POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Anomaly Detection via learned models and preference trick

Tesi di Laurea Magistrale in
Computer Science Engineering
Ingegneria Informatica

Author: **Danilo Catone**

Student ID: 951402
Advisor: Prof. Luca Magri
Co-advisors: Giacomo Boracchi, Filippo Leveni
Academic Year: 2021-22

Abstract

Anomaly Detection is a field of data mining applied in numerous contexts, such as quality inspection, fraud detection, and medicine, that seeks to detect the anomalies in a dataset, defined as those instances that do not have a well-defined behavior or that deviate from usual behavior.

In this thesis we focus on anomalies in datasets containing structured normal points, i.e., those in which normal points follow well-defined pattern(s), while points not following the pattern(s) are considered anomalous.

In this context, we want to extend PIF, a model-based anomaly detection algorithm that embeds the points in the dataset in the preference space and applies iVor algorithm on it, in order to compute an anomaly score for each instance. The embedding in the preference space is done through a pool of models sampled from data, containing patterns of which we assume to know the formulation.

We will show that is possible to build an ensemble of more general models that extract patterns autonomously, directly from data, without the knowledge of the patterns to search for. Therefore, we will demonstrate (i) that is possible to locally approximate normal data with a model, and (ii) that the models can learn the patterns without any information on the type of pattern. This is possible thanks to two types of Neural Networks, auto-encoders and self-organizing maps, that learn the pattern to search instead of fixing it as in PIF.

We tested our algorithm on publicly available synthetic datasets, comparing it with two state-of-the-art density-based methods such as iFor and LOF, showing that our algorithm is superior in terms of ROC AUC on all datasets; we also explored different models and architectures for each model, looking for the right hyper-parameters.

Keywords: anomaly detection, neural networks, preference embedding, self organizing maps, auto encoders

Abstract in lingua italiana

L'Anomaly Detection è un campo del data mining applicato in numerosi contesti, come l'ispezione della qualità, il rilevamento delle frodi e in medicina, che cerca di individuare le anomalie presenti in un set di dati, definite come quelle istanze che non hanno un comportamento ben definito o che si discostano dal comportamento abituale.

In questa tesi ci concentriamo sulle anomalie nei set di dati contenenti punti normali strutturati, cioè quelli in cui i punti normali seguono schemi (pattern) ben definiti, mentre i punti che non seguono gli schemi sono considerati anomali.

In questo contesto, vogliamo estendere PIF, un algoritmo di Anomaly Detection che incorpora i punti del set di dati nello spazio delle preferenze e applica Voronoi Isolation Forest su di essi, al fine di calcolare un punteggio di anomalia per ogni istanza. L'incorporazione nello spazio delle preferenze avviene attraverso un insieme di modelli campionati dai dati, contenenti pattern di cui si presume di conoscere la formulazione.

Dimostreremo che è possibile costruire un ensemble di modelli più generali che estraggono i pattern autonomamente, direttamente dai dati, senza la conoscenza dei pattern da ricercare. Pertanto, dimostreremo (i) che è possibile approssimare localmente i dati normali con un modello, e (ii) che i modelli possono apprendere i pattern senza alcuna informazioni sul tipo di modello. Questo è possibile grazie a due tipi di reti neurali, gli auto-encoders e le self-organizing maps, che apprendono il modello da ricercare invece di fissarlo come nel PIF.

Abbiamo testato il nostro algoritmo su dataset sintetici disponibili pubblicamente, confrontandolo con due metodi dello stato dell'arte basati sulla densità come iFor e LOF, dimostrando che il nostro algoritmo è superiore in termini di ROC AUC su tutti i dataset; abbiamo inoltre esplorato diversi modelli e architetture per ogni modello, alla ricerca degli hyper-parametri giusti.

Parole chiave: rilevamento di anomalie, reti neurali, preference embedding, self-organizing maps, auto-encoder

Acknowledgements

List of Symbols

Variable	Description
$\mathcal{D}$	generic dataset
$\mathcal{M}$	generic manifold
$\mathcal{N}$	set of normal points
$\mathcal{A}$	set of anomalies
$\mathbf{p}$	a generic point of the dataset
$\mathcal{MSS}$	the Minimum Sample Set
ρ	size of Minimum Sample Set
τ	inlier threshold
AE	generic auto-encoder
SOM	generic self organizing map

Contents

1 | Introduction

From financial fraud detection to healthcare insurance, anomaly detection is one of the most studied problem in data analysis and inspired the research towards the design of statistical or machine learning algorithms.
The objective of this task is to identify **anomalies**, defined as a dataset instance that deviates from expected behavior, or is otherwise distinct from other instances.
In the literature, *normal* points are data instances that lie in denser regions or that follow a certain structure, while *anomalies* fall in low-density regions or that do not follow any structure.

Applications for anomaly detection algorithms are fraud detection, cyber-security intrusions, insurance forgery and other fraudulent or hazardous activities, but they are also employed in other contexts such as finding anomalous pixels inside an image or, in quality inspection, understand if a component is faulty or not.
The methods to solve this problems has been used in a range of industries in order to advance IT safety and detect potential abuse or assaults, since anomalies in information systems frequently indicate some security breaches or violations. Thus, anomaly detection methods can also guarantee a greater level of security.

In this thesis we will mainly focus on structured datasets, in which anomalies are generated from a different structure, or pattern, with respect to the one that generated normal points; in this case, the density of the points can be unimportant to identify anomalies because it does not say anything about the pattern. There are situations in which the instances of the dataset with a normal behaviour follow a precise pattern, e.g. can be plotted on a line or on a circle or on more complex figures, while anomalies are outside this structures. In other words, the normal behavior is guided by an unknown pattern, or manifold, and anomalies lies outside this manifold and do not conform to any specific behavior. As an example, in figure 1.1 is shown a dataset with a sinusoidal pattern.

In this context, Anomaly Detection can be seen as a pattern-recognition problem, solved

tb

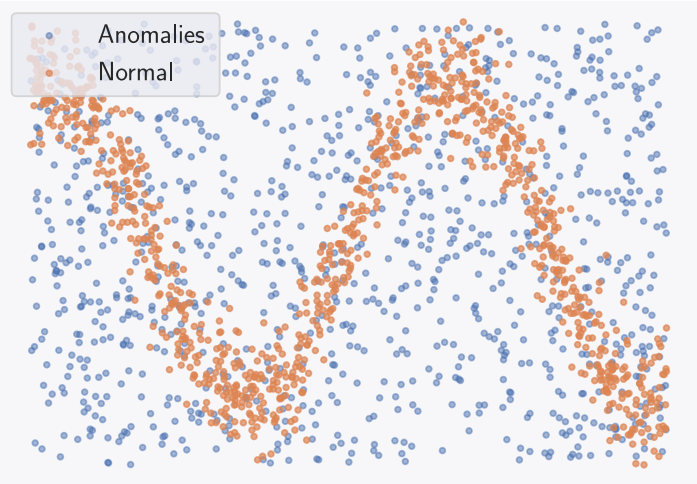


Figure 1.1: Example of structured dataset $\mathcal{D} \in \mathbb{R}^2$. **Normal** points are shown in **orange**, while **anomalies** are shown in **blue**. The pattern in the dataset is a sinusoidal function $f(x) = y = sin(x)cos(x)$. Anomalies are random points that do not follow any pattern, laying in the same space of the normal points.

using parametric techniques, in which we need to identify the patterns that generated the data in order to understand if a point follows one of them or it is an anomaly not following any pattern.
The challenging element is that the manifolds are unknown, and we need to identify the anomalies in the dataset in order to create a model that replicates the structures. But we need the original pattern in order to identify the anomalies. In this context, we have developed a **novel** algorithm able at identifying the patterns in the data using **robust model regression** and the **preference trick**.

1.1. Use cases

In order to give a context to this task, we show here some use cases in which Anomaly Detection is the key to avoid bad consequences.

- **Cyber-intrusion**: Cyber-security is usually guaranteed with the help of network behavior anomaly detection (NBAD) technology [1]. The system analyzes packet signatures to detect security threats and block incoming/outgoing data that is compromised. NBAD also conducts continuous network monitoring to detect suspicious events or trends
- **Fraud**: Graph-based anomaly detection (GBAD) is used to prevent fraud with

credit cards, bank accounts, and insurance [24]. Machine Learning systems also enable online banking fraud with the help of behavioral biometrics that also detects anomalies in consumer spending in real-time [20].

- **Medical anomaly detection**: Outlier identification has been applied in clinical settings in a variety of ways. For instance, the density-based clustering method can be applied to patient careflow log analysis [32] to see whether the particular patient's careflow trace is anomalous. Anomaly detection in medical image analysis is helpful in accurate diagnostics, while treatment plan analysis may help determine potentially fatal errors in the treatment plans [29].
- **Industrial damage**: In the conditions of industrial automation, anomaly detection systems use data coming from numerous sensors to identify any malfunctions in the machinery [25] [30], thus able to detect abnormalities early to prevent further damage or manufacturing defects.
- **Image processing**: The ability of anomaly detection systems to compare and analyze images allows accurate fraud detection in banking and insurance (when one recipient of a service submits duplicate reimbursement claims or when fraudsters try to receive reimbursement on fake claims)
- **Stock trading**: Anomaly detection algorithms deal quite well with the big masses of unstructured data in the stock exchanges, be it regular stocks or cryptocurrencies. ML systems classify the available data about price movements and sales volumes to detect anomalies and give alerts to the users about price outliers. This information may be instrumental in trading decision-making.

1.2. Thesis Structure

The rest of the thesis will be organized as follows: in Chapter 2 we present some background concepts that can help the reader to understand the basic concepts used in our solution. In Chapter 3 we show a brief taxonomy of Anomaly Detection algorithms, explaining the idea behind the families of algorithms and explaining in-depth the algorithms used in our comparisons.
In Chapter 4 we explain our solution to the problem, giving a formal definition of it and describing how and why we developed this method. In Chapter 5 we will show the results obtained by our algorithm, comparing it to the State-of-Art algorithms on synthetic datasets. Finally, in Chapter 6 we make our final thought about the project and explain how it can be improved and which are the next steps.

2 | Background

In this chapter we will present some of the basic concepts used in the Thesis.

2.1. Line

A straight line in a *two-dimensional* plane has an implicit equation, linear in two variables, of the form $ax+by+c=0$ and each couple of points identifies one and only one line. Thus, given two generic points $p=(x_p, y_p)$ and $q=(x_q, y_q)$, we can estimate the parameters of the straight line with the following equations:

$$a = x_q - y_q \tag{2.1}$$

$$b = y_p - y_q \tag{2.2}$$

$$c = x_p y_q - x_q y_p \tag{2.3}$$

Thanks to this equations it is possible to estimate a simple analytic model that tries to encapsulate the relationship between the two points; this linear model will be used as a baseline of more complex models, like Neural Networks. The geometric distance of a generic point p from the line is defined as follows:

$$d = \frac{ax_p + by_p + c}{\sqrt{a^2 + b^2}} \tag{2.4}$$

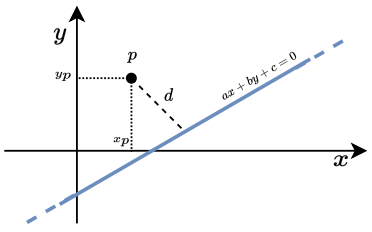


Figure 2.1: Example of a 2D line.

Points lying on the line will have distance zero, while for other points this distance is the magnitude of the normal vector passing through the point and the line. This value will be positive if the point lies above the line, while it will be negative if it is below the line; for this reason, we can take the absolute value to have only the magnitude of the normal vector.

2.2. Plane

As we have done for the *two-dimensional* line, we can give a formal definition of a Plane. A plane is a *three-dimensional* objects that is formed by three non-collinear[1] points; also this object have an implicit equation of the form $ax + by + cz + d = 0$.
Given three generic points in the 3D space $p = (x_p, y_p, z_p)$, $q = (x_q, y_q, z_q)$ and $r = (x_r, y_r, z_r)$, we can estimate the parameters of the plane.
First of all, we define two vectors $\mathbf{v}_1$ and $\mathbf{v}_2$ as the difference between two points and the third one:

$$\begin{aligned} \mathbf{v}_1 &= q - p = (v_{1_x}, v_{1_y}, v_{1_z}) = (x_q - x_p, y_q - y_p, z_q - z_p) \\ \mathbf{v}_2 &= r - p = (v_{2_x}, v_{2_y}, v_{2_z}) = (x_r - x_p, y_r - y_p, z_r - z_p) \end{aligned} \tag{2.5}$$

The parameters a, b, c are the components of the normal vector to $\mathbf{v}_1$ and $\mathbf{v}_2$:

$$\begin{aligned} (a, b, c) = \mathbf{v}_1 \times \mathbf{v}_2 = det\left(\begin{bmatrix} \hat{i} & \hat{j} & \hat{k} \\ v_{1_x} & v_{1_y} & v_{1_z} \\ v_{2_x} & v_{2_y} & v_{2_z} \end{bmatrix}\right) = \\ = (v_{1_y} v_{2_z} - v_{1_z} v_{2_y})\hat{i} - (v_{1_x} v_{2_z} - v_{1_z} v_{2_x})\hat{j} + (v_{1_x} v_{2_y} - v_{1_y} v_{2_x})\hat{k} \end{aligned} \tag{2.6}$$

From the above solution, we get:

$$\begin{aligned} a &= (y_q - y_p)(z_r - z_p) - (z_q - z_p)(y_r - y_p) \\ b &= (x_q - x_p)(z_r - z_p) - (z_q - z_p)(x_r - x_p) \\ c &= (x_q - x_p)(y_r - y_p) - (y_q - y_p)(x_r - x_p) \\ d &= -ax - by - c \end{aligned} \tag{2.7}$$

Thanks to this equations, it is possible to estimate from data a model encapsulating the plane on which the points lie. Again, we can compute a geometric distance of a generic

[1] Points that do not lie on the same line

point p from the plane as we have done for the line:

$$d = \frac{|ax_p + by_p + cz_p + d|}{\sqrt{a^2 + b^2 + c^2}} \tag{2.8}$$

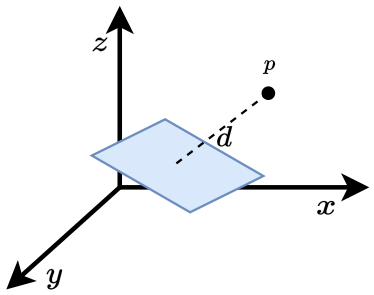


Figure 2.2: Example of a 3D plane.

This distance measures the magnitude of the normal vector passing through the point and the plane. It can be seen as a measure of how much the point deviates from the plane.

2.3. Neural Networks

After showing the two simplest models, the Line and the Plane, we will analyze more complex models that can be estimated from data.
Artificial Neural Networks (ANN) are one of the most discussed Machine Learning model in the last years, thanks to their ability to handle non-linear transformations of data with very high performances. ANNs perform representation learning, in which each layer of the network learns a representation from the previous layer. By building more robust and detailed representations from a layer to the other, ANNs can accomplish tasks such as speech recognition, computer vision and machine translation [23].

Conceptually, it resembles the structure of human's brain, in which there are two main components: neurons and synapses, that are connections between neurons. In ANNs the neurons are the basic building block and are placed in layers. The typical structure is made of one *input layer*, one (or more) *hidden layer* and finally an *output layer*; the number of hidden layers defines how *deep* the neural network is. Each hidden layer can be thinked as an intermediate computation that allows the network to perform complex function approximation.
In addition, each layer has a certain number of nodes that build up the layer; the nodes of each layer are connected to the nodes of the following layer.

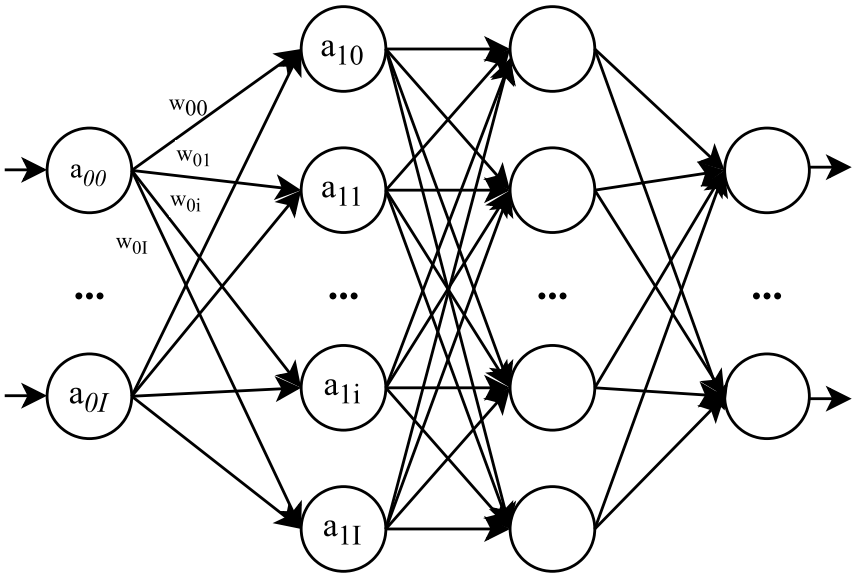


Figure 2.3: Example of a neural network. The various a_{ij} represent the *j-th* neuron of *i-th* layer. Weights w_{kj} represent the weight of the connection from neuron k of the preceding layer to neuron j of the current layer.

In order to build an ANN to solve our specific problem, we need to define several *hyper-parameters*, that will define the structure of the network and how it works. We can play with the number of hidden layers, the number of nodes in each layer and the activation functions to use. The activation function determines what value of the current layer is fed into the next layer of the network. It can be linear, e.g. the identity function $id(\mathbf{x}) = \mathbf{x}$, but the most interesting ones are non-linear, since can project the output of each neuron into a non-linear space, helping in the majority of the cases.
Some of the most used non-linear activation functions are

$$sigmoid(x) = \frac{1}{1 + e^{-x}} \tag{2.9}$$

$$tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \tag{2.10}$$

$$relu(x) = \begin{cases} x & x > 0 \\ 0 & otherwise \end{cases} \tag{2.11}$$

Given a neuron j, its output is computed as

$$h_j(\mathbf{x} \mid \mathbf{w}, b) = h_j\left(\sum_{i=1}^{I} w_{ij} x_i + b_j\right) = h_j(\mathcal{W}_j^T \mathbf{x}) \tag{2.12}$$

where I is the number of neurons of the preceding layer, $\mathcal{W}_j = (w_{1j}, ..., w_{ij}, ..., w_{Ij})$ are the weights of the connections between the neurons of the preceding layer and neuron j, $\boldsymbol{x}$ is the input vector, given as the outputs of the neurons of the preceding layer, $b_j = w_0$ is the bias of the current neuron and h_j is the activation function of the neuron.

Depending on the problem, it is possible to adapt the output layer for regression, in which the output(s) span in $\mathbb{R}^m$, or classification, in which the output is an array $\Omega \in \mathbb{R}^k$, where k is the number of classes.

There are also different types of learning techniques that are possible with Neural Networks (and in Machine Learning in general), e.g. supervised learning and unsupervised learning.
In the former case, the input layer represents the features that are fed into the neural network, and the output layer represents the label assigned to each observation. During the training process, the neural network determines which weights across the neural network help minimize the error between its predicted label for each observation and the true label.
In the latter case, instead, the neural network learns representations of the input layer via the various hidden layers, but is not guided by labels.

There are several loss functions that can be defined depending on the task that we want to solve, for example usually for regression is used the Mean Squared Error (MSE) while for classification is used the Cross Entropy. The MSE is defined as

$$E(\mathbf{w}) = \sum_{n=1}^{N} (\mathbf{t}_n - g(\mathbf{x}_n,\ \mathbf{w}))^2 \tag{2.13}$$

where N is the number of input data, $\mathbf{t}_n$ is the label of input $\mathbf{x}_n$, while $g(\mathbf{x}_n,\ \mathbf{w})$ is the output for input $\mathbf{x}_n$.
We want to find the weights matrix $\mathcal{W}$ that minimizes the error, so we apply *Stochastic Gradient Descent* (SGD) and update the weight of each connection as follows:

$$\mathbf{w}^{k+1} = \mathbf{w}^k - \eta \frac{\partial J(\mathbf{w})}{\partial \mathbf{w}}\bigg|_{\mathbf{w}^k} \tag{2.14}$$

Although there are multiple types of neural networks, such as *recurrent neural networks* in which data can flow in any direction and *convolutional neural networks*, we will focus on the more straightforward feed-forward neural network, in which data moves just forward.

In this networks, the training procedure is divided in two phases: the *forward* pass and the *backward* pass. The former computes the output from the input, while the latter computes the error of the output and the gradient with respect the weights of each layer, updating them.

2.3.1. Auto-Encoders

Auto-Encoders (AE) are a particular kind of ANN that is comprised by two parts: an *encoder* and a *decoder*. The encoder converts the input set of features into a different representation via *representation learning*, while the decoder converts this newly learned representation to the original format.

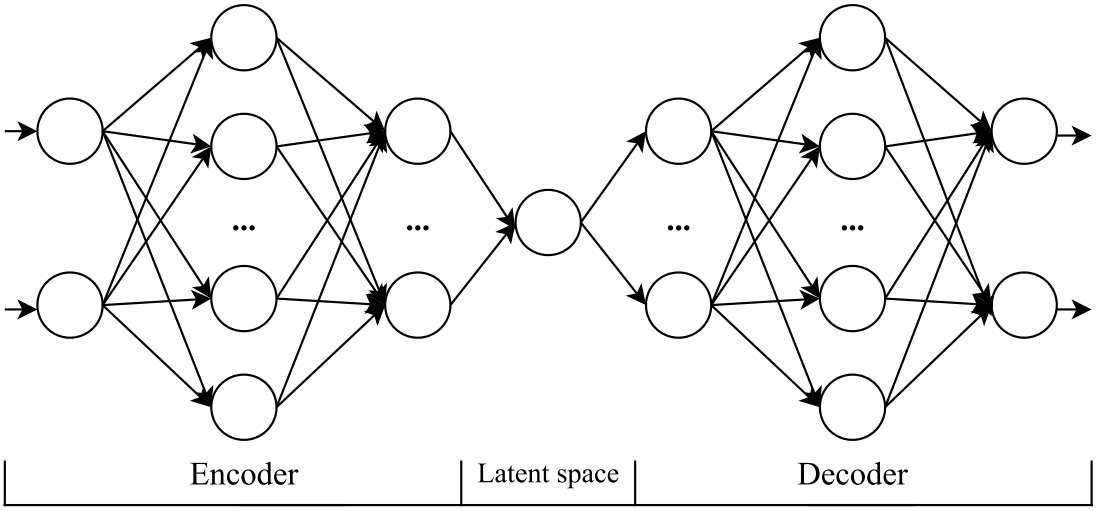


Figure 2.4: Example of an architecture for an auto-encoder. The input (and output) is *2-dimensional*, thus the latent space must be *1-dimensional*.

An Auto-Encoder does not learn the *identity function*, which would simply be an identical copy of the original input, but they must approximate the original observations as closely as possible - but not exactly - using a newly learned representation. In other words, they learn an *approximation* of the identity function.
Since the auto-encoder is constrained, it is forced to learn the most salient properties of the original data, capturing the underlying structure of the data. The constraint is a very important attribute of auto-encoders, because it forces the auto-encoder to intelligently choose which important information to capture and which irrelevant information to discard.

As already said, the encoder part is the one that learns a new representation of the original data; we will refer to the encoder function as $h = f(\mathbf{x})$, in which $\mathbf{x}$ is the input

vector and $f(\cdot)$ is the newly learned representation. The decoder function that reconstructs the original observations using the output of the encoder is $r = g(h)$. If done correctly, $r = g(f(\mathbf{x}))$ will not be exactly equal to $\mathbf{x}$ everywhere, but will be close enough.

In order to restrict the encoder function to only *approximate* the input vector $\mathbf{x}$, we can constrain the encoder function to have fewer dimensions than $\mathbf{x}$. This is known as an *under-complete auto-encoder*, since the encoder's dimensions are fewer than the original input dimensions. Thanks to this constraint, the auto-encoder tries to minimize a *loss function* we define, e.g. MSE (2.13), such that the reconstruction error is as small as possible. The reconstruction error is defined as the error calculated between the original input and the output of the decoder's function: $Loss(\mathbf{x}, g(h(\mathbf{x}))$.

When the decoder is linear and the loss function is the mean squared error, an under-complete auto-encoder learns the same type of representation as **PCA** [21], an algorithm for dimensionality reduction that projects data on the highest variance planes. However, if the encoder and decoder functions are nonlinear, the auto-encoder can learn much more complex non-linear representations.

2.3.2. Self Organizing Maps

Another special kind of Neural Network used for unsupervised learning are Self organizing maps [15] (SOM), able to produce a *low-dimensional* representation of a higher dimensional data set, preserving the topological structure of the data.
This network is made up of two layers: one *input* layer and one *output* layer. The output layer is made of a set of weights $\mathcal{W} = \{ \mathbf{w} \mid \mathbf{w} \in \mathbb{R}^m \}$, that are vectors in the same *m-dimensional* space as the input $\mathbf{x} \in \mathbb{R}^m$. The set of weights can have different shapes, for example can be *one-dimensional* and act like a line, or can be *two-dimensional* with a grid or hexagonal displacement.

The weights of this network lies in the same space as the input, so each weight represents a point in the space of the input vector, that is $\mathbb{R}^m$. At each iteration, the weights are update to be nearer to the input data; after the training, the weights of the SOM will be placed near the most concentrated areas of the input data while preserving the topological structure of the weights. Indeed, weights that are neighbor in the network are also near in the *m-dimensional* space.

This type of network is trained via *competitive learning*, different from the "standard" *error-correction learning*, meaning that for each input vector in the dataset the weights

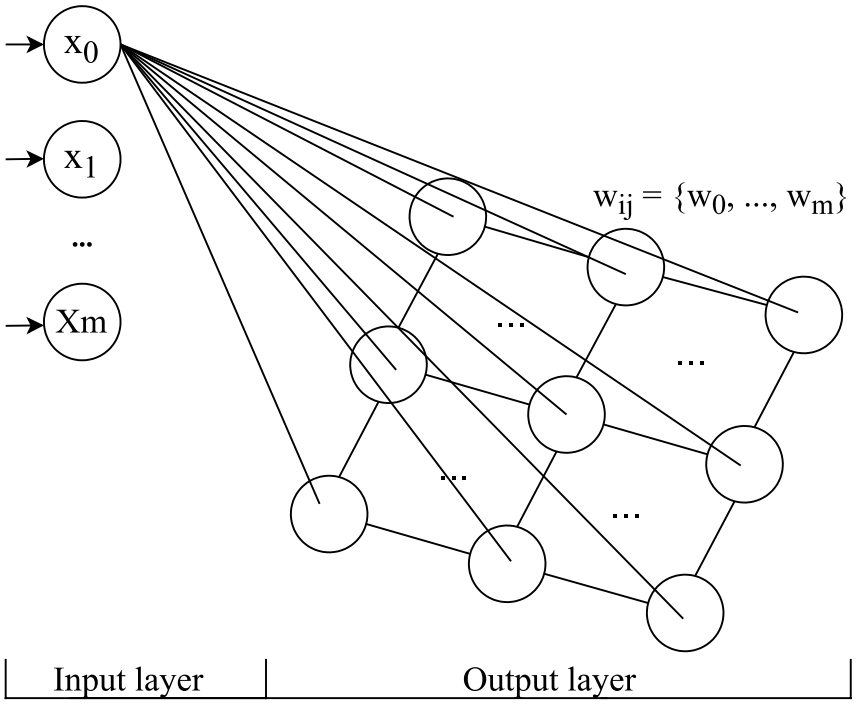


Figure 2.5: Ideal structure of SOM. Each element of the input vector, x_i, is connected to each neuron of the output layer, displaced as a grid.

of the network compete to win; the winning vector is the nearest weight to the current input vector.
When a winning weight is chosen, this vector and its neighbors in the structure of the network will be updated to be nearer to the current input vector. In this way, the weights will concentrate where there are more input points, while preserving the topology of the network.

The training algorithm works as follows: after an initialization of the weights, that can be at random, with a grid-pattern over the input space or using the two largest principal component eigenvectors, for each input data $\mathbf{p}$ of a dataset $\mathcal{D} = \{ \mathbf{p} \mid \mathbf{p} \in \mathbb{R}^m \}$ we find the weight of the network that minimizes the distance from the input data and we call it **BMU**, i.e. *Best Matching Unit*. The distance usually is Euclidean, so

$$distance(\mathbf{p}, \mathbf{w}) = \sqrt{\sum_{i=0}^{m} \mathbf{p}_i^2 - \mathbf{w}_i^2}$$

After having found the nearest weight, we can update the weights with the following

formula:

$$\mathbf{w}_k = \mathbf{w}_k + \eta(t) \cdot h_{ik}(t) \cdot (\mathbf{p}^n - \mathbf{w}_k) \tag{2.15}$$

where k is the index of the current weight, i is the index of the BMU, η is the learning rate that scales with time, i.e. as the epochs pass the learning rate gets smaller, computed as

$$\eta(t) = \eta_0 \cdot e^{\left(-\frac{t}{T}\right)}$$

The function $h_{ik}(t)$, instead, is the *neighboring function* that takes into account the distance in the **topology** of the current vector $\mathbf{w}_k$ from the BMU $\mathbf{w}_i$. This function can be any function that is inversely proportional with this distance in the lattice and that decreases while time increases. One example is

$$h_{ik}(t) = e^{\left(-\frac{d_{ik}^2}{2\sigma^2(t)}\right)} \tag{2.16}$$

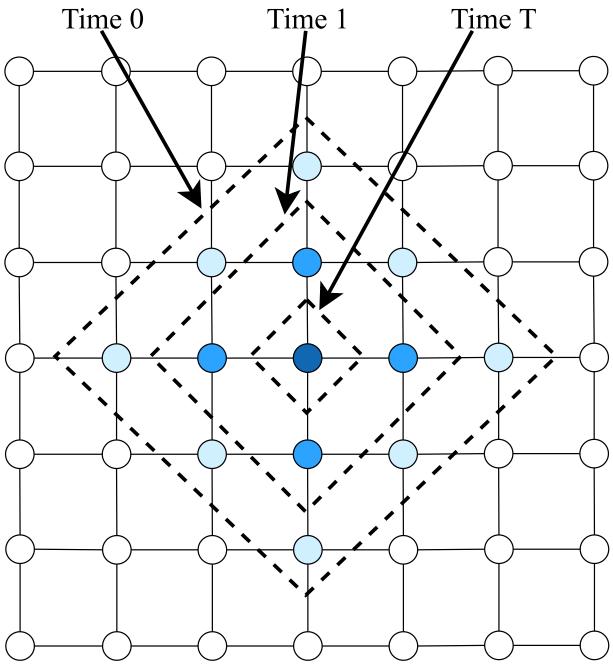


Figure 2.6: Image that shows the effect of the neighboring function. At time 0 several neighbors are considered, while as the time passes ever less neighbors are updated. The color represents the update's intensity; the higher the intensity, the closer the weight is to the BMU.

where d_{ik} is the distance in the lattice and $\sigma(t)$ is a neighborhood size decay rule, defined

as

$$\sigma(t) = \sigma_0 \cdot e^{\left(-\frac{t}{T}\right)}$$

Summarizing, the full train loop is the following:

Algorithm 2.1 SOM Training loop

Input: $\mathcal{D} = \{\ \mathbf{p} \mid \mathbf{p} \in \mathbb{R}^m\ \}$, η_0, σ_0, E ▷ E is the number of epochs

$\mathcal{W} \leftarrow$ Initialize weights
$T \leftarrow |\mathcal{D}| * E$
$t \leftarrow 0$
while $e < E$ **do**
  $\mathcal{D} \leftarrow$ Shuffle $\mathcal{D}$
  for all $\mathbf{p} \in \mathcal{D}$ **do**
    $i \leftarrow \underset{k}{\text{argmin}}\ \{distance(\mathbf{p}, \mathbf{w}_k)\}$
    for all $\mathbf{w}_k \in \mathcal{W}$ **do**
      $\mathbf{w}_k \leftarrow \mathbf{w}_k + \eta(t) \cdot h_{ik}(t) \cdot (\mathbf{p}^n - \mathbf{w}_k)$
    end for
    $\eta(t) \leftarrow \eta_0 \cdot e^{(-\frac{t}{T})}$
    $\sigma(t) \leftarrow \sigma_0 \cdot e^{(-\frac{t}{T})}$
    $t \leftarrow t + 1$
  end for
end while

As can be noticed from the algorithm, the weights of the network try to lay down to input data, updating the vectors depending on time and on the distance of the current vector from the winning one. In this way, at the beginning the network moves a lot all the weights, but eventually only the winning vectors will be updated.

2.4. Voronoi Tessellation

Voronoi tessellation is a partition of a metric space into b regions defined by b samples, called *seeds*: $\mathcal{S} = \{s_i\}_{i=1,\dots,b}$. The *i-th* region produced by the tessellation contains all the points $\mathbf{p}$ of the space having s_i as the closest seed in $\mathcal{S}$.

2.5. Manifold

A Manifold [27] is a topological space (a geometrical space in which closeness is defined) that locally resembles Euclidean Space near each point. As an example to clarify this

concept, consider the ancient idea that the Earth was flat; this idea arises from the fact that on the small humans' scale, the Earth looks flat.
Essentially, the Earth is a sphere (a 2D manifold) that locally can be thought as a plane. *One-dimensional* manifolds include lines and circles, while *two-dimensional* manifolds include planes and spheres. More concisely, any object that can be "charted" is a manifold.

