

UTRECHT UNIVERSITY

MASTER THESIS

---

# Decision Boundary Maps for Supporting User-Driven Pseudo-labeling

---

*Author:*  
Cristian GROSU (0721808)

*Supervisor:*  
Dr. Alex TELEA

*Second Supervisor:*  
Dr. Ad FEELDERS



**Utrecht  
University**

*A thesis submitted in fulfillment of the requirements  
for the degree of Master of Computing Science*

*at the*

Department of Information and Computing Science  
Faculty of Science  
Utrecht University

December 21, 2023



## Declaration of Authorship

I, Cristian GROSU (0721808), declare that this thesis titled, “Decision Boundary Maps for Supporting User-Driven Pseudo-labeling” and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: Cristian GROSU (0721808)

---

Date: December 21, 2023

---



UTRECHT UNIVERSITY

## *Abstract*

Faculty of Science  
Utrecht University

Master of Computing Science

### **Decision Boundary Maps for Supporting User-Driven Pseudo-labeling**

by Cristian GROSU (0721808)

Training classifier models with semi-labeled datasets, which often have only a limited number of labeled samples, is challenging. This thesis proposes a user-centric methodology for pseudo-labeling semi-labeled data, fusing automatic pseudo-labeling algorithms with user-driven correction of mislabeled data points.

The methodology is supported by a number of visual analytics approaches involving sample visualization via dimensionality reduction techniques and visualization of classifier decision boundaries using so-called Decision Boundary Maps (DBMs). These visuals allow users to find regions of uncertainty where automatic pseudo-labeling may have made errors and correct these accordingly. To speed up the visual analytics loop, we propose various heuristics for efficient and accurate DBM computation. Conducted user experiments show that both domain expert and non-expert users were able to consistently correct wrong labels and improve classifier performance for different datasets and classifier models, with only a limited effort in a limited amount of time.

The study underscores the importance and potential of visualization tools in the context of semi-labeled datasets and semi-supervised learning and provides a foundation for future research in this area.

*Keywords:* **Semi-supervised learning, Pseudo-labeling, Dimensionality Reduction, Decision Boundary Maps, Data visualization.**



## *Acknowledgements*

I would like to express my deepest appreciation to the supervisor Dr. Alex Telea for his invaluable patience and feedback. This journey would not have been possible without his knowledge and expertise in the domain of data visualization.

I am also grateful to Phd. Barbara Benato (University of Campinas) for providing a real-world complex semi-labeled data set, for her feedback regarding the usefulness of the visualization tool, and for participating in the experiments part of this research.





# Contents

<b>Declaration of Authorship</b>	<b>iii</b>
<b>Abstract</b>	<b>v</b>
<b>Acknowledgements</b>	<b>vii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Problem description	1
1.1.1 Context	1
1.1.2 Problem requirements	2
1.1.3 Research questions	3
1.2 Structure of this thesis	4
<b>2 Related Work</b>	<b>7</b>
2.1 Pseudo-labeling Methods in Semi-Supervised Learning	7
2.1.1 Label Propagation	7
2.1.2 Optimal Path Forest	8
2.1.3 Deep Feature Annotation	8
2.2 Projection Methods	9
2.2.1 Projections as Visual Aids for Classification Systems	9
2.2.2 Principal Component Analysis	9
2.2.3 t-distributed Stochastic Neighbor Embedding	9
2.2.4 Uniform Manifold Approximation and Projection	10
2.2.5 Projection methods comparison	11
2.3 Projection Errors	11
2.4 Inverse Projection Methods	12
2.5 Inverse Projection Errors	13
2.6 Self-Supervised Neural Projection	14
2.7 Image-based visualization of Classifier Decision Boundaries	15
2.8 Conclusions	15
<b>3 Solution Design</b>	<b>17</b>
3.1 Projecting the data set and generating the DBM	18
3.1.1 Projection methods implementation details	18
3.1.2 Inverse projection methods implementation details	19
3.1.3 DBM computation algorithm	22
3.2 Projection and inverse projection errors	23
3.2.1 Projection errors	24
3.2.2 Inverse Projection errors	29
3.3 Data points re-labeling and classifier retraining	30
3.4 Optimization heuristics for DBM generation	33
3.4.1 Binary Split heuristic	35
Binary Split intuition	35

	Binary Split algorithm . . . . .	37
	Binary Split heuristic complexity analysis . . . . .	39
	Binary Split vs Vanilla DBM . . . . .	40
3.4.2	Confidence-based Split heuristic . . . . .	42
	Confidence-based Split intuition . . . . .	42
	Confidence-based Split algorithm . . . . .	43
	Confidence-based Split heuristic complexity analysis . . . . .	45
	Confidence-based Split vs Binary Split vs Vanilla DBM . . . . .	46
3.4.3	Confidence interpolation heuristic . . . . .	47
	Confidence interpolation intuition . . . . .	48
	Confidence interpolation algorithm . . . . .	49
	Confidence interpolation heuristic complexity analysis . . . . .	50
	Confidence interpolation vs Vanilla DBM . . . . .	50
3.4.4	DBM generation algorithms comparison . . . . .	51
	Run time complexity analysis . . . . .	52
	Accuracy analysis . . . . .	53
	Discussions . . . . .	54
3.5	Visualization tool overview . . . . .	54
3.5.1	Configuration window . . . . .	55
3.5.2	Data set visualization and re-labeling window . . . . .	57
3.6	Conclusions . . . . .	60
<b>4</b>	<b>Experiments And Results</b>	<b>63</b>
4.1	Data sets used in the experiments . . . . .	63
4.2	DBM optimization heuristic algorithms . . . . .	64
4.2.1	Experiments and Methodology . . . . .	65
4.2.2	Results and Discussions . . . . .	67
4.2.3	Conclusions . . . . .	74
4.3	Visualization tool evaluation . . . . .	74
4.3.1	Experiments and Methodology . . . . .	75
4.3.2	Results and Discussions . . . . .	79
	MNIST data set experiments series . . . . .	79
	Protozoan cysts data set experiments series . . . . .	87
4.4	Conclusions . . . . .	90
<b>5</b>	<b>Conclusions And Future Work</b>	<b>93</b>
5.1	Conclusions . . . . .	93
5.2	Future work . . . . .	94
<b>A</b>	<b>Figures</b>	<b>95</b>
<b>B</b>	<b>Experiments environment set up</b>	<b>99</b>

# List of Figures

1.1	Visualization tool preview . . . . .	4
2.1	DBM vs SDBM. Data set: MNIST. White and black pixels represent the data samples . . . . .	16
3.1	Pipeline of uploading data set and classifier to the visualization tool . . . . .	17
3.2	DBM and 2D data set embedding computation and visualization pipeline (R1 and R2) . . . . .	18
3.3	NNinv Architecture . . . . .	20
3.4	Autoencoder Architecture . . . . .	21
3.5	SSNP Architecture . . . . .	22
3.6	Errors computation pipeline (R4 and R5) . . . . .	23
3.7	Example of a 2D projection of a subset of MNIST data set. The colored points represent 2D projections of the data points . . . . .	25
3.8	Example of projection errors $\epsilon'_{10}(\cdot)$ computed using algorithm 2. Interpolation method: RBF (linear $\phi(r) = -r$ ). White regions: high error values, dark regions: low error values . . . . .	26
3.9	Examples of how the neighbors' order is preserved depending on how well $\mathcal{P}^{-1}$ approximates $\mathcal{P}_+^{-1}$ . . . . .	28
3.10	Example of projection errors $\epsilon''_{10}(\cdot)$ computed using algorithm 3. White regions: high error values, dark regions: low error values . . . . .	29
3.11	Example of inverse projection error computation using the approximation of inverse projection derivative . . . . .	30
3.12	Example of the inverse projection error gradient map. White regions: high error values, dark regions: low error values . . . . .	30
3.13	Data set re-labeling and classifier re-training pipeline (R3.1 and R3.2) . . . . .	31
3.14	Example of re-labeling and classifier re-training . . . . .	32
3.15	Dummy DBM algorithm 1 run-times for different image resolutions. Dataset: MNIST, $\mathcal{P}$ : t-SNE, $\mathcal{P}^{-1}$ : NNinv . . . . .	33
3.16	DBM construction using the Binary Split heuristic for a classifier with 2 classes . . . . .	36
3.17	Binary Split label errors. Data set: MNIST, $\mathcal{P}$ : t-SNE, $\mathcal{P}^{-1}$ : NNinv Resolution: $256 \times 256$ , Initial blocks resolution: $B = 32$ Number of errors: $100 \cdot \frac{1}{256^2} = 0.0015\%$ . . . . .	40
3.18	Binary Split vs Dummy DBM run-times. Data set: MNIST, $\mathcal{P}$ : t-SNE, $\mathcal{P}^{-1}$ : NNinv Initial blocks resolution: $B = 32$ . . . . .	41
3.19	Binary Split algorithm accuracy . . . . .	42
3.20	Block split in "Binary Split" and "Confidence-based Split" heuristics . . . . .	44
3.21	Confidence-based Split label errors. Data set: MNIST, $\mathcal{P}$ : t-SNE, $\mathcal{P}^{-1}$ : NNinv Resolution: $256 \times 256$ , Initial blocks resolution: $B = 32$ Number of errors: $100 \cdot \frac{8}{256^2} = 0.0122\%$ . . . . .	46

3.22	Confidence-based Split vs Binary split vs Dummy DBM run-times. Data set: MNIST, $\mathcal{P}$ : t-SNE, $\mathcal{P}^{-1}$ : NNinv Initial blocks resolution: $B = 32$ . . . . .	47
3.23	Confidence-based Split algorithm accuracy . . . . .	48
3.24	Confidence interpolation intuition for generating a DBM with 4 classes using bilinear interpolation . . . . .	49
3.25	Confidence Interpolation label errors. Data set: MNIST, $\mathcal{P}$ : t-SNE, $\mathcal{P}^{-1}$ : NNinv Resolution: $256 \times 256$ , Initial blocks resolution: $B = 32$ . . . . .	50
3.26	Confidence Interpolation vs Dummy DBM run-times. Data set: MNIST, $\mathcal{P}$ : t-SNE, $\mathcal{P}^{-1}$ : NNinv Initial blocks resolution: $B = 32$ . . . . .	51
3.27	Confidence Interpolation algorithm accuracy . . . . .	51
3.28	DBM heuristics run-times comparison. Data set: MNIST, $\mathcal{P}$ : t-SNE, $\mathcal{P}^{-1}$ : NNinv, Initial blocks resolution: $B = 32$ . . . . .	52
3.29	DBM algorithms accuracy comparison. Data set: MNIST, $\mathcal{P}$ : t-SNE, $\mathcal{P}^{-1}$ : NNinv Initial blocks resolution: $B = 32$ Confidence map approximation method: Bilinear for Binary and Confidence-base split, bicubic for the Confidence Interpolation algorithm . . . . .	53
3.30	Explanation of label errors . . . . .	54
3.31	Visualization tool usage workflow . . . . .	55
3.32	Configuration window of the visualization tool . . . . .	56
3.33	DBM visualization window . . . . .	57
3.34	DBM visualization window, tooltip image, and information about the data point on hover . . . . .	58
3.35	Re-labeling the data points using the visualization tool . . . . .	59
3.36	Choosing the DBM generation algorithm in the visualization tool . . . . .	60
4.1	Classifier $\mathcal{C}$ architecture used in the experiments . . . . .	65
4.2	Experimental pipeline for DBM optimization heuristic algorithms, (a) Experiment preparation, (b) Initial number of blocks $B$ hyperparameter tuning, (c) <i>confidence_interpolation</i> hyperparameter tuning, (d) Results analysis . . . . .	66
4.3	Experimental pipeline for the tool usage evaluation . . . . .	75
4.4	Feature extractor architecture for Protozoan cysts data set . . . . .	78
4.5	Classifier $\mathcal{C}$ performance gains after 5 iterations of tool usage for User 1. Data set: MNIST . . . . .	85
4.6	Classifier $\mathcal{C}$ performance gains after 5 iterations of tool usage for User 2. Data set: MNIST . . . . .	85
A.1	Upload a classifier to the visualization tool . . . . .	95
A.2	Successfully computed DBM in the configuration window . . . . .	95
A.3	Uploading a data set to the visualization tool . . . . .	96
A.4	The DBM techniques menu . . . . .	97
A.5	DBM plots based on the checkboxes from DBM visualization window . . . . .	98
A.6	Example of confidence, direct and inverse error values encoded in the opacity of the pixels . . . . .	98

# List of Tables

3.1	Projections and inverse projection methods possible combinations . . .	22
3.2	Run times of algorithm 1 for DBM resolution $512 \times 512$ on different machines. Data set: MNIST . . . . .	33
3.3	DBM heuristics complexity analysis . . . . .	52
4.1	Overview of the data sets used in the experiments . . . . .	64
4.2	Initial number of blocks $B$ hyper-parameter tuning. Data set: MNIST, <i>confidence_interpolation = cubic</i> . . . . .	67
4.3	<i>confidence_interpolation</i> hyper-parameter tuning. Data set: MNIST, $B = 32$ , $\mathcal{P}$ : t-SNE, $\mathcal{P}^{-1}$ : NNinv . . . . .	69
4.4	<i>confidence_interpolation</i> hyper-parameter tuning. Data set: MNIST, $B = 32$ , $\mathcal{P}$ : UMAP, $\mathcal{P}^{-1}$ : NNinv . . . . .	70
4.5	<i>confidence_interpolation</i> hyper-parameter tuning Data set: MNIST, $B = 32$ , $\mathcal{P}$ : PCA, $\mathcal{P}^{-1}$ : NNinv . . . . .	70
4.6	<i>confidence_interpolation</i> hyper-parameter tuning. Data set: MNIST, $B = 32$ , $\mathcal{P}, \mathcal{P}^{-1}$ : Autoencoder . . . . .	71
4.7	<i>confidence_interpolation</i> hyper-parameter tuning. Data set: MNIST, $B = 32$ , $\mathcal{P}, \mathcal{P}^{-1}$ : SSNP . . . . .	71
4.8	DBM algorithms comparison. Data set: MNIST. Hyperparameters: $B = 32$ , <i>confidence_interpolation = cubic</i> for Confidence Interpolation algorithm, <i>confidence_interpolation = linear</i> for Binary and Confidence Split algorithms . . . . .	73
4.9	Key notations for tool usage evaluation . . . . .	76
4.10	Users' knowledge in experimental sessions . . . . .	77
4.11	Impact of pseudo-labels on classifier $\mathcal{C}$ performance: A comparison with ground truth labels. Data set: MNIST, the pseudo-labels generated by DeepFA . . . . .	79
4.12	Classifier $\mathcal{C}$ performance after 5 iterations of tool usage. Data set: MNIST . . . . .	80
4.13	Classifier performance trace over 5 iterations of tool usage. Data set: MNIST, User: User 1, Performance metric: Accuracy . . . . .	81
4.14	Classifier performance trace over 5 iterations of tool usage. Data set: MNIST, User: User 1, Performance metric: Cohen's kappa score . . . . .	82
4.15	Classifier performance trace over 5 iterations of tool usage. Data set: MNIST, User: User 2, Performance metric: Accuracy . . . . .	83
4.16	Classifier performance trace over 5 iterations of tool usage. Data set: MNIST, User: User 2, Performance metric: Cohen's kappa score . . . . .	84
4.17	Classifier $\mathcal{C}$ performance gains after 5 iterations of tool usage. Data set: MNIST . . . . .	85
4.18	Impact of pseudo-labels on classifier $\mathcal{C}$ performance: A comparison with ground truth labels. Data set: Protozoan cysts, the pseudo-labels generated by DeepFA . . . . .	87

4.19	Classifier performance trace over 5 iterations of tool usage. Data set: Protozoan cysts . . . . .	87
4.20	Comparison of classifier performance trace over 5 iterations of tool usage per user. Data set: Protozoan cysts . . . . .	88
4.21	Amount of corrected labels per user after 5 iterations of tool usage. Data set: Protozoan cysts . . . . .	89
B.1	All dependent libraries and versions . . . . .	99
B.2	System characteristics of the machine on which the experiments were run . . . . .	99

# List of Abbreviations

<b>DBM</b>	Decision Boundary Map
<b>LP</b>	Label Propagation
<b>OPF</b>	Optimal Path Forest
<b>DeepFA</b>	Deep Feature Annotation
<b>PCA</b>	Principal Component Analysis
<b>UMAP</b>	Uniform Manifold Approximation and Projection
<b>t-SNE</b>	t-distributed Stochastic Neighbor Embedding
<b>NNP</b>	Neural Network Projection
<b>iLAMP</b>	Inverse Linear Affine Multidimensional Projection
<b>NNinv</b>	Neural network for inverse projection / Inverse Neural Network
<b>SSNP</b>	Self-Supervised Neural Projection
<b>SDBM</b>	Supervised Decision Boundary Maps
<b>RBF</b>	Radial Basis Function





# List of Notations

$D$	Data set
$D_{2d}$	Data set 2D projection
$\mathcal{D}$	Data space
$\mathcal{C}$	Classifier
$C$	Classifier call cost/number of operations
$\mathcal{P}$	Projection function
$\mathcal{P}^{-1}$	Inverse projection function
$C(\cdot)$	Continuity metric
$T(\cdot)$	Trustworthiness metric
$\epsilon_k(\cdot), \epsilon'_k(\cdot), \epsilon''_k(\cdot)$	Projection errors metric
$\ \cdot\ $	Vector norm
$\mathcal{M}_{n,m}(\cdot)$	All matrices of size $n \times m$
$\mathcal{O}_{n,m}$	Zeros matrix of size $n \times m$
$\mathcal{B}$	Block of pixels $(x, y, w, h)$ , $x, y$ - central pixel, $w, h$ - the size of the block
$B$	Number of blocks
$\epsilon_{labels}$	Percentage of mislabeled pixels
$NRMSE_{confidence}$	Normalized mean square error between ground truth and approximated confidence values



## Chapter 1

# Introduction

### 1.1 Problem description

#### 1.1.1 Context

Machine learning is more and more becoming a part of our lives. Even if we are not data scientists or data engineers, all of us at some point in time will use machine learning algorithms. The most popular tasks in this field are such tasks as classification (i.e. predicting a categorical label or class for a given input), clustering (i.e. grouping similar data points together based on their features without any prior knowledge of the classes or labels), dimensionality reduction (i.e. reducing the number of features or variables in a data set while retaining as much information as possible) and more others.

The classification task is usually approached as follows, the data scientist has a labeled data set and uses it to train a model that will predict labels for new points that are not in the data set. This is a classical example of Supervised Learning.

In practice, however, one of the largest problems concerns the availability of suitable data sets for training and/or testing. Getting a reasonable-size labeled data set is not impossible but at least very time-consuming. An approach to this problem is to have at least a small amount of the data points labeled and then based on these data points use specific algorithms (e.g. clustering algorithms) which assign the label for a data point based on the most similar data point that is already labeled. This is an example of a sub-type of machine learning called Semi-Supervised Machine Learning. In general Semi-Supervised Learning is an intermediate approach between supervised and unsupervised learning, where the model learns from partially labeled data and uses the unlabelled data to improve the model.

Of course, automation in labeling non-labeled data points is saving a lot of human hours in such kinds of tasks. However, the core approach of assigning the same label if two data points share similar features works only if the data can be perfectly and easily separated into clusters. Thus, training a classifier on data labeled in such a way is probably not going to give the best results for all kinds of data sets. Moreover, if one could use a machine to achieve perfect pseudo-labeling, then starting from a poorly labeled data set, a classifier can be directly trained from that data in the first place. That means that pseudo-labeling and actually training a classifier are not very different tasks conceptually. As we have acknowledged, constructing and training a classifier with a few labels is a hard task. This implies that pseudo-labeling is also not going to be an 'easy' automated solution.

On the other hand, the best results can be achieved by hiring a domain expert for the specific problem we are trying to solve and letting him/her label the whole data set manually. However, this approach is very time-consuming.

A trade-off will be to use an algorithm (e.g. clustering) that will assign the labels automatically and then include the domain expert for fixing the labels where the algorithm might have made mistakes. For this however, we need a special tool to support the domain expert, which will indicate where the errors are, and give insights into how the model (e.g. classifier) works (i.e. in which regions of the data space the model predictions change, what is the confidence of the model in a specific region, etc.). In this thesis, we consider the development of such a visualization tool that will aid a domain expert in training a classifier. The scope of this thesis is to analyze if the use of a visualization tool helps in building models in semi-labeled data set contexts. Moreover, we want to analyze how the results differ when the visualization tool is used when compared to manually assigning all the labels.

### 1.1.2 Problem requirements

A visualization tool for helping pseudo-labeling as described at the end of section 1.1.1 is developed in this project. In this section, we introduce a list of requirements (R1, ..., R5) that such a tool should comply with.

#### Low dimensional space visualization

Humans are not used to thinking in the  $n$  dimensional spaces where  $n > 3$ . However, in practice, the data points are usually in these high-dimensional spaces. Therefore a critical requirement for such a visualization tool is to represent the data set in a low-dimensional space (i.e. 2D or 3D). We aim to design our tool for optimal user-friendliness, and typically, humans find 2D representations the most intuitive and easy to work with. Thus, we have the first requirement for our visualization tool as follows:

- **R1** Allow data visualization in the low dimensional 2D space

#### Classifier integration

Let  $D = \{(x_i, y_i) \mid i \in \{1, \dots, N\}\}$  be our data set. This data set is a subsample of the data space  $\mathcal{D}$  (i.e.  $D \sim \mathcal{D}$ ). The goal of the end-user of our visualization tool is to train a classifier  $\mathcal{C}$  that given a point  $x \notin D$  will predict a label  $\hat{y} = \mathcal{C}(x)$  such that  $\hat{y} = y$  where  $(x, y) \in \mathcal{D}$ . Consider two points from our data set  $x_i$  and  $x_j$  ( $i \neq j$ ,  $x_i \neq x_j$ ) and the corresponding labels  $y_i$  and  $y_j$  where ( $y_i \neq y_j$ ). Since the data space  $\mathcal{D}$  is dense, there is a set of points between  $x_i$  and  $x_j$  where the label changes from  $y_i$  to  $y_j$ . The better the classifier  $\mathcal{C}$  embeds this information the better the performance of  $\mathcal{C}$ . These regions of points where a point  $x$  has a label  $y_1$  and a neighbor of  $x$  has a different label  $y_2$  ( $y_1 \neq y_2$ ) are called **Decision Boundaries**. The set of all these decision boundaries for a data set  $D$  together with the **Decision Zones** (i.e. the areas that are delimited by the decision boundaries) is called a **Decision Boundary Map** (DBM).

Starting from the assumption that an end-user of the tool has domain knowledge about the ground truth DBM (i.e. the DBM of  $\mathcal{D}$ ), if the user can visualize the DBM of the classifier  $\mathcal{C}$ , he/she can use it to guide the classifier (i.e. by re-assigning labels) so that the classifier DBM matches the ground truth DBM. From this reasoning, we have two additional requirements for our visualization tool:

- **R2** Display a 2D Decision Boundary Map of a classifier  $\mathcal{C}$  (which is provided by the user)
- **R3** Allow data points re-labeling in 2D space and re-training of the classifier  $\mathcal{C}$

Let  $\mathcal{P}$  be a function that for a given  $n$ -dimensional data point is returning a 2D representation of this point. Requirement **R3** states that the user needs to be able to re-label a 2D representation of an  $n$ -dimensional data point. In order to re-train the classifier after such labeling in the 2D space we need an inverse projection  $\mathcal{P}^{-1}$  from the 2D space to the  $n$ D.

At first sight, this inverse projection function might seem very straightforward. The projection  $\mathcal{P}$  takes an  $n$ D data point from the data set and gives us a 2D data point,  $x_i^{2D} = \mathcal{P}(x_i)$  for  $i \in \{1, \dots, |D|\}$ . Then for the 2D point  $x_j^{2D}$  the  $n$ D counterpart is just  $x_i$ . This gives us only a discrete representation of  $\mathcal{P}^{-1}$ . However, recall that we are also interested in the points outside of the data set  $D$  in order to compute the DBM (**R2**). Therefore, we need another way to compute  $\mathcal{P}^{-1}$ . In this project, we are presenting some of the methods present in the literature, that can generate such an inverse projection.

### Errors visualization

By reducing the dimensionality of the data we lose information. Consider for instance that one wants to represent an  $N \times M$  image with 2 coordinates in the 2D space. If it was the case that we have such a projection method  $\mathcal{P}$  and an inverse projection  $\mathcal{P}^{-1}$  that when combined gives us the exact initial image (i.e.  $x = \mathcal{P}^{-1}(\mathcal{P}(x))$ ) then this will mean that we need only two numbers for storing any image. Even though this would be wonderful and would have a lot of good implications, so far this is not the case. Therefore, we can assume from the beginning that any projection method we are using is going to have errors. The same reasoning holds for the inverse projection method. The user must be aware of these errors and should be able to visualize them in our tool. Thus, we have the following additional requirements:

- **R4** Display projection errors for each 2D point
- **R5** Display inverse projection errors for each 2D point

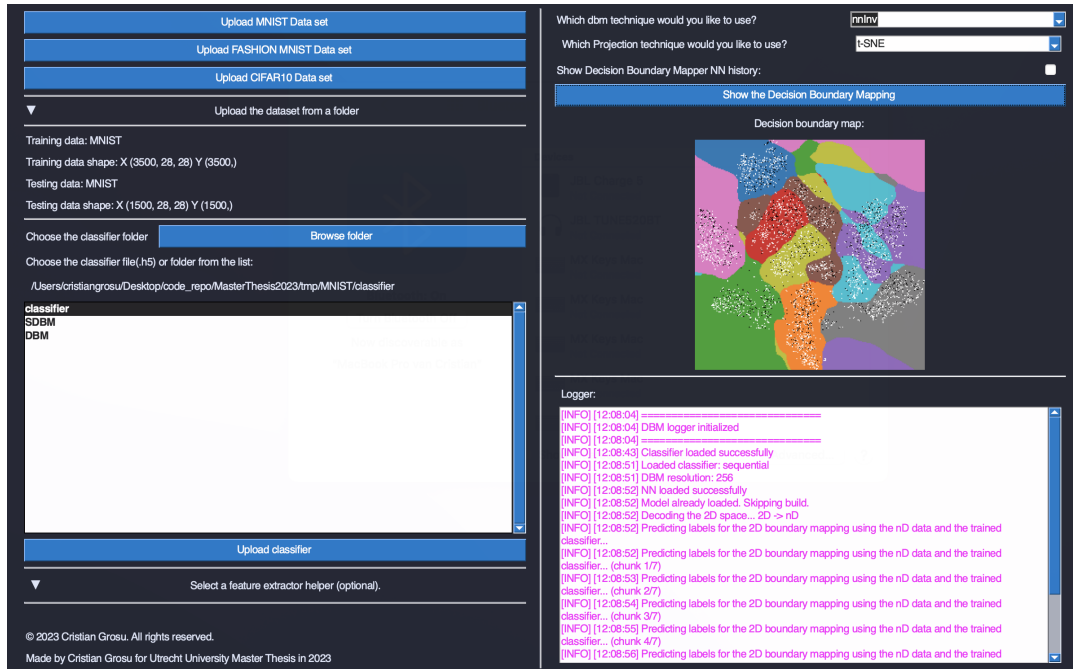
### 1.1.3 Research questions

In this project, we want to study how we can develop an interactive visualization tool (and subsequent API) that takes as input from a user a classifier  $\mathcal{C}$  and a completely pseudo-labeled data set  $D$  and helps the user in correcting the pseudo-labels and improving the classifier performance by providing functionalities that fulfill requirements **R1**, ..., **R5**. Another research question of this study is to assess the utility of such a tool and make a conclusion about how feasible is the proposed hybrid pipeline (i.e. automatic pseudo-labeling followed by manual label correction) in practice.

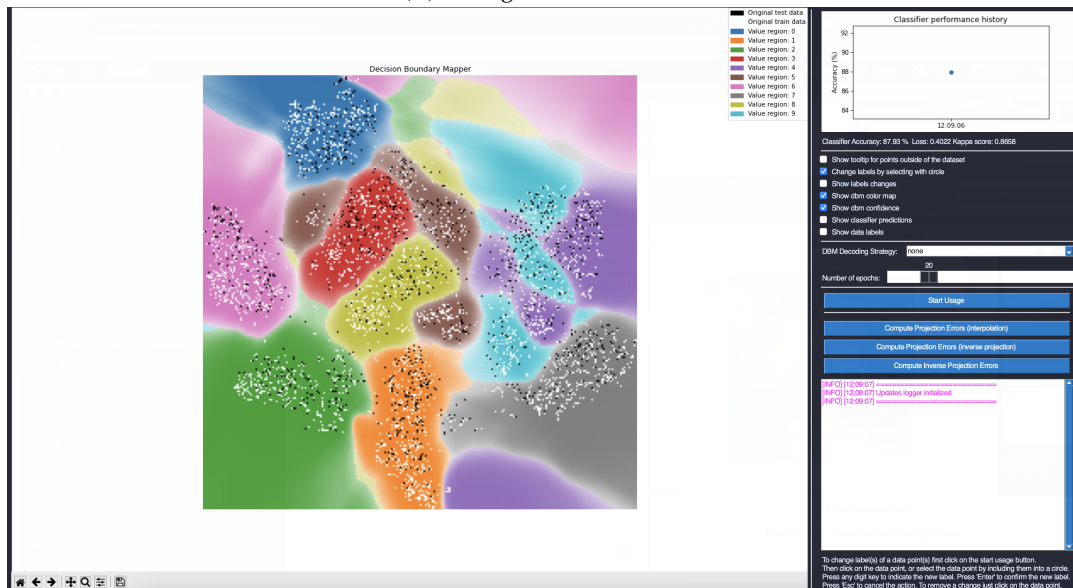
To address these inquiries, we have developed a tool, presented as a Python library named *decision-boundary-mapper*<sup>12</sup>. The core functionalities of this tool, including a graphical user interface (GUI) application (previewed in figure 1.1), are elucidated in the subsequent chapters. These chapters delve into the implementation details of the tool and illustrate how it meets the stipulated requirements (**R1** through **R5**). Furthermore, the library introduces innovative algorithms, such as a novel approach for pixel-based projection error computation, utilized by the visualization tool to fulfill some of the specified requirements. Additionally, the library incorporates novel algorithms aimed at expediting the computation of Decision Boundary Maps (DBMs).

<sup>1</sup><https://pypi.org/project/decision-boundary-mapper>

<sup>2</sup>[Github repository: https://github.com/cristi2019255/MasterThesis2023](https://github.com/cristi2019255/MasterThesis2023)



(A) Configuration window



(B) Visualization tool

FIGURE 1.1: Visualization tool preview

## 1.2 Structure of this thesis

The project is organized as follows: In the current chapter 1, we articulated the rationale behind the utility of a visualization tool in assisting users during classifier training in the context of semi-labeled data sets. We established a set of requirements for such a tool and deliberated on the proposed solution methodology for addressing the identified problem.

In chapter 2 we are presenting the related work present in the literature. In this chapter, we start by discussing some of the most used pseudo-labeling methods, after which we analyze the most known projection methods and compare them. The

available inverse projection methods present in the literature are also presented in this chapter along with techniques for learning both direct and inverse projection. Existing metrics that spot the projection and inverse projection errors are also debated in this chapter.

Chapter 3 presents the solution design of the visualization tool and the related implementation details. This chapter starts by presenting the methods of projecting the data sets in 2D and the construction of the DBM. Then we continue by presenting metrics that can be used by the tool in order to spot direct and inverse projection errors. A series of algorithms that aims to speed up the computation of the DBMs is then presented in the same chapter. Chapter 3 comes also with a section in which we present an overview and a short usage guide for the newly implemented visualization tool.

Chapter 4 exhibits a series of experiments that aims to show to what extent the visualization tool is useful in the context of training a classifier when the data set is semi-labeled. This chapter starts by analyzing the novel algorithms that are supposed to speed up the computation of the DBM. Following, we present a series of experiments where users are asked to train a classifier with the visualization tool. In this chapter, we answer the core questions of this project, namely: "Is a visualization tool helpful in this kind of task?".

Finally, in chapter 5 general discussions and conclusions about this project are listed along with future directions of research and development.





## Chapter 2

# Related Work

In this chapter, we present the literature-related works to the problem introduced in chapter 1 as well as its requirements 1.1.2. This chapter is structured as follows, in section 2.1 we make a short introduction regarding pseudo-labeling methods in the semi-supervised learning context and present ways of automatic labeling. In section 2.2 we present a study that shows the advantages of low-dimensional space visual representations of high-dimensional data in the context of getting insights about a classification system. Several studies about projection techniques are presented in the same section. Section 2.3 presents a paper that introduces a way of evaluating projection errors. In 2.4, we present a paper that uses these projection methods and introduces a methodology for how an inverse projection can be generated depending on the direct projection method. Section 2.5 presents a paper that introduces a method for computing inverse projection errors. A way to generate both projection and inverse projection using the data set is presented in section 2.6. Section 2.7 exhibits papers that address ways of visually presenting the DBMs. This chapter ends with section 2.8 in which we summarize the current knowledge base state, and identify and list the gaps that our research aims to address.

## 2.1 Pseudo-labeling Methods in Semi-Supervised Learning

In chapter 1, we already stated that the scope of our visualization tool is to help a user train a model in the semi-supervised learning context (i.e. the data set which is used for training is not completely labeled). As we mentioned the most straightforward way to get a completely labeled data set is to have a domain expert who will label the data points manually based on the domain knowledge. This method is very expensive time-wise. An automatic way of solving this task saves a lot of time. In this section, we present several automatic pseudo-labeling algorithms.

### 2.1.1 Label Propagation

Label Propagation (LP) introduced in paper [20] is an example of a method that aims to label the complete data set given a small subset of labeled data points. The core assumption of the algorithm is that the data points that are close to each other are likely to have the same label. Starting from this assumption the algorithm constructs a graph where each data point is a node and the edges between the nodes represent similarities between the data points. For the data points that are labeled, the algorithm assigns the correct label, for the unlabelled data points arbitrary labels are assigned. Based on the similarities of a data point's neighbors in the graph and the neighbors' labels the algorithm updates the label of each data point. This process is repeated until a maximum number of iterations (defined by the user) is achieved or the number of changes in label assignments between iterations is greater than

a user-defined threshold. This easy-to-implement algorithm can handle non-linear decision boundaries.

### 2.1.2 Optimal Path Forest

Another algorithm that starts with the same assumption as the Label Propagation algorithm was introduced in paper [1]. The Optimal Path Forest (OPF) algorithm constructs a decision tree over the data where nodes represent clusters of data samples and edges represent the similarity between clusters. A cost function that aims to maximize the separability between classes is used to determine the best split at each node. After the tree is constructed, the labels of the labeled samples are propagated through the tree. The labels of the unlabelled samples are assigned based on the closest labeled samples. The algorithm can be extended to use an ensemble of such decision trees and use a majority voting system for instance to assign the label of an unlabelled sample.

OPF is more robust to noise and outliers in the data than the LP algorithm (see 2.1.1). The reason is that OPF aims to maximize the separability between classes rather than simply propagating the labels based on local similarities. On the other hand, OPF is more expensive computationally than the simple label propagation algorithm since it involves constructing and traversing one or more decision tree(s). OPF has been shown to achieve competitive or even state-of-the-art performance on a wide range of data sets and classification tasks, especially when the number of labeled samples is small or when the data is noisy or high-dimensional.

### 2.1.3 Deep Feature Annotation

Deep Feature Annotation (DeepFA) [3], [2] is a more complex pseudo-labeling technique that is built on top of the OPF algorithm. This method iteratively repeats three steps: 1) deep feature learning, 2) feature space projection, and 3) pseudo-labeling. In the first step, the DeepFA uses a convolutional neural network in order to extract features of the raw data samples. Complex CNNs like VVG-16 can be involved by leveraging the CNNs' ability to transfer knowledge and the small set of initially labeled samples in the first iteration. The features extracted in the first step are then projected into the 2D space by means of a dimensionality reduction method (i.e. t-SNE, see 2.2.3). In the last step a version of the OPF algorithm, namely OPFsemi, is used for pseudo-labeling on the low dimensional space projection. In the same step, each un-labeled sample is assigned a confidence value which represents the confidence of the OPF algorithm in the label that has been assigned. All data samples for which the labels were assigned with confidence values above a certain threshold are then used for the CNN (i.e. feature extractor) re-training in the next iterative loop. This pseudo-labeling technique outperforms the OPF algorithm, due to the additional feature learning and feature space projection steps. Moreover, this method requires a minimal amount of annotated training samples per class.

In order to keep the generality of our visualization tool we do not embed any of the labeling methods in the tool. The user needs to choose a preferred pseudo-labeling method and provide the tool with the complete pseudo-labeled data set (see implementation details in chapter 3). In our experiments (i.e. chapter 4) we mostly use the DeepFA method due to the arguments provided above.

## 2.2 Projection Methods

Projecting the data from an  $n$ D high dimensional space to a low dimensional space is known in the literature as **Dimensionality Reduction**. A function  $\mathcal{P}$  that takes a high dimensional data point and converts it to a low dimensional representation is called in domain literature a **Projection method**. The goal of a projection method is to reduce the number of features in a data set whilst preserving its inherent structure.

### 2.2.1 Projections as Visual Aids for Classification Systems

Rauber et al. [15], examined a hypothesis that states that the concepts of a visually well-separated low dimensional projection and an easily separable  $n$ D data set (e.g. by means of a classifier) are equivalent. Extending on this hypothesis this article proposes a visual representation of the data sets based on dimensionality reduction in order to give feedback on classification efficacy. Furthermore, the study demonstrates the utility of such projections in tasks involving the identification of outliers and the identification of regions containing a mixture of class labels. In summation, this research underscores the efficacy of dimensionality reduction to low-dimensional spaces, such as 2D, thereby presenting a compelling alternative for high-dimensional data visualization. Moreover, this paper serves as an additional argument for the requirement **R1**.

Some of the best-known dimensionality reduction methods available are Principal Component Analysis (**PCA**), Uniform Manifold Approximation and Projection (**UMAP**), and t-distributed Stochastic Neighbor Embedding (**t-SNE**), which we describe next.

### 2.2.2 Principal Component Analysis

PCA method was introduced by [14], the main idea of this method is to project the high dimensional data by finding the eigenvectors of the data's covariance matrix. The eigenvectors represent the directions in which data has the greatest variance, the corresponding eigenvalues representing the magnitude of this variance. Before computing the eigenvectors and the eigenvalues, data is first standardized (e.g. using z-score or min-max normalization). This is done because we want to analyze what features have the biggest variance. We are more interested in the relative values of the variance rather than in the absolute ones. If features have different scales without normalizing the data the comparison of the variances might be misleading. After the eigenvectors are sorted based on the eigenvalues in decreasing order the PCA projects the original data onto the principal components by rotating the data to align with these principal components. This is done by multiplying the standardized data (in matrix form) with the matrix of sorted eigenvectors. The resulting data has the same number of data points, however, the number of features is equal to the number of principal components selected, which can be set as a parameter.

### 2.2.3 t-distributed Stochastic Neighbor Embedding

T-SNE method [11] is a nonlinear dimensionality reduction technique. The scope of t-SNE is to preserve the pairwise similarity relationships between data points. This projection method models the high dimensional similarities between data points using a student t-distribution and then finds a low dimensional representation that preserves these similarities as closely as possible. The algorithm of the t-SNE projection method starts by computing pairwise similarities between all pairs of high

dimensional data points. The similarities are typically based on a Gaussian kernel, which measures the similarity between two points based on their distance in the high dimensional space. The pairwise similarities are then converted into conditional probabilities, which represent the likelihood of one point being selected as a neighbor of another point, given their similarities. The conditional probabilities are calculated using the student's  $t$ -distribution, which has a heavier tail than the Gaussian distribution and is better suited for capturing nonlinear relationships in the data. A low-dimensional representation of the data is then created (e.g. at random).  $t$ -SNE then iteratively adjusts the low dimensional representation to minimize the difference between the conditional probabilities in the high dimensional space and those in the low dimensional space. Gradient descent can then be used to adjust the position of each low-dimensional point based on the similarity to other points.

This optimization process is controlled by several hyperparameters, one of which is perplexity, which controls the balance between local and global structure in the low dimensional representation. In other words, perplexity is defined as the effective number of neighbors of each data point in the high dimensional space. A low perplexity value will force  $t$ -SNE to focus on local structure, preserving only nearby neighbors and producing a compact and clustered low-dimensional representation. A high perplexity value, on the other hand, will allow  $t$ -SNE to capture more global structure, preserving both nearby and distant neighbors and producing a more spread-out and continuous low-dimensional representation. Different datasets may require different levels of focus on local versus global structure, thus, the perplexity parameter is data-dependent. This projection method is often used for clusters and pattern identification in the data.

#### 2.2.4 Uniform Manifold Approximation and Projection

UMAP method [12] shares some concepts with  $t$ -SNE but uses a different mathematical framework. Similar to  $t$ -SNE, UMAP preserves the local structure of the data by modeling the distribution of nearest neighbor distances in the high dimensional space and then finding a low dimensional space that preserves this distribution.

