EC3.21

Methodological Guidelines for OODeel library

L3.5.2.3

contact@confiance-ai.fr | www.confiance.ai

Document reference: 321DC

Contributors

	Name	Organisation	Role
Responsible for the deliverable	Yannick Prudent	IRT Saint Exupéry	Main contributor EC3.21 (OODeel)
Scientific responsible	Corentin Friedrich	IRT Saint Exupéry	Main contributor EC3.21 (OODeel)
Co-authors			

Document Control

Revision	Date	Commentary	Author
v1.0			

Contents

A. Introduction and abstract

A.1. General introduction to trustworthy AI challenges

Trustworthiness in AI within critical systems (systems that can directly or indirectly affect human life and moral entities) is essential for its widespread adoption (by the industry, the decision makers, the general public, etc.) and poses the following significant challenges.

- First, how to design AI models, so that, by construction, they satisfy trustworthy properties (accuracy, robustness. . .).
- Secondly, how to characterize these AI models, for example to understand and explain their behavior and their adequacy to the operational domain.
- Then, how to implement and embed those AI models on hardware, by making them fit for the target without losing their trustworthy properties.
- Another question is, what methods of data engineering to apply in order to, among other topics, manage important volumes of data and adapt to the evolution of the operational domain.
- At system level, what verification and certification processes to consider specifically for AI-based systems.
- Finally, a federation of all these matters is necessary to build an end-to-end methodological approach, supported by a consistent engineering environment compatible with industrial practices.

These are the challenges, among others, that the Confiance.ai program addresses.

A.2. Out-of-Distribution detection

Machine learning models face a common challenge when dealing with out-of-distribution (OOD) inputs. These are essentially data samples from a different distribution than what the model has encountered during its training phase. Consequently, a dependable classifier should not only excel at accurately categorizing known in-distribution (ID) samples but also be able to label any OOD input as "unknown." This highlights the critical importance of OOD detection, a process that discerns whether an input belongs to the ID or OOD category and equips the model to take necessary precautions. In this context, our focus centers on post-hoc OOD detection methods, which offer the significant advantage of ease of use without necessitating alterations to the training process and objectives. This attribute is particularly valuable for the adoption of OOD detection methods in real-world production settings, where the costs and complexities associated with retraining models can be prohibitive.

A.3. OODeel library

The OODeel[1] python library is specifically developed for post-hoc deep Out-of-Distribution (OOD) detection, with a focus on pre-trained neural network image classifiers. The underlying

[1] OODeel documentation is available here: https://deel-ai.github.io/oodeel/

ethos of this library is the prioritization of performance and user-friendliness over sheer quantity. Consequently, OODeel offers a streamlined, adaptable API that is thoughtfully designed for simplicity and ease of adoption. It incorporates and validates various baseline methods into a unified framework, ensuring compatibility with TensorFlow and PyTorch frameworks.

A.4. Abstract of this Guidelines document

This section provides a concise overview of the document, summarizing the scientific challenges addressed, trustworthiness attributes considered, target audience, and the overall organization of the guidelines.

Scientific challenges addressed
The scientific challenges addressed by OODeel library are:

- Components with integrated self-monitoring of the detection of the exit of the operating zone
- Calculation of the confidence score by various approaches (ensemble, calibration, introspection)

Trustworthiness attributes addressed
The OODeel component address the *Out-of-Distribution Detection* trustworthiness attribute.

Target audience
This document is meant to be useful for the following audience:

- Data scientists, AI algorithm engineers, software engineers, or any users interested in identifying out-of-distribution samples for their neural networks within specific use cases.
- System engineers and other users aiming to seamlessly integrate out-of-distribution sample detection into their classification pipelines.
- System IV&V engineers, safety engineers, anyone interested in identifying abnormal samples prone to incorrect predictions by their models.

The OODeel Python library serves as a core tool to assist the aforementioned target users with these tasks. These guidelines will primarily focus on the correct usage of this library.

