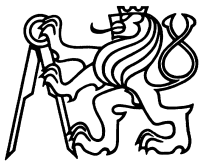
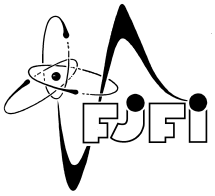


Czech Technical University in Prague
Faculty of Nuclear Sciences and Physical Engineering



Semisupervised Learning of Heterogenous Structured Data

Odhad modelu nekompletních heterogenních strukturovaných dat

Master's thesis

Author: **Bc. Michaela Mašková**

Supervisor: **doc. Ing. Václav Šmídl, Ph.D.**

Academic year: 2021/2022

Katedra: matematiky

Akademický rok: 2021/2022

ZADÁNÍ DIPLOMOVÉ PRÁCE

Student:	Bc. Michaela Mašková
Studijní program:	Aplikované matematicko-stochastické metody
Název práce (česky):	Odhad modelu nekompletních heterogenních strukturovaných dat
Název práce (anglicky):	Semisupervised learning of heterogenous structured data

Pokyny pro vypracování:

1) Seznamte se s problémem učení modelů heterogenních strukturovaných dat pro účely klasifikace. Zvláštní pozornost věnujte metodám založeným na hlubokých neuronových sítích a latentních prostorech.
2) Analyzujte latentní prostor modelů strukturovaných dat naučených diskriminativním způsobem (se znalostí správné klasifikace). Zvláštní pozornost věnujte metrice tohoto prostoru z hlediska shlukové analýzy.
3) Seznamte se s metodami založenými na vzdálenostech dvou bodů používanými pro shlukování dat. Vypracujte seznam vhodných metrik pro heterogenní stukturovaná data. Na vhodných datech proveďte srovnání blízkosti definované těmito metrikami s jejich známou třídou. Tj. zdali blízká data jsou ze stejné třídy.
4) Navrhněte metodu kombinující vlastnosti diskriminativního učení s učením na bázi shlukové analýzy. Na vhodných datech ověřte vhodnost metody pro reprezentaci dat například na problému detekce anomálií a studujte její vlastnosti.
5) Vyberte data z kolekce MIProblems, která obsahují více tříd, a proveďte na nich experimentální studii srovnávající kvalitu výsledků přiřazení do tříd na základě shlukování bez učitele se známými třídami. Proveďte citlivostní studii na parametry metody, například na počet komponent.

Doporučená literatura:

1) T. Pevny, P. Somol. Discriminative models for multi-instance problems with tree structure. In `Proceedings of the 2016 ACM Workshop on Artificial Intelligence and Security', ACM, 2016, 83–91.

2) J. Kim, J. Yoo, J. Lee, S. Hong. SetVAE: Learning hierarchical composition for generative modeling of set-structured data. arXiv preprint arXiv:2103.15619, 2021.

3) H. Zhang, S. Wang, X. Xu, T. WS Chow, Q. M. J. Wu. Tree2Vector: learning a vectorial representation for tree-structured data. IEEE transactions on neural networks and learning systems, 2018, 5304-5318.

Jméno a pracoviště vedoucího diplomové práce:

Doc. Ing. Václav Šmídl, Ph.D.
Ústav teorie informace a automatizace, Pod Vodárenskou věží 4, 182 00, Praha 8

Jméno a pracoviště konzultanta:

Datum zadání diplomové práce: 31.10.2021

Datum odevzdání diplomové práce: 2.5.2022

Doba platnosti zadání je dva roky od data zadání.

V Praze dne 01.11.2021

.....
garant oboru

.....
vedoucí katedry

.....
děkan

Acknowledgment:

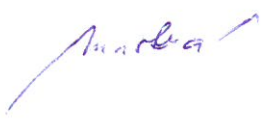
I would like to thank doc. Šmídl for his guidance during the last three years of my studies. I will always be grateful for the opportunity to work with him and the whole Anomaly Detection group at AIC. To my girlfriend, thank you for all the love, kindness and patience during the months that led to finishing this work. Thanks also go to my family for always believing in me and supporting me no matter what.

Author's declaration:

I declare that this Master's thesis is entirely my own work and I have listed all used sources in the bibliography.

Prague, May 2, 2022

Michaela Mašková



Název práce:

Odhad modelu nekompletních heterogenních strukturovaných dat

Autor: Bc. Michaela Mašková

Program: Aplikované matematicko-stochastické metody

Druh práce: Diplomová práce

Vedoucí práce: doc. Ing. Václav Šmídl, Ph.D., Pod vodárenskou věží 4, Praha 8

Abstrakt: Semi-supervised metody se snaží vyřešit problém dostupnosti výstupů pro data v aplikacích strojového učení. Modely jsou navrženy tak, aby se učily z dat se známou i neznámou třídou současně, a to s využitím metrického učení, generativního modelování a samoučení. Modely jsou aplikovány na standardní formáty dat, jako jsou vektory, obrázky nebo text. Tato práce rozšiřuje a zkoumá zavedené metody pro složitější datové struktury, jako jsou skupinová data a JSON soubory, pomocí nového balíčku HMill. Navržené modely jsou vyhodnoceny na třech různých datových sadách z domén víceinstančního učení, shluků bodů a kybernetické bezpečnosti. Vylepšené semi-supervised modely vykazují slibné zlepšení oproti standardním metodám učení s učitelem, ale generativní semi-supervised modely nedosahují dostatečně dobrých výsledků a vyžadují další prozkoumání.

Klíčová slova: shluková analýza, učení metrik, semi-supervised učení, generativní modely

Title:

Semisupervised Learning of Heterogenous Structured Data

Author: Bc. Michaela Mašková

Abstract: Semi-supervised learning tries to solve the problem of label availability for data in machine learning applications. Models are designed to learn from labeled and unlabeled data simultaneously, with the use of metric learning, generative modeling, and self-training, and applied to standard data formats such as vectors, images, or text. This thesis extends and explores established methods for more complex data structures such as group data and JSON files, with the help of a novel HMill framework. The models designed are evaluated on three distinct datasets from the domains of multi-instance learning, point clouds, and cybersecurity. Enhanced semi-supervised models show promising improvement over standard supervised methods, but generative semi-supervised models fail to achieve high enough performance and call for further research.

Key words: clustering, metric learning, semi-supervised learning, generative models

Contents

Introduction

Machine learning and AI are the go-to buzzwords in today's data-driven world. Researchers are constantly coming up with new ways to model data and the resources for model training are expanding. However, most use cases still rely on supervised learning and standard data structures. This thesis breaks the pattern and goes on a journey into less explored areas of semi-supervised learning and hierarchical structured data.

Semi-supervised learning is a means to close the gap between large amounts of data available and the scarcity of correct labels accompanying them. The first chapter gives an overview of semi-supervised learning, the assumptions, and experimental practices for evaluation. The list of models discussed contains self-supervised classifiers, semi-supervised generative models and methods based on clustering in the input space. There is a challenge with clustering non-numerical data, since a similarity metric might not be available. Metric learning is introduced as a method for learning the similarity function on data that is not composed of standard vectors of values in $\mathbb{R}$, such as images. This is followed by a section describing clustering methods, which can be used on the embedding learned by metric learning.

Most research in the domain of semi-supervised learning assumes that data come in well-known and well-researched formats: numbers, images, or text. This thesis focuses on hierarchical data, and the second chapter dives into the format of group data, multi-instance problems, and JSON file structure. A novel framework, HMill, is introduced as an automatic processing and learning library for heterogeneous hierarchical data.

Chapter three stands as the core of this thesis. The ideas of semi-supervised learning are connected with the data paradigm from the second chapter. The standard semi-supervised models are extended to work on group data and a generative model is designed for a specific dataset of JSON files. The models described are implemented in the Julia language to provide a comparative study of the algorithms on various datasets.

The experimental setup is outlined in chapter four. Three datasets are introduced, a multi-instance learning dataset, point cloud MNIST dataset, and dataset of JSON files from the security domain provided by Avast. The experiments are designed to evaluate both the classification accuracy and the quality of the learned embedding space in terms of clustering accuracy. Supervised and semi-supervised methods are compared for varying numbers of known labels provided for training.

The main objective is clear: Experimentally verify whether semi-supervised models work on more complex data structures and if they bring any advantages over standard supervised approaches. An accompanying objective is to explore the latent space of models and try to use methods to improve the discriminability of classes in the embedding.

The results show that semi-supervised approaches can boost the performance compared to simple classifiers, especially when only a few labeled samples are available. When triplet loss is added as a regularization to a classifier, the accuracy improves and the latent space becomes more separable. A self-supervised classifier and a self-supervised metric learning model both show

promising properties and, in most cases, were able to compete with the triplet-regularized classifier. The semi-supervised generative models fall short of expectations, with the exception of the semi-supervised M2 model on the multi-instance dataset. More work needs to be done in this area to design more powerful models that leverage the structure of the whole sample (group, JSON file) in training to truly unveil the potential of unlabeled data in training.

Chapter 1

Theoretical background

Machine learning and the use of AI has seen an unprecedented rise of popularity in the 2010's and continues to grow rapidly. Complex deep learning models are able to approximate incredibly complicated functions through large amounts of data and drive intelligent solutions in fields of anomaly detection (fraud, malware detection), image classification, object recognition, planning, robotics, and many other domains.

Most machine learning algorithms are supervised; that is, there are both datapoints and labels available as pairs (x, y) on which the model can be trained. Unfortunately, the cost of obtaining the correct data and labels can be significant in areas such as computer vision. This problem led to a whole new field of new data labeling and collection companies, and just in 2020, this market size was valued at 1.3 billion dollars and is expected to grow even further [1].

Sometimes, there might be a large amount of data available and only a limited budget for labeling the data. However, it would not make sense to throw out all the data that cannot be labeled, since even data with unknown labels can help the model in learning if we are able to develop an algorithm that use them. Such an approach is called semi-supervised learning, and the first chapter of this thesis is used to introduce that area of machine learning, describe state-of-the-art methods, and build the foundation for extending known algorithms to a new data paradigm.

1.1 Semi-supervised learning

Semi-supervised learning is one category of machine learning, lying between completely supervised and unsupervised learning. Nowadays, semi-supervised learning is gaining traction due to a simple challenge that arises in many machine learning tasks – the scarcity of correct labels. This might be due to the lack of labeled data in general or because of a significant cost of obtaining the labels, which frequently happens in areas like image recognition or speech annotation. Semi-supervised learning tries to solve this problem and design algorithms which are able to leverage information in both labeled and unlabeled data.

Problem definition

The assumption of semi-supervised learning is that the available dataset $\mathcal{D}$ is composed of two parts: (correctly) labeled data $\mathcal{D}_L = \{(\mathbf{x}_i, y_i)\}_{i=1}^{n}$ and unlabeled data $\mathcal{D}_U = \{\mathbf{x}_i\}_{i=n+1}^{m}$. The aim is to train a model, for example, a classifier, by using not only the data with known labels, but also

the data with unknown labels. The hope is that by utilizing the general assumptions discussed in the following paragraphs, the models will be able to learn from a larger data pool.

Assumptions of semisupervised learning

First, the assumptions of semi-supervised learning need to be discussed. The survey [2] provides a good overview of both the assumptions and evaluation methods.

For the classification task, it is always assumed that the data $\mathbf{x}$ come from an underlying distribution $p(\mathbf{x})$ which contains information about the posterior distribution $p(y \mid \mathbf{x})$, otherwise it would not be possible to construct a meaningful classifier. This assumption is closely followed by another condition that the data with known and unknown labels come from approximately the same data distributions: $p_L(\mathbf{x}) \approx p_U(\mathbf{x})$.

Other assumptions include

- **smoothness assumption:** when two samples $\mathbf{x}_1, \mathbf{x}_2$ are close in the input space, their corresponding labels y_1, y_2 should be the same,
- **low-density assumption:** the decision boundary between classes should not go through high-density areas of the input space,
- **manifold assumption**: samples on the same low-dimensional manifold should have the same label.

Good practices of semi-supervised learning

When it comes to supervised learning, there are established methods for the evaluation of novel algorithms and their comparison with state-of-the-art methods. The data set is divided into train, validation, and test subdatasets, where the model is trained only on the training data, and the validation set is used to spot signs of overfitting or finding the best combination of hyperparameters. Test data are not seen in training or validation phases and are only used in the end to calculate the real performance metrics of the model, such as its accuracy.

In semi-supervised learning, the training, validation and testing phase are not that straightforward. First, the data need to be partitioned into labeled and unlabeled sets. The ratio of data with known vs. unknown labels is, without question, an important parameter and should be varied to show the differences. For example, a researcher's aim might be to create a model that can use as few labels as possible and can work with only 1 % of labeled data. However, the performance of the model should be evaluated for higher percentage of known labels to show how the model behaves in different scenarios.

Second, the evaluation might be measured in two ways: either measure performance on the unlabeled data used for training, or measure performance on a disjoint test set. This work focuses on the latter, since the aim is to leverage the information in unlabeled data and boost performance of standard supervised methods.

The field of semi-supervised learning is rich and it is beyond the scope of this thesis to review all the methods available. Therefore, only a handful of algorithms are chosen with the intent of modifying them in the later chapters to allow for semi-supervised learning on more complicated data structures such as group data or JSON files.

1.1.1 Self-training

The first and probably best known methods for semi-supervised classification are the *self-training* methods. The first usage dates back to 1995, when it was used for the disambiguation of the meaning of words based on context [3]. Since then, many techniques have been developed, [4] provides a comparative study of self-training algorithms.

The general procedure is to iterate between training a classifier and obtaining labels for currently unclassified points. The name *self-training* is used because the model used for training and classification of unknown data is the same.

One iteration of the algorithm has two steps: training and pseudo-labeling. The first step is trivial and does not need further discussion. However, there are multiple design choices to be made for the pseudo-labeling procedure. Generally, only unlabeled data classified by the trained classifier should be pseudo-labeled and sent through to the next iteration. The question is: How confident does the prediction need to be and how much does it influence the outcome? A classifier usually returns a probability distribution of the target classes, leaving the researcher to choose a confidence threshold such as 95 %.

Note that the higher the confidence threshold, the less helpful the new data would be for further training of the classifier, since the classification loss stays low. Consequently, the classifier needs to be retrained from scratch in every iteration of this particular semi-supervised method. This method might also significantly skew the distribution of the number of samples from each class in the training data. For example, recognizing digit 1 in the MNIST dataset is probably easier than distinguishing digit 6, resulting in the classifier being more likely to label and add 1's rather than 6'. In the next iteration, we might have hundreds of new digits 1, but only a handful of 6's and other digits. This problem can be resolved by either sampling a balanced minibatch when training, or balancing the classes in the pseudo-labeling step.

The self-training algorithm can be modified to use multiple classifiers, such an approach is named *co-training*. In this setting, each classifier is used to pseudo-label data which are used as an input to a different classifier in the ensemble.

1.1.2 Methods using generative models

Kingma et. al. [5] came up with the idea to use generative models for semi-supervised learning. The proposed model extends the ideas of a variational autoencoder [6] to include a classification network. Two models, M1 and M2, are presented. It is also possible to use a combination of both (M1+M2).

M1 model

The first model presented is a latent-feature discriminative model. The idea is to train a variational autoencoder as an unsupervised model on both labeled and unlabeled data, leveraging the power of the latent space created in the process. Samples of the same class are expected to lie close to each other in the latent space, since they share the same underlying representation and are therefore projected into a dense area in the latent space.

The variational autoencoder [6] is a probabilistic extension of classical autoencoders. Suppose that a vector $\mathbf{x}$ comes from some unknown distribution $p(\mathbf{x})$. It might be an incredibly challenging task to learn $p(\mathbf{x})$ if it is not a simple distribution such as a Gaussian or a mixture of Gaussians. The

idea behind VAEs is to use the Bayes rule and write the distribution as

$$p(\mathbf{x}) = \int p(\mathbf{x}|\mathbf{z})p(\mathbf{z})\mathrm{d}\mathbf{z}, \tag{1.1}$$

where the latent variable $\mathbf{z}$ is supposed to carry information about $\mathbf{x}$ and $p(\mathbf{z})$ is supposed to be a simpler distribution than $p(\mathbf{x})$. Usually, the latent variable distribution is chosen to be isotropic Gaussian $\mathcal{N}(\mathbf{0}, \mathbb{I})$. The task is to learn two mappings: encoder $p(\mathbf{z}|\mathbf{x}) : \mathcal{X} \mapsto \mathcal{Z}$ and decoder $p(\mathbf{x}|\mathbf{z}) : \mathcal{Z} \mapsto \mathcal{X}$. As this is usually an intractable task, the distribution $p(\mathbf{z}|\mathbf{x})$ is approximated by variational inference as $q(\mathbf{z}|\mathbf{x}) = \mathcal{N}\left(\mu(\mathbf{x}), \Sigma(\mathbf{x})\right)$ and is forced with the Kullback-Leibler divergence to encode $\mathbf{x}$ values to $\mathbf{z} \sim p(\mathbf{z}) = \mathcal{N}(\mathbf{0}, \mathbb{I})$.

The decoding mapping $p(\mathbf{x}|\mathbf{z})$ can be another Gaussian, usually a distribution in the form $p(\mathbf{x}|\mathbf{z}) = \mathcal{N}(f(\mathbf{z}), \sigma)$. Learning the mapping function $f(\mathbf{z})$ is done via ELBO optimization. ELBO stands for evidence lower bound and it is indeed the lower bound of the likelihood $p(\mathbf{x})$. Variational inference is needed, since direct optimization is intractable. Finally, the VAE model is trained by minimizing the negative ELBO in the form of

$$\mathcal{L} = \mathbb{E}_{\varepsilon \sim \mathcal{N}(\mathbf{0}, \mathbb{I})} \left[\log p\left(\mathbf{x} \mid \mathbf{z} = \mu(\mathbf{x}) + \varepsilon \cdot \Sigma^{1/2}(\mathbf{x}) \right) - \mathcal{D}_{KL}\left(q(\mathbf{z} \mid \mathbf{x}) \,||\, p(\mathbf{z}) \right) \right], \tag{1.2}$$

making use of the reparametrization trick to ensure the equation is differentiable.

Variational autoencoders are powerful models because they make it possible to sample from the distribution $p(\mathbf{x})$ by first sampling $\mathbf{z} \sim p(\mathbf{z}) = \mathcal{N}(\mathbf{0}, \mathbb{I})$ and then simply using the trained transformation function to get $\mathbf{x} = f(\mathbf{z})$.

When VAE is trained, all datapoints in the training data can be mapped to the latent space as samples from the approximate posterior $q(\mathbf{z} \mid \mathbf{x})$ and used as input features for a standard supervised classifier. The learned features are expected to be more clearly separable and also can significantly reduce the dimension of the problem, since the dimension of a VAE latent space is usually smaller than the dimension of the input space.

M2 model

The second model is a generative semi-supervised model, designed to use the available labels during training of the generative part of the model. The generative process is described with distributions

$$p(y) = \mathrm{Cat}(y \mid \boldsymbol{\pi}); \quad p(\mathbf{z}) = \mathcal{N}(\mathbf{z} \mid \mathbf{0}, \mathbf{I}); \quad p_\theta(\mathbf{x} \mid y, \mathbf{z}) = f(\mathbf{x}; y, \mathbf{z}, \boldsymbol{\theta}). \tag{1.3}$$

The label distribution is the multinomial distribution parameterized with $\boldsymbol{\pi}$. If y is not available, it is treated as a latent variable. The variables $\mathbf{z}$ are latent variables of the samples. The function f is a likelihood function of a chosen probability distribution, such as Gaussian as before, or Bernoulli.

Simply put, the M2 model is composed of a VAE (M1 model) and a classifier, with the change that a label y is also used as input for the generative model. The inferred posterior distribution $p_\theta(y \mid \mathbf{x})$ serves as the classification model.

M1+M2

The combination of M1 and M2 models is obtained by first learning a latent representation $\mathbf{z}_1$ using the generative model M1, then learning the M2 model using $\mathbf{z}_1$ as an input feature instead of the input vector $\mathbf{x}$.

The resulting joint distribution becomes

$$p_\theta\left(\mathbf{x}, y, \mathbf{z}_1, \mathbf{z}_2\right) = p(y)p\left(\mathbf{z}_2\right)p_\theta\left(\mathbf{z}_1 \mid y, \mathbf{z}_2\right)p_\theta\left(\mathbf{x} \mid \mathbf{z}_1\right),$$

the prior distributions $p(y)$ and $p\left(\mathbf{z}_2\right)$ do not change, $p_\theta\left(\mathbf{z}_1 \mid y, \mathbf{z}_2\right)$ and $p_\theta\left(\mathbf{x} \mid \mathbf{z}_1\right)$ are parametrized with neural networks.

Variational inference

Calculating the exact posterior distribution is intractable in both models presented, therefore variational inference is used to find approximate distributions q. The inference networks chosen are Gaussian distributions for both M1 and M2 models. For the M2 model, it is assumed that the approximate distribution is factorized as $q_\phi(\mathbf{z}, y \mid \mathbf{x}) = q_\phi(\mathbf{z} \mid \mathbf{x}, y)q_\phi(y \mid \mathbf{x})$. The approximates are parametrized with $\boldsymbol{\mu}, \boldsymbol{\sigma}, \boldsymbol{\pi}$ as neural networks

$$\begin{aligned}
&\text{M1: } q_\phi(\mathbf{z} \mid \mathbf{x}) = \mathcal{N}\left(\mathbf{z} \mid \boldsymbol{\mu}_\phi(\mathbf{x}), \operatorname{diag}\left(\boldsymbol{\sigma}^2_\phi(\mathbf{x})\right)\right), \\
&\text{M2: } q_\phi(\mathbf{z} \mid y, \mathbf{x}) = \mathcal{N}\left(\mathbf{z} \mid \boldsymbol{\mu}_\phi(y, \mathbf{x}), \operatorname{diag}\left(\boldsymbol{\sigma}^2_\phi(\mathbf{x})\right)\right); \quad q_\phi(y \mid \mathbf{x}) = \operatorname{Cat}\left(y \mid \boldsymbol{\pi}_\phi(\mathbf{x})\right).
\end{aligned}$$

The objective of the M1 model is to maximize the likelihood function, which can be approximated with the evidence lower bound as mentioned before. For M1, the loss is simply a negative ELBO in the form of

$$\log p_\theta(\mathbf{x}) \geq \mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})}\left[\log p_\theta(\mathbf{x} \mid \mathbf{z})\right] - KL\left[q_\phi(\mathbf{z} \mid \mathbf{x}) \mid\mid p_\theta(\mathbf{z})\right] = -\mathcal{J}(\mathbf{x}). \tag{1.4}$$

Since no labels are needed, both labeled and unlabeled samples are used interchangeably during training.