UMAP starts by converting the data into a weighted graph representation, where each data point is a node and the weights represent the similarities between the 2 points. This graph is then converted into a fuzzy topological representation. For each data point, UMAP identifies the  $k$ -nearest neighbors, these neighbors are then used to construct a "simplicial complex" (a mathematical structure composed of vertices, edges, triangles, and higher-dimensional triangles called simplex/simplices). The fuzzy topological representation is constructed by first assigning each simplex in the simplicial complex a fuzzy degree of membership. This degree of membership is a measure of how closely the simplex resembles a uniform distribution of points. Simplices that contain data points that are more similar to each other will have a higher degree of membership, while simplices that contain data points that are more dissimilar will have a lower degree of membership. UMAP then constructs a low-dimensional representation of the data and uses a process called graph layout to embed the fuzzy topological representation into the low-dimensional space. This process involves iteratively adjusting the position of each point in the low-dimensional space based on its similarity to other points in the high-dimensional space.

The fuzzy topological representation allows UMAP to capture both local and global structure in the high dimensional data, as it incorporates information about the nearest neighbors of each data point as well as the overall distribution of points

in the data. This allows UMAP to create a low-dimensional representation that preserves both the local and global structure of the data.

UMAP also incorporates a balancing parameter that controls the trade-off between preserving the global structure and preserving the local density of the data points. This parameter allows the algorithm to adjust the emphasis placed on preserving different types of structures in the data.

### 2.2.5 Projection methods comparison

All the projection methods presented so far have advantages and disadvantages. PCA is a simple and fast algorithm, useful for visualizing the overall structure of the data and identifying patterns and correlations between variables. On the other hand, PCA may not be very effective in identifying clusters of similar data points compared to t-SNE and UMAP. Moreover, PCA is very sensitive to outliers in the data since such outliers might determine the principal components.

t-SNE is able to preserve the local structure of the data. Furthermore, t-SNE can be used to identify outliers in the data, which can be difficult with PCA and UMAP. However, t-SNE is computationally intensive and may require tuning of hyperparameters (e.g. perplexity) which might be time-consuming.

UMAP technique has the advantage of scalability over t-SNE and PCA. UMAP also tends to better preserve the global structure of data than t-SNE. Parameter tuning of UMAP is on the other hand a disadvantage when compared to PCA. UMAP and t-SNE are both stochastic algorithms which means the results of the projection might differ from one run to another of the algorithms, PCA on the other hand is deterministic.

Each projection method has its own goals and the usage of one projection method over another is to be decided by the user depending on the data and the insights he/she wants to get from it. In our experiments, we use all these projection methods.

## 2.3 Projection Errors

In general, none of the projection methods will be able to fully preserve all the information and the structure in the original high-dimensional data. Hence, we need metrics for projection methods that will tell us how much such a technique can be trusted. In other words, evaluating how good the results given by the method for the task at hand are. In paper [18] two metrics for the evaluation of projection techniques were proposed.

### Trustworthiness

The first metric called *Trustworthiness* is defined as follows:

$$T(k) = 1 - \frac{2}{nk(2n - 3k - 1)} \sum_{i=1}^n \sum_{j \in U(i,k)} [r_{ij} - k] \quad (2.1)$$

where  $n$  is the number of data points,  $k$  is a parameter that indicates the number of neighbors we take into account,  $U(i, k)$  is the set of  $k$  nearest neighbors to the point  $x_i \in D$  in the high dimensional space,  $r_{ij}$  is the rank of the distance between points  $x_i$  and  $x_j$  in the high dimensional space.

Trustworthiness measures the proportion of high dimensional pairwise similarities that are preserved in the low dimensional projection up to a certain rank  $k$ . A

perfect projection would have  $T(k) = 1$  for all values of  $k$ , indicating that all pairwise similarities in the high dimensional space are preserved in the low dimensional space.

### Continuity

Another measure that counts the proportion of pairwise distances in the low dimensional space that are also present in the high dimensional space up to a certain rank  $k$  is called *Continuity*. The continuity metric is defined as follows:

$$C(k) = 1 - \frac{2}{nk(2n - 3k - 1)} \sum_{i=1}^n \sum_{j \in V(i,k)} [\hat{r}_{i,j} - k] \quad (2.2)$$

where  $\hat{r}_{i,j}$  is the rank of the distance between the projections of points  $x_i$  and  $x_j$  in the low dimensional space, and  $V(i,k)$  is the set of  $k$  nearest neighbors to the point  $\mathcal{P}(x_i)$  in the low dimensional space. A perfect projection would have  $C(k) = 1$  for all values of  $k$ , indicating that the low-dimensional projection accurately represents the geometry of the high-dimensional space.

Trustworthiness and continuity capture both the preservation of pairwise similarities and the accuracy of the overall geometry. Although these metrics are widely used for the evaluation of projection techniques such as UMAP and t-SNE the ideas that these metrics capture can be applied to evaluate any projection method. Paper [8] presents a survey that uses metrics including the ones presented in order to compare a wide body of projection techniques on several data sets with different characteristics.

These metrics describe the projection method on a global level rather than on a local level, in the sense that by looking only at one particular data point we can not tell how good or bad the projection is. In section 3.2 we start from the definitions introduced in the current section and adjust the equations 2.1 and 2.2 in order to assess how good a projection method is for each individual data point.

## 2.4 Inverse Projection Methods

Inverse projection methods aim to reconstruct the high dimensional data from their low dimensional projections. This problem of inverse dimensionality reduction was studied in several papers, most notably: [17], [9].

### Inverse Linear Affine Multidimensional Projection

In the paper [17], the authors introduce a technique known as Inverse Linear Affine Multidimensional Projection (**iLAMP**). This method computes an affine transformation  $f(\cdot)$  as follows. For a given low-dimensional point  $p$ , the method identifies the nearest  $k$  neighbors within the set of instances in the low-dimensional space obtained through the application of the projection method on the data set. Subsequently, the corresponding high-dimensional counterparts are determined. The affine transformation is designed in a manner that optimizes the distances between the high-dimensional point  $f(p)$  and the high-dimensional counterparts of  $p$ 's neighbors, aiming for these distances to closely mirror the distances between point  $p$  and its neighbors in the low-dimensional space. The formula of  $f(\cdot)$  is then given by solving a system of equations.

Regardless of the strong mathematical foundation, this technique has some disadvantages. For instance, the hyperparameter  $k$  (i.e. the number of neighbors),

which is determining the inverse projection results. The value of this hyperparameter is problem-dependent and needs to be carefully chosen. Another disadvantage is the fact that this affine transformation is constructed based on the geometrical coordinates in the high and low dimensional spaces, which makes this inverse projection method not very suitable when used in combination with such projection techniques like t-SNE that are focused more on preserving neighborhoods rather than preserving geometrical distances.

### Inverse Neural Network Projection

In the paper [5] the authors propose to use neural networks as projection methods for dimensionality reduction. The core idea is to train a deep neural network based on samples drawn from a given data universe, and their corresponding low-dimensional projections, computed with any available projection technique (e.g. t-SNE, UMAP, PCA, etc.). Then the network is used to infer projections of any data set from the same universe. The biggest advantage of such Neural Network Projection (NNP) is that it can directly handle out-of-sample data. In other words, it can project data outside of the initial data set. Another advantage is that such a projection is less computationally expensive and has no complex-to-set user parameters.

Based on the idea of NNPs, in paper [7], the authors proposed to use neural networks for inverse projections as well. The proposed approach is to train an Inverse Neural Network (NN<sub>inv</sub>) that will map the low-dimensional points to the high-dimensional points. The network operates as a decoder, taking the low-dimensional embedding derived through the utilization of a projection method of the original data as input, and its primary function is to reconstruct the original data. The goal of NN<sub>inv</sub> is to minimize the disparity between the original high-dimensional data set and the reconstructed data, which is derived by passing the low-dimensional embedding through the NN<sub>inv</sub>.

On one hand, this method requires the developer to design a neural network. On the other hand, this approach does not require any hyperparameters and is less computationally expensive when compared to the iLAMP method, in the sense that once the NN<sub>inv</sub> network is trained we can just use it to compute the high-dimensional representation of a low-dimensional point straight away without the need to search for neighbors in the data set.

The biggest advantage of the NN<sub>inv</sub> as the NNP is that they can handle data outside of the data set. We will see in the next chapters that this property for an inverse projection method is crucial for the computation of DBMs. Therefore throughout this project, we use NN<sub>inv</sub> in combination with direct projections introduced previously (t-SNE, UMAP, PCA).

## 2.5 Inverse Projection Errors

Espadoto et al. [9] presented a metric for evaluating the inverse projection errors at each point in the data space. This is done by computing a gradient image, which is a 2D scalar field representing a pseudo total derivative of the inverse projection  $\mathcal{P}^{-1}$

computed using central differences as follows:

$$\begin{aligned}
 D_x(p) &= \frac{\mathcal{P}^{-1}(p + (w, 0)) - \mathcal{P}^{-1}(p - (w, 0))}{2w} \\
 D_y(p) &= \frac{\mathcal{P}^{-1}(p + (0, h)) - \mathcal{P}^{-1}(p - (0, h))}{2h} \\
 D(p) &= \sqrt{\|D_x(p)\|^2 + \|D_y(p)\|^2}
 \end{aligned} \tag{2.3}$$

where  $p$  is the 2D projection of a data point and  $w, h$  is the pixel's width and height respectively. The regions with large gradient values illustrate where the high-dimensional distance is changing most rapidly with respect to the changes in low-dimensional distance. In other words, these regions indicate where the inverse projection method distorts the low-dimensional space.

Observe that this metric is not dependent on any data set. We will see why this is important in section 3.2 where we present how this metric is used by our visualization tool.

## 2.6 Self-Supervised Neural Projection

So far we presented a list of projection methods  $\mathcal{P}$  and a way to compute inverse projections  $\mathcal{P}^{-1}$ . Given a data set  $D$  the user has to choose one of the projection methods and then based on the 2D projection  $D_{2d} = \mathcal{P}(D)$  learn the inverse projection  $\mathcal{P}^{-1}$  such that  $\mathcal{P}^{-1}(D_{2d}) = D$ . This approach has some disadvantages, namely, the same projection method might give better or worse results depending on the data, and the majority of these projection methods have hyper-parameters that have to be chosen by the user (e.g. perplexity parameter in t-SNE).

In the paper [6] the authors propose a data-driven way for generating both  $\mathcal{P}$  and  $\mathcal{P}^{-1}$ . The core idea is that when composing the projection and inverse projection functions we must get the identity function, i.e.  $\mathcal{P}^{-1}(\mathcal{P}(x)) = x$ . For this purpose, an auto-encoder neural network can be used where the encoder part is responsible for the dimensionality reduction and the decoder part takes the role of the inverse projection. Such methods of using auto-encoders for dimensionality reduction are not new. However, using only an auto-encoder on its own has been noticed to yield worse visual cluster separation than popular projection methods such as t-SNE and/or UMAP. In their paper, the authors propose to use such an auto-encoder architecture but with an additional layer after the decoder part. This approach is called Self-Supervised Neural Projection (**SSNP**). Based on the data labels or pseudo-labels the last layer of the network is used to improve the cluster separation. This additional layer will force the projection method (i.e. the encoder) to group all data samples with the same labels into the same clusters (see figure 2.1).

In the context of our problem where the labels of data samples might be misleading the SSNP approach might be error-prone and might confuse the user. On the other hand, the cluster separation per class might help the user in taking faster decisions about the correct labels of the data samples. Therefore, in our experiments of chapter 4 we involve both the vanilla autoencoder and the SSNP approaches.



## 2.7 Image-based visualization of Classifier Decision Boundaries

In section 1.1.2 one of the requirements of our visualization tool is to display to the user a 2D Decision Boundary Map given a classifier  $\mathcal{C}$ . A DBM is a graphical representation that shows how the machine learning model separates data points into different classes based on their input features. Once we have the low dimensional embedding of the data and an inverse projection function  $\mathcal{P}^{-1}$  constructing a graphical representation is a straightforward task. An image-based representation is the most understandable for a human user. The easiest way to generate such an image is to choose a width and height and then for each pixel take its coordinates  $(x, y)$ , re-scale them with respect to the 2D embedding of the data to obtain  $(\hat{x}, \hat{y})$ , and then obtain the classifier predicted label  $l = \mathcal{C}(\mathcal{P}^{-1}(\hat{x}, \hat{y}))$ . For each class label, we assign all the pixels with that label a certain color. The colored image is a representation of the classifier decision boundary map.

In paper [16] the authors use an approach as the one described before and two inverse projection methods: iLAMP [17] and NNinv [9] in order to generate image-based visualizations for classifiers' decision boundaries. In [13] the authors introduce a technique called **SDBM** (Supervised Decision Boundary Maps). This method uses SSNP [6] in order to generate image-based representations of classifiers' decision boundaries.

Figure 2.1 presents two examples of the DBM for the same classifier  $\mathcal{C}$  and the MNIST data set [10]. Notice how the SDBM + SSNP approach is giving smoother decision boundaries in 2D, whilst the NNinv together with the t-SNE projection is giving noisier results. The simple DBM approach allows us to use any projection method, whereas the SDBM approach enforces us to use the "built-in" projection method (i.e. the encoder part). From this point of view, the DBM approach proposed in [16] gives us more flexibility. On the other hand, before using the DBM approach we need a good inverse projection method. Designing or choosing such a method might be challenging. Thus, from this point of view, SDBM is better because it generates both the projection and inverse projection together.

In both papers [16] and [13] the authors are building the DBM representation pixel-based, in the sense that each 2D pixel is projected into the high-dimensional space, then the classifier is used to assign each pixel with a color based on the predicted label. The complexity of this approach increases along with the resolution of the desired DBM image. In this project, we analyze several ways of optimizing the DBM computation.

## 2.8 Conclusions

In this chapter, we presented several works from the literature regarding pseudo-labeling methods, direct and inverse projection methods along with metrics for spotting the errors, and methods for the construction of DBMs.

In section 2.2.5 we discussed the advantages and disadvantages of each of the presented direct projection methods. We saw that the choice of technique should be driven by the task the user is trying to solve. The same holds for the inverse projection methods. Section 2.7 presented a discussion about the pros and cons of using the SDBM technique over the technique that generates the DBM only based on the inverse projection. We could not identify beforehand what is the best method or

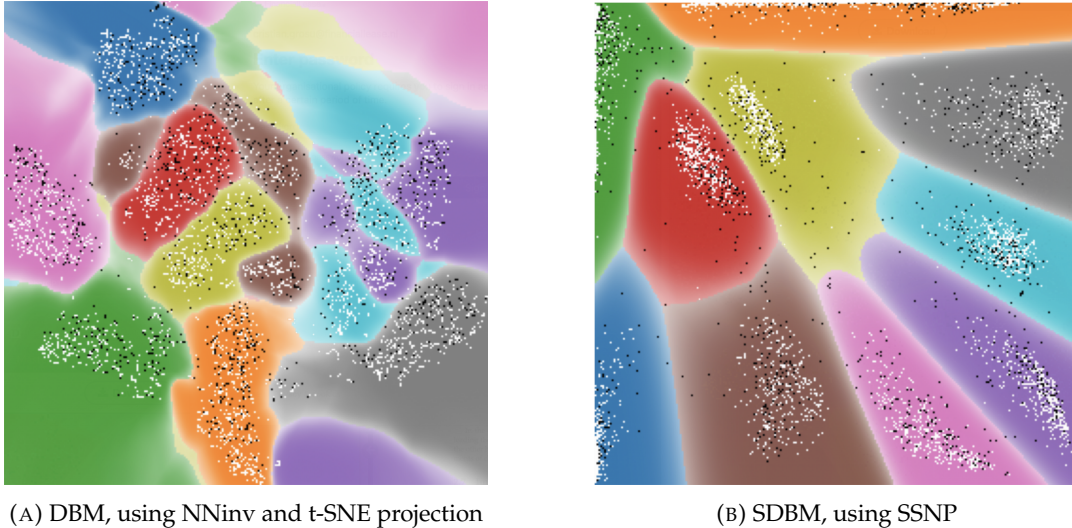


FIGURE 2.1: DBM vs SDBM.

Data set: MNIST.

White and black pixels represent the data samples

combination of methods in general. Therefore, in our work, we consider comparing different alternative methods for the same single task.

We observed that the current way of pixel-based DBM computation is not very scalable in the sense that for high-resolution DBM images, the computation run times might increase considerably, which can be a problem for visualization systems that need to show fast feedback to the user. Thus, in this project, we want to fill this gap by providing new and faster ways of DBM computation.

We could not identify any study in the current literature that incorporates user interaction via a visualization tool in the pipeline of improving/correcting the pseudo-labels generated by an automatic algorithm.

## Chapter 3

# Solution Design

In this chapter, we present implementation details of our visualization tool that aims to fulfill the requirements from section 1.1.2:

- **R1** Allow data set visualization in 2D space
- **R2** Generate and display a Decision Boundary Map of a user-provided classifier  $\mathcal{C}$
- **R3** Allow data points re-labeling in 2D space and re-training of a classifier
  - **R3.1** Allow data points re-labeling in 2D space
  - **R3.2** Allow classifier re-training with re-labeled points
- **R4** For the projection method in use display projection errors for each 2D point
- **R5** For the inverse projection method in use display inverse projection errors for each 2D point

Our visualization tool requires two inputs from the user, namely: the data set  $D$  and the classifier  $\mathcal{C}$  as shown in figure 3.1.

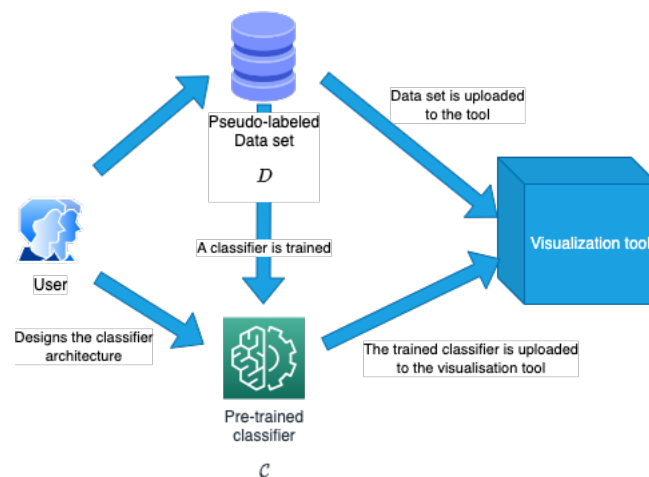


FIGURE 3.1: Pipeline of uploading data set and classifier to the visualization tool

Once this input is provided, the tool must be able to generate the 2D projection of the data set and the DBM of the classifier as shown in the pipeline presented in figure 3.2. In section 3.1 of this chapter, we present implementation details regarding the data points projection and the computation of the DBM image (**R1**, **R2**).

In section 3.2 we present methods for computing projection and inverse projection errors in order to fulfill requirements R4 and R5. Section 3.3 gives implementation details about how the visualization tool allows the data points re-labeling and classifier retraining (R3.1, R3.2). Section 3.4 presents a list of algorithms that our visualization tool can use to speed up the computations for generating the DBM. Section 3.5 presents an overview of the visualization tool and a short usage guide. This chapter ends with section 3.6 in which we analyze how well our proposed solution design matches the problem requirements.

### 3.1 Projecting the data set and generating the DBM

In this section, we present implementation details about the projection methods the tool supports and the inverse projection methods. Furthermore, we explicitly present an algorithm for the computation of the DBM images.

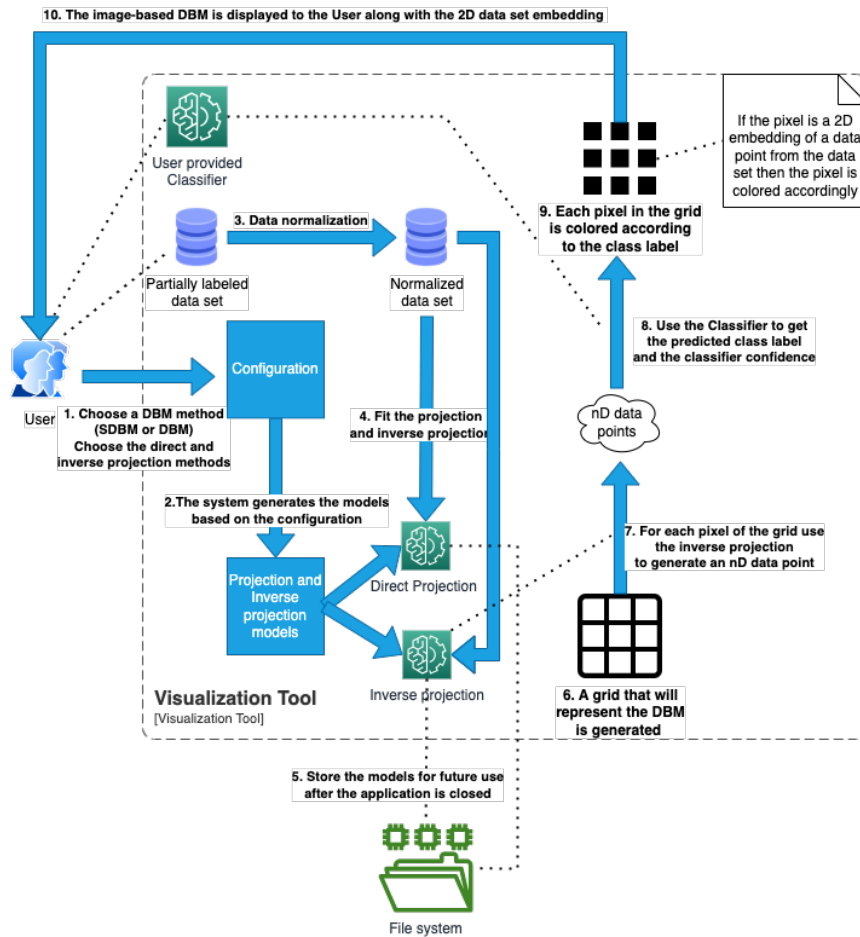


FIGURE 3.2: DBM and 2D data set embedding computation and visualization pipeline (R1 and R2)

#### 3.1.1 Projection methods implementation details

As we saw in section 2.2 there exists a variety of projection methods that allow getting a 2D representation of  $n$ D data. In order to create a visualization tool that is as general as possible we allow the user to choose from one of three built-in projection

methods: t-SNE, UMAP, and PCA. The user can also provide the 2D embedding of data obtained by any other possible projection method.

We use the t-SNE and PCA algorithms from the sklearn Python library. The perplexity in the t-SNE algorithm is set to 30, all the other parameters are using the defaults from the library. For the UMAP projection method, we are using a specialized Python library called "umap". All the parameters are using the default values from that library. We allow for 2 more built-in projection methods, namely by using the encoder of an Autoencoder neural network or an SSNP network (see section 2.6).

### 3.1.2 Inverse projection methods implementation details

In this section, we give details about how we construct our inverse projection methods.

#### Data normalization

We would like to have all the features of our  $n$ D samples in the same range so that all features contribute equally to all the operations we perform on the data. Moreover, some projection and inverse projection methods (especially when using deep learning) need the data to have values in the range  $[0,1]$ . For this purposes we normalize each data sample  $d = (x_1, \dots, x_n) \in D$  as follows:

$$(\hat{x}_1, \dots, \hat{x}_n) = \left( \frac{x_1 - \min_{i \in \overline{1,|D|}} x_{i,1}}{|\max_{i \in \overline{1,|D|}} x_{i,1} - \min_{i \in \overline{1,|D|}} x_{i,1}|}, \dots, \frac{x_n - \min_{i \in \overline{1,|D|}} x_{i,n}}{|\max_{i \in \overline{1,|D|}} x_{i,n} - \min_{i \in \overline{1,|D|}} x_{i,n}|} \right) \quad (3.1)$$

Furthermore, we want to show the user a 2D plot of size  $W \times H$  (i.e. the DBM image), therefore we can use the same normalization to  $[0,1]$  as before and apply it to the 2D projected data  $D_{2d}$ . Converting the data such that the  $x$  coordinates are in the range  $[0,W]$  and  $y$  coordinates are in the range  $[0,H]$  reduces to simply multiplying the normalized projection data to  $W$  and  $H$  respectively. By abusing the notation throughout this thesis we notate with  $D$  the normalized data set and with  $D_{2d}$  the normalized 2D representation of the data set  $D$ .

#### NNinv

One of the goals of our visualization tool is to visualize the Decision Boundary Map for a given classifier  $\mathcal{C}$ . The DBM can be represented by an image where each pixel is colored according to the classifier label (see section 2.7). For the pixels that represent samples from the data set, we can use the classifier to get the label, however not any pixel represents a data sample. Therefore, we need an inverse projection method  $\mathcal{P}^{-1}$ . To learn the inverse projection method we can use NNinv (see section 2.4) which aims to decode a pair of 2D coordinates into an  $n$ D feature vector. The architecture of the NNinv is presented in figure 3.3. The size of the last layer in this architecture is determined by the data shape, i.e. the size is the data dimension  $n$ . In figure 3.3 the size of 784 is taken from the MNIST data set where each data point is a  $28 \times 28$  gray-scale image. In case another data set is used the value of  $n$  might be different which means that the last layer of our decoder will have more or fewer neurons depending on the value of  $n$ . The rest of the architecture stays the same regardless of the data set in use. Observe that the architecture of our network is quite simplistic, and the higher the value of dimensions  $n$  in the high-dimensional space the worse the network will perform. The more complex the network is the more time is required for training. We want to achieve an optimal trade-off between

the training time and the ability to generate features in a relatively high-dimensional space. Therefore, we choose exactly this type of architecture.

**Activation functions** We used a ReLU activation function for all hidden layers except the last one. Since we normalize the data set  $D$  so that each feature is in the range  $[0,1]$  we use the sigmoid activation function for the last output layer.

**Weights initialization** In all the layers we use the *HeUniform* kernel initializer, the bias parameter is set initially to value 0.01.

**Regularization** The first dense layer uses an L2 regularization penalty with a regularization constant set to 0.0002.

**Training and loss function** We train our network for 300 epochs (with early stopping strategy applied), as a loss function we use mean squared error (MSE).

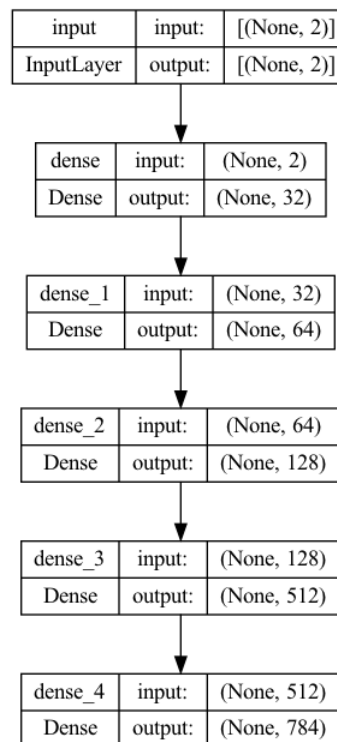


FIGURE 3.3: NNinv Architecture

### Vanilla Autoencoder

Our visualization tool provides the user the option to use an autoencoder in order to generate projection and inverse projection functions. The autoencoder architecture is presented in figure 3.4.

The decoder part of the autoencoder matches one-on-one to the NNinv architecture, we use the same activation functions and the same methods for weights initialization. In the encoder part of the autoencoder, all the dense layers use the ReLU activation function except the last one which uses sigmoid as the activation function. The first dense layer uses an L2 regularizer with the constant set to 0.0002.

The weights and biases are initialized in the same way as for the decoder and NNinv. The number of training epochs is 300 and the loss function is the mean squared error.

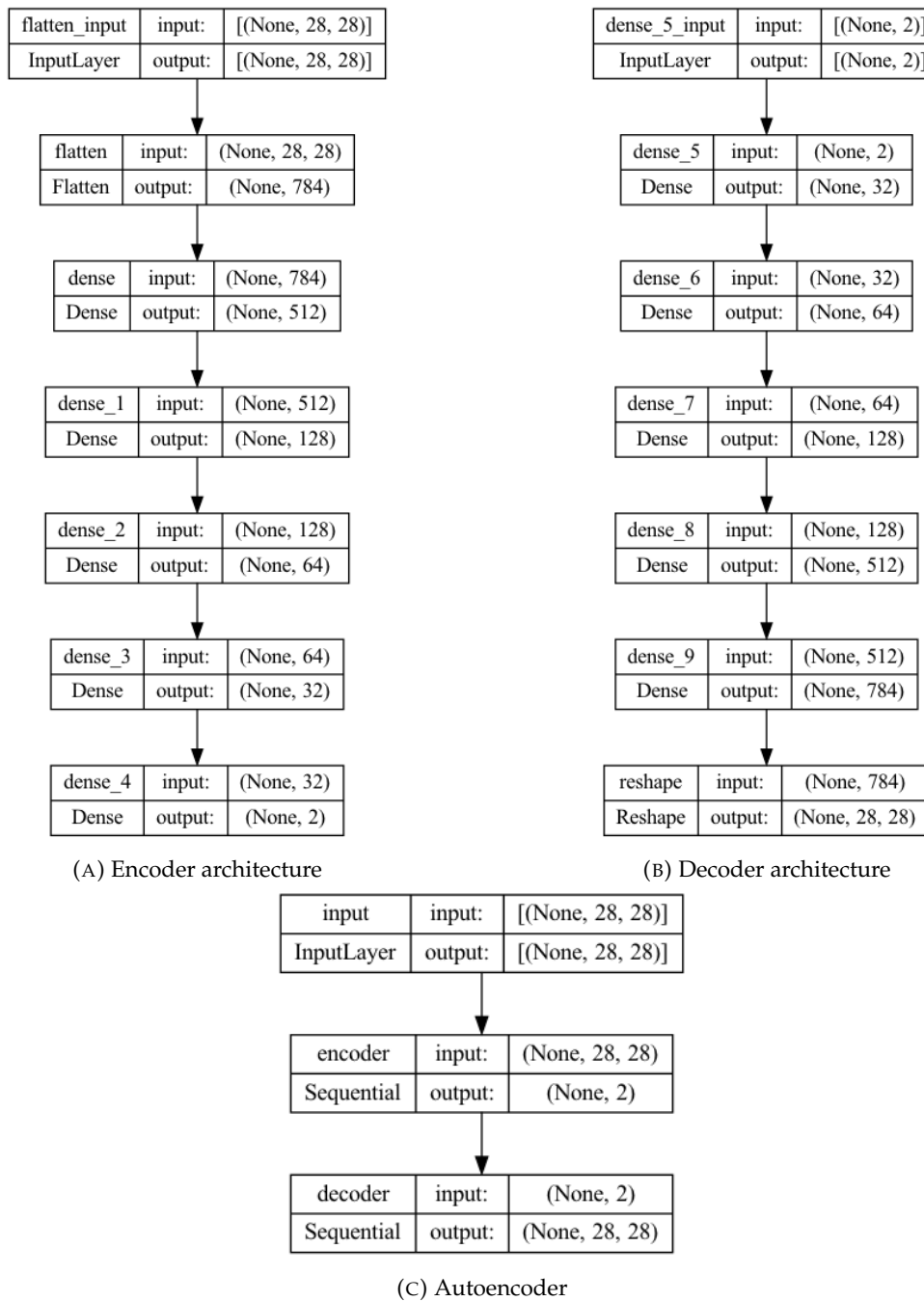


FIGURE 3.4: Autoencoder Architecture

### SSNP Autoencoder

The user can opt for the usage of an SSNP as well. The main difference between the SSNP architecture and the vanilla Autoencoder architecture introduced previously is that SSNP has an additional clustering layer as shown in figure 3.5. Notice how the architecture of the clustering layer is data-dependent. The architecture presented in figure 3.5a is for the MNIST data set. The dense layer in the clustering part of SSNP uses the softmax activation function.

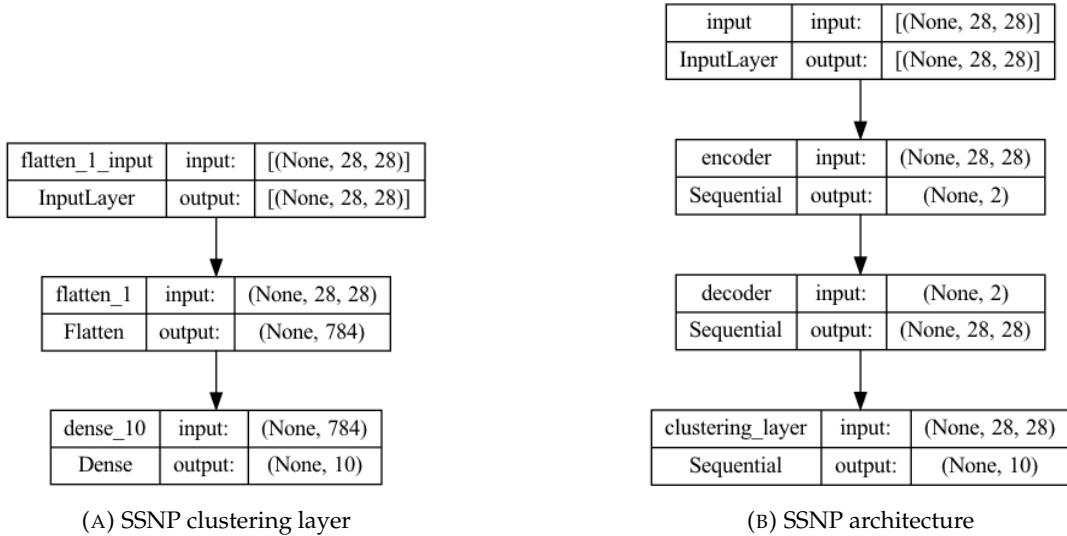


FIGURE 3.5: SSNP Architecture

Whilst the Autoencoder has only one loss function to optimize, SSNP has two optimization targets, namely optimizing the loss function related to the data decoding (i.e. the same as the Autoencoder) and optimizing the loss function related to the data clustering. For the last one, we use the sparse categorical cross-entropy loss. The ability to generate accurate  $n$ D data is more important than the ability of the encoder to generate well-separated clusters. Therefore, we assigned different weights to the loss functions. The MSE loss (which controls the SSNP quality of generating  $n$ D data) was assigned with weight  $w = 1$ , and the sparse categorical cross-entropy loss (which controls the clustering separation) was assigned with weight  $w = 0.125$ .

The NNinv technique works with any chosen projection (see table 3.1). However, in practice, a method that involves SSNP or Autoencoder is easier to train and use. Moreover, the SSNP approach seems to give visually smoother decision boundaries (e.g. see figure 2.1). Our visualization tool allows the user to make the decision of which combination of projection and inverse projection techniques to use based on the user's needs. All the possible combinations that our tool supports are listed in table 3.1.

Projection method $\mathcal{P}$	Compatible inverse projection method $\mathcal{P}^{-1}$	Supported in our tool
t-SNE UMAP PCA	NNinv	Yes
Vanilla Autoencoder (encoder part)	Vanilla Autoencoder (decoder part)	Yes
	NNinv	No
SSNP (encoder part)	SSNP (decoder part)	Yes
	NNinv	No

TABLE 3.1: Projections and inverse projection methods possible combinations

### 3.1.3 DBM computation algorithm

The most straightforward approach for generating an image representing the DBM is to take each pair of points that represents pixel coordinates of the image and use the inverse projection  $\mathcal{P}^{-1}$  to predict the high-dimensional feature vector which corresponds to these coordinates, then use the user-provided classifier  $\mathcal{C}$  to predict the label of that pixel after which color the pixel in a certain color depending on the label.



Algorithm 1 presents the pseudo-code for this approach. We call such an algorithm **Dummy DBM** because as we will see later on, this straightforward approach is not the most optimal one. The output of algorithm 1 are two matrices of size  $W \times H$ , namely  $dbm$  and  $confmap$ . The former is a matrix of integers that represents the labels assigned by  $\mathcal{C}$  to each pixel of the DBM image.  $confmap$  is a matrix that contains values in the range  $[0,1]$  representing the confidence of the classifier in the assigned label for a specific pixel.

---

**Algorithm 1** Dummy DBM
 

---

**Input:**  $W$  (int),  $H$  (int),  $\mathcal{P}^{-1} : [0, 1]^2 \rightarrow [0, 1]^n$ ,  $\mathcal{C} : [0, 1]^n \rightarrow \mathbb{N} \times [0, 1]$

**Output:**  $dbm \in \mathcal{M}_{W,H}(\mathbb{N})$

$confmap \in \mathcal{M}_{W,H}([0, 1])$

- 1:  $dbm \leftarrow \mathcal{O}_{W,H}$  // Zeros matrix of size  $W \times H$
  - 2:  $confmap \leftarrow \mathcal{O}_{W,H}$
  - 3: **for**  $i \leftarrow 0$  to  $W$  **do**
  - 4:   **for**  $j \leftarrow 0$  to  $H$  **do**
  - 5:      $d \leftarrow \mathcal{P}^{-1}(i/W, j/H)$
  - 6:      $label, confidence \leftarrow \mathcal{C}(d)$
  - 7:      $dbm[i][j] \leftarrow label$
  - 8:      $confmap[i][j] \leftarrow confidence$
  - 9:   **end for**
  - 10: **end for**
  - 11: **return**  $dbm, confmap$
- 

## 3.2 Projection and inverse projection errors

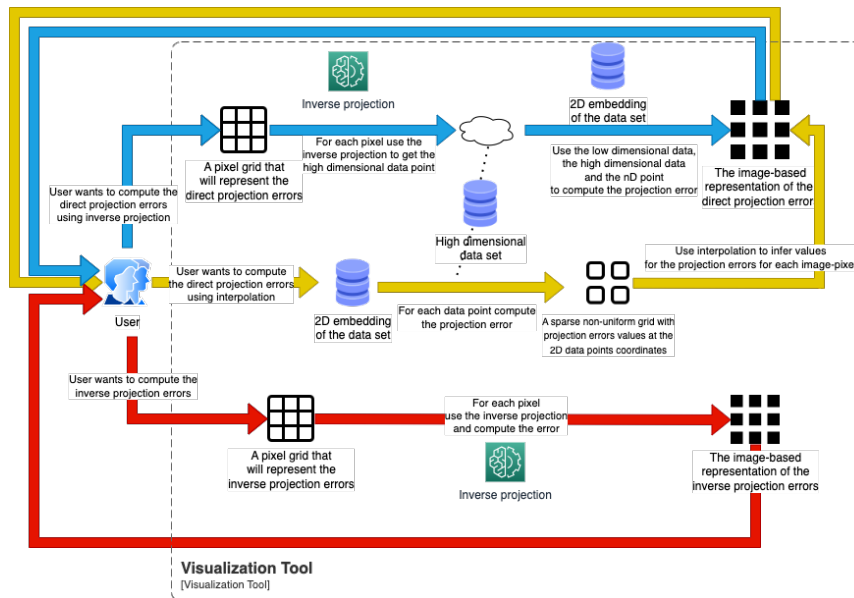


FIGURE 3.6: Errors computation pipeline (**R4** and **R5**)

In sections 1.1.2 and 2.3, we highlighted that converting an  $n$ -dimensional point to a 2D representation results in substantial information loss. We aim to visually present to the user areas where a projection might mislead. Additionally, constructing the

DBM image involves employing the inverse projection, which can also be misleading in specific areas. Hence, this section introduces methods for calculating both projection and inverse projection errors.

The pipeline in which the user interacts with the tool to compute these errors is presented in figure 3.6.

### 3.2.1 Projection errors

#### Local projection errors metrics

In Chapter 2, we presented two metrics, namely **Trustworthiness** and **Continuity**, designed to assess the quality of a projection. It's important to note that equations 2.1 and 2.2 compute a value that characterizes the projection on a global scale. Essentially, these metrics inform us about the overall effectiveness of a projection method but do not pinpoint specific regions where mistakes occur in the 2D space. By "mistakes of a projection" we mean that the order of the neighbors in 2D differs from the order in  $nD$ . In our application, we aim to visually highlight these regions with mistakes for the user. We can build upon the foundational concepts from equations 2.1 and 2.2 by introducing the concepts of local Trustworthiness and local Continuity. For a data point with index  $i$ , we define local trustworthiness and continuity as follows:

$$T_k(i) = 1 - \frac{2}{k(2n - 3k - 1)} \sum_{j \in U(i,k)} [r_{i,j} - k]_+ \quad (3.2)$$

$$C_k(i) = 1 - \frac{2}{k(2n - 3k - 1)} \sum_{j \in V(i,k)} [\widehat{r}_{i,j} - k]_+ \quad (3.3)$$

where  $[x]_+ = x$  if  $x \geq 0$  and 0 otherwise. Observe the distinction between equations 2.1, 2.2 when compared to 3.2 and 3.3 respectively. The shift from global to local involves removing the summation over all data points and the subsequent averaging (i.e. the division by  $n$ ). The local trustworthiness and local continuity metrics tell us how the neighbor's order of a point is preserved in the high and low dimensional space respectively. Since we want to visualize the errors we define the following aggregate  $\epsilon_k(i)$  that combines the local continuity and trustworthiness metrics.

$$\epsilon_k(i) = \frac{((1 - T_k(i)) + (1 - C_k(i)))}{2} \quad (3.4)$$

This metric ranges in  $[0, 1]$ , the closer the value of this metric is to 1 for a point  $x_i \in D$  the worse the neighborhood of  $x_i$  in  $nD$  matches with the neighborhood of  $\mathcal{P}(x_i)$  in 2D. On the other hand, values close to 0 indicate that the neighborhood is preserved well, which means we can trust the 2D representation given by the projection method for the point  $x_i$ .