Document Organization
This guidelines document is structured to comprehensively address the utilization of the OODeel Python library for posthoc out-of-distribution detection in classification neural networks. The document unfolds in the following manner:

- First, chapter B offers a concise exploration of the OOD detection problem, accompanied by a discussion on the typical metrics prevalent in the literature for evaluating OOD detection performance.
- Then, serving as an introduction to the OODeel library, chapter C provides an overview of the standard pipeline employed for OOD detection.
- Finally, chapter D constitutes a focal point, delivering explicit guidelines on tuning OOD detection methods and aiding users in selecting the most suitable approach for their specific requirements.

B. Preliminaries on OOD detection

This chapter delves into the foundational aspects of out-of-distribution (OOD) detection in machine learning. It begins by addressing the core problem of determining whether a given sample resides within the training distribution or is out-of-distribution, emphasizing the role of a detection threshold. Subsequently, the discussion shifts to essential OOD metrics, such as AUROC, FPR95TPR, and TPR5FPR, pivotal for evaluating detection performance. These fundamental concepts set the stage for a comprehensive understanding of OOD detection methodologies and guidelines presented in the ensuing chapters.

B.1. OOD detection problem

The aim of an OOD detector is to decide whether a sample $x \in \mathscr{X}$ is from the training distribution $\mathscr{P}_{in}$ or not. The detection can thus be formulated as a binary classification problem, where positives correspond to out-of-distribution (OOD) data, and negatives to in-distribution (ID) data. Then, the decision is made in relation to a detection threshold λ, chosen such that:

$$G_\lambda(x) = \begin{cases} \text{OOD} & S(x) \geq \lambda \\ \text{ID} & S(x) < \lambda \end{cases}$$

where $S(x)$ is the score returned by the OOD detector.

B.2. Usual OOD metrics

To assess the performance of an OOD detector, we evaluate its ability to distinguish between in-distribution and out-of-distribution (OOD) data. Typically, the test set of the classifier represents in-distribution (ID) data, while other datasets with alternative distributions serve as OOD data. In this benchmark, we will evaluate different detectors over these three metrics: AUROC, FPR95TPR, and TPR5FPR.

- **AUROC**: AUROC (Area Under the Receiver Operating Characteristic curve) is a widely utilized metric for assessing binary classifier performance. It quantifies the detector's ability to differentiate between positive and negative samples across all possible decision thresholds (λ). To construct the AUROC, we follow these steps:

  1. Calculate the true positive rate (TPR) and false positive rate (FPR) for various threshold values of λ.
  2. Plot a receiver operating characteristic (ROC) curve with TPR on the y-axis and FPR on the x-axis. This curve represents the classifier's performance at different decision thresholds.
  3. The AUROC is determined by computing the area under the ROC curve. It ranges from 0 to 1, where 0.5 indicates random performance, and a value of 1 signifies perfect discrimination.

- **FPR95TPR**: This metric calculates the false positive rate when the true positive rate is held constant at 95%. In simpler terms, it answers the question: "If I want to be really sure I catch 95% of OOD data, how often will the In-Distribution (ID) data be wrongly classified as OOD?"

- **TPR5FPR**: This metric calculates the true positive rate when the false positive rate is held constant at 5%. Put differently, it addresses the question: "If I can tolerate a 5% rate of mistakenly identifying ID data as OOD, how many of the actual OOD data can I successfully identify?" This metric is less commonly used in research but could be particularly relevant for manufacturers or systems with strict constraints on false alarms.

C. OODeel: Getting started

In this chapter, we provide an in-depth description of the modules and features offered by the OODeel library. It serves as a comprehensive guide to understanding the library's capabilities and how to leverage them for effective OOD detection. We begin by covering installation instructions, before going on to an overview of the OODeel pipeline API.