2.6. Classification Metrics

Our goal is to identify anomalies with respect to normal points. Therefore, we are doing a *two-class* classification, labelling as the Positive class the normal points and as the Negative class the anomalous ones.
In this context, exist a lot of different metrics that we can use to compare our algorithms. All of this metrics rely on the *Confusion Matrix*, that puts in relationship four quantities:

- the **True Positives** (TP) points, that are the inlier points actually identified as normal;
- the **False Negatives** (FN) points, that are the inlier points labelled as anomalies;
- the **False Positives** (FP) points, that are the anomalous points identified as normal;
- the **True Negatives** (TN) points, the ones that are anomalous and are labelled as anomalies.

This four quantities can be synthetized in the following matrix:

		prediction outcome		
		p	**n**	**total**
actual value	**p′**	TP	FN	P′
	n′	FP	TN	N′
	total	P	N	

And on this matrix we can compute several different metrics:

- **Accuracy:** Represents the fraction of correctly labelled instances over the total

number of instances.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{2.17}$$

- **Precision:** Represents the fraction of items labelled as positive that are actually positive. The higher the Precision, the lower the FPs.

$$Precision = \frac{TP}{TP + FP} \tag{2.18}$$

- **Recall:** Represents the fraction of positive items labelled as positive. The higher the Recall, the lower the FNs.

$$Recall = \frac{TP}{TP + FN} \tag{2.19}$$

- **F1-score:** combination of Precision and Recall to better ignore the biases of the precision and recall: the higher the F1-score, the lower FPs and FNs.

$$F1score = \frac{2 \cdot P \cdot R}{P + R} \tag{2.20}$$

- **True Positive Rate (TPR):** Represents the ability to correctly identify the elements of the positive class:

$$TPR = \frac{TP}{TP + FN} \tag{2.21}$$

- **True Negative Rate (TNR):** Represents the probability to correctly identify the elements of the negative class:

$$TNR = \frac{TN}{TN + FP} \tag{2.22}$$

Usually classifiers models outputs a score, a sort of probability to belong to one of the two class; this is also our case. Thus, there is the need of introducing a "cutting-off" threshold, which says at which value samples are classified as positive. For example, the basic threshold is set at 0.5, thus all samples with a $score < 0.5$ are labelled as positive and all samples with a score $score >= 0.5$ are labelled as negative.
Depending on the problem and on the requirements of the model, there can be the necessity to modify the threshold. Indeed, using a near one threshold we have a pessimistic classifier, with high Precision (few FPs) and low Recall (many FNs); using a near zero

threshold, instead, we have an optimistic classifier since the Precision will be low (high false positive, everything is positive) and the Recall will be high (few false negatives).

2.6.1. Receiver Operating Characteristic curve

The ROC curve is another curve that we can generate to analyze how model's performances change as the threshold changes. It plots the TPR (equation 2.21) against the TNR (equation 2.22) and each point of the curve identifies a different classifier, in our case different "versions" of the same model.
The ideal classifier would have $TPR = 1$ and $TNR = 0$, so would be located at the point $(0, 1)$; if the classifier always outputs the **positive class**, it would be located at $(1, 1)$ while if it always predicts the **negative class**, it is located at $(0, 0)$. We can also compare the algorithm with the random-guessing model[2], since its performances are on the diagonal line.
The random-guessing model will have a ROC AUC of 0.5, thus every expert model are likely to have a greater AUC; if it is smaller, predictions must be flipped, e.g. if an instance is labeled as positive it should be changed as negative and the other way round.

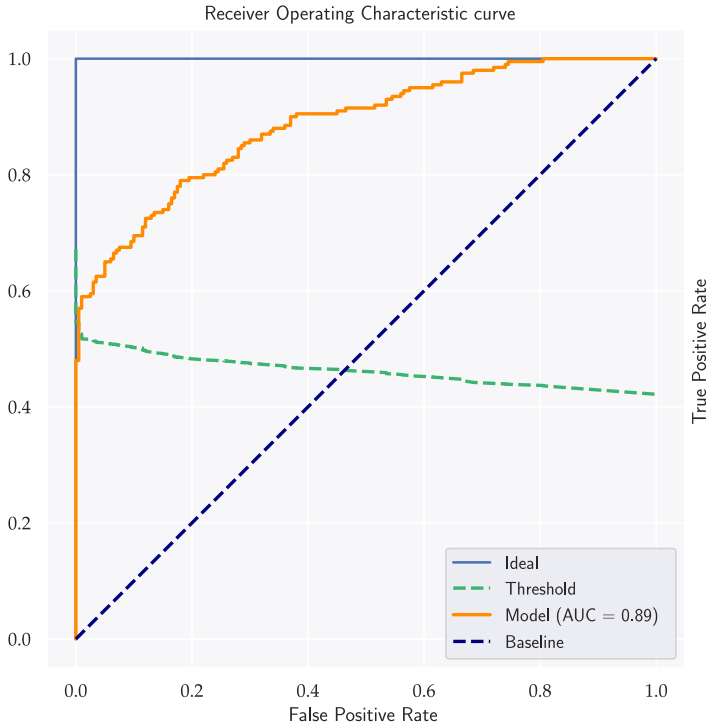


Figure 2.7: An example of a Receiver Operating Characteristic curve, showing both the ideal curve and the curve from a generic model, with good results.

[2]A very simple model that outputs a random score. It is used as a baseline for classifier models.

3 | State of art

Anomaly Detection refers to the problem of finding patterns in data that do not conform to expected behavior. This is a well-known problem in the literature and a lot of work has been done in this field since the end of the eighties.

Generally speaking, there are three broad categories of anomaly detection algorithms, all based on the type of learning. Indeed, we can find anomaly detection exploiting *supervised learning*, *semi-supervised learning* and *unsupervised learning*. In particular,

- **Supervised learning** means that all the instances of the dataset have a label indicating if the instance is anomalous or not. Algorithms of this kind are trained as two (or one)-class classifiers, such as *One-Class SVM* [28].
- **Semi-supervised learning** means that only a portion of data is labelled, that can be a subset of both inliers and outliers or only of one of them. It can also mean that we only have samples of the inliers and we train an algorithm on those; then, we can use the same algorithm to discriminize between inliers and outliers. One example of the latter technique are Auto-Encoders for computer vision, in which we train an Auto-Encoder to replicate the normal images and then we measure how much an instance is probably an outlier depending on the reconstruction error.
- **Unsupervised learning** means that we do not have labels at all, but only data instances. This is the most commonly used due to their wider and relevant application.

Another possible taxonomy of anomaly detection algorithms envisages three main categories: *distance-based*, *density-based* and *model-based*. The first two categories are quite similar, because they define a measure of outlierness based on the number of neighboring points for each point, considered in the space of input vectors. The three categories can be summarized as follows:

- **Distance-based** methods compute, for each point $\mathbf{p} \in \mathcal{D}$ in the dataset, the number of points inside the neighboring of radius ε around this point; the set of neighbors

is composed as $Neigh(\mathbf{p}) = \{ \mathbf{q} \in \mathcal{D} \mid distance(\mathbf{p}, \mathbf{q}) < \varepsilon \}$. If this number is below a certain threshold, the point is considered as anomalous.
An example of algorithm belonging to this family is *K-Nearest Neighbors* [12].

- **Density-based** methods instead compute the density for each point $\mathbf{p}$ as the fraction of points in the k-neighborhood of $\mathbf{p}$ with respect the density around other points. In particular, a point is considered **inlier** if the density around the point is similar to the density around its *k-neighbors*, while the point is considered as an **outlier** if the density around the point is significantly different from the local density of its *k-neighbors*.
  An example of algorithm belonging to this family is *LOF* (3.1) [9].

- **Reconstruction-based** methods usually employ Neural Networks, training them to reconstruct the normal instances minimizing the reconstruction error [10]. The anomaly score is then computed as the reconstruction error: the higher it is, the higher the probability of being an anomaly.

- **Model-based** methods assume that inlier data points are generated from a model, thus the anomaly score is computed as the degree of deviation of the point from the model. The more an instance deviates from the model, the higher its probability to be an outlier. This methods try to fit some models on the data and then measure the deviation of the points from this fitted models.
  Examples of this approach are classification based methods [7] and clustering based methods [13]. Other examples of algorithms belonging to this family are Isolation Forest (iFor) [17], that builds an ensemble of Random Trees to isolate outliers, and Preference Isolation Forest (PIF) [16], from which we take inspiration.

In the following we will analyze more in depth the algorithms that we will use for our comparisons in the results part, i.e. LOF, Isolation Forest and Preference Isolation Forest, that extends Isolation Forest.
We choose those algorithms because we want to show that in the context of structured datasets, the density as used in LOF or iFor is not informative because the patterns can have different densities or anomalies can be found in high density regions even if they are not following any pattern. PIF has been chosen because it is our starting point and because we want to show the goal of our Thesis, that is building general models without any assumption on the nature of the patterns.

3.1. Local Outliers Factor

As already said, this algorithm belongs to the density-based family. In order to define what is the local density of a point, we need to introduce some concepts.
First of all, we can introduce the notion of **k-distance** of a point $\mathbf{p} \in \mathcal{D}$ in a dataset as the distance $distance(\mathbf{p}, \mathbf{q})$ between $\mathbf{p}$ and $\mathbf{q} \in \mathcal{D}$ such that, for any $k \in \mathbb{N}$:

(i) for at least k objects $\mathbf{q}' \in \mathcal{D}\backslash\{\mathbf{p}\}$ it holds that $distance(\mathbf{p}, \mathbf{q}') \leq distance(\mathbf{p}, \mathbf{q})$, and

(ii) for at most $k-1$ objects $\mathbf{q}' \in \mathcal{D}\backslash\{\mathbf{p}\}$ it holds that $distance(\mathbf{p}, \mathbf{q}') < distance(\mathbf{p}, \mathbf{q})$

Then, we can define the *k-distance neighborhood*, $N_k(\mathbf{p})$, of an object $\mathbf{p}$ as the set of points whose distance from $\mathbf{p}$ is not greater than the *k-distance*, i.e.

$$N_k(\mathbf{p}) = N_{k\text{-}distance(\mathbf{p})}(\mathbf{p}) = \{\ \mathbf{q} \in \mathcal{D}\backslash\{\mathbf{p}\} \mid distance(\mathbf{p}, \mathbf{q}) \leq k\text{-}distance(\mathbf{p}, \mathbf{q})\ \} \tag{3.1}$$

We can now introduce the notion of *reachability-distance* of an object $\mathbf{p}$ with respect to object $\mathbf{o}$ as:

$$reach\text{-}dist_k(\mathbf{p}, \mathbf{o}) = max\ \{\ k\text{-}distance(\mathbf{o}), distance(\mathbf{o}, \mathbf{p})\ \} \tag{3.2}$$

This distance is equal to the actual distance between object $\mathbf{p}$ and object $\mathbf{o}$ if the two are far enough, while when they are "sufficiently" close, the distance becomes the *k-distance* of $\mathbf{o}$. This is done because with the parameter k we can control the smoothing effect of the statistical fluctuations of the $distance(\mathbf{o}, \mathbf{p})$.

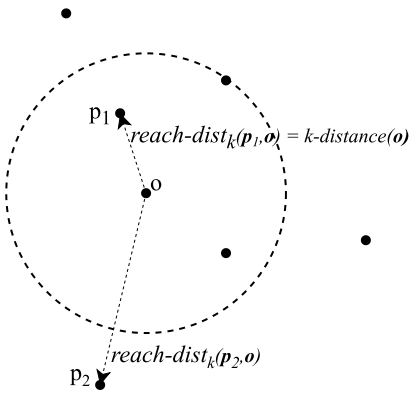


Figure 3.1: *reach-dist*($\mathbf{p}_1$, $\mathbf{o}$) and *reach-dist*($\mathbf{p}_2$, $\mathbf{o}$) for $k = 4$

Usually, in other density-based algorithms, there are two parameters that define the notion of density:

(i) **MinPts** specifying a minimum number of objects

(ii) **Volume** specifying the volume in which to compute outlierness

These two parameters determine a density *threshold* for the algorithm to operate: objects or regions are connected if their neighborhood densities exceed the given density threshold. In LOF, instead, only **MinPts** is needed and to determine the density in the neighborhood of object $\mathbf{p}$ it uses the values $\textit{reach-dist}_{MinPts}(\mathbf{p}, \mathbf{o})$, for $\mathbf{o} \in N_{MinPts}(\mathbf{p})$.

We can now compute the *local reachability density* of a point $\mathbf{p}$ as

$$lrd_{MinPts}(\mathbf{p}) = \frac{1}{\left(\frac{\sum_{\mathbf{o} \in N_{MinPts}(\mathbf{p})} \textit{reach-dist}_{MinPts}(\mathbf{p}, \mathbf{o})}{|N_{MinPts}(\mathbf{p})|} \right)} \tag{3.3}$$

Intuitively, the local reachability density of an object $\mathbf{p}$ is the inverse of the average reachability distance based on the *MinPts*-nearest neighbors of $\mathbf{p}$. This quantity can be ∞ if all the reachability distances sums up to 0, that can occur whene there are at least *MinPts* duplicates of $\mathbf{p}$ in the dataset.

Finally, we can introduce the *Local Outlier Factor* of an object $\mathbf{p}$ as:

$$LOF_{MinPts}(\mathbf{p}) = \frac{\sum_{\mathbf{o} \in N_{MinPts}(\mathbf{p})} \frac{lrd_{MinPts}(\mathbf{o})}{lrd_{MinPts}(\mathbf{p})}}{|N_{MinPts}(\mathbf{p})|} \tag{3.4}$$

that captures the degree to which we call $\mathbf{p}$ an outlier. It is computed as the average of the ratio of the *local reachability density* of $\mathbf{p}$ and those of $\mathbf{p}$'s *MinPts*-nearest neighbors. The lower the $\mathbf{p}$'s *local reachability density* is, and the higher the *local reachability density* of $\mathbf{p}$'s *MinPts*-nearest neighbors are, the higher is the *LOF* value of $\mathbf{p}$.

In the paper it is demonstrated that the *LOF* value of each point is bounded as follows:

$$\frac{direct_{min}(\mathbf{p})}{indirect_{max}(\mathbf{p})} \leq LOF(\mathbf{p}) \leq \frac{direct_{max}(\mathbf{p})}{indirect_{min}(\mathbf{p})} \tag{3.5}$$

where $direct_{min}(\mathbf{p}) = min\ \{\ \textit{reach-dist}(\mathbf{p}, \mathbf{q}) \mid \mathbf{q} \in N_{MinPts}(\mathbf{p})\ \}$ is the minimum reachability distance between $\mathbf{p}$ and a *MinPts*-nearest neighbor of $\mathbf{p}$ and $direct_{max}(\mathbf{p})$ the cor-

responding maximum; $indirect_{min}(\mathbf{p}) = min\ \{\ reach\text{-}dist(\mathbf{q}, \mathbf{o}) \mid \mathbf{q} \in N_{MinPts}(\mathbf{p}) \ \wedge \ \mathbf{o} \in N_{MinPts}(\mathbf{q})\ \}$ instead represents the minimum reachability distance between $\mathbf{q}$ and a *MinPts*-nearest neighbor of $\mathbf{q}$ and $indirect_{max}(\mathbf{p})$ the corresponding maximum.

Summing up, LOF is a very famous algorithm that is able to capture the relative degree of isolation between points. Before this algorithm, most of the methods for outliers detection consider being an outlier as a binary property. In this algorithm, instead, they give a *degree* of outlierness based on the isolation of the points. When an object is deep inside a cluster, its LOF value is approximately 1, while for other objects there are upper and lower bounds based on the *MinPts* parameter.

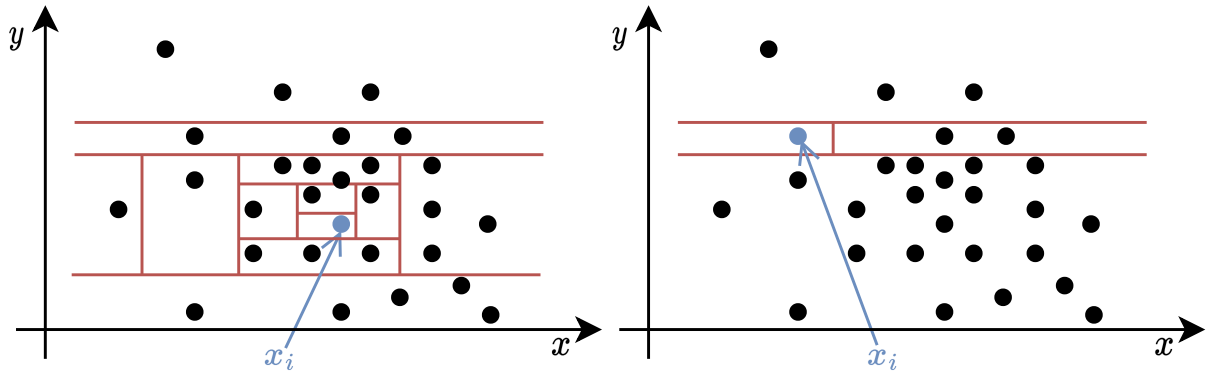
3.2. Isolation Forest

This algorithm [17] belongs to the family of *model-based* algorithms. This algorithm suggests an alternative method that explicitly isolates anomalies instead of incorporating normal instances behavior, unlike most existing model-based approaches that construct a profile of normal instances and then identify as anomalies those points that do not conform to the normal behavior.

The main idea of the algorithm is to build an **ensemble** of *random trees* that recursively partition a subset of the dataset $\mathcal{D}$ and gives a score to each point $\mathbf{p} \in \mathcal{D}$ based on the average path-length of the point in the ensemble. The depth in which we find a point determines how many partitions are required to isolate the point; ideally, if a point is easy to isolate it is in a less dense region or has some parameters significantly different from other points, so it is more probably an outlier. On the contrary, normal points usually are in denser regions and have attribute values similar to other points, so they are more difficult to isolate and they are more probably inliers.

The algorithm is divided in two phases: **training** and **testing**. In the former, we build the ensemble of trees that isolate points; in the latter, we compute the path-length of the points and so the anomaly score for each point.
The only parameters needed by the algorithm are the sub-sampling size ψ and the number of trees in the ensemble, t; in the paper, values suggested for this parameters are $\psi = 256$ and $t = 100$.

In order to show how the algorithm works, we need to introduce some concepts. First of all, we said that this algorithm builds an **ensemble** of **random trees**; an ensemble is



(a) Isolation of a normal point (b) Isolation of an anomaly

Figure 3.2: Two examples of how a point can be isolated. If it is a normal point, figure 3.2a, it is harder to isolate and more splits are required. If instead it is an anomaly, figure 3.2b, it is easier to isolate and less splits are required.

simply a collection of models that are used all together, in order to have a more accurate result.
A random tree, instead, is a *binary decision tree* that, for each *test-node*, selects randomly an attribute and a value for the split. Let T be a node of the tree; T can be an external node with no child, or an internal-node with one test and exactly two daughter nodes, (T_L, T_R). A test consists of an attribute a and a split value v such that the test $a < v$ divides data points in T_L and T_R.

Given a dataset of points $\mathcal{D} = \{ \mathbf{p}_1, \ ..., \ \mathbf{p}_n \}$ with n instances, to build an isolation tree (iTree), we recursively divide $\mathcal{D}$ by randomly selecting an attribute a and a value v until one of the following conditions are met:

(i) The tree reaches a height limit;

(ii) $|\mathcal{D}| = 1$;

(iii) all data in $\mathcal{D}$ have the same value.

When the tree is fully grown, the number of nodes is $2n - 1$ and so the total memory requirement is bounded linearly to n.

In order to give an *outlierness score* to points, it is possible to use the average path-length of the point in the trees, scaled by some factor. We can define the *path-length* of a point $\mathbf{p}$, $h(\mathbf{p})$, as the number of edges traversed in an iTree from the root node until an external node containing $\mathbf{p}$ is reached. To normalize the *path-length* we can compute

the average path-length as an unsuccessful search in a BST[1], since they are structurally equivalent to iTrees, that is:

$$c(n) = 2H(n-1) - \frac{2(n-1)}{n} \tag{3.6}$$

where $H(i)$ is the harmonic number, estimated as $H(i) \approx \ln(i) + e$, where the second term is the *Euler's constant*.
Since $c(n)$ is the average of $h(\mathbf{p})$ given n, we can compute the anomaly score as

$$s(\mathbf{p}, n) = 2^{-\frac{E(h(\mathbf{p}))}{c(n)}} \tag{3.7}$$

where $E(h(\mathbf{p}))$ is the average of $h(\mathbf{p})$ from a collection of iTrees. From equation (3.7) it is possible to notice that s is monotonic to $h(\mathbf{p})$:

- when $E(h(\mathbf{p})) \to 0$, $s \to 1$; $\mathbf{p}$ is an anomaly
- when $E(h(\mathbf{p})) \to n-1$, $s \to 0$; $\mathbf{p}$ is an inlier
- when $E(h(\mathbf{p})) \to c(n)$, $s \to 0.5$; $\mathbf{p}$ ambiguous situation

At this point, we can analyze more in depth the training and evaluation stages.

3.2.1. Training stage

In this stage, we build the iTrees by recursively partitioning the dataset, until instances are isolated or a specific height threshold is reached. In this case, the tree height limit l is automatically set by the sub-sampling size ψ : $l = ceiling(\log_2 \psi)$; this value represents the average tree height and the rationale to limit the trees at this height is that we are only interested in data points that have shorter-than-average path lengths, as those points are more probable to be outliers.
Moreover, in the paper is shown that building the ensemble on a sub-sampled dataset without replacement[2] leads to better results with respect when the whole dataset is used.
The algorithm of iForest is the following:

[1]Binary Search Tree
[2]Sampling means that you (randomly) select a sub-set of the original set. Without replacement means that samples in the sub-set are not statistically independent because once you have selected an object, you cannot select the same object again; in other words, the same object cannot be selected twice.

Algorithm 3.1 $iForest(\mathcal{D},\ \psi,\ t)$

Input: $\mathcal{D} = \{\ \mathbf{p} \mid \mathbf{p} \in \mathbb{R}^m\ \}$, ψ, t
Output: a set of t iTrees

1: $F \leftarrow$ Initialize forest
2: $l \leftarrow ceiling(\log_2 \psi)$ ▷ set height limit
3: **for all** $i \in [0,\ t]$ **do**
4: **D'** $\leftarrow sample(\mathcal{D},\ \psi)$
5: $F \leftarrow F\ \cup\ iTree($**D'**$,\ 0,\ l)$
6: **end for**
7: **return** F

Algorithm 3.2 $iTree(\mathcal{D},\ e,\ l)$

Input: $\mathcal{D} = \{\ \mathbf{p} \mid \mathbf{p} \in \mathbb{R}^m\ \}$, e - current tree height, l - height limit
Output: an iTree

1: **if** $e \geq l$ *or* $|\mathcal{D}| \leq 1$ **then**
2: **return** $exNode\{\ Size \leftarrow |\mathcal{D}|\ \}$
3: **else**
4: $A \leftarrow$ list of attributes in $\mathcal{D}$
5: $a \in A$ select random attribute
6: $v \in [min(a),\ max(a)]$ select random split value
7: $\mathcal{D}_L \leftarrow filter(\mathcal{D},\ a < v)$
8: $\mathcal{D}_R \leftarrow filter(\mathcal{D},\ a \geq v)$
9: **return** $inNode\{\ Left \leftarrow iTree(\mathcal{D}_L,\ e+1,\ l),$
10: $Right \leftarrow iTree(\mathcal{D}_R,\ e+1,\ l),$
11: $SplitAtt \leftarrow a$
12: $SplitValue \leftarrow v\}$
13: **end if**

3.2.2. Evaluation Stage

In this stage, an anomaly score s is derived from the expected path length, $E(h(\mathbf{p}))$, for each point in the dataset $\mathbf{p} \in \mathcal{D}$. Using *PathLength* (3.3), a single path length $h(\mathbf{p})$ is derived by counting the number of edges e from the root node to an external node as instance $\mathbf{p}$ traverses through an iTree. When $\mathbf{p}$ is terminated at an external node with more than one single point, the return value is $e + c(Size)$, where the adjustment account for an unbuilt subtree beyond the tree height limit.
After having calculated $h(\mathbf{p})$ for each iTree in the ensemble, we can compute $s(\mathbf{p},\ \psi)$ (3.7) with a complexity of $O(n \cdot t \cdot \log \psi)$, where $n = |\mathcal{D}|$.

Summing up, this algorithm proposes a new *model-based* solution to Anomaly Detection, by isolating instances rather than profiling normal instance. It exploits the property of outliers that are "few and different" and an iTree is able to isolate anomalies closer to

Algorithm 3.3 $PathLength(\mathbf{p},\ T,\ e)$

Input: $\mathbf{p} \in \mathcal{D}$, T - an iTree, e - current tree height
Output: path length of $\mathbf{p}$

```
1: if T is an external node then
2:     return e + c(T.size)                    ▷ c(·) is defined in equation 3.6
3: end if
4: a ← T.splitValue
5: if p_a < T.splitValue then
6:     return PathLength(p, T.left, e + 1)
7: else                                        ▷ p_a ≥ T.splitValue
8:     return PathLength(p, T.right, e + 1)
9: end if
```

the root of the tree, compared to normal points. Exploiting random decision trees, it is able to have both spatial and temporal linear complexities.
When examined more closely, it can be noticed that in reality this algorithm is *density-based*, because anomalies are found depending on their isolation level: if a point is in a high density region, it will be harder to isolate. On the contrary, if a point is in a less dense region, it will be easier to isolate; therefore, the concept of isolation is in reality based on the concept of density.

3.3. Preference Isolation Forest

Also this algorithm belongs to the *model-based* family, considering the problem of anomaly detection in a *pattern-recognition* setup, where anomalies are samples that deviate from certain structured patterns. As previously stated, this differs from other algorithms, such as *density-based* ones, in that we try to forecast the models (or patterns) that produced normal data rather than using a statistical measure to determine the outlierness score. Although density-based algorithms, in which the model describing inliers is a pdf[3], can be seen as a special case of these *pattern-recognition* algorithms, density-based and model-based algorithms are traditionally treated separately in the literature because they use different algorithms and methodologies.

Anomaly detection in this setting is very challenging, since anomalies cannot be directly removed without having identified each and every structure first, but at the same time anomalies hinder the identification of existing structures. For this reason, anomalies are often detected as a byproduct of a multi-structure estimation process, performed with robust model fitting algorithms, like e.g. RANSAC [11], J-Linkage [31] or T-Linkage [19].

[3]Probability Density Function

Within this framework, the structures underlying normal data are first identified and then all those points that do not conform with them are labeled as anomalous.

RANSAC[4] tries to fit a model to a dataset with a significat percentage of gross errors in a robust way. It follows the idea that rather than usign as much of the data as possible to obtain a model from data, it uses as small an initial dataset as feasible and enlarges this set with consistent data when possible. For example, given the task of fitting a line or an arc of a circle to a set of *two-dimensional* points, RANSAC would select respectively a set of two or three points, since two points are required to fit a line and three are required to fit a circle.
More formally, we can state the basic RANSAC paradigm as follows:
Given a model that requires a minimum of n points to instantiate its free parameters, and a set of data points $\mathcal{D}$ such that the number of points in $\mathcal{D}$ is greater than n ($|\mathcal{D}| \geq n$), randomly select a subset S_1 of n data points from $\mathbf{P}$ and instantiate the model. Use then the instantiated model M_1 to determine the subset S_1^* of points that are within some error tolerance of M_1. The set S_1^* is called **consensus set** of S_1.
If $|S_1^*| \geq t$, with t a threshold that is a function of the estimate of the number of errors in $\mathcal{D}$, use S_1^* to compute a new model M_1^*.
If $|S_1^*| < t$, randomly select a new subset S_2 and repeat the above process. If, after some predetermined number of trials, no consensus set with t or more members has been found, either solve the model with the largest consensus set found or terminate in failure.

Exploiting this robust model-fitting methods, PIF detects anomalies among structures whose nature is described by a given parametric function of unknown parameters. In order to do this, data is embedded in a high dimensional space called *preference space* and then performs anomaly detection in this space relying on *Pi-Forest*, an efficient tree-based method exploiting Voronoi Tessellations. Given a dataset $\mathcal{D}$ of *m-dimensional* points, the anomaly score of a point $s(\mathbf{p})$ is computed in two main steps:

(i) Embed the data in the preference space and

(ii) Adopt a tree-based isolation approach to detect anomalies in the preference space.

The entire algorithm is detailed in 3.4.

[4]Random Sample Consensus

Algorithm 3.4 *PIF*

Input: $\mathbf{p} \in \mathcal{D}$, t - number of trees, ψ - sub-sampling size, b - branching factor
Output: Anomaly scores $\{ s_\psi(\mathcal{E}(\mathbf{p}_i)) \}_{i=1,\ ...,\ n}$

```
/* Preference embedding */
M_k ← { θ_i }_{i=1, ..., k}                          ▷ Sample k models from D
P ← preferenceEmbedding(D, M_k) ∈ R^{nk}

/* Train Preference Isolation Forest */
F ← PI-Forest(P, t, ψ, b)

/* Scoring input data */
for all i ∈ [1, |P|] do
    h ← [0, ..., 0] ∈ R^t
    for all j ∈ [1, t] do
        T ← j-th PI-Tree in F
        [h]_j ← pathLength(pr_i, T, 0)
    end for
    s_ψ(p_i) ← 2^{-E(h(p_i))/c(ψ)}
end for
return { s_ψ(p_i) }_{i=1, ..., n}
```

3.3.1. Preference Embedding

Preference embedding refers to the procedure in which we map each point $\mathbf{p} \in \mathcal{D},\ \mathbf{p} \in \mathbb{R}^m$ in a k-dimensional vector, having components in the interval $[0,\ 1]$, via a mapping $\mathcal{E} : \mathcal{D} \to [0,\ 1]^k$. The space $[0,\ 1]^k$ is called **preference space**.
More precisely, the embedding depends on a family of models $\mathcal{F}$ parametric in θ, a set of k model instances $\{ \boldsymbol{\theta}_i \}_{i=1,\ ...,\ k}$ and an estimate of the standard deviation of the noise affecting the data, σ.
A sample $\mathbf{p}_i \in \mathcal{D}$ is then embedded to a vector $\mathbf{pr}_i = \mathcal{E}(\mathbf{p}_i)$ whose *j-th* component is defined as:

$$\mathbf{pr}_i = \begin{cases} \phi(\delta_{ij}) & \delta_{ij} \leq 3\sigma \\ 0 & otherwise \end{cases} \tag{3.8}$$

where $\delta_{ij} = \mathcal{F}(\mathbf{p}_i, \boldsymbol{\theta}_j)$ is the residual of $\mathbf{p}_i$ from the model $\boldsymbol{\theta}_j$; in other words, it represents how much the point deviates from the model. It is possible to use different functions to compute the preference, but ϕ must be monotonically decreasing in $[0,\ ...,\ 1]$ and such that $\phi(0) = 1$. In the paper the authors chose a Gaussian function of the form $\phi(\delta) = e^{-\frac{\delta^2}{\sigma}}$.
To give an idea of what the preference matrix P represents, we can say that the *j-th* component of the preference vector $\mathbf{pr}_i$, namely $[\mathbf{pr}_i]_j$, is the preference that a model $\boldsymbol{\theta}_j$

expresses for point $\mathbf{p}_i$: the closer the point $\mathbf{p}_i$ to the model $\boldsymbol{\theta}_j$, the higher the preference. The embedding function $\mathcal{E}$ maps dataset $\mathcal{D}$ to the set of preference vectors:

$$P = \{\ \mathbf{pr}_i = \mathcal{E}(\mathbf{p}_i) \mid \mathbf{p}_i \in \mathcal{D}\ \}$$

which represents the image of $\mathcal{D}$ through the embedding $\mathcal{E}$. As already said, the pool of models $\{\ \boldsymbol{\theta}_i\ \}_{i=1,\dots,k}$ of k models is built using a RANSAC-like strategy: at each step, a new model is built sampling from the whole dataset.

In order to measure the distance in this new space, one very useful metric is the *Tanimoto Distance*[5], because points conforming to the same structures share similar preferences, yielding low distances. On the contrary, outliers would result in null (or few) preferences to the majority of structures, thus resulting in sparse preference vectors that tend to have distance close to 1 with the majority of other samples.
Given two samples $\mathbf{pr}_i = \mathcal{E}(\mathbf{p}_i)$ and $\mathbf{pr}_j = \mathcal{E}(\mathbf{p}_j)$, their Tanimoto distance is:

$$\tau(\mathbf{pr}_i, \mathbf{pr}_j) = 1 - \frac{\langle \mathbf{pr}_i, \mathbf{pr}_j \rangle}{||\mathbf{pr}_i||^2 + ||\mathbf{pr}_j||^2 - \langle \mathbf{pr}_i, \mathbf{pr}_j \rangle} \tag{3.9}$$

3.3.2. Pi-Forest

At this point, after having built the preference matrix, we can isolate instances in the new space to identify outliers. Since in this space the Euclidean Distance, used e.g. in Isolation Forest 3.2, is not informative, the authors needed to introduce a new type of Decision Tree, called *Pi-Tree*, based on *Voronoi Tessellation* (2.4) and *Tanimoto Distance*.
The construction of a Pi-Tree is described in algorithm 3.5 and starts from a single region corresponding to the whole space $[0,\ 1]^k$ that is then recursively split in b sub-regions by randomly selecting b seeds $\{\ \mathbf{s}_i\ \}_{i=1,\dots,b} \subset P$. At this point, the dataset is partitioned into b subsets $\mathcal{P} = \{\ P_i\ \}_{i=1,\dots,b}$ and in each $P_i \subset P$ there are the points that have $\mathbf{s}_i$ as the closest seed, according to Tanimoto Distance. The number of seeds b is the branching factor of the tree associated to the splitting process; the construction of the tree stops when it is not possible to further split a region (i.e., the number of points in the region is less than b) or the tree has reached a maximum height, set as $l = \log_b \psi$, where ψ is the number of points used to build the tree. In figure 3.3 is illustrated the splitting procedure of a Pi-Tree.