The objective of the M2 model needs to be divided into two parts: for cases where the label is present and for those where it is not. For the observed label, the variational lower bound extends (1.4) to

$$\begin{aligned}
\log p_\theta(\mathbf{x}, y) &\geq \mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x},y)}\left[\log p_\theta(\mathbf{x} \mid y, \mathbf{z}) + \log p_\theta(y) + \log p(\mathbf{z}) - \log q_\phi(\mathbf{z} \mid \mathbf{x}, y)\right] = \\
&= \mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x},y)}\left[\log p_\theta(\mathbf{x} \mid y, \mathbf{z})\right] + \log p_\theta(y) - \mathcal{D}_{KL}\left[q_\phi(\mathbf{z} \mid \mathbf{x}, y) \mid\mid p(\mathbf{z})\right] = \\
&= -\mathcal{L}(\mathbf{x}, y)
\end{aligned} \tag{1.5}$$

and for a sample without a label becomes

$$\begin{aligned}
\log p_\theta(\mathbf{x}) &\geq \mathbb{E}_{q_\phi(y,\mathbf{z}|\mathbf{x})}\left[\log p_\theta(\mathbf{x} \mid y, \mathbf{z}) + \log p_\theta(y) + \log p(\mathbf{z}) - \log q_\phi(y, \mathbf{z} \mid \mathbf{x})\right] \\
&= \mathbb{E}_{q_\phi(y,\mathbf{z}|\mathbf{x})}\left[\log p_\theta(\mathbf{x} \mid y, \mathbf{z})\right] - \mathcal{D}_{KL}\left[q_\theta(\mathbf{z} \mid \mathbf{x}, y) \mid\mid p(\mathbf{z})\right] - \mathcal{D}_{KL}\left[q_\phi(y \mid \mathbf{x}) \mid\mid p_\theta(y)\right] = \\
&= \sum_y q_\phi(y \mid \mathbf{x})(-\mathcal{L}(\mathbf{x}, y)) + \mathcal{H}\left(q_\phi(y \mid \mathbf{x})\right) = -\mathcal{U}(\mathbf{x}).
\end{aligned} \tag{1.6}$$

Both bounds (1.5) and (1.6) need to be combined to calculate the loss over the whole dataset (or a minibatch)

$$\mathcal{J} = \sum_{(\mathbf{x},y)\sim\tilde{p}_l} \mathcal{L}(\mathbf{x}, y) + \sum_{\mathbf{x}\sim\tilde{p}_u} \mathcal{U}(\mathbf{x}). \tag{1.7}$$

The distribution $q_\phi(y \mid \mathbf{x})$ acts as a discriminative model and is used during test time to predict data labels. However, note that the dicriminator is trained only on unlabeled samples, which is not

a desirable property. To fix it and force the classifier to be trained on labeled samples as well, the overall loss is extended with a supervised cross-entropy loss to

$$\mathcal{J}^{\alpha} = \mathcal{J} + \alpha \cdot \mathbb{E}_{\widetilde{p}_l(\mathbf{x},y)} \left[-\log q_{\phi}(y \mid \mathbf{x}) \right], \tag{1.8}$$

where α is a hyperparameter, [5] use $\alpha = 0.1 \cdot N$ in their experiments.

1.1.3 Methods based on clustering

It is possible to use unsupervised methods, such as clustering, prior to supervised learning and combine these two approaches in a version of a semi-supervised algorithm. Let us go back to one of the assumptions of semi-supervised learning, the smoothness assumption. It states that when two samples are close in the input space, their corresponding labels should be the same. If this is true, it is possible to use, for example, a clustering algorithm in the input space to obtain labels of unlabeled samples and then use the full dataset to train a supervised model.

An illustration of this approach can be seen in Figure 1.1. The plot of the left shows two clusters of labeled data, blue and green, and some unlabeled data. The labels of unlabeled data can be obtained by using a simple clustering algorithm. The right plot shows the difference between a classification boundary learned in a supervised manner from only the labeled data, and the improved boundary when all data are used in a semi-supervised approach.

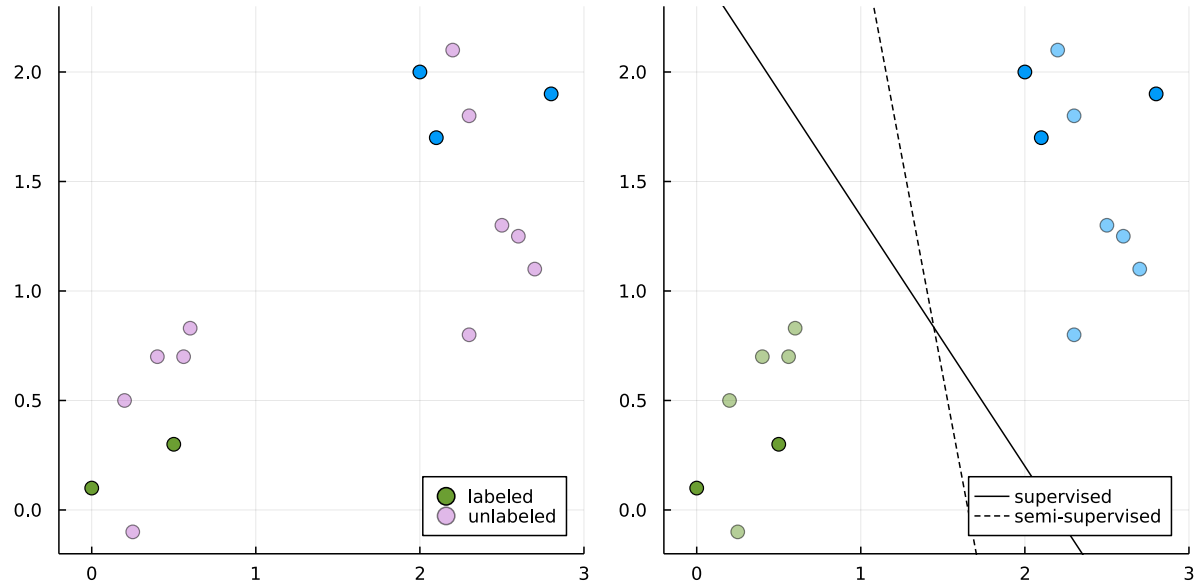


Figure 1.1: Comparison of supervised and semi-supervised classification boundary using clustering in the input space to infer unknown labels.

Unfortunately, it is not always possible to measure distances of datapoints in the input space, for example, for image data. The next section discusses metric learning, which can help in learning an embedding with a similarity metric, that can be later used to cluster data and design a cluster-based semi-supervised (self-supervised) algorithms.

1.2 Metric learning

As humans, we are quite good at understanding similarity. We can look at two images of kids playing on a beach and conclude that the images look similar. When given an image of kids playing in the woods, we can say that the picture is similar to the previous two in depicting playing kids but different in the surroundings.

Computer algorithms have a much harder time on this task, since it can be demanding to quantify similarity, especially when the similarity is conditioned on some context.

When dealing with tabular data, there are numerous metrics to use as a similarity metric. For example, clustering in a 2D space can be done using Euclidean distance to calculate the distance matrix. On the other hand, measuring the distance between other types of data, such as images, audio sequences, or sets of points, can be tricky. Metric learning serves as a method for learning a distance function, which might later serve as a similarity metric for algorithms such as k-nearest neighbors.

Metric learning is used to learn a distance function that follows two simple principles:

1. similar objects should be close,
2. dissimilar objects should be far away from each other.

Say, a dataset is composed of labeled images, where labels can be one-dimensional (dog vs. cat) or multi-dimensional (kids, beach vs. kids, woods). The task is to learn a distance function that would put images of cats close to each other, images of dogs close to each other, and these two groups farther from each other.

The metric learning domain is mainly focused on image data, but the ideas are applicable in other domains as well. Typically, metric learning uses a combination of embedding learning and optimizing classification loss. One of the first proposed metric is the contrastive loss [7] acting on pairs of data. The aim is to optimize a function f parameterized as a neural network to learn a distance metric $d(\mathbf{x}_1, \mathbf{x}_2) = ||f(\mathbf{x}_1) - f(\mathbf{x}_2)||$, preferably in a lower-dimensional space created by f. It is assumed that the input point is $(\mathbf{x}_1, \mathbf{x}_2, y)$, where y is the label of the pair, $y = 0$ when $\mathbf{x}_1, \mathbf{x}_2$ are similar and $y = 1$ otherwise. The contrastive loss is then defined as

$$\mathcal{L}_{contrastive}(y, \mathbf{x}_1, \mathbf{x}_2) = \frac{1}{2}(1-y)||f(\mathbf{x}_1) - f(\mathbf{x}_2)||^2 + \frac{1}{2}y\big(\max\{0, m - ||f(\mathbf{x}_1) - f(\mathbf{x}_2)||\}\big), \quad (1.9)$$

where $m > 0$ is a margin.

What makes contrastive loss and other losses in the metric learning paradigm powerful is the use of data augmentation. In an extreme case, no labels are needed, and the augmentations might be sufficient to learn a similarity metric. The creation of pairs (or triplets in the case of the triplet loss) is simple. For pairs in the same class, take an image and its augmentation. For pairs with different labels, take two different images from the training data, possibly augmenting one or both. The augmentations can be horizontal or vertical flip, change of color (colored, grayscale, etc.), rotation, crop, and others.

During the years, many other losses have been proposed, and two of them will be described in more detail: triplet loss and angular margin loss in the ArcFace architecture.

1.2.1 Triplet Loss

The aim of a *triplet network* [8] is to learn a useful representation of similarity using triplets of data. Given a dataset $\mathcal{D} = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$ and a chosen rough similarity measure $r(\mathbf{x}, \mathbf{x}')$, the task is

to learn a similarity function $S(\mathbf{x}, \mathbf{x}')$. The objective is optimized using data in the form of *triplets* $\mathbf{x}, \mathbf{x}_1, \mathbf{x}_2$ where $r(\mathbf{x}, \mathbf{x}_1) > r(\mathbf{x}, \mathbf{x}_2)$. The rough metric can be simply an indicator function of class labels, therefore the triplets can be denoted $\mathbf{x}, \mathbf{x}^+, \mathbf{x}^-$, and $r(\mathbf{x}, \mathbf{x}^+) = 1$ when $\mathbf{x}, \mathbf{x}^+$ have the same labels, and $r(\mathbf{x}, \mathbf{x}^-) = 0$ if $\mathbf{x}, \mathbf{x}^-$ have different labels.

The focus is on finding a function $f(\mathbf{x})$ such that $S(\mathbf{x}, \mathbf{x}') = ||f(\mathbf{x}) - f(\mathbf{x}')||_2$. The function $f(\mathbf{x})$ is represented as a deep neural network with trainable parameters to optimize the triplet loss defined as

$$\text{loss}(d_+, d_-) = ||(d_+, d_- - 1)||_2^2 = \text{const.} \cdot d_+^2, \tag{1.10}$$

where

$$\begin{aligned} d_+ &= \frac{\mathrm{e}^{||f(\mathbf{x})-f(\mathbf{x}^+)||_2}}{\mathrm{e}^{||f(\mathbf{x})-f(\mathbf{x}^+)||_2} + \mathrm{e}^{||f(\mathbf{x})-f(\mathbf{x}^-)||_2}} \\ d_- &= \frac{\mathrm{e}^{||f(\mathbf{x})-f(\mathbf{x}^-)||_2}}{\mathrm{e}^{||f(\mathbf{x})-f(\mathbf{x}^+)||_2} + \mathrm{e}^{||f(\mathbf{x})-f(\mathbf{x}^-)||_2}}. \end{aligned}$$

The training depends on the choice of triplets during training. Problems can arise especially when the amount of data becomes large. The number of triplets rises cubically, and it is important to choose triplets which carry meaningful information to learn an informative embedding. That is why the choice – *mining* – of triplets is modified in subsequent literature to achieve better convergence and learn embeddings of higher quality. For a chosen *anchor* point $\mathbf{x}$, [9] propose to find hard negatives, samples which look similar to $\mathbf{x}$ but are of different class, and hard positives, samples from the same class as the anchor but looking differently from $\mathbf{x}$.

Given a distance D (for example Euclidean, Cosine, etc.), the metric of interest is $D_{i,j} = D\left(f(\mathbf{x}_i), f(\mathbf{x}_j)\right)$. In every iteration, a batch is sampled randomly, first, a random number of C classes is chosen and N samples from each class $c \in C$ are drawn and added to the batch. The loss, called *Batch Hard* loss, becomes

$$\begin{aligned} \mathcal{L}(X) = \sum_{c=1}^{C} \sum_{i=1}^{N} \Big[m + \max_{k=1,\dots,N} D\big(f(\mathbf{x}_i^c), f(\mathbf{x}_k^c)\big) \\ - \min_{\substack{j=1,\dots,C \\ n=1,\dots,N \\ j \neq c}} D\big(f(\mathbf{x}_i^c), f(\mathbf{x}_n^j)\big) \Big]_+ . \end{aligned} \tag{1.11}$$

1.2.2 ArcFace

One task in the image recognition field is face representation and recognition for classification. The challenge can be explained with a simple example: many current smartphones come with a facial recognition software. Its aim is to quickly learn the representation of the owners face and be able to instantly classify whether the face the phone is seeing is the owner's face or not. The task is similar to few-shot learning, since the model is trying to learn a general face representation, which can then be reused in any phone to add a new face and be able to recognize it quickly.

The goal of these models is to use convolutional neural networks and train them to extract features that would put different images of the same face close to each other in an embedding space. At the same time, any image of a different person should be projected as far as possible from the target feature in this learned space. Furthermore, the ambition is to learn such a rich feature space that yet unseen images would lie in a previously empty space and satisfy the same

requirements that images during training must fulfill: images of the same person are close, while images of different people are further, and these clusters should be separable.

The problem is sometimes called an open-set recognition problem, where we want to be able to detect a new class present in the test data. To make detection possible, the class needs to be projected into a dense cluster in the feature space.

As any other classification problem, the face recognition problem can be solved using a simple cross-entropy loss to train a classification network. However, the dimension of the output of such a classifier must be the same as the number of classes. When the number of classes increases, the complexity of the network grows as well. Triplet loss is able to project classes into embeddings of a much smaller dimension, but has difficulty with a large number of classes as well, since the increase in the number of classes results in a combinatorial increase of the number of triplets needed for training.

Lately, researchers have focused on an approach using an angular margin, trying to maximize the angle between different class projections. Sphereface [10] was the first model to use the angular margin, followed by CosFace [11]. ArcFace [12] follows the main ideas and modifies them to efficiently beat the previous models in accuracy.

The softmax loss is defined as

$$\mathcal{L}_{softmax} = -\frac{1}{N}\sum_{i=1}^{N}\log\frac{e^{W_{y_i}^T x_i + b_{y_i}}}{\sum_{j=1}^{n} e^{W_j^T x_i + b_j}}, \tag{1.12}$$

where the authors fix the bias to zero and use normalized weights W and features x to transform the *logit* to $W_j^T x_i = ||W_j|| \cdot ||x_i|| \cdot \cos\theta_j$. For image data, features x are outputs of a CNN architecture. Consequently, θ_j is the angle between the weight and the feature. Weights are normalized to 1, features are normalized to s, where s is a hyperparameter. This setting makes the embedding distributed on a hypersphere of radius s. The last step is to add the additive angular margin penalty m, which should force the embedding to make samples of the same class more compact and increase the discrepancy between different classes.

When everything is put together, the ArcFace loss takes the form of

$$\mathcal{L}_{AF} = -\frac{1}{N}\sum_{i=1}^{N}\log\frac{e^{s\left(\cos\left(\theta_{y_i}+m\right)\right)}}{e^{s\left(\cos\left(\theta_{y_i}+m\right)\right)} + \sum_{j=1, j\neq y_i}^{n} e^{s\cos\theta_j}}. \tag{1.13}$$

The algorithm of the ArcFace architecture follows roughly these steps:

1. Get feature x from the input sample using a feature-extraction model.
2. Normalize x to $\frac{x}{||x||}$.
3. Normalize W to $\frac{W}{||W||}$.
4. Calculate the logit as $\cos\theta_j = \frac{W_j^T x_i}{||W|| \cdot ||x||}$.
5. Get angle $\theta_j = \arccos(\cos\theta_j)$.
6. Add the additive angular margin penalty to transform the logit to $\cos(\theta_{y_i} + m)$.
7. Scale to $s \cdot \cos(\theta_{y_i} + m)$.
8. Apply softmax function and use the standard cross-entropy loss in the form (1.13).

There is a possibility to add even more regularization to the model. For example, ArcFace can be used in combination with the triplet loss. The triplet loss objective would then be optimized on the output of the feature-extraction algorithm to add a next layer of discriminability of classes.

1.3 Clustering

Metric learning introduced in the previous sections allows us to learn a discriminative embeding space. When the space is found and created, it is time to use clustering methods to detect classes in the embedding.

Clustering algorithms try to solve one of the most prominent questions in unsupervised learning – finding groups of data based on the similarity of datapoints. Such groups are called clusters. There is no standard definition of a cluster, but in short, the goal of a clustering algorithm is to find distinct clusters of data based on two main ideas:

1. instances in the same cluster must be similar as much as possible,
2. instances in different clusters must be different as much as possible.

The similarity and dissimilarity of instances has to be defined as a measurement, for example, a distance between instances. Such a measure can be, for example, a Euclidean distance. For now, let us denote the distance metric chosen as $d(x) : \mathbb{R}^n \mapsto \mathbb{R}$. The distance function should satisfy the properties of a metric:

- positivity: $d(x, y) \geq 0, \quad \forall x, y \in \mathcal{X}$,
- symmetry: $d(x, y) = d(y, x), \quad \forall x, y \in \mathcal{X}$,
- triangular inequality: $d(x, y) \leq d(x, z) + d(z, y), \quad \forall x, y, z \in \mathcal{X}$.

A clustering task, or cluster analysis, can be divided into four basic steps [13]:

1. **Feature selection or extraction:** Data needs to be described by (preferably) numerical features, and there should exist a metric in the feature space. Finding a feature space for tabular data poses no problems, on the other hand, feature space of images, for example, needs to be learned with feature-extraction algorithms or models such as CNNs.
2. **Clustering algorithm design or selection:** Usually, first, we need to find or define a similarity measure in the feature space. Then we can choose the best clustering algorithm for our task from a variety of methods.
3. **Cluster validation:** Every clustering algorithm returns assignments of points to clusters, but the quality of given clustering needs to be evaluated. When assessing the quality, we can look at evaluation methods that will be described in further sections.
4. **Results interpretation:** The end goal would be to interpret the results, for example, figuring out what do the created groups of data have in common and how can this information be used further.

1.3.1 Clustering methods

Clustering algorithms fall into categories based on the methods they use to create clusters. Three main methods will be described: partitional, hierarchical, and distribution-based.

Partitional clustering

Partitional clustering methods directly divide data into a predefined number of groups. One of the best-known and easiest clustering algorithms is the k-means algorithm with its modification to k-medoids.

K-means algorithm [14], as the name suggests, finds k clusters of data. The simplest version of the algorithm starts by defining k random centroids $c_i, i \in 1, \ldots, k$ in the feature space. Assume that each centroid c_i has its corresponding label y_i. Two steps are performed in every iteration of the algorithm:

1. Assign every point to its nearest centroid c_i based on the distance $d(c_i, x^j), j \in 1, \ldots, n$ where n is the number of datapoints. Every point is assigned a label $y^j \sim \arg\min_{i \in 1,\ldots,k} d(c_i, x^j)$.
2. Based on the assignments from step 1, calculate a new centroid c_i as the center of mass of the cluster for each i.

These two steps are repeated until the algorithm converges and there is no change for each cluster.

Although k-means is easy to implement and its computational complexity of $O(nkd)$ is feasible even for larger datasets, it also comes with major disadvantages. First and foremost, the final clustering heavily depends on the initialization of the centroids. Generally, the solution to this problem is to run the algorithm multiple times and choose either the best partitioning based on a criterion (such as silhouettes values) or find a better centroid initialization values. The algorithm can also be run for multiple subsets of the dataset. Another problem is that even though the convergence of the algorithm is guaranteed , there is no guarantee of convergence to a global minimum. K-means is also sensitive to outliers and noise. Outliers can be either forced to a cluster and distort its shape, or create a cluster containing just a single datapoint and therefore may be forced to join relatively distinct clusters in other areas of the feature space.

A variation of the k-means algorithm is called **k-medoids** [15]. The algorithm starts with an arbitrary selection of k points from the data as medoids. Then, these four steps are repeated until no change in the medoids is detected:

1. Assign each datapoint to its nearest medoid.
2. Randomly select non-medoid point $\hat{x}$.
3. Calculate the total cost S of swapping the initial medoid with $\hat{x}$.
4. If $S < 0$, swap the initial medoid with $\hat{x}$.

K-medoids algorithm is a bit more costly than k-means for smaller datasets, but provides more resistance to outliers and noise.

Hierarchical clustering

Hierarchical clustering methods can be further divided into two categories: agglomerative and divisive methods. Both usually depict their results as a dendrogram or binary tree. The height of the connections of points or clusters usually represents a cost or distance. Hierarchical methods have the advantage that they do not need a predefined number of clusters to begin with. Both build a dendrogram and the clustering itself is created by cutting the dendrogram at a certain level. Therefore, it is possible to control the cost of cutting the dendrogram at any level.

Agglomerative clustering starts with representing each point in the dataset as a single cluster. What follows is a merging algorithm that iteratively groups points to smaller clusters and small clusters to bigger ones, until all points are assigned to one large data cluster. At each step, a cost function value is calculated and its increase after every merge operation determines the height of the dendrogram. In the end, the dendrogram can be cut either after the cost reaches a certain threshold, or at a point where the number of clusters reaches k. **Divisive clustering** proceeds in the

opposite direction – first assigns all points to one single cluster and then splits it until every point becomes its own cluster. Divisive clustering is computationally expensive, therefore agglomerative clustering will be used in this work.

The algorithm is simple. In the initialization step, every datapoint gets assigned to its own cluster. A proximity matrix is calculated for the n clusters, where the (i, j)-th element of the matrix represents the distance $d(c_i, c_j)$. Therefore, the proximity matrix is square and symmetric. Two steps are repeated until all points are merged into one cluster:

1. Calculate the proximity matrix for the current number of clusters.
2. Search the minimal distance $d(c_i, c_j) = \min_{1 \leq m,l<n,m \neq l} d(c_m, c_l)$ and merge clusters c_i, c_j that are closest to each other into one.