C.1. Installation of OODeel

The OODeel library can be readily installed through PyPI using the following command:

```
pip install oodeel
```

Tensorflow or PyTorch is expected to be already installed on the system and OODeel will not automatically install these dependencies to avoid potential conflicts with existing installations. Note that the library undergoes regular testing to ensure compatibility with the following Python and framework versions:

- Python 3.8 + (Torch 1.11 or TensorFlow 2.5)
- Python 3.9 + (Torch 1.13 or TensorFlow 2.8)
- Python 3.10 + (Torch 2.0 or TensorFlow 2.11)

Users are encouraged to set up their Python environments with one of these compatible configurations to ensure the smooth and reliable operation of the OODeel library.

C.2. Overview of the OODeel Pipeline API

The OODeel library offers a comprehensive pipeline for OOD detection, including data loading, classifier training, OOD detector fitting, and result evaluation. In this section, we provide an overview of the key components within each module of the library.

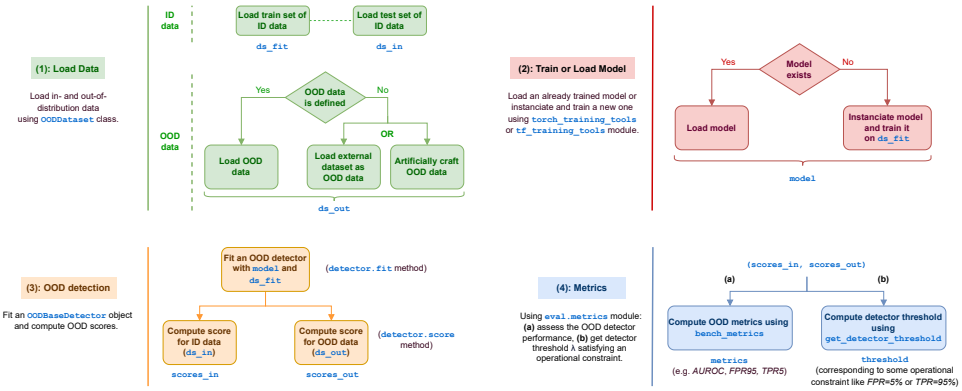

Figure C.1: Diagram of OODeel detection pipeline.

C.2.1 Loading Data

OOD detection begins with the loading of datasets and OODeel simplifies this process through the `OODDataset` class.

With the `backend` argument, users can seamlessly transition between the `"torch"` and `"tensorflow"` deep learning frameworks. When specifying a string for the `dataset_id` argument, `OODDataset` automatically searches for and retrieves the dataset from either the `tensorflow_datasets` or `torchvision.datasets` catalogs. Alternatively, users have the flexibility to provide their custom dataset using the same argument. Then, the `prepare` method offers various features, including data preprocessing, batching, and shuffling, for the loaded dataset. This functionality streamlines the entire process. Code C.2.1 shows an example of this pipeline with the MNIST dataset.

```python
from oodeel.datasets import OODDataset

backend = "torch"

# === Load MNIST Train Set ===
oods_fit = OODDataset(
    dataset_id='MNIST',
    load_kwargs={"train": True, "root": data_path, "download": True},
    backend=backend,
)

# === Preprocessing, Batching, Shuffling... ===
# Note: The following code is backend-agnostic.
def preprocess_fn(*inputs):
    x = inputs[0] / 255
    return tuple([x] + list(inputs[1:]))

# Get a DataLoader for torch or a ready-to-train tf.Dataset for TensorFlow.
ds_fit = oods_fit.prepare(
    batch_size=128, preprocess_fn=preprocess_fn, shuffle=True
)
```

Code C.1: Loading data pipeline for MNIST dataset.