[5]First cited in [26], it is a generalization of the Jaccard Distance. If all the elements of the two vectors are binary (can be only 0 or 1), the two distances are the same. They are not the same if the two vectors can also have discrete values between 0 and 1.

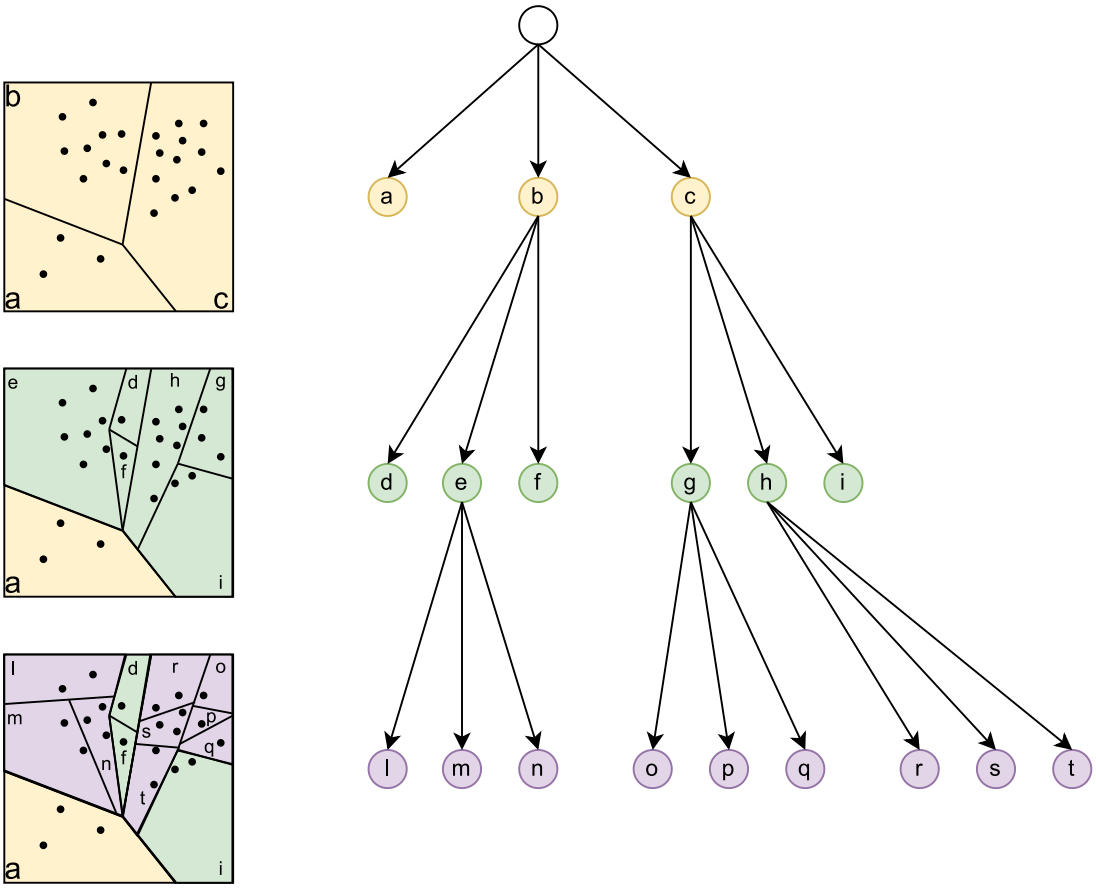


Figure 3.3: Example of a Pi-Tree with branching factor $b = 3$ and height limit $l = 3$, built from a set of points in $\mathbb{R}^2$. Every region is recursively split in b sub-regions and can be noted that the most isolated samples fall in leaves at lowest heights, such as a and d cells.

As already said for Isolation Forest (3.2), the outlierness score of a point depends on the height of the tree in which lies the point. This height is directly related to its separability, so a lower height corresponds to points that are separable with few splits, while a higher height corresponds to points hard to separate, thus are less probable to be outliers.
In order to decrease the random fluctuations of the results of the algorithm, it is possible to build an ensemble of models called *Pi-Forest*. We can build t Pi-Trees each trained on a subset $P' \subset P$ of the preference representations of the dataset $\mathcal{D}$; the sub-sampling factor is controlled by ψ.
The procedure is detailed in algorithm 3.6.

3.3.3. Anomaly Score

Anomaly scores are computed as in Isolation Forest and other tree-based isolation methods, like [18] and [22]. With reference to Isolation Forest, in this framework to compute

Algorithm 3.5 *Pi-Tree*

Input: P - preference matrix, e - current tree height, l - height limit, b - branching factor
Output: A Pi-Tree

if $e \geq l$ or $|P| < b$ **then**
  return $exNodeSize \leftarrow |P|$
else
  randomly select a set of b seeds $\{ \mathbf{s}_i \}_{i=1,...,b} \subset P$
  $\mathcal{P} \leftarrow voronoiPartition(P,\ \{ \mathbf{s}_i \}_{i=1,...,b})$
  $chNodes \leftarrow \emptyset$
  for all $i \in [1,\ b]$ **do**
    $chNodes \leftarrow chNodes \cup Pi\text{-}Tree(P_i,\ e+1,\ l,\ b)$
  end for
  return $inNode\ \{ ChildNodes \leftarrow chNodes,$
    $SplitPoints \leftarrow \{ \mathbf{s}_i \}_{i=1,...,b} \}$
end if

Algorithm 3.6 *Pi-Forest*

Input: P - preference matrix, ψ - sub-sampling size, t - number of trees, b - branching factor
Output: A set of t Pi-Tree

$F \leftarrow \emptyset$
set height limit $l = \log_b \psi$
for all $i \in [1,\ t]$ **do**
  $P' \leftarrow subSample(P,\ \psi)$
  $F \leftarrow FPi\text{-}Tree(P',\ 0,\ l,\ b)$
end for
return F

the anomaly scores of the points we need first to embed the points in the preference space, build the Pi-Forest and then pass each instance $\mathbf{pr} \in P$ through all the Pi-Trees of the ensemble. The heights reached in every tree are computed and collected in a vector $\mathbf{h}(\mathbf{pr}) = [h_1(\mathbf{pr}),\ ...,\ h_t(\mathbf{pr})]$ and are used to compute the anomaly score s_ψ of an element as in Isolation Forest:

$$s_\psi(\mathbf{pr}) = 2^{\left(-\frac{E(\mathbf{h}(\mathbf{p}))}{c(\psi)}\right)} \tag{3.10}$$

where $E(\mathbf{h}(\mathbf{pr}))$ is the mean value over the elements of $\mathbf{h}(\mathbf{pr})$ and $c(\psi)$ is an adjustment factor with the same role as in Isolation Forest, see equation 3.6.
Also the heights of elements in the trees are computed with a function similar to *Path-Length* - algorithm 3.3 - in Isolation Forest, but adapted for the Voronoi Tessellation:

Also in this case, all the observations for the adjustment factor has already been discussed in Isolation Forest. Notice that when the branching factor $b = 2$, the adjustment factor

Algorithm 3.7 *PathLength*

Input: **pr** - a sample, T - a Pi-Tree, e - current path length
Output: Path length of **pr**

1: **if** T*is an external node* **then**
2: **return** $e + c(T.size)$
3: **end if**
4: $childNode \leftarrow voronoiLocate(\mathbf{pr},\ T.splitPoints,\ T.childNodes)$
5: **return** $PathLength(\mathbf{pr},\ childNode,\ e+1)$

becomes as in 3.6, but if $n = 1$ the adjustment is 0 and if $n = 2$, the adjustment is 1.

The complexity of building the Pi-Forest is $\mathcal{O}(\psi \cdot t \cdot b \cdot log_b \psi)$ and the scoring phase has a complexity of $\mathcal{O}(n \cdot t \cdot b \cdot log_b \psi)$, where $n = |\mathcal{D}|$; with respect to Isolation Forest there is an additional overhead due to the preference embedding $\mathcal{E}(\cdot)$.

Concluding, we can state that Preference Isolation Forest has two main advantages over other isolation-based anomaly detection tools, that are the *so-called* preference trick and the Voronoi Tessellation. The former, allows to integrate useful information about normal data and to better characterize structure-less anomalies. The latter, instead, preserves the intrinsic distance of the preference space during the splitting process, since the Tanimoto Distance (3.9) is employed.
In the paper have been conducted several experiments on datasets generated by some pattern, like a line or a circle; in the majority of the experiments, this method gets better results because is able to distinguish between model-generated points and anomalous points. Other methods, instead, has troubles in identifying the patterns that generated the data.

3.4. Considerations

The novel **model-based** algorithm presented in this thesis aims to address the shortcomings of current approaches, in particular extending PIF applying a reconstruction-based ensemble. In the case of structured anomalies the density-based and distance-based approaches do not work, because those concepts are uninformative. In the model-based approach, instead, PIF represents an interesting approach since uses the preference trick to embed the dataset into a new space and applies a kind of Isolation Forest on this new space.
The preference trick, as explained in 3.3.1, is a measure of the deviation of the point from the learned model. Thus, the trick is to generate a new space in which the more models

have high preference (low deviation) for a point, the more that point is probably a normal point.

This approach is notable, but it assumes that the patterns in the dataset have a known shape and the algorithm cannot generalize if the pattern can not be, or it is too complex to be, expressed analytically. In order to build the preference representation, PIF samples a high number of models that try to express preferences towards normal points. Therefore, if we use a model representing a line to search for the pattern, we will discover only patterns that are represented as a line. Yet, if we use circles we will find only patterns with a circular shape.

Our approach tries to build more general models learned directly from data, which can take any shape; in this way, the model adapts to the shapes found in the dataset and do not forces the patterns to be as assumed. Is a similar, but actually different, solution to PIF, but in our approach the models automatically adapts depending on the manifolds in the dataset.

4 | Proposed Solution

4.1. Problem Formulation

First, we must conceptually distinguish between two concepts: the models that produce the data and the estimated machine learning models. **Pattern** refers to the actual model that specifies the manifold on which the data is located; it can represent geometrical figures, such as lines, parables, and circles, as well as more intricate curves; rather, we will refer to **models** as those that are built from data using statistical and machine learning techniques.

The ***input*** data is a dataset $\mathcal{D}$ made of noisy points in a *m-dimensional* space: $\mathcal{D} = \{\mathbf{p} + \eta | \mathbf{p} \in \mathbb{R}^m\}$, where $\eta = [\eta_1, .., \eta_i, ..., \eta_m]$ and $\eta_i \approx \mathcal{N}(0,\ \sigma_i^2)$ represents the noise affecting the data. The noise is modeled as an additive Gaussian noise with zero mean and unknown variance, applied to each dimension of the input vector.

Data points are divided in two disjoint subsets, the ***normal*** points and the ***anomalous*** points: $\mathcal{D} = \mathcal{N} \cup \mathcal{A}$ with $\mathcal{A} \cap \mathcal{N} = \emptyset$, where $\mathcal{N}$ is the set of normal points and $\mathcal{A}$ is the set of anomalies.
Normal points lies on a set of k **manifolds** in the *m-dimensional* space, $\mathcal{N} \subseteq \mathcal{M}_1 \cup ... \cup \mathcal{M}_k$; each manifold represents a generic parametric function $\mathcal{F}_{\theta_i}$, where θ_i represents the parameter vector of the *i-th* manifold. In a noisy-free setup all normal points lie on the manifolds, so each point p belonging to the *i-th* manifold, must satisfy $\mathcal{F}_{\theta_i}(\mathbf{p}) = 0$. Note that the *i-th* manifold is represented by $\mathcal{F}_{\theta_i}(\mathbf{p})$ and so it is composed as $\mathcal{M}_i = \{\mathbf{p} : \mathcal{F}_{\theta_i}(\mathbf{p}) = 0\}$. Since in our case the points are noisy, we can expect $\mathcal{F}_{\theta_i}(\mathbf{p}) \approx 0\ \forall\ \mathbf{p} \in \mathcal{M}_i$.
Anomalous points $\mathcal{A}$, instead, do not have a standard behaviour and so do not lie in any manifold; thus, they form a subset $\mathcal{A} \subset \mathcal{D}$ and we know that $\mathcal{A} \not\subseteq \mathcal{M}_1 \cup ... \cup \mathcal{M}_k$. An anomaly is an observation that deviates so much from other observations as to arouse suspicion that it was generated by a different mechanism.

The ***output*** of the algorithm $\mathcal{S} = \{s(\mathbf{p}) \in [0,1] \mid \mathbf{p} \in \mathcal{D}\}$ is an anomaly score for each point, $s : \mathcal{D} \to \mathbb{R}$, representing a sort of "probability" that the point is anomalous. Thus, the score of a point is higher for points that are more probable to be anomalous and lower for normal points: $s(a) \gg s(n)$ for all $a \in \mathcal{A}$ and $n \in \mathcal{N}$.
This task is quite challenging, since the total number of manifolds, their structure and the noise affecting the data are unknown. It is necessary to learn the parameters that govern the patterns directly from data, contaminated by the anomalies.

4.2. Rationale

Considering the problem of Anomaly Detection in a structured dataset, the approach presented in PIF represents a big change. Thanks to the preference trick we are able to embed points in a new space, representing how much the learned models express a preference for each point.
The preference expressed by a model $\mathcal{G}_j$ for a point $\mathbf{p}_i \in \mathcal{D}$ represents the inverse of the deviation of the point from the model. In practice, the more the point is near to its representation built by the model, the more the preference $[\mathbf{pr}_i]_j$ is high. Therefore, the preference space reflects how much each point is fitted by each model.
If we would know the true pattern that generated normal points, all normal points would have a very high preference - because the deviation from the pattern is relatively small - and all the anomalies would have a very low preference - because the deviation is higher.

Note that in the following we will use the terms *preference space* and *preference matrix* as synonyms, but in reality they are different: the preference *matrix* refers to a particular pool of models, while the preference *space* is the space in which data points are projected. As an example, if we build two different pools of models the preference matrices will be different, but the space in which the points are projected is the same.

The preference space allows us to isolate data points depending on how many models have expressed a high preference for the points, thus it is mandatory to be able to build a preference space that reflects the nature of the dataset. In order to do that, we need to build models that can understand which patterns governs the data and replicate them.
This is where the **novelty** of the approach developed in this thesis comes in: we build a pool of non-linear Neural Networks that are able to extract meaningful and appropriate patterns from the dataset without any prior knowledge on the nature of this patterns, as opposed to the way it is done in PIF, in which we assume a specific pattern to search and use that model to build the preference matrix. We have studied two kinds of Neural

Networks, the Auto-Encoders [8] and the Self-Organizing-Maps [15]. Both this models are able to adapt their weights to create a *lower-dimensional* representation of the dataset, being able to hold only important features; since the anomalies represent data outside the patterns, the Neural Networks are able to discard their information and rely only on normal points to build the pattern representation.

PIF assumes that the type of patterns found in the dataset is known; e.g., it is known that normal data are originated from a line or from a circle; consequently, PIF will use, respectively, a line and a circle as learners and therefore the preference matrix will depend on the preferences expressed by this models.
Such assumption reduces the use cases of PIF, because especially in real-world cases it is rare that the normal behavior is expressed by such simple models.
Our algorithm, instead, does not have that limiting assumptions, but exploits the fact that several little models are built and try to approximate the patterns using those models. We want to build more general models learned directly from data, which can take any shape; in this way, the model adapts to the shapes found in the dataset and do not forces the patterns to be as assumed. Is a similar, but actually different, solution to PIF, because in our approach the models automatically adapts depending on the manifolds in the dataset.

The **goal of this thesis** is to show that using an ensemble of non-linear models such as Neural networks, trained on a subset of the dataset, we are able to approximate the patterns.
We build models of lower degree with respect to the patterns' degree that try to approximate the manifold locally. At first, we will use as models Linear Regression and very simple Neural Networks with Auto-Encoder structure, but then we will also explore more complex and different models, such as deeper and wider Auto-Encoders and Self Organizing Maps.

4.3. Algorithm

The algorithm we propose for Anomaly Detection in structured normal data contaminated with high percentage of anomalies is structured similarly as PIF algorithm 3.4. It is composed of four main phases, that are listed below. The algorithm is shown in 4.1.

- **Model building**: (lines 1-3) build the pool of models and train each model on a subset of the dataset, $\mathcal{MSS} \subset \mathcal{D}$, called **Minimum Sample Set** and has **cardinality** ρ, i.e. $|\mathcal{MSS}| = \rho$. This subset has the same role as in PIF, with the difference that in this case it represents the data instances on which train each model, while

in PIF it represents the minimum samples needed to instantiate a model, e.g. two or three points are needed to instantiate respectively a line or a circle.

- **Preference Embedding**: (lines 5-18) embed each point into the preference space, generating the preference matrix $P \in \mathbb{R}^{N \times k}$ with $|\mathcal{D}| = N$ and k the number of models. The value of element (i, j) depends on the preference expressed by model $\mathcal{G}_j$ for point $\mathbf{p}_i$, noted as δ_{ij}.
- **Isolation Voronoi Forest**: (lines 20-21) apply the iVor (3.6) on the generated preference matrix, building the isolation forest made of Pi-Trees (3.5) on which compute the score.
- **Score computation**: (lines 23-30) compute the path length of each instance in each tree and finally compute the score, averaging between the computed path lengths and scaling by $c(\psi)$, refer to 3.6.

4.4. Pattern learners

4.4.1. Line and Plane

The first model from which to start is the most simple one, a straight line. It is known [6] that it is possible to linearly approximate all kind of curves at each point thanks to the tangent passing for that point.
Given a function $y = f(x)$ that is differentiable at a point p, the function $f(\cdot)$ is locally linear and at the point $(p, f(p))$ can be approximated by its tangent. Thus, we can approximate the original function $f(\cdot)$ with a simpler function $L(\cdot)$ that is linear. We need to recall that when $f(x)$ is differentiable at $x = p$, the value of $f'(p)$ provides the slope of the tangent line to $y = f(x)$ at the point $(p, f(p))$. Therefore, the tangent line will have an explicit form like

$$L(x) = f'(p)(x - p) + f(p) \tag{4.1}$$

We will call $L(x)$ the *first degree local linearization* of $f(x)$ at the point $(p, f(p))$ i.e. $L(x) \approx f(x)$ for $x \approx p$.

At this point we know that we can locally approximate any given function $f(\cdot)$ with a line, at each point; thus, ideally, with an infinite number of lines we can also derive the shape of the function. Starting from this idea, we would like to approximate any manifold with local approximations. In a *two-dimensional* case, the linear approximation can be

Algorithm 4.1 *Neural-PIF*

Input: $\mathcal{D}$ - dataset, k - number of models, ρ - $\mathcal{MSS}$ size, τ - threshold, t - number of trees, ψ - sub-sampling size, b - branching factor
Output: Anomaly scores $\{\ s_\psi(\mathcal{E}(\mathbf{p}_i))\ \}_{i=1,\dots,n}$

```
/* Build models */
MSS ∈ R^{N×ρ} ← uniformSampling(D, ρ)   ▷ Sample the Minimum Sample Sets, one for each instance n ∈ N, of size ρ
M_k ← buildModels(D, k, ρ)

/* Preference embedding E */
P ∈ R^{nk}                                   ▷ initialize matrix
for all i ∈ [0, n] do
    p_i ← i-th point of dataset
    for all j ∈ [0, k] do
        p̂_j ← prediction of j-th model for p_i
        δ_ij ← distance between p_i and p̂_i
        if δ_ij ≤ τ then
            P_ij ← exp(−δ_ij/τ)
        else
            P_ij ← 0
        end if
    end for
end for

/* Train Preference Isolation Forest */
F ← PI-Forest(P, t, ψ, b)

/* Scoring input data */
for all i ∈ [1, |P|] do
    h ← [0, ..., 0] ∈ R^t
    for all j ∈ [1, t] do
        T ← j-th PI-Tree in F
        [h]_j ← pathLength(p_i, T, 0)
    end for
    s_ψ(p_i) ← 2^{−E(h(p_i))/c(ψ)}
end for
return s_ψ(p_i)_{i=1, ..., n}
```

thought as a line; in a *three-dimensional* case, the linear approximation is a plane.

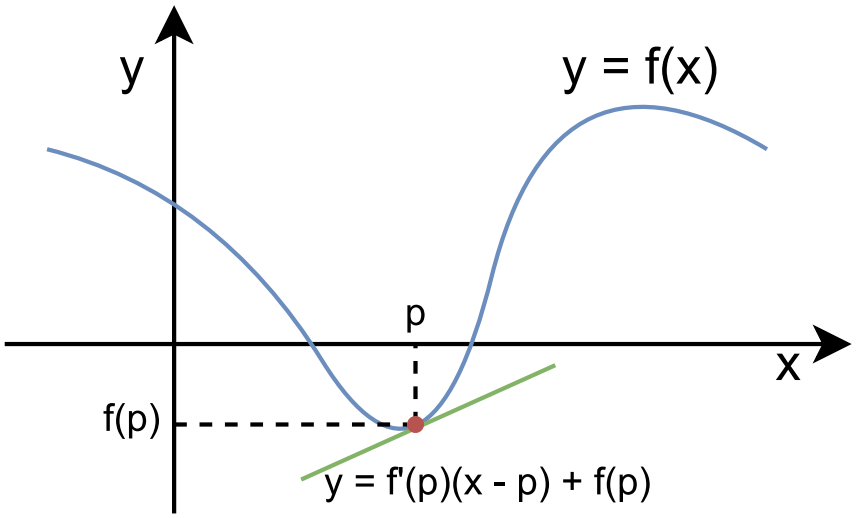


Figure 4.1: Example of a linear approximation on a generic function $y = f(x)$. In blue the true function, in green the tangent at the point $(p, f(p))$ that is marked in red.

In fact, considering a plane containing the point $\mathbf{p} = (x_0, y_0, z_0)$ with a normal vector $\mathbf{n} = \begin{bmatrix} a, & b, & c, \end{bmatrix}$, its equation is given by

$$a(x - x_0) + b(y - y_0) + c(z - z_0) = 0. \tag{4.2}$$

We know [4] also that the gradient vector $\nabla f(x_0, y_0)$ is perpendicular to the level curve $f(x, y) = k$ at the point $\mathbf{p}$; likewise, the gradient vector $\nabla f(x_0, y_0, z_0)$ is perpendicular to the level surface $f(x, y, z) = k$ at the point $\mathbf{p}$.

We should recall that the gradient vector is $\nabla f = \begin{bmatrix} \frac{\partial f}{\partial x}, & \frac{\partial f}{\partial y}, & \frac{\partial f}{\partial z} \end{bmatrix}$. Therefore, the tangent plane to the surface given by $f(x, y, z) = k$ at the point $\mathbf{p}$, has the equation

$$\frac{\partial f(x_0, y_0, z_0)}{\partial x}(x - x_0) + \frac{\partial f(x_0, y_0, z_0)}{\partial y}(y - y_0) + \frac{\partial f(x_0, y_0, z_0)}{\partial z}(z - z_0) = 0 \tag{4.3}$$

If we consider now a generic function of the form $z = f(x, y)$ and $F(x, y, z) = f(x, y) - z$ we can see that the surface given by $z = f(x, y)$ is identical to the surface given by $F(x, y, z) = 0$. The gradient of this function will be $\nabla F = \begin{bmatrix} \frac{\partial F}{\partial x}, & \frac{\partial F}{\partial y} & -1 \end{bmatrix}$, because the last element is computed as $\frac{\partial}{\partial z}(f(x, y) - z)$.
From the equation of the plane obtained above and solving for z, we obtain the local 2D approximation of a 3D surface:

$$z = \frac{\partial f(x_0, y_0)}{\partial x}(x - x_0) + \frac{\partial f(x_0, y_0)}{\partial y}(y - y_0) + f(x_0, y_0) \tag{4.4}$$

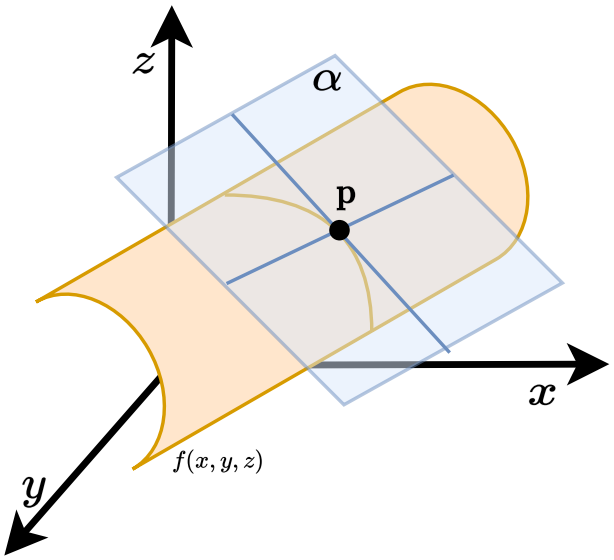


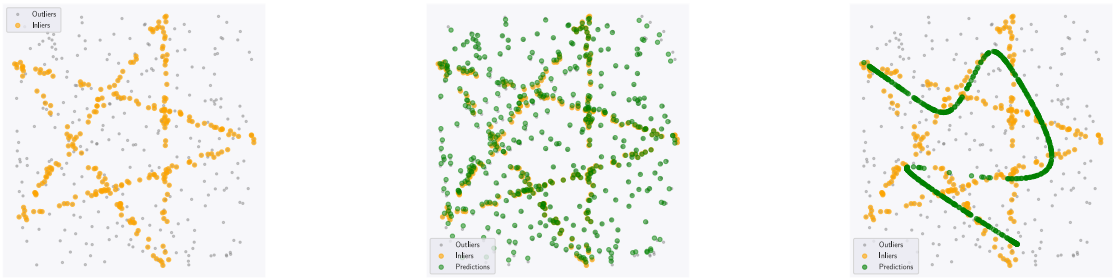
Figure 4.2: Example of a plane, α, the is tangent to the surface $f(x, y, z)$ at the point **p**.

Therefore, our idea is to use the model line (or plane for the *three-dimensional* space) not as the known pattern as in PIF, but as the model with which we can approximate any manifold. The algorithm using this model will function in the same way as in PIF when the pattern is a set of lines, but we show this procedure in this context in order to have a baseline to the next models, that should be able to locally approximate more complex functions and to better incorporate the patterns.

4.4.2. Auto-encoders

Auto-encoders (2.3.1) are a particular type of neural network whose structure consists of two parts: an encoder and a decoder.
The *encoder* tries to encode the input data, by reducing its dimensionality and creating the *latent representation*. The *decoder*, on the other hand, receives the latent representation and tries to reconstruct the original input. The trick that allows the network to **not** learn the identity matrix, which would be a copy of the original dataset, comes from the fact that the latent representation has fewer dimensions than the original input, thus it is forced to learn a representation of the input different from the identity.
The output of the network and the training performed varies a lot depending on this factors:

1. The **depth** of the network;
2. The **width** of the network;
3. The **activation** function of the layers. The major difference is between linear and



(a) The dataset (b) MLP (c) Auto-Encoder

Figure 4.3: This figure shows the utility of the constraint on the latent representation's number of dimensions. In image 4.3b, a Multi Layer Perceptron (MLP) network has learned to reconstruct the input vector generating a new vector in the same space; the problem is that this network didn't generate any latent representation with fewer dimension, thus it has learnt the identity matrix. On the contrary, in image 4.3c, an Auto-Encoder with a latent representation with less dimensions than the input space has learned a new representation of the input dataset, not the identity matrix.

non-linear activation functions: the latter allow the latent-space to be a non-linear transformation of the original input space and so allows to better capture the structure of the dataset.

Usually, as explained in [14], auto-encoders for anomaly detection are trained on a dataset of normal instances that allow the network to understand the underlying pattern, being able to incorporate the patterns present in the normal instances. In the evaluation stage, the one in which an anomaly score is given to the instances, the score depends on the reconstruction error made by the network: the higher the error, the more probable the instance is anomalous. Indeed, since the network has incorporated the normal pattern, when the input is anomalous the reconstructed output will be very different from the input, while when the input is a normal point, the output of the network will be very similar to the original input.

This method produces very good results, but it highly depends on the dataset used for training. In fact, it needs a human-labelled dataset in which normal points are separated from anomalies and the part of the dataset containing normal points should take in account all the possible behaviours that can be defined "normal"; if the set of normal points contains contaminations, the network will learn also the anomalous behaviour and if it is not highly representative of the normal behaviour, the model will produce many

false negatives.
The advantage is that it can be used to label single instances as normal or anomalous; indeed, once the network is trained it is possible to use it for any other instance of the same kind as the training data.

Our algorithm, instead, is able to train the pool of models directly on the **unlabelled** dataset, without the need for a separation between normal points and anomalies. Training the auto-encoders on a subset of the dataset allows each model to learn the pattern of a portion of the data and extract from it the normal behaviour. Using the preference trick, each model assigns a preference for its own "normal" instances using the learnt representation; in this way, we have several experts of some portion of the dataset and with Pi-Forest the algorithm can understand which instances are normal and which are not.

In the following sections we will show that an auto-encoder trained on a pure dataset, i.e. that does not contain anomalies, is able to learn the equation of a line or of a plane and the relationship between network's weights and coefficients of the pattern.
This is crucial for our purposes, because it means that the auto-encoder is actually able to learn a useful representation of the pattern encoded, both in the encoder and in the decoder. Obviously this are examples showing the case in which the dataset is pure and the pattern is rather simple, but they nonetheless demonstrate the possibilities of an auto-encoder.

Relationship between auto-encoders and lines

Consider an auto-encoder with two input neurons, two output neurons, one hidden layer comprising one neuron and having the identity function $f(x) = x$ as activation. It will have four weights $(w_{02}, w_{12}, w_{23}, w_{24})$ and three biases (b_2, b_3, b_4). As we know, the output of a neuron is computed as the dot product of the weights and the input vector. Thus, the output of the hidden neuron can be computed as follows:

$$h = w_{02}x + w_{12}y + b_2 \tag{4.5}$$

Therefore, the output of the network, $\hat{x}, \hat{y}$ is:

$$\hat{x} = w_{23} \cdot h + b_3 \tag{4.6}$$

$$\hat{y} = w_{24} \cdot h + b_4 \tag{4.7}$$

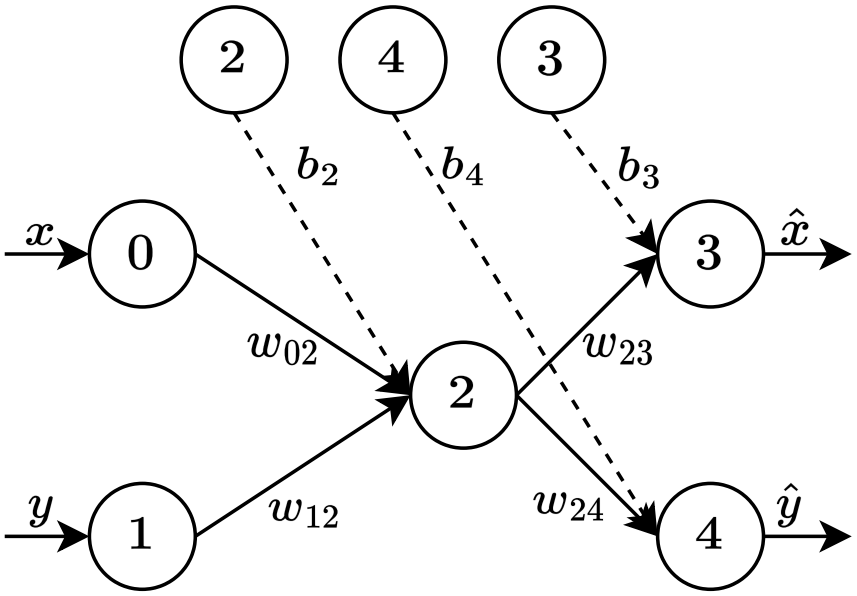


Figure 4.4: Structure of the network used in this section. The pedices of the weights are the starting neuron's index and the connected neuron's index. So, for example, weight w_{02} goes from neuron n_0 to neuron n_2, or weight w_{23} goes from neuron n_2 to neuron n_3. The biases have the same index to which they belong; for example, bias b_4 is the bias of neuron n_4.

The goal of the network is to encode the input vector computing its latent representation and then decode the latent representation to reconstruct the input vector. It is trained to minimize the MSE (equation 2.13), thus the error is computed calculating the euclidean distance between the prediction and the real point. Thus, the network must incorporate the data on which is trained to be able to replicate the same structure.
Training this kind of network on a pure dataset in which the pattern is a line, we will show that exists a relationship between the weights and biases of the network and the coefficients of the pattern. Indeed, it is possible to compute those coefficients using the weights of the decoder part of the network.