The properties of the agglomerative clustering algorithm depends on the distance used. For some distances, special names are used. The most common options are

- **single:** minimum distance between any cluster members,
- **complete:** maximum distance between cluster members,
- **average:** mean distance between cluster members,
- **ward:** the distance is the increase in the average squared distance of a point to its cluster centroid after merging the two clusters.

Distribution based clustering

Distribution based clustering algorithms, DBCAs for short, can help in the case of previous methods failing due to various problems. One common challenge in clustering is the effect of outliers and noise on the creation of clusters. Since every point needs to be assigned to a cluster, one outlier can skew the centroids in k-means and k-medoids algorithms and create a significant cost jump in a hierarchical clustering dendrogram.

DBCAs can overcome some of the challenges simply thanks to the properties of the probability density function [16]. Let us start with an example of using Gaussian mixtures for clustering. Using the EM algorithm, we can fit a predefined number of Gaussian distributions k to the data and evaluate the probability density value of each datapoint belonging to every Gaussian component. Such an approach allows us to pick a threshold that a datapoint needs to surpass to be clustered, otherwise it can be labeled as noise or an outlier.

Of course, using Gaussian Mixtures for clustering is not widespread and many more complex algorithms have been developed, DBSCAN [17] being probably the most prominent of them all.

DBCA can be divided into categories based on the way the density is defined and calculated, how sensitive they are to a parameter change, or by the way of computation (for example, whether the calculation of probabilities can be parallelized). The density definition falls into three categories

1. **point based:** density is calculated based on the number of points in a neighborhood,
2. **grid based:** density is calculated on a grid and dense regions are connected to a cluster,
3. **probability based:** density is calculated as a true probability density function.

1.3.2 Evaluation metrics

Every algorithm needs to be provided with evaluation metrics to measure its performance. Most metrics are used to measure the agreement between two clustering partitions, or clustering and

true labels, if available. These evaluation methods are further divided into pair-counting based measures and information-theory based measures. There exists another metric called silhouettes, which stands out by measuring not a comparison of two clusterings, but the quality of one clustering's assignments based on distances between points in the separate clusters.

In general, clustering metrics should follow some desirable properties, namely

- metric property: the metric should be a true metric and satisfy the properties of positivity, symmetry, and triangle inequality,
- normalization: metric values should lie in a bounded interval, for example $[0, 1]$,
- constant baseline: measure value between two independent clusterings should be constant.

First, let us introduce the notation and important terms used. The dataset is a collection of N points, $\mathcal{D} = \{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$. The comparison is made between two clusterings (partitions) of the data $P = \{P_1, \ldots, P_k\}$, $Q = \{Q_1, \ldots, Q_l\}$. The clustering P consists of k clusters, Q is composed of l clusters. It is possible for Q to be the true labels for the data. Naturally, a confusion matrix can be defined for the two clusterings P, Q as

$P \backslash Q$	Q_1	Q_2	$\ldots$	Q_l	Sums
P_1	n_{11}	n_{12}	$\ldots$	n_{1l}	a_1
P_2	n_{21}	n_{22}	$\ldots$	n_{2l}	a_2
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$
P_k	n_{k1}	n_{k2}	$\ldots$	n_{kl}	a_k
Sums	b_1	b_2	$\ldots$	b_l	$\sum_{ij} n_{ij} = N$

(1.14)

Pair counting based measures

Pair counting measures operate on counts derived from the confusion matrix as defined in (1.14). **Cross-tabulation** is a method in which the whole confusion matrix is analyzed. It is possible to look at the individual entries and see if there is agreement between the two clusterings, similar to examining the diagonal of a standard $n \times n$ confusion matrix.

RandIndex [18] operates on four defined counts of data. Given a set of N datapoints and two clusterings P, Q, define

- N_{11}: number of pairs of datapoints in S that are in the same subset in P and in the same subset in Q,
- N_{00}: number of pairs of datapoints in S that are in different subsets in P and different subsets in Q,
- N_{01}: number of pairs of datapoints in S that are in the same subset in P and in different subsets in Q,
- N_{10}: number of pairs of datapoints in S that are in different subsets in P and in the same subset in Q.

Then, the Rand index measure RI is defined as

$$\mathrm{RI}(P, Q) = \frac{N_{00} + N_{11}}{N_{00} + N_{01} + N_{10} + N_{11}} = \frac{N_{00} + N_{11}}{\binom{N}{2}}. \tag{1.15}$$

The value of RI lies in the range $[0, 1]$, however, in practice the value lies in the range $[0.5, 1]$. Therefore, an adjustment to the original Rand Index has been made, called Adjusted Rand Index

and defined as

$$\mathrm{ARI}(P,Q) = \frac{2\left(N_{00}N_{11} - N_{01}N_{10}\right)}{\left(N_{00}+N_{01}\right)\left(N_{01}+N_{11}\right)+\left(N_{00}+N_{10}\right)\left(N_{10}+N_{11}\right)}. \tag{1.16}$$

Information theoretic based measures

Information theory based measures [19] are derived from the concepts of information theory. Entropy, joint entropy, conditional entropy and mutual information are defined naturally as

$$H(P) = -\sum_{i=1}^{k}\frac{a_i}{N}\log\frac{a_i}{N}$$

$$H(P,Q) = -\sum_{i=1}^{k}\sum_{j=1}^{l}\frac{n_{ij}}{N}\log\frac{n_{ij}}{N}$$

$$H(P \mid Q) = -\sum_{i=1}^{k}\sum_{j=1}^{l}\frac{n_{ij}}{N}\log\frac{n_{ij}/N}{b_j/N}$$

$$I(P,Q) = \sum_{i=1}^{k}\sum_{j=1}^{l}\frac{n_{ij}}{N}\log\frac{n_{ij}/N}{a_i b_j/N^2}$$

Mutual information measures the shared information in P, Q. Since it is upper-bounded, it can be normalized to the interval $[0,1]$ such that $\mathrm{MI} = 1$ for identical clusterings and 0 for independent clusterings.

Another measure derived from information theory is called **V-measure**. It is defined as a harmonic mean of homogeneity h and completeness c, combining the two in the same way as precision and recall are sometimes combined in the F-measure. Say datapoints come from k classes and a clustering algorithm produces l clusters. A clustering result is said to satisfy homogeneity if all of its clusters contain only samples from the same class. On the other hand, clustering satisfies the completeness criterion if points of a given class are members of the same cluster. Similarly to F-measure, increasing one often results in decreasing the other, and the weights of completeness versus homogeneity can be controlled with a weight parameter β. Mathematically, the V-measure is defined as

$$V_\beta = (1+\beta)\frac{h \cdot c}{\beta \cdot h + c}. \tag{1.17}$$

Silhouettes

Silhouettes [20] is a method for evaluating how well each point lies within its cluster. The Silhouette value for i-th datapoint is defined as

$$s_i = \frac{b_i - a_i}{\max(a_i, b_i)}, \tag{1.18}$$

where a_i is the average distance from the i-th point to other points in the same cluster c_i, and $b_i = \min_{k \neq c_i} b_{ik}$, where b_{ik} is the average distance from the i-th point to the points in the k-th cluster. It can be seen that $s_i \leq 1$ and the closer the silhouette value is to 1 the better. If we want to measure the quality of a given clustering, we can calculate the silhouette value for each

datapoint x_i and take the mean value, for example, as a reference measure of quality. A higher mean silhouettes value indicates better separation of clusters w.r.t. to point distances.

We should be aware that the metric depends on the distance between a point and *the center* of a cluster it does not belong to. Such a metric might be influenced by the cluster shape and would work well for condensed clusters resembling a Gaussian distribution, but it would be worse for clusters defined as chains of datapoints.

1.3.3 Clustering in latent space

In this thesis, the interest is in the quality of the latent space of trained machine learning models. A model has a latent space of great quality, if it is possible to cluster test data in the latent space with high accuracy (or other evaluation metrics). However, clustering algorithms are unsupervised by nature, therefore there is a need to come up with a procedure of assigning a cluster to an existing class in the data.

There are multiple possible approaches to take. One of them is to cluster both train and test data and assign the cluster to the class that is most represented in the cluster, essentialy leveraging a majority vote. Another one is to cluster only test data and assign a cluster to the class that is closest to the cluster in the latent space based on the distances of the cluster centers.

1.4 Other usefull algorithms

1.4.1 k-Nearest Neighbors

The kNN algorithm [21], [22] is one of the standard supervised algorithms used in classification and regression. The algorithm uses a chosen distance d to find k points closest to a target point – the k nearest neighbors. In classification, the target point is assigned to the class that is most frequent between its neighbors. For regression, the target point is assigned a value as the mean of values of its k nearest neighbors.

The algorithm has its drawbacks, it can suffer from the curse of dimensionality, and its output might not be reliable if the classes in the dataset are not balanced.

1.4.2 UMAP

Uniform Manifold Approximation and Projection [23] is an algorithm used for dimension reduction with the option of clustering on the created embedding. In the past, many dimension reduction techniques have been developed, such as PCA (principal component analysis) [24], or t-SNE [25]. Both t-SNE and UMAP allow for visualization of data in a 2D or 3D space. t-SNE has been a state-of-the-art method used when trying to visualize a high-dimensional space; however, UMAP is able to beat t-SNE both in speed and in manifold quality, and on top of it, it is based on a solid theoretic foundation.

The UMAP computational algorithm works in two phases. In the first, a weighted k-neighbor graph is constructed. In the second phase, the goal is to compute a low-dimensional layout of this graph. There are also three theoretical assumptions that must be met for the algorithm to work as intended:

1. There exists a manifold on which the data would be uniformly distributed.
2. The undelying manifold of interest is locally connected.

3. Preserving the topological structure of this manifold is the primary goal.

The weighted k-neighbor graph can be constructed with any nearest neighbor search algorithm, both the authors and the implementation of UMAP in Julia [26] use the nearest neighbor descent algorithm [27].

The properties of the UMAP algorithm are controlled with four hyperparameters:

- n: number of neighbors,
- d: the dimension of the output embedding,
- min-dist: the desired gap between close points in the embedding space,
- epochs: the number of training epochs to use when optimizing the representation.

The number of neighbors n mostly affects the general structure of the embedding. A smaller n results in smaller clusters of datapoints, where the manifold is more detailed and fragmented. On the other hand, a larger n forces the algorithm to maintain a more global structure. The distance min-dist affects the output directly because it controls the distance between points close to each other. Lower values should result in more dense clusters; higher values make points spread more and aid visual inspection.

UMAP can achieve performance superior to that of t-SNE and other dimensionality reduction algorithms in terms of computational complexity and speed. Thanks to the use of fuzzy topological structure, it is able to compute even higher embedding dimensions (>5) fairly quickly. UMAP scales well with higher dimensions and the number of samples as well.

Although UMAP can be very effective, there are downsides and weaknesses when using it. First, the interpretability is not as strong as with PCA. Second, UMAP always tries to find a manifold representation, even if none exists in the data. For example, using UMAP on random noise with a small number of n neighbors can result in separate clusters which carry no meaning. Furthermore, when there is noise in the data, UMAP can try to find a manifold representing both clusters of datapoints and the noise present.

Chapter 2

Heterogenous structured data and HMill framework

The amount of data that circulate throughout the world grows bigger every day. The amount of so-called Big Data changed numerous fields of technology and industry forever. It allowed large deep machine learning models [28] to draw business conclusions, make life easier for people and large companies alike through task automation, protect customers from malware, improve healthcare, and help doctors make better decisions. There is no question that it is the availability of large datasets that made rapid advancement possible. Still, it is not just the volume of data that researchers and engineers need to worry about: the format of the data stands at the beginning of every new task that we aim to solve with machine learning algorithms or AI.

Most machine learning models expect the input to be a *feature vector*, a vector of (preferably) real values $\mathbf{x} \in \mathbb{R}^d$, where d is the dimension of the feature space. A dataset is then a set of these features $\mathbf{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$ and can be used for a multitude of tasks. It is assumed that the features $\mathbf{x}_i$ are i.i.d., independent and identically distributed. The i.i.d. assumption is of great importance because it allows us to train a model on already collected (history) data and use it for prediction on new data that comes in the future. These two basic assumptions – availability of numerical feature vectors and the i.i.d. assumption – can very quickly become extremely limiting.

First, most data come in different formats: text, images, video, and audio sequences, or structured machine-generated files such as JSONs or XML files. For us to be able to perform any data analysis on such a dataset, we first need to figure out how to transform data of different formats to numerical features. This process is usually called *feature extraction* or *feature engineering*. In the beginning, researchers would use expert knowledge to come up with the features themselves. Nowadays, there are feature extraction algorithms for most types of data, such as (deep) convolutional neural networks for image data or the Word2Vec model for text.

Second, most data streams are not stationary and show signs of a *concept drift*. Concept drift can be defined as a change in the original data distribution over a period of time. Additionally, completely new categories of data can appear over time, such as new malware types that antivirus software needs to detect. The task of detecting emerging classes is called *novelty detection*.

Another problem can arise when the data is *structured*. For example, a user starts browsing an e-commerce site and adds multiple items to their shopping cart. Say the task would be to predict whether or not the user will actually go through with their purchase or to recommend an item to go with the items already in the shopping cart. The data we have about the user and their shopping cart is hierarchical in nature: there are general features regarding the user and multiple items in

the shopping cart, which all have their own features. The structure, when written in a JSON-like file, might look like

```
username: user123,
date_registered: 2020-11-30,
completed_orders: 3,
...,
shopping_cart: {
    basketball: {
       price: 9.99,
       quantity: 1,
       in_stock: true,
       color: brown,
    },
    basketball_shoes: {
       price: 49.99,
       quantity: 1,
       in_stock: true,
       color: [white, blue],
       size: 6.5
    },
    ankle_bandage: {
       price: 3.49,
       quantity: 5,
       in_stock: true,
       color: beige
    }
}
```

We assume that every sample for each customer would have the same structure, but there might be variations in the number of items in the shopping cart or anywhere else. The question is, how do we process such data and how do we define a feature vector representing it?

Looking at the example sample, there are multiple challenges to tackle:

- **Varying variable type:** Strings need to be encoded differently to numbers, categorical variables, or boolean values.
- **Varying item size:** The shopping cart can include any number of items. If we decide to encode it with a fixed size feature vector, we might end up either having too large of a vector for most shopping carts, or losing information when the number of items surpasses the predefined dimension of the feature vector.
- **Missing information:** Some items might have more features available than others. For example, information about shoes should include shoe size, but shoe size is not available for most other products. Therefore, the value of shoe size would be missing for any product other than shoes.

Overall, it is clear that the hierarchical structure of the data introduces new challenges. Since data representation is paramount to any machine learning model's success, there is a need to define a model which can be trained on such data. The HMill framework presented in the next section is designed specifically to work with data resembling the given example and is able to process hierarchical structured data and solve the problems mentioned.

2.1 HMill framework

The name HMill stands for *Hierarchical Multi-Instance Learning Library* [29] and is developed by Pevný and Mandlík [30] as a package `Mill.jl` written in Julia language. The library allows users to create trainable machine learning models that process tree-structured data, from simple sets to JSON files. HMill, its properties and usage are well described in a thesis by Mandlík [31]. What follows is a gentle introduction to the framework, its terminology, and composition.

First, on the terminology used. The *HMill sample* or just *sample* refers to one data sample, for example, one JSON file. Samples in a data set are expected to have a tree-like structure that can be described in general *schema*. Data inside the structure is wrapped in *data nodes* and modeled with corresponding *model nodes*.

2.1.1 Data nodes

HMill defines three types of data nodes: Array Node, Bag Node and Product Node. Each type is used to process and store different types of information about the data.

Array Node stores the lowest-level features. The features x should be defined in a feature space $\mathcal{F}$, where the only constraint on $\mathcal{F}$ is that there exists a reasonable representative mapping $h : \mathcal{F} \mapsto \mathbb{R}^n$ which transforms features of any type (strings, boolean values, categorical values) into a Euclidean space. The array node is then simply a wrapper for the extracted feature $\text{an}(h(x))$. The function h can be viewed as a preprocessing function and should be defined with the nature of the data in mind to best represent the values. For example, if x is a categorical variable, h can be one-hot encoding.

Bag Node stands at the core of the framework. Bag node is a wrapper on a set of instances, where the number of instances can vary and may even be zero. The set is also called a bag and is simply $b = \{t_1, \ldots, t_{|b|}\}$, where t_i are instances and can be referred to as children. Bag node is denoted as $\text{bn}(b)$. Apart from the varying number of instances, a bag has another important property: permutation invariance. This means that the order of instances in the bag is irrelevant and that any permutation should lead to the same result. The instances may be array nodes or other bag nodes, but need to be of the same type. This allows for nesting and deeper tree structures.

Product Node is the last type of data node. Product node is used to join multiple instances like the bag node, but forces a static order on its children, making it possible to aggregate data from different sources or of different structures.

2.1.2 HMill models

So far, we have described how to preprocess the raw low-level features and store the data. Now, we need to define the hierarchical model that can be trained on the data.

Array model acts on the array nodes and simply applies a parametric function to the already preprocessed feature. If $x \in \mathcal{F}$ is the low-level feature and h is the mapping to Euclidean space $\mathbb{R}^n$, the array model is a function $f : \mathbb{R}^n \mapsto \mathbb{R}^m$, where f is a feedforward neural network.

The task of a **Bag model** is to take a bag node as input and output a vector in the Euclidean space. The mapping of a bag model is composed of three functions, $\text{bm}(f_I, g, f_B)$: instance model f_I, aggregation g, and bag mapping f_B. First, the instance model f_I is applied to every instance t_i in the bag node $\text{bn}(\{t_i\}_{i=1}^k)$, resulting in a transformed bag node $\text{bn}(\{f_I(t_i)\}_{i=1}^k)$. To get a single vector representation of the transformed bag node, the aggregation function g is applied. Such a function needs to be permutation invariant; an example is the mean or maximum function, or their

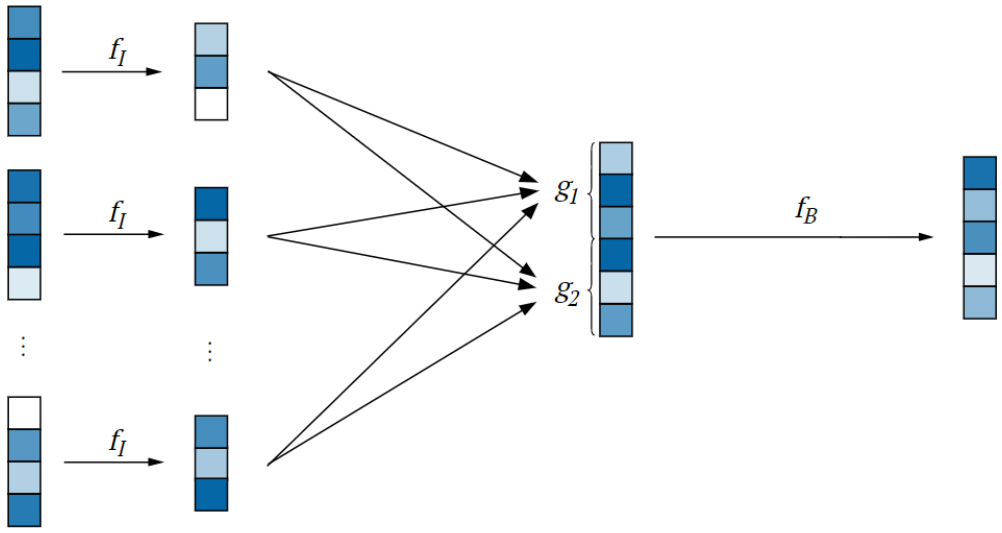


Figure 2.1: Illustration of the functions applied in the BagModel structure. All instances are processed with instance network $f_I : \mathbb{R}^4 \mapsto \mathbb{R}^3$, aggregated with functions g_1, g_2, and sent through a bag network $f_B : \mathbb{R}^6 \mapsto \mathbb{R}^5$.

combination. The aggregation results in a vector with real values $\hat{x} = g\left(\mathrm{bn}(\{f_I(t_i)\}_{i=1}^k)\right)$, $\hat{x} \in \mathbb{R}^p$. Lastly, the bag mapping $f_B : \mathbb{R}^p \mapsto \mathbb{R}^q$ is applied to $\hat{x}$. The mapping functions are feedforward neural networks as well as for the array model. Figure 2.1 shows the flow of data through the BagModel.

Product model $\mathrm{pm}(f_1, \ldots, f_l, f)$ is basically a collection of submodels for each of the children of the product node it acts on, followed by a last mapping function f. After applying the submodels f_j, the resulting vectors are concatenated, creating the Cartesian product of the target spaces of $f_1, \ldots, f_l$. Finally, the concatenated vector is transformed once more with the f network. Visualization of the process can be seen in Figure 2.2.

The hierarchical structure of models makes it possible to differentiate through all the neural networks and update the parameters to minimize a loss function, for example, cross-entropy for classification.

2.1.3 Further details

The described properties of HMill framework are just the most important features. `Mill.jl` provides users with tools to model data in numerous ways. The aggregation functions implemented include not just mean and maximum, but also bag count, sum, or parametric aggregations such as the p-norm. HMill is also able to handle missing data. We have discussed the property of a bag, where a bag can contain *any* number of instances, meaning that even *zero* number of instances is possible. `Mill.jl` is designed to model even these cases, as well as missing parts of JSON files etc. These nuances in the framework implementation make it versatile, powerful and customizable.

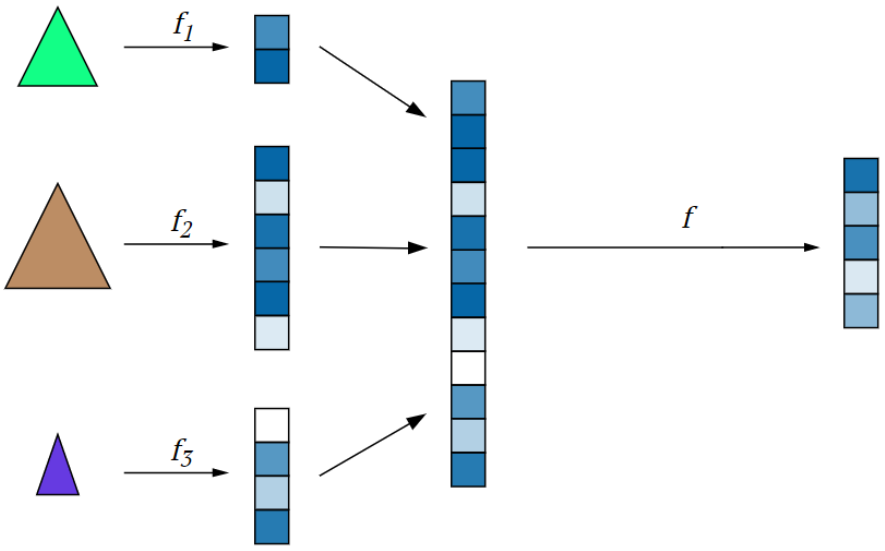


Figure 2.2: Illustration of the functions applied in the ProductModel structure. Each input is processed with its own function f_i to a vector representation in Euclidean space. These output vectors are concatenated and go through a final network f.