C.2.2 Training tools

Training a classifier is a fundamental step in OOD detection, and OODeel provides a user-friendly interface for this task. Users can train a classifier with ease using the library's training tools. Code C.2.2 shows an example of how to train a classifier with OODeel on torch framework. The equivalent function, `train_tf_model`, is available for TensorFlow with a closely similar syntax.

```python
from oodeel.torch_training_tools import train_torch_model

# Define your training configuration in a dictionary
train_config = {
    "model": "resnet18",
    "num_classes": num_classes,
```

```
    "epochs": epochs,
    "save_dir": model_path,
    "validation_data": ds_in, # Validation set
    "cuda_idx": cuda_idx
}
# The train_torch_model function will automatically instantiate, train,
#   and save your model.
model_torch = train_torch_model(ds_train,  **train_config)
```

Code C.2: Training a classifier with OODeel (torch example).

C.2.3 OOD Detection

After loading the classifier, the OODeel library empowers users to effortlessly integrate an OOD detection method. With just a few lines of code, users can fit the model – as well as the training dataset if required by the OOD detection method – and compute OOD scores on both in-distribution and out-of-distribution datasets. Code C.2.3 shows an example using the simple baseline method MLS (Maximum Likelihood Score). Note that the syntax is agnostic to the underlying deep learning framework.

```
from oodeel.methods import MLS

# Initialize the MLS OOD detector
mls = MLS()

# Fit the OOD detector to the classifier model
mls.fit(model)

# Compute OOD scores for ID and OOD datasets
scores_in, info_in = mls.score(ds_in)
scores_out, info_out = mls.score(ds_out)
```

Code C.3: Using MLS (Maximum Likelihood Score) for OOD detection and retrieving scores.

C.2.4 Metrics and Plots

Once the OOD scores for both in-distribution and out-of-distribution data have been computed, the library provides users with the capability to assess OOD detection performance using widely-recognized domain-specific metrics (as introduced in the following chapter). Furthermore, users can visualize these results through various plots. An illustrative example is presented in Code C.2.4. Once again, the whole code is framework agnostic.

```
from oodeel.eval.metrics import bench_metrics
from oodeel.eval.plots import plot_ood_scores, plot_roc_curve

# Compute AUROC and FPR95TPR metrics
metrics = bench_metrics(
    (scores_in, scores_out),
    metrics=["auroc", "fpr95tpr"],
)

# Visualize the ROC curve and the distribution of scores
```

```
plot_ood_scores(scores_in, scores_out, log_scale=False)
plot_roc_curve(scores_in, scores_out)
```

Code C.4: Computation of OOD metrics and results visualization.

D. Out-of-Distribution Detection Guidelines

This chapter offers comprehensive guidelines for effectively tuning initialization and fitting hyperparameters of OOD detection methods involved in step 3 of Figure C.1.

D.1. Insights on Feature Extraction

Certain OOD detection methods necessitate access to a model's internal features to forecast OOD scores. Users can specify a specific feature layer by providing its identifier as a string through the `feature_layers_id` parameter within the detector's `fit` method.

To identify layer identifiers within different frameworks, consider the following procedures:

- For **PyTorch** models, users can retrieve the list of layer identifiers by executing: `print(dict(model.named_modules()).keys())`.
- With **TensorFlow** models, users can visualize the entire model, including layer identifiers, by using: `model.summary()`.

Code snippet D.1 presents an example demonstrating this fitting process.

```
# === Initialize OOD detector ===
dknn = DKNN(nearest=50)
# === Fit detector with model and fit_dataset ===
# Note: "avgpool" is the identifier of the feature layer to target
dknn.fit(model, ds_fit, feature_layers_id=["avgpool"])
```

Code D.1: Targeting a feature layer during detector fitting.

D.2. Tuning OOD Detector Hyperparameters

D.2.1 Logit-based Methods: MLS, MSP, Energy, Entropy

Fundamental OOD detection methods involve computing statistics using logit information exclusively to derive an OOD score. These are referred to as "logit-based" detection methods, encompassing MLS ("Max Logit Score") [1], MSP ("Max Softmax Probability") [2], Energy [3], and Entropy [4]. These methods are known for their swift execution, requiring no dataset fitting or access to internal features of the model.