From [2] we know that given two points, $P_1 = \begin{bmatrix} x_1 & y_1 \end{bmatrix}$ and $P_2 = \begin{bmatrix} x_2 & y_2 \end{bmatrix}$, the line passing the two points is equal to the vector P_2P_1: $v_r = \begin{bmatrix} x_2 - x_1 \\ y_2 - y_1 \end{bmatrix}$. Given the vector, we can write the vectorial equation of a line:

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x_1 \\ y_1 \end{bmatrix} + t \cdot \begin{bmatrix} x_2 - x_1 \\ y_2 - y_1 \end{bmatrix} \tag{4.8}$$

Considering now a generic point P belonging to the line, this forms a vector with respect to a point of the line, e.g. P_1:

$$\overrightarrow{\mathrm{P_1P}} = P - P_1 = \begin{bmatrix} x - x_1 \\ y - y_1 \end{bmatrix} \tag{4.9}$$

The vector P_1P must be parallel and proportional to the vector P_1P_2, because it lies on the line equivalent to the vector v_r; thus, the vectors P_1P and P_1P_2 must be linearly dependent and the equation of the line can be obtained solving the equation

$$det\left(\begin{bmatrix} x - x_1 & x_2 - x_1 \\ y - y_1 & y_2 - y_1 \end{bmatrix}\right) = 0 \tag{4.10}$$

that can be solved as follows:

$$\begin{gathered} (x - x_1)(y_2 - y_1) - (y - y_1)(x_2 - x_1) = 0 \\ xy_2 - x_1y_2 + x_1y_1 - yx_2 + yx_1 + y_1x_2 - y_1x_1 = 0 \\ x_1y - x_2y - xy_1 + x_2y_1 + xy_2 - x_1y_2 = 0 \\ y(x_1 - x_2) + x(y_2 - y_1) - x_1y_2 + x_2y_1 = 0 \\ y(x_1 - x_2) = -(y_2 - y_1)x + x_1y_2 - x_2y_1 \\ y(x_1 - x_2) = (y_1 - y_2)x + x_1y_2 - x_2y_1 \\ y = \frac{y_1 - y_2}{x_1 - x_2}x + \frac{x_1y_2 - x_2y_1}{x_1 - x_2} \end{gathered} \tag{4.11}$$

From this, we know that the slope of the line is $m = \frac{y_1 - y_2}{x_1 - x_2}$ and the intercept is $q = \frac{x_1y_2 - x_2y_1}{x_1 - x_2}$.

Given two points $\hat{P}_1$ and $\hat{P}_2$ predicted by the network and using equations 4.6 and 4.7, we can write the following:

$$\begin{aligned} \hat{P}_1 &= \begin{bmatrix} \hat{x}_1 \\ \hat{y}_1 \end{bmatrix} = \begin{bmatrix} w_{23}h_1 + b_3 \\ w_{24}h_1 + b_4 \end{bmatrix} \\ \hat{P}_2 &= \begin{bmatrix} \hat{x}_2 \\ \hat{y}_2 \end{bmatrix} = \begin{bmatrix} w_{23}h_2 + b_3 \\ w_{24}h_2 + b_4 \end{bmatrix} \end{aligned} \tag{4.12}$$

where h_1 and h_2 are the hidden representations for the first and second point, respectively. Substituting this values in the slope formula, we obtain the following result:

$$\begin{aligned} m &= \frac{\hat{y}_1 - \hat{y}_2}{\hat{x}_1 - \hat{x}_2} \\ m &= \frac{w_{24}h_1 + b_4 - w_{24}h_2 - b_4}{w_{23}h_1 + b_3 - w_{23}h_2 - b_3} \\ m &= \frac{w_{24}(h_1 - h_2)}{w_{23}(h_1 - h_2)} \\ m &= \frac{w_{24}}{w_{23}} \end{aligned} \tag{4.13}$$

The same can be done with the equation of the intercept:

$$\begin{aligned} q &= \frac{\hat{x}_1\hat{y}_2 - \hat{x}_2\hat{y}_1}{\hat{x}_1 - \hat{x}_2} \\ q &= \frac{(w_{23}h_1 + b_3)(w_{24}h_2 + b_4) - (w_{23}h_2 + b_3)(w_{24}h_1 + b_4)}{w_{23}h_1 + b_3 - w_{23}h_2 - b_3} \\ q &= \frac{w_{23}h_1b_4 + w_{24}h_2b_3 - w_{23}h_2b_4 - w_{24}h_1b_3}{w_{23}(h_1 - h_2)} \\ q &= \frac{w_{24}b_3(h_2 - h_1) + w_{23}b_4(h_1 - h_2)}{w_{23}(h_1 - h_2)} \\ q &= \frac{(h_1 - h_2)(w_{23}b_4 - w_{24}b_3)}{w_{23}(h_1 - h_2)} \\ q &= b_4 - \frac{w_{24}}{w_{23}}b_3 \end{aligned} \tag{4.14}$$

At this point, we know that the network is able to encode the two input numbers in one number and then construct the equation of the line in the decoder. Indeed, we have shown that from the weights and the biases of the decoder is possible to reconstruct the equation of the line that generated the pattern.

One simple case that shows the correctness of this result is when the latent representation is equal to the x coordinate of the input point.
In this case, to compute the output for the x coordinate is necessary to have $w_{23} = 1$ and $b_3 = 0$. Therefore, the slope of the line will be $m = \frac{w_{24}}{w_{23}} = \frac{w_{24}}{1} = w_{24}$ and the intercept will be $q = b_4 - 1 \cdot b_3 = b_4$.
Thus, the y coordinate of the input point is computed in the exact same way as is done when generating the pattern, because they are computed in the following way:

$$\begin{aligned} y &= m \cdot x + q \\ \hat{y} &= w_{24} \cdot h + b_4 = m \cdot x + q \quad if\ x = h \end{aligned} \tag{4.15}$$

Relationship between auto-encoders and planes

Moving from the *two-dimensional* to the *three-dimensional* space, we want to show that the auto-encoder architecture is also able to incorporate the equation of a plane, that is the *three-dimensional* "equivalent" of the line.
In this case the network has a structure made of three input neurons, two neurons in the hidden layer and three neurons in the output layer. The *three-dimensional* input vector is embedded in a *two-dimensional* vector and then the original point will be reconstructed from the latent representation thanks to the decoder ability.

Similarly as we have done for the *two-dimensional* case, we will define the network as follows: we have three input neurons, n_0, n_1, n_2 corresponding to the tree coordinate of an input point, two hidden neurons n_3 and n_4 and finally three output neurons, n_5, n_6, n_7, corresponding to the three coordinates of the predicted point. Also in this case

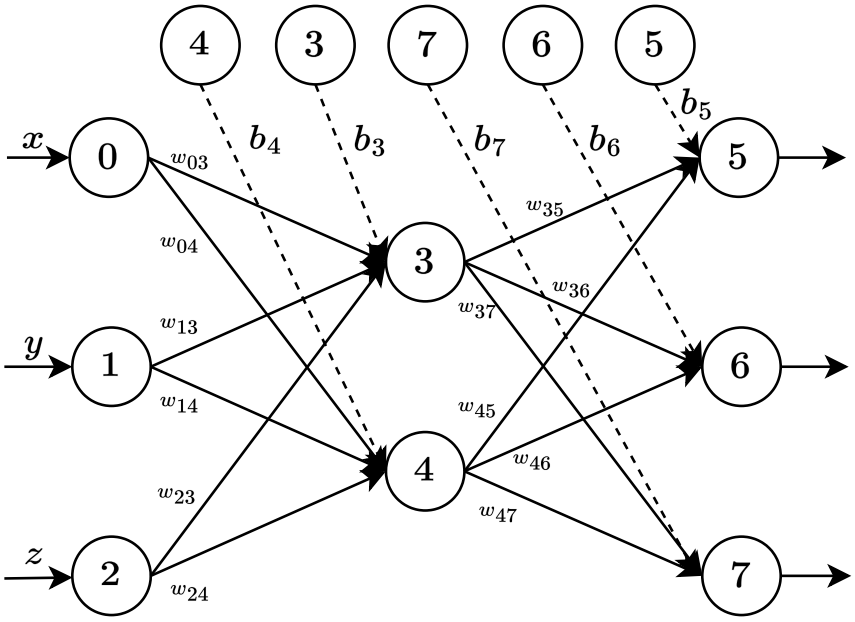


Figure 4.5: The structure of the network used in this section. The convention for naming the weights and biases is the same used in figure 4.4.

we can define the output of each neuron combining the weights entering the neuron and the biases. The hidden representation will be represented as a *two-dimensional* vector, defined as follows:

$$\begin{aligned} h &= (h_3, h_4) \\ h_3 &= w_{03}x + w_{13}y + w_{23}z + b_3 \\ h_4 &= w_{04}x + w_{14}y + w_{24}z + b_4 \end{aligned} \tag{4.16}$$

The output values are instead computed as follows:

$$\begin{aligned}\hat{x} &= w_{35}h_3 + w_{45}h_4 + b_5 \\ \hat{y} &= w_{36}h_3 + w_{46}h_4 + b_6 \\ \hat{z} &= w_{37}h_3 + w_{47}h_4 + b_7\end{aligned} \tag{4.17}$$

As already done for the line, we will show how it is possible to formulate the equation of a plane. At section A.1 there is the code with which we made the calculations to obtain the formulas.
From [3] we know that the canonical equation of a plane is $ax + by + cz + d = 0$, where $a, b, c, d \in \mathbb{R}$ and a, b, c are not simultaneously null. From Euclidean Geometry, we know that for three non-aligned points passes one and only one plane; thus, fixing an ortonormal cartesian reference system $RC(O, i, j, k)$, consider three non-aligned points:

$$\begin{aligned}P_1 &= (x_1, y_1, z_1) \\ P_2 &= (x_2, y_2, z_2) \\ P_3 &= (x_3, y_3, z_3)\end{aligned} \tag{4.18}$$

Given that P_1, P_2, P_3 are not aligned, the vectors of each chosen couple, $\overrightarrow{P_1P_2}, \overrightarrow{P_1P_3}, \overrightarrow{P_2P_3}$ are linearly independent between them. If this is not the case, one couple will be linearly dependent and would be parallel, thus we would have three points lying on the same line. Considering a generic point $P = (x, y, z)$, will belong to the plane if and only if the vectors $\overrightarrow{P_1P}, \overrightarrow{P_1P_2}, \overrightarrow{P_1P_3}$ are coplanar. This is equivalent to ask that the determinant of the matrix having as rows the components of these vectors with respect to the base vector (i, j, k) is null.
We can rewrite the three vectors as following:

$$\begin{aligned}\overrightarrow{P_1P} &= P - P_1 = (x - x_1)i + (y - y_1)j + (z - z_1)k \\ \overrightarrow{P_1P_2} &= P_2 - P_1 = (x_2 - x_1)i + (y_2 - y_1)j + (z_2 - z_1)k \\ \overrightarrow{P_1P_3} &= P_3 - P_1 = (x_3 - x_1)i + (y_3 - y_1)j + (z_3 - z_1)k\end{aligned} \tag{4.19}$$

Finally, we can say that the three vectors $\overrightarrow{P_1P}$, $\overrightarrow{P_1P_2}$ and $\overrightarrow{P_1P_3}$ are coplanars if and only if

$$det\left(\begin{bmatrix} x - x_1 & y - y_1 & z - z_1 \\ x_2 - x_1 & y_2 - y_1 & z_2 - z_1 \\ x_3 - x_1 & y_3 - y_1 & z_3 - z_1 \end{bmatrix}\right) = 0 \tag{4.20}$$

Solving the equation with Laplace rules with respect to the first row, we get

$$a(x - x_1) + b(y - y_1) + c(z - z_1) = 0 \tag{4.21}$$

where

$$\begin{aligned} a &= \ det\left(\begin{bmatrix} y_2 - y_1 & z_2 - z_1 \\ y_3 - y_1 & z_3 - z_1 \end{bmatrix}\right) \\ b &= -det\left(\begin{bmatrix} x_2 - x_1 & z_2 - z_1 \\ x_3 - x_1 & z_3 - z_1 \end{bmatrix}\right) \\ c &= \ det\left(\begin{bmatrix} x_2 - x_1 & y_2 - y_1 \\ x_3 - x_1 & y_3 - y_1 \end{bmatrix}\right) \end{aligned} \tag{4.22}$$

Rearranging equation 4.21 we get

$$\begin{aligned} a(x - x_1) + b(y - y_1) + c(z - z_1) &= 0 \\ ax - ax_1 + by - by_1 + cz - cz_1 &= 0 \\ ax + by + cz - ax_1 - by_1 - cz_1 &= 0 \end{aligned} \tag{4.23}$$

And with $d = -ax_1 - by_1 - cz_1$, we get the equation of the plane: $ax + by + cz + d = 0$. From this, we can explicit the z-coordinate obtaining

$$z = -\frac{a}{c}x - \frac{b}{c}y - \frac{d}{c} \tag{4.24}$$

At this point, we know how to compute the parameters of the equation of a plane given three points that are no coplanar. Our goal is to show that from the parameters of the neural network at figure 4.5 it is possible to compute the equation of a plane, independent from the input point and the latent representation.

As done in the case of the line, consider three points predicted from the network:

$$\begin{aligned}
\hat{P}_1 = \begin{bmatrix} \hat{x}_1 \\ \hat{y}_1 \\ \hat{z}_1 \end{bmatrix} &= \begin{bmatrix} w_{35}h_{3_1} + w_{45}h_{4_1} + b_5 \\ w_{36}h_{3_1} + w_{46}h_{4_1} + b_6 \\ w_{37}h_{3_1} + w_{47}h_{4_1} + b_7 \end{bmatrix} \\
\hat{P}_2 = \begin{bmatrix} \hat{x}_2 \\ \hat{y}_2 \\ \hat{z}_2 \end{bmatrix} &= \begin{bmatrix} w_{35}h_{3_2} + w_{45}h_{4_2} + b_5 \\ w_{36}h_{3_2} + w_{46}h_{4_2} + b_6 \\ w_{37}h_{3_2} + w_{47}h_{4_2} + b_7 \end{bmatrix} \\
\hat{P}_3 = \begin{bmatrix} \hat{x}_3 \\ \hat{y}_3 \\ \hat{z}_3 \end{bmatrix} &= \begin{bmatrix} w_{35}h_{3_3} + w_{45}h_{4_3} + b_5 \\ w_{36}h_{3_3} + w_{46}h_{4_3} + b_6 \\ w_{37}h_{3_3} + w_{47}h_{4_3} + b_7 \end{bmatrix}
\end{aligned} \tag{4.25}$$

Now, we can compute the coefficients of the plane substituting the three points in place of the generic coordinates, developing equation 4.22:

$$\begin{aligned}
a &= \hat{y}_1\hat{z}_2 - \hat{y}_1\hat{z}_3 - \hat{y}_2\hat{z}_1 + \hat{y}_2\hat{z}_3 + \hat{y}_3\hat{z}_1 - \hat{y}_3\hat{z}_2 \\
b &= -\hat{x}_1\hat{z}_2 + \hat{x}_1\hat{z}_3 + \hat{x}_2\hat{z}_1 - \hat{x}_2\hat{z}_3 - \hat{x}_3\hat{z}_1 + \hat{x}_3\hat{z}_2 \\
c &= \hat{x}_1\hat{y}_2 - \hat{x}_1\hat{y}_3 - \hat{x}_2\hat{y}_1 + \hat{x}_2\hat{y}_3 + \hat{x}_3\hat{y}_1 - \hat{x}_3\hat{y}_2 \\
d &= -a\hat{x}_1 - b\hat{y}_1 - c\hat{z}_1 = -\hat{x}_1\hat{y}_2\hat{z}_3 + \hat{x}_1\hat{y}_3\hat{z}_2 + \hat{x}_2\hat{y}_1\hat{z}_3 - \hat{x}_2\hat{y}_3\hat{z}_1 - \hat{x}_3\hat{y}_1\hat{z}_2 + \hat{x}_3\hat{y}_2\hat{z}_1
\end{aligned} \tag{4.26}$$

For our calculations, we need also to compute the $-\frac{a}{c}$, $-\frac{b}{c}$ and $-\frac{d}{c}$ values, but we omit them for brevity.

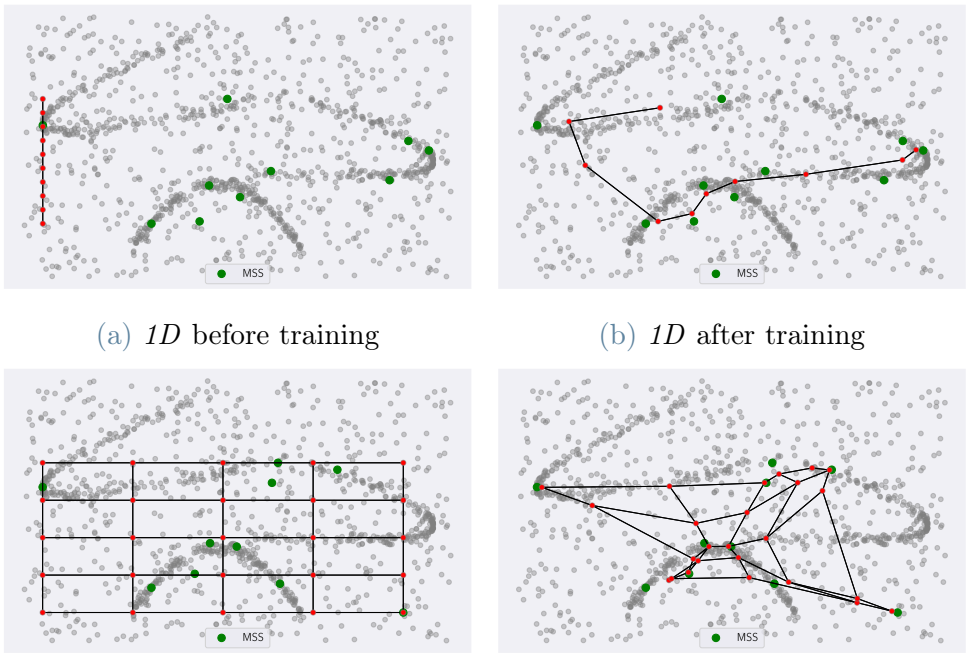
The last step of our demonstration is to substitute the values of the various $\hat{x}$, $\hat{y}$ and $\hat{z}$ with the values of the points in 4.25. We get

$$\begin{aligned}
-\frac{a}{c} &= \frac{-w_{36}w_{47} + w_{37}w_{46}}{w_{35}w_{46} - w_{36}w_{45}} \\
-\frac{b}{c} &= \frac{w_{35}w_{47} - w_{37}w_{45}}{w_{35}w_{46} - w_{36}w_{45}} \\
-\frac{d}{c} &= \frac{b_5w_{36}w_{47} - b_5w_{37}w_{46} - b_6w_{35}w_{47} + b_6w_{37}w_{45} + b_7w_{35}w_{46} - b_7w_{36}w_{45}}{w_{35}w_{46} - w_{36}w_{45}} = \\
&= b_7 - b_5\left(\frac{-w_{36}w_{47} + w_{37}w_{46}}{w_{35}w_{46} - w_{36}w_{45}}\right) - b_6\left(\frac{w_{35}w_{47} - w_{37}w_{45}}{w_{35}w_{46} - w_{36}w_{45}}\right) = \\
&= b_7 + \frac{a}{c}\cdot b_5 + \frac{b}{c}\cdot b_6
\end{aligned} \tag{4.27}$$

This equations show that an auto-encoder structured in this form is able to reconstruct a 3D point belonging to a plane from its 2D latent representation. In the decoder we have

all the ingredients to incorporate the equation of the plane and to generate new points belonging to it.
The entire network works in the training stage learns how to reproduce the input points, until it converges and the equation of the plane is encoded in the weights of the decoder part. At this point, we are able to generate 3D points belonging to the plane starting from 2D points generated e.g. from noise.

4.4.3. Self Organizing Maps



(a) *1D* before training (b) *1D* after training

(c) *2D* before training (d) *2D* after training

Figure 4.6: Two examples of how SOM weights can be arranged. Red dots are weights in the input space, while the black lines are the connections between weights in the topology of the network. In plots 4.6a and 4.6c are shown the initial weights, representing also the arrangement of the weights matrix. Remember that in general, the displacement in the input space is different from the displacement in weight's matrix. In figures 4.6b and 4.6d show the displacement of the weights in the **input space** after the training procedure, that preserve the connections in the weights matrix.

Self organizing maps (SOM) are a special kind of Neural Network used for unsupervised learning, to produce a *low-dimensional* representation of a higher dimensional data set, preserving the topological structure of the data.
This network is composed of two layers: one *input* layer and one *output* layer. The output layer is made of a set of weights $\mathcal{W} = \{\mathbf{w} \mid \mathbf{w} \in \mathbb{R}^m\}$, that are vectors in the same

m-dimensional space as the input vector $\mathbf{p} \in \mathbb{R}^m$. The set of weights can have different shapes, for example can be *one-dimensional* and act like a line, or can be *two-dimensional* with a grid or hexagonal displacement.
The weights of this network lies in the same space as the input, so each weight represents a point in the space of the input vector, that is $\mathbb{R}^m$. At each iteration of the training procedure, the weights are update to become nearer to the input data: using competitive learning, the nearest weight to the input data and its neighbors are updated; after the training, the weights of the SOM will be placed near the most concentrated areas of the input data while preserving the topological structure of the weights. Indeed, weights that are neighbor in the network are also near in the *m-dimensional* space.

What we need to know for our purposes is that this kind of model keeps an embedding of data in the weights, thus each weight $\mathbf{w}$ is a representative of some samples in the dataset that have $\mathbf{w}$ as the nearest weight. Therefore, the displacement of the weights both in the input space and in the topological space can be used to determine the patterns in the input dataset.
In figure 4.6 can be seen two different SOMs trained on a $\mathcal{MSS}$ of size $\rho = 10$. In green there is the dataset, in red the weight's positions in the input space and the black lines are the connections between weights in the topology of the network.
The first one is a SOM with a linear topology that represents a sort of line. After the training, the weights arranged in the input space moving towards denser regions, preserving the topology of the network thanks to the neighboring function 2.16.
In the second one the SOM has a rectangular topology and the same effect can be noticed, allowing weights to fall in dense regions preserving the topology.

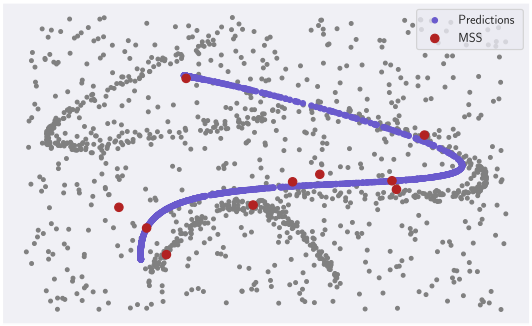
4.5. Preference Embedding

As said, the algorithm trains the pool of model, in which each learner is trained on a portion of the dataset and then the preference is computed. The preference in PIF paper is expressed as a function of the residual between a point and the model; in this thesis, instead, we consider the preference as a function of a quantity called **pseudo-preference** that depends on the model used, but as similarly as in the PIF's residual, therefore if a model express a low pseudo-preference for a point, the corresponding entry in the preference matrix will be higher, since is computed as a decreasing function of the pseudo-preference.

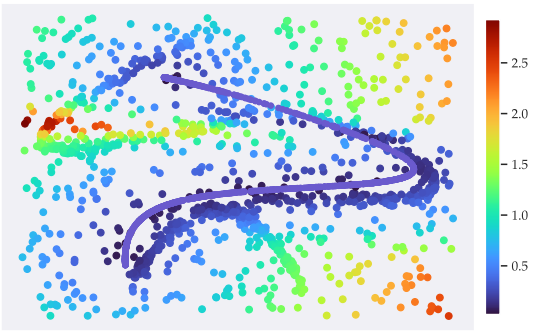
In particular, Auto-Encoders compute the preference as in PIF formulation, thus as the euclidean distance between the point and the prediction for that point. We choose this distance because is the same on which the network is trained, thus it represents the reconstruction error. Therefore, the preference of the Auto-Encoder $\mathcal{G}_j$ expressed for point p_i is the following:

$$\delta_{ij} = (\mathbf{p}_i - \hat{\mathbf{p}}_i)^2 \tag{4.28}$$

where $\hat{\mathbf{p}}_i = \mathcal{G}_j(\mathbf{p}_i)$



(a) Set of predictions for the whole dataset, $\{\mathcal{G}(\mathbf{p})|\mathbf{p} \in \mathcal{D}\}$.



(b) Set of predictions for the whole dataset, $\{\mathcal{G}(\mathbf{p})|\mathbf{p} \in \mathcal{D}\}$.

Figure 4.7: This figure shows the pseudo-preference expressed by an auto-encoder trained on a $\mathcal{MSS}$ of size $\rho = 10$. The higher the value (red), the lower the preference.

Regarding Self Organizing Maps, however, express a meaningful preference from the network is not simple. We tried a lot of approaches, but we figured out that a method exploiting both the positions of the weight, i.e. position in the input space $\in \mathbb{R}^m$ and position in the topology of the weights set $\mathcal{W}$, could lead to better results.
The process is composed of three steps:

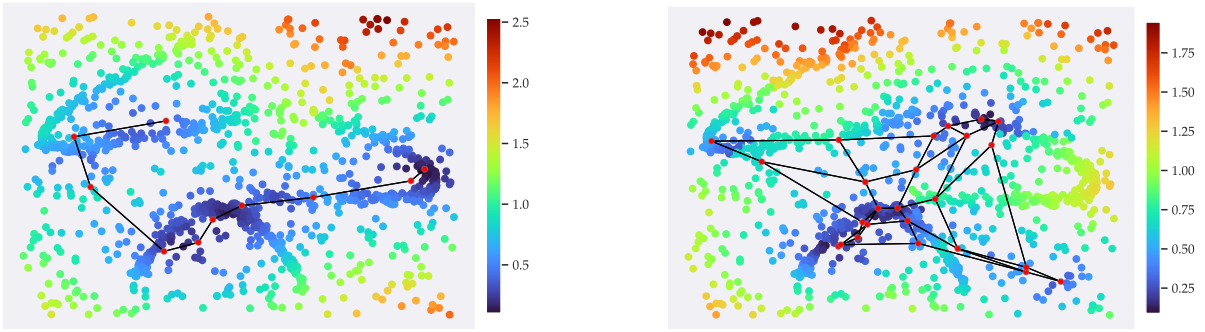
(i) find, in the input space, the weight with minimum distance from $\mathbf{p}$ among all the weights $\mathbf{w} \in \mathcal{W}$;

(ii) find the set of weights $\mathcal{Q} \subset \mathcal{W}$ that are neighbors to $\mathbf{w}$ in the arrangement of $\mathcal{W}$;

(iii) compute the preference for a point as the mean value of the distances between $\mathbf{p}$ and $\mathcal{Q} \cup \mathbf{w}$ in the input space.

This computation takes advantage of the weights' natural arrangement in the network, whereby weights that are close together in the input space are likewise close together in the weights matrix's topology. Anomalies, on the other hand, are not defined by the pattern, therefore few weights will stay in its area. As a result, normal points that describe a

pattern will have numerous weights in its neighborhood because during training weights were distributed there.

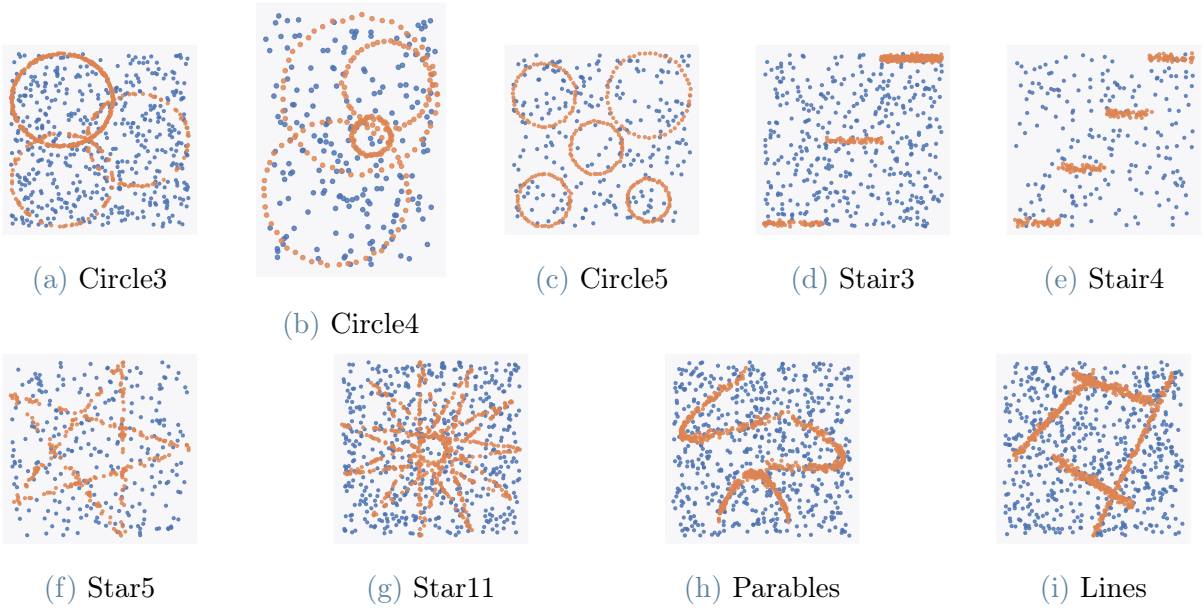


(a) Linear SOM (b) Square SOM

Figure 4.8: This figure shows the pseudo-preference expressed by two SOMs, one with a linear topology (fig 4.8a), the other with a square topology (fig 4.8b). Both have been trained on a $\mathcal{MSS}$ of size $\rho = 10$. The higher the value (red), the lower the preference.

As an example, in figure 4.8 are shown the *pseudo-preferences* expressed by the two SOMs of figure 4.6 on the same dataset. The more an instance is towards the red color, the more it has high *pseudo-preference* - and so smaller preference in the matrix. Therefore, the points belonging to the patterns have a smaller pseudo-preference and are more probably normal points.

5 | Experiments



(a) Circle3 (b) Circle4 (c) Circle5 (d) Stair3 (e) Stair4

(f) Star5 (g) Star11 (h) Parables (i) Lines

Figure 5.1: List of all the 2D datasets used in the tests. Three of them contains circles, five contains lines and one contains parables.

5.1. Data

To validate our work, we experimented the algorithm on the same geometrical datasets used in PIF, that are synthetic 2D datasets describing different types of patterns. Moreover, we have generated other two 2D datasets and three 3D, containing four lines, three parables, one plane, one paraboloid and one sphere, respectively.
The characteristics of the dataset are described in tables 5.1 and 5.2 and the 2D datasets can be visualized in figure 5.1.
In all the datasets the points are contaminated with random Gaussian noise and the anomalies are sampled uniformly in the ranges of the normal points. For each dataset, the number of anomalies is equal to the number of normal points, i.e. $|\mathcal{A}| = |\mathcal{N}|$.
For each dataset, thanks to the available ground truth we were able to estimate the vari-

ance of the patterns. In particular, we estimated the variance of each pattern present in the dataset, e.g. in dataset *star11* there are eleven patterns, and the dataset variance has been estimated as the mean value of the variances of each pattern $i \in [1, |\mathcal{M}|]$.