2.2 Examples of different data structures and their properties

Now that we have introduced both the concept of hierarchical structured data and a framework to use for those data, we can go a little deeper and look at the three types of data used in this thesis in the experimental sections. First, two variations of *group data* are described: multi-instance problems and point cloud datasets. The other type of data discussed are JSON files.

2.2.1 Multi-instance learning and point clouds

Multi-instance learning datasets and point clouds can also be called *group datasets*. Both data structures are only one level deep, a sample $\mathbf{X}$ is a set of points $\mathbf{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$, where $n \in \mathbb{N}$ can vary for different sets. The dataset is then made up of sets $\mathcal{D} = \{\mathbf{X}_1, \ldots, \mathbf{X}_d\}$. It is important to note that the order of instances $\mathbf{x}_i$ in the set is irrelevant. Therefore, the joint probability distribution of instances must satisfy a permutation invariance condition

$$p(\mathbf{x}_1, \ldots, \mathbf{x}_n) = p(\mathbf{x}_{\pi(1)}, \ldots, \mathbf{x}_{\pi(n)}). \tag{2.1}$$

for $n!$ permutations π. When modeling these data, a permutation invariant function should be used to ensure this requirement [32]. Such functions include, for example, `mean` or `max`. This idea coincides with the usage of an aggregation function g in the HMill framework.

Permutation invariance introduces a responsibility problem, as described in [33]. The illustration is clear on problems such as generating point clouds. A point cloud is a set of points, usually coordinates in 2D or 3D space. An example of a set of such points are MNIST digits in Figure 2.3. Since the points in the cloud are independent and their order does not influence the outcome, the question is: Which neurons in the neural network should be responsible for generating which points? Using the standard mean square error for training a neural network becomes impossible

Figure 2.3: Examples of digits from the point cloud MNIST dataset.

due to the permutation invariance. This problem can be solved using different loss functions, for example, the Chamfer distance [34]. For two sets $\mathbf{X}, \mathbf{Y}$ with samples $\mathbf{x}_i, \mathbf{y}_i$, $i \in \{1, \ldots, n\}$, the Chamfer distance is computed as follows:

$$\mathcal{L}_{CH}(\mathbf{X}, \mathbf{Y}) = \sum_i \min_j ||\mathbf{x}_i - \mathbf{y}_i||^2 + \sum_j \min_i ||\mathbf{x}_i - \mathbf{y}_j||^2 . \tag{2.2}$$

The loss is permutation invariant and calculates the distance between the closest points. It is also possible to use this loss to calculate the distance between sets of various sizes. Chamfer loss can be used for training or evaluation of generative models such as the PointFlow model [35] on point cloud datasets. Chamfer distance remains popular in point cloud modeling thanks to its quick computation compared to other commonly used point cloud loss functions such as the Earth mover's distance.

Even though the underlying data structure is the same, multi-instance datasets have a different interpretation and are usually used as an anomaly detection problem. The definition of multi-instance learning dates back to 1997. One of the first illustrations of the problem is:

Imagine there are multiple people with keys on a keychain. There is a door and only one key opens this door. We can gather information about which person can open the door with their keychain, but we do not have information about which key on the keychain is the one. The task is to find the key.

This gives rise to the standard MI problem: A bag is negative if all instances in the bag are negative, and positive if at least one instance is positive. This setting comes from one of the first multiple-instance problems – drug activity prediction. A molecule of the drug can take multiple shapes (which will be the instances in a bag). If it can take a shape that binds to a target protein – also known as lock-key pair – the binding will be successful. Otherwise, the drug will not be effective. This problem is described in detail in [36].

Nowadays, MI problems are approached similarly to group anomaly detection. The object of importance is a set, a bag of instances. The task is to train a model such that it learns the best representation of the normal class and can detect anomalous bags which do not share the same feature distribution as normal samples. For the purposes of this thesis, a multi-instance dataset is used for multi-class classification.

2.2.2 JSON files

JSON stands for **J**ava**S**cript **O**bject **N**otation and acts as a native format for data in JavaScript applications [37]. JSON eventually became the go to data interchange format [38], surpassing the previously popular XML format, especially thanks to its simplicity and compact semantics.

The JSON object is built on two basic structures [39]:

- a collection of key-value pairs,
- an ordered list of values.

Since these structures are universal accross modern programming languages, JSON files can be easily shared and processed. For example, JSONs are used for sharing information through APIs (application programming interface). An example of a real JSON file (with some keys deleted to shorten it) is in code 2.1.

Every JSON file can be defined by its structure and this is leveraged in HMill with the use of *schemata*. A schema allows for extraction of data from raw JSON files and building the data in the format of product nodes and bag nodes. For each value in the JSON file, a data type is either inferred with heuristic rules, or defined by users. Either way, the HMill model gets the information that a value in the key-value pair `"name": "John Doe"` is a String and would be consequently modeled with n-grams.

The inferred schema of the JSON file 2.1 can be seen in the code block 2.2. The function `schema` from JsonGrinder.jl [40] is able to infer the type of value, mostly Strings, but Integer for key `"total_tracks"` or Boolean value for key `"is_local"`. The next step is to differentiate nuances in the data, for example, defining when a String is a string, and when it should be treated as categorical variable.

```
{
  "album": {
    "album_type": "album",
    "artists": [
     {
       "id": "7xTcuBOIAAIGDOSvwYFPzk",
       "name": "Daniel Powter",
       "type": "artist",
     }
    ],
    "id": "4zhigAhPwqp43XVHBiVeQI",
    "name": "Daniel Powter",
    "release_date": "2005-02-22",
    "total_tracks": 10,
    "type": "album",
  },
  "artists": [
    {
     "id": "7xTcuBOIAAIGDOSvwYFPzk",
     "name": "Daniel Powter",
     "type": "artist",
    }
  ],
  "disc_number": 1,
  "duration_ms": 233640,
  "explicit": false,
  "id": "0mUyMawtxj1CJ76kn9gIZK",
  "is_local": false,
  "name": "Bad Day",
  "popularity": 75,
  "track_number": 3,
  "type": "track"
}
```

Code 2.1: Example of a real JSON file from the Spotify API [41].

```
[Dict]
  |---- explicit: [Scalar - Bool],
  |-------- album: [Dict]
  |          |---------------------- id: [Scalar - String],
  |          |--------------------- type: [Scalar - String],
  |          |--------------------- name: [Scalar - String],
  |          |-------------- album_type: [Scalar - String],
  |          |------------ total_tracks: [Scalar - Int64],
  |          |------------ release_date: [Scalar - String],
  |          |----------------- artists: [List]
  |                                |-- [Dict]
  |                                       |---- id: [Scalar - String],
  |                                       |-- type: [Scalar - String],
  |                                       |-- name: [Scalar - String],
  |-- duration_ms: [Scalar - Int64],
  |-- disc_number: [Scalar - Int64],
  |----- is_local: [Scalar - Bool],
  |- track_number: [Scalar - Int64],
  |------ artists: [List]
  |          |-- [Dict]
  |                |---- id: [Scalar - String],
  |                |-- type: [Scalar - String],
  |                |-- name: [Scalar - String],
  |--------- name: [Scalar - String],
  |--- popularity: [Scalar - Int64],
  |----------- id: [Scalar - String],
  |--------- type: [Scalar - String],
```

Code 2.2: Inferred schema from JSON file shown in code 2.1.

2.3 Model training with HMill

The HMill model serves as a tool for learning features of hierarchical data in a way similar to how convolutional neural networks extract features from images. Therefore, the HMill model can be used as a preprocessing network, and its output becomes an input to any other network, for example, a standard classifier.

The implementation makes model creation easy. The Julia package Mill.jl [29] contains a function `reflectinmodel` which takes the input data, and optionally other arguments, and returns a HMill model. For example, the code 2.3 takes training data and infers the type of model. The second argument tells the function to create a model with a dense layer, 64 neurons in hidden dimension, and ReLU activation function. Lastly, the aggregation function used in the BagModel is the concatenation of the mean and maximum functions.

```
mill_model = reflectinmodel(
    Xtrain,
    d -> Dense(d, 64, relu),
    SegmentedMeanMax
)
```

Code 2.3: Example of usage of the function `reflectinmodel`.

The created `mill_model` is just a means to learn features of data. Say the aim is to classify a MNIST point cloud dataset with c classes. What we need to do is define a classifier on top of the

features of the HMill model. Flux.jl [42] in Julia implements a simple function to chain multiple functions together. To create a classifier, we simply need to add code 2.4.

```
classifier = Chain(
    mill_model,
    Mill.data,
    Dense(64, 64, relu), Dense(64, c)
)
```

Code 2.4: Code to build a classifier with HMill.

The code chains the predefined `mill_model`, `Mill.data` function, which only changes the type of input from a specific ArrayNode type to a matrix of floating numbers, and two dense layers. The classifier therefore returns a vector of the same length as the number of classes. In the end, the only thing needed is a loss function, for example, a simple cross-entropy on a softmax of the model's output.

To train the model, only four more lines of code in 2.5 are needed. The first function defines a multiclass cross-entropy loss in a more stable manner, without needing to apply softmax directly. In the second line, all the parameters of our classifier are collected and in the third line the ADAM optimizer is initialized with the learning rate $\eta = 0.001$. The last line brings everything together and optimizes the model to minimize the loss function. It should be noted that all parameters are updated in one training phase, learning both the embedding of the HMill model and the classifier on top of it simultaneously.

```
loss(x, yonehot) = Flux.logitcrossentropy(classifier(x), yonehot)
ps = Flux.params(classifier)
opt = ADAM(0.001)
Flux.train!(loss, ps, data, opt)
```

Code 2.5: Training a model with Flux.jl.

Chapter 3

Semi-supervised learning for group and hierarchical data

The core of this thesis is the extension of methods used for vector data or images described in Chapter 1 to the hierarchichal paradigm introduced in Chapter 2. The list of models starts with discussion about a straightforward usage of self-supervised learning for structured data in section 3.1, extends the classification networks with ideas from metric learning domain in section 3.2 and finally, generative models for one-level hierarchical data are build in sections 3.3 and 3.4.

3.1 Self-supervised models

Two self-supervised models will be discussed in this section. The first one is a self-supervised classifier, which directly copies the algorithm described in section self-training in Chapter 1. A HMill classifier is initialized, trained and used to pseudo-label unlabeled data. All data classified with probability higher than given threshold are supplied to the classifier as labeled data for the next iteration of the algorithm. The threshold is approached as a hyperparameter of the model.

The second self-supervised model combines the ideas of metric learning and self-training. The model is designed as a HMill model with subsequent layers and serves as a feature extraction algorithm. The algorithm mimics the ArcFace model, where instead of a CNN, the feature extraction algorithm is the HMill model. The training procedure stays the same.

Pseudo-labeling is done on the learned feature space with kNN algorithm, where k is a hyperparameter. After every training iteration of the HMill ArcFace model, all unlabeled data is pseudo-labeled with the kNN algorithm and mean distance from the k neighbors is calculated for every unlabeled point. The distances are sorted and a distance threshold is picked based on a chosen quantile $q \in (0, 1)$ (q is also a hyperparameter). Only the points whose distances lie below the threshold are used as labeled data in the next iteration of the algorithm. With this approach, q % of unlabeled data are pseudo-labeled and added to the training data in the next iteration. The HMill ArcFace model is retrained in every iteration just like the self-supervised classifier.

Further improvement of the ArcFace self-supervised model can be achieved with added regularization. It is possible to use a triplet loss, for example, on the features outputted by the feature extraction HMill model and force a more discriminative features before maximizing the angular margin between classes.

3.2 Metric learning and clustering

In section 1.1.3 we have introduced the idea of using clustering in the input space to add labels to unlabeled data in the training dataset. However, clustering on the input space of hierarchical data is a complicated, if not impossible, task. How do we define a similarity metric to measure closeness in input space for JSON files or group data? In the one-level case with numerical features such as MIL or point cloud datasets, it is possible to use a metric for sets such as the Chamfer distance (2.2) for calculation of the distance matrix, and use it as input for a clustering algorithm or kNN. But what about more complicated structures? And what about simple group datasets where the Chamfer distance does not correspond with the nature of the data as well as it does with point cloud datasets?

The possible solution to our problem has been mentioned previously when describing the M1 model as an unsupervised preprocessing algorithm for feature extraction (and possibly dimensionality reduction). HMill models naturally act as feature extraction algorithms, since they are used to project a set or a JSON file to a Euclidean embedding space. The only question is, what objective to optimize to learn a discriminative embedding of the data?

It is possible to train a discriminative model, an HMill classifier, as before, and look at the embedding space learned by the HMill model in terms of cluster quality. The disadvantage of such approach is that there is no condition on the properties of the embedding space. The only property needed is the separation of classes, but no margin is forced to be within the classes to better distinguish them. That is when metric learning comes in.

Metric learning can be used in combination with the HMill models. Triplet loss is used in this thesis, but the same ideas would apply to other losses in the metric learning domain. There are two approaches to classification that can be taken

1. use supervised learning with triplet loss and kNN (or any clustering algorithm) to obtain labels based on distances in the embedding space, or
2. add triplet loss minimization as a regularization to a standard classifier.

The first approach is very similar to the ArcFace self-supervised model discussed in the previous section. The second one coincides with one of the objectives of this thesis, which is to improve the latent space created by standard models. Therefore, one of the questions posed for the experimental part of this work becomes: *Does triplet loss regularization improve the accuracy of a classifier, and does it create an embedding space of higher quality?*

A **classifier with triplet regularization** is designed to try and optimize two objectives simultaneously: multi-class cross-etropy for standard classification and triplet loss calculated on the encoding of the HMill component of the classifier. For a batch of n samples, the input data sample $\mathbf{X}_i$ (a bag, JSON, etc.) is processed with the HMill model with parameters ϕ to the embedding vector ξ_i and the loss is calculated as

$$\mathcal{L}(\theta, \phi) = -\frac{1}{n}\sum_{i=1}^{n}\sum_{y\in\mathcal{C}}[I\{y = y_i\}\log q_\theta(c \mid \xi_i)] + \mathcal{L}_{triplet}(\{\xi_1, \ldots, \xi_n\}), \tag{3.1}$$

where the classifier q_θ is parameterized as a neural network and both the classifier and HMill model are trained simultaneously.

3.3 Generative model for sets of instances

The M2 model for semi-supervised learning and classification described in section 1.1 assumes that there exists both a classification and a generative model for the input data. Unfortunately, we have yet to design a generative model for hierarchical structured data such as JSON files, however, there are existing methods for generative modeling of sets of instances. Multiple approaches can be used. First, we will discuss a direct extension of the M2 model to sets, then introduce the Neural Statistician generative model for group data.

3.3.1 Extension of the M2 model

A set is a collection of instances $b = \{x_i\}_{i=1}^{|b|}$, with one label per set y. If the instances are iid, they can be modeled with an instance generative model such as VAE. Modifying the M2 model is fairly straightforward. The prior distribution $p(\mathbf{z})$ remains the same with the only change that for each bag b there are $|b|$ latent variables $\mathbf{z}_i$ for each instance $\mathbf{x}_i \in b$. The distribution $p_\theta(\mathbf{x} \mid y, \mathbf{z})$ is modified in the same way and becomes $p_\theta(\mathbf{x}_i \mid y, \mathbf{z}_i)$, where the bag label y is copied $|b|$ times for each $\mathbf{x}_i$. The approximate distribution $q_\phi(\mathbf{z} \mid y, \mathbf{x})$ changes accordingly with the previous distributions. To get the likelihood, we simply use the rule for the joint probability distribution of iid random variables, where $p(x_1, \ldots, x_n) = \prod_{i=1}^{n} p(x_i)$ and $\log p(x_1, \ldots, x_n) = \sum_{i=1}^{n} \log p(x_i)$.

The most significant change occurs with the classification distribution $q_\phi(y \mid \mathbf{x})$, since the neural network $\boldsymbol{\pi}_\phi$ needs to be an HMill model to extract useful information about the entire bag b into a one-vector representation that is then sent to a classifier as before. The distribution is conditioned on b, $q(y \mid b)$.

As a result of the changes, the objectives of the M2 model for sets become

$$\begin{aligned}
\log p(b, y) &= \sum_{\mathbf{x}_i \in b} \log p_\theta(\mathbf{x}_i, y) \geq \\
&\geq \sum_{\mathbf{x}_i \in b} \mathbb{E}_{q_\phi(\mathbf{z}_i \mid \mathbf{x}_i, y)} \big[\log p_\theta(\mathbf{x}_i \mid y, \mathbf{z}_i) + \log p_\theta(y) + \log p(\mathbf{z}_i) - \log q_\phi(\mathbf{z}_i \mid \mathbf{x}_i, y) \big] = \\
&= \sum_{\mathbf{x}_i \in b} \Big(\mathbb{E}_{q_\phi(\mathbf{z}_i \mid \mathbf{x}_i, y)} \left[\log p_\theta(\mathbf{x}_i \mid y, \mathbf{z}_i) \right] + \log p_\theta(y) - \mathcal{D}_{KL} \left[q_\phi(\mathbf{z}_i \mid \mathbf{x}_i, y) \mid\mid p(\mathbf{z}_i) \right] \Big) = \\
&= -\mathcal{L}(b, y)
\end{aligned} \tag{3.2}$$

for a labeled bag and

$$\begin{aligned}
\log p(b) &= \sum_{\mathbf{x}_i \in b} \log p_\theta(\mathbf{x}_i) \geq \\
&\geq \sum_{\mathbf{x}_i \in b} \mathbb{E}_{q_\phi(y, \mathbf{z}_i \mid \mathbf{x}_i)} \big[\log p_\theta(\mathbf{x}_i \mid y, \mathbf{z}_i) + \log p_\theta(y) + \log p(\mathbf{z}_i) - \log q_\phi(y, \mathbf{z}_i \mid \mathbf{x}_i) \big] = \\
&= \sum_{\mathbf{x}_i \in b} \Big(\mathbb{E}_{q_\phi(y, \mathbf{z}_i \mid \mathbf{x}_i)} \left[\log p_\theta(\mathbf{x}_i \mid y, \mathbf{z}_i) \right] - \mathcal{D}_{KL} \left[q_\theta(\mathbf{z}_i \mid \mathbf{x}_i) \mid\mid p(\mathbf{z}_i) \right] - \mathcal{D}_{KL} \left[q_\phi(y \mid \mathbf{x}_i) \mid\mid p_\theta(y) \right] \Big) = \\
&= \sum_{\mathbf{x}_i \in b} \left(\sum_y q_\phi(y \mid b)(-\mathcal{L}(\mathbf{x}_i, y)) + \mathcal{H}\left(q_\phi(y \mid b) \right) \right) = -\mathcal{U}(b)
\end{aligned} \tag{3.3}$$

for a bag without a label.

Another question is how to modify the hyperparameter α. In the base version of the M2 model, $\alpha = 0.1 \cdot N$, where N is the number of datapoints with labels. Through experimental evaluation, it seems that for the bag model, this hyperparameter value still applies with a slight change, where N is not the number of *bags* with known labels, but the number of *instances* in bags with known labels.

The model described does have its limitations, namely that the instances are modeled independently and the only thing distinguishing them is the label, which is inferred as the label of the bag it belongs in. It is possible to extend the information given to the encoding distribution $q(\mathbf{x}_i \mid \mathbf{z}_i, y)$ with the information about the bag. First, the bag b is projected with an HMill model to a one vector feature h, and the encoder is modified to $q(\mathbf{x}_i \mid \mathbf{z}_i, y, h)$, meaning the input to the encoder is a vertical concatenation of instance $\mathbf{x}_i$, one-hot encoded label y and bag projection h.

3.3.2 Neural Statistician

A Neural Statician model [43] presented back in 2016 acts as a direct extension of the variational autoencoder. The assumption is that each set $\mathbf{X}_i$ consists of iid instances $\mathbf{x}$ generated from distribution p_i and there exists a generative process p such that $p_i(\mathbf{x}) = p(\mathbf{x} \mid \mathbf{c}_i)$. The vector $\mathbf{c}_i$ is called *context* and should contain information about the entire set $\mathbf{X}_i$. It is assumed that $\mathbf{c}_i$ comes from its own distribution $p(\mathbf{c})$.

Following the configuration of VAE, the likelihood of a set $\mathbf{X}$ is given by

$$p(\mathbf{X}) = \int p(\mathbf{c}) \left[\prod_{\mathbf{x} \in \mathbf{X}} \int p(\mathbf{x} \mid \mathbf{z}) p(\mathbf{z} \mid \mathbf{c}) \mathrm{d}\mathbf{z} \right] \mathrm{d}\mathbf{c}. \tag{3.4}$$

With the usage of approximate inference networks, an ELBO approximation is derived, similarly to the ELBO for the variational autoencoder, as

$$\mathcal{L} = \mathbb{E}_{q(\mathbf{c}|\mathbf{X})} \left[\sum_{\mathbf{x} \in \mathbf{X}} \mathbb{E}_{q(\mathbf{z}|\mathbf{x},\mathbf{c})} \left[\log p(\mathbf{x} \mid \mathbf{z}) \right] - \mathcal{D}_{KL}\big(q(\mathbf{z} \mid \mathbf{x}, \mathbf{c}) \mid\mid p(\mathbf{z} \mid \mathbf{c})\big) \right] - \mathcal{D}_{KL}\big(q(\mathbf{c} \mid \mathbf{X}) \mid\mid p(\mathbf{c})\big). \tag{3.5}$$

The KL divergence at the end of the equation makes the posterior $q(\mathbf{c}|\mathbf{X})$ map to the isotropic Gaussian $p(\mathbf{c}) = \mathcal{N}(\mathbf{0}, \mathbb{I})$ and encode information about the whole set $\mathbf{X}$. The KL divergence within $\mathbb{E}_{q(\mathbf{c}|\mathbf{X})}$ is an error term that is supposed to map latent variables $\mathbf{z}$ so that they are tied to context $\mathbf{c}$. All distributions except $p(\mathbf{c})$ are chosen to be Gaussian with diagonal covariance matrix and are parameterized by trainable neural networks.