Here are recommendations for initializing and fitting these methods:

- **Initialization**: Retain default values for the initialization arguments, except for the MSP method, instantiated using the `MLS` class with the argument `output_activation="softmax"`.

- **Fitting**: The fitting process is simply performed with `detector.fit(model)`.

D.2.2 ODIN: Temperature scaling and Input Perturbation

Similar to previous methods, ODIN [5] operates without requiring access to internal network features for predicting an OOD score, categorizing it as a "logit-based" approach. However, ODIN sets itself apart by applying a gradient step to perturb the input image with the aim of amplifying the strongest softmax score, scaled by a temperature factor. Subsequently, the OOD score is derived from the softmax probability of the perturbed input.

Guidelines for initializing and fitting ODIN are outlined as follows:

- **Initialization** (`odin = ODIN(temperature, noise)`):
  - `temperature` argument: This parameter divides the logits before the softmax operation. In the original paper, a default value of 1000 was used for some models trained on CIFAR-10. For fine-tuning, the user should start with a low value (e.g., 1, equivalent to an unscaled softmax), then gradually increase the temperature until further improvement in OOD detection ceases. The original paper recommend to tune this parameter before tweaking the `noise` argument.
  - `noise` argument: This parameter scales the gradient step during input perturbation. The default value in the original paper is 0.0014. For fine-tuning, users should initiate with a value of 0 and slowly increase it until reaching an optimal point.

- **Fitting**: The fitting process is executed simply with `odin.fit(model)`.

D.2.3 ReAct: Enhancing Logit-based Methods

The ReAct ("Rectified Activations") method [6] offers an approach to improve the performance of logit-based OOD detection methods, including MLS, MSP, Energy, Entropy, and ODIN. ReAct achieves this enhancement by applying a threshold-based clipping operation to the activations in the penultimate layer of the neural network.

[Disclaimer]: This method should only be applied to a model when its penultimate layer returns positive features (after a ReLU activation for instance).

Here are practical guidelines for enabling and fine-tuning ReAct within the context of these logit-based OOD detection methods:

- **Initialization**: When initializing MLS, MSP, Energy, Entropy, and ODIN methods, adjust the following arguments:
  - `use_react` argument: Set this parameter to `True` to activate ReAct within the detection process.
  - `react_quantile` argument: The activation threshold is determined as the q-th quantile of penultimate layer activations from the training dataset. This parameter represents the value of q within the range of [0, 1], with a default value of 0.8. For fine-tuning, it is recommended to start with a higher value close to 1 (indicating a softer clipping), gradually decreasing the value to find the quantile that maximizes the OOD detection metrics.

- **Fitting**: To compute the activation threshold for ReAct, the detector needs to calculate a quantile based on activations from a calibration set. Hence, the fitting process should be executed using `detector.fit(model, ds_fit)`, where `ds_fit` represents the training dataset of the classifier. Ensure the dataset is formatted as a `tf.data.Dataset` if using the TensorFlow framework or as a `torch.utils.data.DataLoader` for the PyTorch framework.

D.2.4 VIM: Virtual-logit Matching

VIM ("Virtual-logit Matching") [7] method predicts the OOD nature of an image by computing an additional logit representing the virtual out-of-distribution class. This logit is calculated based on the norm of features projected into the residual subspace $P^{\perp}$ of the model's penultimate layer. This residual subspace is derived by taking the orthogonal complement of the subspace P formed by the first k principal components obtained from the in-distribution (ID) data in the penultimate layer's feature space. The method identifies that out-of-distribution samples tend to exhibit larger norms in $P^{\perp}$ compared to in-distribution samples. Consequently, the norm, multiplied by a scaling factor, is used as the virtual out-of-distribution logit, which is then transformed into a probability after a softmax operation, assessing the OOD-ness of a sample.