	$\lvert\mathcal{D}\rvert$	$\lvert\mathcal{A}\rvert$	$\lvert\mathcal{N}\rvert$	$E[\hat{\sigma}_{i=1,...,\lvert\mathcal{M}\rvert}]$	$\lvert\mathcal{M}_1\rvert$, ..., $\lvert\mathcal{M}_k\rvert$
circle3	1000	500	500	0.006	$\lvert\mathcal{M}_1\rvert = 376$, $\lvert\mathcal{M}_i\rvert = 62\ \forall i \in \{2,3\}$
circle4	400	200	200	0.004	$\lvert\mathcal{M}_i\rvert = 50\ \forall i \in \{1,...,4\}$
circle5	500	250	250	0.003	$\lvert\mathcal{M}_i\rvert = 50\ \forall i \in \{1,...,5\}$
stair3	800	400	400	0.003	$\lvert\mathcal{M}_1\rvert = 272$, $\lvert\mathcal{M}_i\rvert = 64\ \forall i \in \{2,3\}$
stair4	400	200	200	0.004	$\lvert\mathcal{M}_i\rvert = 50\ \forall i \in \{1,...,4\}$
star5	500	250	250	0.009	$\lvert\mathcal{M}_i\rvert = 50\ \forall i \in \{1,...,5\}$
star11	1100	550	550	0.005	$\lvert\mathcal{M}_i\rvert = 50\ \forall i \in \{1,...,11\}$
parables	600	300	300	0.02	$\lvert\mathcal{M}_i\rvert = 100\ \forall i \in \{1,...,3\}$
lines	800	400	400	0.02	$\lvert\mathcal{M}_i\rvert = 100\ \forall i \in \{1,...,4\}$

Table 5.1: Synthetic (2D) datasets settings

	$\lvert\mathcal{D}\rvert$	$\lvert\mathcal{A}\rvert$	$\lvert\mathcal{N}\rvert$	$\hat{\sigma}$	$\lvert\mathcal{M}_1\rvert$, ..., $\lvert\mathcal{M}_k\rvert$
plane	1000	500	500	0.05	$\lvert\mathcal{M}\rvert = 500$
paraboloid	1000	500	0.05	500	$\lvert\mathcal{M}\rvert = 500$
sphere	1000	500	500	0.05	$\lvert\mathcal{M}\rvert = 500$

Table 5.2: Synthetic (3D) datasets settings

5.2. Methodology

5.2.1. Hyper-parameters

To be able to study the behavior of the algorithm in different scenarios, some hyper-parameters changes test by test while others remain stable. In the latter case we find the number of training epochs, the learning rate, the number of models used and the parameters of the Voronoi Forest built in the last phase of the algorithm; this ones are kept the same as suggested in PIF paper.
In contrast, what changes depending on the experiment are the Minimum Sample Set Size ρ, determining the cardinality of the sub-set on which the models are trained, the architecture of the networks and most important the inlier threshold τ, used for the preference embedding. Possible values of the parameters are listed in table 5.3

Hyper-parameter	Values
AE architecture 1 (AE_1)	(2, 4, 1, 4, 2)
AE architecture 2 (AE_2)	(2, 8, 4, 1, 4, 8, 2)
AE architecture 3 (AE_3)	(3, 5, 2, 5, 3)
AE architecture 4 (AE_4)	(3, 9, 5, 2, 5, 9, 3)
SOM architecture 1 (SOM_1)	(1, 10)
SOM architecture 2 (SOM_2)	(5, 5)
ρ	[2, 5, 10, 20, 30, 100]
τ	$[1, 4, 7, 10, 25, 50, 100] \cdot \sigma$
training epochs	500
n.o models	2000
learning rate	0.01
activation function	$\tanh(\cdot)$

Table 5.3: Auto-Encoders architectures list the number of neurons in each layer. SOM architectures list the number of neurons as a tuple (num of rows, num of cols).

We choose to keep only some of the hyper-parameters variables, because are the ones that have a higher impact on the performances of the dataset. First, the network's architecture determines how much the network is big and how it is arranged in the case of the SOM. Bigger networks have bigger potential because could learn more, but the number of data on which is trained (ρ) is the same; thus, its performances are not necessarily better, because the lack of data and the higher number of parameters to learn could not bring the network to convergence.

As said, ρ determines the size of the sub-set on which networks are trained. Usually, Neural Networks need a high number of samples for training, because they have a high number of parameters compared to other methods; we decided to employ relatively small neural networks in order to have fewer parameters and a better training convergence since ρ is relatively small. This choices leads to have smaller computational complexity, and follow the RANSAC's principle.
Moreover, ρ controls also the probability of sampling a *pure* $\mathcal{MSS}$, i.e. a $\mathcal{MSS}$ that contains only normal points, without anomalies. Since we employed a randomized sampling, the probability of each point to be selected is $\frac{1}{N}$; therefore, with $\rho = 1$ we have $\frac{1}{2}$

probability of getting a pure $\mathcal{MSS}$. With $\rho = 2$, we have four possibilities of sampling, only one of which comprises both normal samples. Continuing, for each $\rho \in [1, N]$, we can say that the probability of getting at least one pure $\mathcal{MSS}$ is $\frac{1}{2^\rho}$. Therefore, with a higher ρ we increase the number of samples on which each network is trained, but we also reduce the possibility of getting pure models, i.e. those models trained on pure $\mathcal{MSS}$.

Another important hyper-parameter is τ, the *inlier threshold*. This threshold, according to equation 3.8, determines if a point is considered normal, with its degree of "normality", or an anomaly, with a score of zero, for the current model. Therefore, a near-zero threshold would consider normal points only those that have a very low residual, while a high threshold allows more points to be considered normal.
This threshold influences the values in the preference matrix, thus changing the functioning of the Isolation Voronoi Forest. Finding the good trade-off between the inlier threshold and accuracy of the model is not easy and a grid-search is required to reach the best performances depending on our requirements.

5.2.2. Testing Procedure

The testing procedure has been structured hierarchically and we tested each possible combination comprising the dataset to use, the architecture, the threshold τ and the size of the sub-sampling set ρ.
Therefore, the testing routine is structured as follows, where *Neural-PIF* is detailed in algorithm 4.1:

Algorithm 5.1 *Testing procedure*

Input: $\boldsymbol{\mathcal{D}}$ - the set of dataset, $\mathcal{G}$ - model to use, $\boldsymbol{\tau}$ - all possible values of the normal threshold, $\boldsymbol{\rho}$ - all possible values of the sub-sampling size

```
for all 𝒟 ∈ 𝓓 do
    for all τ ∈ τ do
        for all ρ ∈ ρ do
            scores = Neural-PIF(𝒟, τ, ρ, 𝒢)
            AUC = computeROCauc(scores)
        end for
    end for
end for
```

After running all the experiments, we ended up with the ROC's AUC reached by each combination of parameters. In appendix A there are the plots showing for each dataset and for each model how the AUC varies depending on τ and ρ.

5.3. Results

5.3.1. Two-dimensional data

In this section we will analyze the results obtained with each model, considering different combinations of the aforementioned hyper-parameters.
In table 5.4 there are the results reached by each of the presented model, plus two State Of Art algorithms' results. Results of LOF and iFor have been computed running one hundred experiments for each dataset and taking then the mean value, while PIF's results are taken from its paper. In appendix A in lists 2 and 3, we show the code to reproduce the above experiments.
Results of the models presented in this thesis, instead, are taken as the maximum value reached for that specific architecture for that dataset; in other words, we took the maximum value independently of τ and ρ from the plots in A.

Dataset	LOF	iFor	AE_1	AE_2	SOM_1	SOM_2	PIF
circle3	0.691	0.708	0.865	0.872	0.869	0.874	**0.930**
circle4	0.610	0.633	0.656	0.689	0.756	0.726	**0.897**
circle5	0.577	0.567	0.660	0.688	0.711	0.725	**0.780**
stair3	0.717	0.940	0.968	0.972	0.975	**0.977**	0.971
stair4	0.809	0.889	**0.957**	0.950	0.955	0.956	0.952
star5	0.753	0.731	0.806	0.829	0.831	0.850	**0.910**
star11	0.665	0.742	0.773	0.775	0.760	0.768	**0.796**
Mean_{PIF}	0.689	0.744	0.812	0.825	0.837	0.839	**0.891**
parables	0.838	0.813	0.907	**0.920**	0.911	0.918	-
lines	0.809	0.818	0.914	0.913	0.914	**0.916**	-
Mean_{TOT}	0.719	0.760	0.834	0.845	0.854	**0.857**	-

Table 5.4: Results on synthetic 2D datasets. Bold elements represent the higher value on the row. Underlined elements instead represents the maximum value on the row, but without considering PIF results.

We compared our algorithm, in four different variants, with LOF and iFor; PIF results are shown only as a benchmark and not as a comparision, because it would not be fair. Indeed, PIF has higher knowledge with respect our algorithm, because it assumes to know the correct pattern to search. Therefore, our algorithm lies in the middle between PIF and other density-based or model-based algorithms, because we exploit the preference trick and the structure of the dataset, but we are not as informed as in PIF.

Therefore, we will make the comparisons by comparing only those algorithms that have the same dataset information as ours, i.e., none.

First of all, we will compare our algorithm in general with the two state-of-art competitors and then we will compare the different versions of our algorithm.

As can be noted from the table, LOF and iFor have much lower performances on this datasets, loosing up to 10% over the mean AUC values. This is due to the fact that this algorithms do not try to understand how data is composed and arranged, i.e. learn the pattern, but rather computes the anomaly scores depending on the density of the point or considering how much the point is easy to isolate.
Since in structured normal data dataset the density is not informative, this algorithms are not able to reach our algorithm's performances.

Considering instead our algorithm, we can notice that using SOMs as learners we reach higher performances. Both the models, AEs and SOMs, are able to learn a pattern and understand useful information from data, but the difference could be due to how the models learn and computes the preference.
Auto-Encoders learn a hidden representation of the data and reconstruct original input vector, therefore are able to generate *m-dimensional* vectors in all $\mathbb{R}^m$ and if we input an infinite number of vectors belonging to a continuous function to the auto-encoder, it will output a continuous function. Therefore, it is as if the auto-encoder learns a *continuous* representation of the $\mathcal{MSS}$ on which it is trained.

On the contrary, SOMs during training moves a finite number of weights in the input space and at inference time it returns the weight closer to the input vector. Therefore, if we input to the SOM an infinite number of points belonging to a pattern, as we could do with the auto-encoder, as output we will get only a finite number of points defined by the weights of the network.
Therefore, it is as if the SOM learns a *discrete* representation of the $\mathcal{MSS}$ on which it is trained, because the possible outputs of the network are defined only where the weights are.

Analyzing this differences in the type of learning and inference performed by the two networks, it is clear why there is such difference in the results in the table. Since the datasets mostly contain several patterns and not only one, the union of all this patterns could be thinked as a unique non-continuous pattern. In this case, therefore, it is better to

learn a discrete representation instead of a continuous one. Therefore, theoretically auto-encoders should perform better when the manifolds are continuous and smooth, while SOMs perform better when the manifolds are discontinuous and broken in pieces.

5.3.2. Three-dimensional data

In table 5.5 are shown results for the same algorithms as in the *two-dimensional* case, applied to synthetic datasets comprising a plane, a paraboloid and a sphere. In this case the results of PIF are not given, because in the paper it has not been tested on this datasets.

Dataset	LOF	iFor	AE_3	AE_4	SOM_1	SOM_2
plane	0.827	0.725	**0.940**	0.936	0.897	0.899
paraboloid	0.690	0.591	0.878	**0.906**	0.845	0.845
sphere	0.651	0.783	0.828	**0.858**	0.789	0.771
Mean	0.722	0.699	0.882	**0.900**	0.844	0.838

Table 5.5: Results on synthetic 3D datasets. Results in bold are the maximum value along the row.

In this situations, the comparison with the state-of-art algorithms is the same as in the *two-dimensional* case, with the state-of-art algorithms loosing up to 20% of ROC AUC. Again, this discrepancy can be explained by the fact that in the dataset all regions have more or less the same density, since anomalies have been sampled from a uniform distribution. Therefore, also in this case the density is not informative and the algorithms perform poorly.

This results also confirm another observation made for the *two-dimensional* case, that is the fact that SOMs prefer discontinuous and non-smooth manifolds, while auto-encoders prefer continuous and smooth manifolds. Since in each of this dataset there was only one continuous and smooth manifold, the auto-encoder performed better.
Differences between the two versions of the auto-encoders could be explained by the increased complexity of the network, supported also by the fact that as shown in images in sections A.1.3 and A.1.4, larger $\mathcal{MSS}$ brings better results, meaning that the networks can use more data to train on.

6 | Conclusions and future developments

In this thesis we explored the problem of Anomaly Detection in the context of structured normal data and high number of anomalies, representing half of the dataset. We have studied the possibility of implementing general learners in the preference framework presented in PIF [16], evidencing the needs of models learned autonomously from data, instead of forcing a precise pattern as in previous works.

We have explored two kinds of learners, the Auto-Encoders and the Self-Organizing Maps, showing their ability of learning useful representations of the dataset and the ability to discover patterns without any prior knowledge. Moreover, we have investigated the possibility of building an ensemble of such models and bring them in the preference framework. Thanks to the locally learned representations, this method shows good results if coupled with the preference framework.

This thesis has been only the first step towards the investigation of general pattern learners, opening the doors for researches in patter-recognition problems approached with the preference framework. Each step of the preference framework is a component that can be changed and depending on the problem, we can use different strategies.
For example, in T-Linkage [19] the preference embedding is used in combination with hierarchical clustering to perform multi-model fitting; in this algorithm, the difference with respect to PIF was only in the last phase, in which instead of fitting a Voronoi Isolation Forest on the preference matrix, it applies hierarchical clustering.
Also in T-Linkage there was the limiting assumption as in PIF, forcing the patterns to be discovered to be as assumed; therefore, it is clear that an approach like the one presented in this thesis can be beneficial for T-Linkage.

Unfortunately it is not simple, thus an extensive research in this field is required, in order to discover new relationships between hyper-parameters or discover new models that

could be useful.
For example, it might be interesting to apply our framework in Computer Vision Anomaly Detection, employing Convolutional Neural Networks for pattern-recognition on non-tabular data like images.

Thanks to the modularity of the preference framework, implementing it in new tasks and new contexts should be quite straightforward, because it requires only a class of pattern learners that are able to express preferences for instances of the dataset. This instances potentially could be images, points of a point-cloud or everything else; if there is the possibility of learning the patterns in an unsupervised manner, then the preference framework can be implemented.

Bibliography

[1] Network Anomaly Detection and Network Behavior Analysis | Flowmon. URL https://www.flowmon.com/en/solutions/security-operations/network-behavior-analysis-anomaly-detection.

[2] L'equazione della retta passante tra due punti. URL https://www.andreaminini.org/matematica/spazio-vettoriale/retta-passante-per-due-punti.

[3] Equazione cartesiana del piano, . URL https://www.youmath.it/lezioni/algebra-lineare/geometria-dello-spazio/682-equazione-del-piano.html.

[4] . URL https://tutorial.math.lamar.edu/Classes/CalcIII/GradientVectorTangentPlane.aspx.

[5] Sympy. URL https://www.sympy.org/en/index.html.

[6] 1.8: The Tangent Line Approximation, June 2017. URL https://math.libretexts.org/Under_Construction/Purgatory/Book%3A_Active_Calculus_(Boelkins_et_al.)/01%3A_Understanding_the_Derivative/1.08%3A_The_Tangent_Line_Approximation.

[7] N. Abe, B. Zadrozny, and J. Langford. Outlier detection by active learning. volume 2006, pages 504–509, 01 2006. doi: 10.1145/1150402.1150459.

[8] D. Bank, N. Koenigstein, and R. Giryes. Autoencoders, Apr. 2021. URL http://arxiv.org/abs/2003.05991. arXiv:2003.05991 [cs, stat].

[9] M. M. Breunig, H.-P. Kriegel, R. T. Ng, and J. Sander. LOF: Identifying Density-Based Local Outliers. page 12.

[10] N. Chawla and W. Wang, editors. *Proceedings of the 2017 SIAM International Conference on Data Mining*. Society for Industrial and Applied Mathematics, Philadelphia, PA, June 2017. ISBN 978-1-61197-497-3. doi: 10.1137/1.9781611974973. URL https://epubs.siam.org/doi/book/10.1137/1.9781611974973.

[11] M. A. Fischler and R. C. Bolles. Random sample consensus: A paradigm for model

fitting with applications to image analysis and automated cartography. *Commun. ACM*, 24(6):381–395, jun 1981. ISSN 0001-0782. doi: 10.1145/358669.358692. URL https://doi.org/10.1145/358669.358692.

[12] G. Guo, H. Wang, D. Bell, and Y. Bi. Knn model-based approach in classification. 08 2004.

[13] Z. He, X. Xu, and S. Deng. Discovering Cluster Based Local Outliers. *Pattern Recognition Letters*, 2003:9–10, 2003.

[14] R. Khandelwal. Anomaly detection using autoencoders, Jan 2021. URL https://towardsdatascience.com/anomaly-detection-using-autoencoders-5b032178a1ea.

[15] T. Kohonen. The self-organizing map. *PROCEEDINGS OF THE IEEE*, 78(9):17, 1990.

[16] F. Leveni, L. Magri, G. Boracchi, and C. Alippi. PIF: Anomaly detection via preference embedding. pages 8077–8084, Jan. 2021. doi: 10.1109/ICPR48806.2021.9412658. URL https://ieeexplore.ieee.org/document/9412658/.

[17] F. T. Liu, K. M. Ting, and Z.-H. Zhou. Isolation Forest. In *2008 Eighth IEEE International Conference on Data Mining*, pages 413–422, Pisa, Italy, Dec. 2008. IEEE. ISBN 978-0-7695-3502-9. doi: 10.1109/ICDM.2008.17. URL http://ieeexplore.ieee.org/document/4781136/.

[18] F. T. Liu, K. M. Ting, and Z.-H. Zhou. Isolation-Based Anomaly Detection. *ACM Transactions on Knowledge Discovery from Data*, 6(1):1–39, Mar. 2012. ISSN 1556-4681, 1556-472X. doi: 10.1145/2133360.2133363. URL https://dl.acm.org/doi/10.1145/2133360.2133363.

[19] L. Magri and A. Fusiello. T-linkage: A continuous relaxation of j-linkage for multi-model fitting. In *2014 IEEE Conference on Computer Vision and Pattern Recognition*, pages 3954–3961, 2014. doi: 10.1109/CVPR.2014.505.

[20] A. G. Martín, M. Beltrán, A. Fernández-Isabel, and I. Martín de Diego. An approach to detect user behaviour anomalies within identity federations. *Computers & Security*, 108:102356, Sept. 2021. ISSN 0167-4048. doi: 10.1016/j.cose.2021.102356. URL https://www.sciencedirect.com/science/article/pii/S0167404821001802.

[21] A. Maćkiewicz and W. Ratajczak. Principal components analysis (pca). *Computers and Geosciences*, 19(3):303–342, 1993. ISSN 0098-3004. doi: https://doi.org/

10.1016/0098-3004(93)90090-R. URL https://www.sciencedirect.com/science/article/pii/009830049390090R.

[22] A. Mensi and M. Bicego. A Novel Anomaly Score for Isolation Forests. In *Image Analysis and Processing – ICIAP 2019: 20th International Conference, Trento, Italy, September 9–13, 2019, Proceedings, Part I*, pages 152–163, Berlin, Heidelberg, Sept. 2019. Springer-Verlag. ISBN 978-3-030-30641-0. doi: 10.1007/978-3-030-30642-7_14. URL https://doi.org/10.1007/978-3-030-30642-7_14.

[23] A. A. Patel. Hands-On Unsupervised Learning Using Python. page 170.

[24] T. Pourhabibi, K.-L. Ong, B. H. Kam, and Y. L. Boo. Fraud detection: A systematic literature review of graph-based anomaly detection approaches. *Decision Support Systems*, 133:113303, June 2020. ISSN 0167-9236. doi: 10.1016/j.dss.2020.113303. URL https://www.sciencedirect.com/science/article/pii/S0167923620300580.

[25] M. Riazi, O. Zaïane, T. Takeuchi, A. Maltais, J. Günther, and M. Lipsett. *Detecting the Onset of Machine Failure Using Anomaly Detection Methods*, pages 3–12. 08 2019. ISBN 978-3-030-27519-8. doi: 10.1007/978-3-030-27520-4_1.

[26] D. J. Rogers and T. T. Tanimoto. A computer program for classifying plants. *Science*, 132(3434):1115–1118, 1960. doi: 10.1126/science.132.3434.1115. URL https://www.science.org/doi/abs/10.1126/science.132.3434.1115.

[27] T. Rowland. Manifold, 2022. URL https://mathworld.wolfram.com/Manifold.html.

[28] B. Schölkopf, R. C. Williamson, A. J. Smola, J. Shawe-Taylor, and J. C. Platt. Support Vector Method for Novelty Detection. page 7.

[29] T. Sipes, S. Jiang, K. Moore, N. Li, H. Karimabadi, and J. R. Barr. Anomaly Detection in Healthcare: Detecting Erroneous Treatment Plans in Time Series Radiotherapy Data. *International Journal of Semantic Computing*, 08(03):257–278, Sept. 2014. ISSN 1793-351X. doi: 10.1142/S1793351X1440008X. URL https://www.worldscientific.com/doi/10.1142/S1793351X1440008X. Publisher: World Scientific Publishing Co.

[30] Y. Tagawa, R. Maskeliūnas, and R. Damaševičius. Acoustic Anomaly Detection of Mechanical Failures in Noisy Real-Life Factory Environments. *Electronics*, 10 (19):2329, Sept. 2021. ISSN 2079-9292. doi: 10.3390/electronics10192329. URL https://www.mdpi.com/2079-9292/10/19/2329.

[31] R. Toldo and A. Fusiello. Robust multiple structures estimation with j-linkage. volume 5302, pages 537–547, 10 2008. doi: 10.1007/978-3-540-88682-2_41.

[32] K. P. A. v. Wanrooij. Patient Careflow Discovery. July 2018. URL https://studenttheses.uu.nl/handle/20.500.12932/29558. Accepted: 2018-07-19T17:04:31Z.

A | Appendix A

A.1. Code - Plane calculations

In this appendix we will show the code used to compute the coefficients at section 4.4.2. This code uses the SymPy [5] package available for Python, that allows for symbolic calculations.

```
import sympy as sy

# function to compute coefficients of the plane
def compute_a_b_c_d(x_1, x_2, x_3, y_1, y_2, y_3, z_1, z_2, z_3):
    M = sy.Matrix([[y_2-y_1, z_2-z_1], [y_3-y_1, z_3-z_1]])
    a = sy.simplify(M.det())

    M = sy.Matrix([[x_2-x_1, z_2-z_1], [x_3-x_1, z_3-z_1]])
    b = sy.simplify(-M.det())

    M = sy.Matrix([[x_2-x_1, y_2-y_1], [x_3-x_1, y_3-y_1]])
    c = sy.simplify(M.det())

    d = sy.simplify(-a*x_1 - b*y_1 - c*z_1)
    return a, b, c, d

# initialize printing system
sy.init_printing()
# define symbols for calculations
syms = w_35, w_36, w_45, w_46, \
        w_37, w_47, h_3_1, h_3_2, \
        h_3_3, h_4_1, h_4_2, h_4_3, \
        b_5, b_6, b_7, x_1, \
```

```
24          x_2, x_3, y_1, y_2, \
25          y_3, z_1, z_2, z_3 = sy.symbols(
26                                      "w_35 w_36 w_45 w_46 " +
27                                      "w_37 w_47 h_3_1 h_3_2 " +
28                                      "h_3_3 h_4_1 h_4_2 h_4_3 " +
29                                      "b_5 b_6 b_7 x_1 " +
30                                      "x_2 x_3 y_1 y_2 " +
31                                      "y_3 z_1 z_2 z_3 ")
32
33  a, b, c, d = compute_a_b_c_d(x_1, x_2, x_3, y_1, y_2, y_3, z_1, z_2, z_3)
34
35  print("a: \n" + sy.latex(a) + "\n")
36  print("b: \n" + sy.latex(b) + "\n")
37  print("c: \n" + sy.latex(c) + "\n")
38  print("d: \n" + sy.latex(d) + "\n")
39
40  # define coordinates from network's output
41  x_1 = w_35*h_3_1+w_45*h_4_1+b_5
42  x_2 = w_35*h_3_2+w_45*h_4_2+b_5
43  x_3 = w_35*h_3_3+w_45*h_4_3+b_5
44
45  y_1 = w_36*h_3_1+w_46*h_4_1+b_6
46  y_2 = w_36*h_3_2+w_46*h_4_2+b_6
47  y_3 = w_36*h_3_3+w_46*h_4_3+b_6
48
49  z_1 = w_37*h_3_1+w_47*h_4_1+b_7
50  z_2 = w_37*h_3_2+w_47*h_4_2+b_7
51  z_3 = w_37*h_3_3+w_47*h_4_3+b_7
52
53  a, b, c, d = compute_a_b_c_d(x_1, x_2, x_3, y_1, y_2, y_3, z_1, z_2, z_3)
54
55  print("a: \n" + sy.latex(a) + "\n")
56  print("b: \n" + sy.latex(b) + "\n")
57  print("c: \n" + sy.latex(c) + "\n")
58  print("d: \n" + sy.latex(d) + "\n")
59
60  print("-(a/c): \n" + sy.latex(sy.simplify(-(a/c), syms=syms)) + "\n")
61  print("-(b/c): \n" + sy.latex(sy.simplify(-(b/c), syms=syms)) + "\n")
```

```
print("-(d/c): \n" + sy.latex(sy.simplify(-(d/c), syms=syms)) + "\n")
```

Listing 1: Code reproducing results obtained demonstrating that the auto-encoder is able to learn the representation of the plane.

A.2. Code - LOF experiments

```
from sklearn.neighbors import LocalOutlierFactor
from sklearn.metrics import roc_curve, auc

print("LOF:\n")

lof_aucs = []
for ds_name in ["circle3",
                "circle4",
                "circle5",
                "stair3",
                "stair4",
                "star5",
                "star11",
                "circles_parable3",
                "lines_rects4",
                "plane",
                "sphere",
                "paraboloid"]:
    ds, gt = load_dataset_by_name(
        base_path="PATH/TO/FOLDER", name=ds_name, with_outliers=True)
    aucs = []
    for _ in range(100):
        lof = LocalOutlierFactor().fit(ds)
        scores_lof = -lof.negative_outlier_factor_
        fpr, tpr, thr = roc_curve(gt, scores_lof, pos_label=0)
        aucs.append(auc(fpr, tpr))
    lof_aucs.append(np.mean(aucs))
    print(f"{ds_name:<20} {np.mean(aucs)}")
```

Listing 2: Code reproducing results obtained in the table 5.4 and 5.5.

A.3. Code - iFor experiments

```
from sklearn.ensemble import IsolationForest as iFor
from sklearn.metrics import roc_curve, auc

print("iFor:\n")

ifor_aucs = []
for ds_name in ["circle3",
                "circle4",
                "circle5",
                "stair3",
                "stair4",
                "star5",
                "star11",
                "circles_parable3",
                "lines_rects4",
                "plane",
                "sphere",
                "paraboloid"]:
    ds, gt = load_dataset_by_name(
        base_path="PATH/TO/FOLDER", name=ds_name, with_outliers=True)
    aucs = []
    for i in range(100):
        ifor = iFor().fit(ds)
        scores_ifor = -ifor.score_samples(ds)
        fpr, tpr, thr = roc_curve(gt, scores_ifor, pos_label=0)
        aucs.append(auc(fpr, tpr))
    ifor_aucs.append(np.mean(aucs))
    print(f"{ds_name:<20} {np.mean(aucs)}")
```

Listing 3: Code reproducing results obtained in the table 5.4 and 5.5.

A | Appendix B

A.1. Results - Auto-Encoder

A.1.1. Architecture 1 - 2D

Figure A.1: AE_1 AUCs on circle3.

Figure A.2: AE_1 AUCs on circle4.

Figure A.3: AE_1 AUCs on circle5.

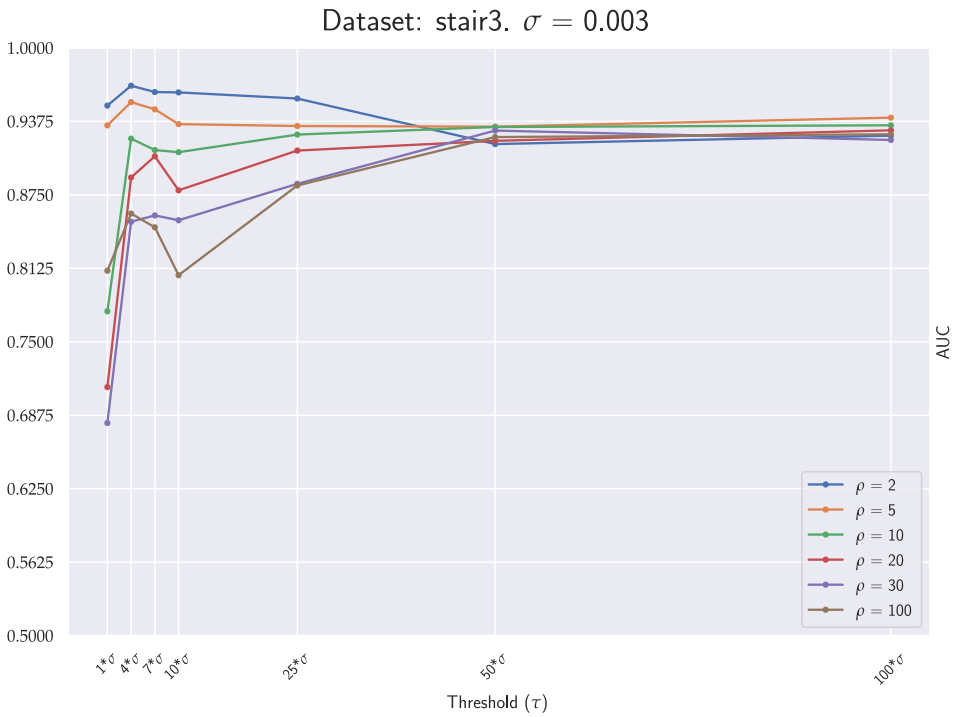


Figure A.4: AE_1 AUCs on stair3.

Figure A.5: AE_1 AUCs on stair4.

Figure A.6: AE_1 AUCs on star5.

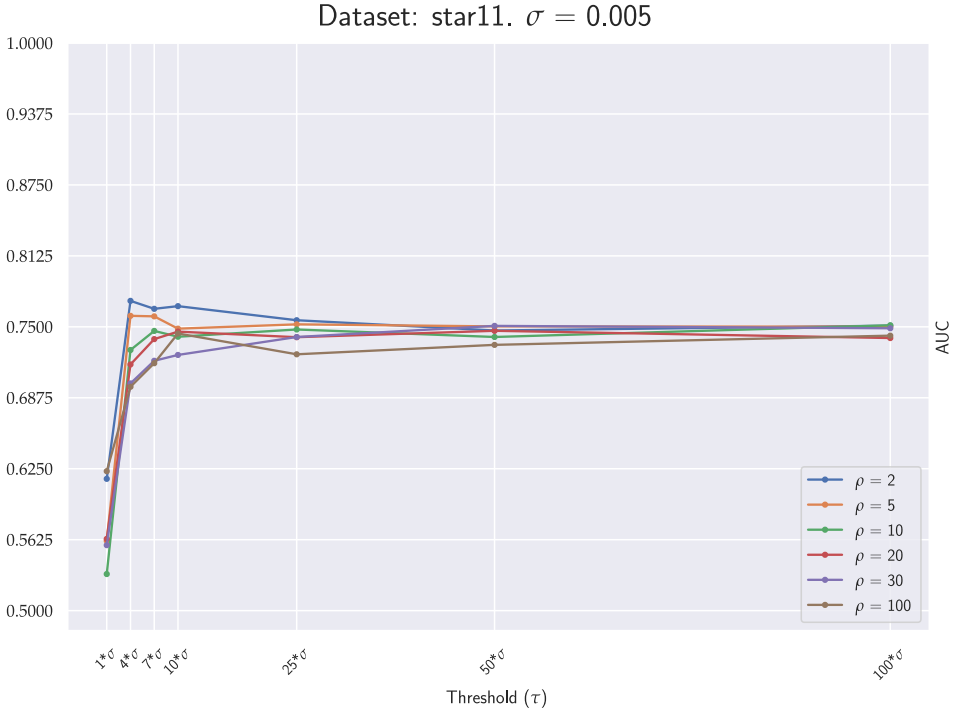


Figure A.7: AE_1 AUCs on star11.

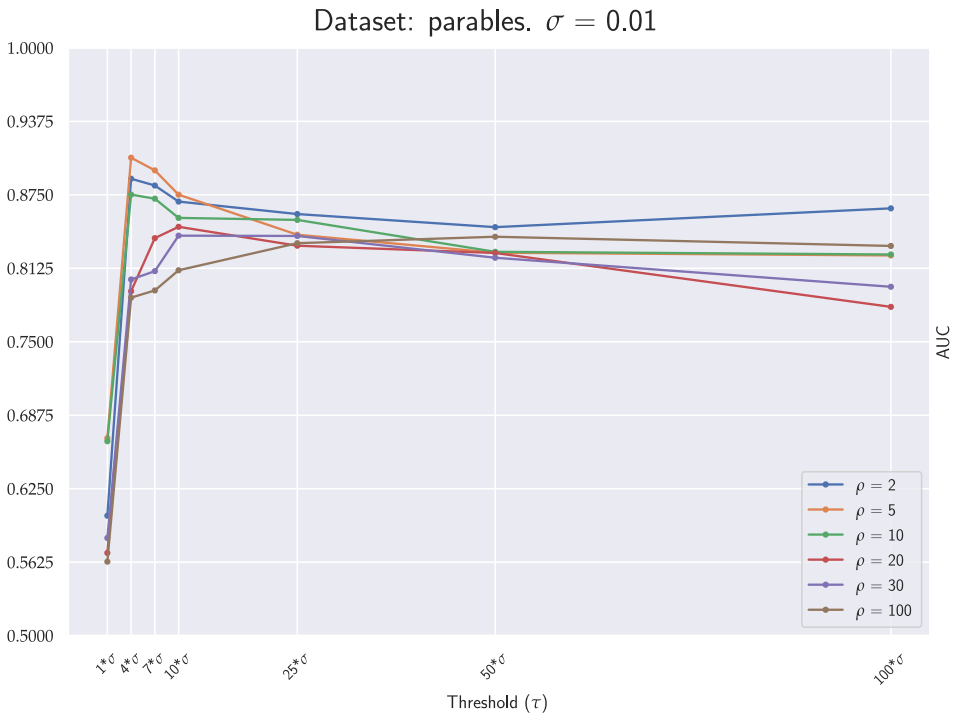


Figure A.8: AE_1 AUCs on parables.