It is possible to extend the Neural Statistician model to become the generative model in the M2 model structure. The distributions are modified to be conditioned on the label of a sample, both in the context and latent space distributions, resulting in $q(\mathbf{c} \mid \mathbf{X}, y)$, $q(\mathbf{z} \mid \mathbf{x}, \mathbf{c}, y)$, $p(\mathbf{z} \mid \mathbf{c}, y)$. The classifier network is the same as for the bag M2 model and the optimization objective is modified accordingly.

3.4 Generative model for specific JSON data

Building on the knowledge of HMill framework and semi-supervised generative models, the last objective of this thesis is to design a prototype of a generative model for JSON-like structured data. The task is not trivial, as the hierarchical nature of the data would require a nested generative

submodel for every branch of the file. Therefore, the decision is to study only one special case of an important JSON structure.

The chosen special case is a subset of JSON files from the GVMA dataset kindly provided by Avast. The schema of each JSON file `Xf` in this dataset is a two-level tree structure. There are three key-value pairs at the top level, and only one of them, `behavior_summary`, is chosen to be modeled. Consequently, the input sample is `X = Xf[:behavior_summary]`. The sample `X` is further composed of 18 leaves `x`, where each leaf is indexed with a subkey, for example, `x = X[:files]`.

The leaf values are strings, which are preprocessed to be either n-gram matrices or one-hot matrices based on the schema provided with the data. The advantage is that both n-grams and one-hot matrices can be modeled and generated as matrices of numbers in $\mathbb{R}$. Therefore, the task has been simplified from generating full JSON files to generating multiple bags of numerical features. As a result, the M2 model paradigm described in section 3.3.1 can be used for semi-supervised learning on this simplified JSON dataset.

The classification network for the M2 bag model is defined as a standard HMill classifier on the input sample `X`. The prior distribution of class labels $p(y)$ remains unchanged and is optimized during training as well.

Each leaf `x = X[:subkey]` in sample `X` is modeled with a single generative submodel that is chosen to be the generative component of the M2 bag model. This model needs to be constructed separately for each leaf because each leaf can have a different input dimension and different properties. For this particular example, 18 generative submodels must be created. The submodels are designed to have the same hyperparameters (activation and aggregation functions, number of neurons, layers, etc.) for simplicity. Every submodel is trained to optimize the objectives of the M2 bag model (3.2), (3.3) for labeled and unlabeled data, respectively. The complete generative model is a collection of generative submodels; therefore, the objectives of the submodels need to be aggregated. The natural choice for aggregation is the sum of single log-likelihoods, assuming the generative submodels are independent.

Figure 3.1 shows a simple diagram to visualize the flow of data through the M2 model architecture. Both the generative models and the classifier are trained only on data from the key `:behavior_summary` in the input JSON file.

The generative models are tied to the classification network through the loss for data with unknown labels (3.3). Let us denote input $\mathbf{X} = \mathtt{X}$ and individual samples marked by subkey $\mathbf{x}_k,\ k \in \{1, \ldots, 18\}$. Revisiting M2 model equations (1.7), (1.8), the loss function becomes

$$\mathcal{J} = \sum_{\mathbf{x}_k \in \mathbf{X}} \left[\sum_{(\mathbf{x}_k, y) \sim \tilde{p}_l} \mathcal{L}_k(\mathbf{x}_k, y) + \sum_{\mathbf{x}_k \sim \tilde{p}_u} \mathcal{U}_k(\mathbf{x}_k) \right] + \mathbb{E}_{\tilde{p}_l(\mathbf{X}, y)} \left[-\log q_\phi(y \mid \mathbf{X}) \right], \tag{3.6}$$

where the loss for known and unknown labels is computed for each submodel k based on the sample $\mathbf{x}_k$ and summed. The important part that ties the individual generators with the classifier is

$$\sum_y q_\phi(y \mid \mathbf{X}) \cdot \mathcal{L}_k(\mathbf{x}_k, y). \tag{3.7}$$

The equation (3.7) leverages two joint conditions for parameter optimization. It is expected that a sample $\mathbf{x}$ would be best reconstructed with the generative model, if it is supplied with the correct label y, and the reconstruction loss would be higher for incorrect labels. Therefore, the classification network is optimized to give higher probabilities to labels that result in smaller reconstruction error. This should also work the other way around, since if a classifier gives a high probability to class y, the generative model tries to minimize the corresponding reconstruction loss accordingly.

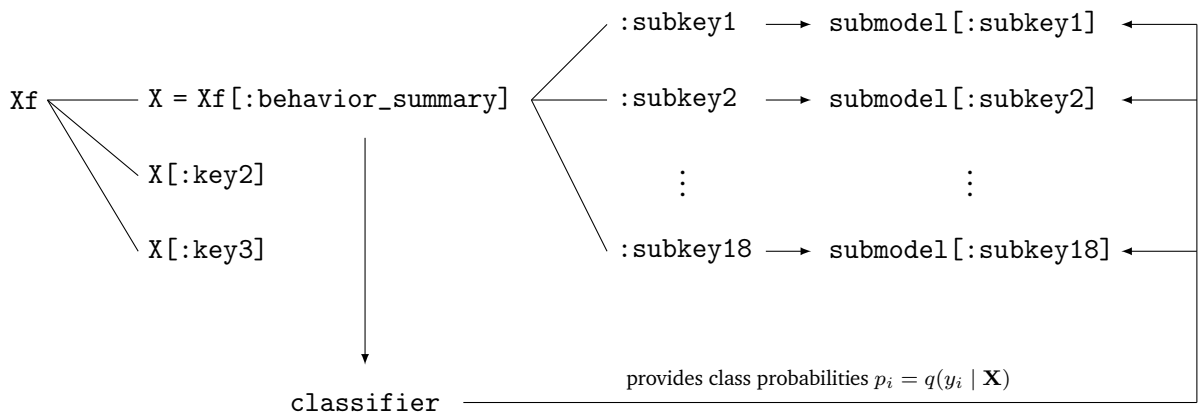


Figure 3.1: A simplified schema for visualization of M2 model for a JSON sample. The JSON file is `Xf`, BagNode `X = Xf[:behavior_summary]` is the input into the M2 model. A submodel is created for each subkey of `X`. The classifier acts on the input and provides class probabilities into loss functions for unlabeled data of the submodels.

Chapter 4

Experimental setup

In the previous chapters, we have defined methods for semi-supervised learning and clustering on embedding space of group data (MI problems and point clouds) and hierarchical structured data. This chapter discusses the setup of experiments to evaluate the methods described and compare them against each other.

4.1 Data

For experimental evaluation, three datasets from different domains were chosen: multi-instance dataset, MNIST point cloud, and JSON data.

4.1.1 MI problems

Multi-instance learning datasets have one disadvantage, the number of bags is quite low. Therefore, three MIL datasets, Elephant, Fox and Tiger [44], were taken and merged to create a multiclass dataset *Animals* with three classes. The dataset has 300 bags in total, 100 bags per class. The input dimension is 230 features. The chosen ratios of interest are 5, 10, 15, and 20 % of data from the training dataset labeled. The data used can be found at `https://github.com/pevnak/MIProblems`.

4.1.2 MNIST point cloud dataset

Well-known MNIST dataset [45] can be easily converted into a point cloud dataset. The features extracted are the position in a 2D space (x and y coordinates) and the pixel value at this position v. Only datapoints with non-zero pixel values are used.

Point clouds expose a specific problem because the coordinates in 2D or 3D space might be discrete values. Generative models such as VAE describe datapoints as densities and approach them as samples from a continuous distribution (e. g. a Gaussian). However, image data is usually stored in integers and therefore discrete. A simple solution is to add uniform noise $\mathbf{u} \sim U(0, 1)$ and dequantize the data [46]. Dequantization is applied to remove the problem of training on discrete data. Each set then consists of datapoints (x, y, v) and the number of points in each set varies between 34 and 351. The dataset contains 70k samples in total.

Two instances of the dataset are examined: a smaller version of only four digits (0, 1, 3, and 4) and the full version of all 10 digits. The ratios chosen are 0.2, 1, 5, and 10 % of samples labeled. This corresponds with 14, 70, 350, and 700 bags with known labels for each class, respectively.

4.1.3 JSON malware dataset

The last dataset was kindly provided by Avast. The dataset is a small experimental subsample of 8000 JSON files with 9 malware classes and one class representing clean samples. The classes are balanced, meaning there are 800 samples from each class.

The data structure is two-level. The sample has three main branches, where the first branch contains 18 BagNodes of inferred type either n-grams or one-hot matrices, the second branch contains three BagNodes of two one-hot matrices and a simple array, and the third branch contains only one BagNode.

The chosen ratios for experimentation are 1, 2, 5, 10, and 20 % of labels provided to the model.

4.2 Training and model selection

To ensure a fair comparison of the models, standard procedures have been used for running experiments.

Data is always split to train, validation, and test dataset, where the training data consists of labeled and unlabeled data. First, the data set is divided according to the given ratio r, where r controls the percentage of data with labels. The ratio of labeled, unlabeled and test data is $(r, 0.5 - r, 0.5)$, meaning that test data is always 50 % of all data. Labeled training data is sampled so that classes are balanced. Validation data is randomly sampled from training data with unknown labels so that the number of labeled samples and validation samples is the same.

Each model has its own training loop designed. Both training and validation loss are checked during training, and for some models early stopping is used when the validation loss stops decreasing. Training is done over multiple splits of data (15 for the MI animals dataset, 5 for MNIST and 6 for malware dataset) with the same hyperparameter combination. The data split is controlled with a random seed to ensure that all models are trained on the same data splits. Optimization is done using the ADAM optimizer [47] with a standard learning rate $\eta = 0.001$.

Hyperparameter tuning is done with random search. For each model, a random combination of hyperparameters is sampled from predefined possible values. The best hyperparameters are chosen after training based on validation accuracy. Experiments are repeated for 50 random combinations of hyperparameters. Test accuracy and other metrics are reported as results.

4.2.1 Model details

The models presented in this thesis vary greatly in complexity. This makes it hard to give each model a fair chance of learning the best representation of the data. In general, models were trained with regard to their architecture to ensure that each model has enough time and resources to train.

A simple classifier, as a well as classifier with triplet regularization, generally got shorter training times, and early stopping was used to stop training when the accuracy on validation data stopped improving. These two models took the shortest training time due to their simplicity.

Self-supervised models were trained in five *train/pseudo-label* iterations in total. After every iteration, the trained model was saved. Validation accuracy was reported for each of the five trained models, and the best model was chosen to represent the particular run. This setting ensured that a bad pseudo-labeling in later iteration would not completely destroy the model's performance.

Generative semi-supervised models were allocated more training time to ensure that the model is given the opportunity to learn both the classification network and the generative model. Validation accuracy was calculated during training epochs and used to save the best model and reduce overfitting. All Gaussian distributions are parameterized with neural networks dependent on input $\mu(\mathbf{x}), \sigma(\mathbf{x})$, and the variance is diagonal.

The generative model for the JSON malware dataset proved to be the most complex. In the end, the model was not composed of 18 models, but only 12, because the number of bags for certain keys was 0 or small enough to ignore it. Still, the model remains very complex and needs to be provided with the longest training time and the most allocated memory.

Neural Statistician has been used in the initial stage of experiments as the generative part of M2 model. However, the model did not improve the M2 model compared to the generative component as described in 3.3.1 and, as a consequence, the model is not used in the comparison.

One more model has been added to the list, later called the *Chamfer kNN* model. The model is simple; a distance matrix between bags is calculated with the Chamfer distance (2.2) and the kNN algorithm is used to predict the labels. The best k is chosen on the validation data and used for kNN on the test dataset. Chamfer kNN is only used on Animals MI dataset due to its infeasible computational complexity on larger datasets.

4.3 Evaluation

The evaluation of models in this thesis is two-fold. The first step is to calculate the accuracy of model predictions. The models are ranked on the basis of reported test accuracy, but train and validation accuracies are presented as well.

In the second part of evaluation, the focus is on the embedding learned through the HMill models in terms of clustering quality. The best models chosen in step one are used to project data into an embedding space and create a latent encoding of the data. Clustering is performed on the latent encoding and its UMAP representation. UMAP makes it possible to calculate a UMAP model on one subset of data, and use this model to project a second subset of data into the same embedding. This feature is used in two ways:

- calculate UMAP embedding on train data and use the model to project test data into the same embedding,
- calculate UMAP embedding on test data and use the model to project train data into the same embedding.

Both options have different properties and leverage different information. The second approach generally uses more data and can leverage the overall structure in an unsupervised manner. Projection in a 2D embedding space is used in all experiments.

Three clustering algorithms are compared: k-means, k-medoids, and hierarchical clustering with average linkage. The number of clusters is chosen to be multiples of the true number of classes: if there are 3 classes of data, clustering is calculated for $k = 3, 6, 9$. Clusters are assigned labels based on the distance of the cluster to training data. A centre of mass is calculated for each class in the training data and a cluster is assigned to the closest class centre. kNN is also used to infer labels

of test data in both the latent space and its UMAP projections. The best k is chosen on validation data.

The metrics described in 1.3.2 have been calculated for the clustering results. However, the metrics are sensitive to the number of clusters created and number of classes in the data. All metrics are generally lower for higher number of clusters, even though more but better separable clusters can be the aim of the researcher. For this reason, only accuracy of infered labels based on the process described in the previous paragraph is reported. The curious can later skip to Appendix A to see the full result tables for clustering algorithms (k-means, k-medoids, hierarchical) with the rand index and mutual information provided, as well as the number of clusters created.

4.4 Implementation

All experiments were run and evaluated using Julia Language [48]. Julia is an open-source programming language designed specifically for scientific computing and comes with numerous packages for machine learning, deep learning, or automatic differentiation.

The code used to carry out the experiments in this thesis is available as a repository on Github at `www.github.com/masenka31/master_thesis`. All models discussed have been implemented by the author with the use of core Julia libraries. The project is available as a package that can be installed and experiments can be run, if data are provided.

The most notable Julia libraries used are

- Mill.jl [29] – implementation of HMill models,
- JsonGrinder.jl [40] – processing of JSON files,
- Flux.jl [42] – deep learning library with Zygote.jl [49] for automatic differentiation,
- DrWatson.jl [50] – system for writing reproducible experimental code,
- BSON.jl, JLD2.jl – tools for data saving.

Experiments were run on a HPC cluster without GPU acceleration.

Chapter 5

Results

This chapter presents results obtained for implemented models on three datasets. Every section describes experimental results for one dataset in two parts – the first is used to discuss prediction accuracy of trained models, the second shows results for clustering on learned latent space. All results are collected in a table and plotted in a graph to provide a visual comparison.

What follows is a complete list of models used in experiments with a brief description and reference to the corresponding sections of the thesis, where these models are described in more detail. Some models are also provided with a shortened version of their names.

List of models:

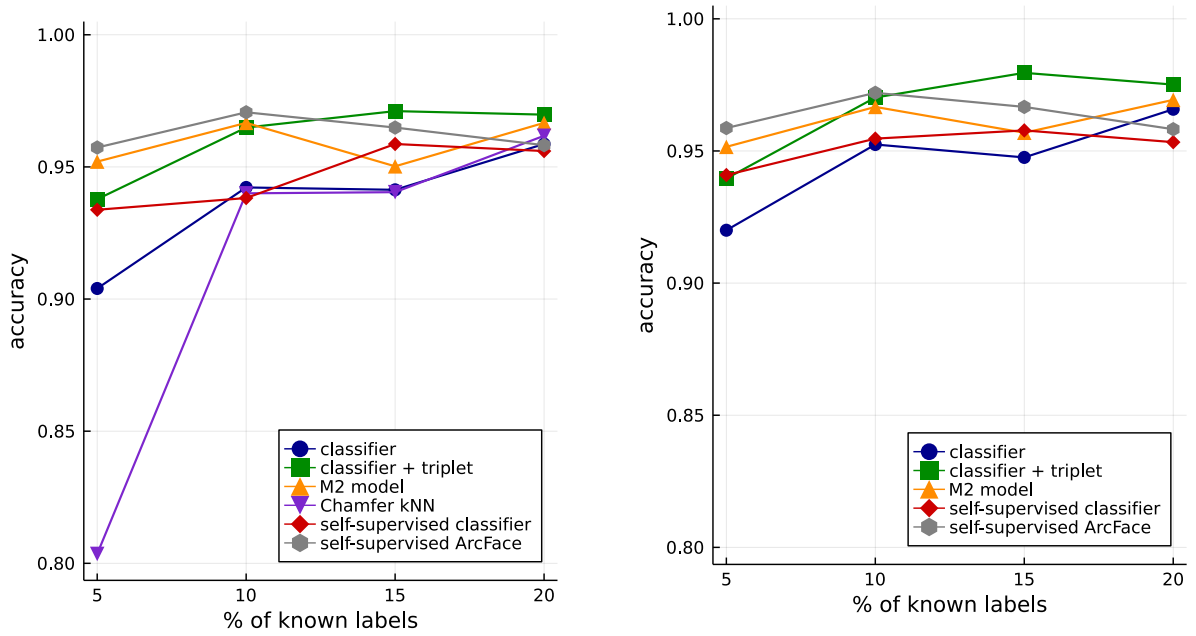
- classifier = standard supervised HMill classifier as described in Section 2.3,
- classifier + triplet = classifier with triplet regularization as presented in Section 3.2,
- self-classifier = self-supervised classifier and
- self-ArcFace = self-supervised ArcFace model (both presented in Section 1.1.3),
- M2 model = M2 model designed in Section 3.3.1,
- M2 + warm-up = the same M2 model, but first trained on labeled data, and only after a given number of epochs, unlabeled data are added to training,
- Chamfer kNN = a kNN performed on a distance matrix calculated with Chamfer distance.

Four clustering methods are used to cluster the embedding space of trained models:

- k-means,
- k-medoids,
- hierarchical with average linkage,
- k-nearest neighbors.

Clustering results for each dataset and model are calculated on three types of embedding:

- encoding – the real output of an HMill model,
- train embedding – 2D UMAP projection calculated on train data,
- test embedding – 2D UMAP projection calculated on test data.



(a) Classification accuracy for each model.

(b) Accuracy of infered labels from clustering on embedding space of given models.

Figure 5.1: Visualizations of accuracy on the Animals MI dataset based on the percentage of known labels.

5.1 Animals MI

Classification

Six models were examined for the Animals multi-instance problem. The results are presented in Table 5.1 and plotted in Figure 5.1a.

Self-supervised ArcFace model ended up with the highest accuracy for the lower percentages of labels (5, 10), followed by M2 model and a classifier with triplet regularization. The regularized classifier shows the best accuracy for higher percentages of known labels. Unfortunately, due to the small number of bags in the dataset, the comparison of accuracies lies at the boundary of significance.

Interestingly enough, we can see some signs of over-fitting on the data, especially for small percentages of labeled data. There are significant differences in validation and test accuracies for classifiers, kNN on Chamfer distance and self-supervised models. The gap between the two accuracies seems to be smaller for the generative M2 model, which becomes second best for 5 % of labeled data due to this property, even though the model is otherwise fourth in validation accuracy.

The confusion matrix of the predictions was also calculated. The class predicted with the highest accuracy inbetween models, seeds, and varying ratios, was Elephant with mostly 100% precision, followed by Tiger and Fox.

model	%	train accuracy	validation accuracy	test accuracy
classifier	5	1.0	0.938	0.904
classifier + triplet	5	1.0	0.978	0.938
M2 model	5	1.0	0.96	0.952
Chamfer kNN	5	1.0	0.858	0.804
self-classifier	5	1.0	0.991	0.934
self-ArcFace	5	1.0	0.987	**0.957**
classifier	10	1.0	0.958	0.942
classifier + triplet	10	1.0	0.982	0.965
M2 model	10	1.0	0.96	0.967
Chamfer kNN	10	1.0	0.967	0.94
self-classifier	10	1.0	0.989	0.938
self-ArcFace	10	1.0	0.996	**0.971**
classifier	15	1.0	0.966	0.941
classifier + triplet	15	1.0	0.984	**0.971**
M2 model	15	1.0	0.961	0.95
Chamfer kNN	15	1.0	0.963	0.94
self-classifier	15	1.0	0.99	0.959
self-ArcFace	15	1.0	0.99	0.965
classifier	20	1.0	0.968	0.959
classifier + triplet	20	1.0	0.984	**0.97**
M2 model	20	1.0	0.98	0.967
Chamfer kNN	20	1.0	0.987	0.962
self-classifier	20	1.0	0.99	0.956
self-ArcFace	20	1.0	0.979	0.958

Table 5.1: Classification accuracy for models trained on the Animals MI dataset based on the percentage of known labels.

Clustering

Clusterings were calculated in accordance with the process described in Section 4.3. For all models, except the Chamfer kNN model, the clustering quality of their latent space was examined. The Chamfer kNN model was not used, since the model is deterministic, not trainable, and does not learn an embedding. The best models chosen based on validation accuracy in the previous section were used to perform clustering on their learned embedding. The results are shown in Table 5.2 as well as in Figure 5.1b. The results are on par with the classification accuracy, sometimes the clustering accuracy achieves slightly better results.

There does not seem to be any visible trend for any clustering method or the type of latent space used (encoding vs UMAP representation), although the k-means and k-medoids algorithms seem to be best suited for this dataset.

model	method	type	accuracy	%
classifier	k-means	encoding	0.92	5
classifier + triplet	k-means	encoding	0.94	5
M2 model	hierarchical	encoding	0.952	5
self-supervised classifier	k-means	test embedding	0.941	5
self-supervised ArcFace	k-means	test embedding	**0.959**	5
classifier	hierarchical	test embedding	0.952	10
classifier + triplet	hierarchical	test embedding	0.97	10
M2 model	hierarchical	encoding	0.967	10
self-supervised classifier	k-means	test embedding	0.955	10
self-supervised ArcFace	k-means	train embedding	**0.972**	10
classifier	k-means	test embedding	0.948	15
classifier + triplet	k-means	test embedding	**0.98**	15
M2 model	k-means	train embedding	0.957	15
self-supervised classifier	k-medoids	encoding	0.958	15
self-supervised ArcFace	k-medoids	encoding	0.967	15
classifier	k-medoids	encoding	0.966	20
classifier + triplet	k-means	test embedding	**0.975**	20
M2 model	kNN	encoding	0.969	20
self-supervised classifier	kNN	test embedding	0.953	20
self-supervised ArcFace	k-means	train embedding	0.958	20

Table 5.2: Clustering results for models on Animals MI dataset. Columns method and type mark the setting of the best clustering algorithm.