Observe that these definitions can handle only 2D points that are representations of  $n$ -dimensional data samples. We can not compute these metrics for any 2D point because we simply do not have its  $n$ -dimensional counterpart. Thus, we can not show the projection errors for all the points in our 2D space. Figure 3.7 supports this statement by showing how for a subset of the MNIST data set, the 2D projection of the data points forms a sparse 2D plot (i.e. there exists 2D points/pixels which are not projections of a data point from the data set).

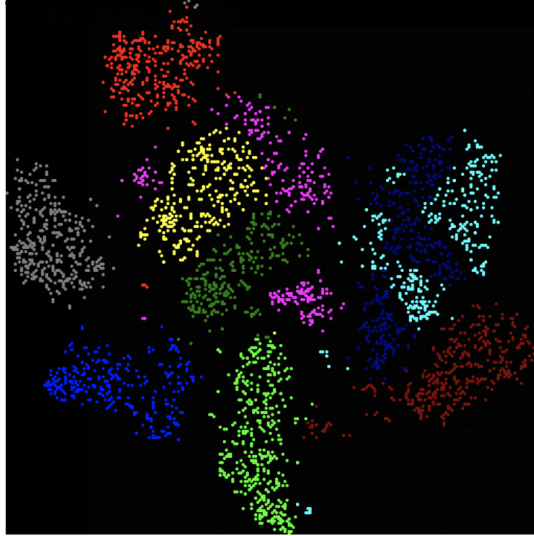


FIGURE 3.7: Example of a 2D projection of a subset of MNIST data set. The colored points represent 2D projections of the data points

We are quite limited in what we can show to the user. We can choose to display the errors only for the pixels that represent a data point from our data set and for the other pixels let the user guess what happens with the projection in these areas (e.g. the dark regions in the figure 3.7). However, this approach is not very informative for the user.

### Computing projection errors using interpolation

We would like to have a function  $\epsilon'_k : [0, 1]^2 \rightarrow [0, 1]$  which takes as an input the 2D coordinates of a point and gives us the projection error. For the points that are 2D representations of samples from our data set  $(x_i, y_i) \in D_{2d}, \forall i \in \overline{0, |D|}$ , we can use the metric defined in equation 3.4,  $\epsilon'_k(x_i, y_i) = \epsilon_k(i)$ . We can assume that  $\epsilon'_k$  is a continuous function. Starting from this assumption and from the fact that we have the values of this function for several points we can use interpolation methods to approximate this function. Once we have an approximation for  $\epsilon'_k$  we can just use it to compute the projection error for each pixel in the DBM image. Algorithm 2 presents the pseudo-code that formalizes these ideas.

The points for which we can compute the actual values of  $\epsilon'_k$  form an unstructured point set. We do not have a lot of options for interpolation techniques we can use in such a case. One option is to perform the Delaunay triangulation, then use per-triangle bilinear interpolation to approximate  $\epsilon'_k$ . However, this would give us a function with  $C^0(\mathbb{R}^2)$  continuity (the derivatives of the reconstructed function would be discontinuous along the edges of those triangles). Ideally, we would prefer a function with  $C^\infty(\mathbb{R}^2)$  continuity. Another option is to use the Radial Basis Function (**RBF**) interpolation method<sup>1</sup>. This method yields a smooth surface that passes through to the projection error values for the given data points. It accomplishes this by using radial basis functions centered at each data point to influence the interpolated values at other locations. The resulting surface is smooth and continuous, and the interpolation process provides a flexible way to estimate values between data points. Hence, in our visualization tool implementation, we use the RBF method to

<sup>1</sup><https://shihchinw.github.io/2018/10/data-interpolation-with-radial-basis-functions-rbfs.html>

infer the projection errors for points outside of the data set. The amount of neighbors  $k$  we take into account when computing  $\epsilon_k(\cdot)$  in our implementation is set to 10 ( $k = 10$ ).

We have several choices for the specific radial basis function we can use, such as Gaussian  $\phi(r) = e^{-(\epsilon r)^2}$ , multiquadric  $\phi(r) = -\sqrt{1 + (\epsilon r)^2}$ , linear  $\phi(r) = -r$  and others. Regardless of the less expressive power and the inability to capture interaction effects in our implementation, we choose the linear  $\phi(r) = -r$  function. The reasons why we preferred this function are interpretability (it is easy to interpret the contributions of errors of individual data points), and computational efficiency.

In figure 3.8 we use the same subset of samples from the MNIST data set (as in figure 3.7) and algorithm 2 in combination with the RBF interpolation method (linear radial basis function) to infer the projection errors for points outside of the data set.

---

**Algorithm 2** Projection errors using interpolation
 

---

**Input:**  $W$  (int),  $H$  (int),  $k$  (int), *interpolation* (string),  $D$ ,  $D_{2d}$

**Output:**  $errors \in \mathcal{M}_{W,H}([0, 1])$

```

1:  $errors \leftarrow \mathcal{O}_{W,H}$ 
2: for  $i \leftarrow 0$  to  $|D|$  do
3:   Compute  $\epsilon_k(i)$  using equation 3.4
4: end for
5: for  $(x_i, y_i) \in D_{2d}$  do
6:    $\epsilon'_k(x_i, y_i) \leftarrow \epsilon_k(i)$ 
7: end for
8: Approximate  $\epsilon'_k$  using interpolation
9: for  $i \leftarrow 0$  to  $W$  do
10:  for  $j \leftarrow 0$  to  $H$  do
11:     $errors[i][j] \leftarrow \epsilon'_k(i/W, j/H)$ 
12:  end for
13: end for
14: return  $errors$ 

```

---

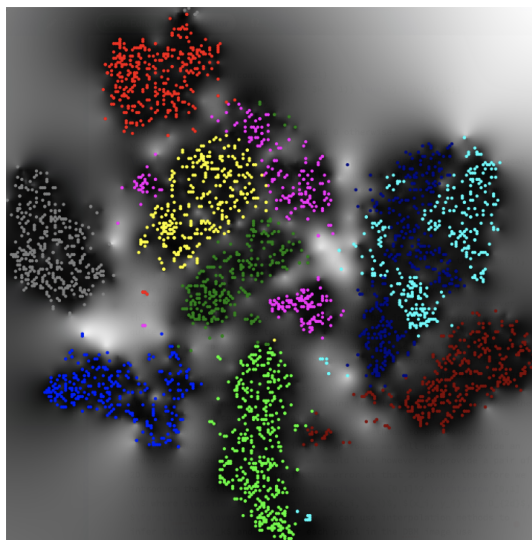


FIGURE 3.8: Example of projection errors  $\epsilon'_{10}(\cdot)$  computed using algorithm 2. Interpolation method: RBF (linear  $\phi(r) = -r$ ). White regions: high error values, dark regions: low error values

When using an interpolation method to approximate a function we are making an assumption about the function shape. For instance, if we are using a bilinear interpolation we are assuming that between two samples for which we have evidence the function behaves linearly. For different data sets this projection error function we are trying to approximate might behave differently. Therefore using such an approach that implies interpolation, in general, might give us poor approximations. For example, see the top-right corner of figure 3.8, the interpolation method tells us that this is a region where the projection might be misleading (due to the high error values). This region is mostly determined by one single point from the data set which is the closest to this region in 2D. Recall that the points in figure 3.8 are just 2D projections of a subset of points from our data set. Thus, there might be another data point in our data set that, when projected in 2D, is closer to that specific region and has a lower projection error associated with it. This means that all the points in this region will have a lower projection error assigned. The same argumentation holds if we use other radial basis functions like Gaussian or multiquadric, the only difference is that we need to add more points to see the effect of the change in the errors.

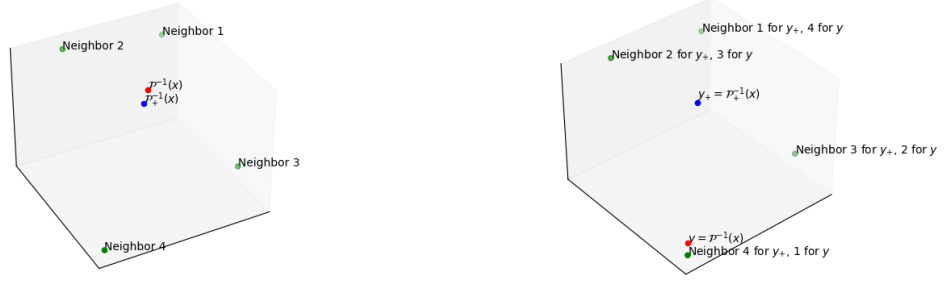
Adding a point (or a set of points) to our data set should only change the projection errors in the neighborhood of this point, not the entire regions of the 2D space. Due to these reasons, we propose another approach for computing the projection errors.

### Computing projection errors using inverse projection

Our data set is just a sample of the space  $\mathcal{D}$ , i.e.  $D \sim \mathcal{D}$ . For simplicity let us assume that we have an ideal inverse projection  $\mathcal{P}_+^{-1}$ , i.e.  $\mathcal{P}_+^{-1}(\mathcal{P}(x)) = x, \forall x \in \mathcal{D}$ . In this case, computing the projection errors of  $\mathcal{P}$  becomes an easy task, because, for any 2D pixel, we can use the inverse projection to get its corresponding  $n$ D vector, then use the 2D coordinates and  $n$ D coordinates to compute the projection errors by using equation 3.4. Notice that for the samples in our data set, we have such an ideal inverse projection, namely  $\mathcal{P}_+^{-1}(x_i, y_i) = (x_{i,1}, \dots, x_{i,n}), \forall i \in \overline{0, |D|}, (x_i, y_i) \in D_{2d}, (x_{i,1}, \dots, x_{i,n}) \in D$ . Now let us take a step back and think about what we are trying to capture with the projection errors. Let  $(x, y) \notin D_{2d}$ , we want to see how the neighbors of  $(x, y)$  are changing in the  $n$ D space, or in other words how the projection is keeping the neighborhood of a certain  $n$ D point when reducing the dimensionality. In general our inverse projection method  $\mathcal{P}^{-1}$  (learned using NNinv, SSNP, etc.) will not give the same  $n$ D point as  $\mathcal{P}_+^{-1}$ , i.e.  $\mathcal{P}^{-1}(\cdot) \neq \mathcal{P}_+^{-1}(\cdot)$ . However, if  $\mathcal{P}^{-1}$  is a good enough approximation of  $\mathcal{P}_+^{-1}$  then if we consider the order with respect to the distance of the  $n$ D neighbors that are in  $D$ , this order will be the same regardless of the usage of  $\mathcal{P}^{-1}$  or  $\mathcal{P}_+^{-1}$ . Figure 3.9 shows that depending of how well our inverse projection method  $\mathcal{P}^{-1}$  approximates  $\mathcal{P}_+^{-1}$  the order of neighbors from  $D$  change or not.

Starting from the assumption that our inverse projection method is an approximation of  $\mathcal{P}_+^{-1}$  that preserves the neighbors' order for each 2D point we can compute the projection errors with the usage of  $\mathcal{P}^{-1}$ . Let  $x_{2d}$  be an arbitrary 2D data point that does not necessarily need to be a 2D embedding of a point from the data set  $D$  and let  $x_{nd} = \mathcal{P}^{-1}(x_{2d})$  then we can compute the projection error for  $x_{2d}$  by using equation 3.5.

$$\epsilon_k''(x_{nd}, x_{2d}) = \frac{1}{(2n - 3k - 1)} \left( \sum_{i \in \mathcal{U}_k(x_{nd})} [r_i(x_{nd}) - k]_+ + \sum_{i \in \mathcal{V}_k(x_{2d})} [\hat{r}_i(x_{2d}) - k]_+ \right) \quad (3.5)$$

(A)  $\mathcal{P}^{-1}$  preserves the neighbors order(B)  $\mathcal{P}^{-1}$  does not preserve the neighbors orderFIGURE 3.9: Examples of how the neighbors' order is preserved depending on how well  $\mathcal{P}^{-1}$  approximates  $\mathcal{P}_+^{-1}$ 

In equation 3.5  $U_k(x)$  is the set of  $k$  closest neighbours to  $x$  in  $D$ , similarly  $V_k(x)$  is the set of closest neighbours in  $D_{2d}$ ,  $r_i(x)$  is the distance rank between  $x_i \in D$  and  $x$ , and  $\hat{r}_i(x)$  is the distance rank between  $x_i \in D_{2d}$  and  $x$ .

We can actually evaluate how well our assumption that  $\mathcal{P}^{-1}$  approximates  $\mathcal{P}_+^{-1}$  holds, by comparing  $\epsilon_k''$  with  $\epsilon_k$  for values from  $D_{2d}$ . For points in  $D_{2d}$ , the values of  $\epsilon_k$  represent the ground truth.

Algorithm 3 shows the pseudo-code for the computation of the projection errors using the approach we described so far and equation 3.5.

Since we can compute the projection error for any arbitrary point  $x_{2d}$  as follows  $\epsilon_k''(\mathcal{P}^{-1}(x_{2d}), x_{2d})$  we do not need to perform any interpolation and make any assumptions about the projection error function form. Thus, overcoming the limitations of the algorithm 2. On the other hand, the accuracy of the projection errors directly depends on how good  $\mathcal{P}^{-1}$  approximates the ideal inverse projection  $\mathcal{P}_+^{-1}$ . Therefore we recommend the usage of this approach only in combination with a visual representation of the inverse projection errors.

Figure 3.10 shows the same subset of data points from the MNIST data set as the one used in figures 3.7 and 3.8. In this figure the algorithm 3 is used to compute the projection errors.

In our visualization tool implementation, we allow the user to use both approaches described so far (i.e. algorithms 2 and 3).

---

**Algorithm 3** Projection errors using inverse projection
 

---

**Input:**  $W$  (int),  $H$  (int),  $\mathcal{P}^{-1}$ ,  $D$ ,  $D_{2d}$

**Output:**  $errors \in \mathcal{M}_{W,H}([0,1])$

- 1:  $errors \leftarrow \mathcal{O}_{W,H}$
  - 2: **for**  $i \leftarrow 0$  to  $W$  **do**
  - 3:     **for**  $j \leftarrow 0$  to  $H$  **do**
  - 4:          $x_{2d} \leftarrow (i/W, j/H)$
  - 5:          $x_{nd} \leftarrow \mathcal{P}^{-1}(x_{2d})$
  - 6:         Compute  $\epsilon_k''(x_{nd}, x_{2d})$  // using equation 3.5
  - 7:          $errors[i][j] \leftarrow \epsilon_k''(x_{nd}, x_{2d})$
  - 8:     **end for**
  - 9: **end for**
  - 10: **return**  $errors$
-

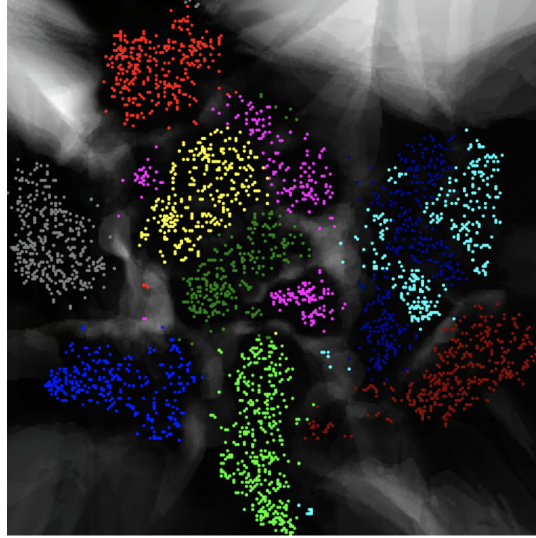


FIGURE 3.10: Example of projection errors  $\epsilon''_{10}(\cdot)$  computed using algorithm 3. White regions: high error values, dark regions: low error values

### 3.2.2 Inverse Projection errors

In order to evaluate how our inverse projection method performs for a certain 2D point  $p$  we compute an approximation of the derivative of  $\mathcal{P}^{-1}$  at that point according to equation 3.6 (see section 2.5 and paper [9]).

$$\begin{aligned}
 D_x(p) &= \frac{\mathcal{P}^{-1}(p + (w, 0)) - \mathcal{P}^{-1}(p - (w, 0))}{2w} \\
 D_y(p) &= \frac{\mathcal{P}^{-1}(p + (0, h)) - \mathcal{P}^{-1}(p - (0, h))}{2h} \\
 D(p) &= \sqrt{\|D_x(p)\|^2 + \|D_y(p)\|^2}
 \end{aligned} \tag{3.6}$$

This equation captures the idea that the low distances between neighbors in 2D should be preserved in the  $n$ D space. Regions with large gradient values illustrate where the high-dimensional distance is changing most rapidly with respect to the low-dimensional distance.

In equation 3.6 we have two parameters  $w$  and  $h$ , since we are interested in the best approximation of the derivative we take  $w = 1$  and  $h = 1$ . Figure 3.11 shows a visual illustration of the way we compute this derivative.

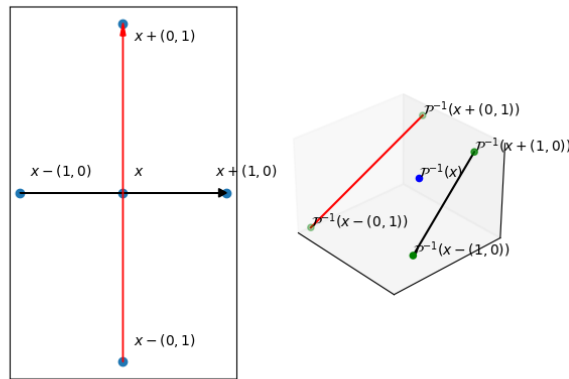


FIGURE 3.11: Example of inverse projection error computation using the approximation of inverse projection derivative

We want to mix the visual representation of the projection errors with the inverse projection errors in our tool. However, the values returned by the metric described in equation 3.6 are not in the range  $[0,1]$ . In order to normalize the values of inverse projection errors after computing these values for each point from our 2D space we perform a min-max normalization. In figure 3.12 we show an example of a gradient map of an inverse projection computed using the equation 3.6. The brightness of the regions tells us how a small position change in the 2D space maps to a position change in the  $nD$  space, the brighter the region the bigger the position change in the high dimensional space.

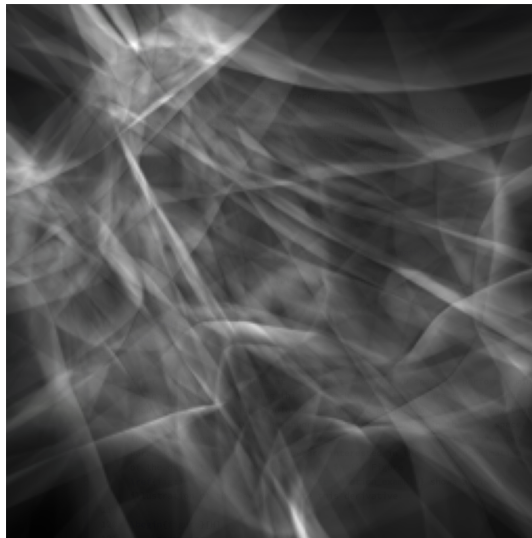


FIGURE 3.12: Example of the inverse projection error gradient map.  
White regions: high error values, dark regions: low error values

### 3.3 Data points re-labeling and classifier retraining

In the previous sections, we presented how for a given classifier  $\mathcal{C}$  we can generate the decision boundary map and compute the related projection and inverse projection errors. The end goal of our visualization tool is to help improve the classifier



performance in the context of pseudo-labeled data sets. In order to achieve this we need to allow the user to re-label data points and then re-train the classifier (i.e. requirement **R3**).

Figure 3.13 presents the pipeline in which the user can interact with the tool to re-label data points to improve the classifier. In order to enable the re-labeling we provide the user with two options as follows:

- Click on a 2D representation of a data point use the keyboard to assign the new label and click *Enter* to confirm the new label
- Use the mouse to draw a circle, then use the keyboard to assign all the points within the circle with the new label

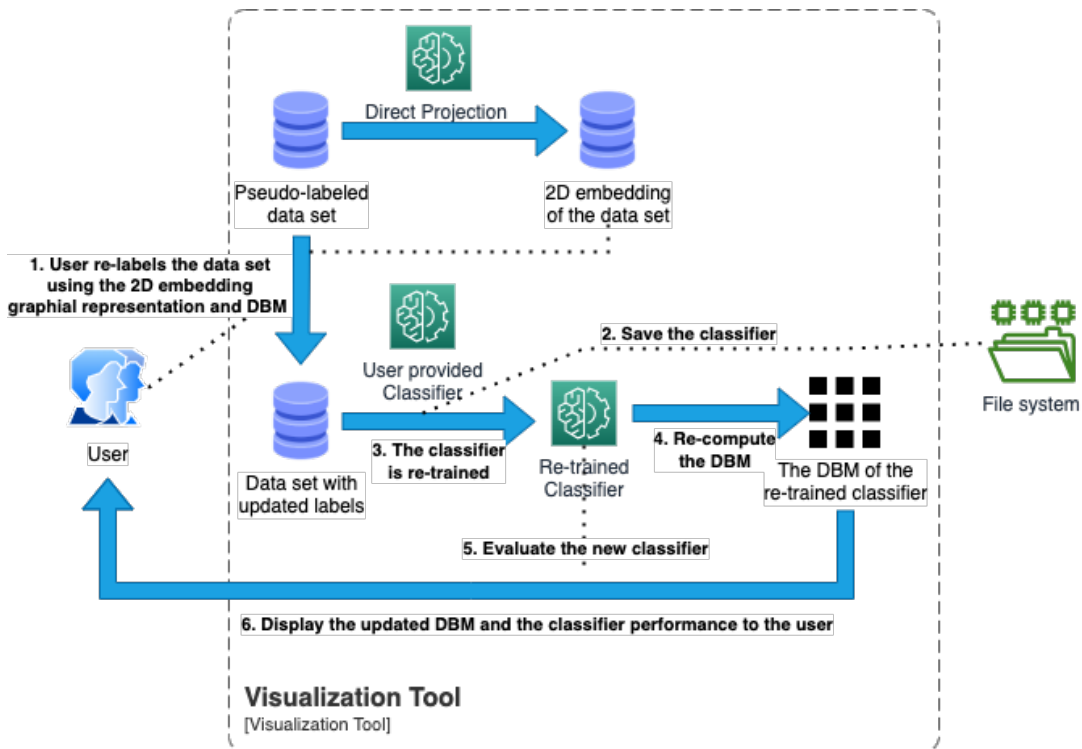


FIGURE 3.13: Data set re-labeling and classifier re-training pipeline (**R3.1** and **R3.2**)

After the user made all the changes in the labels that he/she considers necessary, the classifier  $\mathcal{C}$  is re-trained from scratch for a number of epochs (that can be specified by the user, by default the number of epochs is 20) resulting in a new classifier  $\mathcal{C}'$ . The old classifier is stored along with the label changes so that the changes can be undone. The new classifier is evaluated and a plot that shows the differences in performance is constructed so that the user can decide to undo the changes, continue with the changes, or stop the tool. The classifier  $\mathcal{C}'$  will have a different decision boundary map than  $\mathcal{C}$ , therefore each time the user applies changes in the labels we have to re-compute the DBM image. Notice on the other hand that the projection and the inverse projection are not affected by the changes in data points' labels. Thus, so are the projection and inverse projection errors. This observation tells us that these kinds of errors can be computed at any iteration of the tool usage depending on what the user demands. In conclusion, the only thing that needs to be re-computed per iteration is the DBM image. If the DBM computation is not optimal this can slow down the tool interaction process.

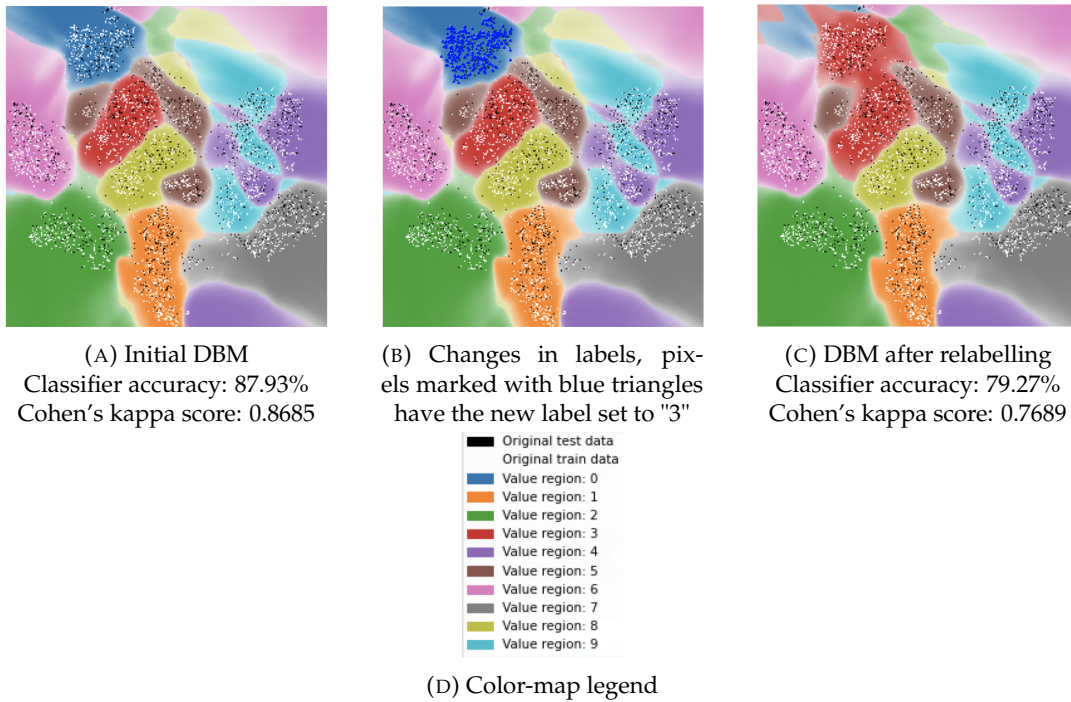


FIGURE 3.14: Example of re-labeling and classifier re-training

Figure 3.14 presents an example of the described tool interaction process. In this example, we use the following configuration:

- MNIST dataset [10] (training set - first 3500 samples from the MNIST training set, test set - first 1500 samples from the MNIST test set)
- DBM image size  $256 \times 256$
- Projection method: t-SNE
- Inverse projection method: Learned using NNinv (loss: mean-squared-error), training and validation sets: 80/20 split of the training data set, training strategy: early stopping strategy monitoring the validation loss, training epochs: 300 (early stopping after 235 epochs)
- Random seed set to 42
- Classifier  $\mathcal{C}$  architecture: Flatten layer followed by a Dense layer with 10 neurons and softmax activation, trained for 20 epochs
- Classifier performance is computed using the testing set
- Initial data labels are the same as the ones in the MNIST data set
- Re-fitting number of epochs: 20
- Algorithm 1 is used for re-computing the DBM after retraining

Observe how after changing the labels that were initially labeled as "0" to the label "3" the classifier performance changes (see figures 3.14a and 3.14c). The drop in performance is caused by the fact that initially the labels were correctly set. Furthermore, notice that the DBM of the classifier after applying the changes in data sample labels (i.e. figure 3.14c) is different from the initial DBM (i.e. figure 3.14a).

### 3.4 Optimization heuristics for DBM generation

The most frequently used functionality in our visualization tool is the computation of the Decision Boundary Map for a given classifier. So far we have presented the algorithm 1 which can generate the DBM.

Let us assume that for our DBM images, we have  $W = H$ , and for simplicity let us denote the width and height with  $n$  (i.e.  $n = W = H$ ). Let us denote the cost of  $\mathcal{C}(\mathcal{P}^{-1}(x, y))$  where  $(x, y)$  is a 2D point, with  $C$ . The complexity of algorithm 1 is then  $O(n^2C)$  since for each  $n^2$  pixels of the DBM image we need to make a call to  $\mathcal{C}(\mathcal{P}^{-1}(\cdot))$ .

Although the complexity of the algorithm 1 is linear in the number of pixels  $n^2$ , depending on the machine on which this algorithm is performed the run-time might not be acceptable for a user of the visualization tool. Table 3.2 shows the run-times for generating DBM for the MNIST data set with the use of algorithm 1 on two different machines. The direct projection is t-SNE and the inverse projection method is computed by using NNinv as described at the end of the previous section. Generating the DBM for a resolution of  $512 \times 512$  takes 18.461 and 27.988 seconds respectively (see figure 3.15 for different resolutions and the corresponding run-time of the algorithm 1). This is problematic because the user will interact with our visualization tool iteratively. At each step, the user will have to re-assign labels, re-train the classifier, generate a new DBM, and then re-judge based on that DBM. Therefore, we would like to make the generation of the DBM as fast as possible, so that the whole process of using the tool is not slowed down by the computation of the DBM. Thus, we need to improve on the complexity of algorithm 1.

OS	CPU(s)	GPU(s)	Run time (seconds)
macOS 12.6	6-Core Intel(R) Core(TM) i7 - 9750H @ 2.60GHz	Not used	18.461
Ubuntu 22.04.2 LTS	2-Core Intel(R) Core(TM) i7 - 7500U @ 2.70GHz	Not used	27.988

TABLE 3.2: Run times of algorithm 1 for DBM resolution  $512 \times 512$  on different machines. Data set: MNIST

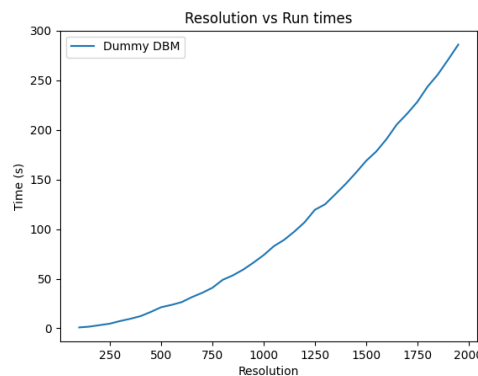


FIGURE 3.15: Dummy DBM algorithm 1 run-times for different image resolutions. Dataset: MNIST,  $\mathcal{P}$ : t-SNE,  $\mathcal{P}^{-1}$ : NNinv

In order to have better complexity we can aim for improving the complexity of the  $\mathcal{C}(\mathcal{P}^{-1}(\cdot))$  operation. However, we want also to keep the generality of our visualization tool so that  $\mathcal{C}$  and  $\mathcal{P}^{-1}$  can be any kind of operations that fall under the requirements of classifying a high-dimensional data point and converting a 2D point

to a high-dimensional counterpart respectively. Thus, to preserve this generality we aim to reduce the  $n^2$  part in the complexity of algorithm 1.

In general a classifier  $\mathcal{C}$  has to achieve a trade-off between the ability to fit the training set (which requires its decision boundaries to ‘twist’ to follow each training sample) and the ability to generalize (which requires the decision boundaries to be relatively smooth and simple in order to avoid the overfitting). Therefore most classifiers aim to create a few decision zones (in general more or equal to the number of classes and less or equal to the training data samples) with relatively smooth boundaries, meaning that the decision boundaries are sparse in the  $nD$  space. The labels of the DBM are assigned based on the probability vector that is given by  $\mathcal{C}(\mathcal{P}^{-1}(\cdot))$ , since  $\mathcal{C}$  and  $\mathcal{P}^{-1}$  are continuous and the derivative of  $\mathcal{P}^{-1}$  is bounded, a small change in the 2D position will lead us to a small change in the probability vector, which means that the decision boundaries are sparse in the 2D space as well. The DBM represents a stair signal (function) that remains constant in the decision zones. We can exploit the stair signal nature of the DBM and the fact that the decision boundaries are not dense in the 2D and  $nD$  spaces by using various sampling strategies that reduce the number of points where we need to evaluate actual labels using  $\mathcal{C}(\mathcal{P}^{-1}(\cdot))$ . Starting from this idea, in sections 3.4.1 and 3.4.2, we present two heuristic methods that compute the DBM and aim to reduce the  $n^2$  term in the complexity of algorithm 1. In section 3.4.3 we propose a heuristic that aims to reduce both the  $n^2$  and  $C$  terms in the  $O(n^2C)$  complexity.

This section is structured as follows. In sections 3.4.1 and 3.4.2 we present two simple heuristics for generating the DBM that aim to skip useless computations based on pixels neighborhood analysis. Algorithm 9 in section 3.4.3 presents a heuristic that infers the pixel label and the confidence of the classifier based on local neighborhood interpolation. Section 3.4.4 compares all the presented heuristics in terms of complexity and accuracy. In this section, we identify the limitations, advantages, and disadvantages of using each of the presented algorithms. For each of the sections 3.4.1, 3.4.2 and 3.4.3 we present the core ideas of the DBM algorithm along with the intuition and the pseudo-code that formalizes the presented ideas. For each algorithm, we analyze the complexity and present how the algorithm performs in terms of speed and accuracy.

For the experiments presented in these sections, we use the following configuration:

- MNIST dataset [10] (training set - first 3500 samples from MNIST training set, test set - first 1500 samples from MNIST testing set)
- Projection method: t-SNE
- Inverse projection method: Learned using NNinv (loss: mean-squared-error), training and validation sets: 80/20 split of the training data set, training strategy: early stopping strategy monitoring the validation loss, training epochs: 300 (early stopping after 235 epochs)
- Random seed set to 42
- Classifier architecture: Flatten layer followed by a Dense layer with 10 neurons and softmax activation, trained for 20 epochs
- Machine configuration: macOS 12.6, 6-Core Intel(R) Core(TM) i7 - 9750H @ 2.60GHz (CPU), GPU not used

### 3.4.1 Binary Split heuristic

In this section, we introduce a heuristic we call **Binary Split**. This technique aims to speed up the computation of the DBM image. We start by presenting the intuition of the algorithm, then present the pseudo-code that formalizes the intuition, followed by a complexity analysis and experimental results that show the differences of the new heuristic when compared to algorithm 1.

#### Binary Split intuition

Let us start by taking a closer look at figure 3.14a. Consider for instance the top row of the image, the label of the first pixel is "0" and continues to remain the same for a considerable amount of pixels in that row. If it was the case that we knew the coordinates  $(x, 0)$  where the classifier starts to predict a different label, there is no need for us to use the classifier  $\mathcal{C}$  to compute all the labels between  $(0, 0)$  and  $(x, 0)$  since we know apriori that the classifier will predict the label "0" in this interval. The problem is that we do not know the value of  $x$ .

Consider the following situation, we have a row of pixels and with coordinates between  $(a, 0)$  and  $(b, 0)$ . For simplicity let us assume that our classifier can predict only two labels  $l_1$  and  $l_2$ . Again for simplicity, we assume that there is only one point  $x \in [a, b]$  where the value of labels changes from  $l_1$  to  $l_2$ . We want to find this point  $x$ . We can apply a simple numerical analysis technique called Bisection or Binary-search method [4] in which we recursively cut our search interval  $[a, b]$  into smaller intervals until we find  $x$ .

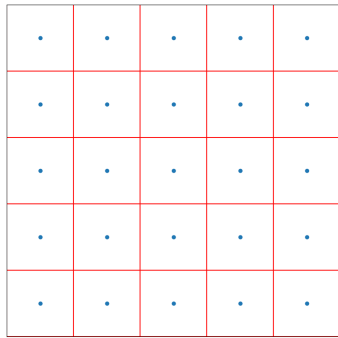
Next, we explain how we can apply the Binary-search method to our DBM generation problem, where we have more than 2 labels and instead of a one-dimensional interval  $[a, b]$  we have a 2D interval  $[a, b] \times [c, d]$ .

For simplicity let us consider only the first row of our DBM from figure 3.14a. We can split the interval  $[0, n - 1]$  into smaller intervals and take a "representative" pixel from each interval. For instance, let us split the first row into 32 intervals, each containing 8 pixels, i.e.  $[0, 7]$ ,  $[8, 15]$ ,  $\dots$ ,  $[248, 255]$ , and take from each interval  $[a_i, a_i + 7]$  the pixel with coordinates  $(a_i + 3, 0)$  (i.e. the middle pixel of the interval) as representatives i.e.  $(3, 0)$ ,  $(11, 0)$ ,  $\dots$ ,  $(251, 0)$ . For each of the representative pixels, we compute the label using the classifier. The result will be a sequence of labels i.e. "0", "0", "0",  $\dots$ , "6", "6". Observe how the first 3 labels in this sequence are "0", because the label is not changing we can assume that all the pixels in the first and the second interval will have the label "0". On the other hand, if in this sequence we have a sub-sequence  $l_{i-1}, l_i, l_{i+1}, l_{i+2}$ , where  $l_{i-1} = l_i \neq l_{i+1} = l_{i+2}$  then the conclusion we can make is that the decision boundary lies somewhere between  $(a_i + 3, 0)$  and  $(a_{i+1} + 3, 0)$ .

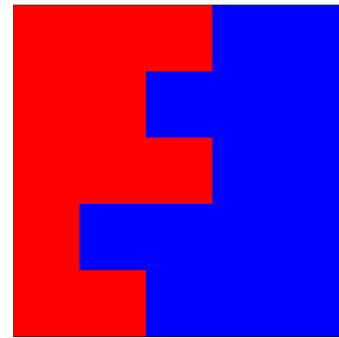
In order to find where exactly the label changes from  $l_i$  to  $l_{i+1}$  we can apply the Binary-search strategy, namely, take the intervals  $[a_i + 3, a_i + 7]$  and  $[a_{i+1}, a_{i+1} + 3]$ , take the representative middle pixels from these intervals and compute their labels  $l'_i, l'_{i+1}$  include the labels in the initial sequence as follows:  $\dots, l_i, l'_i, l'_{i+1}, l_{i+1}, \dots$  and repeat the same process described so far. If for instance  $l_i = l'_i \neq l'_{i+1} = l_{i+1}$  then all the pixels in the interval between the pixel with label  $l_i$  and the one with label  $l'_i$  are assigned with the label  $l_i$ . All the pixels between  $l'_{i+1}$  and  $l_{i+1}$  are assigned with the label  $l_{i+1}$ . The interval between the pixels with labels  $l'_i$  and  $l'_{i+1}$  is split again into 2 sub-intervals. The procedure is repeated until we can not further split an interval.

Notice how based on neighboring intervals and representatives of the intervals we can quickly find where decision boundaries are laying. So far we considered only

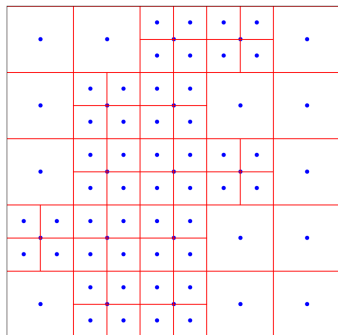
one row of the image and made the analysis from a one-dimensional perspective. These ideas however can be easily extrapolated so that instead of intervals in one dimension we have 2D rectangles (i.e. windows/blocks). The difference now is that instead of looking at the labels to the left and right intervals we need to analyze the labels of the representative pixels of the 2D rectangles from the top, bottom, left, and right. The core idea is to start by splitting the DBM image we want to generate into a set of  $B^2$  blocks. For each block  $\mathcal{B}$ , we take the central pixel as a representative and analyze the representatives of the neighboring blocks. If all the neighbors have the same label as the current block we can assign the same label to all the pixels in that block. In such a way, we skip redundant computation steps. For the blocks that have neighbors with different labels, we are splitting the block into 4 sub-blocks. Once we have done it for all the blocks we can repeat the analysis of neighbors and split further until each block has no neighbors with different labels or the block can not be further split (i.e. the block is a pixel). Figure 3.16 gives an intuition of how the Binary Split algorithm 4 works. Observe how we started with a grid of  $5 \times 5$  blocks in figure 3.16a and in the following iteration, we split only some blocks (see figure 3.16c) in order to find where the decision boundary is.



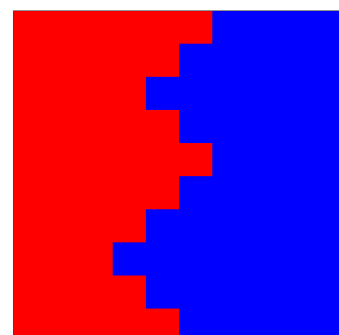
(A) Iteration 1 - 2D blocks and their representative pixels



(B) Iteration 1 - Labels of representative pixels of 2D blocks



(C) Iteration 2 - 2D blocks and their representative pixels



(D) Iteration 2 - Labels of representative pixels of 2D blocks

FIGURE 3.16: DBM construction using the Binary Split heuristic for a classifier with 2 classes

### Binary Split algorithm

So far we described how we can infer the label values for all the pixels. However, along with the decision boundary map, we want to show the classifier confidence to the user for each pixel. We can assign to all the pixels in a block the same confidence as the central representative pixel of the block. However, in this case, we will receive a stair function, and this is usually not how the confidence values of a classifier behave. For each central pixel of a block, we have the real confidence  $c$  of the classifier. We can perform an interpolation over these values to approximate the confidence function. In our experiments, we use three interpolation methods for computing the confidence maps, namely: nearest neighbor, bilinear, and bicubic. We will explicitly show the experiments and results for all the DBM optimization methods together with all the interpolation methods in chapter 4.