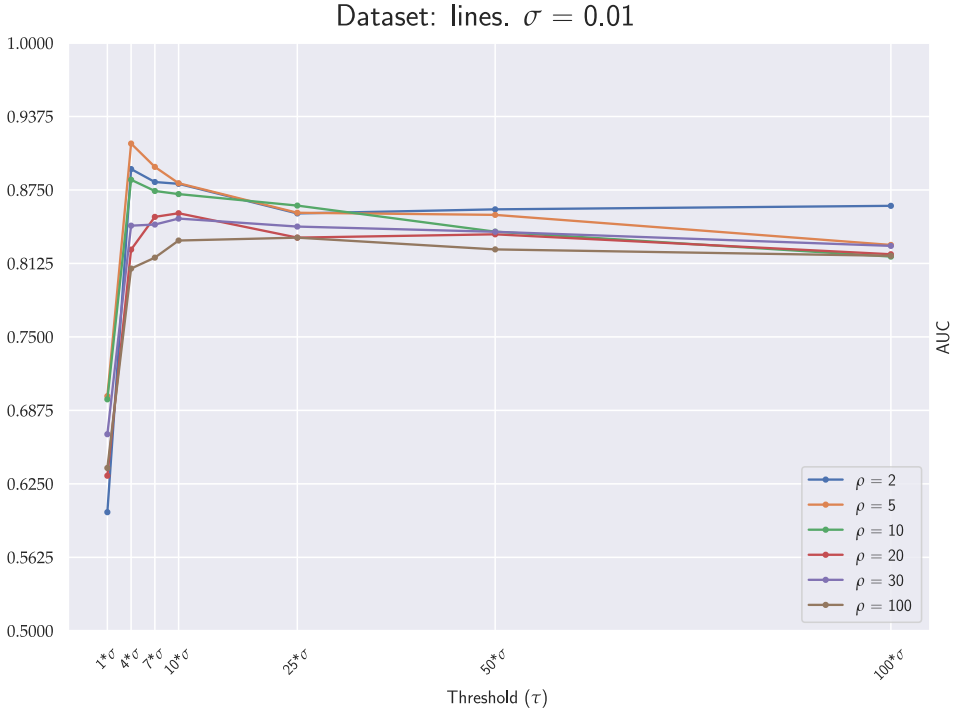


Figure A.9: AE_1 AUCs on lines.

A.1.2. Architecture 2 - 2D

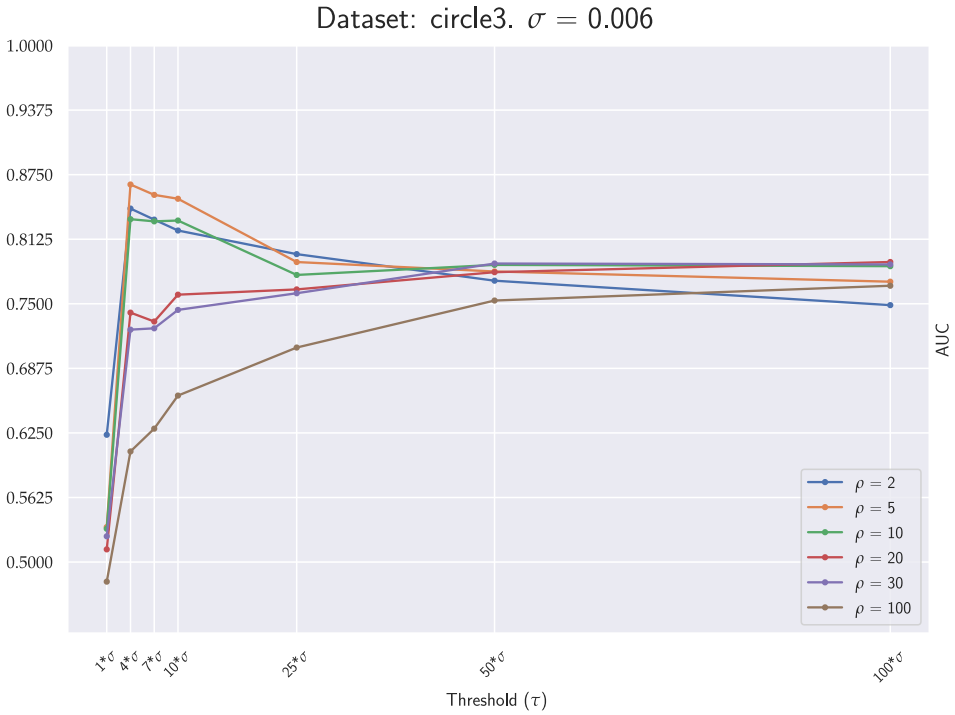


Figure A.10: AE_2 AUCs on circle3.

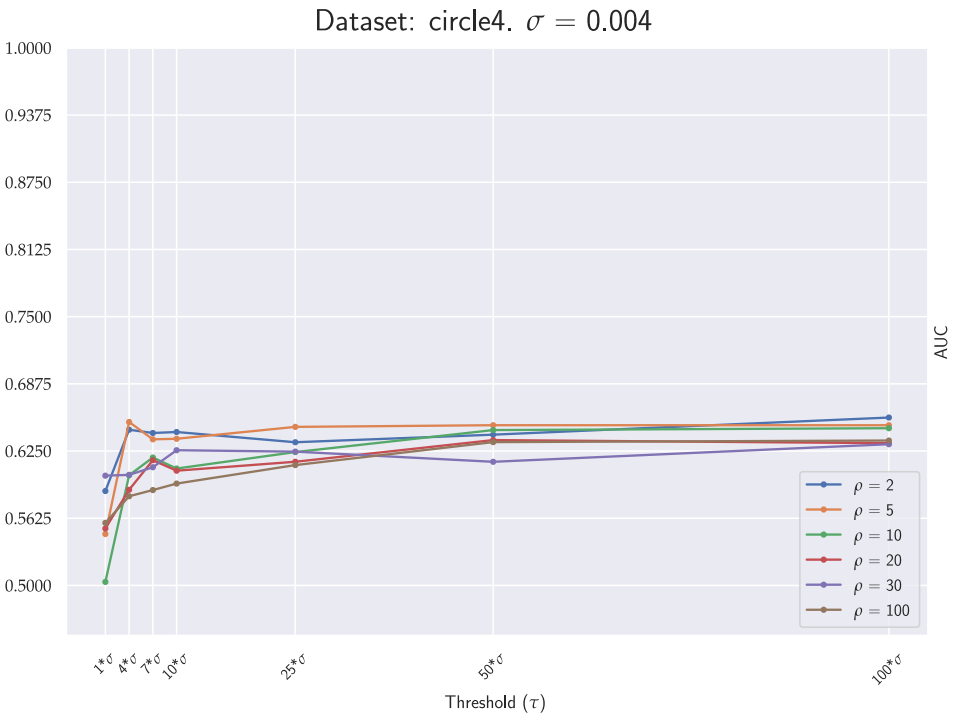


Figure A.11: AE_2 AUCs on circle4.

Figure A.12: AE_2 AUCs on circle5.

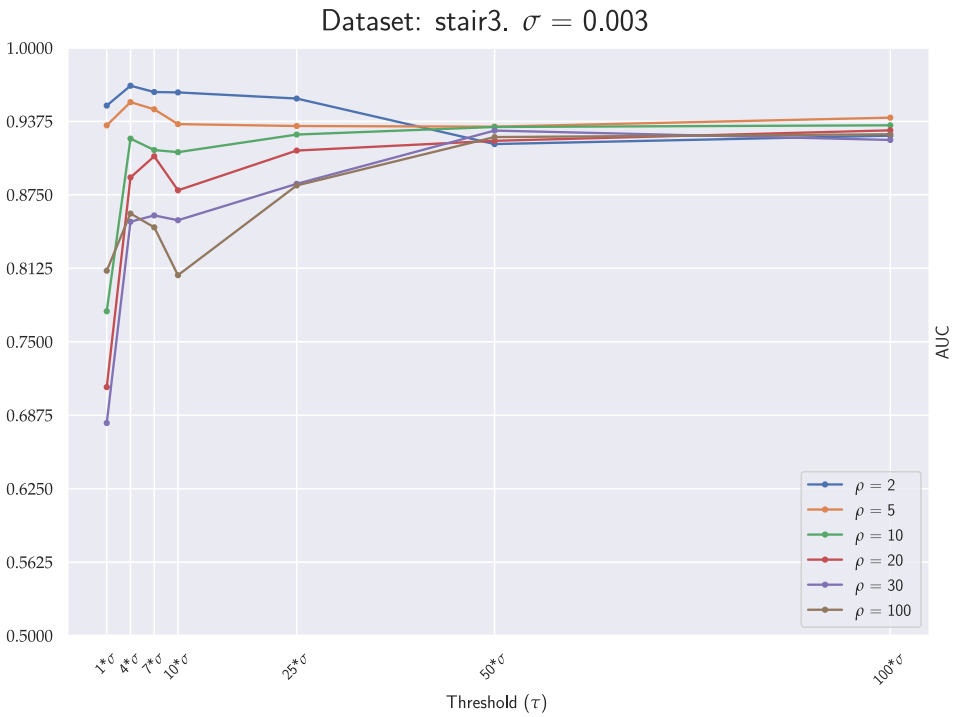


Figure A.13: AE_2 AUCs on stair3.

Figure A.14: AE_2 AUCs on stair4.

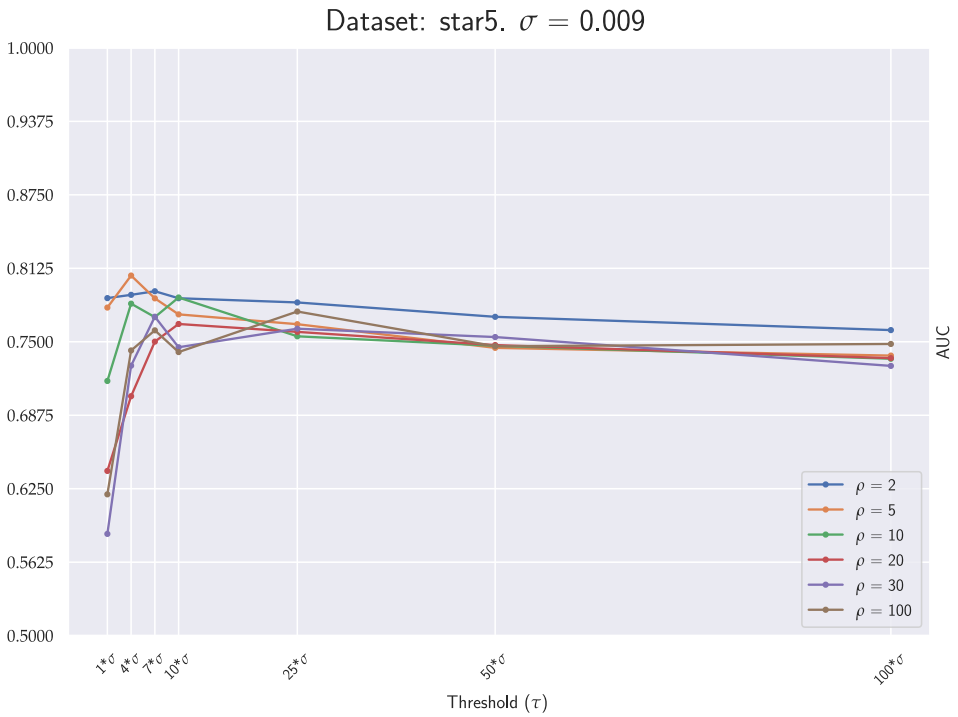


Figure A.15: AE_2 AUCs on star5.

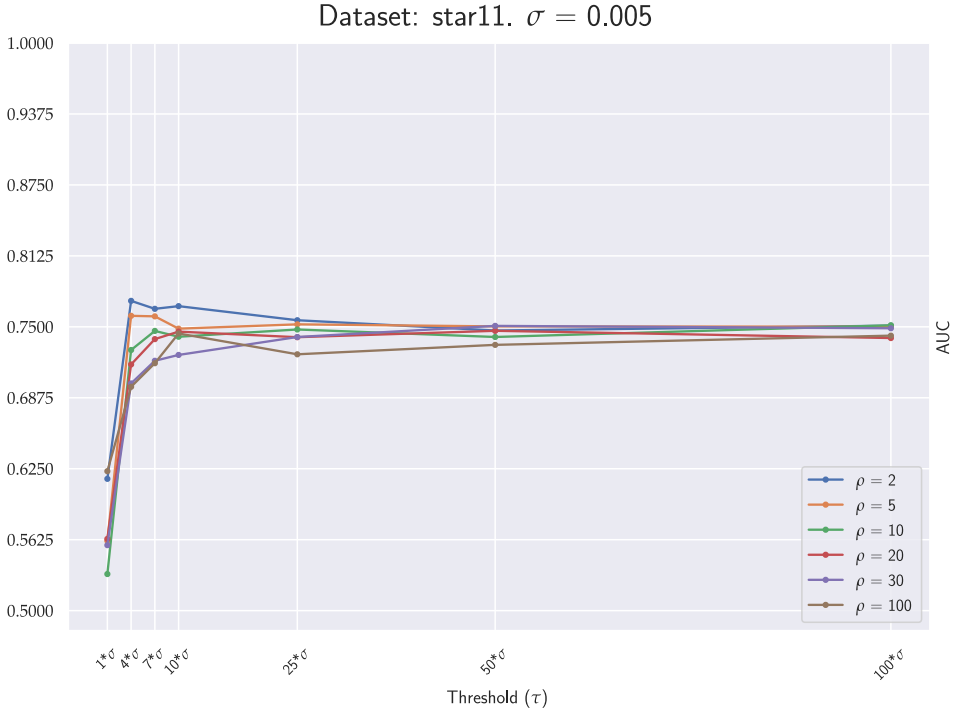


Figure A.16: AE_2 AUCs on star11.

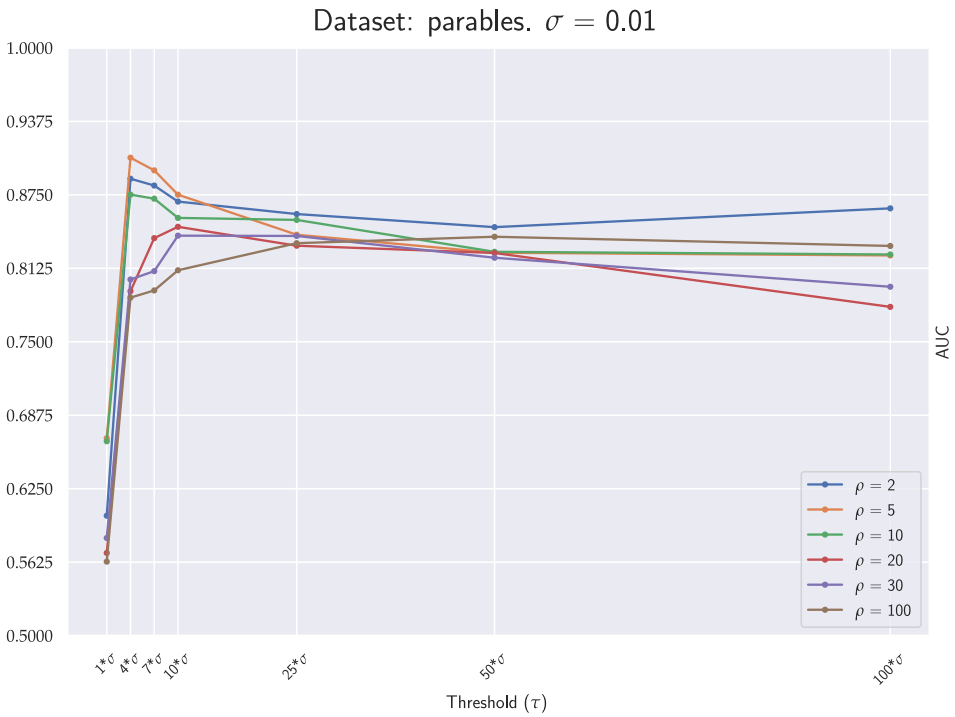


Figure A.17: AE_2 AUCs on parables.

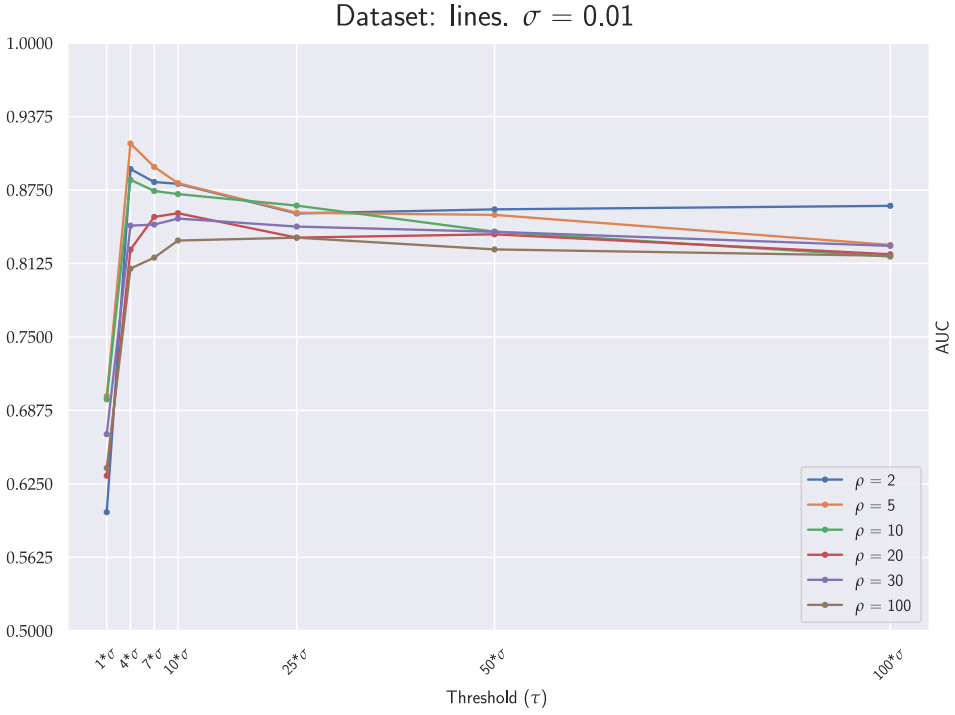


Figure A.18: AE_2 AUCs on lines.

A.1.3. Architecture 3 - 3D

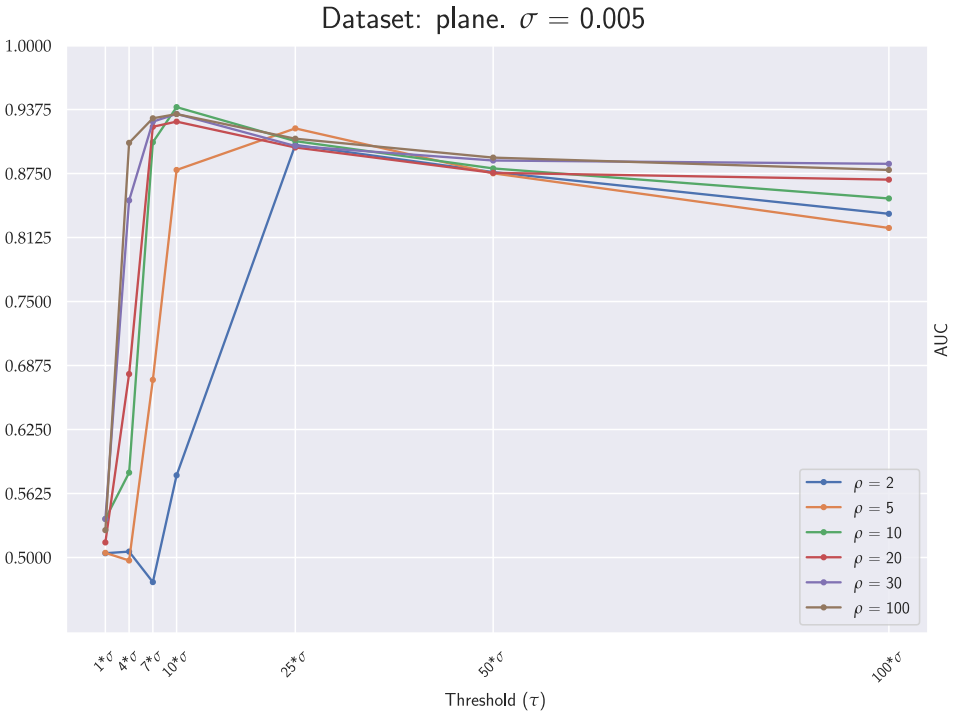


Figure A.19: AE_3 AUCs on plane.

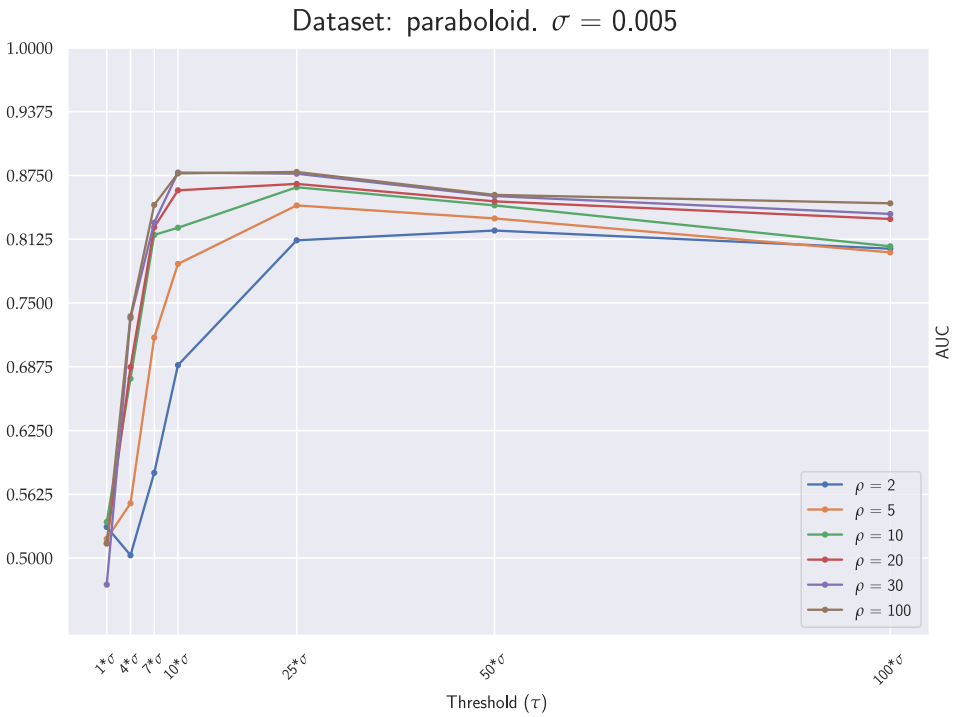


Figure A.20: AE_3 AUCs on paraboloid.

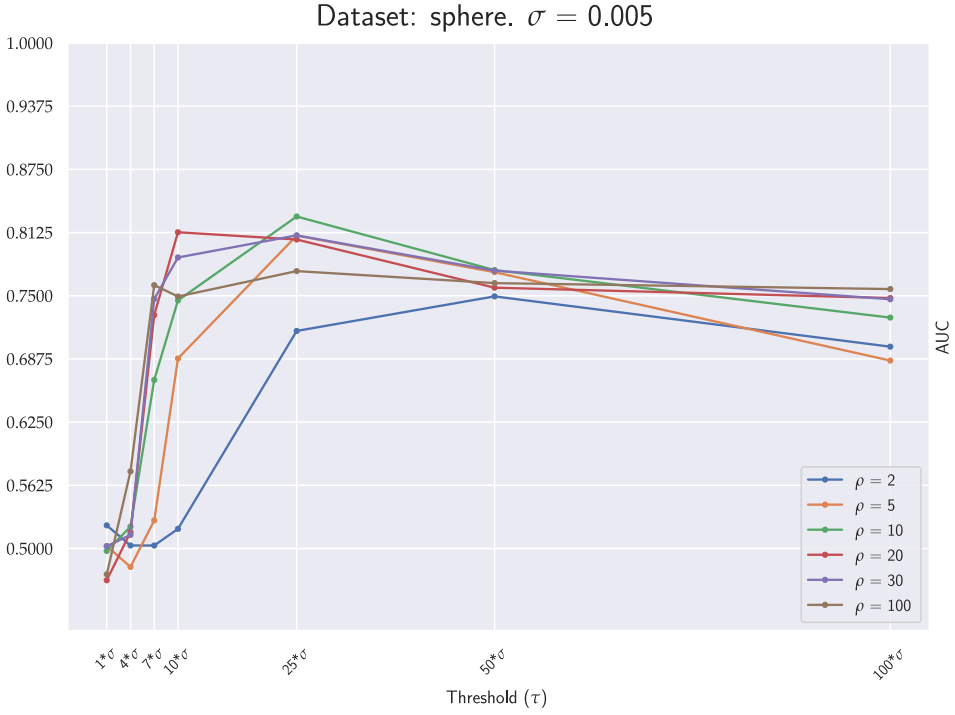


Figure A.21: AE_3 AUCs on sphere.

A.1.4. Architecture 4 - 3D

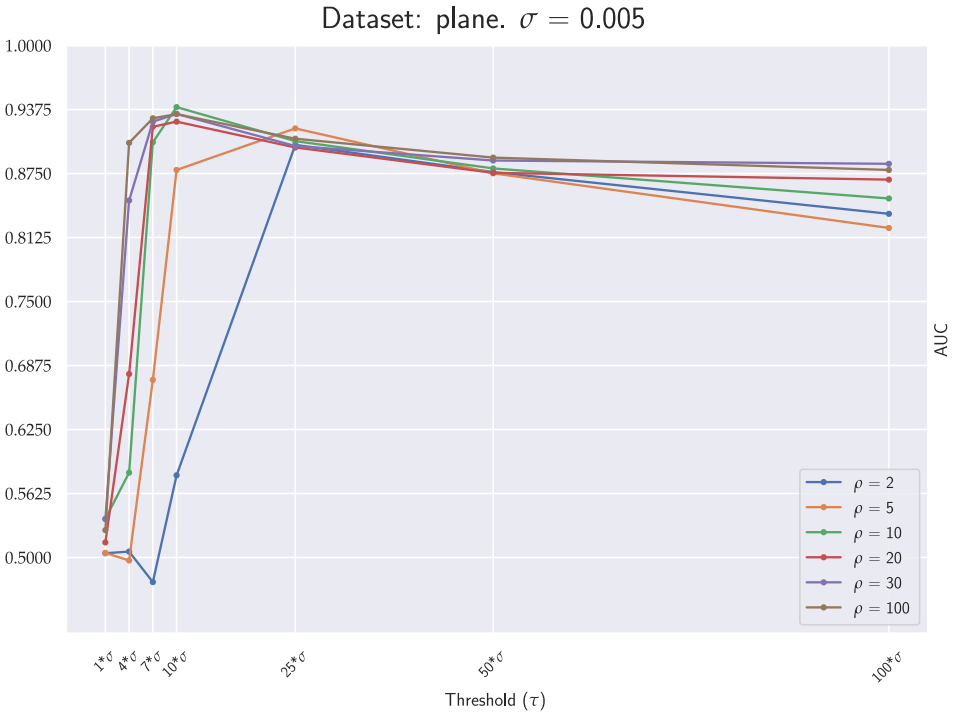


Figure A.22: AE_4 AUCs on plane.

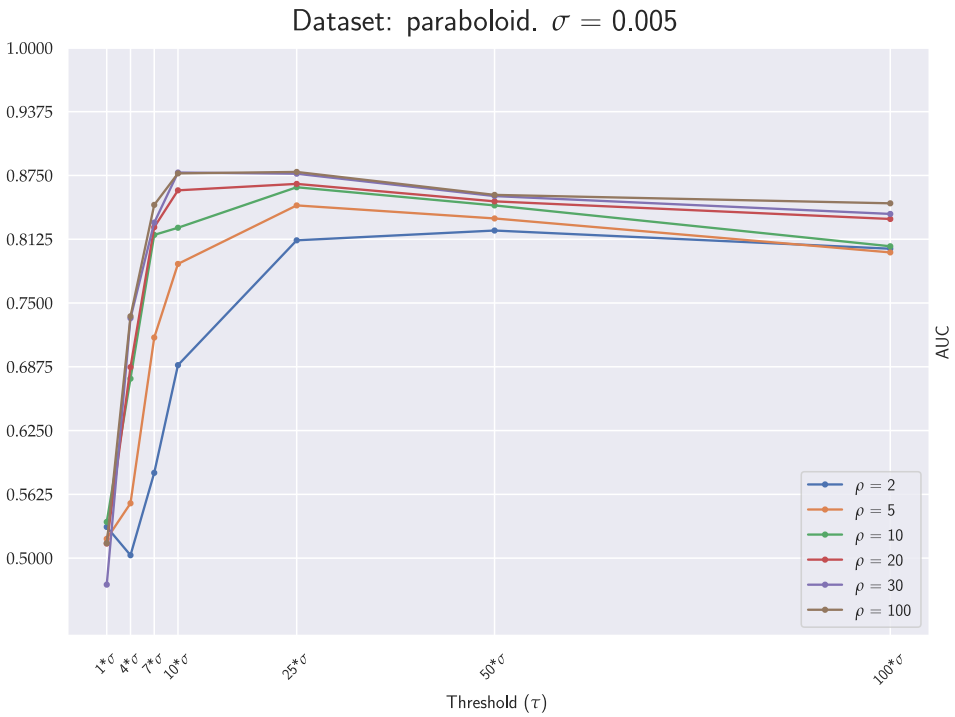


Figure A.23: AE_4 AUCs on paraboloid.

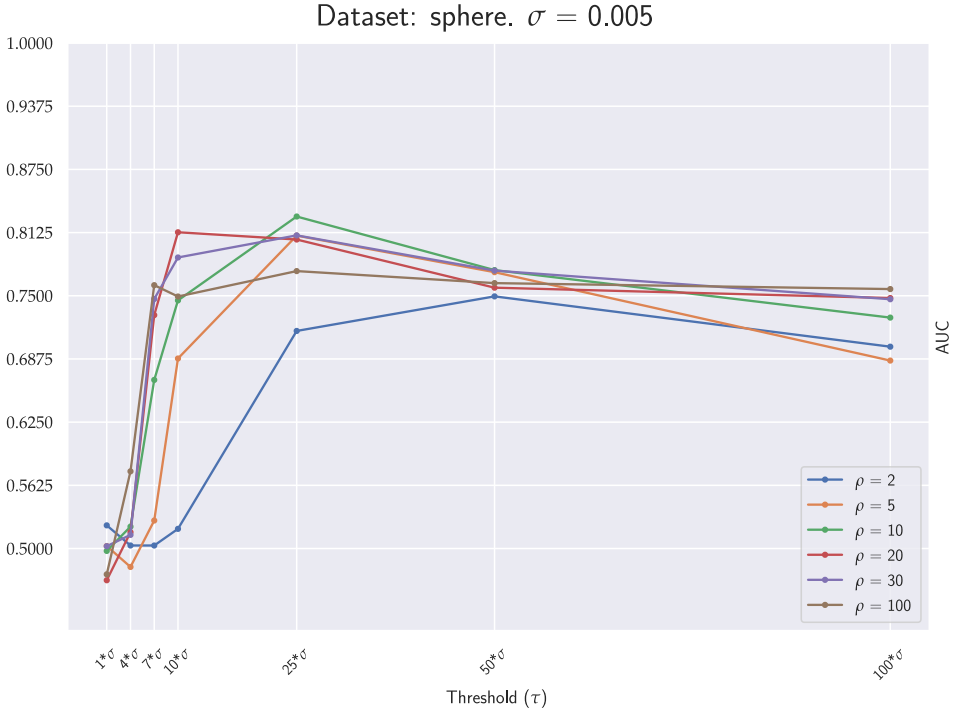


Figure A.24: AE_4 AUCs on sphere.

A.2. Results - Self Organizing Maps

A.2.1. Architecture 1 - 2D

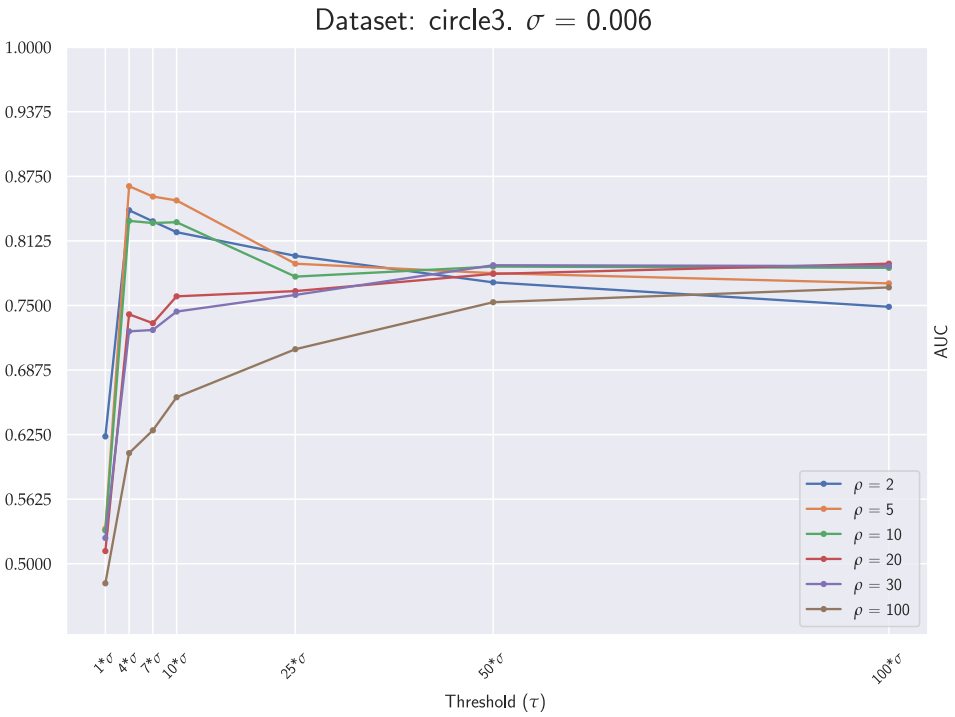


Figure A.25: SOM_1 AUCs on circle3.