5.2 MNIST

Classification

Six models have been trained on the MNIST point cloud datasets in two settings: a downsampled dataset of 4 digits (0,1,3,4) and the full 10-digit dataset. Table 5.3 shows the final accuracies of the best models chosen based on validation accuracy. The results are also visualized in Figure 5.2.

It is clear that the smaller dataset is much easier to learn and the accuracies are significantly higher for all models. There is a rising trend with more labels available for the model to train on. The classifier with triplet regularization proved to be the best model out of the models tested for both MNIST datasets.

Unfortunately, the generative semi-supervised M2 model could not compete with supervised classifiers and their self-supervised counterparts. A possible explanation is that for the model to work well, the reconstruction loss of the generative component needs to be dependent on the provided label. However, the generative model is able to learn reconstruction simply given the context and the latent variables, and probably learns to ignore the label given. One of the solutions that could help would be to make the generative model more dependent on a given label and remove the instance latent space to force the model to generalize better and create a good one-vector representation of the sample. It could also help to swap the generative component with a more powerful model such as SetVAE [51].

	4 digits				10 digits		
model	**train accuracy**	**val accuracy**	**test accuracy**	**%**	**train accuracy**	**val accuracy**	**test accuracy**
classifier	1.0	0.9	0.883	0.2	1.0	0.721	0.703
classifier + triplet	1.0	0.943	**0.91**	0.2	1.0	0.769	**0.743**
M2	0.971	0.864	0.815	0.2	0.851	0.636	0.589
M2 + warm-up	1.0	0.836	0.808	0.2	0.991	0.659	0.63
self-classifier	0.993	0.954	0.891	0.2	0.94	0.786	0.716
self-ArcFace	0.986	0.946	0.907	0.2	0.84	0.664	0.671
classifier	1.0	0.965	0.953	1	0.991	0.87	0.855
classifier + triplet	1.0	0.983	**0.972**	1	1.0	0.88	**0.877**
M2	0.99	0.92	0.901	1	0.897	0.764	0.745
M2 + warm-up	0.982	0.924	0.902	1	0.875	0.757	0.738
self-classifier	0.993	0.982	0.962	1	0.939	0.886	0.871
self-ArcFace	0.985	0.987	0.967	1	0.803	0.72	0.729
classifier	0.998	0.983	0.979	5	1.0	0.92	0.92
classifier + triplet	1.0	0.987	**0.984**	5	0.997	0.932	**0.93**
M2	0.952	0.932	0.932	5	0.779	0.755	0.763
M2 + warm-up	0.97	0.946	0.941	5	0.791	0.766	0.772
self-classifier	0.996	0.989	**0.984**	5	0.954	0.917	0.915
self-ArcFace	0.993	0.98	0.981	5	0.653	0.611	0.623
classifier	0.999	0.989	0.985	10	0.988	0.938	0.934
classifier + triplet	0.999	0.991	**0.988**	10	0.972	0.943	**0.94**
M2	0.961	0.952	0.946	10	0.73	0.723	0.729
M2 + warm-up	0.959	0.95	0.945	10	0.768	0.76	0.764
self-classifier	0.994	0.99	0.985	10	0.948	0.928	0.926
self-ArcFace	0.997	0.989	0.986	10	0.798	0.65	0.659

Table 5.3: Classification accuracies for MNIST point cloud dataset in two settings (4 and 10 digits) for six models and four percentages of available labeled data.

The self-supervised ArcFace model failed on the full MNIST dataset. There are multiple reasons for the low accuracy. The most probable explanation is that the higher number of classes makes it harder to create a discrete embedding of the data using ArcFace. Therefore, in every iteration of the model, a lot of wrongfully labeled samples might be added to training data and the distribution does not coincide with the true data distribution $p(\mathbf{x} \mid y)$ anymore. One solution might be to use better mining techniques to use better batches of data during training, another would be to be more careful when adding unlabeled samples to train data and use a more strict rule for labeling unlabeled samples.

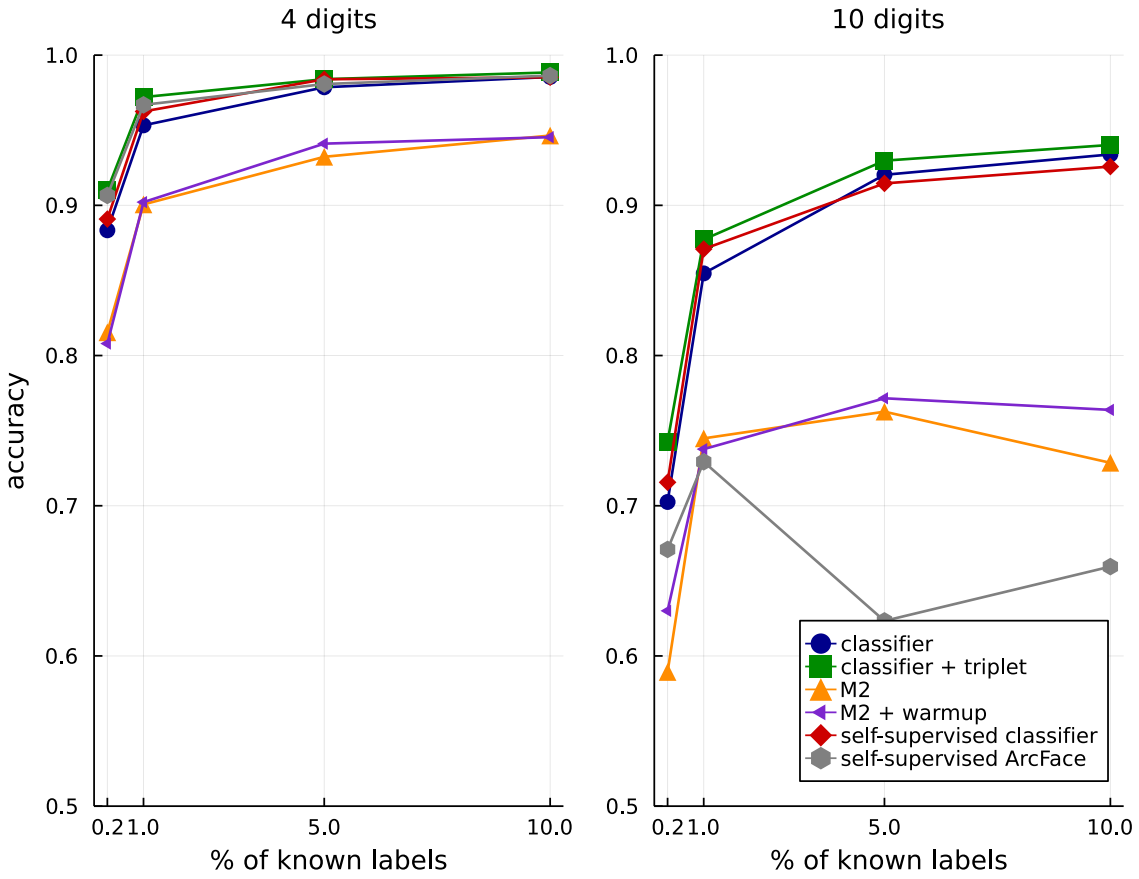


Figure 5.2: Comparison of classification accuracy for six models in two settings of the dataset given the percentage of data with known labels.

		4 digits				10 digits	
model	**method**	**type**	**accuracy**	**%**	**method**	**type**	**accuracy**
classifier	hierarchical	test emb.	0.912	0.2	kNN	encoding	0.647
classifier + triplet	k-medoids	test emb.	**0.941**	0.2	kNN	test emb.	**0.765**
M2 model	kNN	test emb.	0.835	0.2	kNN	encoding	0.569
M2 model + warm-up	kNN	test emb.	0.841	0.2	kNN	encoding	0.608
self-classifier	hierarchical	test emb.	0.925	0.2	kNN	test emb.	0.707
self-ArcFace	kNN	test emb.	0.908	0.2	kNN	encoding	0.678
classifier	kNN	test emb.	0.96	1	kNN	encoding	0.807
classifier + triplet	k-means	test emb.	**0.976**	1	kNN	test emb.	**0.877**
M2 model	kNN	test emb.	0.907	1	kNN	encoding	0.712
M2 model + warm-up	kNN	test emb.	0.911	1	kNN	encoding	0.712
self-classifier	k-means	test emb.	0.968	1	kNN	test emb.	0.851
self-ArcFace	kNN	encoding	0.967	1	kNN	encoding	0.725
classifier	kNN	test emb.	0.976	5	kNN	encoding	0.891
classifier + triplet	kNN	test emb.	**0.986**	5	kNN	encoding	**0.928**
M2 model	kNN	encoding	0.932	5	kNN	encoding	0.753
M2 model + warm-up	kNN	encoding	0.938	5	kNN	encoding	0.765
self-classifier	kNN	test emb.	0.982	5	kNN	encoding	0.894
self-ArcFace	kNN	encoding	0.98	5	kNN	encoding	0.636
classifier	kNN	encoding	0.982	0.1	kNN	encoding	0.913
classifier + triplet	kNN	encoding	**0.988**	0.1	kNN	encoding	**0.932**
M2 model	kNN	encoding	0.946	0.1	kNN	encoding	0.746
M2 model + warm-up	kNN	encoding	0.945	0.1	kNN	encoding	0.758
self-classifier	kNN	test emb.	0.984	0.1	kNN	encoding	0.908
self-ArcFace	kNN	test emb.	0.986	0.1	kNN	encoding	0.664

Table 5.4: Clustering accuracy on learned embedding for MNIST dataset (4 and 10 digits). Columns method and type mark the setting of the best clustering algorithm.

Clustering

The clustering accuracy was calculated for the best model chosen previously and every clustering algorithm was applied with varying number of clusters, or choosing the best k on validation data for the case of kNN.

It is clearly seen in Table 5.4 and Figure 5.3 that the best method for labeling test data in the learned embedding space is the k-nearest neighbors algorithm. For the downsampled dataset, the best encoding seemed to be the UMAP embedding calculated on test data, where the UMAP algorithm is able to create a good representation of the high-dimensional embedding. On the full dataset, the test embedding was chosen as the best only in four cases. The higher number of classes are better distinguishable by using the high-dimensional embedding (encoding) created by the HMill models.

The best model based on the quality of learned embedding is the classifier with triplet loss regularization. Regularization proved to greatly improve the results compared to a simple supervised classifier.

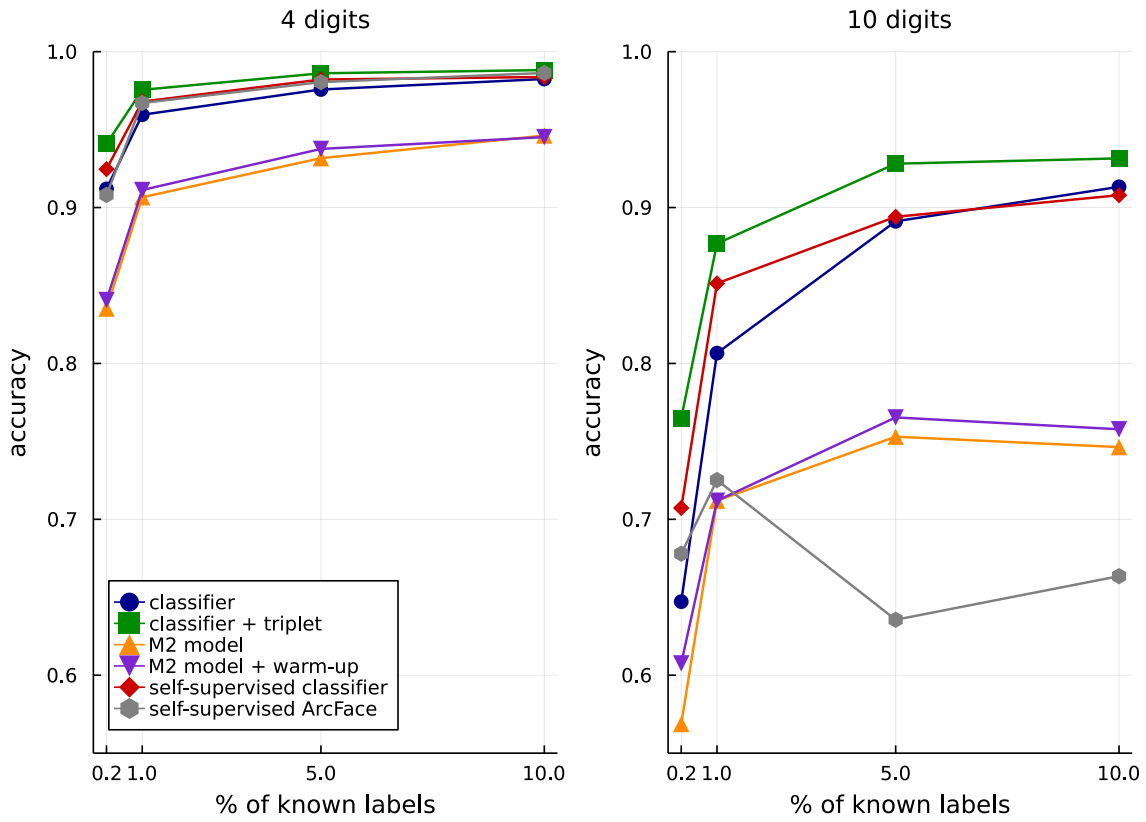


Figure 5.3: Comparison of clustering accuracy for six models in two settings of the dataset given the percentage of data with known labels.

model	%	train accuracy	validation accuracy	test accuracy
classifier	1	1.0	0.879	0.863
triplet classifier	1	1.0	0.892	**0.866**
self-supervised classifier	1	0.992	0.902	**0.866**
self-supervised ArcFace	1	0.99	0.869	0.848
M2 model	1	1.0	0.883	0.838
classifier	2	0.999	0.917	0.904
triplet classifier	2	1.0	0.923	0.902
self-supervised classifier	2	0.992	0.938	**0.915**
self-supervised ArcFace	2	0.994	0.914	0.898
M2 model	2	0.998	0.909	0.89
classifier	5	0.999	0.949	0.952
triplet classifier	5	1.0	0.954	0.952
self-supervised classifier	5	0.985	0.96	**0.955**
self-supervised ArcFace	5	0.991	0.948	**0.955**
M2 model	5	1.0	0.944	0.942
classifier	10	0.999	0.97	0.966
triplet classifier	10	0.998	0.97	0.968
self-supervised classifier	10	0.999	0.973	**0.969**
self-supervised ArcFace	10	0.997	0.973	**0.969**
M2 model	10	0.996	0.96	0.959
classifier	20	0.998	0.982	0.978
triplet classifier	20	1.0	0.982	0.978
self-supervised classifier	20	0.998	0.986	**0.981**
self-supervised ArcFace	20	0.999	0.981	0.977
M2 model	20	0.987	0.973	0.968

Table 5.5: Results for JSON malware dataset for five models and five percentages of known labels.

5.3 JSON malware dataset

Classification

The models for the JSON malware dataset were run for 5 percentages of available labels. The classification results can be seen in Figure 5.4 and Table 5.5.

The test accuracies are mostly the same for all models, except the generative M2 model. There is a clear rising trend for growing number of labels, as expected. For 20 % of known labels, the models are able to achieve 10 % higher accuracy than for 1 % of labels provided.

The self-supervised models seem to have an edge over the classifier and classifier with triplet regularization. The self-supervised classifier achieves the best accuracy for small percentages of known labels and stays high even for more labels available.

The generative M2 model was not able to beat any other model. However, we should keep in mind, that the model is trained on only one key in the JSON structure and therefore misses some information. It is not clear if this is the reason for the model to be suboptimal, but further research is needed and a more powerfull model needs to be designed.

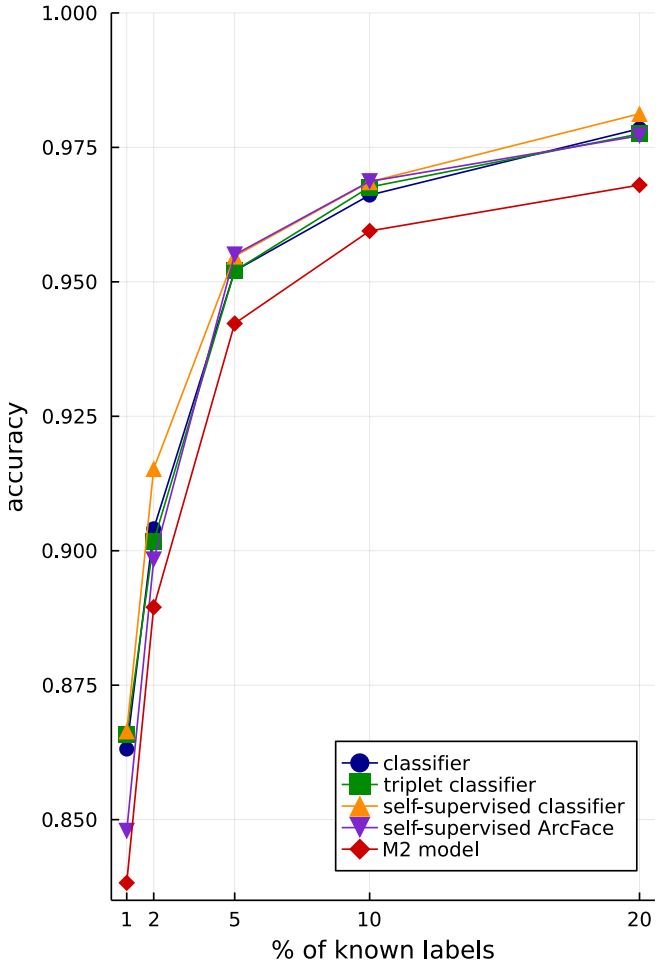


Figure 5.4: Comparison of classification accuracy for five models on the JSON malware dataset.

model	method	type	accuracy	%
classifier	k-means	encoding	0.86	1
classifier + triplet	kNN	encoding	0.862	1
self-supervised classifier	kNN	encoding	**0.868**	1
self-supervised ArcFace	kNN	encoding	0.851	1
M2 model	kNN	encoding	0.833	1
classifier	k-medoids	encoding	0.905	2
classifier + triplet	kNN	encoding	0.899	2
self-supervised classifier	kNN	encoding	**0.913**	2
self-supervised ArcFace	kNN	encoding	0.899	2
M2 model	kNN	encoding	0.893	2
classifier	kNN	encoding	0.951	5
classifier + triplet	k-means	train embedding	0.951	5
self-supervised classifier	kNN	encoding	0.953	5
self-supervised ArcFace	kNN	encoding	**0.957**	5
M2 model	kNN	encoding	0.938	5
classifier	kNN	encoding	0.967	10
classifier + triplet	kNN	encoding	0.966	10
self-supervised classifier	kNN	encoding	**0.971**	10
self-supervised ArcFace	kNN	encoding	0.97	10
M2 model	kNN	encoding	0.96	10
classifier	kNN	encoding	0.977	20
classifier + triplet	kNN	encoding	0.976	20
self-supervised classifier	kNN	encoding	**0.981**	20
self-supervised ArcFace	kNN	train embedding	0.977	20
M2 model	kNN	encoding	0.972	20

Table 5.6: Clustering accuracies for models on the JSON malware dataset. Columns method and type mark the setting of the best clustering algorithm.

Clustering

The clustering accuracies summarized in Table 5.6 improve with higher number of labels available, as expected. The differences between various models are not large. The M2 model ended with the lowest accuracy, but otherwise there is no clear ranking of the models. The self-supervised classifier takes the first spot for four percentages of labels, but usually by only a small margin.

The best method for classifying data based on the embedding space is by far the k-nearest neighbor algorithm, which shows that the closest neighbors are better for classification on the learned embedding than standard clustering methods, since generally the number of neighbors k chosen on the validation set is lower (mainly 1-5) and does not need a full cluster structure.

5.4 Summary

In summary, the aim of this thesis was to answer two main questions:

1. Are semi-supervised methods better than supervised for small number of labels available during training?
2. Does regularization of the latent space improve classification accuracy?

The answer to the first question is somewhat complicated. Yes, simple semi-supervised models (self-supervised classifier and self-supervised ArcFace model) were able to achieve good results but did not show considerable improvement over supervised classifiers. On the other hand, semi-supervised generative models failed to show any improvement and generally, with the exception of the Animals MI dataset, returned much lower accuracies than all other models. The generative part of the M2 model might be at fault, since the modeling of sets still relies on latent space of instances, and the label provided to the model might not be used well in the generative process. There is definitely room for improvement and further testing.

The answer to the second question is clear. Yes, the results show that regularization of the embedding helps the model in training, and a classifier with triplet regularization is able to achieve both higher classification accuracy and accuracy of clustering in the embedding space. The results pose a question, if a combination of triplet regularization and semi-supervised methods can bring even better accuracies. The answer to that question remains for the future.

One of the main takeaways from clustering results is that kNN works best for MNIST point cloud and JSON malware dataset. It shows that the learned embedding space is not discriminative enough to be easily clustered with standard algorithms such as k-means/medoids or hierarchical clustering, but is able to represent point similarity to allow for good k-nearest neighbor classification.

The reader is also encouraged to go through the Appendix for more detailed results. The Appendix A shows the best clustering methods when kNN is not used for label inference in the embedding space. Appendix B includes a nice visualization of the comparison between the clustering accuracy of a simple classifier and the classifier with triplet regularization on the MNIST point cloud dataset.

Conclusion

The thesis demonstrated the usage of semi-supervised learning, metric learning, and generative modeling in a new data paradigm of heterogenous structured data such as JSON files.

The first chapter presented the theoretical foundations of semi-supervised learning, reviewed assumptions, and described standard methods for vector data. Since not all data come in the form of numbers in the real space, metric learning provided methods for learning a similarity function for more complex data structures with contrastive loss, triplet loss, or the ArcFace architecture. A similarity metric can serve as a distance function that allows for clustering on the learned embedding. For this reason, an overview of clustering algorithms is presented with the addition of k-nearest neighbor algorithm and the UMAP algorithm for manifold approximation.

The second chapter described the challenges of modeling complex real-world data and explained the motivation for a general and flexible framework to process such data and design the corresponding machine learning models. The HMill framework is presented as a library that is able to process data in the form of bags, or tree-like structures such as JSON files.

The first two chapters were joined in the third chapter, where the standard models are extended to work in the HMill ecosystem. The models described include self-trained classifiers, self-trained ArcFace model, and extension of a generative semi-supervised model to group data. The last piece of the puzzle is a generative model for a specific JSON dataset. The model is designed to be a collection of generative submodels for leafs in the JSON structure and joined with a classifier to create a hierarchical version of the semi-supervised M2 model.