Here are recommended guidelines for tuning the hyperparameters of the VIM method:

- **Initialization** (`vim = VIM(princ_dims, pca_origin)`):
  - `princ_dims` argument: This parameters corresponds to the number of principal components k constituting the subspace P. If an integer, it must be smaller than the dimension of the feature space. If a float, it represents the ratio of explained variance to determine the number of principal components for P and should be in the range $[0, 1[$. To fine-tune this hyperparameter, it's advised (1) to select a sufficiently large value to achieve a high ratio of explained variance in the principal components for the in-distribution data, ensuring small norms for in-distribution samples, and (2) to choose a small enough value to capture the residual energy of out-of-distribution samples. The default value of 0.99 serves as a good starting point.
  - `pca_origin` argument: This parameter defines the strategy for determining the distribution center used in PCA computation. It's recommended to retain the default value (`pca_origin="pseudo"`).

- **Fitting** (`vim.fit(model, ds_fit, feature_layers_id)`):
  - `ds_fit` argument: This parameter denotes the training dataset of the classifier used for PCA computation. Ensure the dataset is formatted as a `tf.data.Dataset` for TensorFlow or as a `torch.utils.data.DataLoader` for PyTorch.
  - `feature_layers_id`: Indicate here the identifier of the feature layer, used to compute the virtual logit, following the recommendations in D.1. In general, we point to the model's penultimate layer.

D.2.5 DKNN: Deep K-Nearest-Neighbor

The DKNN ("Deep K-Nearest-Neighbor") method [8] offers a straightforward yet effective approach for OOD detection. The methodology can be summarized as follows:

First, the embeddings of in-distribution samples are projected into the feature space of a classifier and saved for reference. Then, to determine the OOD score of a new sample, the method calculates the k-th smallest distance between this sample's embedding and the embeddings of all in-distribution samples. A smaller distance to the k-th nearest neighbor is indicative of a more in-distribution sample, while a larger distance suggests potential out-of-distribution characteristics.

Here are practical guidelines for fine-tuning the hyperparameters of DKNN:

- **Initialization** (`dknn = DKNN(nearest)`):
  - `nearest` argument: This parameter should be set to an integer k representing the rank in ascending order of the distances to consider for the OOD score. A default value of 50 often serves as a good starting point, but it can be adjusted to optimize performance for specific use cases.

- **Fitting** (`dknn.fit(model, ds_fit, feature_layers_id)`):
  - `ds_fit` argument: This parameter refers to the training dataset of the classifier that will be used for KNN fitting. Ensure that the dataset is correctly formatted as a `tf.data.Dataset` if using the TensorFlow framework, or as a `torch.utils.data.DataLoader` for PyTorch.
  - `feature_layers_id`: Specify the identifier of the feature layer used to compute the distances, following the recommendations provided in Section D.1. Typically, it is advisable to select the model's penultimate layer as the feature layer for DKNN OOD detection.

D.2.6 Mahalanobis: Class-Conditional Mahalanobis Distance

The Mahalanobis approach [9] estimates an OOD score by computing the Mahalanobis distance of a sample concerning the closest class-conditional distribution. This distance is typically calculated in a feature space that aligns with the model's penultimate layer. Similar to the preprocessing in the ODIN approach, the input image undergoes a perturbation using a gradient step to boost its confidence score. This preprocessing step is designed to enhance the distinguishability between in-distribution and out-of-distribution samples.

Here are guidelines for tuning the parameters of the Mahalanobis method:

- **Initialization** (`mahalanobis = Mahalanobis(eps)`):
  - `eps` argument: This parameter scales the gradient step during input perturbation. The default value is set to 0.002. For fine-tuning, users should initiate with a value of 0 and incrementally adjust it to find an optimal value that enhances the separation between in-distribution and out-of-distribution samples.

