publishing, Cham, 2019.

[Hoa13] David C. Hoaglin. Volume 16: How to detect and handle outliers. 2013.

[HR04] Johanna Hardin and David M. Rocke. Outlier detection in the multiple cluster setting using the minimum covariance determinant estimator. *Computational Statistics & Data Analysis*, 44(4):625–638, January 2004.

[HXD03] Zengyou He, Xiaofei Xu, and Shengchun Deng. Discovering cluster-based local outliers. *Pattern Recogn. Lett.*, 24(9–10):1641–1650, jun 2003.

[Jan13] J.H.M. Janssens. *Outlier selection and one-class classification*. PhD thesis, 2013. Series: TiCC Ph.D. Series Volume: 27.

[KGPF03] H. Kitagawa, P. B. Gibbons, S. Papadimitriou, and C. Faloutsos. Loci: Fast outlier detection using the local correlation integral. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, page 315, Los Alamitos, CA, USA, mar 2003. IEEE Computer Society.

[KKSZ09] Hans-Peter Kriegel, Peer Kröger, Erich Schubert, and Arthur Zimek. Outlier detection in axis-parallel subspaces of high dimensional data. In *Proceedings of the 13th Pacific-Asia Conference on Advances in Knowledge Discovery and Data Mining*, PAKDD '09, page 831–838, Berlin, Heidelberg, 2009. Springer-Verlag.

[KSZ08] Hans-Peter Kriegel, Matthias Schubert, and Arthur Zimek. Angle-based outlier detection in high-dimensional data. In *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '08, page 444–452, New York, NY, USA, 2008. Association for Computing Machinery.

[KW13] Diederik P Kingma and Max Welling. Auto-encoding variational bayes, 2013.

[LK05] Aleksandar Lazarevic and Vipin Kumar. Feature bagging for outlier detection. volume 21, pages 157–166, 01 2005.

[LLP07] Longin Jan Latecki, Aleksandar Lazarevic, and Dragoljub Pokrajac. Outlier detection with kernel density functions. In *MLDM*, 2007.

[LLZ+18] Yezheng Liu, Zhe Li, Chong Zhou, Yuanchun Jiang, Jianshan Sun, Meng Wang, and Xiangnan He. Generative adversarial active learning for unsupervised outlier detection, 2018.

[LTZ08] Fei Tony Liu, Kai Ming Ting, and Zhi-Hua Zhou. Isolation forest. In *2008 Eighth IEEE International Conference on Data Mining*, pages 413–422, 2008.

[LZB+20] Zheng Li, Yue Zhao, Nicola Botta, Cezar Ionescu, and Xiyang Hu. COPOD: Copula-based outlier detection. In *2020 IEEE International Conference on Data Mining (ICDM)*. IEEE, nov 2020.

[LZH+22] Zheng Li, Yue Zhao, Xiyang Hu, Nicola Botta, Cezar Ionescu, and George H. Chen. Ecod: Unsupervised outlier detection using empirical cumulative distribution functions, 2022.

[LZW+20] Kwei-Herng Lai, Daochen Zha, Guanchu Wang, Junjie Xu, Yue Zhao, Devesh Kumar, Yile Chen, Purav Zumkhawaka, Minyang Wan, Diego Martinez, and Xia Hu. Tods: An automated time series outlier detection system, 2020.

[Omi22] Andrea Omicini. S2 – systematic literature review. *ISE slides 2021-2022*, 2022.

[Pev15] Tomás Pevný. Loda: Lightweight on-line detector of anomalies. *Machine Learning*, 102:275–304, 2015.

[RAHL19] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V. Le. Aging evolution for image classifier architecture search. 2019.

[RD99] Peter J. Rousseeuw and Katrien Van Driessen. A fast algorithm for the minimum covariance determinant estimator. *Technometrics*, 41(3):212–223, 1999.

[RGD+18] Lukas Ruff, Nico Görnitz, Lucas Deecke, Shoaib Ahmed Siddiqui, Robert A. Vandermeulen, Alexander Binder, Emmanuel Müller, and Marius Kloft. Deep one-class classification. In *ICML*, pages 4390–4399, 2018.

[RRS00] Sridhar Ramaswamy, Rajeev Rastogi, and Kyuseok Shim. Efficient algorithms for mining outliers from large data sets. *SIGMOD Rec.*, 29(2):427–438, may 2000.

[SB13] Mahito Sugiyama and Karsten Borgwardt. Rapid distance-based outlier detection via sampling. In C.J. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 26. Curran Associates, Inc., 2013.

[SCSC03] Mei-Ling Shyu, Shu-Ching Chen, Kanoksri Sarinnapakorn, and LiWu Chang. A novel anomaly detection scheme based on principal component classifier. 2003.

[SPST+01] Bernhard Schölkopf, John C. Platt, John Shawe-Taylor, Alex J. Smola, and Robert C. Williamson. Estimating the Support of a High-Dimensional Distribution. *Neural Computation*, 13(7):1443–1471, 07 2001.

[SSW+17] Thomas Schlegl, Philipp Seeböck, Sebastian M. Waldstein, Ursula Schmidt-Erfurth, and Georg Langs. Unsupervised anomaly detection with generative adversarial networks to guide marker discovery, 2017.

[STDA05] Alexander Statnikov, Ioannis Tsamardinos, Yerbolat Dosbayev, and Constantin F. Aliferis. Gems: A system for automated cancer diagnosis and biomarker discovery from microarray gene expression data. *International Journal of Medical Informatics*, 74(7):491–503, 2005. MedInfo 2004.

[TCFC02] Jian Tang, Zhixiang Chen, Ada Wai-Chee Fu, and David W Cheung. Enhancing effectiveness of outlier detections for low density patterns. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 535–548. Springer, 2002.

[ZH18] Yue Zhao and Maciej K. Hryniewicki. XGBOD: Improving supervised outlier detection with unsupervised representation learning. In *2018 International Joint Conference on Neural Networks (IJCNN)*. IEEE, jul 2018.

[ZHC+21] Yue Zhao, Xiyang Hu, Cheng Cheng, Changlin Wan, Wen Wang, Jianing Yang, Haoping Bai, Zheng Li, Cao Xiao, Yunlong Wang, Zhi Qiao, J. Sun, and Leman Akoglu. Suod: Accelerating large-scale unsupervised heterogeneous outlier detection. 01 2021.

[ZL17] Barret Zoph and Quoc V. Le. Neural architecture search with reinforcement learning. 2017.

[ZNHL19] Yue Zhao, Zain Nasrullah, Maciej K Hryniewicki, and Zheng Li. LSCP: locally selective combination in parallel outlier ensembles. In *Proceedings of the 2019 SIAM International Conference on Data Mining, SDM 2019*, pages 585–593, Calgary, Canada, May 2019. SIAM.

[ZNL19] Yue Zhao, Zain Nasrullah, and Zheng Li. Pyod: A python toolbox for scalable outlier detection. *Journal of Machine Learning Research*, 20(96):1–7, 2019.

[ZRA20] Yue Zhao, Ryan Rossi, and Leman Akoglu. Automating outlier detection via meta-learning. *arXiv preprint arXiv:2009.10606*, 2020.