If we start at the top left block and continuously split it until no more splits are possible, then go to the next block and do the same, the neighbors to the left will have a bigger impact on how we split this block than the neighbors on the right because we did not split the block to the right yet. This might lead the algorithm to non-accurate results. In order to avoid this, we would like to assign a priority to a block split, such that splitting a block of the same size with the same amount of neighbors happens in the same iteration of the algorithm. In this order of ideas we need to use a priority queue to store the blocks that need to be split and at each iteration split the blocks with the same priority and place the sub-blocks back in the queue.

Algorithm 4 formalizes the intuition of the Binary Split heuristic. The sub-routines used in algorithm 4 are described in procedures 5 and 6. Algorithm 4 can be modified to include a computational budget, such that the heuristics stop when there is no more computational budget available.

---

#### Algorithm 5 Generate initial blocks sub-routine

---

```

1: procedure GENERATE INITIAL BLOCKS( $n, B$ )
2:    $block\_size \leftarrow \frac{n}{B}$ 
3:    $blocks \leftarrow []$ 
4:   for  $i \leftarrow 0$  to  $B$  do
5:     for  $j \leftarrow 0$  to  $B$  do
6:        $x, y \leftarrow i * block\_size + \frac{block\_size}{2}, j * block\_size + \frac{block\_size}{2}$ 
7:        $blocks \leftarrow blocks + [(x, y, block\_size, block\_size)]$ 
8:     end for
9:   end for
10:  return  $blocks$ 
11: end procedure

```

---



---

#### Algorithm 6 Insert block into priority queue sub-routine

---

```

1: procedure INSERT PRIORITY QUEUE( $priority\_queue, dbm\_image, \mathcal{B}$ )
2:    $x, y, w, h \leftarrow block$ 
3:    $neighbors \leftarrow [(x - \frac{w}{2} - 1, y), (x + \frac{w}{2} + 1, y), (x, y - \frac{h}{2} - 1), (x, y + \frac{h}{2} + 1)]$ 
4:    $cost \leftarrow 0$ 
5:    $label \leftarrow dbm\_image[x][y]$ 
6:   for  $(x, y)$  in  $neighbors$  do
7:     if  $label \neq dbm\_image[x][y]$  then
8:        $cost \leftarrow cost + 1$ 
9:     end if
10:  end for
11:  if  $cost > 0$  then
12:     $priority \leftarrow 1 / (w * h * \frac{cost}{len(neighbors)})$ 
13:     $priority\_queue.put(priority, \mathcal{B})$ 
14:  end if
15: end procedure

```

---

---

**Algorithm 4 Binary Split**


---

**Input:**  $\mathcal{P}^{-1} : [0, 1]^2 \rightarrow [0, 1]^n$ ,  $\mathcal{C} : [0, 1]^n \rightarrow [0, 1]^d$ ,  $n$  (int),  $B$  (int), *interpolation\_method* (string)  
**Output:**  $dbm \in \mathcal{M}_{n,n}(\mathbb{N})$   
 $confmap \in \mathcal{M}_{n,n}([0, 1])$

```

1:  $dbm \leftarrow \mathcal{O}_{n,n}$ 
2:  $confmap \leftarrow []$ 
3:  $blocks \leftarrow GENERATE\_INITIAL\_BLOCKS(n, B)$ 
4: for  $block$  in  $blocks$  do
5:    $x, y, w, h \leftarrow block$ 
6:    $probabilities \leftarrow \mathcal{C}(\mathcal{P}^{-1}(x/n, y/n))$ 
7:    $label, conf \leftarrow \arg \max(probabilities), \max(probabilities)$ 
8:    $dbm[x - \frac{w}{2} : x + \frac{w}{2}][y - \frac{h}{2} : y + \frac{h}{2}] \leftarrow label$ 
9:    $confmap \leftarrow confmap + [(x, y, conf)]$ 
10: end for
11:  $priority\_queue \leftarrow PriorityQueue()$  // initialize a priority queue
12: for  $block$  in  $blocks$  do
13:    $INSERT\_PRIORITY\_QUEUE(priority\_queue, dbm, block)$ 
14: end for
15: while not  $priority\_queue.empty()$  do
16:   // pop the blocks with the highest priority from the queue
17:    $priority, blocks \leftarrow GET\_BLOCKS\_WITH\_HIGHEST\_PRIORITY(priority\_queue)$ 
18:   for  $block$  in  $blocks$  do
19:      $x, y, w, h \leftarrow block$ 
20:     if  $w == 1$  and  $h == 1$  then
21:        $probabilities \leftarrow \mathcal{C}(\mathcal{P}^{-1}(x/n, y/n))$ 
22:        $label, conf \leftarrow \arg \max(probabilities), \max(probabilities)$ 
23:        $dbm[x][y] \leftarrow label$ 
24:        $confmap \leftarrow confmap + [(x, y, conf)]$ 
25:       continue
26:     end if
27:      $\delta_w, \delta_h \leftarrow \frac{w}{4}, \frac{h}{4}$ 
28:      $points \leftarrow [(x - \delta_w, y - \delta_h), (x - \delta_w, y + \delta_h), (x + \delta_w, y - \delta_h), (x + \delta_w, y + \delta_h)]$ 
29:     for  $x, y$  in  $points$  do
30:        $probabilities \leftarrow \mathcal{C}(\mathcal{P}^{-1}(x/n, y/n))$ 
31:        $label, conf \leftarrow \arg \max(probabilities), \max(probabilities)$ 
32:        $dbm[x - \delta_w : x + \delta_w][y - \delta_h : y + \delta_h] \leftarrow label$ 
33:        $confmap \leftarrow confmap + [(x, y, conf)]$ 
34:     end for
35:     for  $x, y$  in  $points$  do
36:        $INSERT\_PRIORITY\_QUEUE(priority\_queue, dbm, (x, y, 2\delta_w, 2\delta_h))$ 
37:     end for
38:   end for
39: end while
40:  $confmap \leftarrow INTERPOLATE(confmap, n, interpolation\_method)$ 
41: return  $dbm, confmap$ 

```

---



### Binary Split heuristic complexity analysis

In this paragraph, we want to analyze the complexity of the algorithm 4. Let  $C$  be the cost of one operation  $\mathcal{C}(\mathcal{P}^{-1}(\cdot))$  and  $n^2$  be the number of pixels in the image of the decision boundary map we want to generate. With this notations the complexity of algorithm 1 is  $O(n^2C)$ . Let  $B^2$ , where  $1 < B < n$  be the initial number of blocks we are analyzing in order to construct the priority queue in algorithm 4. If the number of classes in our data set is 1, our algorithm will not insert anything in the priority queue, which means that the best-case complexity is  $O(B^2C + n^2)$ , where  $B < n$ . The term  $B^2C$  represents the number of computations needed to split the image into  $B^2$  blocks and decode the central representative pixels of each block. The term  $n^2$  represents the number of computations needed in order to perform the confidence map interpolation.

In the worst-case scenario, we have to decode each pixel individually, which means  $n^2$  pixels, i.e. the classes are distributed in such a way in the 2D space that at each iteration for each block we need to replace it into the priority queue 4 new blocks until the block size becomes  $1 \times 1$ . If we start with blocks of size  $\frac{n}{B}$ , and each time split it into 4 sub-blocks we need a total of  $\log_2 \frac{n}{B}$  operations to get blocks of size  $1 \times 1$ . The insertion to the priority queue is  $O(\log(s))$  where  $s$  is the size of the priority queue.

The worst time complexity is then:

$$\begin{aligned} & O(B^2C + 4B^2(C + \log(B^2)) + 4^2B^2(C + \log(4B^2)) + \\ & \dots + 4^k B^2(C + \log(4^{k-1}B^2)) + \dots + \frac{n^2}{4} (C + \log(\frac{n^2}{16})) + n^2C + n^2) \end{aligned} \quad (3.7)$$

The term  $B^2C$  is the number of computations we need for the initial computation of the representative pixels of the blocks the term  $n^2C$  represents the number of computations we need in order to decode each pixel at the end when the block size is  $1 \times 1$ . The term  $n^2$  is the number of computations we need to compute the confidence image using interpolation. The time complexity can be rewritten as:

$$O(n^2 \log(\frac{n}{B})(C + \log(n))) \quad (3.8)$$

We can not simplify the above equation any further, because the complexity depends on two parameters  $C$  and  $n$ . If  $C < \log n$  then  $O(n^2 \log(\frac{n}{B})(C + \log(n))) = O(n^2 \log(\frac{n}{B}) \log(n))$ . However, in practice is very unlikely that the user will want to generate images with such big resolutions and such a simple classifier so that  $C < \log n$ . Therefore is more correct to assume that it usually will be the case that  $C > \log n$ . Then the most simplified formula of the worst-case complexity becomes  $O(n^2 \log(\frac{n}{B})(C + \log(n))) = O(n^2 \log(\frac{n}{B})C)$ .

If  $B$  is taken as a portion of  $n$  then the worst-case complexity becomes  $O(n^2(C + \log(n)))$ . Even though the worst-case complexity of the Binary Split heuristic is worse than the complexity of the vanilla DBM algorithm 1 the average complexity of this heuristic is better. We can slightly change the implementation of the Binary Split heuristic to include a computational budget in order to constrain the worst-case complexity to  $O(n^2C)$ . Moreover, by introducing the computational budget parameter in this algorithm, we can reduce the complexity of the algorithm even further. However, this comes at the cost of an increasing number of errors (i.e. misleading labels). This trade-off depends on how deep the user wants to go in the splitting. In other words, this computational budget embeds the maximum size of the smallest block of pixels that is remaining in the map. In our implementation, we take the

computational budget equal to  $n^2$ , which means the worst-case complexity of the algorithm 4 is  $O(n^2C)$ .

In practice, the number of pixels that construct the decision boundaries is  $Kn$  where  $K$  ( $K \ll n$ ) is a constant that depends on the data set  $D$ , the classifier  $\mathcal{C}$  and the inverse projection  $\mathcal{P}^{-1}$ . Thus, the complexity of the binary split becomes:

$$O(B^2C + c_1B^2(C + \log(B^2)) + \cdots + c_lB^2(C + \log(c_{l-1}B^2)) + KnC + n^2) \quad (3.9)$$

where  $c_iB^2 \leq Kn, \forall i \in \overline{1, l}, l < \log(\frac{n}{B})$ . The average case complexity is then:

$$\begin{aligned} O(B^2C + \log(\frac{n}{B})Kn(C + \log(Kn)) + n^2) = \\ O(B^2C + \log(\frac{n}{B})Kn(C + \log(n)) + n^2) \end{aligned} \quad (3.10)$$

If  $B$  is a constant that does not depend on  $n$  then the complexity becomes  $O((Kn \log n)(C + \log n) + n^2)$ . Notice how by applying the Binary Split heuristic we can reduce the average complexity of the decision boundary map construction from  $O(n^2C)$  to  $O((Kn \log n)(C + \log n) + n^2)$ , where  $K \ll n$ .

### Binary Split vs Vanilla DBM

In this paragraph, we experimentally compare the Binary Split heuristic (algorithm 4) with the Dummy DBM (algorithm 1) in terms of run-time and accuracy. The complete global comparison of all the DBM optimization methods is presented in section 3.4.4.

The Dummy DBM algorithm gives the correct labels and confidence values for each pixel. Therefore, we take the Dummy DBM results as the ground truth. Figure 3.17 presents the errors in the label values of the Binary Split algorithm for the resolution  $256 \times 256$ . An error, in this case, means that the label assigned using the algorithm 4 differs from the label assigned by algorithm 1. The pixels for which this is the case are marked as red circles in figure 3.17.



FIGURE 3.17: Binary Split label errors.  
Data set: MNIST,  $\mathcal{P}$ : t-SNE,  $\mathcal{P}^{-1}$ : NNinv  
Resolution:  $256 \times 256$ , Initial blocks resolution:  $B = 32$   
Number of errors:  $100 \cdot \frac{1}{256^2} = 0.0015\%$

Figure 3.18 shows the run-time (in seconds) for different resolution values  $n$  of the algorithms 1 and 4. Observe from figures 3.17 and 3.18 that the proposed heuristic is generating accurate DBM images way faster than the vanilla DBM approach.

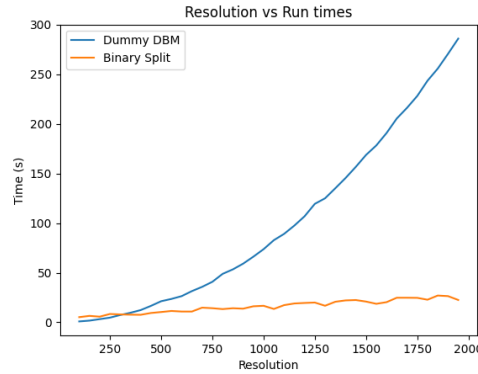


FIGURE 3.18: Binary Split vs Dummy DBM run-times.  
Data set: MNIST,  $\mathcal{P}$ : t-SNE,  $\mathcal{P}^{-1}$ : NNinv  
Initial blocks resolution:  $B = 32$

In order to rigorously test the accuracy of our heuristic algorithms we introduce two metrics described by equations 3.11 and 3.12.

The  $\epsilon_{labels}$  metric describes the percentage of pixels in the DBM image for which the heuristic assigns a different label  $\hat{l}_i$  when compared to the label assigned by Dummy DBM  $l_i$ .  $\delta(l_i, \hat{l}_i)$  is the complement of the Kronecker delta function which equals 0 when  $l_i = \hat{l}_i$  and 1 otherwise.

$$\epsilon_{labels} = 100 \cdot \frac{\sum_{i=1}^{n^2} \delta(l_i, \hat{l}_i)}{n^2} \quad (3.11)$$

The normalized mean square error  $NRMSE_{confidence}$  between the ground truth confidence values  $f_i$  and the approximated confidence values  $\hat{f}_i$  quantifies how well the approximation of the confidence values given by the heuristic matches the ground truth confidence map given by Dummy DBM. For the confidence map, we chose this metric because, in contrast to the labels where we have values coming from a discrete space, the confidence values are coming from a continuous space. Moreover, we are interested in how well a heuristic approximates the ground truth signal globally rather than how well each confidence value is approximated in particular.

$$NRMSE_{confidence} = \frac{\sum_{i=1}^{n^2} (f_i - \hat{f}_i)^2}{\sum_{i=1}^{n^2} f_i^2} \quad (3.12)$$

The relative labels error rate  $\epsilon_{labels}$  for DBM images with different resolutions generated with Binary Split is shown in figure 3.19a. Figure 3.19b presents the error rate  $NRMSE_{confidence}$  of the confidence map generated using several interpolation methods. The confidence values used for interpolation are the confidence values of the pixels for which the classifier was used to compute labels during the execution of the Binary Split heuristic.

Observe that even though the Binary Split is just a heuristic the algorithm is quite accurate. When used in combination with the MNIST data set and the t-SNE

projection the labels error rate  $\epsilon_{labels}$  is not higher than 0.35% and the confidence error rate  $NRMSE_{confidence}$  is not higher than 0.05.

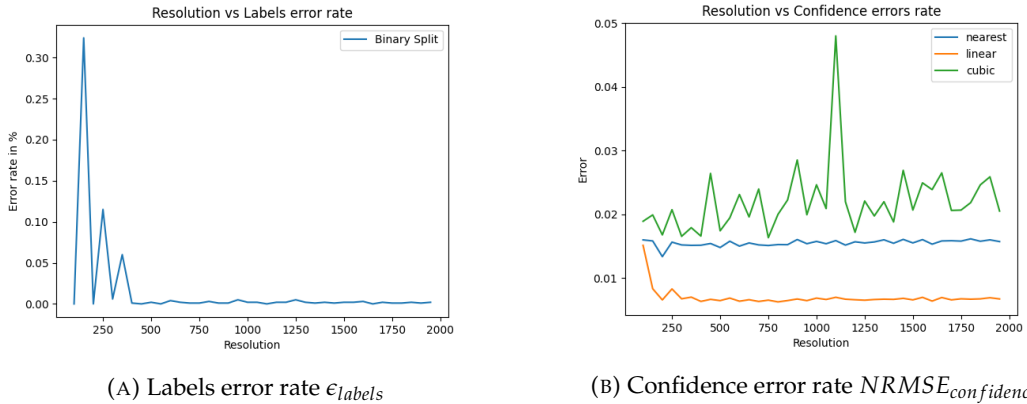


FIGURE 3.19: Binary Split algorithm accuracy

### 3.4.2 Confidence-based Split heuristic

In the binary split heuristic, we are splitting a block into 4 smaller equally sized sub-blocks with respect to a priority queue. Although this approach works, there is one question that still remains, namely: is this approach the most efficient? In this section, we are analyzing how we can use all the information given by the classifier in order to further speed up the DBM computation.

#### Confidence-based Split intuition

Each time we evaluate a pixel's high dimensional counterpart we are not only getting the information about the class label but also the confidence of the classifier that the pixel has this label. Moreover, we get the list of all probabilities the classifier assigns to each class. Let  $d$  be the number of classes in our data set  $D$ , then for an  $n$ -dimensional data point, our classifier  $\mathcal{C}$  yield a list  $p_1, \dots, p_d$ , that represent the probabilities that the data point appertains to each class, i.e.  $\mathcal{C} : [0, 1]^n \rightarrow [0, 1]^d$ ,  $label = \arg \max_{i \in \overline{1, d}} p_i, conf = \max_{i \in \overline{1, d}} p_i$ .

In our next heuristic which we call **Confidence-based Split**, we take a similar approach to the Binary Split, but we are considering the confidence values in order to make better splits of the blocks. A split  $s_1$  of a block  $\mathcal{B}$  is considered to be better than another split  $s_2$  of the same block  $\mathcal{B}$  if we can approximate all the labels of the pixels inside  $\mathcal{B}$  equally good but the number of operations to compute this approximation is less when using  $s_1$  then  $s_2$ .

Figure 3.20a presents a theoretical situation where we have a block for which we computed the central pixel label and confidence using the classifier. The central pixel is labeled as "0" with confidence 0.65. The closest pixels to the central pixel outside of the block are labeled as "1", "2", "3" and "4" with confidence values of 0.87, 0.9, 0.67, and 0.74 respectively. In this figure, a gap between the block and the pixels labeled as "1", "2", "3" and "4" is added in order to show that those pixels have a size and are not just points in the continuous 2D space. These pixels are part of other blocks for which we used the classifier to compute the label and confidence for the central pixels and assigned the same value for all the pixels inside these blocks.

In the situation shown in figure 3.20a, if we were to use the binary split we have to take the 4 sub-blocks as shown, compute the labels of the central pixels and compare each of the new blocks with the closest 4 pixels outside of the block, then repeat this process recursively. On the other hand, if we take a closer look for instance at the point labeled as "1" and the point labeled as "0" we would see that the classifier is more confident in the first prediction (i.e. label "1" is predicted with confidence 0.87, label "0" is predicted with confidence 0.65).

Let us denote the coordinates of the point with the label "1" in figure 3.20 with  $(x_1, y)$  and the coordinates of the point with label "0" with  $(x_2, y)$ . Imagine that we are moving from the point  $(x_1, y)$  in the direction of the point  $(x_2, y)$  (i.e. we are passing through the points  $(tx_1 + (1-t)x_2, y), \forall t \in [0, 1]$ ). Also, imagine that along this way the points we are passing through have the label "1" initially, and then at some point the label changes to "0". Along this way, the confidence will monotonically decrease until a point of inflection where the label changes and the confidence will start increasing. Considering the confidence values of 0.87 and 0.65, the intuition tells us that we will meet this point of inflection  $(t^*x_1 + (1-t^*)x_2, y)$  after half of our way from  $x_1$  to  $x_2$  (i.e.  $t^* > \frac{1}{2}$ ). This means the decision boundary is closer to  $x_2$  than to  $x_1$ .

The 2D line from  $(x_1, y)$  to  $(x_2, y)$  can be described by  $h : [0, 1] \rightarrow \mathbb{R}^2, h(t) = (1-t)(x_1, y) + t(x_2, y)$ . For  $t \in [0, 1]$  we have  $(p_{t,0}, p_{t,1}, \dots, p_{t,d}) = \mathcal{C}(\mathcal{P}^{-1}(h(t)))$ .

Let  $f, g : [0, 1] \rightarrow [0, 1], f(t) = p_{t,1}, g(t) = p_{t,0}$ . In our case the function  $f$  is decreasing as  $t$  is increasing from 0 to 1, whilst  $g$  is increasing. At  $t = 0, f(t) > g(t)$ , at  $t = 1, f(t) < g(t)$ . If we were to know the form of these functions, then we could find fast where the decision boundary is, namely by computing  $t^*$ , where  $f(t^*) = g(t^*)$ . If we assume that these functions are linear, i.e.  $f(t) = at + b, g(t) = ct + d$ , then finding  $t^*$  is trivial,  $t^* = \frac{d-b}{a-c}$ , where  $b = p_{0,1}, d = p_{0,0}, a = p_{1,1} - p_{0,1}, c = p_{1,0} - p_{0,0}$ .

Let us consider the following example:  $\mathcal{C}(\mathcal{P}^{-1}(h(0))) = (0.47, 0.87, \dots)$  and  $\mathcal{C}(\mathcal{P}^{-1}(h(1))) = (0.67, 0.57, \dots)$ . Using the previously described formulas:  $b = 0.87, d = 0.47, a = -0.3, c = 0.2, t^* = \frac{-0.4}{-0.5} = 0.8$ . This means, that if all our assumptions hold the point with coordinates  $(0.2x_1 + 0.8x_2, y)$  is laying on the decision boundary. Splitting the block  $\mathcal{B}$  along  $x$  direction into two sub-blocks that are adjacent at  $0.2x_1 + 0.8x_2$  might make our heuristic faster because we are splitting near the decision boundary (or if the assumptions hold, splitting exactly at the decision boundary along  $x$  direction).

Figure 3.20b presents an example of the split of the same block from figure 3.20a based on the method discussed so far.

Of course, the assumptions we made about confidence values behavior are poor. However, our scope with this approach is not to approximate in one step where the decision boundary lays, but rather to make the splits in such a way that on further iterations the blocks do not need to be split further and we can skip computations.

### Confidence-based Split algorithm

Algorithm 8 represents a modification of the algorithm 4 to include the idea of splitting the blocks based on the confidence values. The subroutines for algorithm 8 are described in procedures 7, 5, 6. Notice that the main difference in the Confidence-based Split algorithm when compared to the Binary Split is the way we are splitting a block into sub-blocks. In the Binary split algorithm, a block is always divided into 4 sub-blocks, whilst in the Confidence-based split algorithm, the number of sub-blocks can vary from 2 to 9 (all the possible configurations are: 2, 3, 4, 6, 9).

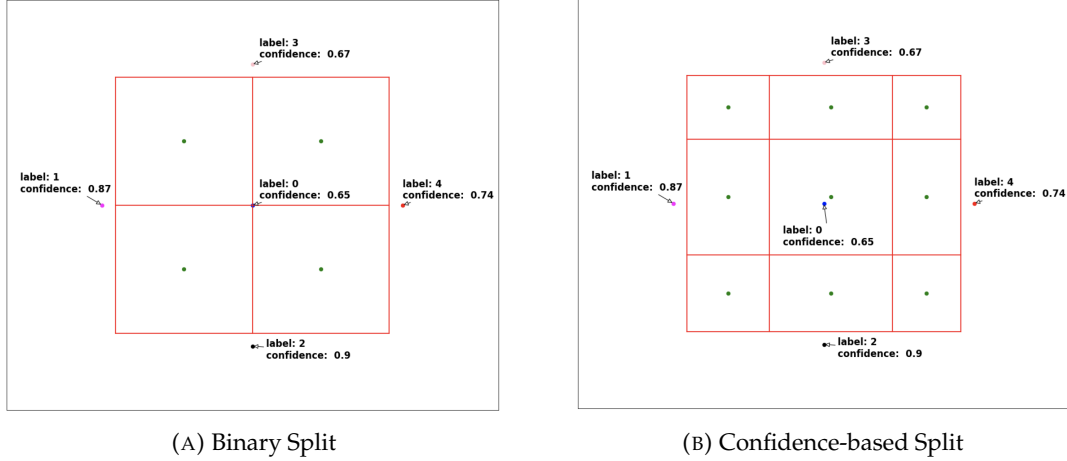


FIGURE 3.20: Block split in "Binary Split" and "Confidence-based Split" heuristics

---

### Algorithm 7 Get the blocks based on confidence split

---

```

1: procedure GET_CONFIDENCE_BASED_SPLIT(dbm, dbm_indexes, probabilities_img, block)
2:   splitsx, splitsy ← GET_SPLITS(dbm, dbm_indexes, probabilities_img, block)
3:   if len(splitsx) == 0 and len(splitsy) == 0 then
4:     return SPLIT_BINARY(block)
5:   end if
6:   return CONSTRUCT_SUB_BLOCKS(splitsx, splitsy, block)
7: end procedure
8: procedure GET_SPLITS(dbm, dbm_indexes, probabilities_img, block)
9:   x, y, w, h ← block
10:  splitsx, splitsy ← [], []
11:  for neighbourx in  $x - \frac{w}{2} - 1, x + \frac{w}{2} + 1$  do
12:    label ← dbm[neighbourx, y]
13:    center_label ← dbm[x, y]
14:    (nx, ny) ← dbm_indexes[neighbourx, y]
15:    if label ≠ center_label then
16:      c1,1, c1,2 ← probabilities_img[x, y][center_label], probabilities_img[nx, y][center_label]
17:      c2,1, c2,2 ← probabilities_img[x, y][label], probabilities_img[nx, y][label]
18:      splitsx ← splitsx + [GET_SPLIT_POSITION(x, nx, neighbourx, c1,1, c1,2, c2,1, c2,2)]
19:    end if
20:  end for
21:  for neighboury in  $y - \frac{h}{2} - 1, y + \frac{h}{2} + 1$  do
22:    label ← dbm[x, neighboury]
23:    center_label ← dbm[x, y]
24:    (nx, ny) ← dbm_indexes[x, neighboury]
25:    if label ≠ center_label then
26:      c1,1, c1,2 ← probabilities_img[x, y][center_label], probabilities_img[x, ny][center_label]
27:      c2,1, c2,2 ← probabilities_img[x, y][label], probabilities_img[x, ny][label]
28:      splitsy ← splitsy + [GET_SPLIT_POSITION(y, ny, neighboury, c1,1, c1,2, c2,1, c2,2)]
29:    end if
30:  end for
31:  return splitsx, splitsy
32: end procedure
33: procedure GET_SPLIT_POSITION(x1, x2, bound, c1,1, c1,2, c2,1, c2,2)
34:   a, b ←  $\frac{c_{1,1} - c_{1,2}}{x_1 - x_2}, c_{1,1} - x_1 * \frac{c_{1,1} - c_{1,2}}{x_1 - x_2}$ 
35:   c, d ←  $\frac{c_{2,1} - c_{2,2}}{x_1 - x_2}, c_{2,1} - x_1 * \frac{c_{2,1} - c_{2,2}}{x_1 - x_2}$ 
36:   if a == c then
37:     return
38:   end if
39:   boundary ← round( $\frac{d-b}{a-c}$ )
40:   if bound < x1 and bound + 1 < boundary < x1 then
41:     return boundary
42:   end if
43:   if bound > x1 and x1 < boundary < bound - 1 then
44:     return boundary
45:   end if
46:   return
47: end procedure
48: procedure SPLIT_BINARY(block)
49:   x, y, w, h ← block
50:    $\delta_w, \delta_h$  ←  $\frac{w}{2}, \frac{h}{2}$ 
51:   return [(x -  $\frac{\delta_w}{2}, y - \frac{\delta_h}{2}, \delta_w, \delta_h$ ), (x -  $\frac{\delta_w}{2}, y + \frac{\delta_h}{2}, \delta_w, \delta_h$ ), (x +  $\frac{\delta_w}{2}, y - \frac{\delta_h}{2}, \delta_w, \delta_h$ ), (x +  $\frac{\delta_w}{2}, y + \frac{\delta_h}{2}, \delta_w, \delta_h$ )]
52: end procedure

```

---

**Algorithm 8** DBM Confidence-based split

---

**Input:**  $\mathcal{P}^{-1} : [0, 1]^2 \rightarrow [0, 1]^n$ ,  $\mathcal{C} : [0, 1]^n \rightarrow [0, 1]^d$ ,  $n$  (int),  $B$  (int), *interpolation\_method* (string)  
**Output:**  $dbm \in \mathcal{M}_{n,n}(\mathbb{N})$   
 $confmap \in \mathcal{M}_{n,n}([0, 1])$

- 1:  $num\_classes \leftarrow$  number of classes in the data set  $D$
- 2:  $dbm \leftarrow \mathcal{O}_{n,n}$
- 3:  $dbm\_indexes \leftarrow \mathcal{O}_{n,n}$
- 4:  $probabilities\_img \leftarrow \mathcal{O}_{n,n,num\_classes}$
- 5:  $confmap \leftarrow []$
- 6:  $blocks \leftarrow GENERATE\_INITIAL\_BLOCKS(n, B)$
- 7: **for**  $block$  in  $blocks$  **do**
- 8:    $x, y, w, h \leftarrow block$
- 9:    $probabilities \leftarrow \mathcal{C}(\mathcal{P}^{-1}(x/n, y/n))$
- 10:    $label, conf \leftarrow \arg \max(probabilities), \max(probabilities)$
- 11:    $left, right, top, bottom \leftarrow x - \frac{w}{2}, x + \frac{w}{2}, y - \frac{h}{2}, y + \frac{h}{2}$
- 12:    $dbm[left : right][top : bottom] \leftarrow label$
- 13:    $dbm\_indexes[left : right][top : bottom] \leftarrow (x, y)$
- 14:    $probabilities\_img[left : right][top : bottom] \leftarrow probabilities$
- 15:    $confmap \leftarrow confmap + [(x, y, conf)]$
- 16: **end for**
- 17:  $priority\_queue \leftarrow PriorityQueue()$  // initialize a priority queue
- 18: **for**  $block$  in  $blocks$  **do**
- 19:    $INSERT\_PRIORITY\_QUEUE(priority\_queue, dbm, block)$
- 20: **end for**
- 21: **while** not  $priority\_queue.empty()$  **do**
- 22:   // pop the blocks with the highest priority from the queue
- 23:    $priority, blocks \leftarrow GET\_BLOCKS\_WITH\_HIGHEST\_PRIORITY(priority\_queue)$
- 24:   **for**  $block$  in  $blocks$  **do**
- 25:      $x, y, w, h \leftarrow block$
- 26:     **if**  $w == 1$  and  $h == 1$  **then**
- 27:        $probabilities \leftarrow \mathcal{C}(\mathcal{P}^{-1}(x/n, y/n))$
- 28:        $label, conf \leftarrow \arg \max(probabilities), \max(probabilities)$
- 29:        $dbm[x][y] \leftarrow label$
- 30:        $dbm\_indexes[x][y] = (x, y)$
- 31:        $probabilities\_img[x][y] \leftarrow probabilities$
- 32:        $confmap \leftarrow confmap + [(x, y, conf)]$
- 33:       **continue**
- 34:     **end if**
- 35:      $sub\_blocks \leftarrow GET\_CONFIDENCE\_BASED\_SPLIT(dbm, dbm\_indexes, probabilities\_img, block)$
- 36:     **for**  $sub\_block$  in  $sub\_blocks$  **do**
- 37:        $x, y, w, h \leftarrow sub\_block$
- 38:        $probabilities \leftarrow \mathcal{C}(\mathcal{P}^{-1}(x/n, y/n))$
- 39:        $label, conf \leftarrow \arg \max(probabilities), \max(probabilities)$
- 40:        $dbm[x - w : x + w][y - h : y + h] \leftarrow label$
- 41:        $dbm\_indexes[x - w : x + w][y - h : y + h] \leftarrow (x, y)$
- 42:        $probabilities\_img[x - w : x + w][y - h : y + h] \leftarrow probabilities$
- 43:        $confmap \leftarrow confmap + [(x, y, conf)]$
- 44:     **end for**
- 45:     **for**  $sub\_block$  in  $sub\_blocks$  **do**
- 46:        $INSERT\_PRIORITY\_QUEUE(priority\_queue, dbm, sub\_block)$
- 47:     **end for**
- 48:   **end for**
- 49: **end while**
- 50:  $confmap \leftarrow INTERPOLATE(confmap, n, interpolation\_method)$
- 51: **return**  $dbm, confmap$

---

**Confidence-based Split heuristic complexity analysis**

In this paragraph, we analyze the complexity of algorithm 8. In the best-case scenario, when the data set has only one class, the complexity of the Confidence-based Split does not differ from the complexity of the Binary Split since we are doing the same operations. Therefore, the complexity, in this case, is  $O(B^2C + n^2)$ . In the worst-case scenario, the Confidence-based Split algorithm will have to call  $\mathcal{C}(\mathcal{P}^{-1}(\cdot))$  for each of the  $n^2$  pixels.

Thus, the worst time complexity is:

$$O(B^2C + c_1B^2(C + \log(B^2)) + \dots + c_lB^2(C + \log(c_{l-1}B^2)) + n^2C + n^2) \quad (3.13)$$

where  $c_i B^2 < n^2, \forall i \in \overline{1, l}$  and  $l$  is the number of iterations the algorithm makes.

The worst split possible is when we split a block  $\mathcal{B}$  of size  $w \times h$  into only two blocks  $\mathcal{B}_1$  with size  $w \times (h - 1)$  and  $\mathcal{B}_2$  with size  $w \times 1$ . Alternatively splitting into  $\mathcal{B}_1$  with size  $(w - 1) \times h$  and  $\mathcal{B}_2$  with size  $1 \times h$ . Supposing that we have this situation for all the blocks at any iteration  $i$ , then  $c_i = 2c_{i-1}, c_0 = 1$ . Since  $c_i B^2 \leq n^2$  and  $c_i = 2^i$ , we have that  $2^i \leq (\frac{n}{B})^2$  and  $l \leq 2 \log_2(\frac{n}{B})$ . The worst-case complexity then becomes:

$$O(B^2 C + n^2 \log(\frac{n}{B})(C + \log n)) = O(n^2 \log(\frac{n}{B})(C + \log n)) \quad (3.14)$$

Using the same argumentation as in section 3.4.1, the worst-case complexity can be bounded by  $O(n^2 C)$  by means of a computational budget parameter.

In the average case the term  $c_i B^2 \leq Kn$ , where  $K$  is a constant defined by the classifier, inverse projection, and the data set (see section 3.4.1). Using the same argumentation as for the worst-case complexity the number of iterations  $l$  is bounded by  $2 \log_2(\frac{n}{B})$ . Thus, the average case complexity is:

$$\begin{aligned} O(B^2 C + c_1 B^2 (C + \log(B^2)) + \dots + c_l B^2 (C + \log(c_{l-1} B^2)) + n^2) = \\ O(B^2 C + Kn \log(\frac{n}{B})(C + \log n) + n^2) \end{aligned} \quad (3.15)$$

### Confidence-based Split vs Binary Split vs Vanilla DBM

In this paragraph, we repeat the experiments we performed when comparing the Binary split heuristic with the Dummy DBM in section 3.4.1 and extend by adding the newly introduced Confidence-based Split heuristic. The complete global comparison of all the DBM optimization methods is presented in section 3.4.4.

Figure 3.21 presents the label errors when the labels are generated using the Confidence-based Split algorithm for the resolution  $256 \times 256$  (compare with figure 3.17). The pixels for which the labels differ when using algorithms 1 and 8 are marked as red circles.



FIGURE 3.21: Confidence-based Split label errors.  
 Data set: MNIST,  $\mathcal{P}$ : t-SNE,  $\mathcal{P}^{-1}$ : NNinv  
 Resolution:  $256 \times 256$ , Initial blocks resolution:  $B = 32$   
 Number of errors:  $100 \cdot \frac{8}{256^2} = 0.0122\%$



Figure 3.22 shows the run-time (in seconds) for different resolution values of the algorithms 1, 4 and 8. Observe that the Confidence-based Split algorithm takes less to compute the DBM when compared to the brute force Dummy DBM algorithm. However, the Binary Split heuristic seems to be the fastest. This might be caused by the fact that the different way of block splitting is leading the algorithm to generate more blocks than the Binary Split, which causes the algorithm to make more calls to the classifier. The total number of blocks with neighbors having the same label as the central pixel of the block is smaller than in the Binary Split case which makes the algorithm 4 cut down more redundant operations. Another explanation might be that for this specific projection method, the decision boundaries are placed in such a way in 2D that simply makes the "binary" way of splitting a block the most optimal. In chapter 4 we generate the DBM for different projection methods. This series of experiments answers the question of which algorithm brings better run times.

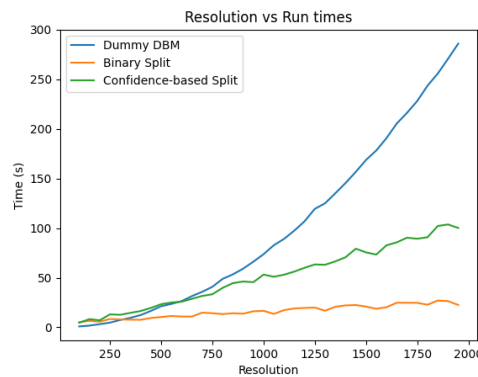


FIGURE 3.22: Confidence-based Split vs Binary split vs Dummy DBM run-times. Data set: MNIST,  $\mathcal{P}$ : t-SNE,  $\mathcal{P}^{-1}$ : NNinv  
Initial blocks resolution:  $B = 32$

The labels error rate  $\epsilon_{labels}$  of DBMs generated with Confidence-based Split for different resolutions is shown in figure 3.23a. Observe that the Binary Split is performing better than the Confidence-based Split in terms of label accuracy, even though the differences are small. Again this behaviour might be caused by the different ways of splitting the blocks which causes the algorithms to generate different sets of blocks. Figure 3.23b presents the error rates  $NRMSE_{confidence}$  in the confidence maps generated using different interpolation methods. For the nearest neighbor and bilinear interpolation methods, the  $NRMSE_{confidence}$  metric gives almost the same values for Binary and Confidence-based Split algorithms.

### 3.4.3 Confidence interpolation heuristic

In the Confidence-based Split heuristic, algorithm 8, we perform a local interpolation based on the confidence values given by the classifier to find where we should split a block. If we are splitting the image into a set of blocks and computing the confidence values for the centers of the blocks, why not use these samples to approximate directly the confidence functions? This will reduce the computation cost since after we have these functions we can use them to compute the confidence values for each pixel instead of using the classifier and the inverse projection function which is a costly operation. In this section, we present another heuristic we call **Confidence Interpolation**, that aims to reduce the  $C$  term in the  $O(n^2C)$  complexity of the algorithm 1.

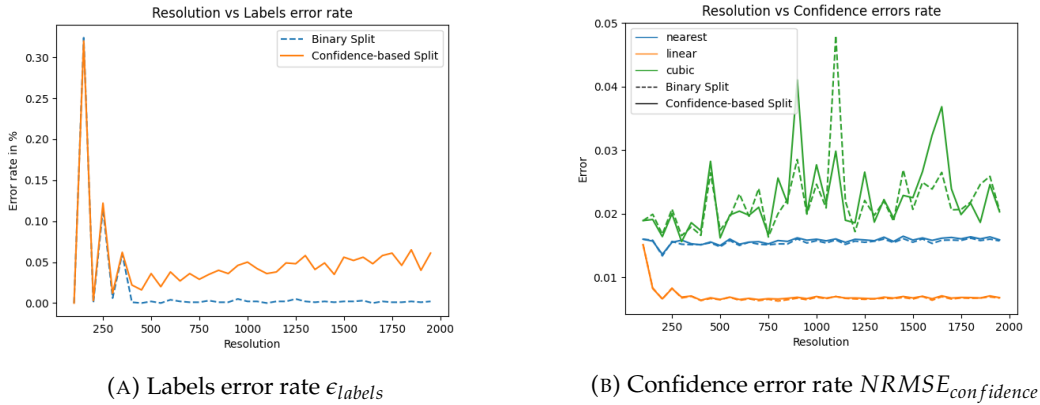


FIGURE 3.23: Confidence-based Split algorithm accuracy

### Confidence interpolation intuition

Let  $d$  be the number of classes in our data set  $D$ . For each pixel  $(x, y)$  the composition  $\mathcal{C}(\mathcal{P}^{-1}(x, y))$  is returning a tuple of probabilities  $(p_1, \dots, p_d)$ . Let  $f_1, \dots, f_d : [0, 1]^2 \rightarrow [0, 1]$ , such that  $\mathcal{C}(\mathcal{P}^{-1}(x, y)) = (f_1(x, y), \dots, f_d(x, y))$ . Algorithm 1 can then be easily re-written so that instead of the usage of  $\mathcal{C}(\mathcal{P}^{-1}(\cdot))$  we use the functions  $f_1, \dots, f_d$ . Namely, for each pixel  $(x, y)$  the label of that pixel is determined by  $\arg \max_{i \in \overline{1, d}} f_i(x, y)$  and the confidence is  $\max_{i \in \overline{1, d}} f_i(x, y)$ .