- **Fitting** (`mahalanobis.fit(model, ds_fit, feature_layers_id)`):
  - `ds_fit` argument: Refers to the training dataset used to compute the means and covariances of each class-conditional distribution. Ensure that the dataset is appropriately formatted as a `tf.data.Dataset` for TensorFlow or as a `torch.utils.data.DataLoader` for PyTorch.
  - `feature_layers_id`: Specify the identifier of the feature layer used to compute

the Mahalanobis distances, following the recommendations provided in Section D.1. Usually, the model's penultimate layer is selected as the feature layer for Mahalanobis-based OOD detection.

D.2.7 Gram: Anomaly Detection via Gram Matrices

The Gram method [10] determines an OOD score by identifying anomalies present in the Gram matrices of samples projected into the feature space of a classifier. These anomalies are assessed by measuring the deviation of each Gram matrix value concerning its observed range over the training data. The final OOD score is calculated by summing the deviations across Gram matrices computed over different feature layers and raised to distinct power orders $p_i \in \mathbb{N}^*$.

Here are practical guidelines for fine-tuning the hyperparameters of the Gram method:

- **Initialization** (`gram = Gram(orders, quantile)`):
  - `orders` argument: This parameter is a list of integers $[p_1, \cdots, p_N] \in (\mathbb{N}^*)^N$ representing the power orders to be considered for the total score. By default, this parameter is the list going from integers 1 to 10. If the computation of the OOD score is slow, users may try reducing the number of orders.
  - `quantile` argument: For a given sample, the deviation is nonzero for Gram matrix values outside the range defined between the q-th and $1-q$-th quantiles of the respective Gram matrix values from the training dataset. This parameter denotes the value of q within the range of [0, 1], with a default value of 0.01. For fine-tuning, start with the default value of 0.01 (where the deviation for in-distribution data is close to 0), and incrementally adjust the value to optimize OOD detection metrics.

- **Fitting** (`gram.fit(model, ds_fit, feature_layers_id)`):
  - `ds_fit` argument: Refers to the training dataset used to compute the quantiles of the Gram matrices. Ensure the dataset is properly formatted as a `tf.data.Dataset` for TensorFlow or as a `torch.utils.data.DataLoader` for PyTorch.
  - `feature_layers_id`: Specify a list of identifiers for the feature layers considered for computing Gram matrices, following the recommendations provided in Section D.1. Typically, the penultimate layer and a few slightly shallower layers are selected for Gram-based OOD detection.

D.3. Selecting the Best OOD Detection Method

Determining the ideal OOD detection method can be challenging due to the variability in results across different datasets and model architectures. This section provide some recommendations to help navigate the selection process.

D.3.1 Variability Across Use Cases and Models

Identifying a universal best-performing method across all use cases is intricate. What works best on one dataset might not yield the same results on another. Additionally, outcomes are often sensitive to the specific model architecture and training hyperparameters used. This variability underscores the importance of employing a diverse array of methods to explore OOD detection comprehensively. The OODeel library offers a suite of methods with tools for easy bench-

marking to assist in method comparison. To achieve the most reliable OOD detection results, it's advisable to systematically try various methods following the pipeline outlined in Section C.2 and adhere to the hyperparameter tuning recommendations in Section D. Subsequently, the methods can be evaluated and ranked based on the OOD metrics described in Section B.2.

D.3.2 Insights from Benchmark Studies

The "Benchmark on OODeel Library" deliverable provides valuable insights into method characteristics, focusing on detection performance and computational efficiency. Some key conclusions from this benchmark analysis include:

- Logit-based approaches exhibit rapid execution and are resource-efficient. These approaches can be interesting when there are strong constraints on the speed of inference. They can easily be enhanced by being coupled with ReAct method, provided that the activations from the penultimate layer are positive.
- DKNN, VIM, and Gram methods showcase promising results according to the benchmark studies. However, they may demand relatively more computational time.