Figure A.26: SOM_1 AUCs on circle4.

Figure A.27: SOM_1 AUCs on circle5.

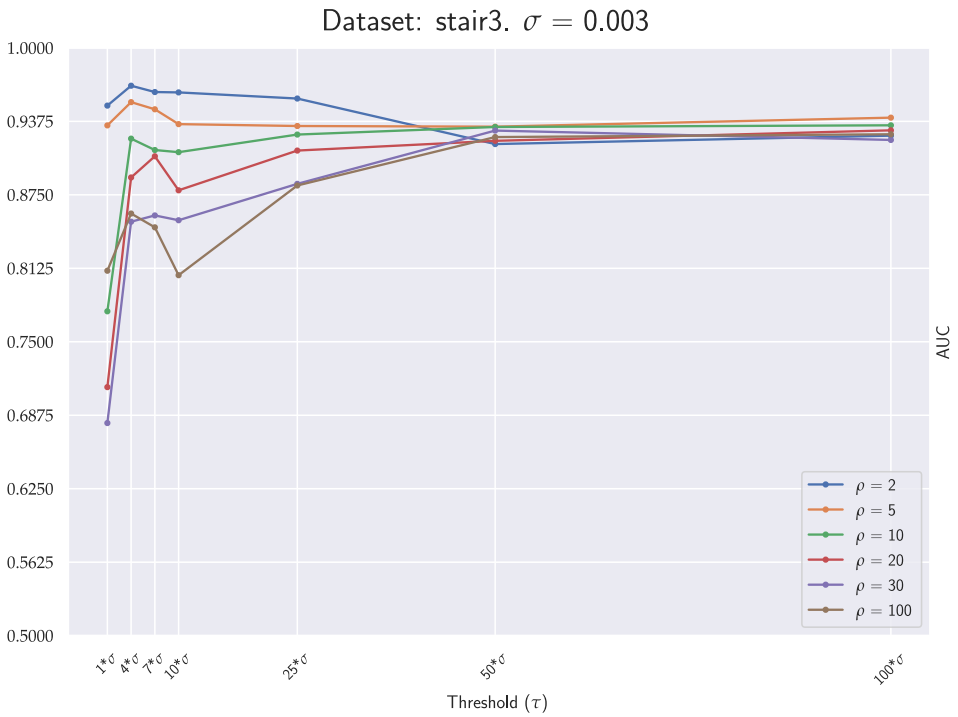


Figure A.28: SOM_1 AUCs on stair3.

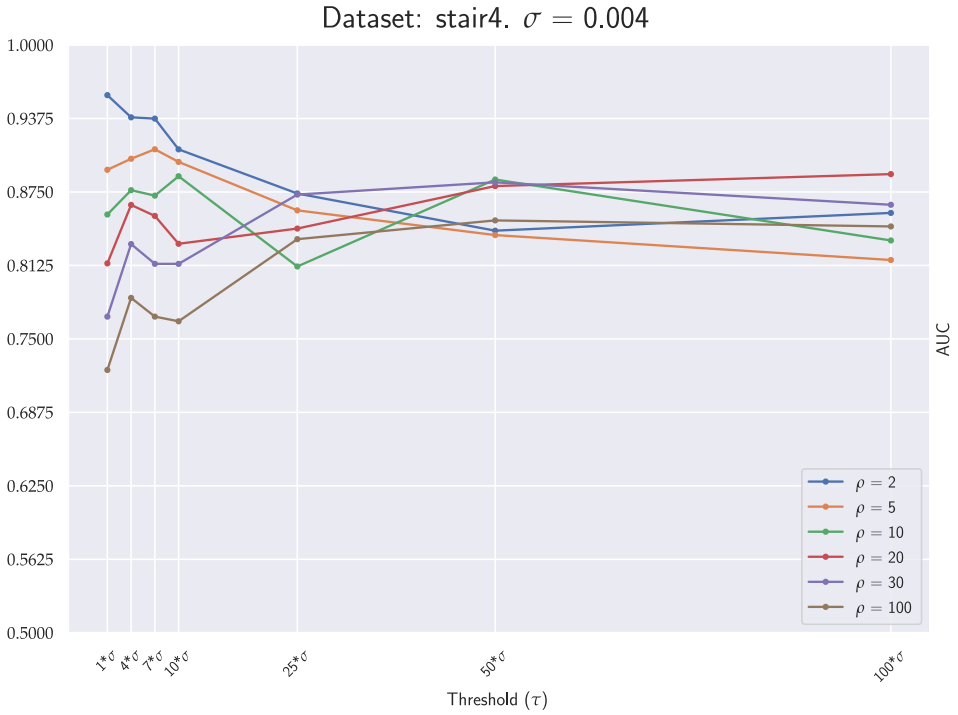


Figure A.29: SOM_1 AUCs on stair4.

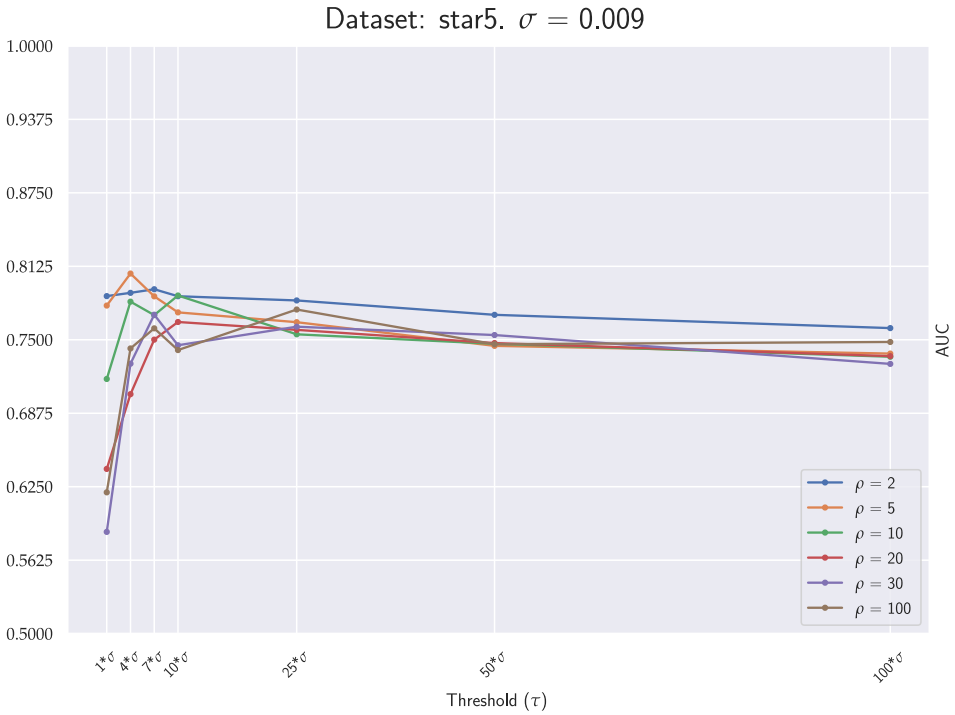


Figure A.30: SOM_1 AUCs on star5.

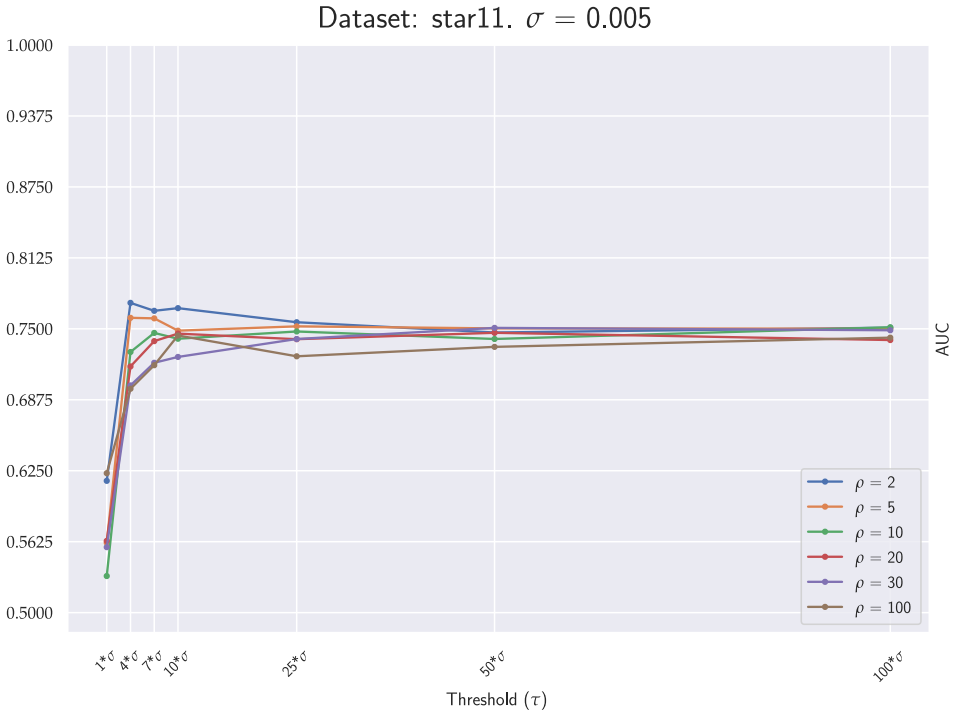


Figure A.31: SOM_1 AUCs on star11.

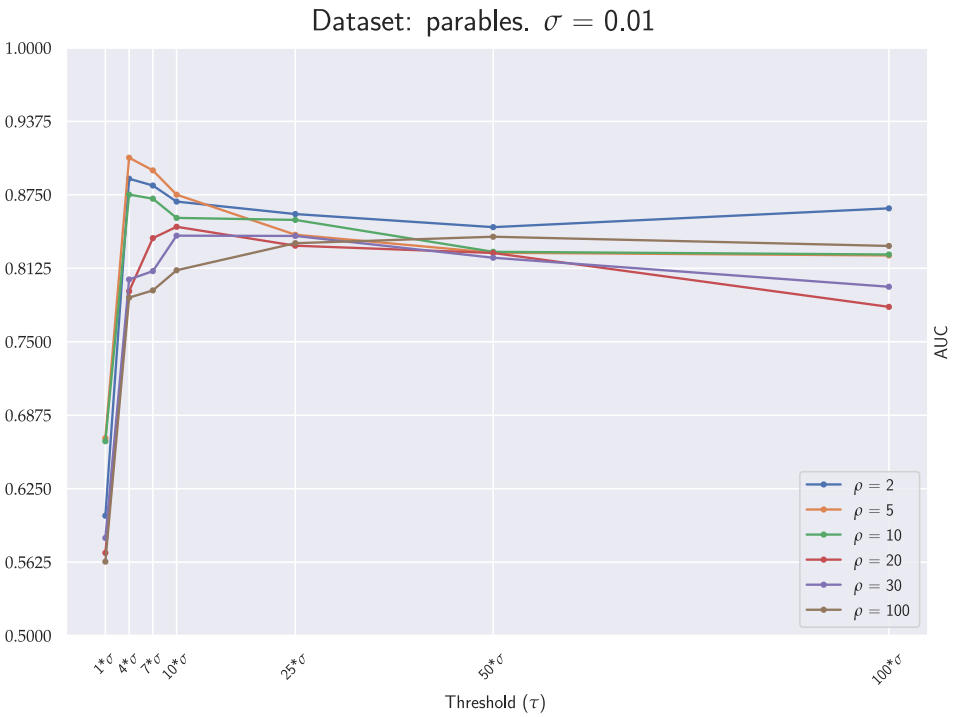


Figure A.32: SOM_1 AUCs on parables.

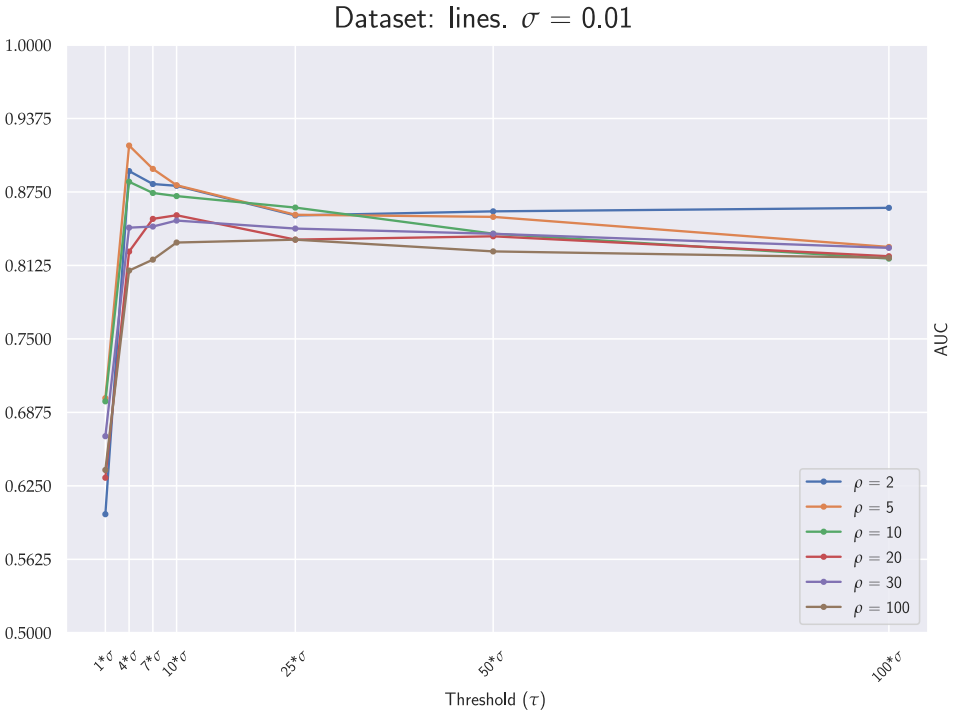


Figure A.33: SOM_1 AUCs on lines.

A.2.2. Architecture 1 - 3D

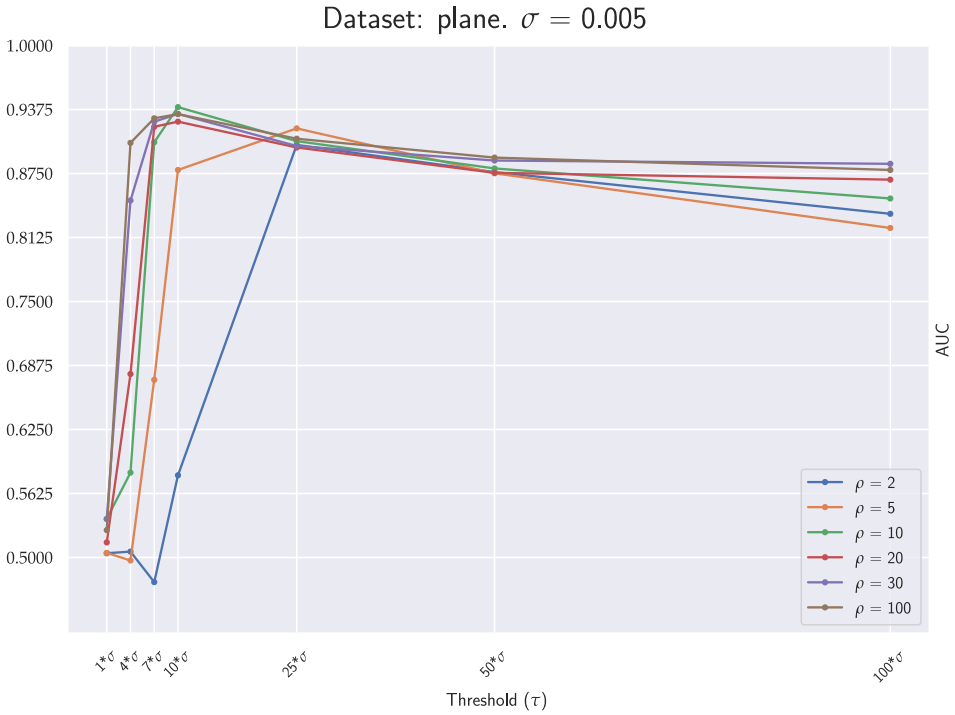


Figure A.34: SOM_1 AUCs on plane.

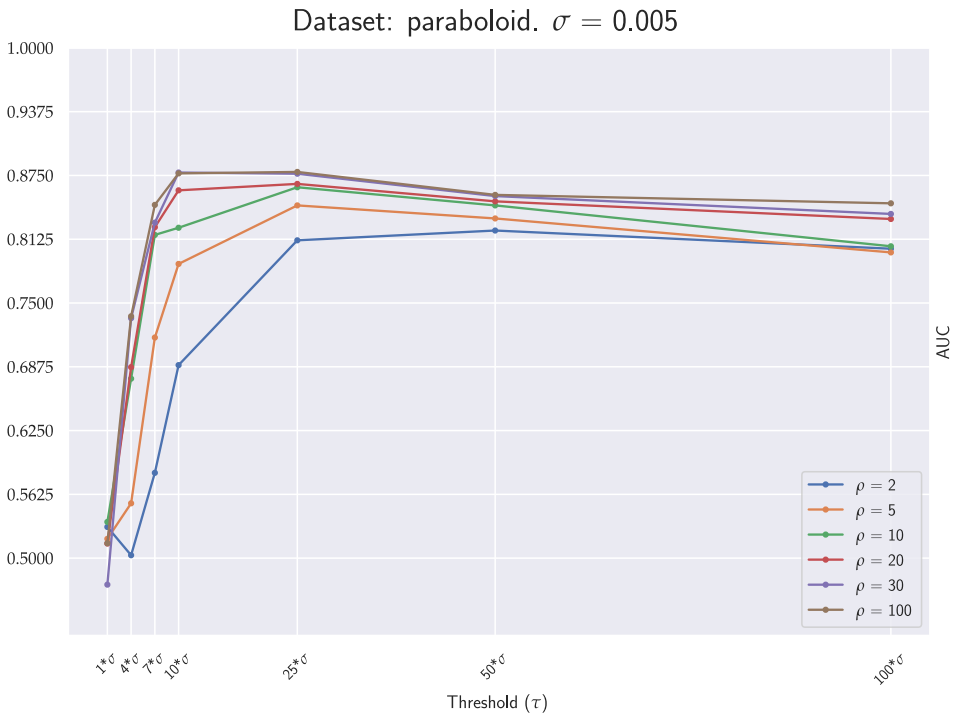


Figure A.35: SOM_1 AUCs on paraboloid.

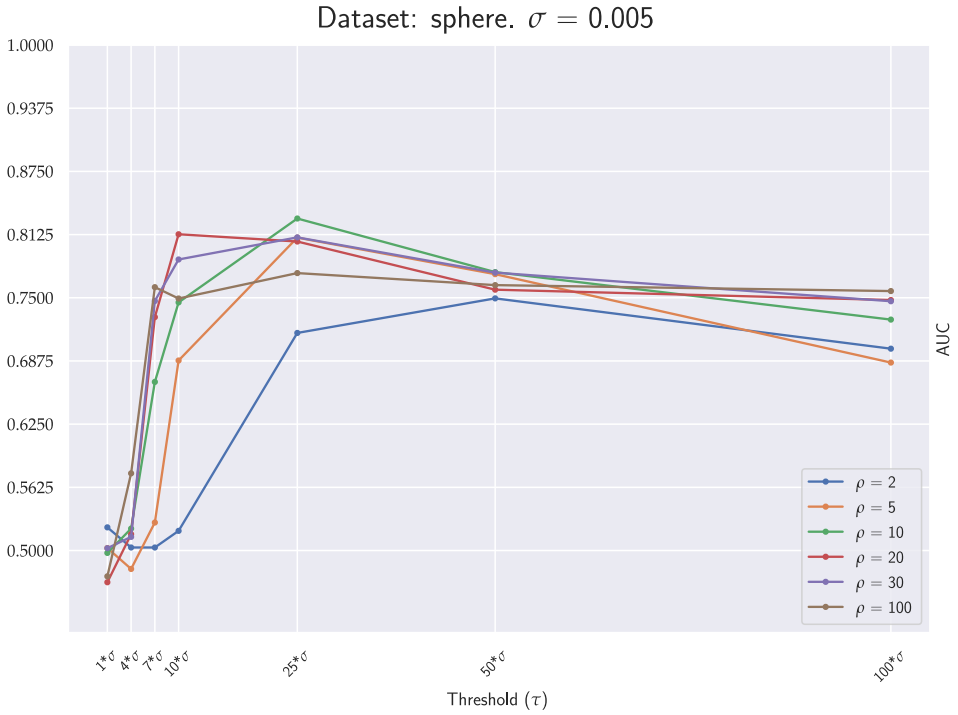


Figure A.36: SOM_1 AUCs on sphere.

A.2.3. Architecture 2 - 2D

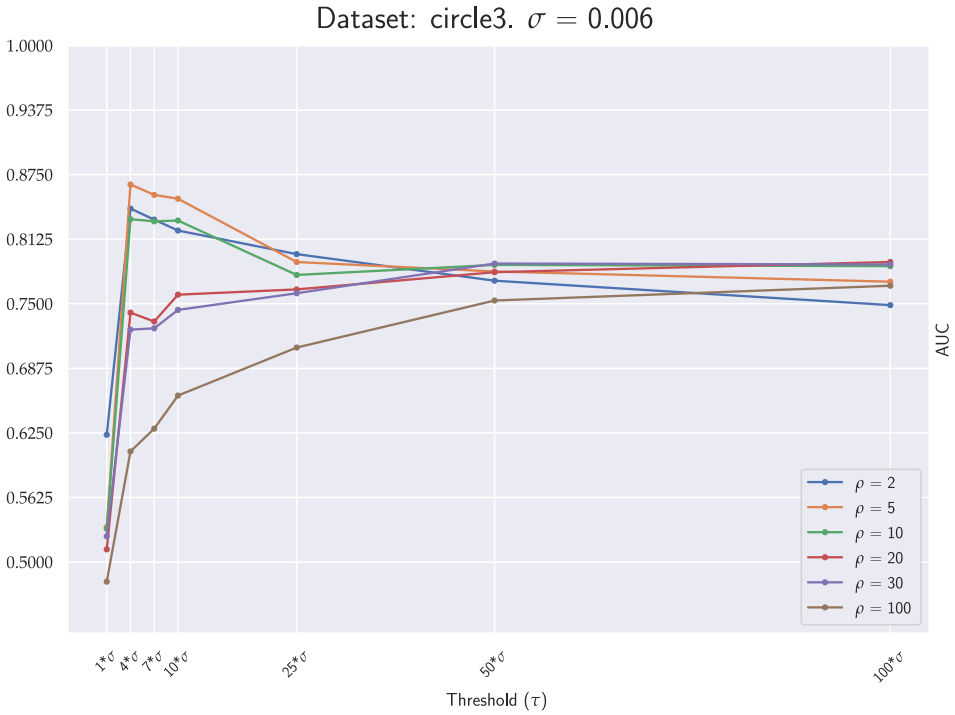


Figure A.37: SOM_2 AUCs on circle3.

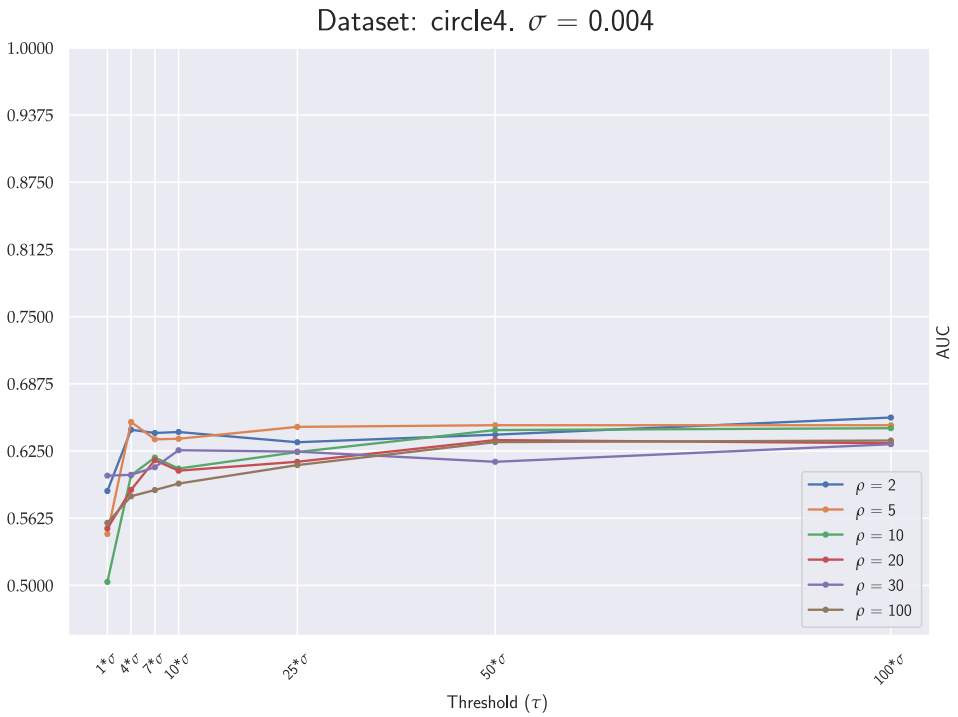


Figure A.38: SOM_2 AUCs on circle4.

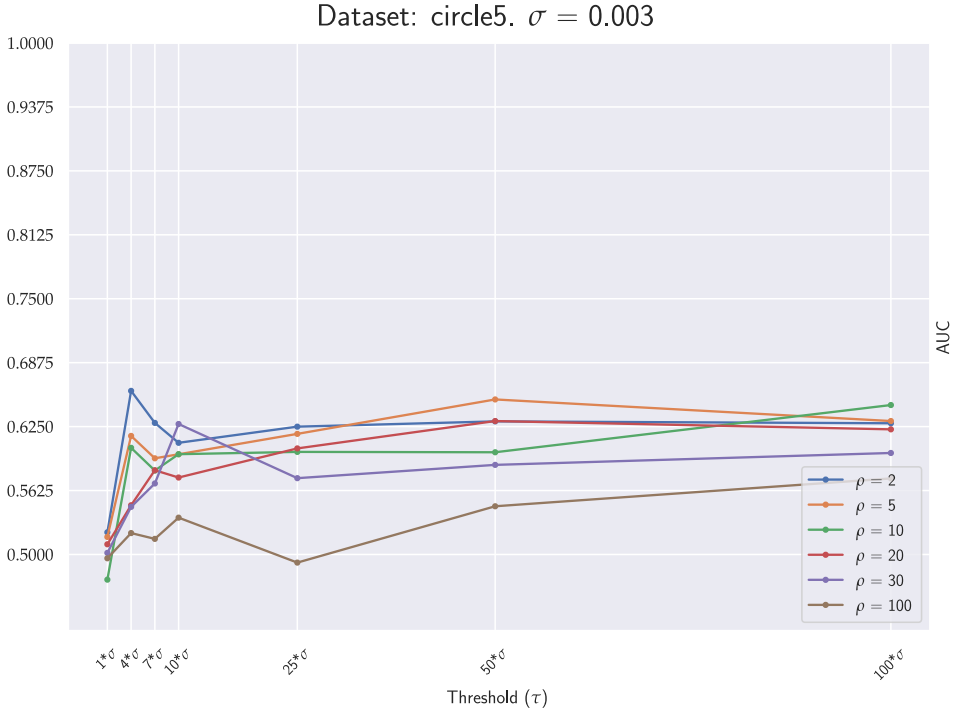


Figure A.39: SOM_2 AUCs on circle5.

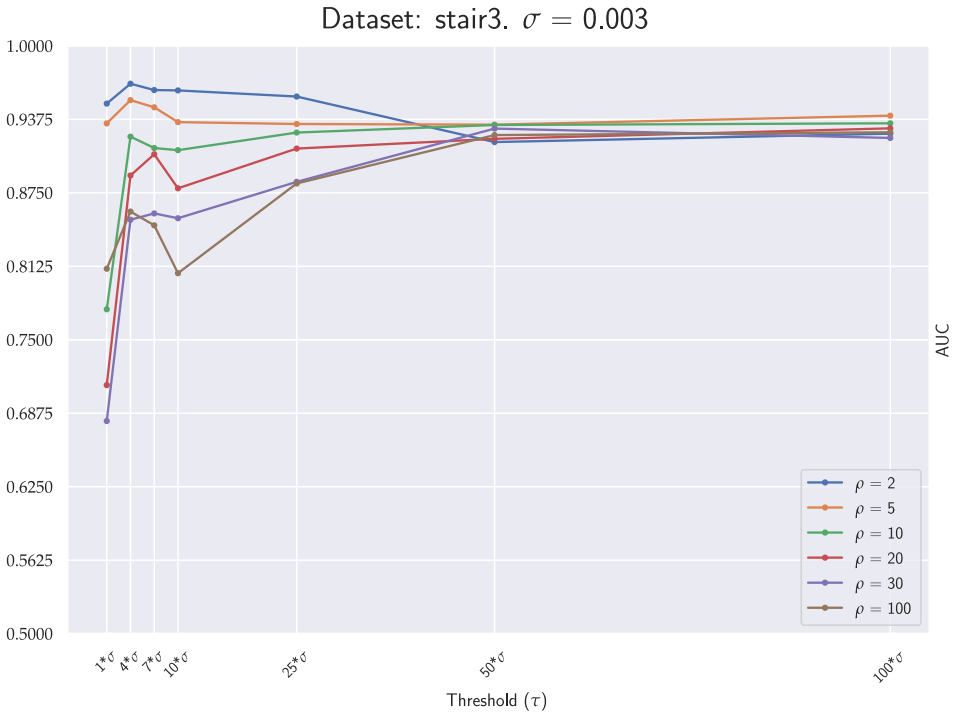


Figure A.40: SOM_2 AUCs on stair3.

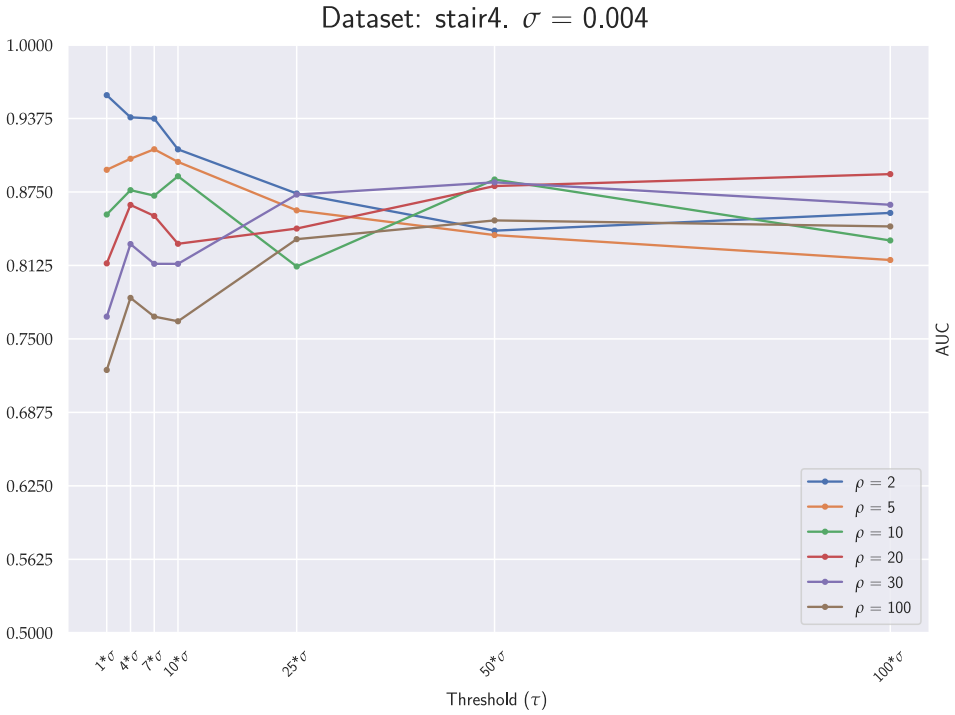


Figure A.41: SOM_2 AUCs on stair4.