If we would have a fast and cheap way to infer the functions  $f_1, \dots, f_d$  and the cost of the call to all these functions would be lower than the call to  $\mathcal{C}(\mathcal{P}^{-1}(\cdot))$  we can speed up the algorithm 1.

We can infer the functions  $f_1, \dots, f_d$  by taking a set of samples and applying an interpolation method (e.g. bilinear interpolation). Let  $B^2$  be the number of samples that we are taking in order to approximate the functions. For each such sample, we need to compute the actual probability values using the classifier and the inverse projection. In our implementation, we are taking these samples as follows: given the desired resolution  $W \times H$  of the decision boundary map image the user wants to generate, split the image into  $B^2$  blocks, then take the center of each block as a sample. The complexity of getting the samples and generating the probabilities values is  $O(B^2C)$  where  $C$  is the cost of the operation  $\mathcal{C}(\mathcal{P}^{-1}(\cdot))$ .

Once we have the  $B^2$  samples, we can perform a local interpolation (e.g. bilinear interpolation, see figure 3.24) to compute the probabilities values for each pixel in the DBM image we want to generate. Our implementation allows three interpolation methods: nearest neighbor, bilinear, and bicubic. All these methods are doing local interpolation, which means that for a pixel  $(x, y)$  we are computing each probability vector  $(p_1, \dots, p_d)$  based on the neighbors of that pixel. The number of neighbors we use is constant and the number of operations we are performing is also constant. Thus, finding  $f_i(x, y), i \in \overline{1, d}$  has the complexity  $O(1)$ . The complexity of computing all the probability vectors for all the pixels of an image of size  $n \times n$  is, therefore,  $O(n^2d)$ .

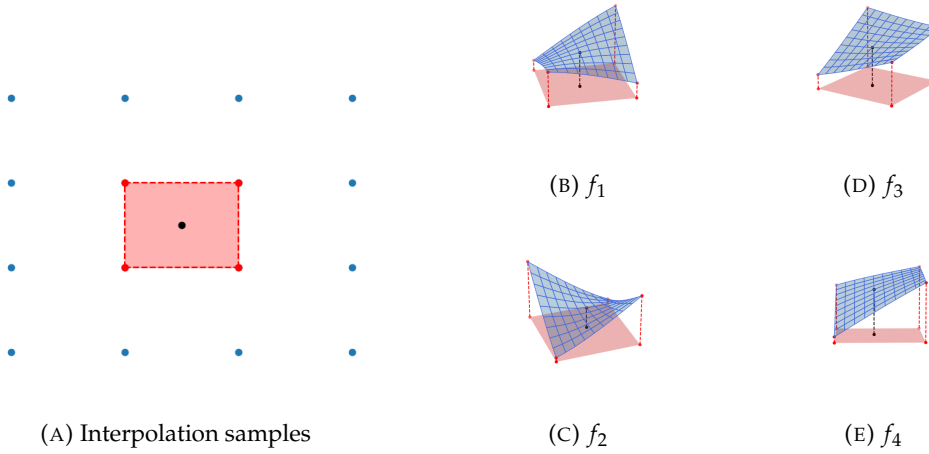


FIGURE 3.24: Confidence interpolation intuition for generating a DBM with 4 classes using bilinear interpolation

### Confidence interpolation algorithm

Algorithm 9 formalizes the intuition presented previously. It starts by generating  $B^2$  samples which are central pixels of  $B^2$  blocks. For each pixel  $(x, y)$  and each class  $c$  the algorithm uses the interpolation to find  $f_c(x, y)$ . Then for each pixel, the algorithm assigns the label and the confidence value based on the values of  $f_i(x, y), \forall i \in \overline{1, d}$ .

---

#### Algorithm 9 DBM Confidence-interpolation

---

**Input:**  $\mathcal{P}^{-1} : [0, 1]^2 \rightarrow [0, 1]^n, \mathcal{C} : [0, 1]^n \rightarrow [0, 1]^d, n$  (int),  $B$  (int), *interpolation\_method* (string)  
**Output:**  $dbm \in \mathcal{M}_{n,n}(\mathbb{N})$   
 $confmap \in \mathcal{M}_{n,n}([0, 1])$

- 1:  $num\_classes \leftarrow$  number of classes in the data set  $D$
- 2:  $dbm \leftarrow \mathcal{O}_{n,n}$
- 3:  $probabilities\_map \leftarrow []$
- 4:  $conf\_map \leftarrow \mathcal{O}_{n,n}$
- 5:  $img\_confidences \leftarrow \mathcal{O}_{n,n,num\_classes}$
- 6:  $blocks \leftarrow GENERATE\_INITIAL\_BLOCKS(n, B)$
- 7: **for**  $block$  in  $blocks$  **do**
- 8:    $x, y, w, h \leftarrow block$
- 9:    $probabilities \leftarrow \mathcal{C}(\mathcal{P}^{-1}(x/n, y/n))$
- 10:    $probabilities\_map \leftarrow probabilities\_map + [(x, y, probabilities)]$
- 11: **end for**
- 12: **for**  $k \leftarrow 0$  to  $num\_classes$  **do**
- 13:    $map \leftarrow []$
- 14:   **for**  $item$  in  $probabilities\_map$  **do**
- 15:      $(x, y, probabilities) \leftarrow item$
- 16:      $map \leftarrow map + [(x, y, probabilities[k])]$
- 17:   **end for**
- 18:    $img\_confidences[:, :, k] \leftarrow INTERPOLATE(map, n, interpolation\_method)$
- 19: **end for**
- 20: **for**  $i \leftarrow 0$  to  $W$  **do**
- 21:   **for**  $j \leftarrow 0$  to  $H$  **do**
- 22:      $dbm[i, j] \leftarrow \arg \max(img\_confidences[i, j])$
- 23:      $confmap[i, j] \leftarrow \max(img\_confidences[i, j])$
- 24:   **end for**
- 25: **end for**
- 26: **return**  $dbm, confmap$

---

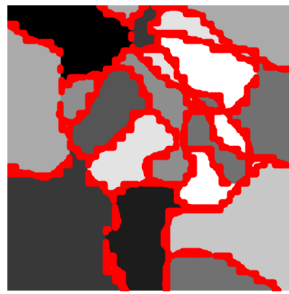
### Confidence interpolation heuristic complexity analysis

Let  $B$  represent the initial resolution we are considering in order to get the points that decide the interpolation. We need to decode all these points (i.e. use the classifier and the inverse projection to get the probability vectors), which means we need to perform  $B^2C$  operations. Then for each of  $n^2$  pixels, we need to calculate the probability of each of the  $d$  classes. Thus, the complexity of the algorithm 9 is  $O(B^2C + n^2d)$ .

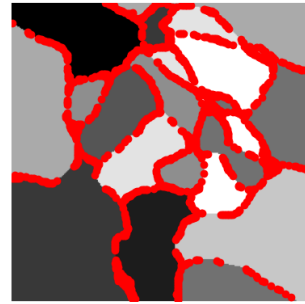
### Confidence interpolation vs Vanilla DBM

In this paragraph, we compare the Confidence Interpolation heuristic in terms of run times with the Dummy DBM (algorithm 1). Furthermore, we are analyzing the accuracy of the algorithm 9. The complete global comparison between all the DBM generation algorithms is presented in section 3.4.4.

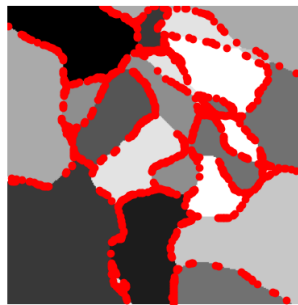
Figure 3.25 presents the label errors when the labels are assigned using algorithm 9 using different interpolation methods and the initial blocks resolution of  $B = 32$  (compare with figures 3.17 and 3.21). The pixels with label errors are marked with red circles.



(A) Interpolation method: nearest neighbor  
 $\epsilon_{labels}: 100 \cdot \frac{3947}{256^2} = 6.0226\%$



(B) Interpolation method: bilinear  
 $\epsilon_{labels}: 100 \cdot \frac{1413}{256^2} = 2.1560\%$



(C) Interpolation method: bicubic  
 $\epsilon_{labels}: 100 \cdot \frac{953}{256^2} = 1.4541\%$

FIGURE 3.25: Confidence Interpolation label errors.  
 Data set: MNIST,  $\mathcal{P}$ : t-SNE,  $\mathcal{P}^{-1}$ : NNinv  
 Resolution:  $256 \times 256$ , Initial blocks resolution:  $B = 32$

Figure 3.26 shows how the Confidence Interpolation algorithm performs in terms of run times for different resolutions when compared to the algorithm 1. Observe that this heuristic is orders of magnitude faster than the Dummy DBM algorithm.

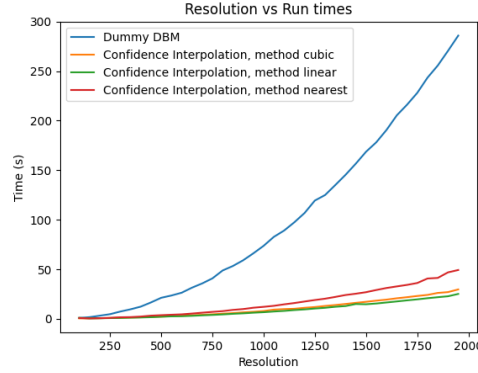
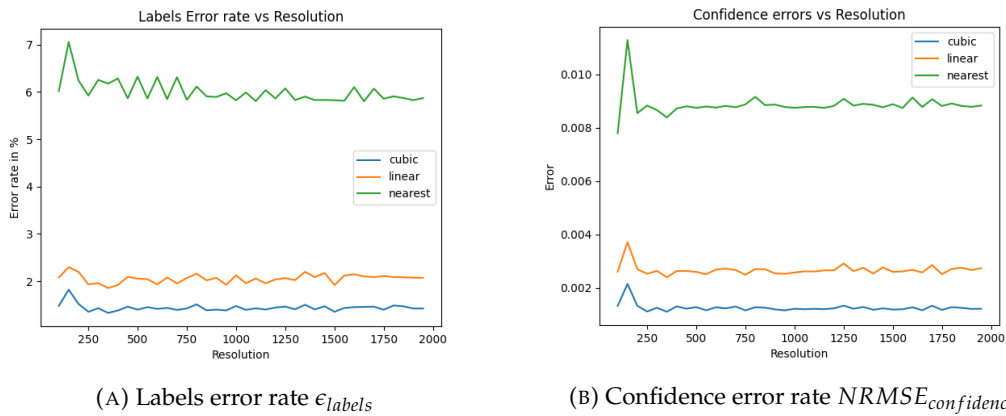


FIGURE 3.26: Confidence Interpolation vs Dummy DBM run-times.  
Data set: MNIST,  $\mathcal{P}$ : t-SNE,  $\mathcal{P}^{-1}$ : NNinv  
Initial blocks resolution:  $B = 32$

The label error rate  $\epsilon_{labels}$  when using algorithm 9 is shown in figure 3.27a. The errors in the confidence map  $NRMSE_{confidence}$  are shown in figure 3.27b. Observe that when the nearest neighbor interpolation method is used we achieve the worst labels and confidence values approximations. The most accurate results are achieved when the Confidence Interpolation algorithm is used in combination with the bicubic interpolation method.



(A) Labels error rate  $\epsilon_{labels}$

(B) Confidence error rate  $NRMSE_{confidence}$

FIGURE 3.27: Confidence Interpolation algorithm accuracy

### 3.4.4 DBM generation algorithms comparison

In this section, we compare all the heuristics introduced so far for the computation of the decision boundary maps. We are comparing the methods in terms of run-time complexity and accuracy. Moreover, we provide a discussion about the trade-offs of using each algorithm and in which context one algorithm is better than another. For each experiment and each heuristic, we use the initial number of blocks  $B = 32$ .

### Run time complexity analysis

Table 3.3 presents the theoretical complexity classes for each algorithm. Figure 3.28 presents a comparison in terms of run-times of all the 3 introduced heuristics and the algorithm 1 for different values of  $n$ .

If we are looking from the perspective of the best-case complexity in table 3.3 the Binary and Confidence-based Split heuristics are clear winners. On the other hand, in the worst case, these two heuristics might perform even worse than the Dummy DBM algorithm. In the worst-case scenario time-wise the Confidence Interpolation algorithm seems to be the best. In the average case scenario  $\exists n$  such that  $n(d-1) > K \log(\frac{n}{B})(C + \log n)$ . Therefore, for big enough values of  $n$  the Binary and Confidence-based Split heuristics might perform better than the Confidence interpolation. Notice in figure 3.28 that this is the case for the Binary Split heuristic that starts to perform better than the Confidence Interpolation for high-resolution DBM images.

For small values of  $n$ , the Dummy DBM algorithm is outperforming in terms of speed the Binary and Confidence-base Split heuristics. However, notice the advantages of the heuristics when  $n$  increases. From the perspective of run-time complexity in practice when used together with the t-SNE projection and NNinv inverse projection method the Binary Split and Confidence Interpolation algorithms are the fastest. However, we can not state that this will be the case for other projection methods as well, because the performance of our algorithms depends on how the data is placed geometrically in the 2D space. Therefore, in chapter 4 we perform a broad series of experiments where we involve all our heuristics in generating DBM when different projection and inverse projection methods are used.

Algorithm	Best case complexity	Average case complexity	Worst case complexity
Dummy DBM (algorithm 1)	$O(n^2C)$	$O(n^2C)$	$O(n^2C)$
Binary Split (algorithm 4)	$O(B^2C + n^2)$	$O(B^2C + Kn \log(\frac{n}{B})(C + \log(n)) + n^2)$	$O(n^2 \log(\frac{n}{B})(C + \log n))$
Confidence-based Split (algorithm 8)	$O(B^2C + n^2)$	$O(B^2C + Kn \log(\frac{n}{B})(C + \log(n)) + n^2)$	$O(n^2C)$ (when computational budget included) $O(n^2 \log(\frac{n}{B})(C + \log n))$
Confidence Interpolation (algorithm 9)	$O(B^2C + n^2d)$	$O(B^2C + n^2d)$	$O(n^2C)$ (when computational budget included) $O(B^2C + n^2d)$

TABLE 3.3: DBM heuristics complexity analysis

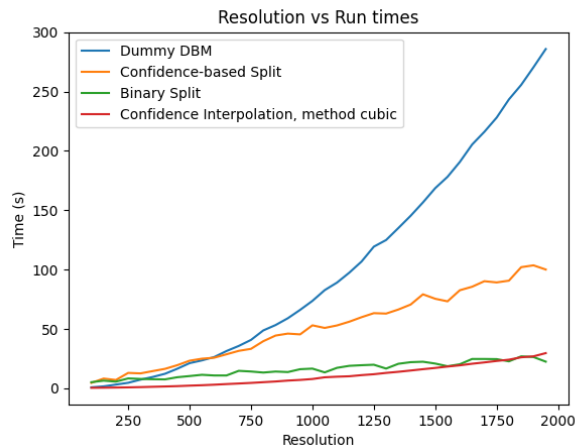


FIGURE 3.28: DBM heuristics run-times comparison.

Data set: MNIST,  $\mathcal{P}$ : t-SNE,  $\mathcal{P}^{-1}$ : NNinv,

Initial blocks resolution:  $B = 32$

### Accuracy analysis

In order to compare how each heuristic performs in terms of accuracy for different resolutions we compute the label error rates  $\epsilon_{label}$  and the confidence error rates  $NRMSE_{confidence}$  for different DBM image resolutions using all the algorithms described so far and present the results in figure 3.29.

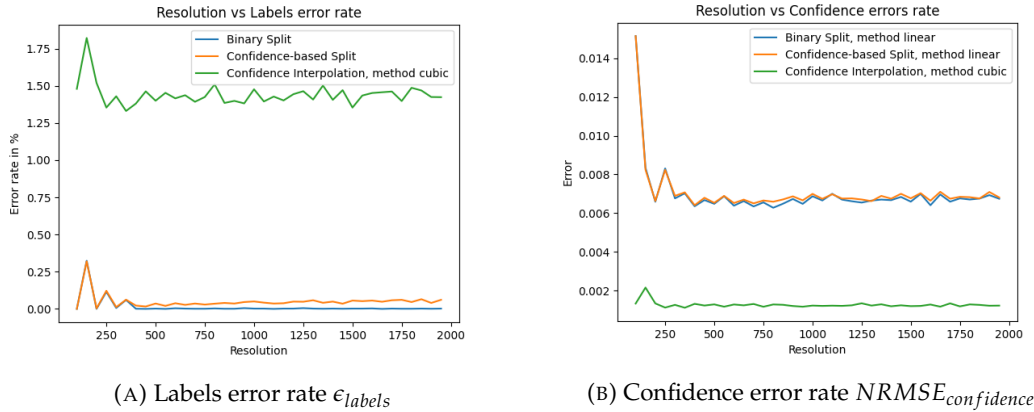


FIGURE 3.29: DBM algorithms accuracy comparison.

Data set: MNIST,  $\mathcal{P}$ : t-SNE,  $\mathcal{P}^{-1}$ : NNinv

Initial blocks resolution:  $B = 32$

Confidence map approximation method: Bilinear for Binary and Confidence-base split, bicubic for the Confidence Interpolation algorithm

Observe from figure 3.29a that the number of errors in regards to the labels (i.e.  $\epsilon_{labels}$ ) is bigger when the Confidence Interpolation heuristic is involved when compared to the Binary and Confidence-based Split. These differences can be explained as follows. Look at one of the green regions from figure 3.30 and imagine the central pixel of this region. For the high-dimensional counterpart of that central pixel let us assume that the classifier assigns a 0.51 probability of it having the green label and 0.49 for the red label. For simplicity assume that the classifier assigns all the pixels in red regions with probability/confidence 1. If we use the Confidence Interpolation algorithm and construct the functions  $f_1$  and  $f_2$  that assign the confidence for red and green labels respectively, then for some of the pixels in the green region we will have  $f_1 > f_2$ , which will give us label errors, all because of the fact that the central pixel probability vector of a block and the probability vectors of the central pixels of the neighboring blocks are directly determining all the labels of the pixels in the block. Whereas when using the Binary or Confidence Split heuristic, the block will be divided into sub-blocks and an analysis of the new central pixels of the sub-blocks will be performed. Thus, the initial central pixel determines just to some extent the labels of the other pixels in the blocks but not completely. Hence the label errors in the Binary and Confidence split heuristics are smaller.

Another argumentation is that at the boundaries the classifier is quite uncertain about which label to assign, and the probability vector  $(p_1, \dots, p_d)$  the classifier generates will have in the majority of the cases at least two very similar probabilities (i.e.  $\exists i, j \in \{1, d\}$  such that  $|p_i - p_j| < \epsilon$ , where  $\epsilon \geq 0$  is a very small number). Even though the approximation method in the Confidence Interpolation algorithm makes it possible to approximate the probabilities vectors quite well (see  $NRMSE_{confidence}$  in figure 3.29b) even very small approximation errors might turn into label errors. This type of error will especially appear at the decision boundaries, which can be

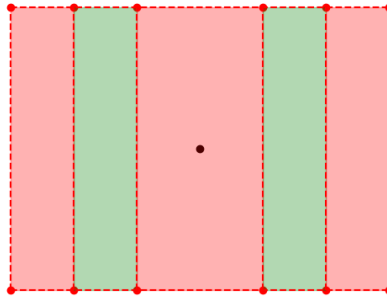


FIGURE 3.30: Explanation of label errors

seen in figure 3.25 as well. On the other hand, the Binary Split and Confidence Split algorithms do not use any approximations to infer the labels and, thus are more accurate in generating DBMs with correct labels.

Even though the Confidence Interpolation algorithm gives more label errors, it also gives more accurate confidence maps (see figure 3.29b) when compared to the other two heuristics. This is caused by the fact of how the confidence maps are generated. In the case of the Confidence Interpolation method, approximations of the probability vectors are used, whereas in the case of Binary or Confidence-based Split, only the confidence values are used to approximate the confidence map.

### Discussions

In this chapter, we used only the t-SNE projection and the NNinv inverse projection to test our DBM generation algorithms. In all our tests we used the initial block's resolution as  $B = 32$ . For this setup, we identified that for the Confidence Interpolation algorithm, the best results are achieved when used together with the bicubic interpolation. The confidence map is best approximated when the bilinear interpolation method is used in the Binary and Confidence-based Split algorithms. For the current setup seems that the Binary Split algorithm is the best to use if we want a fast and accurate alternative for the Dummy DBM algorithm. In chapter 4 we provide a broader set of tests in order to make more accurate and general statements about the performance and usability of our heuristics.

## 3.5 Visualization tool overview

The previous sections describe the implementation details of different functionalities of our visualization tool. In this section, we present our visualization tool from a user perspective and give a guide on how the user can make use of the functionalities of the tool. Figure 3.31 presents a simplified pipeline of user interaction with the visualization tool. The user initially interacts with a configuration window where he/she uploads the data set and the classifier, chooses the technique for DBM generation, and generates the DBM. Our visualization tool does not perform the automatic labeling of a semi-labeled data set. The user can choose the preferred automatic pseudo-labeling method (e.g. OPF, LP, DeepFA, etc.) and then use the visualization tool to correct wrong labels by re-labeling data points. Therefore, the visualization tool takes a completely labeled data set. After the DBM image is generated the user



can interact with a visualization window where he/she can re-label data points and re-train the classifier iteratively until the results match the user's expectations. At each step, after the classifier is retrained a copy of the classifier along with other relevant metadata is stored in a temporary folder that the user can access after the tool is closed.

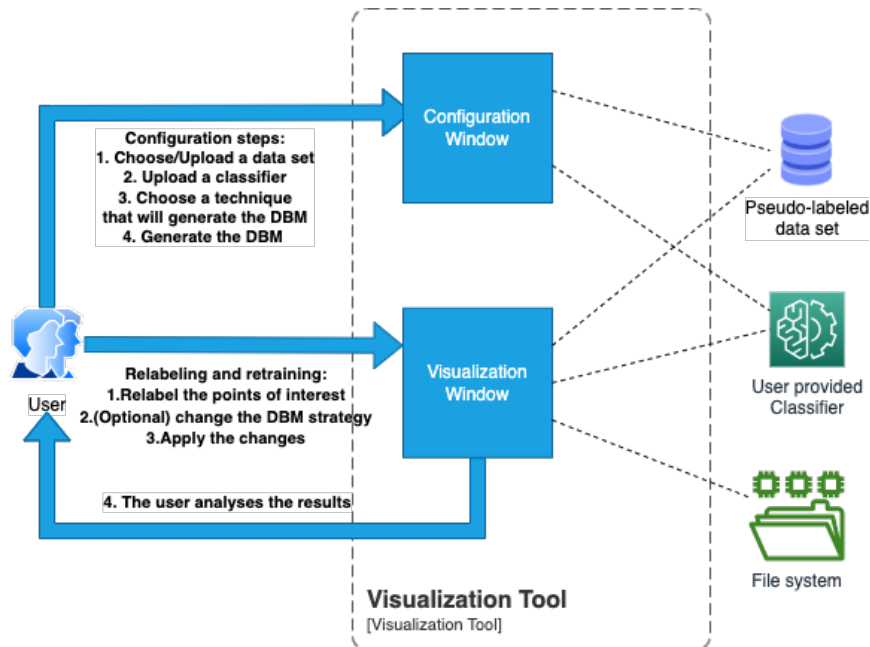


FIGURE 3.31: Visualization tool usage workflow

### 3.5.1 Configuration window

Initially, when the user starts the visualization tool, a configuration window pops up as the one shown in figure 3.32. In this window, the user has to perform the following steps to generate a Decision Boundary Map (see figure 3.31, configuration steps):

- Step 1: Choose/Upload a data set
- Step 2: Upload a classifier
- Step 3: Choose the technique/algorithm that will generate the DBM.
- Step 4: Press the "Show the Decision Boundary Mapping" button in order to generate the DBM.

**Step 1: Choosing/Uploading a data set to the visualization tool** The upload of a data set can be performed in two ways. The user might opt for one of 3 data sets that are supported by default, namely: MNIST, Fashion MNIST, or CIFAR10. In this case, the uploading is done by just clicking the corresponding button as shown in figure A.3a (the buttons are surrounded by red rectangles). In case the data set is not one of these pre-determined data sets, the data set can be uploaded from a local folder by selecting the files with the data set. In this case, we ask the user to provide a file or folder with a training subset and a file or folder with the testing subset. Once the respective file/folder is selected the user can click the corresponding button to

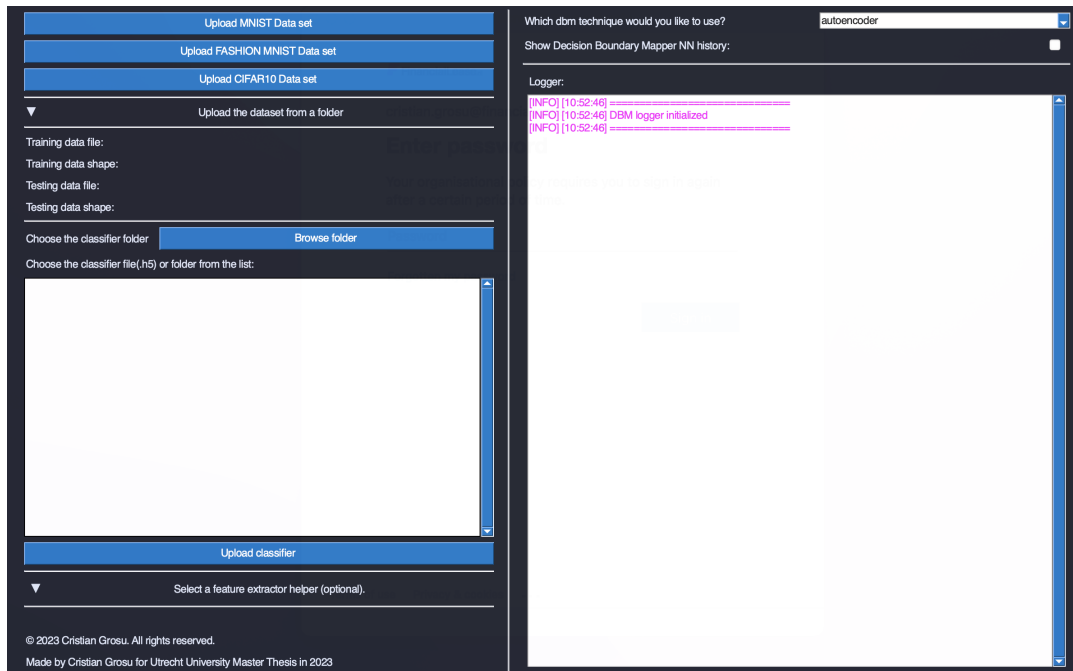


FIGURE 3.32: Configuration window of the visualization tool

upload either of the subsets as shown in figure A.3b (the related parts of the UI are surrounded by red rectangles). After the data set was successfully uploaded metadata about the shape of the training and testing sets will appear as shown in figure A.3c (the related UI components are surrounded by a green rectangle). In case the data set can not be uploaded an error message will be shown in the Logger part of the UI.

**Step 2: Uploading a classifier to the visualization tool** The uploading of the classifier to the visualization tool can be performed as shown in figure A.1 (the relevant UI parts are surrounded by red rectangles). The user needs to select the folder in which the classifier is stored, the classifier can be stored in a file with a ".h5" extension or a folder. After selecting the necessary file/folder the user can click on the "Upload classifier" button. If both the data set and the classifier were uploaded the user will see the "Show the Decision Boundary Mapping" button as shown in figure A.1 (surrounded by a green rectangle).

**Step 3: Choosing the Decision boundary mapping generation technique** In this step, the user can opt for one of the two techniques, to generate the decision boundary map. The first technique called SDBM has two options, namely *Autoencoder* and *SSNP*. By choosing one of these two options, both the direct and inverse projections are learned by the visualization tool. The user has the option to provide the direct projection ( $nD$  to  $2D$ ) by choosing the *NNinv* option as shown in figure A.4a. If this option is selected a drop-down menu with 4 options will appear as shown in figure A.4b asking for a projection method to be chosen. The user can choose one of the predefined projection methods such as: *t-SNE*, *UMAP* or *PCA*, or can provide a custom projection by choosing the option *CUSTOM* in this menu. If the custom option is chosen the user will see a UI element that will allow him to upload the 2D projections of the training and testing subsets of the data set as shown in figure A.4c.

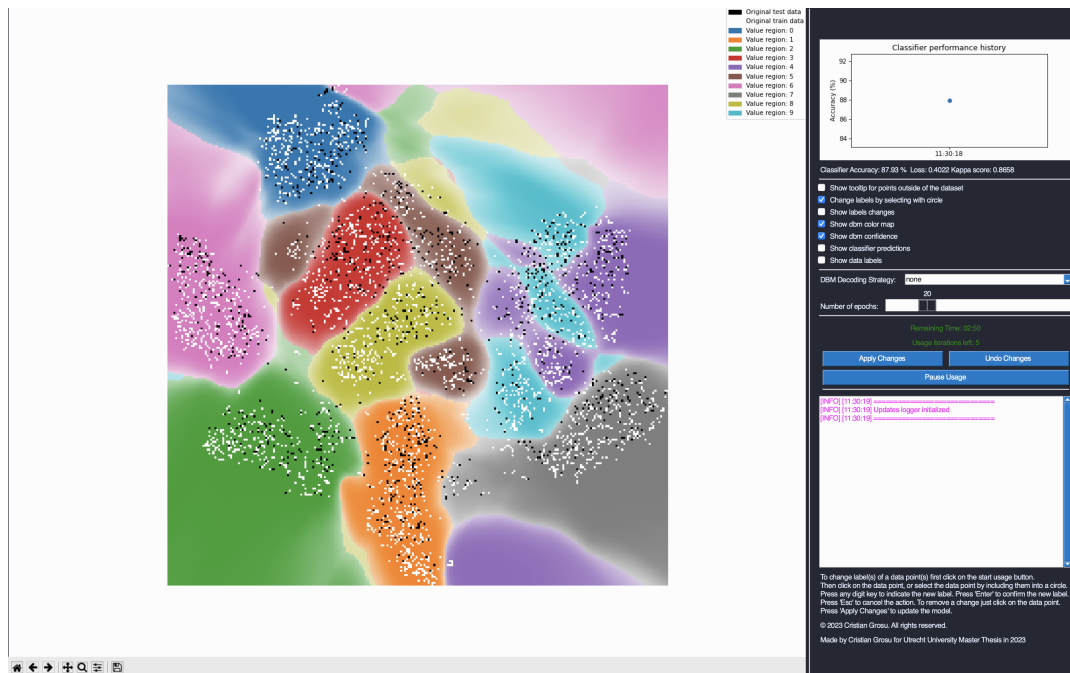


FIGURE 3.33: DBM visualization window

**Step 4: Generating the Decision Boundary Mapping** After the user successfully uploads the data set of interest along with the classifier he/she wants to improve upon and chooses the DBM technique, the Decision Boundary Map can be generated. If the combination of data set, classifier, and DBM technique is used for the first time, the computation of the DBM can take some time because the visualization tool is learning the direct and inverse projections first. On the second run with the same configuration the computation of the DBM will be faster because the tool stores the projections in a temporary folder, that the user can further use. After the DBM is successfully computed the DBM image will be displayed in the configuration window as shown in figure A.2 (surrounded by a green rectangle). The user can click on that image, in this case, a new window will pop up as the one shown in figure 3.33.

### 3.5.2 Data set visualization and re-labeling window

In the previous section, we presented how the user can upload the data set and the initial classifier in order to generate the DBM. In the last step, we showed how by clicking on the image generated in the configuration window a DBM visualization window is created (see figure 3.33). In this section, we present the components of this window as well as a short usage guide regarding the functionalities it provides.

In the left part of the window, the user can see a plot with the 2D projection of the data set, the points from the training set are presented as white pixels and the data points from the testing set are colored in black. All of the other pixels are initially colored with respect to the label the classifier assigns to the  $nD$  representation of that pixel (given by the inverse projection). The opacity (i.e. alpha channel) of each pixel encodes the classifier's confidence regarding the prediction for that pixel. The smaller the opacity the smaller the confidence.

When a pixel is hovered a tooltip image with the  $nD$  representation of the pixel is shown. If the pixel is a 2D representation of a data point from the data set then

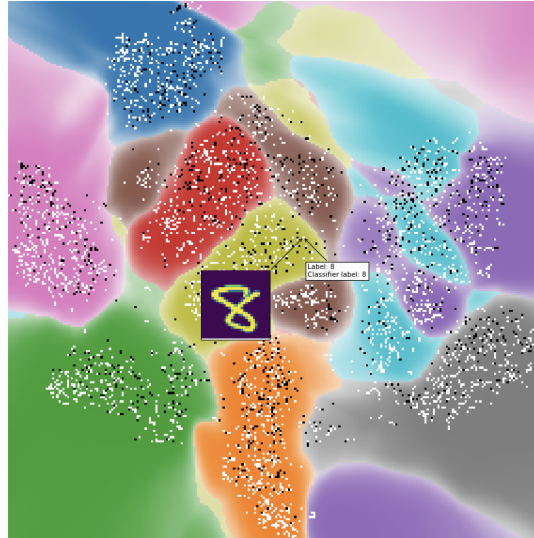


FIGURE 3.34: DBM visualization window, tooltip image, and information about the data point on hover

additional information such as the assigned label is shown as presented in figure 3.34.

The classifier's accuracy is plotted in the right top corner. Below this plot the tool shows the accuracy value and the loss of the current classifier.

Initially, the projection errors and the inverse projection errors are not computed, the user can compute them by clicking on the respective buttons. For the projection errors, the user can choose one of two options, namely *Compute Projection Errors (interpolation)* (which will compute the projection errors of the points in the data set and interpolate on the rest of the points using algorithm 2) or *Compute Projection Errors (inverse projection)* (which will compute the projection errors by the use of the inverse projection following the algorithm 3). The direct projection and inverse projection errors are encoded in the opacity of the pixel, the greater the error the less the opacity of that pixel. Regions with low opacity mean regions with high error values.

On the right side of the visualization window, below the accuracy plot, there is a list of check-boxes that the user can use to visualize such things as confidence, direct and inverse projection errors, and classifier predictions (see figure A.5 for examples). The user can mix these check-boxes to get more insights (e.g. figure A.6). Since the confidence values, along with the direct and inverse projection errors values are encoded in the opacity of the pixel we are computing the opacity of a pixel  $p$  as  $c \cdot (1 - \epsilon_p) \cdot (1 - \epsilon_{p-1})$ , (where  $c$  is the confidence value, and  $\epsilon_p, \epsilon_{p-1}$  are the direct and inverse projection error for  $p$  respectively). The more confident the classifier is in the label and the less the projection and inverse projection errors are, the bigger the pixel's opacity.

### Re-labelling and re-training classifier

In order to retrain the classifier the user needs to perform a series of actions as follows (see figure 3.31, re-labeling and re-training steps):

- Step 1: Re-label the points of interest
- Step 2 (Optional): Change the DBM generation strategy

- Step 3: Apply the changes
- Step 4: Analyze the results and return to Step 1 if necessary or undo the changes

**Step 1: Relabel the points of interest** The user has the option to select a chunk of data points by clicking with the mouse and releasing to draw a circle as shown in figure 3.35a. If the user wants to undo the drawing, this can be done by pressing the *Esc* key from the keyboard. After the circle is drawn, the user can press on the keyboard any digit key to assign a new label to all the points within the circle. By pressing enter the new labels are fixed, all the data points within the circle are marked with triangles and the circle is removed (see figure 3.35b). This process can be repeated until the user is satisfied with all the changes.

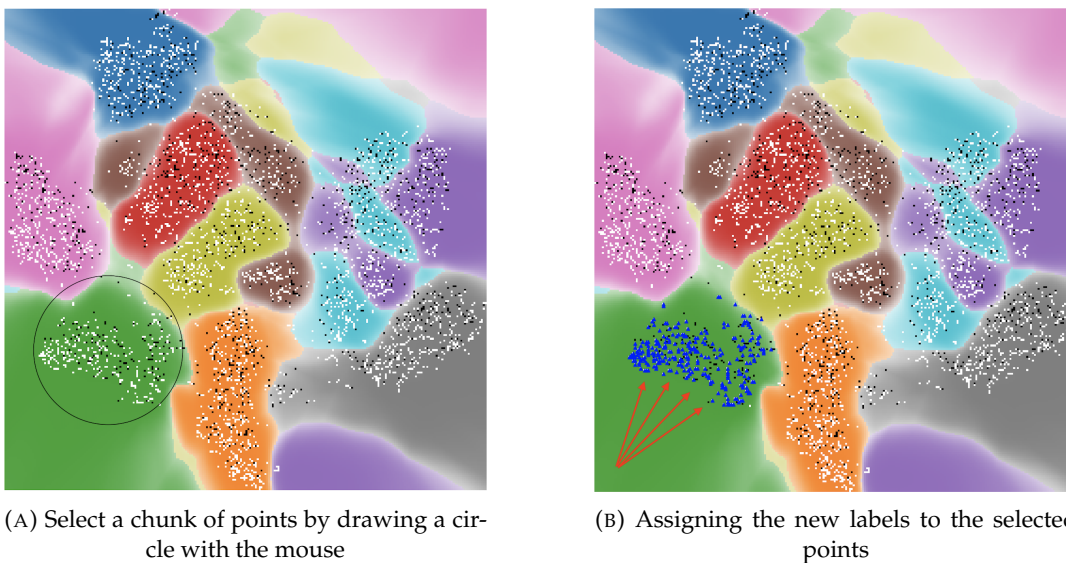


FIGURE 3.35: Re-labeling the data points using the visualization tool

**Step 2: Choose the DBM generation strategy** Once the user has changed the labels of the data points he/she thinks will improve the classifier performance, one of 4 algorithms can be used to re-compute the DBM once the classifier is re-trained (see figure 3.36). The options are: *none* (algorithm 1), *binary\_split* (algorithm 4), *confidence\_split* (algorithm 8) and *confidence\_interpolation* (algorithm 9).

**Step 3: Apply the changes** The classifier can be re-trained to consider the new re-assigned labels by clicking on the *Apply changes* button. Once the user clicks on this button a copy of the current classifier is stored as well as all the changes in labels made by the user, after which the classifier is re-trained from scratch for a number of epochs, and the DBM is re-computed using the strategy indicated in the previous step. The Logger component shows the progress of the process.

**Step 4: Analyze the results** Once the new DBM with the corresponding changes from step 1 was computed the user can analyze how the classifier performs. If the 2D plot showing the DBM does not adhere to what the user thinks about the data set or the user is not satisfied with the new classifier's performance, he/she can undo the last changes by clicking on the *Undo changes* button. In this case, the previous

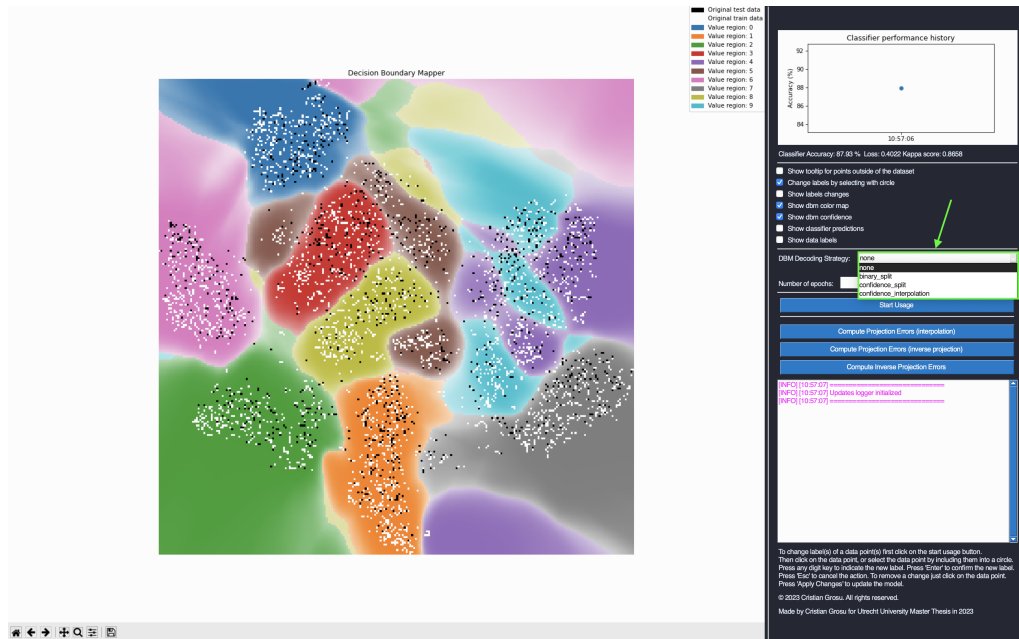


FIGURE 3.36: Choosing the DBM generation algorithm in the visualization tool

classifier will be restored and the DBM will be generated using that classifier. In case the user is satisfied with the results, he/she can repeat the process from steps 1 to 4 to further improve the classifier, or stop the visualization tool. The classifier from the last iteration is stored in a folder from which the user can retrieve and further use it in other pipelines.