E. Conclusion

This document has outlined a comprehensive guide to the OODeel library, offering a robust pipeline for Out-of-Distribution (OOD) detection in machine learning models. Covering key components such as data loading, classifier training, OOD detector fitting, and result evaluation, users are equipped with versatile tools for effective OOD detection. The document further presented detailed insights and guidelines for ten OOD detection methods, addressing their initialization, fitting, and tuning hyperparameters. While acknowledging the diversity of use cases and models, the document emphasized the importance of systematic exploration and benchmarking to select the most suitable method. As the field evolves, the OODeel library remains committed to advancing OOD detection capabilities, providing users with a dynamic toolkit for enhanced model robustness and reliability.

Bibliography

[1] S Vaze, K Han, A Vedaldi, and A Zisserman. Open-set recognition: A good closed-set classifier is all you need? In *International Conference on Learning Representations (ICLR)*, 2022.

[2] Dan Hendrycks and Kevin Gimpel. A baseline for detecting misclassified and out-of-distribution examples in neural networks. In *International Conference on Learning Representations*, 2016.

[3] Weitang Liu, Xiaoyun Wang, John Owens, and Yixuan Li. Energy-based out-of-distribution detection. *Advances in neural information processing systems*, 33:21464–21475, 2020.

[4] Jie Ren, Peter J Liu, Emily Fertig, Jasper Snoek, Ryan Poplin, Mark Depristo, Joshua Dillon, and Balaji Lakshminarayanan. Likelihood ratios for out-of-distribution detection. *Advances in neural information processing systems*, 32, 2019.

[5] Shiyu Liang, Yixuan Li, and R Srikant. Enhancing the reliability of out-of-distribution image detection in neural networks. In *International Conference on Learning Representations*, 2018.

[6] Yiyou Sun, Chuan Guo, and Yixuan Li. React: Out-of-distribution detection with rectified activations. *Advances in Neural Information Processing Systems*, 34:144–157, 2021.

[7] Yiyou Sun, Yifei Ming, Xiaojin Zhu, and Yixuan Li. Out-of-distribution detection with deep nearest neighbors. In *International Conference on Machine Learning*, pages 20827–20840. PMLR, 2022.

[8] Jingkang Yang, Pengyun Wang, Dejian Zou, Zitang Zhou, Kunyuan Ding, Wenxuan Peng, Haoqi Wang, Guangyao Chen, Bo Li, Yiyou Sun, et al. Openood: Benchmarking generalized out-of-distribution detection. *Advances in Neural Information Processing Systems*, 35:32598–32611, 2022.

[9] Kimin Lee, Kibok Lee, Honglak Lee, and Jinwoo Shin. A simple unified framework for detecting out-of-distribution samples and adversarial attacks. *Advances in Neural Information Processing Systems*, 31, 2018.

[10] Chandramouli Shama Sastry and Sageev Oore. Detecting out-of-distribution examples with gram matrices. In *International Conference on Machine Learning*, pages 8491–8501. PMLR, 2020.

Confiance ai

Title: OODeel methodological guidelines

Keywords: Out-of-Distribution Detection, Image classification

The document introduces the OODeel library, presenting a detailed overview of its Out-of-Distribution (OOD) detection pipeline. Covering data loading, classifier training, OOD detector fitting, and result evaluation, the library offers a versatile framework accommodating both PyTorch and TensorFlow. Users can seamlessly transition between frameworks, access diverse OOD detection methods, and leverage user-friendly interfaces for model training. The guidelines provided address method-specific hyperparameter tuning, ensuring optimal performance. Emphasizing benchmark studies and the variability of results across datasets and models, the document aids users in method selection and offers valuable insights into the rapidly evolving landscape of OOD detection.

Our partners:

Air Liquide AIRBUS Atos cea Inria
NAVAL GROUP GROUPE RENAULT SAINT EXUPÉRY SAFRAN sopra steria
SystemX THALES Valeo