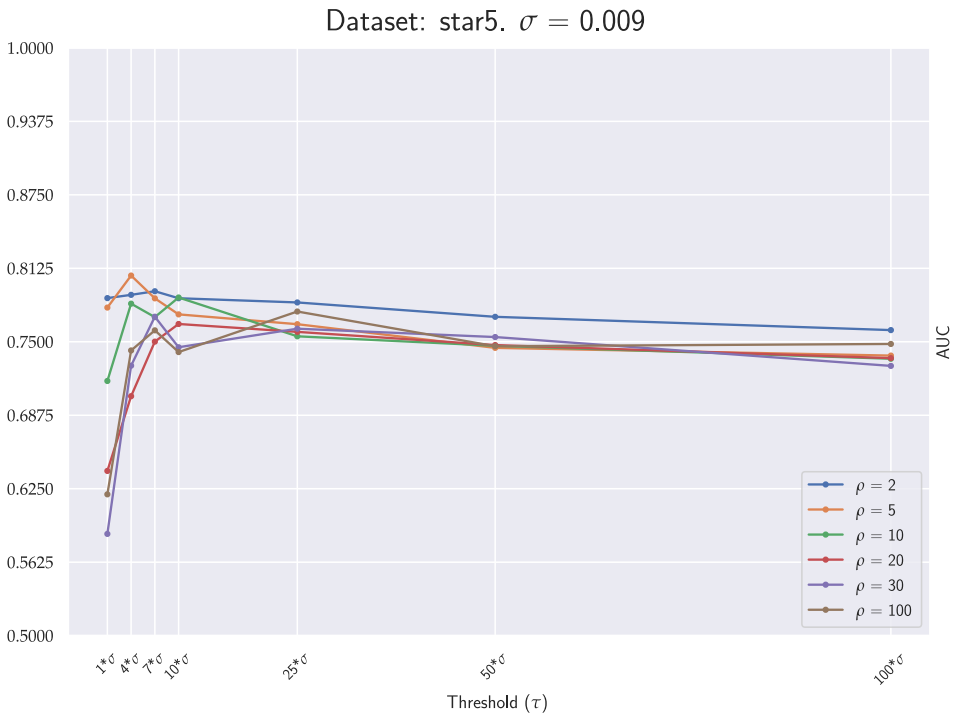


Figure A.42: SOM_2 AUCs on star5.

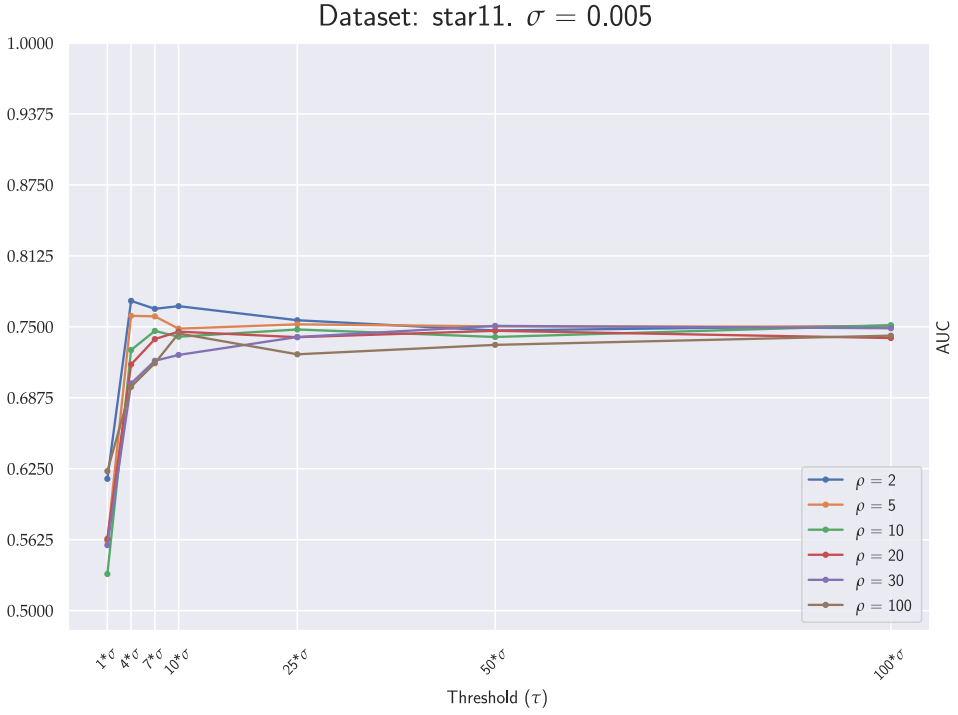


Figure A.43: SOM_2 AUCs on star11.

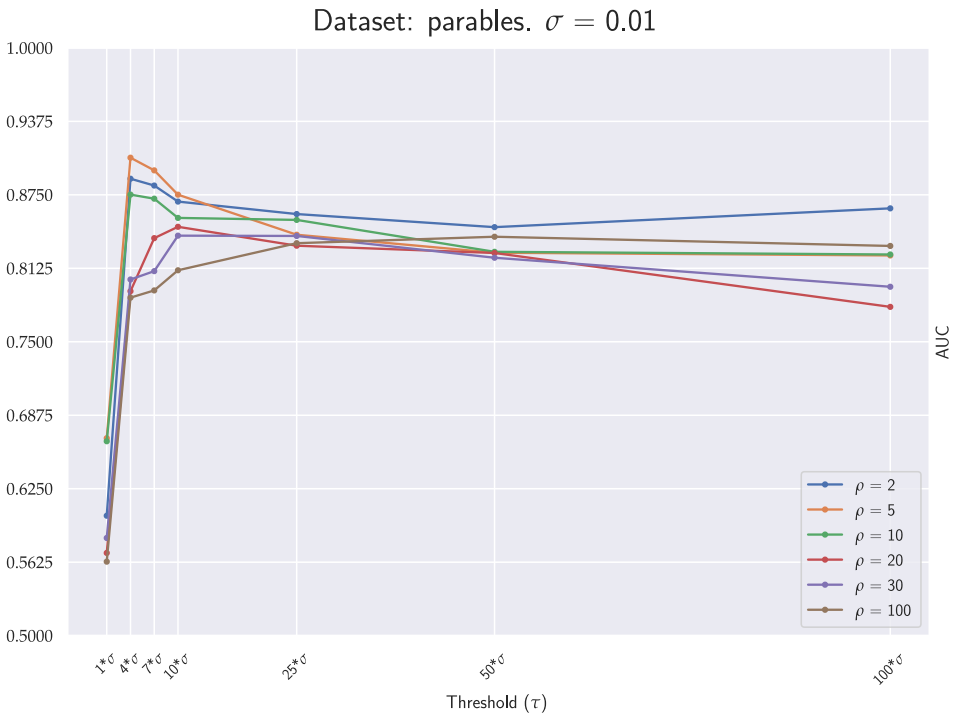


Figure A.44: SOM_2 AUCs on parables.

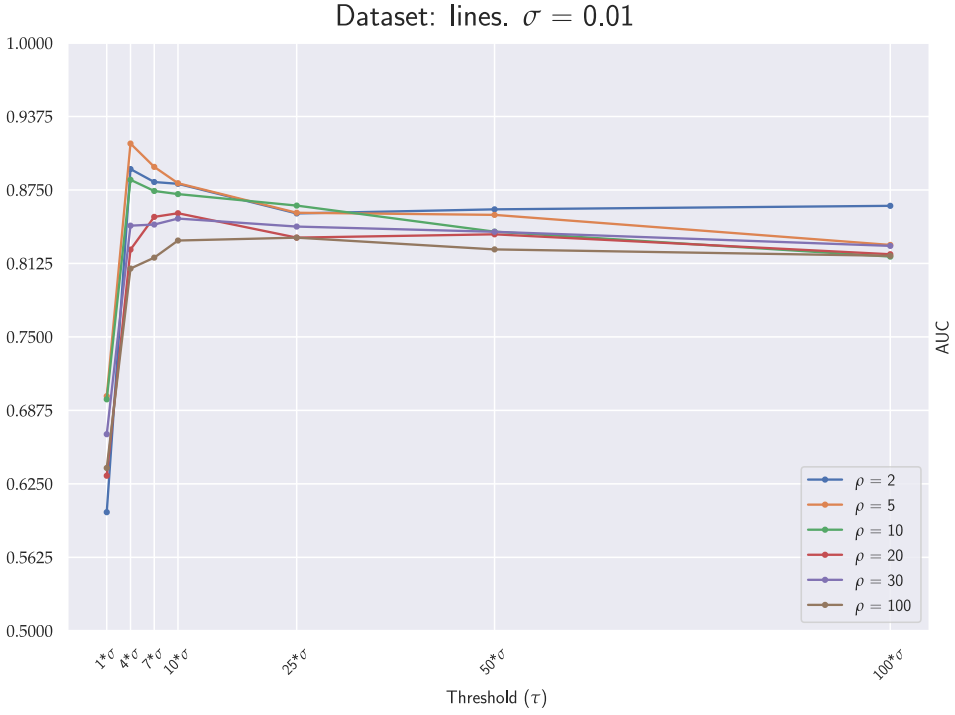


Figure A.45: SOM_2 AUCs on lines.

A.2.4. Architecture 2 - 3D

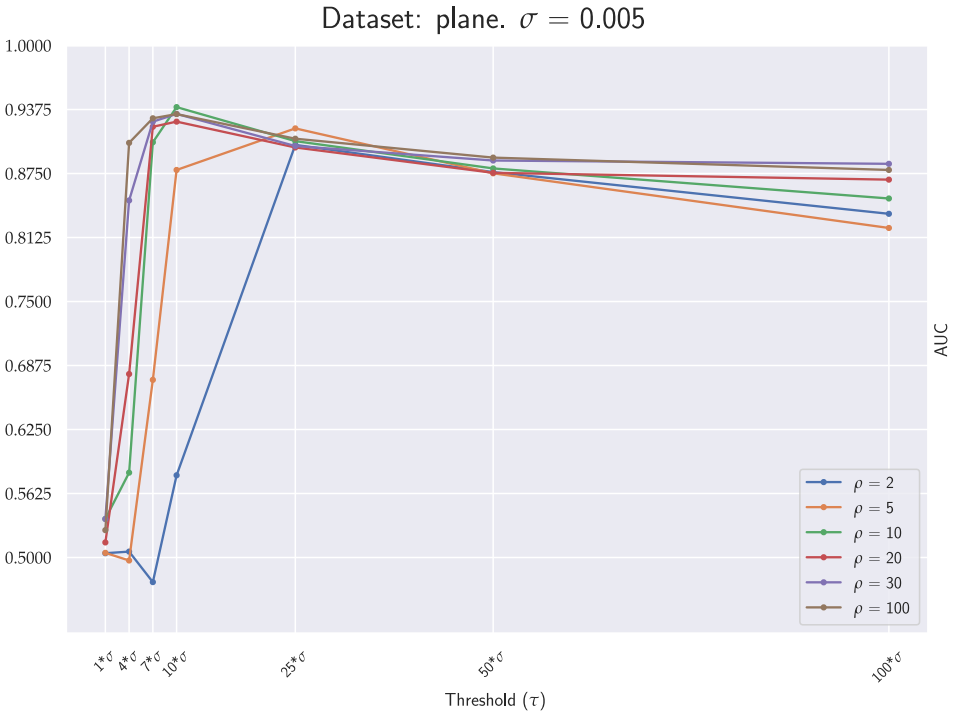


Figure A.46: SOM_2 AUCs on plane.

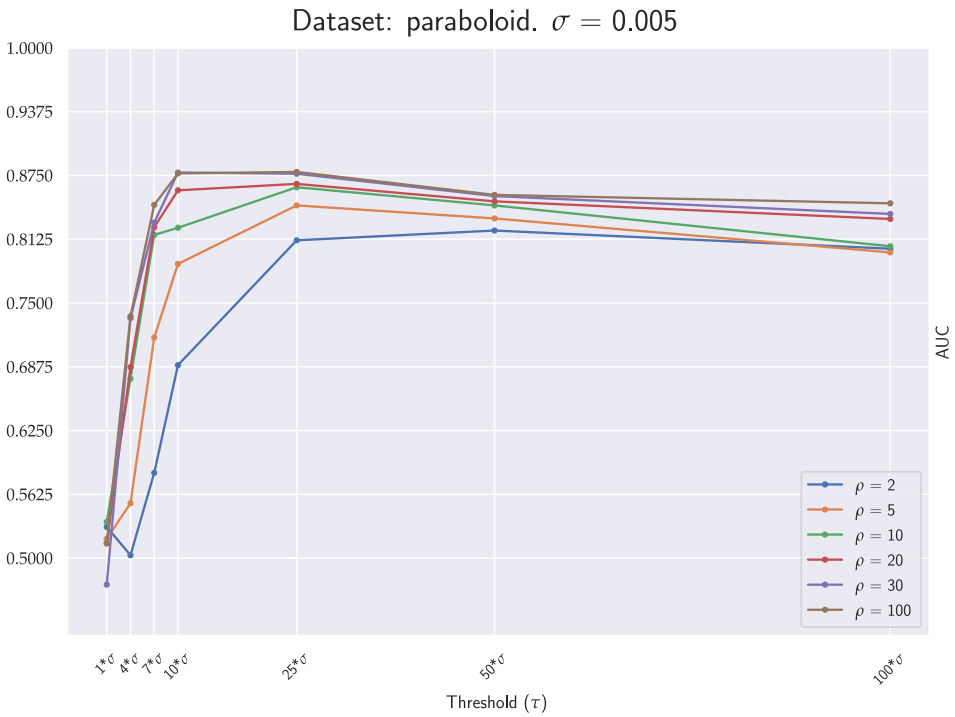


Figure A.47: SOM_2 AUCs on paraboloid.

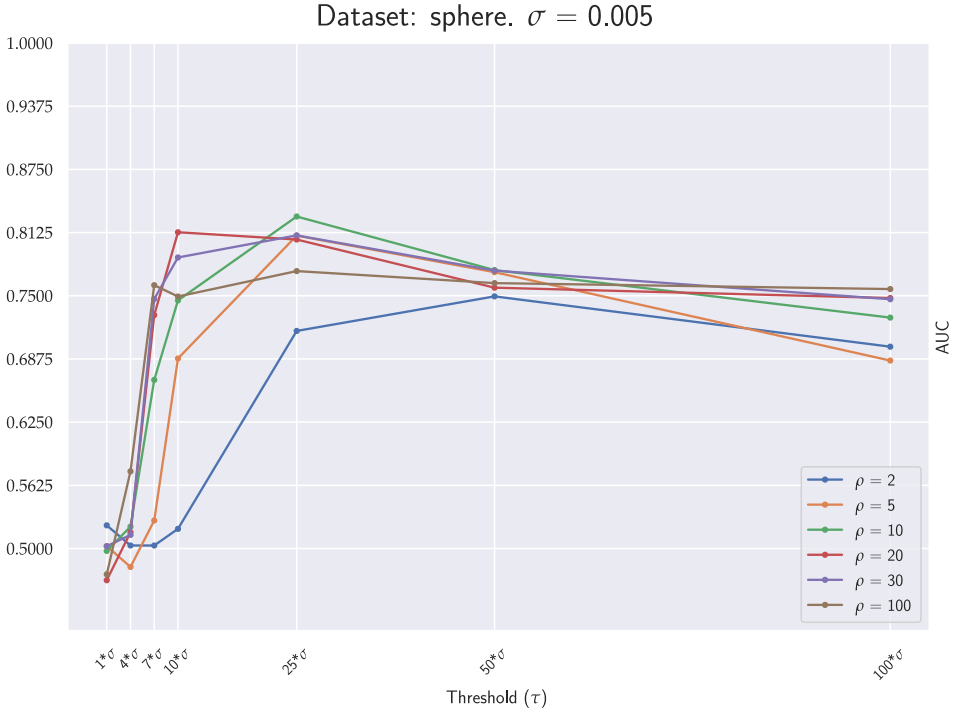


Figure A.48: SOM_2 AUCs on sphere.

A.3. Results - Comparison

Figure A.49: Comparison between models on circle3.

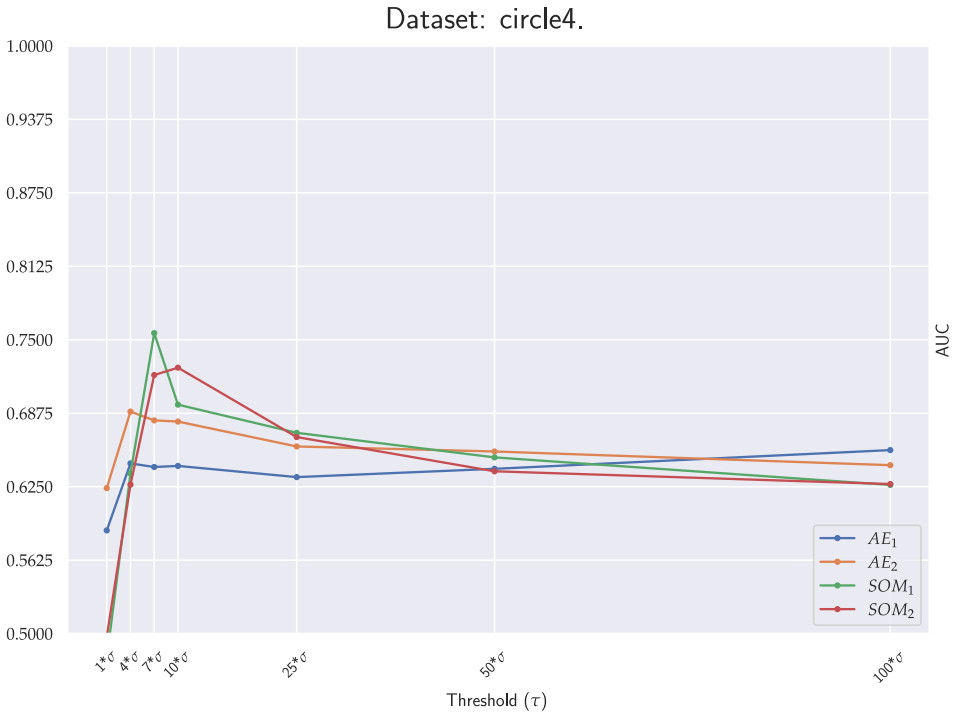


Figure A.50: Comparison between models on circle4.

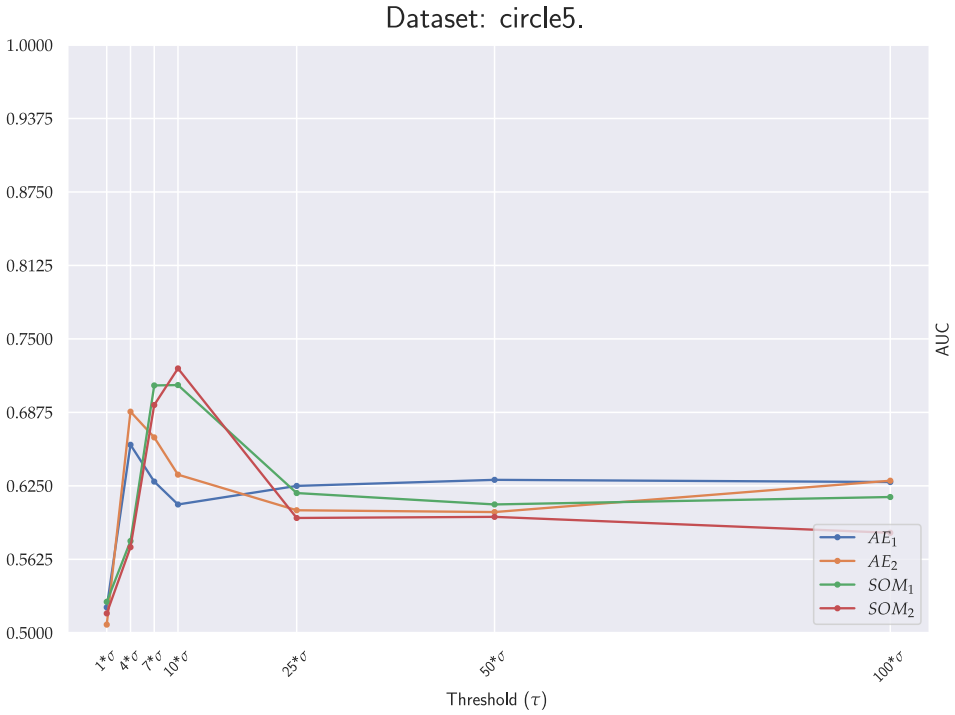


Figure A.51: Comparison between models on circle5.

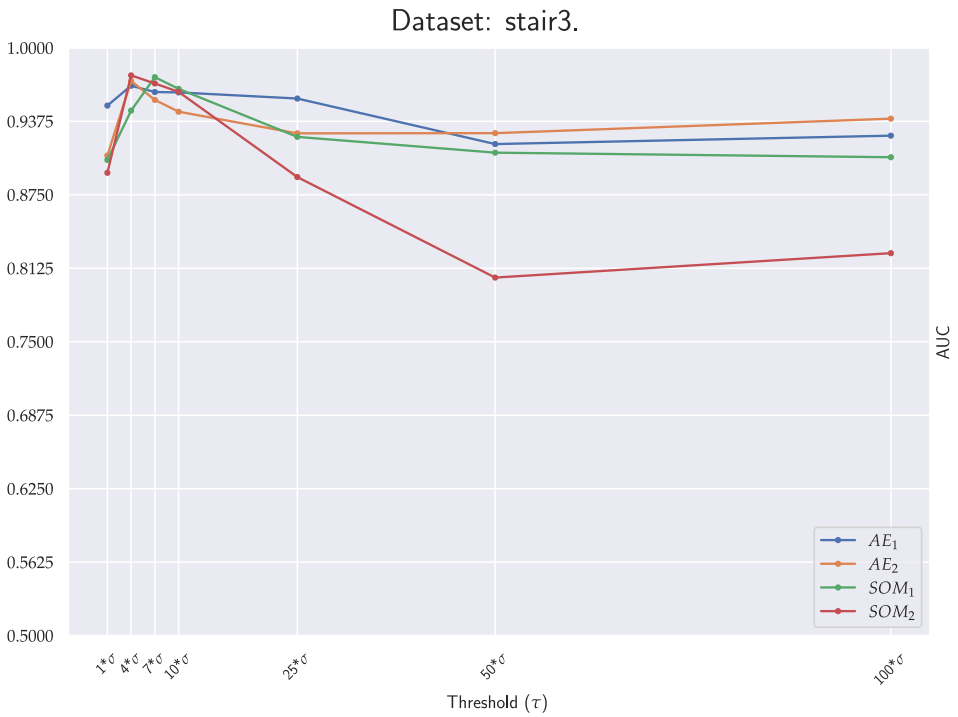


Figure A.52: Comparison between models on stair3.

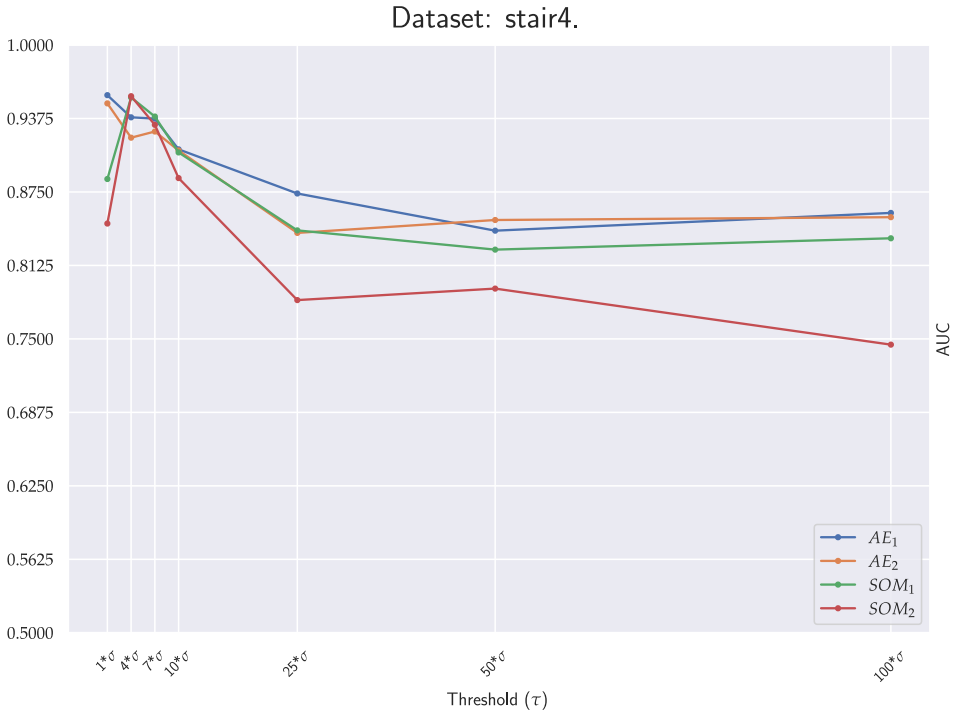


Figure A.53: Comparison between models on stair4.

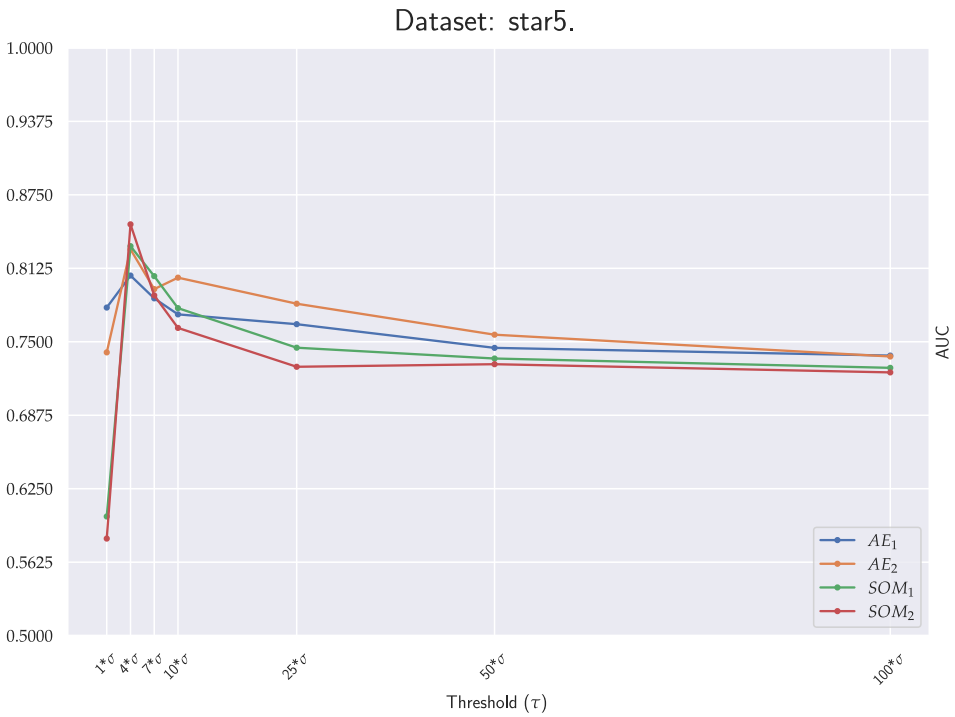


Figure A.54: Comparison between models on star5.

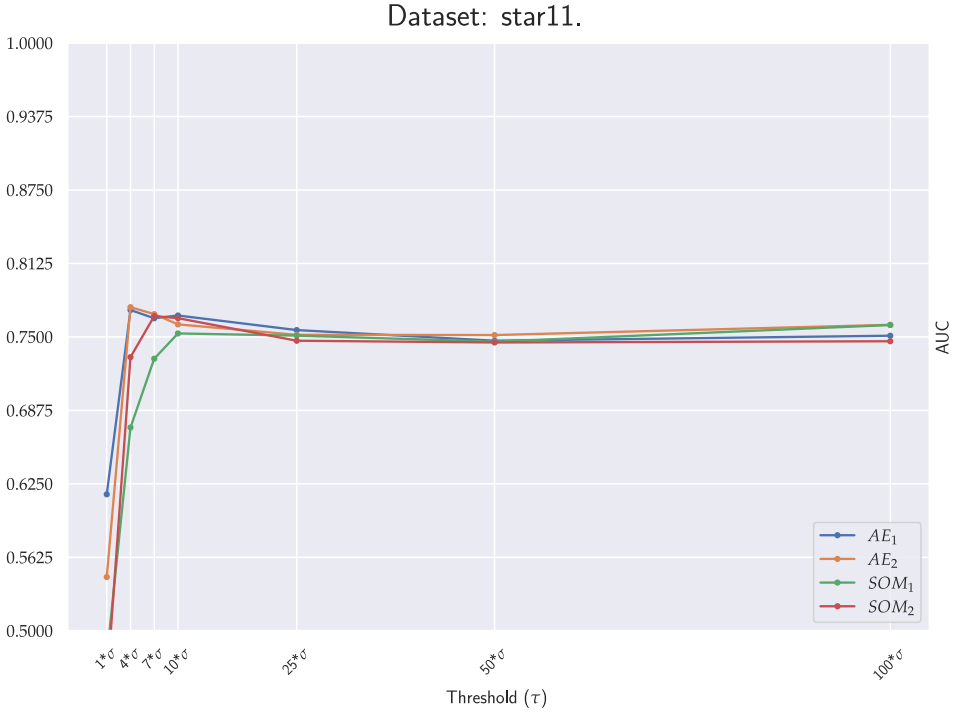


Figure A.55: Comparison between models on star11.

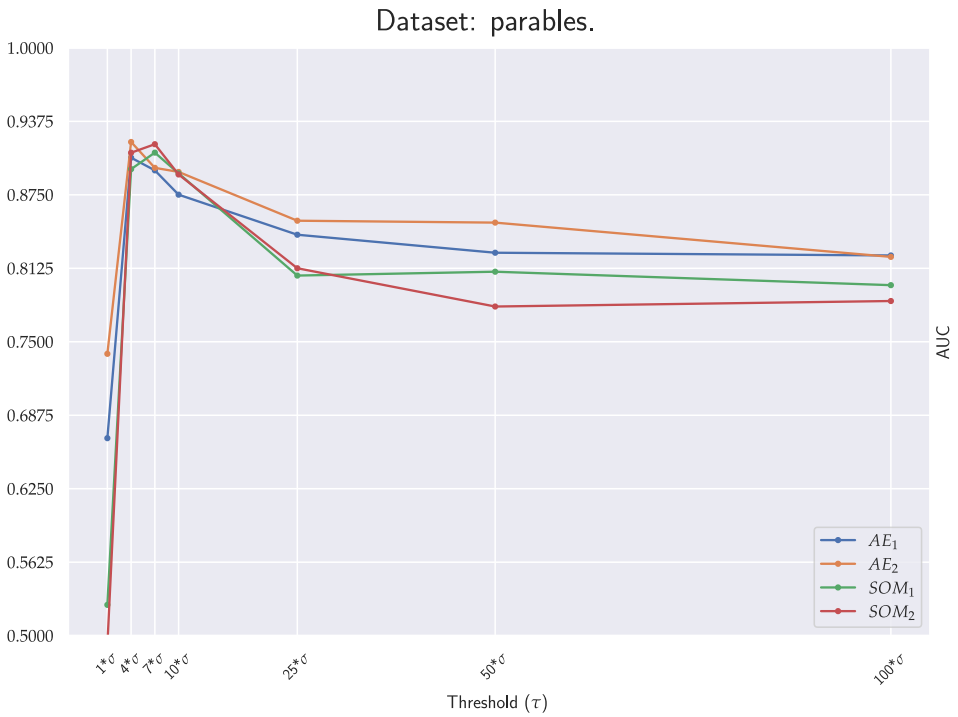


Figure A.56: Comparison between models on parables.

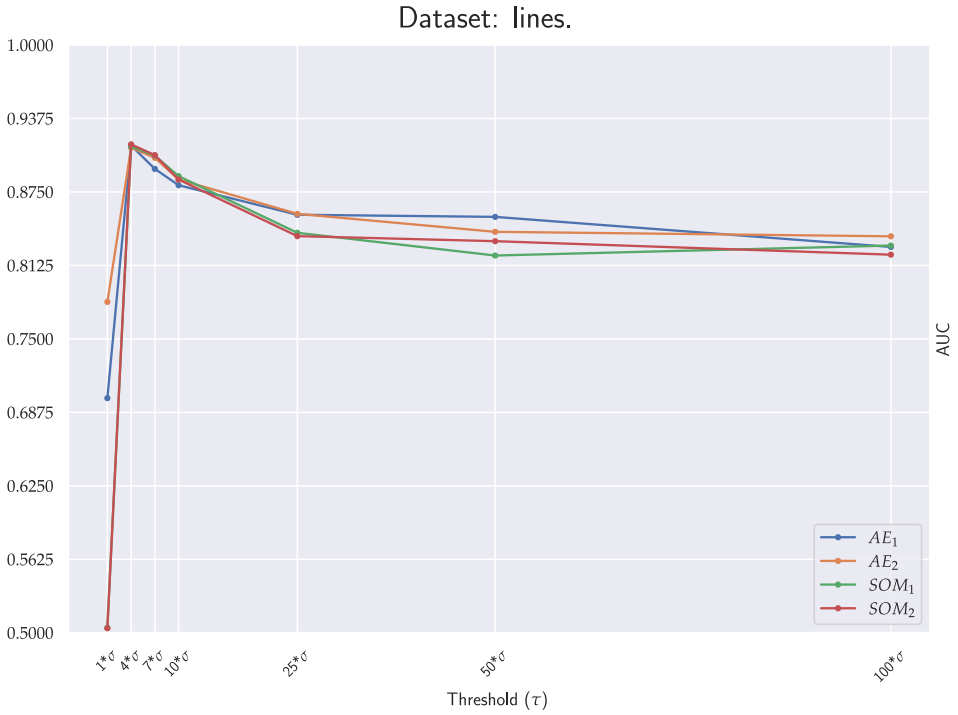


Figure A.57: Comparison between models on lines.

List of Figures

List of Tables

List of Algorithms

List of Listings