Experiments were set up in the fourth chapter. The three datasets used in the comparative study are presented there. The emphasis is placed on a fair comparison of models. Standard procedures for train, validation, and test splits are followed, hyperparameters are optimized with a random search and the best combination is chosen based on validation accuracy. Each model is trained over multiple splits of data and average accuracies are reported.

The results chapter presented experimental comparison of supervised, self-supervised, and semi-supervised methods for classification. The general take-away is that semi-supervised learning has the potential of improvement over standard supervised classifier, if properly trained and set up. The self-supervised models – classifier and ArcFace model – were able to compete with other models, with the exception of the self-supervised ArcFace model on the full MNIST dataset with 10 digits.

Unfortunately, the M2 generative semi-supervised model fell short of expectations. The model was only able to achieve the highest accuracy for the Animals multi-instance dataset, where the number of bags was relatively small (300). For MNIST and JSON malware dataset, the model's performance was significantly lower than that of other models. The problem might be that the generative model learns an instance embedding and can reconstruct the data well even when ignoring the label provided with them.

The embedding space of trained models has been analyzed as well. The clustering results

from the two larger datasets, MNIST point cloud and JSON data, give important insights about the data and model's embedding properties. kNN worked best on these datasets, implying that the space created is good at representing similarity with nearest neighbors having the same class, but not discriminative enough to create distinct cluster that would be detected with k-means or other clustering algorithms. The classifier with triplet regularization reported the highest clustering accuracy, showing that regularization does indeed create a higher-quality embedding space.

Further research should focus on designing a better generative model for data, as well as extension to multi-level tree-structured data to allow generation of whole JSON files. The reconstruction needs to be better tied to the provided label to ensure that the reconstruction quality is highest for a true label given. Such improvements should result in an improvement in the semi-supervised setting, since the models should be able to better represent data with unknown labels.

Another area of research in this domain can also focus on designing and testing possible data augmentations. Image augmentation helps tremendously in the field of computer vision to create and train robust models of high quality, and the ideas should be transferable to the domain of heterogenous structured data as well.

Bibliography

[1] Grand View Research, Inc. "Data collection and labeling market size report, 2021-2028." (2021), [Online]. Available: https://www.grandviewresearch.com/industry-analysis/data-collection-labeling-market.

[2] J. E. van Engelen and H. H. Hoos, "A survey on semi-supervised learning," *Machine Learning*, vol. 109, no. 2, pp. 373–440, Feb. 1, 2020, ISSN: 1573-0565. DOI: 10.1007/s10994-019-05855-6. [Online]. Available: https://doi.org/10.1007/s10994-019-05855-6.

[3] D. Yarowsky, "Unsupervised word sense disambiguation rivaling supervised methods," in *33rd Annual Meeting of the Association for Computational Linguistics*, Cambridge, Massachusetts, USA: Association for Computational Linguistics, 1995, pp. 189–196. DOI: 10.3115/981658.981684. [Online]. Available: https://aclanthology.org/P95-1026.

[4] I. Triguero, S. García, and F. Herrera, "Self-labeled techniques for semi-supervised learning: Taxonomy, software and empirical study," *Knowledge and Information Systems*, vol. 42, no. 2, pp. 245–284, Feb. 1, 2015, ISSN: 0219-3116. DOI: 10.1007/s10115-013-0706-y. [Online]. Available: https://doi.org/10.1007/s10115-013-0706-y.

[5] D. P. Kingma, S. Mohamed, D. Jimenez Rezende, and M. Welling, "Semi-supervised learning with deep generative models," in *Advances in Neural Information Processing Systems*, vol. 27, Curran Associates, Inc., 2014. [Online]. Available: https://proceedings.neurips.cc/paper/2014/hash/d523773c6b194f37b938d340d5d02232-Abstract.html.

[6] D. P. Kingma and M. Welling, "Auto-encoding variational bayes," *arXiv:1312.6114 [cs, stat]*, May 1, 2014. arXiv: 1312.6114. [Online]. Available: http://arxiv.org/abs/1312.6114.

[7] R. Hadsell, S. Chopra, and Y. LeCun, "Dimensionality reduction by learning an invariant mapping," in *2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'06)*, ISSN: 1063-6919, vol. 2, Jun. 2006, pp. 1735–1742. DOI: 10.1109/CVPR.2006.100.

[8] E. Hoffer and N. Ailon, "Deep metric learning using triplet network," in *Similarity-Based Pattern Recognition*, A. Feragen, M. Pelillo, and M. Loog, Eds., ser. Lecture Notes in Computer Science, Springer International Publishing, 2015, pp. 84–92, ISBN: 978-3-319-24261-3. DOI: 10.1007/978-3-319-24261-3_7.

[9] A. Hermans, L. Beyer, and B. Leibe, "In defense of the triplet loss for person re-identification," *arXiv preprint arXiv:1703.07737*, 2017.

[10] W. Liu, Y. Wen, Z. Yu, M. Li, B. Raj, and L. Song, "SphereFace: Deep hypersphere embedding for face recognition," presented at the Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2017, pp. 212–220. [Online]. Available: https://openaccess.thecvf.com/content_cvpr_2017/html/Liu_SphereFace_Deep_Hypersphere_CVPR_2017_paper.html.

[11] H. Wang, Y. Wang, Z. Zhou, *et al.*, "CosFace: Large margin cosine loss for deep face recognition," presented at the Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2018, pp. 5265–5274. [Online]. Available: https://openaccess.thecvf.com/content_cvpr_2018/html/Wang_CosFace_Large_Margin_CVPR_2018_paper.html.

[12] J. Deng, J. Guo, N. Xue, and S. Zafeiriou, "ArcFace: Additive angular margin loss for deep face recognition," presented at the Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2019, pp. 4690–4699. [Online]. Available: https://openaccess.thecvf.com/content_CVPR_2019/html/Deng_ArcFace_Additive_Angular_Margin_Loss_for_Deep_Face_Recognition_CVPR_2019_paper.html.

[13] R. Xu and D. Wunsch, "Survey of clustering algorithms," *IEEE Transactions on Neural Networks*, vol. 16, no. 3, pp. 645–678, May 2005, ISSN: 1941-0093. DOI: 10.1109/TNN.2005.845141.

[14] M. Bramer, *Principles of Data Mining*, ser. Undergraduate Topics in Computer Science. London: Springer London, 2016, pp. 314–320, ISBN: 978-1-4471-7307-6. DOI: 10.1007/978-1-4471-7307-6. [Online]. Available: http://link.springer.com/10.1007/978-1-4471-7307-6.

[15] T. S. Madhulatha, "Comparison between k-means and k-medoids clustering algorithms," in *Advances in Computing and Information Technology*, D. C. Wyld, M. Wozniak, N. Chaki, N. Meghanathan, and D. Nagamalai, Eds., ser. Communications in Computer and Information Science, Berlin, Heidelberg: Springer, 2011, pp. 472–481, ISBN: 978-3-642-22555-0. DOI: 10.1007/978-3-642-22555-0_48.

[16] P. Bhattacharjee and P. Mitra, "A survey of density based clustering algorithms," *Frontiers of Computer Science*, vol. 15, no. 1, p. 151 308, Sep. 29, 2020, ISSN: 2095-2236. DOI: 10.1007/s11704-019-9059-3. [Online]. Available: https://doi.org/10.1007/s11704-019-9059-3.

[17] M. Ester, H.-P. Kriegel, J. Sander, X. Xu, *et al.*, "A density-based algorithm for discovering clusters in large spatial databases with noise.," in *kdd*, vol. 96, 1996, pp. 226–231.

[18] L. Hubert and P. Arabie, "Comparing partitions," *Journal of classification*, vol. 2, no. 1, pp. 193–218, 1985.

[19] N. X. Vinh, J. Epps, and J. Bailey, "Information theoretic measures for clusterings comparison: Variants, properties, normalization and correction for chance," p. 18,

[20] P. J. Rousseeuw, "Silhouettes: A graphical aid to the interpretation and validation of cluster analysis," *Journal of computational and applied mathematics*, vol. 20, pp. 53–65, 1987.

[21] E. Fix and J. L. Hodges, "Discriminatory analysis. nonparametric discrimination: Consistency properties," *International Statistical Review/Revue Internationale de Statistique*, vol. 57, no. 3, pp. 238–247, 1989.

[22] N. S. Altman, "An introduction to kernel and nearest-neighbor nonparametric regression," *The American Statistician*, vol. 46, no. 3, pp. 175–185, 1992.

[23] L. McInnes, J. Healy, and J. Melville, "UMAP: Uniform manifold approximation and projection for dimension reduction," *arXiv preprint arXiv:1802.03426*, 2018.

[24] H. Hotelling, "Analysis of a complex of statistical variables into principal components.," *Journal of educational psychology*, vol. 24, no. 6, p. 417, 1933.

[25] L. Van der Maaten and G. Hinton, "Visualizing data using t-sne.," *Journal of machine learning research*, vol. 9, no. 11, 2008.

[26] D. Daudert, *UMAP.jl*, Mar. 22, 2022. [Online]. Available: `https://github.com/dillondaudert/UMAP.jl`.

[27] W. Dong, C. Moses, and K. Li, "Efficient k-nearest neighbor graph construction for generic similarity measures," in *Proceedings of the 20th international conference on World wide web*, 2011, pp. 577–586.

[28] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT press, 2016.

[29] T. Pevny and S. Mandlik, *Mill.jl framework: A flexible library for (hierarchical) multi-instance learning*, version 2.6.0, 2.7.1. [Online]. Available: `https://github.com/CTUAvastLab/Mill.jl`.

[30] S. Mandlik, M. Racinsky, V. Lisy, and T. Pevny, *Mill.jl and JsonGrinder.jl: Automated differentiable feature extraction for learning from raw json data*, 2021. arXiv: `2105.09107 [stat.ML]`.

[31] Š. Mandlík and T. Pevný, "Mapping the internet: Modelling entity interactions in complex heterogeneous networks," *arXiv:2104.09650 [cs]*, Apr. 19, 2021. arXiv: `2104.09650`. [Online]. Available: `http://arxiv.org/abs/2104.09650`.

[32] M. Zaheer, S. Kottur, S. Ravanbakhsh, B. Poczos, R. Salakhutdinov, and A. Smola, "Deep sets," *arXiv preprint arXiv:1703.06114*, 2017.

[33] Y. Zhang, J. Hare, and A. Prügel-Bennett, "Fspool: Learning set representations with featurewise sort pooling," *arXiv preprint arXiv:1906.02795*, 2019.

[34] H. Fan, H. Su, and L. J. Guibas, "A point set generation network for 3d object reconstruction from a single image," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 605–613.

[35] G. Yang, X. Huang, Z. Hao, M.-Y. Liu, S. Belongie, and B. Hariharan, "PointFlow: 3D point cloud generation with continuous normalizing flows," in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2019, pp. 4541–4550.

[36] T. G. Dietterich, R. H. Lathrop, and T. Lozano-Pérez, "Solving the multiple instance problem with axis-parallel rectangles," *Artificial intelligence*, vol. 89, no. 1-2, pp. 31–71, 1997.

[37] S. Safris. "A deep look at JSON vs. XML, part 1: The history of each," Toptal Engineering Blog. (2019), [Online]. Available: `https://www.toptal.com/web/json-vs-xml-part-1`.

[38] Stack Overflow. "Stack overflow trends." (2022), [Online]. Available: `https://insights.stackoverflow.com/trends?tags=json%5C%2Cxml%5C%2Ccsv%5C%2Csoap`.

[39] json.org. "JSON." (2022), [Online]. Available: `https://www.json.org/json-en.html`.

[40] T. Pevny and M. Racinsky, *Jsongrinder.jl: A flexible library for automated feature engineering and conversion of jsons to mill.jl structures*, version v2.2.0. [Online]. Available: `https://github.com/CTUAvastLab/JsonGrinder.jl`.

[41] Spotify. "Console | Spotify for Developers." (2022), [Online]. Available: `https://developer.spotify.com/console/`.

[42] M. Innes, E. Saba, K. Fischer, *et al.*, "Fashionable Modelling with Flux," *CoRR*, vol. abs/1811.01457, 2018. arXiv: `1811.01457`. [Online]. Available: `https://arxiv.org/abs/1811.01457`.

[43] H. Edwards and A. Storkey, "Towards a neural statistician," *arXiv preprint arXiv:1606.02185*, 2016.

[44] V. Cheplygina, D. M. Tax, and M. Loog, "Multiple instance learning with bag dissimilarities," *Pattern Recognition*, vol. 48, no. 1, pp. 264–275, 2015.

[45] Y. LeCun, C. Cortes, and C. Burges, "Mnist handwritten digit database," *ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist*, vol. 2, 2010.

[46] L. Theis, A. van den Oord, and M. Bethge, *A note on the evaluation of generative models*, 2016. arXiv: 1511.01844 [stat.ML].

[47] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014. arXiv: 1412.6980 [cs.LG].

[48] J. Bezanson, S. Karpinski, V. B. Shah, and A. Edelman, "Julia: A fast dynamic language for technical computing," *arXiv preprint arXiv:1209.5145*, 2012.

[49] M. Innes, "Don't Unroll Adjoint: Differentiating SSA-Form Programs," *CoRR*, vol. abs/1810.07951, 2018. arXiv: 1810.07951. [Online]. Available: http://arxiv.org/abs/1810.07951.

[50] G. Datseris, J. Isensee, S. Pech, and T. Gál, "DrWatson: The perfect sidekick for your scientific inquiries," *Journal of Open Source Software*, vol. 5, no. 54, p. 2673, 2020. DOI: 10.21105/joss.02673. [Online]. Available: https://doi.org/10.21105/joss.02673.

[51] J. Kim, J. Yoo, J. Lee, and S. Hong, "Setvae: Learning hierarchical composition for generative modeling of set-structured data," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2021, pp. 15 059–15 068.

[52] T. Cover and P. Hart, "Nearest neighbor pattern classification," *IEEE Transactions on Information Theory*, vol. 13, no. 1, pp. 21–27, 1967. DOI: 10.1109/TIT.1967.1053964.

[53] J. Davis and M. Goadrich, "The relationship between precision-recall and roc curves," in *Proceedings of the 23rd International Conference on Machine Learning*, ser. ICML '06, Pittsburgh, Pennsylvania, USA: Association for Computing Machinery, 2006, pp. 233–240, ISBN: 1595933832. DOI: 10.1145/1143844.1143874. [Online]. Available: https://doi.org/10.1145/1143844.1143874.

[54] V. Estivill-Castro and J. Yang, "Fast and robust general purpose clustering algorithms," in *PRICAI 2000 Topics in Artificial Intelligence*, R. Mizoguchi and J. Slaney, Eds., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 2000, pp. 208–218, ISBN: 978-3-540-44533-3. DOI: 10.1007/3-540-44533-1_24.

Appendix A

Clustering results

The kNN algorithm proved to be the best way to infer labels of test data for the MNIST point cloud dataset and JSON malware dataset. Unfortunately, that meant that the other algorithms performance got lost as only the best combination of method-embedding type was presented.

This supplementary material presents the best clustering algorithm chosen for each dataset and a given percentage of known labels provided, when kNN is excluded. Complementary to clustering accuracy, rand index (RI) and mutual information (MI) metrics are provided in the results tables as well. The number k represents the number of clusters the embedding was divided into prior to assigning clusters to true classes.

The results for the Animals MI dataset are presented in Table A.1. There are no surprising entries, the best results are mostly on UMAP embedding calculated on test data. A higher number of clusters created by the clustering algorithm also seems to generate better accuracy.

MNIST dataset presents interesting results especially for the full dataset with 10 digits. The results for 4 digits can be seen in Table A.2, the results for 10 digits in Table A.3. Both show substantial differences in accuracy between the standard classifier and the classifier with triplet regularization. The regularized classifier can achieve 10% higher accuracy, sometimes even more.

Finally, Table A.4 presents clustering results without kNN for the JSON malware dataset. The accuracies of different models are quite close, sometimes the standard classifier achieved higher accuracy than the regularized version.

model	method	type	accuracy	RI	MI	k	%
classifier	k-medoids	test embedding	0.922	0.815	0.607	6	5
classifier + triplet	k-means	test embedding	0.939	0.77	0.572	9	5
M2 model	hierarchical	encoding	0.952	0.905	0.753	9	5
self-supervised classifier	k-medoids	test embedding	0.936	0.829	0.638	6	5
self-supervised ArcFace	k-means	test embedding	**0.96**	0.834	0.676	6	5
classifier	hierarchical	test embedding	0.952	0.772	0.593	9	10
classifier + triplet	k-medoids	test embedding	0.969	0.843	0.703	6	10
M2 model	k-medoids	encoding	0.968	0.835	0.681	9	10
self-supervised classifier	k-means	test embedding	0.953	0.837	0.67	6	10
self-supervised ArcFace	k-means	test embedding	**0.973**	0.965	0.891	3	10
classifier	k-means	test embedding	0.946	0.931	0.804	3	15
classifier + triplet	k-means	test embedding	**0.98**	0.974	0.919	3	15
M2 model	k-means	train embedding	0.957	0.945	0.838	3	15
self-supervised classifier	k-means	test embedding	0.959	0.948	0.843	3	15
self-supervised ArcFace	k-medoids	encoding	0.966	0.85	0.693	9	15
classifier	k-medoids	encoding	0.964	0.954	0.86	3	20
classifier + triplet	k-means	test embedding	**0.975**	0.967	0.904	3	20
M2 model	k-medoids	encoding	0.969	0.904	0.781	6	20
self-supervised classifier	hierarchical	test embedding	0.938	0.773	0.589	9	20
self-supervised ArcFace	k-means	train embedding	0.958	0.946	0.843	3	20

Table A.1: Clustering results without kNN for the Animals MI dataset. Columns method, type and k mark the setting of the best clustering algorithm.

model	method	type	accuracy	RI	MI	k	%
classifier	hierarchical	test embedding	0.912	0.823	0.584	12	0.2
classifier + triplet	k-medoids	test embedding	**0.941**	0.862	0.67	8	0.2
self-supervised classifier	hierarchical	test embedding	0.925	0.858	0.642	8	0.2
M2 model	hierarchical	test embedding	0.805	0.798	0.47	12	0.2
M2 model + warm-up	k-means	test embedding	0.787	0.799	0.47	12	0.2
classifier	k-means	test embedding	0.913	0.87	0.692	8	1
classifier + triplet	k-means	test embedding	**0.976**	0.976	0.904	4	1
self-supervised classifier	k-means	test embedding	0.968	0.969	0.879	4	1
M2 model	k-medoids	test embedding	0.87	0.843	0.586	8	1
M2 model + warm-up	k-medoids	encoding	0.839	0.821	0.529	8	1
classifier	hierarchical	test embedding	0.975	0.976	0.904	4	5
classifier + triplet	k-means	test embedding	**0.983**	0.983	0.933	4	5
self-supervised classifier	k-means	test embedding	0.981	0.981	0.922	4	5
M2 model	hierarchical	train embedding	0.917	0.923	0.754	4	5
M2 model + warm-up	hierarchical	test embedding	0.875	0.929	0.781	4	5
classifier	hierarchical	train embedding	0.978	0.978	0.912	4	10
classifier + triplet	hierarchical	test embedding	**0.986**	0.986	0.941	4	10
self-supervised classifier	hierarchical	test embedding	0.981	0.981	0.925	4	10
M2 model	k-means	test embedding	0.936	0.94	0.8	4	10
M2 model + warm-up	hierarchical	train embedding	0.929	0.934	0.786	4	10

Table A.2: Clustering results without kNN for MNIST with 4 digits. Columns method, type and k mark the setting of the best clustering algorithm.

model	**method**	**type**	**accuracy**	**RI**	**MI**	**k**	**%**
classifier	k-medoids	encoding	0.513	0.895	0.391	30	0.2
classifier + triplet	hierarchical	test embedding	**0.732**	0.923	0.601	20	0.2
self-supervised classifier	k-means	test embedding	0.602	0.908	0.508	30	0.2
M2 model	k-medoids	encoding	0.461	0.892	0.367	30	0.2
M2 model + warm-up	k-medoids	encoding	0.501	0.895	0.378	30	0.2
classifier	k-medoids	encoding	0.604	0.901	0.453	30	1
classifier + triplet	hierarchical	train embedding	**0.822**	0.948	0.726	20	1
self-supervised classifier	k-medoids	encoding	0.734	0.911	0.535	30	1
M2 model	k-medoids	encoding	0.554	0.898	0.409	30	1
M2 model + warm-up	k-medoids	encoding	0.593	0.899	0.43	30	1
classifier	k-medoids	encoding	0.631	0.899	0.482	20	5
classifier + triplet	hierarchical	test embedding	**0.882**	0.968	0.826	10	5
self-supervised classifier	k-medoids	test embedding	0.712	0.918	0.726	10	5
M2 model	k-means	encoding	0.525	0.891	0.408	20	5
M2 model + warm-up	k-medoids	encoding	0.597	0.902	0.458	30	5
classifier	k-medoids	encoding	0.712	0.91	0.531	30	10
classifier + triplet	hierarchical	test embedding	**0.862**	0.965	0.82	10	10
self-supervised classifier	k-medoids	encoding	0.682	0.909	0.525	30	10
M2 model	k-medoids	encoding	0.541	0.892	0.422	20	10
M2 model + warm-up	k-medoids	encoding	0.548	0.898	0.418	30	10

Table A.3: Clustering results without kNN for MNIST with 10 digits. Columns method, type and k mark the setting of the best clustering algorithm.

model	method	type	accuracy	randindex	MI	k	%
classifier	k-means	encoding	**0.86**	0.952	0.797	20	1
classifier + triplet	k-medoids	encoding	0.858	0.951	0.774	30	1
self-supervised classifier	k-means	encoding	0.857	0.953	0.786	30	1
self-supervised ArcFace	k-medoids	encoding	0.851	0.946	0.762	30	1
M2 model	k-medoids	encoding	0.825	0.944	0.757	30	1
classifier	k-medoids	encoding	**0.905**	0.948	0.778	30	2
classifier + triplet	k-medoids	train embedding	0.898	0.929	0.718	30	2
self-supervised classifier	k-means	encoding	0.893	0.953	0.792	30	2
self-supervised ArcFace	k-medoids	train embedding	0.892	0.928	0.718	30	2
M2 model	k-medoids	encoding	0.856	0.941	0.762	30	2
classifier	hierarchical	train embedding	0.94	0.962	0.852	20	5
classifier + triplet	k-means	train embedding	0.951	0.939	0.778	30	5
self-supervised classifier	k-means	encoding	0.932	0.955	0.812	30	5
self-supervised ArcFace	hierarchical	encoding	**0.956**	0.984	0.928	10	5
M2 model	hierarchical	train embedding	0.926	0.944	0.781	30	5
classifier	k-medoids	encoding	0.964	0.951	0.812	30	10
classifier + triplet	hierarchical	encoding	0.966	0.986	0.931	30	10
self-supervised classifier	k-medoids	encoding	**0.968**	0.966	0.863	20	10
self-supervised ArcFace	hierarchical	encoding	0.967	0.986	0.933	20	10
M2 model	k-means	encoding	0.903	0.952	0.811	30	10
classifier	hierarchical	encoding	0.97	0.983	0.924	20	20
classifier + triplet	k-medoids	encoding	0.974	0.967	0.855	30	20
self-supervised classifier	hierarchical	encoding	**0.976**	0.984	0.926	20	20
self-supervised ArcFace	k-medoids	encoding	0.975	0.961	0.842	30	20
M2 model	k-medoids	encoding	0.874	0.952	0.827	20	20

Table A.4: Clustering result for JSON malware dataset. Columns method, type and k mark the setting of the best clustering algorithm.