### 3.6 Conclusions

In this chapter, we presented the implementation details of our visualization tool. We started with section 3.1 by describing how we can find the inverse projection when a direct projection is given, and how to learn both the direct and inverse projection based on the data set. In the same section, we presented an algorithm that can be used to compute the DBM by means of the inverse projection. Section 3.1 gave the implementation details necessary for the visualization tool to fulfill the requirements: **R1** and **R2**. In section 3.2 we discussed how we can solve the requirements **R4** and **R5**. We showed metrics for the errors of the direct and inverse projections that are based on the neighbors in the 2D and the  $n$ D space. Section 3.3 described how the visualization tool can fulfill requirements **R3.1** and **R3.2**. After re-labeling data points and re-training the classifier we need to re-compute the DBM each time. In section 3.4 we presented three new heuristics that aim to speed up the process of computing the DBM. The heuristics introduced in this section are compared against each other on only one data set and one projection method. We could not identify which heuristic is the best in the general case. In our next chapter, we present a series of broad experiments in order to better compare and understand which heuristic is the best in which context. In other words, we want to answer the following question: **Q1** "How fast and accurate are our DBM computation heuristics and can they be used as alternatives to the Dummy DBM algorithm?". Section 3.5 presented an overview of the visualization tool we have built. Moreover, a step-by-step usage guide was presented in the same section.

The proposed solution design fulfills all the problem requirements and more. However, we are interested in how well these requirements are fulfilled and more important if the proposed product is fulfilling the core goal of helping a user improve the performance of a classifier. Therefore, in our next chapter, we answer the following question: **Q2** "How useful is the visualization tool in improving a classifier in the context of semi-labeled data sets?".





## Chapter 4

# Experiments And Results

The main goal of this chapter is to answer two main questions defined in the previous chapter:

- **Q1** How fast and accurate are our DBM computation heuristics and can they be used as an alternative to the Dummy DBM algorithm?
- **Q2** How useful is our visualization tool in improving a classifier in the context of semi-labeled data sets?

In order to answer these questions in this chapter we present a series of experiments and discussions based on the results of these experiments. Details about the environmental setup for our experiments can be found in tables [B.1](#) and [B.2](#).

Section [4.1](#) presents the data sets used for our experiments. Sections [4.2](#) and [4.3](#) have two subsections each, namely: Methodology and Experiments, Results and Discussions. In the methodology and experiments section, we describe the list of experiments we perform as well as the methodology we follow where we describe a list of criteria in order to make the experiments reproducible.

In section [4.2](#) we are performing a series of experiments in which we compare several configurations of our heuristic methods for DBM computation. The scope of these experiments is to give us insights into which method is the best in terms of speed and accuracy and answer question **Q1**.

In section [4.3](#) we aim to see both quantitatively and qualitatively how useful our visualization tool is in aiding users to improve classifiers' performance. This section aims to answer the question **Q2**.

The chapter ends with section [4.4](#) in which we provide conclusions based on our experimental results about the value our visualization tool adds to the process of training a classifier in the context of semi-labeled data sets. This section reiterates answers to questions **Q1** and **Q2**.

## 4.1 Data sets used in the experiments

In our experiments, we are using two data sets, namely MNIST [[10](#)] and Protozoan cysts [[19](#)]. In this section, we present a short overview of the properties of these data sets.

The MNIST dataset consists of a collection of 28x28 pixel (784 pixels in total) grayscale images (each pixel value ranges from 0 to 255) of handwritten digits (0 through 9), along with their corresponding labels, indicating which digit each image represents. This data set contains 70,000 images of which 60,000 are usually used for training and 10,000 for testing. In our experiments, we are using a sub-sample of 5000 images from which 70% are used for training (i.e. 3,500 samples) and 30% are used for testing (i.e. 1,500 samples). We are taking the first 3,500 data points from

the 60,000 provided in the training set and the first 1,500 from the 10,000 provided in the testing set. Both the training and testing subsets contain a balanced amount of samples from each class. We use a limited amount of samples of the MNIST data set (i.e. 5000 samples) due to computational efficiency reasons (i.e. faster pseudo-labeling and faster run times when re-training a classifier).

The Protozoan cysts [19] data set consists of a collection of 200x200 pixel (40,000 pixels in total) color images (each pixel is represented by a tuple of 3 values, each value in the range from 0 to 255) representing microscopical images of human intestinal parasites. The data set consists of 3,852 images from which 2,696 are the training set and 1,156 are the testing set. The classes in this data set are as follows:

0. E.coli (train: 590 images, test: 216 images)
1. E.histolytica (train: 12 images, test: 23 images)
2. E.nana (train: 440 images, test: 217 images)
3. Giardia (train: 486 images, test: 192 images)
4. I.butshlii (train: 1074 images, test: 451 images)
5. B.hominis (train: 94 images, test: 57 images)

The split into training and testing subsets for the Protozoan cysts data set was done at random, and the indexes of the data samples that are in each of these subsets are stored in a git repository <sup>1</sup>.

Table 4.1 presents an overview of the properties of the data sets we are using.

Data set	Number of classes	Data point shape	Train set samples	Test set samples
MNIST	10	28x28x1	3500	1500
Protozoan cysts	6	200x200x3	2696	1156

TABLE 4.1: Overview of the data sets used in the experiments

## 4.2 DBM optimization heuristic algorithms

In this section, we want to answer the question **Q1** of how scalable is our visualization tool. For this purpose, we analyze the following criteria:

- What is the gain in speed when using the proposed heuristics as opposed to when using the Dummy DBM algorithm?
- How accurate are our heuristics compared to the Dummy DBM algorithm in terms of mislabelling and errors in confidence values?

Moreover, in this section, we are going to determine what hyper-parameters are the best for our heuristics.

This section is structured as follows, in section 4.2.1 we present the series of experiments we perform in order to test the above-described criteria along with details about the experimental setup. In the following section 4.2.2, we present the results of our experiments and perform a critical analysis of our heuristics. Section 4.2.3 lists our conclusions about how good in terms of scalability each heuristic is.

<sup>1</sup><https://github.com/cristi2019255/ParasitesDatasetIndexes>

### 4.2.1 Experiments and Methodology

In this section, we describe the experiments we perform to test the speed and accuracy of each DBM image generation heuristic, along with the methodology we follow.

In order to test the accuracy of our heuristic algorithms we use the same metrics  $\epsilon_{labels}$  and  $NRMSE_{confidence}$  introduced in section 3.4.1 by equations 3.11 and 3.12.

For our experiments, we use the MNIST data set as described in section 4.1 (a). In our experiments we are going to generate  $n \times n$  size images, where  $n \in \{100, 150, 200, \dots, 1950, 2000\}$ . These values for  $n$  were chosen from practical considerations because in practice the probability that the user would like to generate DBM images with a resolution less than  $100 \times 100$  or greater than  $2000 \times 2000$  is very low. The DBM images we generate in our experiments are representations of a classifier  $\mathcal{C}$  decision boundaries. The architecture of the classifier  $\mathcal{C}$  is described in figure 4.1. For our experiments in this section, we train  $\mathcal{C}$  on the completely labeled MNIST data set as shown in figure 4.2.

Our heuristics have two hyper-parameters:

- $B$  - The initial number of blocks
- *confidence\_interpolation* - The confidence interpolation method

In our first experiment, we want to determine what is the best value for the parameter  $B$ . The higher the value of this parameter the lower the amount of mis-labeled pixels  $\epsilon_{labels}$ . On the other hand, the higher the value of  $B$  the higher the chance that the heuristic algorithms will halt slower. Consider  $B_1 < B_2$ , if we start our heuristic algorithms with  $B_1$  amount of blocks instead of  $B_2$  initially we will have to spend less computational power to get all the probability vectors of the central pixels of the blocks. However, it might happen that during the process we have to split a lot of blocks and for each new block we have to use the classifier in order to get the label values of the central pixels. Whereas if we start with a greater amount of initial blocks  $B_2$  initially we have to perform more computation to decode the central pixels of these blocks, but depending on how the classifier boundaries are geometrically placed in 2D we might need fewer block splits in the long term. Thus, the run time taken by the same heuristic for  $B_1$  might be bigger than the run time taken for  $B_2$ . We want to find for which values of  $B$  the run times and  $\epsilon_{labels}$  are minimal. Due to the above-described values of  $n$  and to the constraint  $1 < B < n$  the values of  $B$  we are going to test are  $\{8, 16, 24, \dots, 88, 96\}$ .

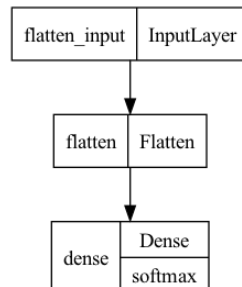


FIGURE 4.1: Classifier  $\mathcal{C}$  architecture used in the experiments

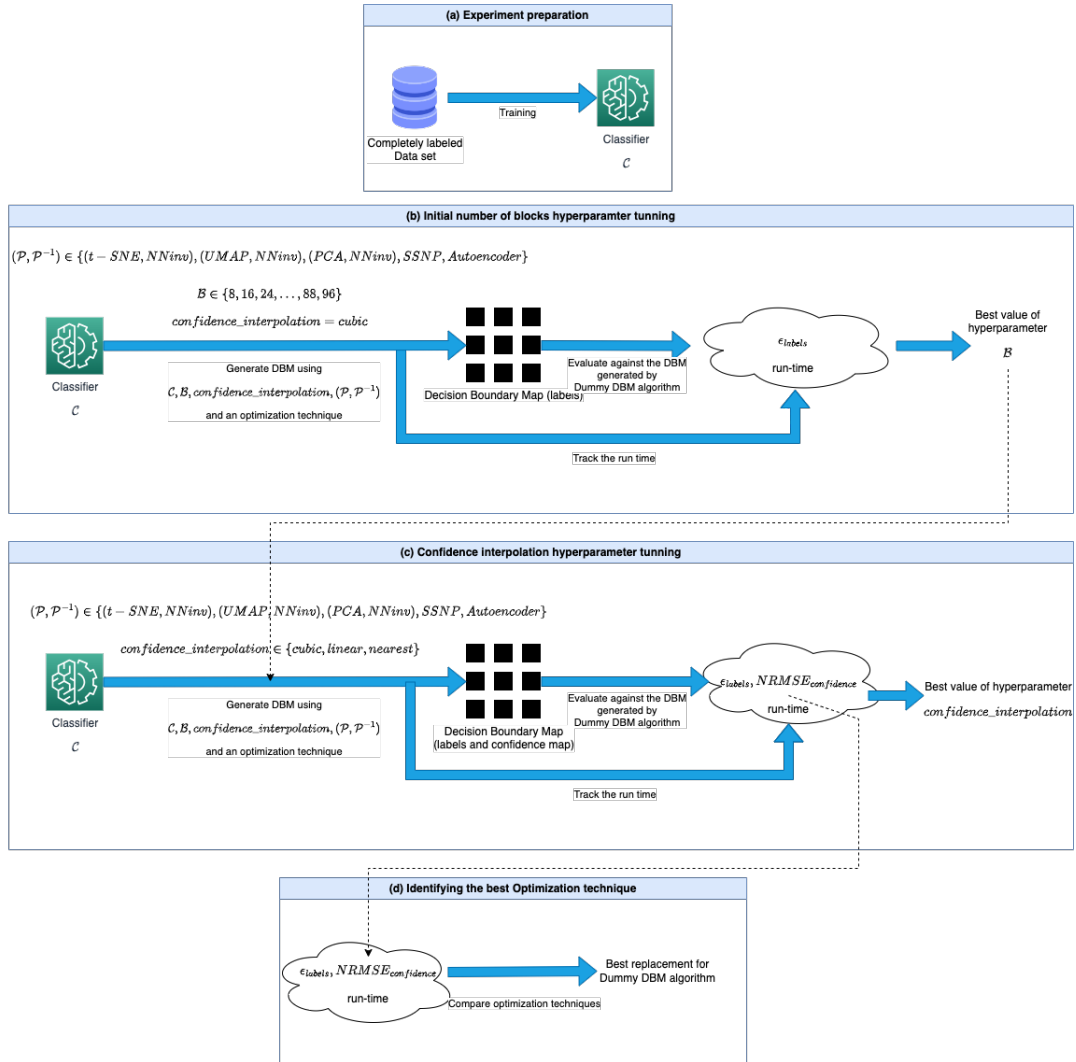


FIGURE 4.2: Experimental pipeline for DBM optimization heuristic algorithms, (a) Experiment preparation, (b) Initial number of blocks  $B$  hyperparameter tuning, (c) *confidence\_interpolation* hyperparameter tuning, (d) Results analysis

In order to identify which values of  $B$  are the best we perform the following experiment. For all the projection methods mentioned so far, we generate a  $2000 \times 2000$  DBM image using Binary Split and Confidence Interpolation heuristics for different values of  $B$ , we track the run times and compute the  $\epsilon_{labels}$  as shown in figure 4.2 (b). Based on this experiment, we infer for which values of  $B$  our heuristics achieve a trade-off between run times and the amount of label errors. In this experiment, we fix the *confidence\_interpolation* parameter to "cubic" and generate the DBM for several projection and inverse projection methods as described in figure 4.2 (b).

For our next series of experiments, we generate DBM images using all the described heuristics for each projection method and for each value of  $n \in \{100, 150, \dots, 1950, 2000\}$ . For each heuristic, we use the best value of  $B$  found in the previous experiment. We track the run times, the number of label errors  $\epsilon_{labels}$ , and the  $NRMSE_{confidence}$  metric as shown in figure 4.2 (c). In this experiment, we want to test which heuristic is the best in terms of run times and accuracy and find what is the best value of the hyperparameter *confidence\_interpolation*  $\in \{nearest, linear, cubic\}$ . In the Binary Split

and Confidence Split algorithms, this hyper-parameter is used to generate the confidence map at the end of the algorithm, which means the only things that would differ in the results of these algorithms for different values of  $confidence\_interpolation$  would be the results of  $NRMSE_{confidence}$  metric and negligible differences in the run-times. On the other hand, for the Confidence Interpolation algorithm, this parameter will directly influence the run time and the values of  $\epsilon_{labels}$  and  $NRMSE_{confidence}$  metrics.

Our end goal is to identify which heuristic is the best from the presented ones and identify if is possible and under which conditions we can use these optimization techniques as replacements for the Dummy DBM algorithm in practice. Therefore we compare the run times and performance of each heuristic when used together with the best identified values of hyper-parameters as shown in figure 4.2 (d).

## 4.2.2 Results and Discussions

### Initial number of blocks $B$ hyperparameter tuning

Table 4.2 presents the results of the  $B$  hyper-parameter tuning experiments. Observe how the  $\epsilon_{labels}$  metric decreases drastically for values of  $B$  under 32, regardless of the heuristic or the projection method. Notice that when the Binary Split heuristic is used in combination with the projections such as Autoencoder or SSNP for  $B = 16$ ,  $\epsilon_{labels}$  is greater when compared to  $B = 8$ . This behavior might be caused by the fact that for these values of  $B$  the central pixel labels of certain neighboring blocks are the same and the algorithm does not split these blocks into smaller blocks, however, these blocks contain a portion of one or more decision boundaries in them, thus the algorithm is making mistakes. However, observe that the overall trend is that the bigger the value of  $B$  the less the number of errors, the decrease being considerable for  $B < 32$ . Moreover, the overall trend is not dependent on the projection or the heuristic method in use.

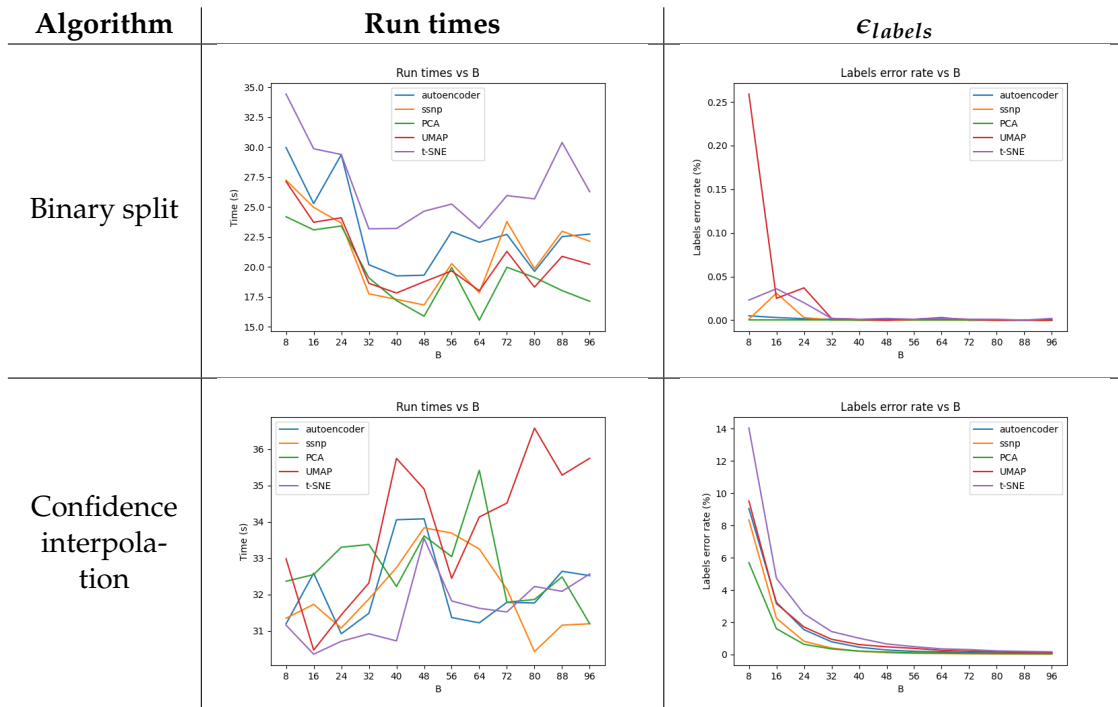


TABLE 4.2: Initial number of blocks  $B$  hyper-parameter tuning.  
Data set: MNIST,  $confidence\_interpolation = cubic$

In terms of run times, there is no clear winner, in the sense that there is no value of  $B$  for which each heuristic in combination with each projection method achieves optimal run times. For instance, when using the Autoencoder projection the best run times are achieved when  $B \in \{40, 48\}$  for the Binary Split and  $B = 24$  for Confidence Interpolation. When using a different projection method different values are optimal, for instance, when PCA is used in combination with Binary Split best value is  $B = 64$ , and when in combination with the Confidence Interpolation best value is  $B = 40$ . Notice however that if we choose  $B = 32$  we can achieve run-times near optimal for all the heuristic and projection methods combinations.

Given the previous argumentation about the  $\epsilon_{labels}$  trend for values of  $B < 32$ , we can conclude that  $B = 32$  is the best value from the ones tested in our experiments. Therefore, in our next experiments as the methodology describes we are using  $B = 32$ . We are not claiming that this value is going to bring optimal results in terms of performance for all the values of  $n$  and for all projection methods. However, this value will bring results near to optimal, and by choosing this value we are tending to achieve a balance in terms of speed and accuracy.

### Confidence interpolation hyperparameter tuning

Tables 4.3, 4.4, 4.5, 4.6 and 4.7 present the run times and the accuracy metrics for our heuristics when used with different values for the *confidence\_interpolation* hyperparameter. Observe that regardless of the projection method when the Binary and Confidence Split algorithms are used in combination with linear interpolation for the confidence maps (i.e. *confidence\_interpolation = linear*) we obtained the smallest errors in the  $NRMSE_{confidence}$  metric. When the cubic interpolation method is used in these algorithms we have the worst approximation of the confidence image. This is caused by the differences in how the cubic and linear interpolation approximate a signal. Consider the situation where our algorithm has generated two neighboring blocks (from left to right)  $b_1, b_2$  where the central pixels of these blocks have confidence values near 1. Suppose that  $b_1$  has a block on the left  $b_0$  and  $b_2$  has a neighboring block at the right  $b_3$ , if the central pixels of the blocks  $b_0, b_3$  have values near 0, then by using the cubic interpolation all the pixels between the central pixels of  $b_1$  and  $b_2$  will be assigned confidence values bigger than 1. This is not the case if we use linear or nearest interpolation. Since the ground truth confidence values are in the range  $[0, 1]$  the cubic interpolation approximation shows worse results if we have such blocks as described previously since the term  $(f_i - \hat{f}_i)^2$  in the  $NRMSE_{confidence}$  metric would be higher as opposed to when using linear interpolation. Therefore, for the Binary and Confidence Split algorithms, the best value of the *confidence\_interpolation* hyper-parameter is linear. On the other hand, when using the Confidence Interpolation algorithm, the best results are achieved for the cubic interpolation, followed by linear interpolation and then the nearest neighbor interpolation. This is caused by the fact that the cubic interpolation approximates the probability functions  $f_1, \dots, f_d$  the best. Recall that the confidence interpolation algorithm approximates a function for each class and then assigns the label and the confidence value of each pixel by looking at all the probability functions together. This difference in how this heuristic assigns the confidence values is why the cubic interpolation is better than linear interpolation in this case when opposed to Binary or Confidence Split where the confidence image is generated by approximating a signal based on only one value per central pixels. In other words, the Confidence Interpolation approximates  $d$  signals (where  $d$  is the number of classes of the data set)

and then constructs the confidence signal by taking the max value of these approximated signals for each pixel, whereas the other heuristics first get the max confidence value for each pixel and then using these values approximate the confidence image.

Run-time-wise, the Confidence interpolation algorithm is the fastest when used with the value of the *confidence\_interpolation* hyper-parameter set to linear. The worst run times for this heuristic are achieved for the nearest neighbor interpolation method. This is caused due to the fact that in order to use the nearest interpolation method the values of the central pixels of the blocks are not enough. With this type of interpolation, the algorithm needs to generate some additional blocks outside of the DBM image such that the confidence values for the pixels at the edges of the image can be approximated as well. This is causing the number of initial blocks to be bigger and therefore the longer run times. The fact that when used with cubic interpolation the algorithm takes slightly longer when compared to linear interpolation is explained by the fact that the cubic approximation requires a slightly more number of operations.

Algorithm	Run times	$\epsilon_{\text{labels}}$	$\text{NRMSE}_{\text{confidence}}$
Binary split (algorithm 4)			
Confidence split (algorithm 8)			
Confidence interpolation (algorithm 9)			

TABLE 4.3: *confidence\_interpolation* hyper-parameter tuning.  
Data set: MNIST,  $B = 32$ ,  $\mathcal{P}$ : t-SNE,  $\mathcal{P}^{-1}$ : NNinv

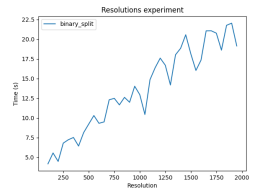
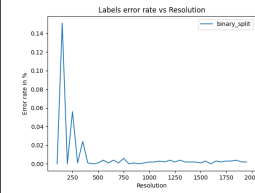
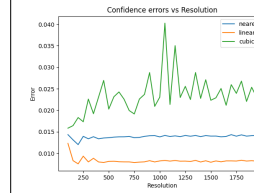
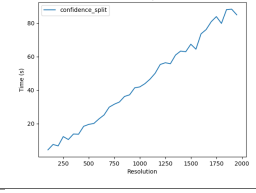
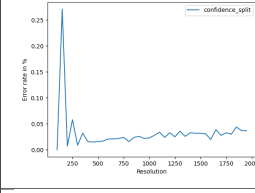
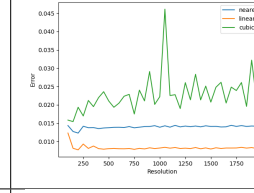
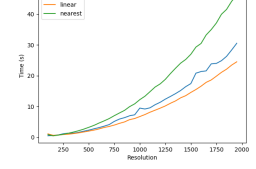
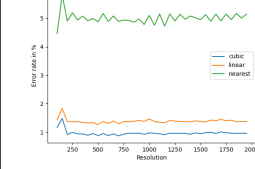
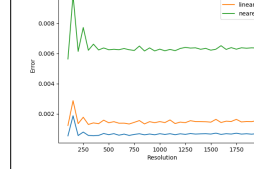
Algorithm	Run times	$\epsilon_{\text{labels}}$	$\text{NRMSE}_{\text{confidence}}$
Binary split (algorithm 4)			
Confidence split (algorithm 8)			
Confidence interpolation (algorithm 9)			

TABLE 4.4: *confidence\_interpolation* hyper-parameter tuning.  
Data set: MNIST,  $B = 32$ ,  $\mathcal{P}$ : UMAP,  $\mathcal{P}^{-1}$ : NNinv

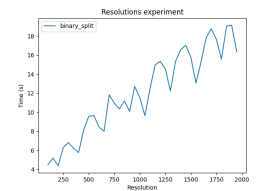
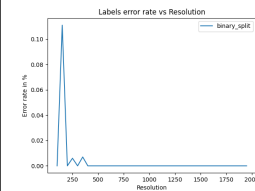
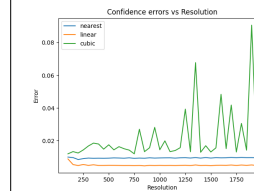
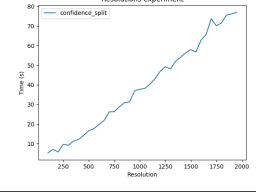
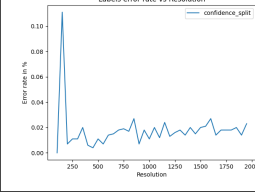
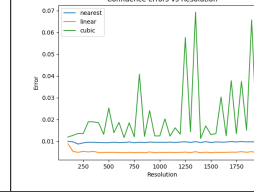
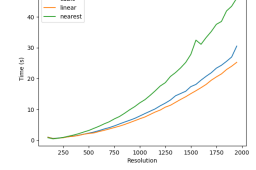
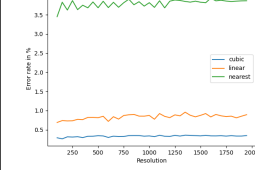
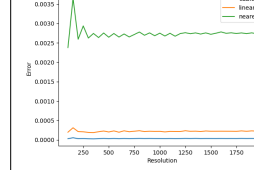
Algorithm	Run times	$\epsilon_{\text{labels}}$	$\text{NRMSE}_{\text{confidence}}$
Binary split (algorithm 4)			
Confidence split (algorithm 8)			
Confidence interpolation (algorithm 9)			

TABLE 4.5: *confidence\_interpolation* hyper-parameter tuning  
Data set: MNIST,  $B = 32$ ,  $\mathcal{P}$ : PCA,  $\mathcal{P}^{-1}$ : NNinv



Algorithm	Run times	$\epsilon_{\text{labels}}$	$\text{NRMSE}_{\text{confidence}}$
Binary split (algorithm 4)			
Confidence split (algorithm 8)			
Confidence interpolation (algorithm 9)			

TABLE 4.6: *confidence\_interpolation* hyper-parameter tuning.  
Data set: MNIST,  $B = 32$ ,  $\mathcal{P}, \mathcal{P}^{-1}$ : Autoencoder

Algorithm	Run times	$\epsilon_{\text{labels}}$	$\text{NRMSE}_{\text{confidence}}$
Binary split (algorithm 4)			
Confidence split (algorithm 8)			
Confidence interpolation (algorithm 9)			

TABLE 4.7: *confidence\_interpolation* hyper-parameter tuning.  
Data set: MNIST,  $B = 32$ ,  $\mathcal{P}, \mathcal{P}^{-1}$ : SSNP

### Heuristics performance analysis and comparison

Table 4.8 presents the performance of our heuristics when used with the best-identified hyper-parameters:  $B = 32$ , *confidence\_interpolation* = *linear* for Binary and Confidence Split;  $B = 32$ , *confidence\_interpolation* = *cubic* for Confidence Interpolation.

Observe that in terms of run-time, the Binary Split, and Confidence Interpolation are performing almost the same. However, the Binary Split shows a quasilinear time

complexity in terms of  $n$ , whereas the Confidence Interpolation algorithm shows a quadratic time complexity trend. The Confidence-based Split algorithm is showing worse results in terms of run times when compared to both Binary Split and Confidence Interpolation. However, this algorithm also shows a quasilinear time complexity trend. Therefore, for big enough values of  $n$ , the Confidence-based Split algorithm might outperform the Confidence Interpolation algorithm. Notice that for  $n > 1800$  the Binary split algorithm is outperforming the Confidence Interpolation heuristic. All the algorithms outperform the Dummy DBM algorithm run times for  $n$  starting at 650. Confidence Interpolation is faster than the Dummy DBM algorithm for any value of  $n$ . The Binary Split algorithm starts to outperform the Dummy DBM algorithm for  $n$  bigger than 400. Recall that with the Dummy DBM algorithm, we are using the classifier  $n^2$  times to compute the DBM pixel-wise. In our heuristics we are using the classifier less than  $n^2$  times, however, we are doing additional computations, such as updating a priority queue for instance. Moreover, to make the classifier predictions we are using the *Keras* library, which is very optimized for batch processing. This is why our algorithms really start to shine only at big enough values of  $n$  (i.e.  $n > 650$ ). The higher the value of  $n$  the bigger the gains in terms of speed we are getting. Notice that this trend is occurring regardless of the projection technique in use.

In terms of accuracy if we are analyzing the  $\epsilon_{labels}$  column from table 4.8 we observe that the Confidence Interpolation algorithm is returning the worst results when compared to the other algorithms. Notice however that the error rate is not bigger than 2% for any projection technique and any heuristic. Observe that the highest error rates for the Confidence Interpolation algorithm are for such projection techniques as UMAP and t-SNE. This is caused by the fact that these projection methods create more small clusters of similar points when compared to SSNP for instance, which tends to create one cluster per class. Since in the Confidence Interpolation algorithm we are looking at the 2D coordinates and interpolating the classifier confidence functions, our approximation of the confidence functions depends on how well the projection method preserves the global structure of our data rather than the local structure. In other words, if we start with the same amount of the initial blocks, the worse the global structure is preserved by the projection technique the worse the approximations for the confidence functions would be, due to a more complex signal to approximate, leading to more errors in the generated labels.

Observe that both Binary and Confidence Split are performing better in the sense of fewer label errors when compared to the Confidence Interpolation algorithm. This is caused by the fact that these algorithms are locally based (i.e. each block is being split based on the neighboring blocks), whereas in the Confidence Interpolation algorithm, we are generating the DBM based on a global (per block) approximation of the confidence functions. In other words, in the Confidence Interpolation algorithm, a finite set of pixels at certain geometrical positions determines the labels. The relative error rate for Binary and Confidence Split algorithms is no bigger than 0.3%.

If we analyze the confidence maps generated by the algorithms, all of them show values for the  $NRMSE_{confidence}$  less than 0.015, which means that the confidence maps are quite good approximations of the ground truth ones. The Confidence Interpolation algorithm is showing better results than the other algorithms with the  $NRMSE_{confidence}$  not bigger than 0.002. The Binary and Confidence Split algorithms are showing almost the same performance when compared to each other. This similarity might be caused by the fact that even though the Binary and Confidence Split

might converge to different partitions of the image into blocks, the confidence values of the central pixels of these blocks lead to almost the same approximation of the confidence map.

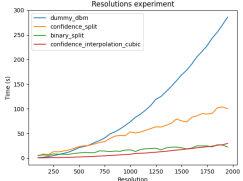
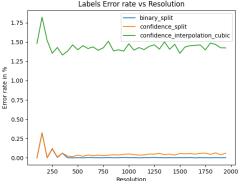
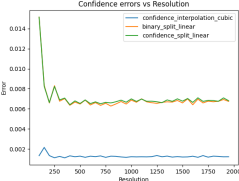
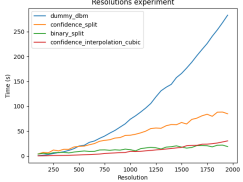
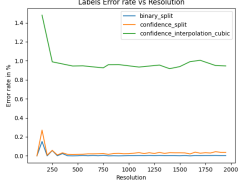
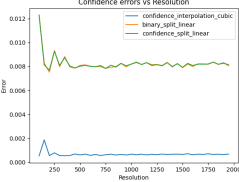
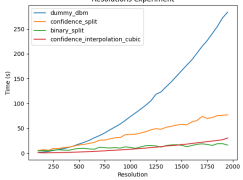
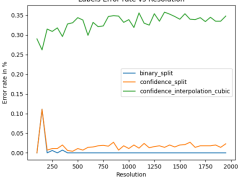
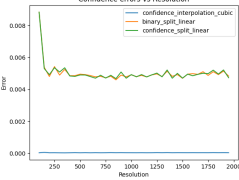
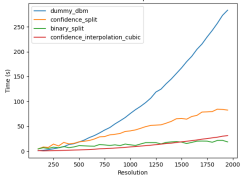
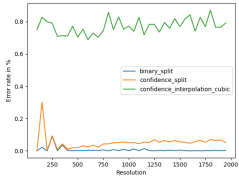
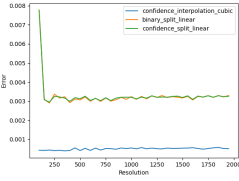
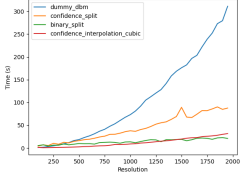

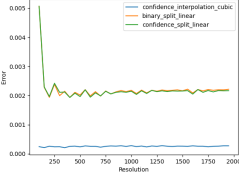
Projection and inverse projection method	Run times	$\epsilon_{\text{labels}}$	$\text{NRMSE}_{\text{confidence}}$
t-SNE, NNinv			
UMAP, NNinv			
PCA, NNinv			
Autoencoder			
SSNP			

TABLE 4.8: DBM algorithms comparison.

Data set: MNIST.

Hyperparameters:  $B = 32$ ,  $\text{confidence\_interpolation} = \text{cubic}$  for Confidence Interpolation algorithm,  $\text{confidence\_interpolation} = \text{linear}$  for Binary and Confidence Split algorithms

The discrepancy in the results given by Confidence Interpolation and the other algorithms is again explained by the different ways of generating the confidence map. In the Confidence Interpolation case we are computing all the confidence values for each pixel (by approximating all the class confidence functions of the classifier), we construct the confidence map by assigning the max confidence value for each pixel. Whereas in the Binary and Confidence Split, we save only the max confidence value for each central pixel of the blocks and then we are interpolating based on these values in order to obtain the confidence map. Thus in this case we have

a more poor approximation technique. Therefore the  $NRMSE_{confidence}$  is slightly higher. A way to achieve smaller error rates for the Split heuristics is to use the same approach as in the Confidence Interpolation algorithm and store all the probability vectors for block representative pixels, interpolate the confidence functions instead of the confidence map directly, and construct the confidence map based on these functions. However, this approach would increase the complexity of these algorithms. If we were to use such an approach we would trade from the speed of the algorithm(s) in order to gain a slightly better confidence map. In practice, the user will not observe the differences in the confidence map but will observe differences in speed. Therefore, speed is a more important factor than a small increase in the accuracy of the confidence map.

### 4.2.3 Conclusions

**Answer to question Q1:** The proposed heuristics show a considerable gain in speed for generating highly accurate DBM images. The Binary Split method is a simple and comprehensible heuristic that can serve as the best alternative for the Dummy DBM algorithm in practice for the fast generation of accurate DBM images.

In this section, we experimentally analyzed how good the proposed DBM generation heuristics are in terms of performance. A set of hyperparameter tuning experiments was carried out in order to reveal the optimal hyperparameter values for our heuristics. We found that Binary and Confidence Split heuristics yield the most accurate approximations of the confidence map when used together with the linear interpolation. The Confidence Interpolation algorithm gives the best results when the cubic interpolation is used. We presented both theoretical arguments and experimental results that support the choice of  $B = 32$  as the initial number of blocks hyper-parameter.

In terms of speed, we proved that all of our algorithms outperform the Dummy DBM algorithm. The bigger the resolution of the DBM image we want to generate the bigger the difference in run-times between our heuristics and the Dummy DBM algorithm. For instance, when the t-SNE projection method is used to generate a  $2000 \times 2000$  DBM image the Dummy DBM algorithm takes 285 seconds, the Confidence-based Split takes 100 seconds, the Confidence Interpolation algorithm takes 30 seconds and the Binary Split only 22 seconds. This trend is persistent regardless of the projection method in use.

All our heuristics were able to generate quite accurate DBM with at most 1.8% of mislabelled pixels and  $NRMSE_{confidence}$  no greater than 0.0015.

Based on the results presented in this section we can conclude that the Binary Split heuristic is the most scalable algorithm when compared to the other 2 options. This algorithm should be used in practice as an alternative to the Dummy DBM for fast DBM image generation. Not only does this algorithm yield DBM images that show the lowest error rates in the number of labels and very good approximations for the confidence map, but also it shows a quasilinear run time complexity. Moreover, further optimizations such as splitting blocks in parallel are possible and can be incorporated into the proposed algorithm to achieve even more gains in speed.

## 4.3 Visualization tool evaluation

This section aims to answer question Q2. Based on this question we define the goals of our experiments for this section as follows:

- **Goal 1** Quantitatively assess the effectiveness of our visualization tool in enhancing classifier performance in comparison to relying solely on an automatic labeling algorithm. Additionally, identify the most beneficial projection methods within the context of using the tool.
- **Goal 2** Evaluate qualitatively the tool’s impact on classifier performance in scenarios where users have varying levels of knowledge about the data, and assess the utility of direct and inverse projection error visualizations in data points labeling and enhancement of classifier performance.

This section is structured as follows: section 4.3.1 outlines the methodology and the experiment structure adopted in order to test the above-presented goals; section 4.3.2 delves into discussions surrounding the experimental results.

Due to the fact that in this section our main focus is to find out how useful the tool is, in all the experiments described further we chose to configure the tool such that it generates DBM images of size  $256 \times 256$  using the Dummy DBM algorithm. For such a resolution of the DBM images all the optimization heuristics are performing almost the same, the only difference is that the Dummy DBM algorithm is generating DBM images without errors in labels and confidence map.

### 4.3.1 Experiments and Methodology

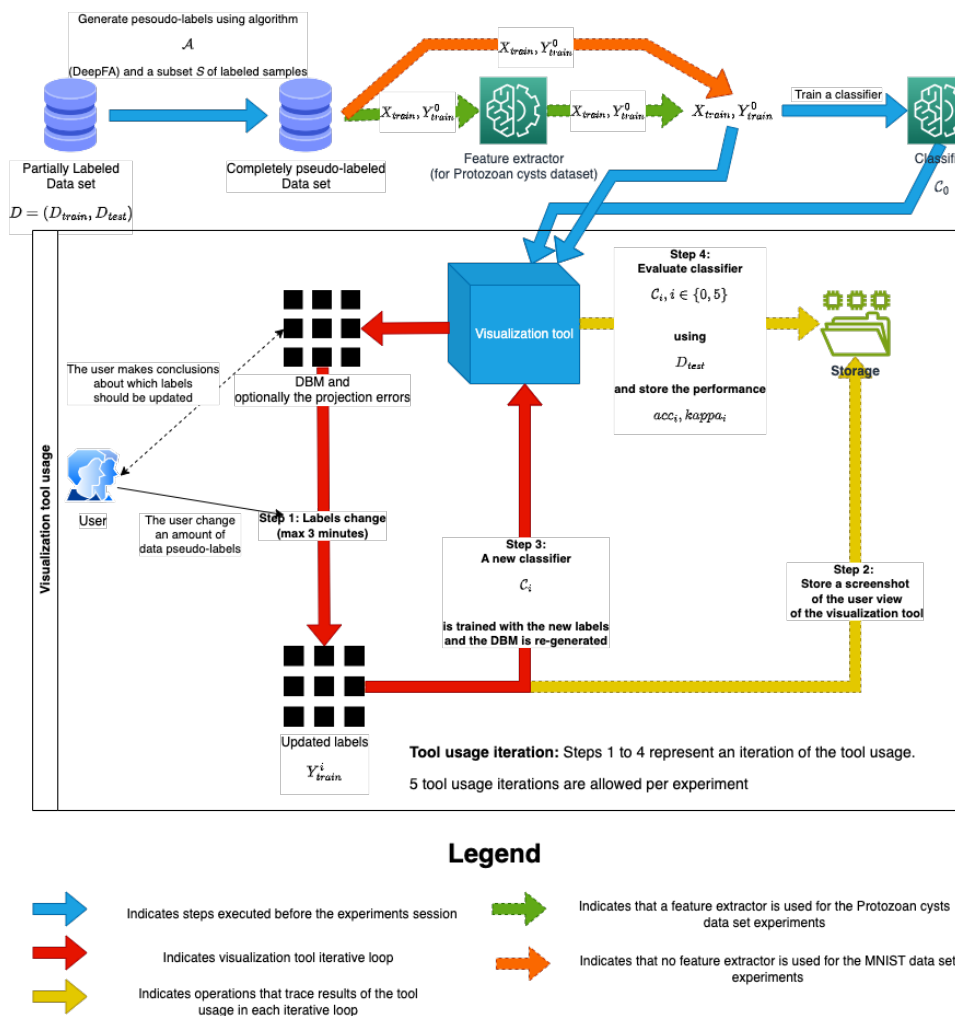


FIGURE 4.3: Experimental pipeline for the tool usage evaluation

In this section, we detail the experiments conducted and the methodology applied for assessing the tool’s utility. The experiments engage two tool users, each operating independently. Their results are kept confidential from one another to prevent mutual biasing.

For each experiment, the user is allowed to re-label the data points, once the user is satisfied with the changes he/she clicks the "Apply changes" button in the tool. This initiates a complete re-training of the classifier with the newly indicated training labels. We refer to this process of re-labeling and classifier re-training as an *iteration* of the tool usage (see figure 4.3). Each *iteration* is subject to a 3-minute timeout, meaning the tool allows the user to re-assign labels no more than 3 minutes per iteration. The tool allows for 5 such *iterations*, after which further changes in labels are restricted. Thus, per each experiment in this section, each user is allowed to interact with the tool for 15 minutes.

To set the groundwork for our experiments, we introduce notations detailed in table 4.9. The classifier  $\mathcal{C}$  employed in our experiments has a simple architecture comprising a flattened layer for reshaping the input and a dense layer with a softmax activation function, the number of neurons aligning with the classes in our data set (the architecture is presented in figure 4.1).

As indicated in our experiments pipeline shown in figure 4.3 we generate the pseudo-labels of all the data samples using an algorithm  $\mathcal{A}$  before starting the tool usage. The algorithm  $\mathcal{A}$  utilized is the **DeepFA** [2]. This iterative pseudo-labeling technique enhances deep neural network training by selecting the most confident unsupervised samples (see section 2.1.3 for more details).

In our experiments, we employed a subset of samples  $S \subset D_{train}$ , where  $|S| = 0.01 \cdot |D_{train}|$ , generated by randomly selecting samples from  $D_{train}$ . The pseudo-labels generated by the DeepFA algorithm (i.e.,  $Y_{train}^0$ ) are stored in a Git repository<sup>2</sup>.

In order to ensure experiment reproducibility and safeguard against the stochastic nature of neural networks, we standardized the random number generators in our tool by setting a seed of 42. This seed is applied to every layer of each neural network within the tool. The HeUniform kernel initializer from Keras, with a seed of 42, is employed for each layer of each neural network and the classifier’s dense layer. This meticulous approach aims to minimize variability and promote consistent results in our experiments.

Notation	Description
$\mathcal{C}$	The classifier used in the experiments (see figure 4.1)
$(\mathcal{P}, \mathcal{P}^{-1}) \in \{\text{Autoencoder}, \text{SSNP}, (t - \text{SNE} + \text{NNinv}), (\text{UMAP}, \text{NNinv}), (\text{PCA}, \text{NNinv})\}$	$\mathcal{P}$ - direct projection $\mathcal{P}^{-1}$ - inverse projection
$D = (D_{train}, D_{test}) = ((X_{train}, Y_{train}), (X_{test}, Y_{test}))$	The data set used in the experiments
$\mathcal{A}$	An automatic labeling algorithm
$Y_{train}^0 = \mathcal{A}(S, D_{train})$	Labels obtained with algorithm $\mathcal{A}$ and a subset $S \subset D_{train}$
$Y_{train}^i, i \in \{1, 2, 3, 4, 5\}$	Labels of the training set after iteration $i$
$C_i, i \in \{0, 1, 2, 3, 4, 5\}$	The classifier obtained when trained with $(X_{train}, Y_{train}^i)$
$acc_i, kappa_i, i \in \{0, 1, 2, 3, 4, 5\}$	The accuracy and Cohen’s kappa score when the classifier $C_i$ is evaluated with $D_{test}$
$D_i = (D_{train,i}, D_{test})$ $i \in \{0.2, 0.4, 0.6, 0.8, 1.0\}$	A subset of data set $D$ . The test set is taken from $D$ . The train set is constructed by taking at random $i \cdot  D_{train} $ samples from $D_{train}$

TABLE 4.9: Key notations for tool usage evaluation

After each iteration the tool stores a screenshot with the view of the tool displayed to the user (i.e. step 2 in figure 4.3) then the tool re-trains a new classifier with the same architecture using the updated labels the user provided (i.e. step 3 in

<sup>2</sup><https://github.com/cristi2019255/ParasitesDatasetIndexes>

figure 4.3). After each iteration, the classifier accuracy and Cohen’s kappa score are evaluated and the results are stored (i.e. step 4 in figure 4.3). The tool then uses the new classifier to generate a new DBM image using the Dummy DBM algorithm and displays the new image to the user. This is the point where a new iteration starts.

### MNIST data set experiments series

In our initial experiment series, subsets of the MNIST data set, denoted as  $D_i$  where  $i \in \{0.2, 0.4, 0.6, 0.8, 1.0\}$ , are employed. For each  $D_i$  and each  $(\mathcal{P}, \mathcal{P}^{-1}) \in \{\text{Autoencoder}, \text{SSNP}, (t - \text{SNE}, \text{NNinv}), (\text{UMAP}, \text{NNinv}), (\text{PCA}, \text{NNinv})\}$ , each user initializes the tool with  $D_i = (D_{\text{train},i}, D_{\text{test}})$ ,  $(\mathcal{P}, \mathcal{P}^{-1})$ , and  $Y_{\text{train},i}^0 = \text{DeepFA}(S, D_{\text{train},i})$ . Users interact with the tool through 5 iterations, with each iteration  $j \in \{1, 2, 3, 4, 5\}$  involving the re-labeling of data points in  $Y_{\text{train},i}^{j-1}$ . After each iteration  $j$ , classifier performance metrics  $acc_j$  and  $kappa_j$  are monitored. The primary objective of this experiment series is to quantitatively assess the impact of different projection methods on classifier performance improvement. Both users possess knowledge about the data set, with User 1 having familiarity with the visualization tool and User 2 being new to its use (see table 4.10).

	Knowledge about the tool (i.e. has used the tool before)	Knowledge about data set (i.e. has seen the data set before)
MNIST data set experiments session		
User 1	Yes	Yes
User 2	No	Yes
Protozoan cysts experiments session		
User 1	Yes	No
User 2	Yes	Yes

TABLE 4.10: Users’ knowledge in experimental sessions

### Protozoan cysts data set experiments series

Following the identification of the best  $(\mathcal{P}, \mathcal{P}^{-1})$  pair in the initial experiment session, each user employs the tool to refine pseudo-labels generated by DeepFA for a more intricate real-world data set—the Protozoan cysts. As this experiment builds upon the initial one, User 2 already possesses knowledge about the tool. User 2 has more extensive knowledge about the Protozoan cysts data set compared to User 1, who encounters the data set for the first time in this experiment (see table 4.10). This experiment series aims to quantitatively assess the tool’s utility in the context of a complex real-world data set. Additionally, we aim to evaluate the impact on classifier performance when the tool is used by someone with no prior domain knowledge compared to someone with some domain knowledge, quantifying the extent of this discrepancy.

The Protozoan cysts data set presents significantly higher dimensionality than the MNIST data set, with raw images being of size 200x200x3 compared to 28x28x1. Using raw images from this data set with the same classifier as described earlier would yield poor results for such complex data points. Moreover, there is a possibility that even if a user accurately corrects all the labels, the classifier performance may be inferior when trained with the correct labels when compared to training with initially incorrect pseudo-labels due to the limitations of the classifier. To mitigate this issue in our experiments while still utilizing the same simple classifier  $\mathcal{C}$ , we introduce a dimensionality reduction step before employing the tool (i.e. see feature extractor in figure 4.3). This means that the tool and the classifier are fed with

reduced feature vectors rather than raw images. To achieve this dimensionality reduction, we utilize an Autoencoder, following the approach outlined in the paper [19]. The architecture of the Autoencoder is outlined in figure 4.4. Let *encoder* and *decoder* represent the encoder and decoder components of the Autoencoder. In this experiment, instead of  $X_{train}$  and  $X_{test}$ , we use  $X_{train,features} = \text{encoder}(X_{train})$  and  $X_{test,features} = \text{encoder}(X_{test})$ . The visualization tool can leverage the *decoder* function to generate and display images in the tooltip for each pixel of the DBM.

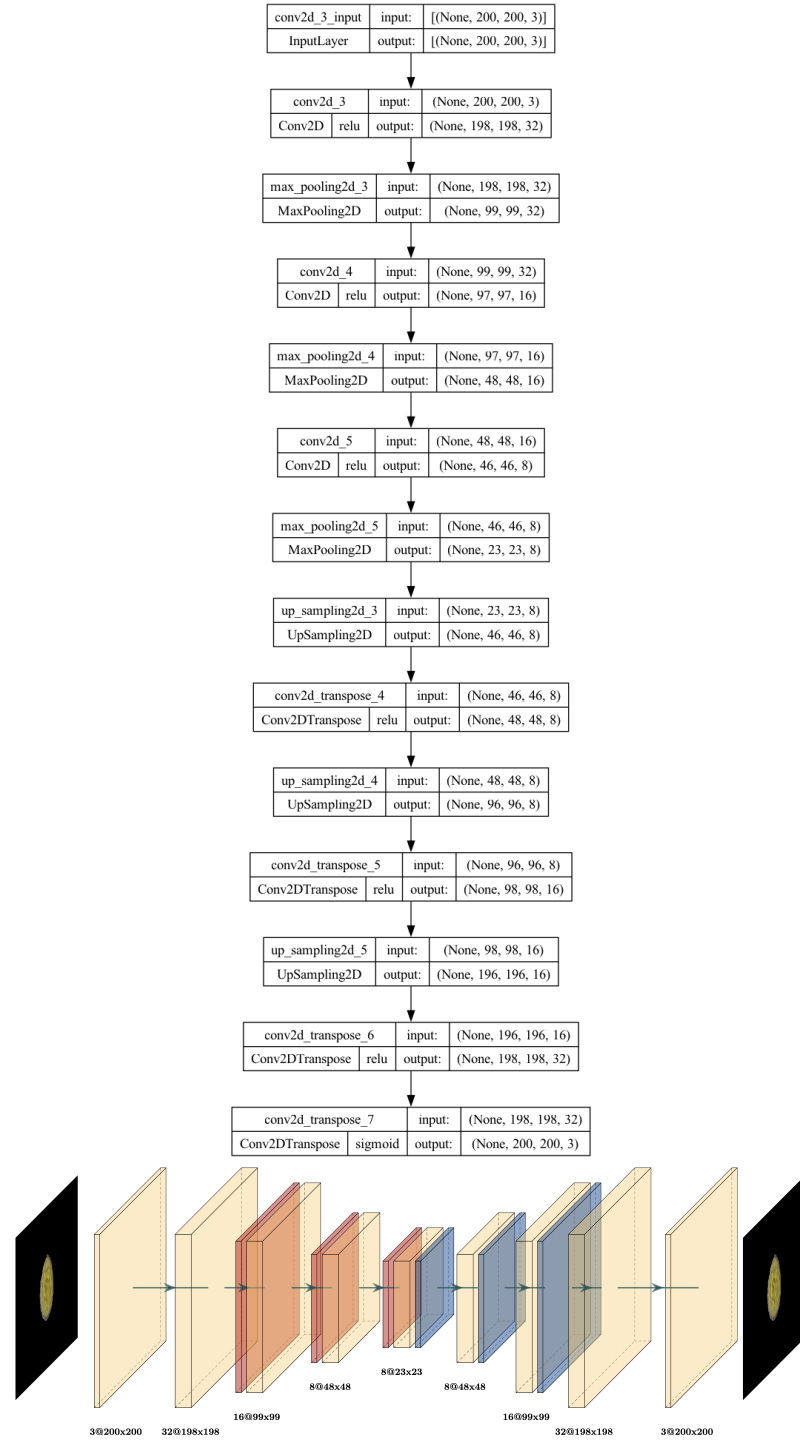


FIGURE 4.4: Feature extractor architecture for Protozoan cysts data set



### 4.3.2 Results and Discussions

In this section, we present the results of the experiments outlined in the previous section, coupled with discussions on the utility of the visualization tool in guiding users through the construction of a correctly labeled data set. The structure of this section is as follows: initially, we present the outcomes of the experiment series utilizing subsets of the MNIST data set paired with various projection and inverse projection methods. Subsequently, the following section delves into the results of the experiment series involving the Protozoan cysts data set, combined with the most effective projection and inverse projection pair identified in the initial experiment series.

#### MNIST data set experiments series

As outlined in the methodology section, we derived a subset  $S$  from  $D_{train}$  and employed the **DeepFA** algorithm to construct  $Y_{train}^0$ . Subsequently, we randomly selected a subset of the training set to create  $D_i$ , where  $i \in \{0.2, 0.4, 0.6, 0.8, 1.0\}$ . We proceeded to train a classifier  $\mathcal{C}$  using  $X_{train,D_i}$  and  $Y_{train,D_i}^0$  (pseudo-labels of the training set of  $D_i$  computed by DeepFA), followed by the evaluation of its accuracy and kappa score. We trained and evaluated the same classifier  $\mathcal{C}$  using  $X_{train,D_i}$  and  $Y_{train,D_i}$  (the true labels of the training set of  $D_i$ ). Table 4.11 presents the absolute and relative differences between  $Y_{train,D_i}$  and  $Y_{train,D_i}^0$  in the first row (i.e. the number of labels with wrong values). The subsequent rows display the classifier performance when trained with  $(X_{train,D_i}, Y_{train,D_i}^0)$  and when trained with  $(X_{train,D_i}, Y_{train,D_i})$  respectively. This table establishes the lower and upper boundaries for classifier performance metrics. When the user initiates the tool, the classifier performance aligns with the metrics described in the second row of table 4.11. When all the labels are correctly assigned, the classifier performance aligns with the values presented in the last row of table 4.11.

	$D_{0.2}$	$D_{0.4}$	$D_{0.6}$	$D_{0.8}$	$D_{1.0}$
Amount of wrong labels after pseudo-labeling and before the tool usage	157 / 700 22.42%	335 / 1400 23.92%	478 / 2100 22.76%	636 / 2800 22.71%	792 / 3500 22.62%
Classifier performance when trained with pseudo-labels					
Accuracy	75.07	77.80	79.87	78.93	75.40
Kappa Score	0.7226	0.7531	0.7761	0.7658	0.7266
Loss	0.8455	0.7481	0.6917	0.7036	0.7518
Classifier performance when trained with ground truth labels					
Accuracy	83.47	88.07	89.60	90.00	89.47
Kappa Score	0.8161	0.8673	0.8844	0.8888	0.8829
Loss	0.6325	0.4830	0.4154	0.3910	0.3800

TABLE 4.11: Impact of pseudo-labels on classifier  $\mathcal{C}$  performance: A comparison with ground truth labels.  
Data set: MNIST, the pseudo-labels generated by DeepFA

$\mathcal{P}, \mathcal{P}^{-1}$	User	$D_{0.2}$		$D_{0.4}$		$D_{0.6}$		$D_{0.8}$		$D_{1.0}$	
		accuracy	kappa	accuracy	kappa	accuracy	kappa	accuracy	kappa	accuracy	kappa
Autoencoder	U1	81.67	0.7961	82.87	0.8094	83.53	0.8169	83.07	0.8117	81.40	0.7932
	U2	81.80	0.7976	79.80	0.7754	81.60	0.7954	82.40	0.8043	81.40	0.7932
SSNP	U1	81.80	0.7976	81.67	0.7961	82.80	0.8087	80.07	0.7784	78.60	0.7621
	U2	78.93	0.7657	78.40	0.7598	79.00	0.7664	79.20	0.7687	79.73	0.7746
t-SNE, NNinv	U1	82.07	0.8005	84.87	0.8317	86.27	0.8473	88.27	0.8695	88.67	0.8740
	U2	84.00	0.8221	85.13	0.8348	84.80	0.8310	86.13	0.8458	86.00	0.8443
UMAP, NNinv	U1	81.67	0.7962	87.27	0.8584	86.67	0.8517	88.87	0.8762	86.33	0.8481
	U2	83.33	0.8147	84.60	0.8288	82.60	0.8065	84.73	0.8303	84.87	0.8318
PCA, NNinv	U1	81.27	0.7917	81.20	0.7909	82.60	0.8065	81.27	0.7917	76.20	0.7355
	U2	76.73	0.7413	77.20	0.7465	78.67	0.7628	76.73	0.7413	73.33	0.7034

TABLE 4.12: Classifier  $\mathcal{C}$  performance after 5 iterations of tool usage.  
Data set: MNIST

Table 4.12 showcases the classifier performance after 5 iterations of tool usage for each user and every combination of  $D_i$  and projection technique. Detailed stack traces of classifier performance after each iteration of tool usage for User 1 are presented in tables 4.13 and 4.14. The same detailed stack traces for User 2 are provided in tables 4.15 and 4.16. Each cell in tables 4.13, 4.14, 4.15 and 4.16 features a plot of the classifier performance metric, accompanied by three helper function plots that offer insights into how the classifier performance evolves over iterations.

For instance, the cell that represents the usage of  $D_{0.2}$  and the PCA projection in table 4.13 shows that the classifier accuracy behaves almost linearly (i.e.  $C_1x + C_2$ ). This suggests that throughout the process, the user consistently improves the labels in each iteration, implying that the projection technique might not be effective, as the user isn't discerning patterns or clusters from the 2D plot of the data set, merely re-assigning labels based on individual data points.

In some cases, the performance plot might follow an exponential trend (i.e.,  $C_1e^x + C_2$ ). For instance, the one that we see in the cell on the intersection of  $D_{0.6}$  column and SSNP row in table 4.13. This trend indicates that in the initial iterations, the user struggled to detect patterns or clusters but gradually understood how to use the tool in conjunction with that projection technique, fixing more and more labels in the later iterations.

On the other hand, if the performance approximates an inverse exponential function (i.e.  $-\frac{C_1}{e^x} + C_2$ ), it suggests that the projection method was effective from the start. The user could identify patterns or clusters early on, making substantial label adjustments in the initial iterations and then focusing on specific points or smaller clusters that required more attention and time. For example, observe the cell on the intersection of the  $D_{0.6}$  column and "t-SNE, NNinv" row in table 4.13. The classifier accuracy increases considerably in the first iterations and the difference in accuracy from one iteration to another becomes smaller and smaller as the iterations keep going.

In some cases, the performance plot might show a decreasing function. For instance, the plots presented in the cells at the intersection of "PCA, NNinv" row and columns  $D_{0.4}, D_{0.6}, D_{0.8}, D_{1.0}$  in table 4.15. This behavior means that the projection method is placing the data points in 2D so badly that the user is constantly making mistakes when re-labeling the data samples.

When the performance trace shows an increasing trend over the iterations, the more it approximates the inverse exponential helper function ( $-\frac{C_1}{e^x} + C_2$ ) the better the projection method is, in the sense that the user is more comfortable when using it.

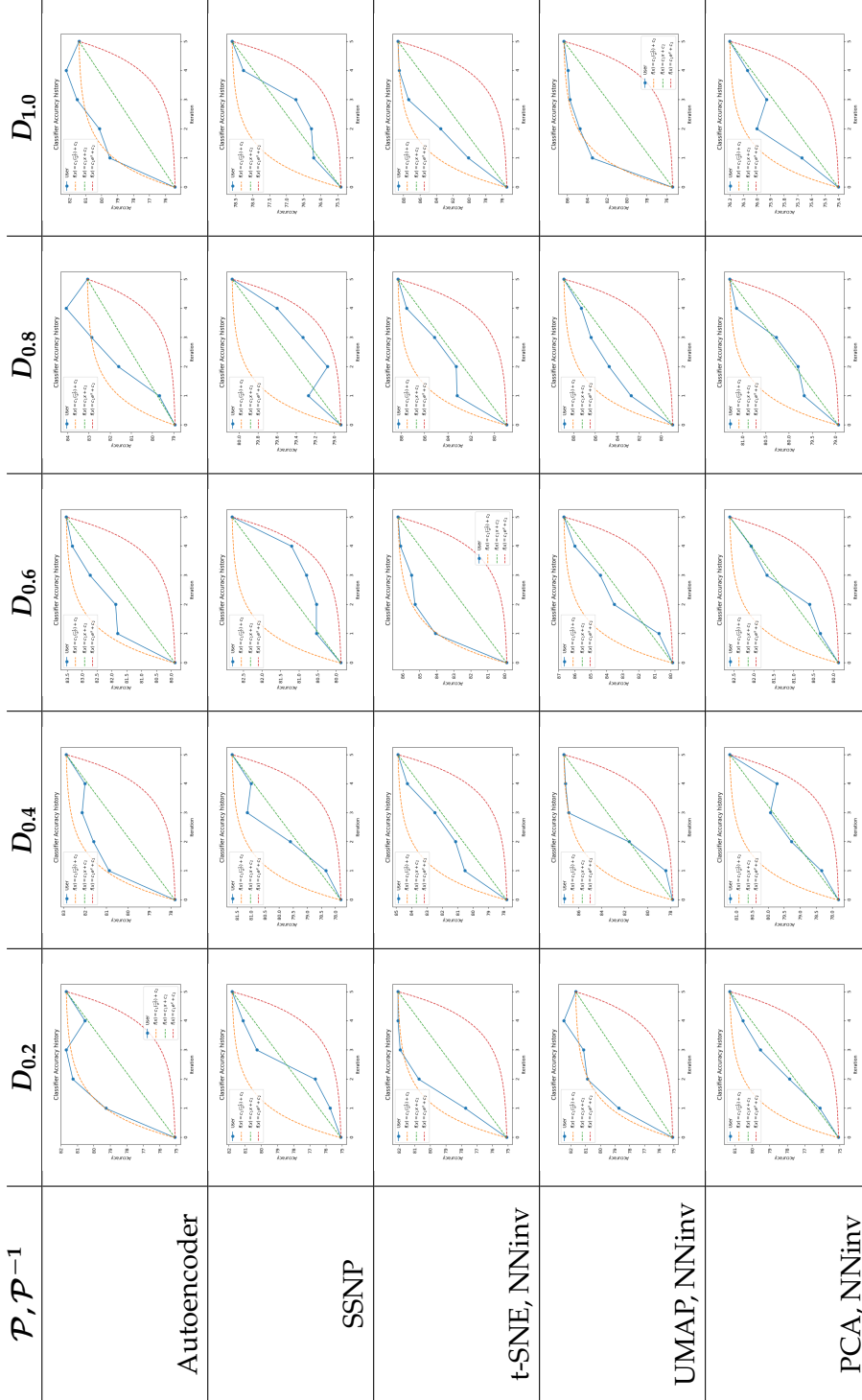


TABLE 4.13: Classifier performance trace over 5 iterations of tool usage.  
Data set: MNIST; User: User 1, Performance metric: Accuracy

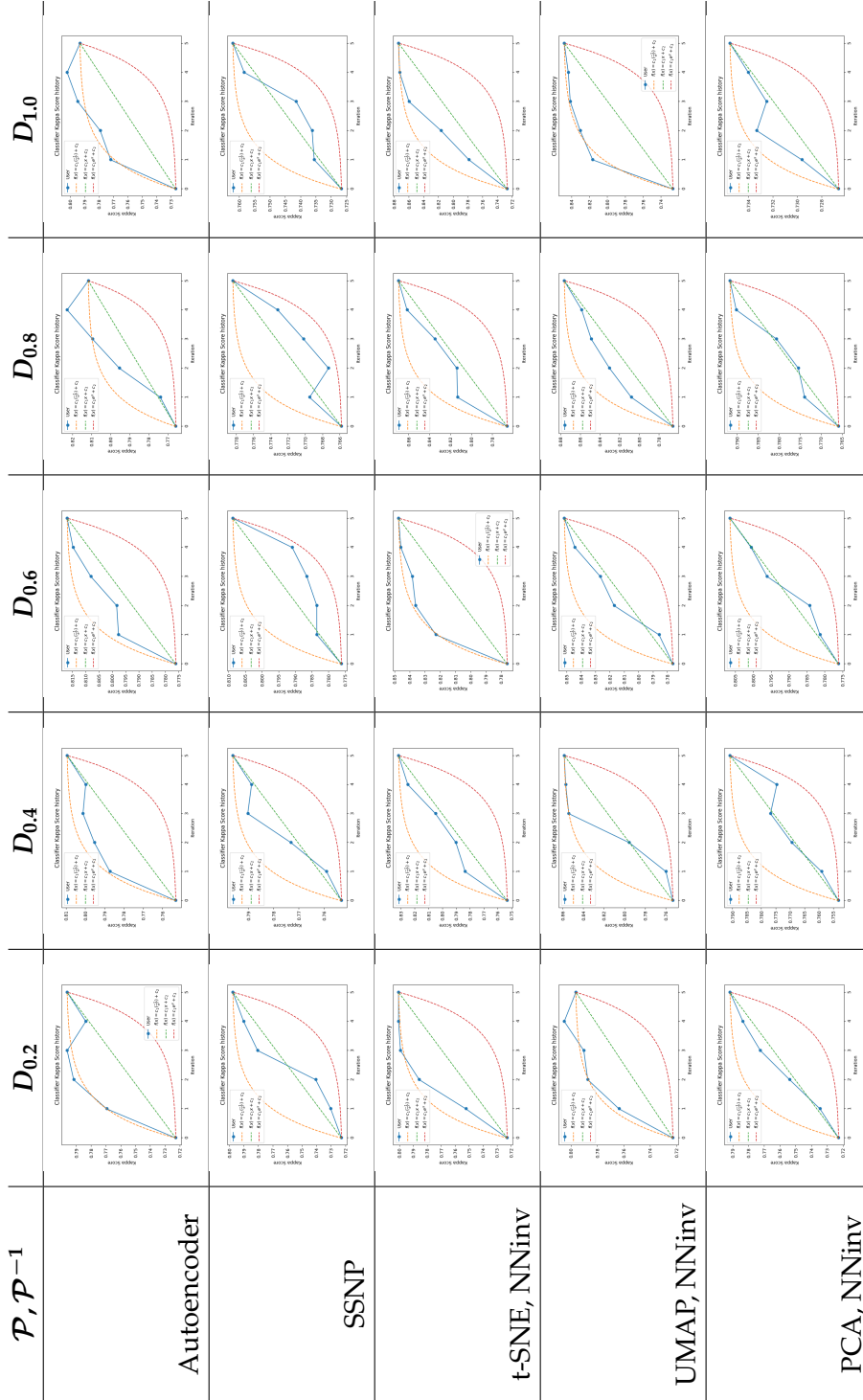


TABLE 4.14: Classifier performance trace over 5 iterations of tool usage.  
Data set: MNIST, User: User 1, Performance metric: Cohen’s kappa score

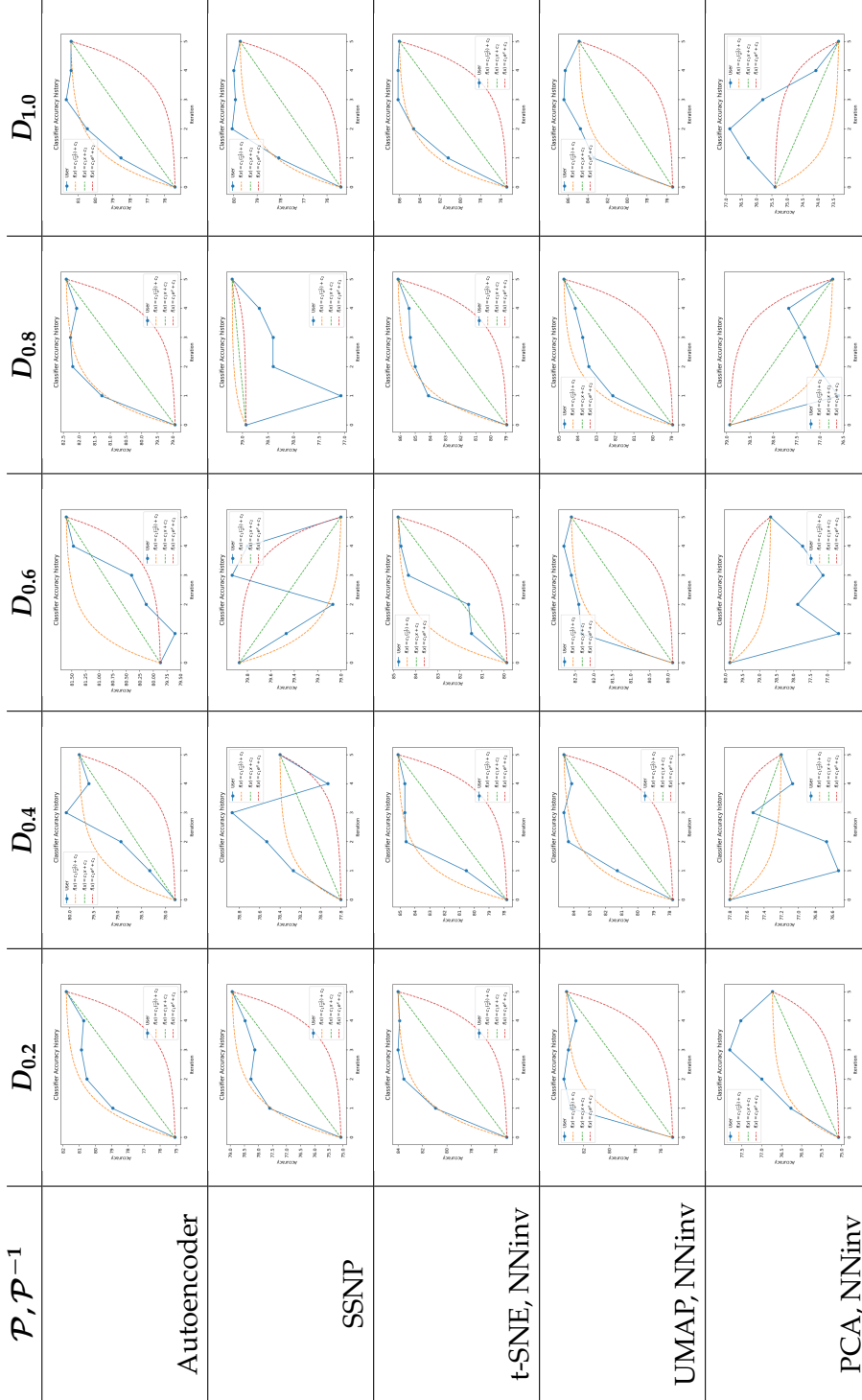


TABLE 4.15: Classifier performance trace over 5 iterations of tool usage. Data set: MNIST, User: User 2, Performance metric: Accuracy

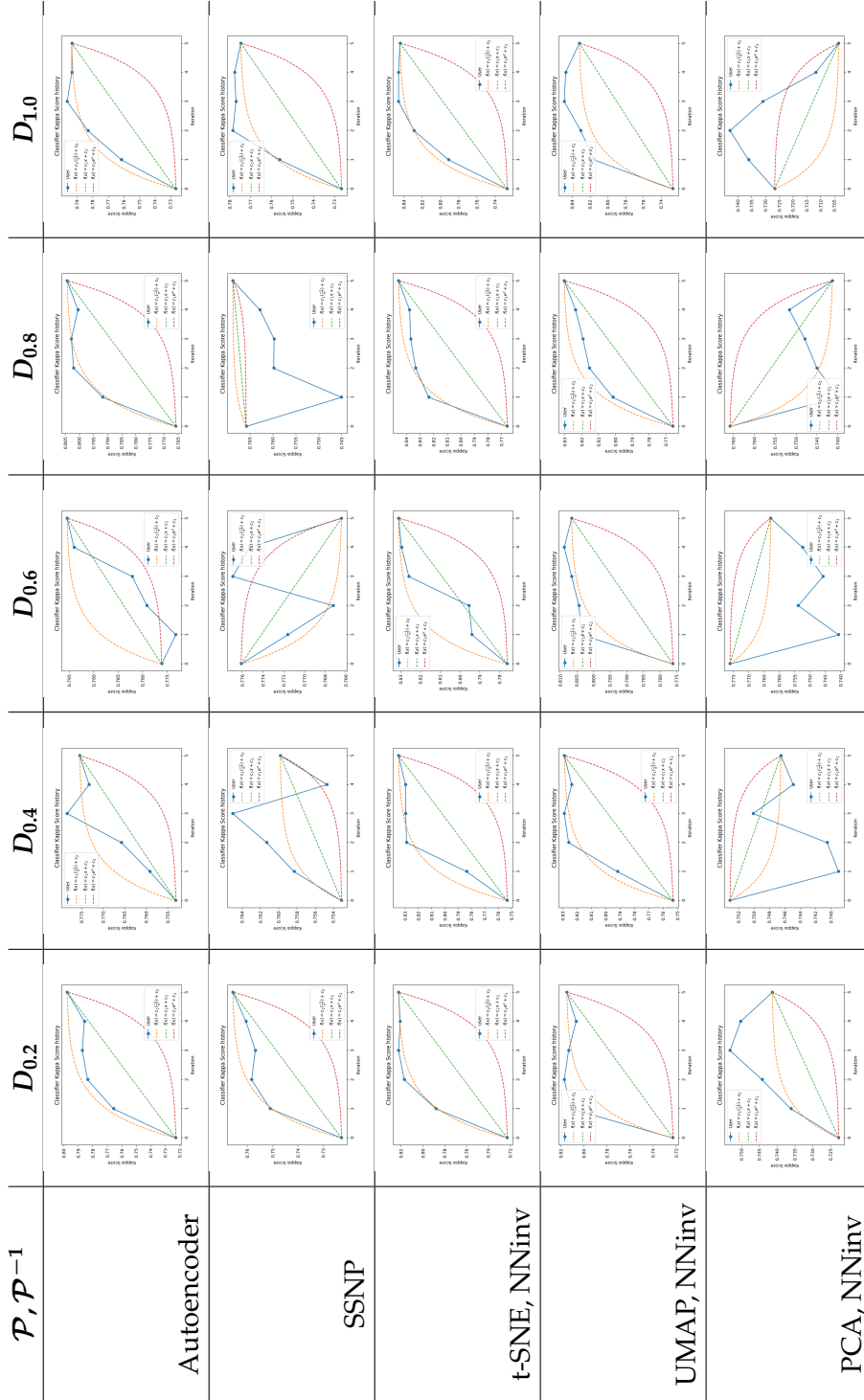


TABLE 4.16: Classifier performance trace over 5 iterations of tool usage. Data set: MNIST, User: User 2, Performance metric: Cohen’s kappa score

Table 4.17 showcases the improvement in classifier performance for several projection techniques. In other words, this table shows the difference in the performance metrics before the user interacts with the tool and after interacting with the tool. Figures 4.5 and 4.6 provide plots for the results from table 4.17 for User 1 and User 2 respectively.

$\mathcal{P}, \mathcal{P}^{-1}$	User	$D_{0.2}$		$D_{0.4}$		$D_{0.6}$		$D_{0.8}$		$D_{1.0}$	
		accuracy	kappa	accuracy	kappa	accuracy	kappa	accuracy	kappa	accuracy	kappa
Autoencoder	U1	6.60	0.0735	5.07	0.0563	3.66	0.0408	4.14	0.0459	6.00	0.0666
	U2	6.40	0.0710	4.40	0.0488	6.20	0.0688	7.00	0.0777	6.00	0.0666
SSNP	U1	6.73	0.0750	3.87	0.0430	2.93	0.0326	1.14	0.0126	3.20	0.0355
	U2	3.86	0.0431	0.60	0.0067	-0.87	-0.0097	0.27	0.0029	4.33	0.0480
t-SNE, NNinv	U1	7.00	0.0779	7.07	0.0786	6.40	0.0712	9.34	0.1037	13.27	0.1474
	U2	8.93	0.0995	7.33	0.0817	4.93	0.0549	7.20	0.0800	10.60	0.1177
UMAP, NNinv	U1	6.60	0.0736	9.47	0.1053	6.80	0.0756	9.94	0.1104	10.93	0.1215
	U2	8.26	0.0921	6.80	0.0757	2.73	0.0304	5.80	0.0645	9.47	0.1052
PCA, NNinv	U1	6.20	0.0691	3.40	0.0378	2.73	0.0304	2.34	0.0259	0.80	0.0089
	U2	1.66	0.0187	-0.60	-0.0066	-1.20	-0.0133	-2.20	-0.0245	-2.07	-0.0232

TABLE 4.17: Classifier  $\mathcal{C}$  performance gains after 5 iterations of tool usage.  
Data set: MNIST

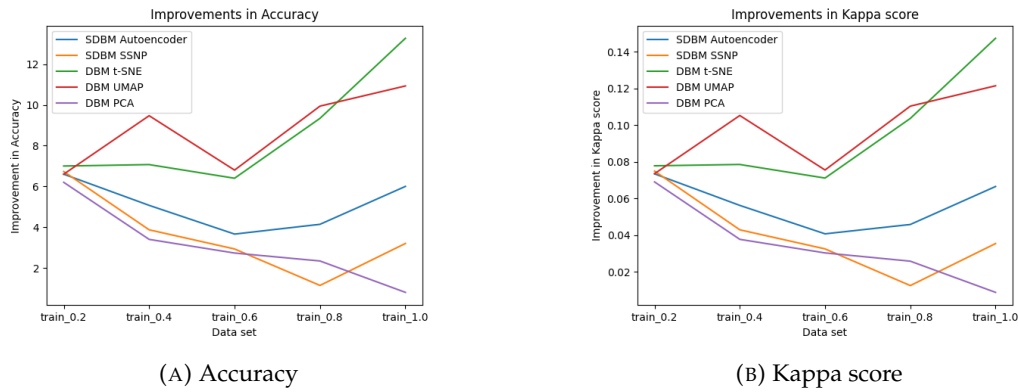


FIGURE 4.5: Classifier  $\mathcal{C}$  performance gains after 5 iterations of tool usage for User 1. Data set: MNIST

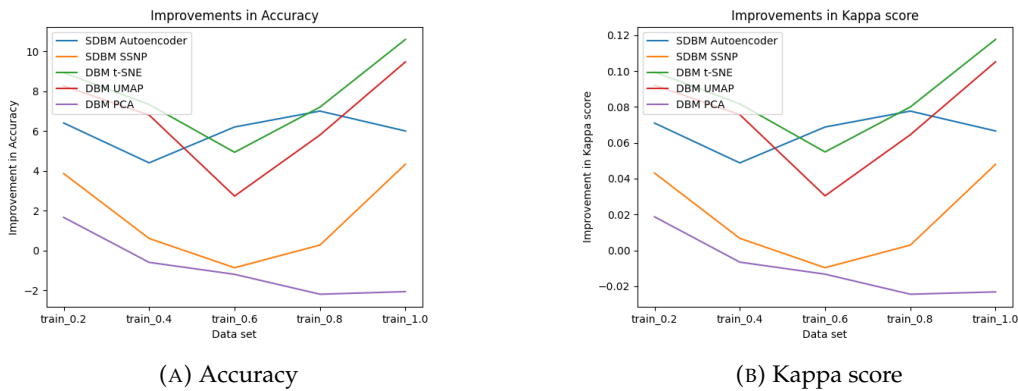


FIGURE 4.6: Classifier  $\mathcal{C}$  performance gains after 5 iterations of tool usage for User 2. Data set: MNIST

Analyzing figures 4.5 and 4.6, it's evident that the improvement in both accuracy and kappa score is less pronounced when more data is utilized for the following projection techniques: SSNP, PCA. Observe that when the tool was used in combination with the PCA projection by User 2, the performance decreased. When the Autoencoder projection method was used, User 1 obtained results comparable to the ones yielded by the SSNP projection in terms of performance gains. User 2 obtained better gains. All in all, the improvement in the classifier performance seems to not exhibit a direct dependency on the amount of data used when the Autoencoder projection is involved. Both users obtained the best gains when either the UMAP or t-SNE projection was used. Notably, for UMAP projection, the improvement in classifier performance, for User 1, does not exhibit a direct dependency on the amount of data used. Conversely, the t-SNE projection illustrates that as more data is employed, the potential improvement in classifier performance that users can achieve increases.

The least improvements in both accuracy and kappa score were observed when PCA or SSNP projections were utilized. Specifically, with PCA projection, User 1 achieved accuracy improvements ranging from 0.8% up to 6.20%. The improvement in the kappa score varied from 0.0089 to 0.0691. User 2 achieved improvements only when  $D_{0.2}$  data set was used, namely 1.66% accuracy improvement and 0.0187 in kappa score. In all other cases, User 2 was not able to improve the initial classifier.

When SSNP projection was used, the accuracy and kappa score improvements for User 1 varied in the ranges of [1.14%, 6.73%] and [0.0126, 0.0750] respectively. User 2 was not able to improve the initial performance when  $D_{0.6}$  data set was used. For all other cases when User 2 was using the SSNP projection, the improvements in accuracy were from 0.27% up to 4.33%, the kappa score improvements are in the range [0.0029, 0.0480].

On the opposite end, the most substantial improvements were observed when t-SNE projection was used. Users obtained classifiers that outperformed the ones trained with pseudo-labels. Specifically, User 1 achieved accuracy improvements in the range [6.40%, 13.27%] and improvements in the kappa score from 0.0712 up to 0.1474. User 2 was able to obtain improvements in the range [4.93%, 10.60%] for accuracy and [0.0549, 0.1177] for the kappa score.

Both users demonstrated the capability to enhance the initial classifier performance in a limited amount of time after employing the visualization tool with projection techniques such as Autoencoder, UMAP, and t-SNE. The clusters were relatively easy to find when using t-SNE and UMAP, as opposed to the more challenging scenarios presented by SSNP and PCA projections.

Based on the results presented in figures 4.5 and 4.6, along with the ones shown in tables 4.13, 4.14, 4.15, and 4.16, we can conclude the following ranking (from best to worst) of projection techniques when used in combination with the visualization tool:

1. t-SNE + NNinv
2. UMAP + NNinv
3. Autoencoder
4. SSNP
5. PCA + NNinv

Observe that when the t-SNE projection is used the performance trace plot almost always approximates an inverse exponential function, which shows that both



users were comfortable when using this projection, making big increases in the performance from the first iterations.

### Protozoan cysts data set experiments series

In this section, we present the experimental results of the second experiment series, as outlined in the methodology. Building upon the insights from the previous section, t-SNE projection was used in this experiment session. The classifier  $\mathcal{C}$  employed in this experiment adheres to the same architecture as described in the methodology (see figure 4.1). The data set used in this experiment is constructed from the feature vectors of the raw images of the Protozoan cysts data set (i.e.,  $X_{train,features}, X_{test,features}$ ), as detailed in the methodology and figure 4.3.

Table 4.18 provides the count of absolute and relative incorrect labels (i.e., labels that differ between  $Y_{train}^0$  and  $Y_{train}$ ), along with the performance metrics of classifier  $\mathcal{C}$  when trained with  $(X_{train,features}, Y_{train}^0)$  and  $(X_{train,features}, Y_{train})$  respectively. Notice that when trained with the ground truth labels, the classifier performance exhibits a difference of 2.25% in terms of accuracy and 0.0334 in terms of the kappa score compared to when trained with the pseudo-labels.

Amount of wrong labels after pseudo-labeling and before the tool usage	388 / 2696 14.39%
Classifier performance when trained with pseudo-labels	
Accuracy	85.64
Kappa Score	0.8043
Loss	0.5410
Classifier performance when trained with ground truth labels	
Accuracy	87.89
Kappa Score	0.8377
Loss	0.3283

TABLE 4.18: Impact of pseudo-labels on classifier  $\mathcal{C}$  performance: A comparison with ground truth labels.

Data set: Protozoan cysts, the pseudo-labels generated by DeepFA

The classifier performance achieved by each user after each iteration is shown in table 4.19. Table 4.20 presents plots of classifier performance per user, along with three helper functions as described in the previous results subsection. Table 4.21 indicates how many labels each user was able to fix after 5 iterations.

Iteration	Accuracy		Kappa score	
	User 1	User 2	User 1	User 2
0	85.64		0.8043	
1	85.38	85.21	0.8019	0.7987
2	85.81	87.28	0.8096	0.8279
3	85.90	87.63	0.8107	0.8322
4	86.16	87.63	0.8131	0.8321
5	86.76	87.37	0.8211	0.8292

TABLE 4.19: Classifier performance trace over 5 iterations of tool usage.

Data set: Protozoan cysts

Both users were successful in achieving improvements in classifier performance when using the visualization tool. User 1 increased the accuracy by 1.12% and the kappa score by 0.0168. User 2 achieved a 1.73% accuracy increase and a 0.0249 increase in the kappa score. It's worth noting that when trained with ground truth labels, the classifier accuracy is 87.89%, and the kappa score is 0.8377. This implies that User 2, through pseudo-labeling and the visualization tool in a limited amount of time, obtained a classifier that performs nearly as well as the one trained with the ground truth labels.

The results obtained for this more complex data set exhibit striking similarities in terms of performance gain when compared to the outcomes achieved for the MNIST data set. Similar trends of performance increase are observed, suggesting that the benefits of the visualization tool in enhancing classifier performance are consistent across different types of data sets, regardless of their complexity. This consistency in results strengthens the argument for the effectiveness and general applicability of the visualization tool in diverse data analysis scenarios.

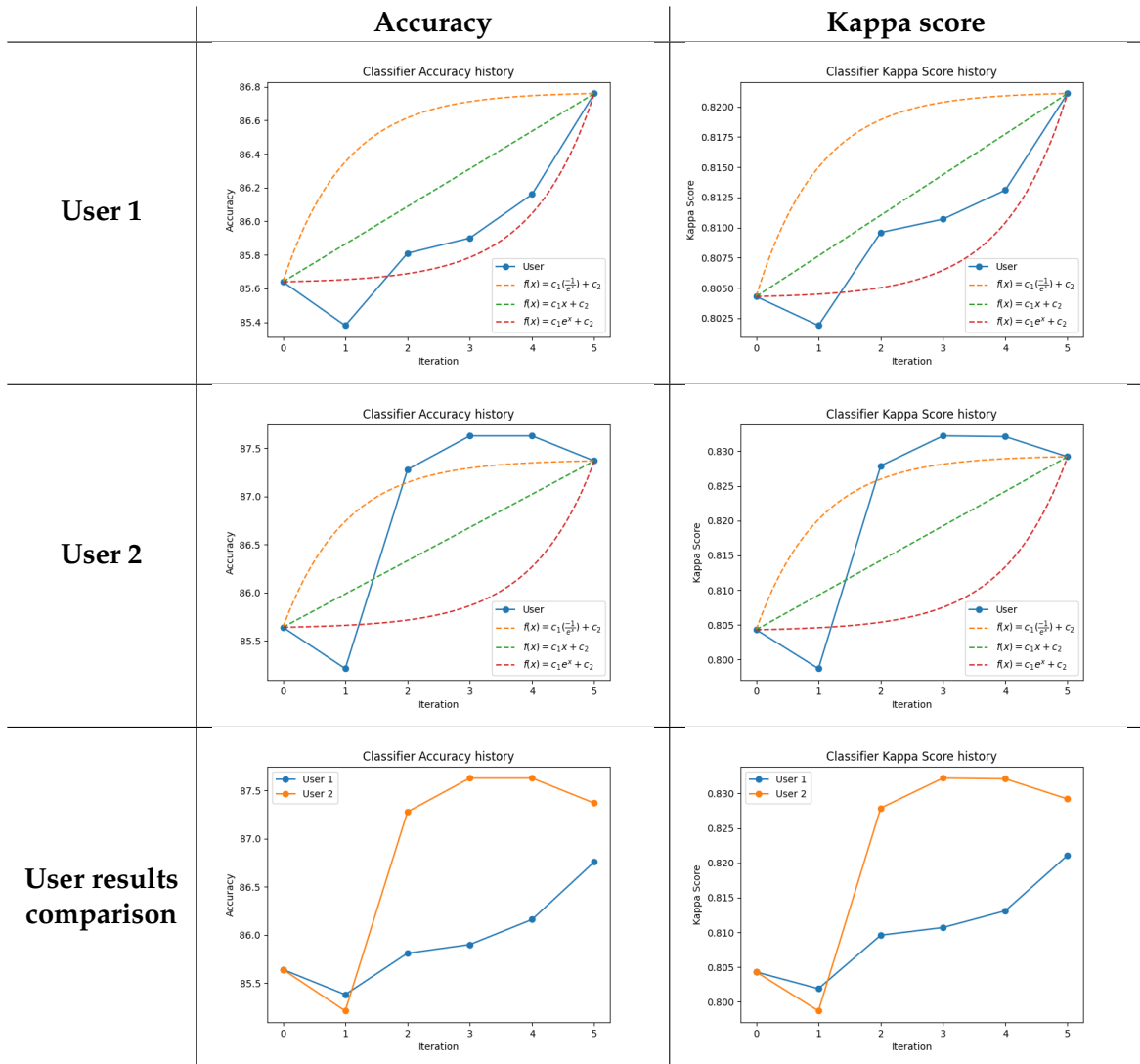


TABLE 4.20: Comparison of classifier performance trace over 5 iterations of tool usage per user.  
Data set: Protozoan cysts

	User 1	User 2
Amount of wrong labels	295 / 2696 ( 10.94% )	359 / 2696 ( 12.09% )
Amount of labels fixed	93 / 2696 ( 3.45% )	29 / 2696 ( 1.08% )

TABLE 4.21: Amount of corrected labels per user after 5 iterations of tool usage.

Data set: Protozoan cysts

From table 4.20 one can observe that User 2 achieved a slightly better classifier performance than User 1. It’s important to note that User 1 had no prior knowledge about the data set before the experiment, while User 2 had previous exposure and experience with the data set (see table 4.10). Interestingly, the results of User 1 show that this user was still able to significantly enhance the classifier’s performance and rectify a substantial number of incorrect pseudo-labels. This is also supported by the results presented in table 4.21.

Observe that User 1, corrected more labels compared to User 2 (i.e. 93 for User 1 and 29 for User 2). Nevertheless, User 2 obtained a slightly better increase in performance, which means that User 2 was able to identify and fix clusters of data samples that had more impact on the classifier training and predictions. In other words, User 2 was able to identify and correct labels of more complex classes.

The outcomes of this section provide valuable insights, suggesting that the visualization tool offers accessible means to identify patterns and clusters in complex data sets, even for users who are not domain experts. Nevertheless, it’s crucial to acknowledge that more evidence is needed to substantiate this hypothesis, and such evidence may not be derived solely from the conducted experiment sessions.

### Quantitative evaluation

If we were to aggregate our findings in some simple averages, we see that in 15 minutes a user achieved an average increase of 5.03% in accuracy and 0.0556 in the kappa score on the MNIST data set. When the tool was used with the t-SNE projection and MNIST the increase on average was of 8.21% in accuracy and 0.0913 in kappa score. An average of 1.43% and 0.0209 increase in the accuracy and kappa score respectively was achieved when the tool was used with the Protozoan cysts data set. We can not generalize this to other data sets or users, however, the results still suggest that there is a measurable added value provided by our visual tool.

The absolute values of the increase in performance depend on the initial classifier  $C_0$  performance and the performance of the classifier when trained with the ground truth labels. To factor out the absolute values we can compute the average increase in performance as follows:

$$c_{metric} = \frac{\sum_{i \in \{1,2\}} (c_{MNIST, user_i, metric} + c_{Protozoan\_cysts, user_i, metric})}{4} \quad (4.1)$$

where  $metric$  is either accuracy or kappa score, and  $c_{MNIST, user_i, metric}, c_{Protozoan\_cysts, user_i, metric}$  are the achieved increase divided by the possible increase (i.e. the value of the metric when the classifier with ground truth labels is trained minus the value of the metric achieved with  $C_0$ ). For instance, when using the Protozoan cysts data set the first user got a classifier that has an accuracy of 86.76%, the  $C_0$  classifier accuracy is 85.64% and the accuracy of the classifier when trained with the ground truth labels is 87.89%. Therefore,  $c_{Protozoan\_cysts, user_1, accuracy} = \frac{86.76 - 85.64}{87.89 - 85.64} = \frac{1.12}{2.25} = 0.4978$ . Analogously,  $c_{Protozoan\_cysts, user_2, accuracy} = \frac{1.73}{2.25} = 0.7689$ . For the MNIST if we take

the results of  $D_{1.0}$  and t-SNE projection,  $c_{MNIST,user_1,accuracy} = \frac{13.27}{14.07} = 0.9431$  and  $c_{MNIST,user_2,accuracy} = \frac{10.60}{14.07} = 0.7534$ . Thus,  $c_{accuracy} = \frac{0.4978+0.7689+0.9431+0.7534}{4} = 0.7408$ . Analogously,  $c_{kappa\_score} = \frac{0.9430+0.7530+0.5030+0.7455}{4} = 0.7361$ .

By starting with an accuracy  $x$  and investing 15 minutes in fixing pseudo-labels by means of the visualization tool when used together with the t-SNE projection a user could obtain a classifier that on average will have an accuracy of  $x + 0.7408 \cdot (y - x)$ , where  $y$  is the accuracy of the classifier when trained with the ground truth labels. When starting with a kappa score  $x$  the increase in the kappa score on average is  $0.7361 \cdot (y - x)$ , where  $y$  is the kappa score of the classifier trained with the correct ground truth labels. These aggregates show us the true power of the proposed visualization tool.

### Projection and inverse projection errors maps usefulness

One of the goals of the current section is to qualitatively evaluate the impact of the direct and inverse projection error visualizations on users' decisions. The users' feedback suggests a hypothesis that the projection and inverse projection error visualizations are prompting users to concentrate and allocate more attention to regions with high error values when rectifying pseudo-labels. This hypothesis implies that the error maps serve as effective guides, directing users to areas where the pseudo-labels may be less accurate or ambiguous. Further investigation and user studies could provide additional evidence to validate or refine this hypothesis, offering insights into the specific mechanisms through which these visualizations contribute to the label-fixing process

## 4.4 Conclusions

In this chapter, we conducted two primary sets of experiments: the DBM optimization heuristic algorithms experiments and the visualization tool evaluation.

### DBM optimization heuristic algorithms

The first set of experiments, outlined in section 4.2, aimed to determine the conditions under which our proposed DBM computation heuristics could serve as a viable alternative to the Dummy DBM algorithm. Through hyperparameter tuning, we identified the optimal parameters for each heuristic. All heuristics exhibited a significant speed improvement compared to the vanilla DBM algorithm, with highly accurate DBM images generated. Notably, the Binary Split algorithm emerged as the best practical alternative to the Dummy DBM algorithm.

### Visualization tool usage

In our second experiment set, we empirically demonstrated that the proposed tool effectively assists users in enhancing classifier performance (when t-SNE projection was used the accuracy and kappa score increased on average with 8.21% and 0.0913 on the toy MNIST data set and with 1.43% and 0.0209 respectively on the complex real-world Protozoan cysts data set) within a limited amount of time (15 minutes). While we cannot definitively conclude that the tool will always provide the same degree of assistance, we can assert that the proposed tool plays a constructive role in refining labels and boosting a classifier's performance. To make precise claims about the extent to which the tool aids in this process, a more comprehensive set of experiments is essential. This expanded experimentation should encompass a

variety of data sets and involve multiple users, providing a broader perspective on the tool's effectiveness in diverse contexts.

We quantified the tool's effectiveness in fixing pseudo-labels with different projection methods, highlighting its optimal performance when used in conjunction with the t-SNE projection. Our experiments led to some open hypotheses, such as the potential impact of projection and inverse projection error visualizations on users' decision-making regarding regions of focus when fixing labels. Another hypothesis explores the idea that, regardless of a user's domain expertise, non-expert users aided by the visualization tool can achieve almost similar success in identifying patterns and correcting mislabels as the users with domain knowledge. Substantiating or challenging these hypotheses will be the focus of future work, involving larger data sets and more users.

In conclusion, this chapter substantiates the effectiveness of the proposed hybrid pipeline, where users commence with a sparse set of labeled data points, leverage a pseudo-labeling algorithm to assign labels to the entire data set, and subsequently engage in manual correction within a limited time frame using the proposed visualization tool. The demonstrated outcomes underscore the achievement of a near-optimal trade-off between the time invested in transforming a semi-labeled data set into a fully labeled one and the accuracy of the pseudo-labels. This strategic approach proves to be a balanced and efficient methodology for enhancing data set labeling precision within practical constraints.



## Chapter 5

# Conclusions And Future Work

### 5.1 Conclusions

This project focuses on the challenge of training machine learning models, specifically classifiers, within a semi-labeled data set context. We have introduced a semi-automatic pipeline designed to assist users in developing such classifiers. The outlined workflow involves automatic pseudo-labeling and manual correction of labels through a visual tool, aiming to accurately transform the semi-labeled data set into a fully labeled one.

Within this project, we scrutinized various visualizations intending to guide users of such a tool towards optimal outcomes. Our system offers a comprehensible representation of data sets through dimensionality reduction from  $nD$  to 2D space. Additionally, we introduced a modified metric for calculating projection errors at a sample-based level, departing from conventional global metrics that only assess projection errors on the entire data set level. Furthermore, our system offers ways for gaining insights into the classifier model developed by the user through the utilization of decision boundary maps (DBMs).

We introduced three novel heuristics designed to efficiently produce precise, high-resolution DBM images. The potential applications of these optimizations extend beyond the confines of this project, laying the groundwork for multiple visualization tools aimed at providing almost real-time insights into the behavior of classifiers. Quantitative evaluation results indicate that the proposed optimization techniques significantly outperform in terms of speed the simplistic approach, which involves employing a classifier for each pixel of the DBM image (referred to as the Dummy DBM algorithm). Specifically, the Binary Split algorithm exhibits a speed enhancement of one order of magnitude. Furthermore, concerning accuracy, our evaluation suggests that the proposed optimization heuristics, in particular the Binary Split approach, can serve as practical alternatives to the straightforward Dummy DBM approach. The label error rate of a DBM image generated with Binary Split does not exceed 0.3%.

The quantitative assessment of our proposed tool indicates that, even when applied for a limited duration (15 minutes) on complex real-world data sets, users can effectively identify and rectify erroneous pseudo-labels. This corrective action subsequently leads to an improved performance of the classifier under development. The degree of benefit derived from the tool varies depending on the data set's characteristics. Drawing a general conclusion about a lower limit or an average performance gain when using this tool is not feasible solely from the experiment series provided in this project.

Based on the outcomes of the experiments, an open hypothesis regarding the visualization tool has been formulated. When assisted by the tool, the impact of domain knowledge level on performance gains is minimal. In other words, the increase

in performance is nearly equivalent when the tool is used by a non-domain expert compared to a domain expert.

## 5.2 Future work

In the prospect of future research endeavors, the validation of our open hypothesis regarding the visualization tool emerges as a critical imperative, necessitating an extensive series of experiments across diverse data sets and involving users with varied backgrounds. This pivotal avenue holds the potential to offer deeper insights into the tool's utility and forms the groundwork for ongoing enhancements. Moreover, such experiments might help in formulating a general hypothesis about the degree to which the visualization tool assists manual annotation of training sets, based on the data set's characteristics.

Further strides in refining the proposed DBM computation heuristics present an enticing focus for future work. The designing of algorithms that leverage parallel computing advantages holds promise in further significantly reducing run times for the generation of high-resolution DBM images, potentially reaching the realm of real-time processing. For instance, a straightforward improvement for the Confidence Interpolation heuristic involves the parallel interpolation of each confidence function. Algorithms that can generate a DBM image in almost real-time can facilitate the development of visualization tools meant to assist classifier designing.

Another compelling trajectory for future research lies in a comparative analysis of performance gains when such a tool as the one developed in this project undertakes the projection of data sets and generates DBMs in 3D space, rather than in 2D. Such a study can tell us for which types of projections the user is more comfortable when the data is presented in 2D and for which in the 3D space. This analysis can help us to improve our visualization tool and make its purpose more general.

The exploration of additional metrics that can visually present information to users, aiding in swift and accurate decisions regarding the correctness of data point pseudo-labels, stands out as a concept of considerable interest for future investigations. This direction has the potential to empower users with enhanced decision-making capabilities, contributing to the overall robustness and user-friendliness of the visualization tool.



## Appendix A

# Figures

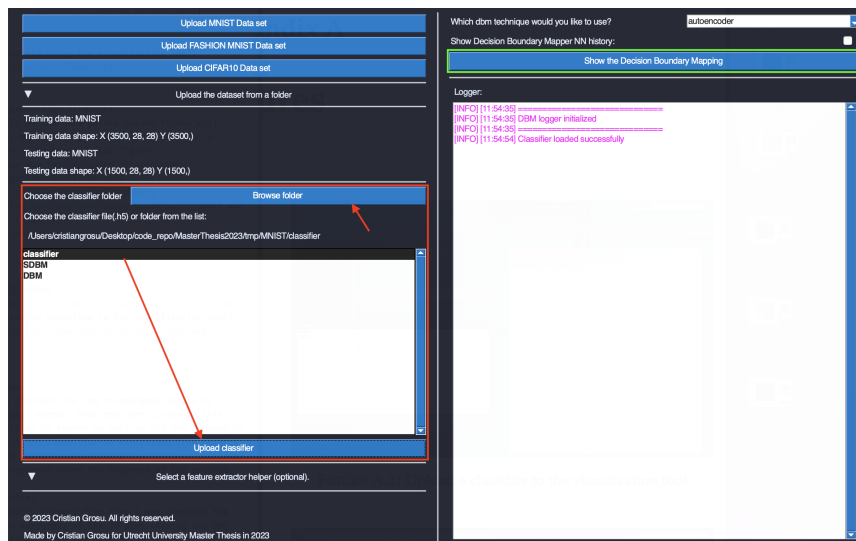


FIGURE A.1: Upload a classifier to the visualization tool

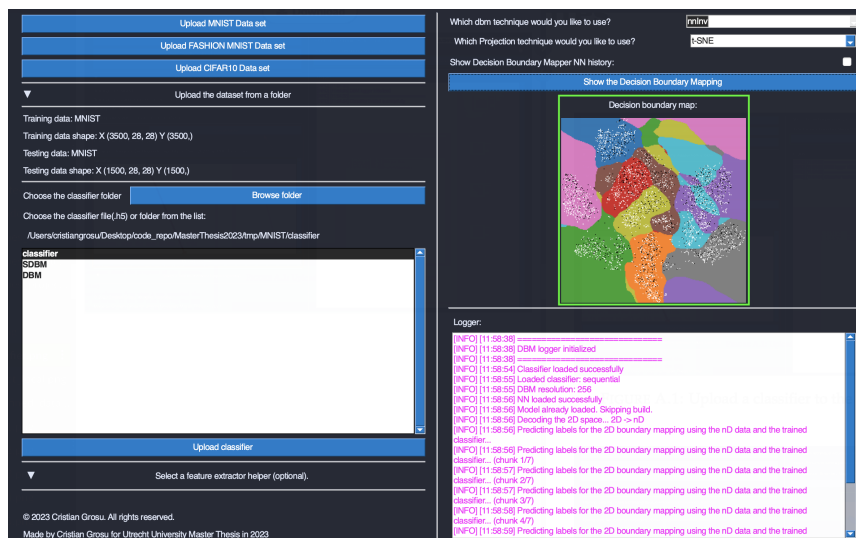
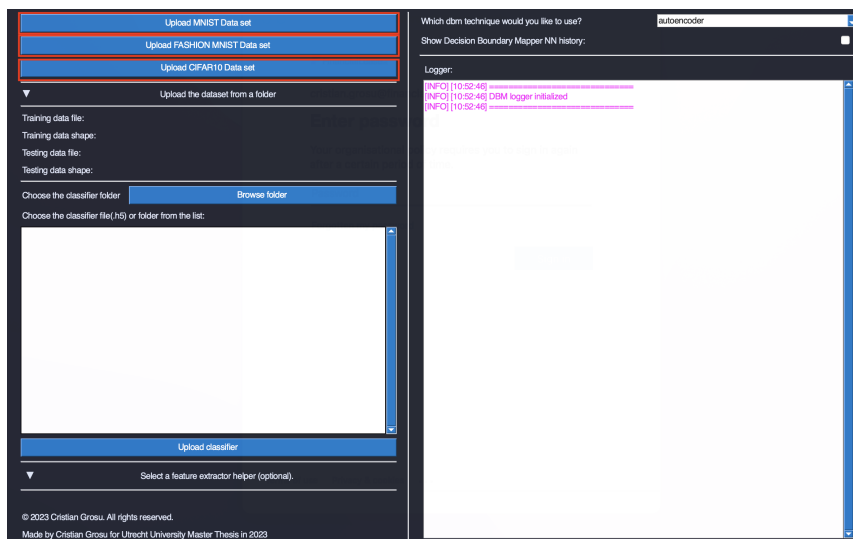
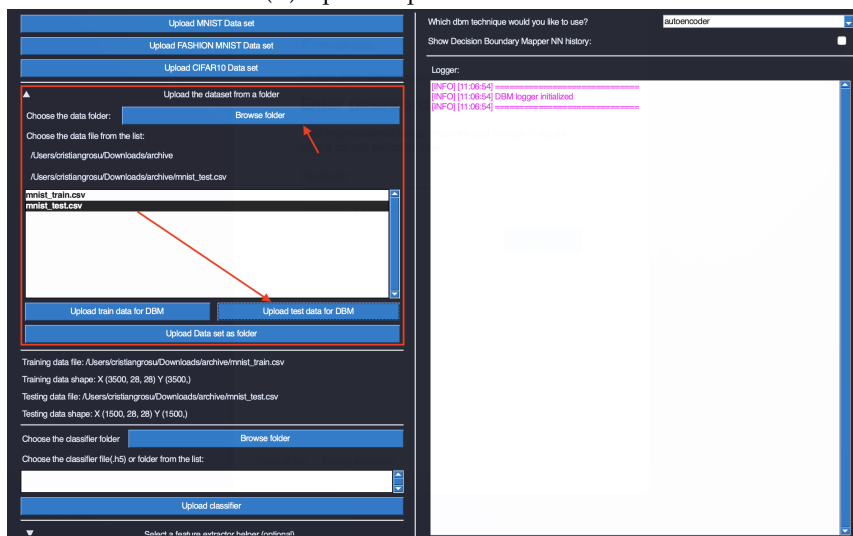


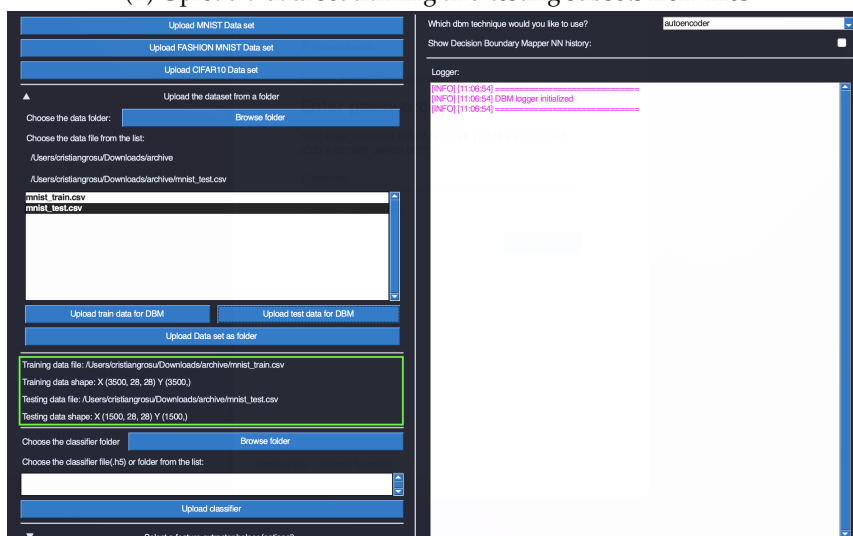
FIGURE A.2: Successfully computed DBM in the configuration window



(A) Upload a predefined data set

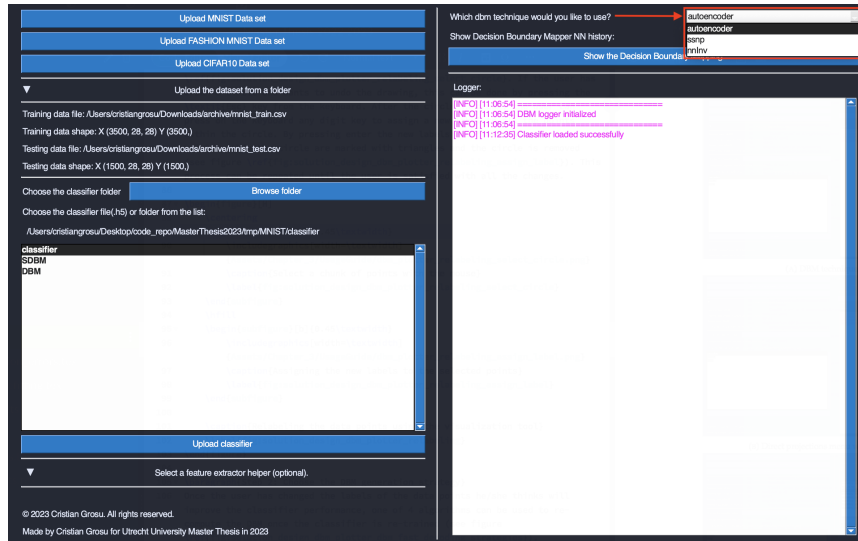


(B) Upload a data set training and testing subsets from files

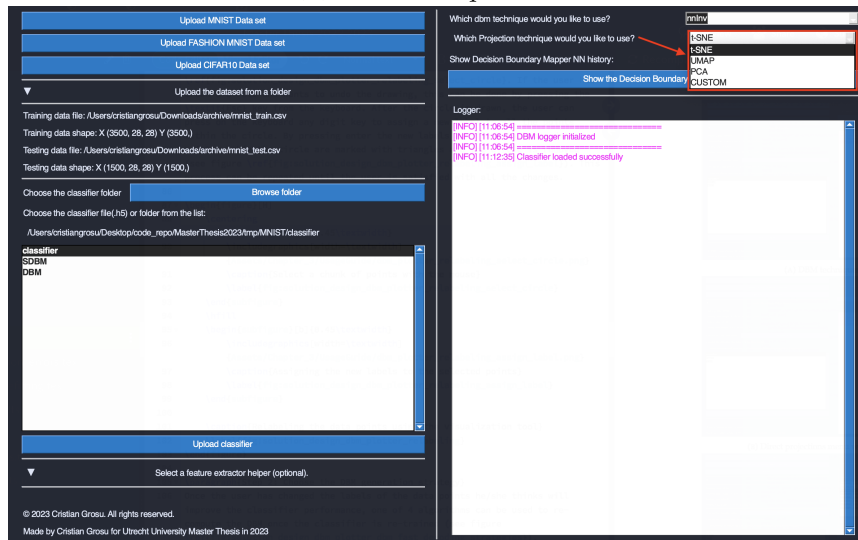


(C) Data set has been uploaded successfully

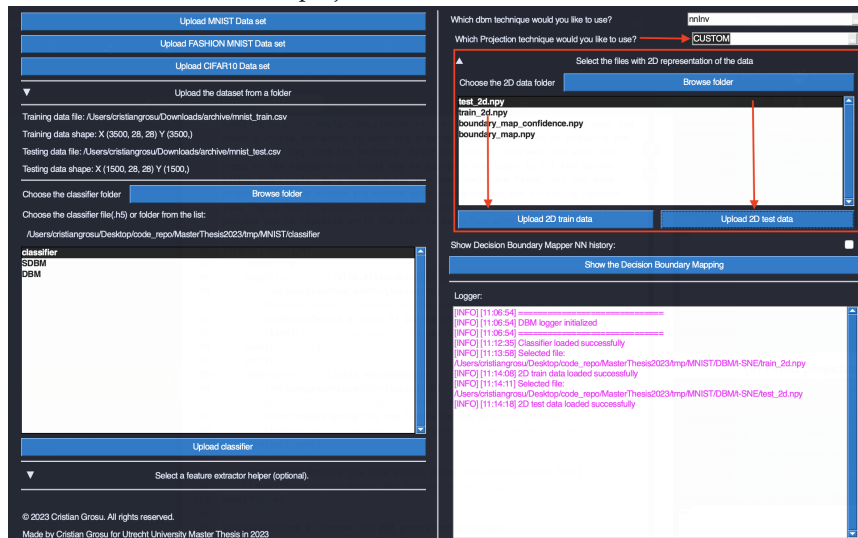
FIGURE A.3: Uploading a data set to the visualization tool



(A) DBM techniques menu



(B) Direct projections menu when NNinv used



(C) Upload custom direct projection

FIGURE A.4: The DBM techniques menu

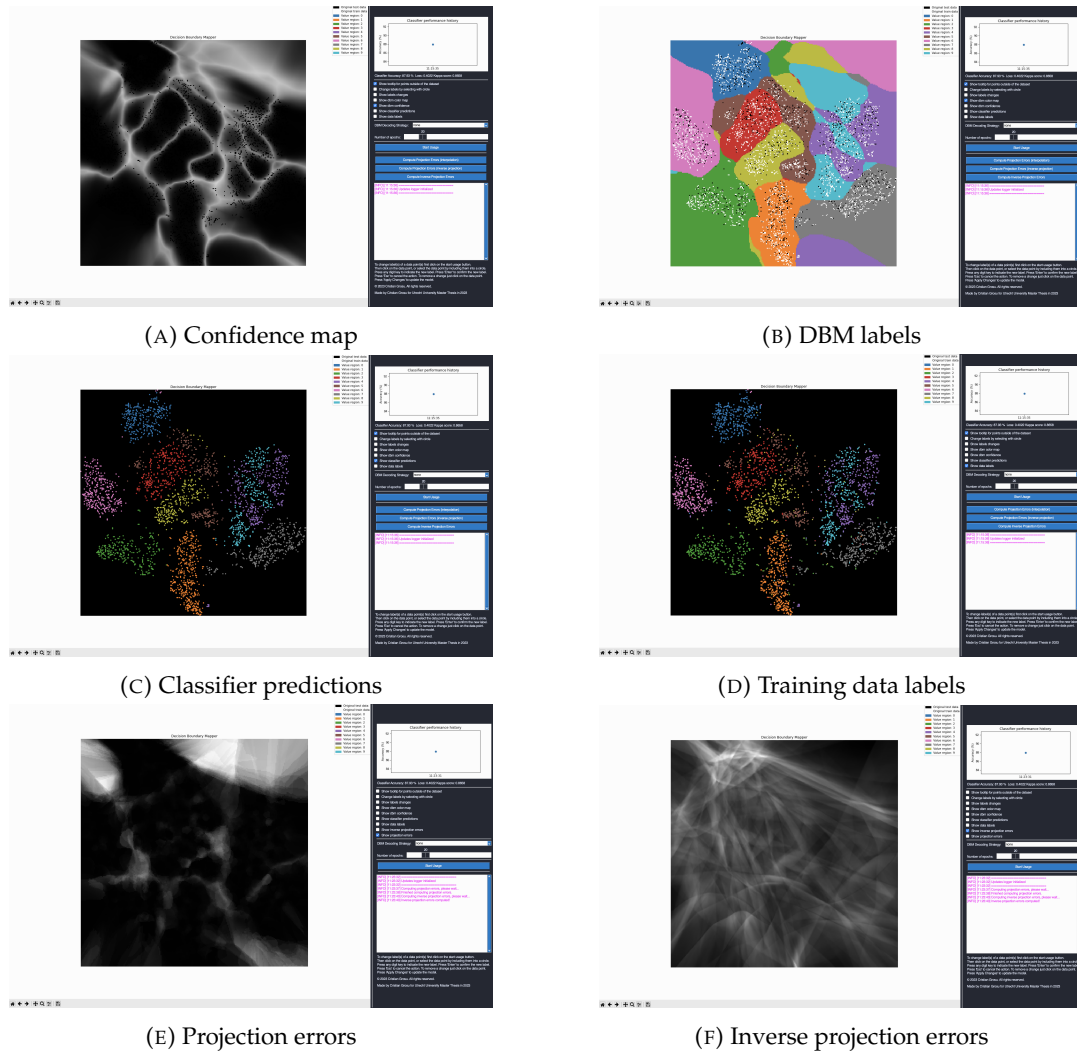


FIGURE A.5: DBM plots based on the checkboxes from DBM visualization window

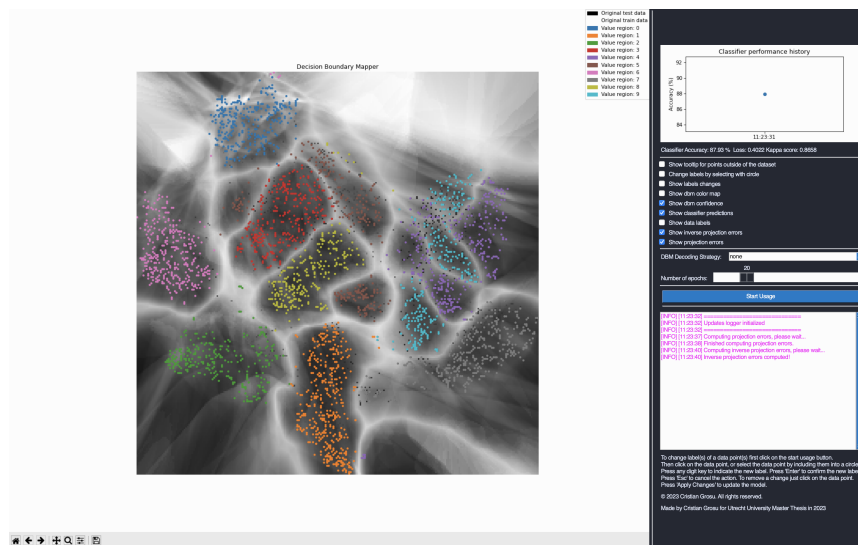


FIGURE A.6: Example of confidence, direct and inverse error values encoded in the opacity of the pixels

## Appendix B

# Experiments environment set up

In this appendix, we present the environment set up for all of the experiments presented in this project.

We developed our visualization tool in Python programming language version 3.10.10. Table B.1 presents all the related libraries we used to develop our visualization tool. Table B.2 presents the (hardware and software) relevant details of the machine on which we run our experiments.

Library	Version
dask	2023.2.0
keras	2.11.0
matplotlib	3.5.1
numba	0.56.2
numba_progress	0.0.4
numpy	1.22.2
opfython	1.0.12
pandas	1.5.3
Pillow	9.4.0
PySimpleGUI	4.60.4
scikit_learn	1.2.2
scipy	1.8.0
tensorflow	2.11.0
termcolor	2.2.0
tqdm	4.62.3
umap	0.1.1

TABLE B.1: All dependent libraries and versions

Characteristic type	Value
Software characteristics	
System Version	macOS 12.6 (21G115)
Kernel Version	Darwin 21.6.0
Hardware characteristics	
CPU	6-Core Intel(R) Core(TM) i7 - 9750H @ 2.60GHz
GPU	Not used
RAM Memory	16GB

TABLE B.2: System characteristics of the machine on which the experiments were run



# Bibliography

- [1] Willian Amorim, Alexandre Falcão, and Marcelo Carvalho. “Semi-supervised Pattern Classification Using Optimum-Path Forest”. In: Aug. 2014, pp. 111–118. DOI: [10.1109/SIBGRAPI.2014.45](https://doi.org/10.1109/SIBGRAPI.2014.45).
- [2] Bárbara C Benato, Alexandru C Telea, and Alexandre X Falcão. “Iterative pseudo-labeling with deep feature annotation and confidence-based sampling”. In: *2021 34th SIBGRAPI Conference on Graphics, Patterns and Images (SIBGRAPI)*. IEEE. 2021, pp. 192–198.
- [3] Bárbara C Benato et al. “Semi-automatic data annotation guided by feature space projection”. In: *Pattern Recognition* 109 (2021), p. 107612.
- [4] Richard L Burden. *Numerical analysis*. Brooks/Cole Cengage Learning, 2011.
- [5] Mateus Espadoto, Nina Sumiko Tomita Hirata, and Alexandru C Telea. “Deep learning multidimensional projections”. In: *Information Visualization* 19.3 (2020), pp. 247–269.
- [6] Mateus Espadoto, Nina Sumiko Tomita Hirata, and Alexandru Cristian Telea. “Self-supervised dimensionality reduction with neural networks and pseudo-labeling”. In: *Proceedings*. 2021.
- [7] Mateus Espadoto et al. “Deep learning inverse multidimensional projections”. In: *Proceedings*. 2019.
- [8] Mateus Espadoto et al. “Toward a quantitative survey of dimension reduction techniques”. In: *IEEE transactions on visualization and computer graphics* 27.3 (2019), pp. 2153–2173.
- [9] Mateus Espadoto et al. “Unprojection: Leveraging inverse-projections for visual analytics of high-dimensional data”. In: *IEEE Transactions on Visualization and Computer Graphics* (2021).
- [10] Y LeCun. *Cortes C(2010) MNIST handwritten digit database*. <http://yann.lecun.com/exdb/mnist>.
- [11] Laurens Van der Maaten and Geoffrey Hinton. “Visualizing data using t-SNE.” In: *Journal of machine learning research* 9.11 (2008).
- [12] Leland McInnes, John Healy, and James Melville. “Umap: Uniform manifold approximation and projection for dimension reduction”. In: *arXiv preprint arXiv:1802.03426* (2018).
- [13] Artur AM Oliveira et al. “SDBM: Supervised Decision Boundary Maps for Machine Learning Classifiers.” In: *VISIGRAPP (3: IVAPP)*. 2022, pp. 77–87.
- [14] Karl Pearson. “LIII. On lines and planes of closest fit to systems of points in space”. In: *The London, Edinburgh, and Dublin philosophical magazine and journal of science* 2.11 (1901), pp. 559–572.
- [15] Paulo E Rauber, Alexandre X Falcao, and Alexandru C Telea. “Projections as visual aids for classification system design”. In: *Information Visualization* 17.4 (2018), pp. 282–305.

- [16] Francisco CM Rodrigues et al. "Constructing and visualizing high-quality classifier decision boundary maps". In: *Information* 10.9 (2019), p. 280.
- [17] Elisa Portes dos Santos Amorim et al. "iLAMP: Exploring high-dimensional spacing through backward multidimensional projection". In: *2012 IEEE Conference on Visual Analytics Science and Technology (VAST)*. IEEE. 2012, pp. 53–62.
- [18] Spyridon Stasis, Ryan Stables, and Jason Hockman. "Semantically controlled adaptive equalisation in reduced dimensionality parameter space". In: *Applied Sciences* 6.4 (2016), p. 116.
- [19] Celso TN Suzuki et al. "Automated diagnosis of human intestinal parasites using optical microscopy images". In: *2013 IEEE 10th international symposium on biomedical imaging*. IEEE. 2013, pp. 460–463.
- [20] Dengyong Zhou et al. "Learning with local and global consistency". In: *Advances in neural information processing systems* 16 (2003).