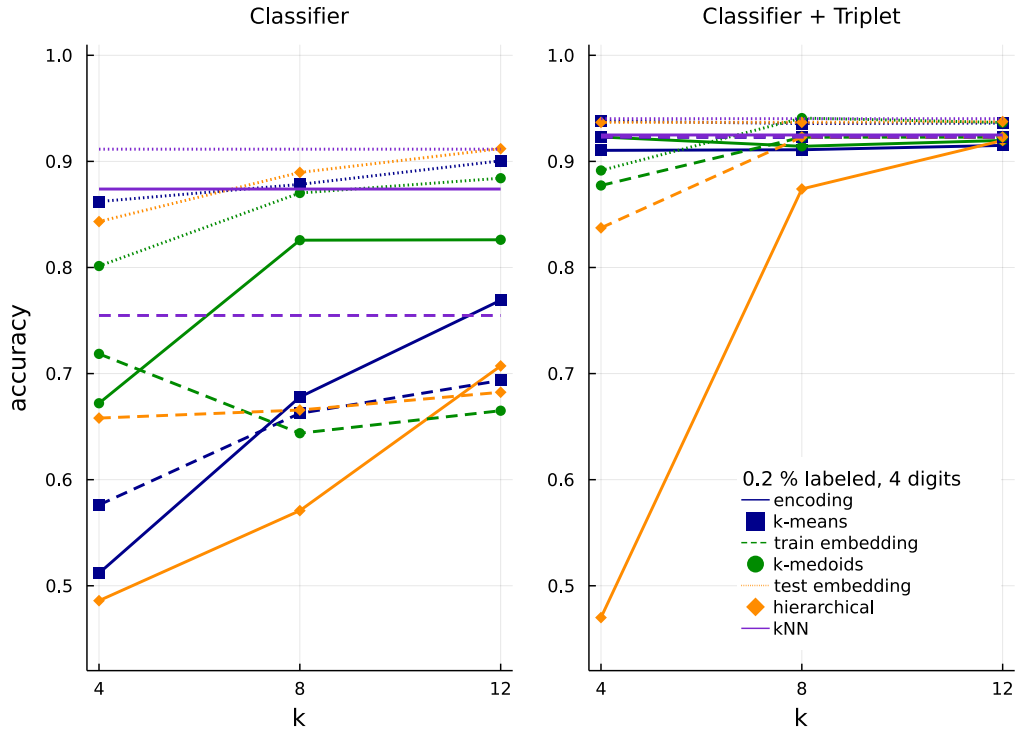
Appendix B

Comparison of classifiers on MNIST

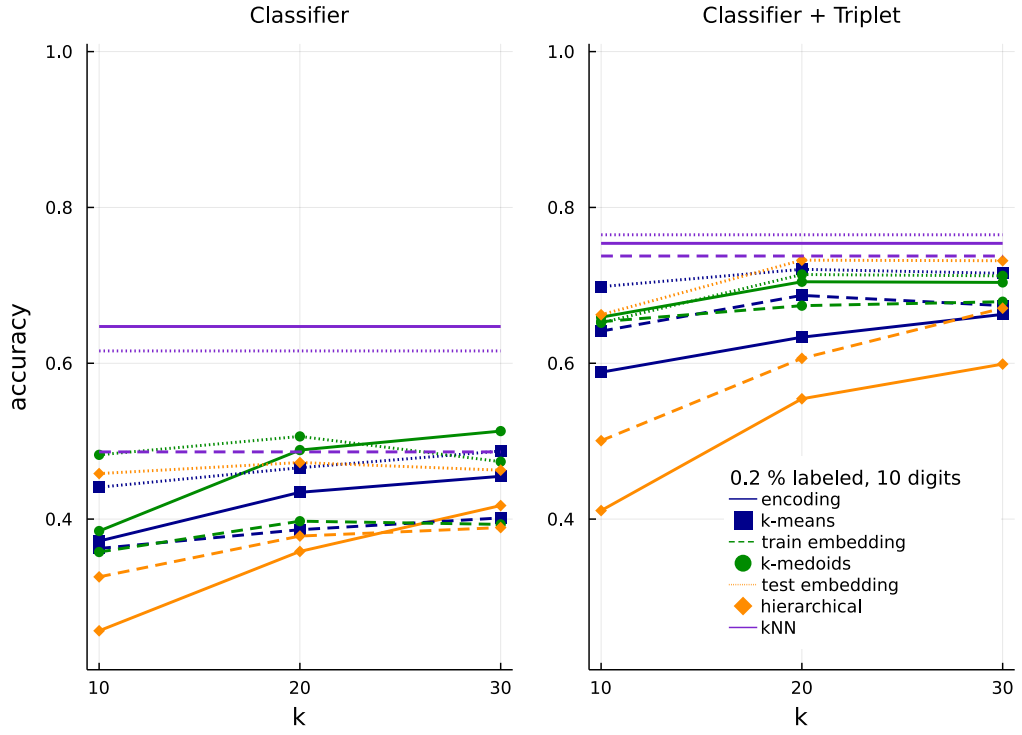
The results presented in Chapter 5 showed that triplet regularization was able to aid a classifier and result in improvement of both classification and clustering accuracy. The results presented only showed the best clustering algorithm, but the details and comparison of the two models are interesting enough to be presented in this supplementary material. The clustering properties for different algorithms as well as different numbers of clusters created are presented here for the MNIST dataset.

The following pictures show the comparison of accuracies of label inference for k-means, k-medoids, hierarchical clustering with average linkage, and kNN algorithm on two versions of the MNIST point cloud dataset.

Figures B.1, B.2, B.3, and B.4 clearly show that the triplet regularization improves the results across different clustering algorithms and embeddings. It also seems that the accuracy rises with the number of clusters created for encoding space, but tends to decrease on the UMAP embeddings. The superior performance of kNN is clearly visible in the case of the standard classifier, where sometimes the margin between kNN and the best accuracy of other clustering algorithms can exceed 10 %.

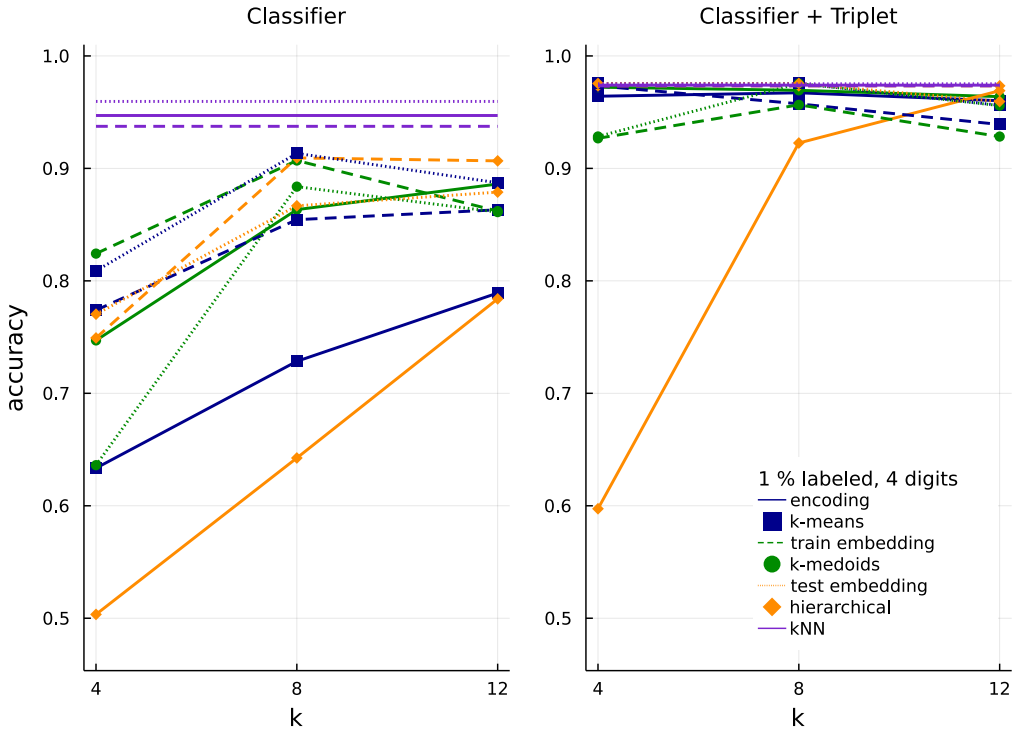


(a) Comparison for 0.2 % of labeled data, 4 MNIST digits.

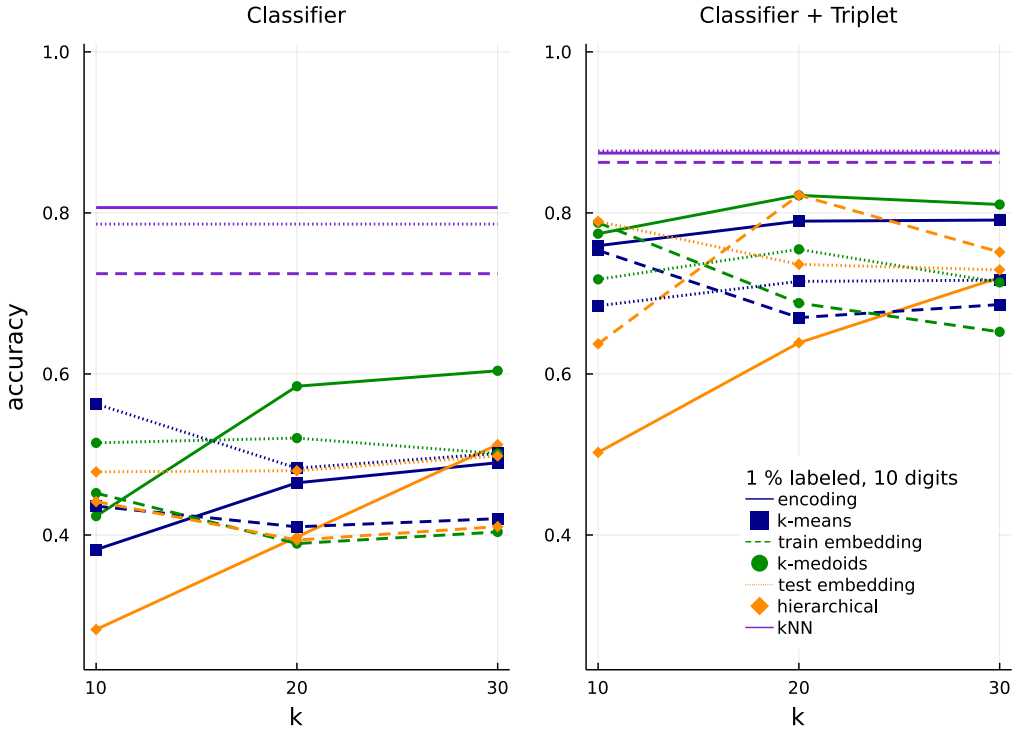


(b) Comparison for 0.2 % of labeled data, 10 MNIST digits.

Figure B.1: Comparison of clustering accuracy on latent space of classifiers with and without triplet regularization; 0.2 % of data labeled.

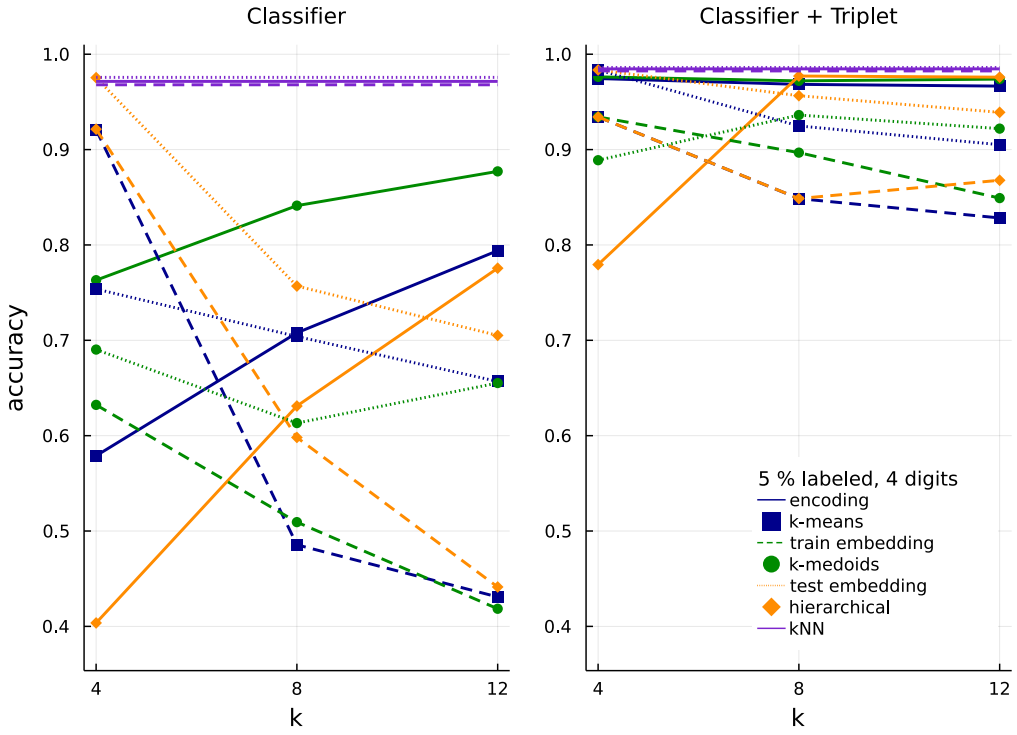


(a) Comparison for 1 % of labeled data, 4 MNIST digits.

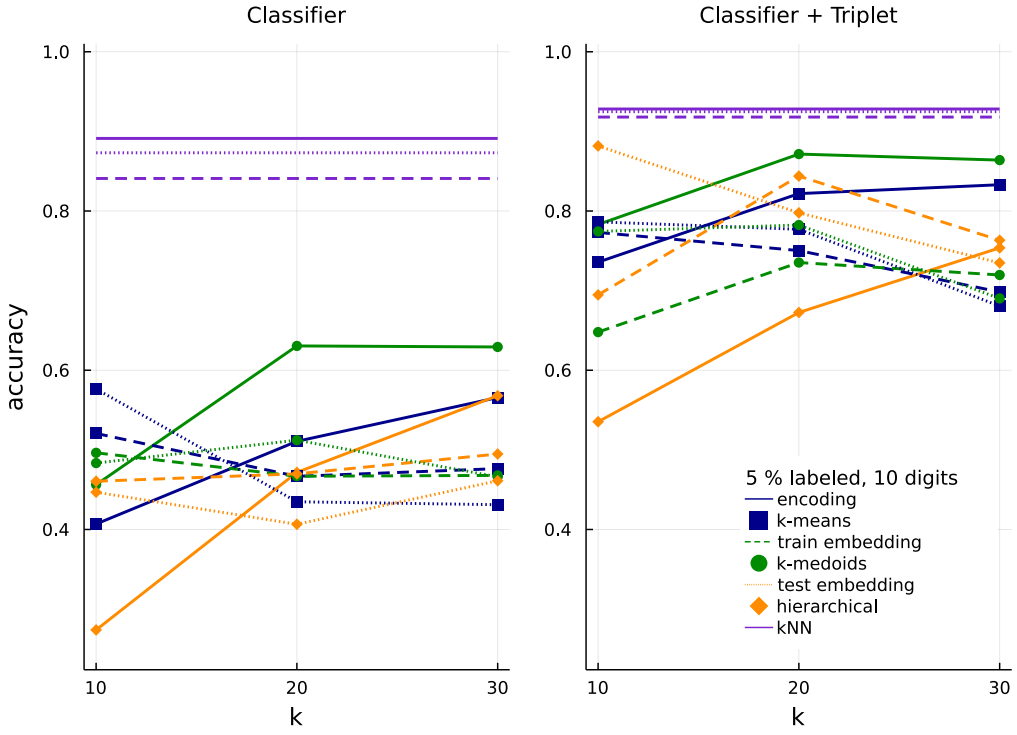


(b) Comparison for 1 % of labeled data, 10 MNIST digits.

Figure B.2: Comparison of clustering accuracy on latent space of classifiers with and without triplet regularization; 1 % of data labeled.

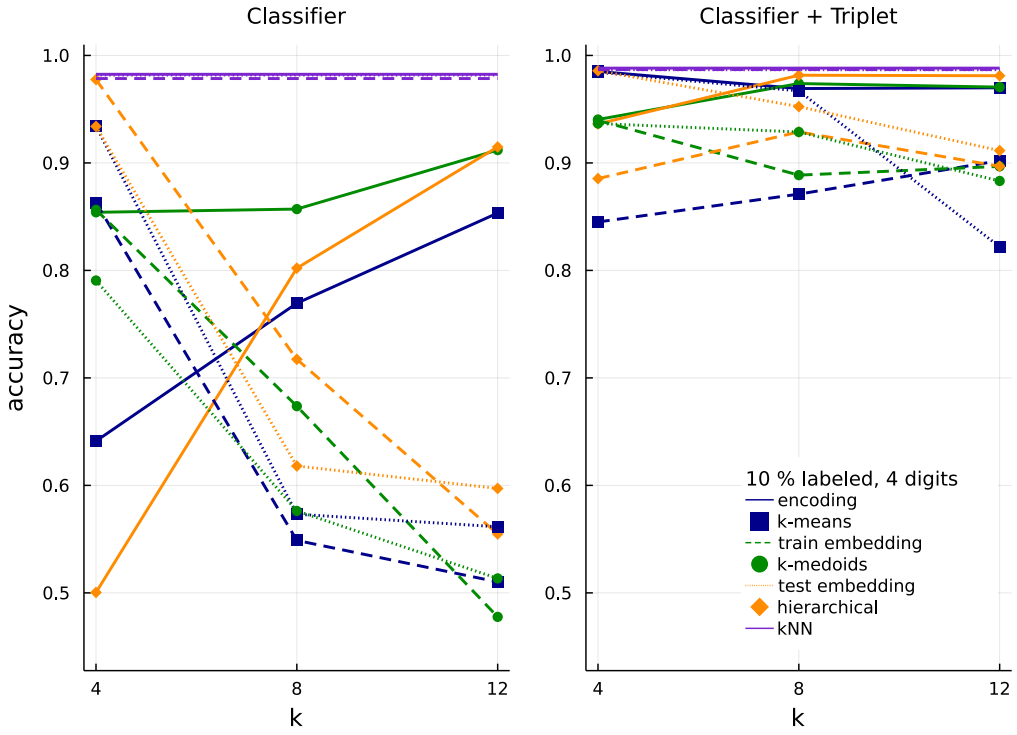


(a) Comparison for 5 % of labeled data, 4 MNIST digits.

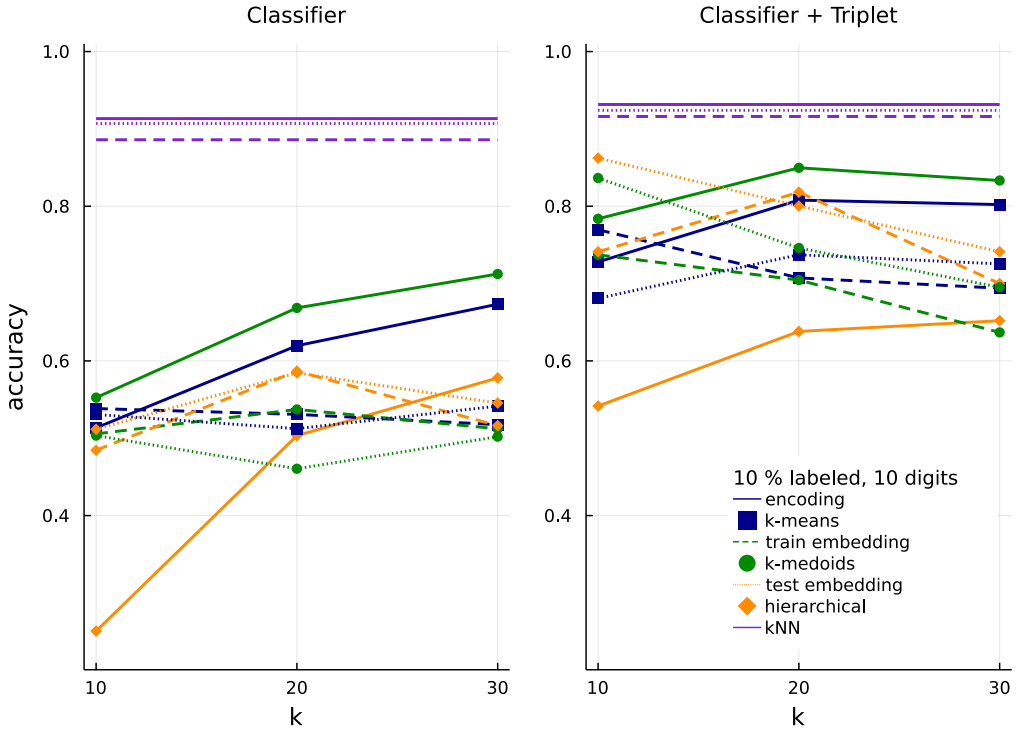


(b) Comparison for 5 % of labeled data, 10 MNIST digits.

Figure B.3: Comparison of clustering accuracy on latent space of classifiers with and without triplet regularization; 5 % of data labeled.



(a) Comparison for 10 % of labeled data, 4 MNIST digits.



(b) Comparison for 10 % of labeled data, 10 MNIST digits.

Figure B.4: Comparison of clustering accuracy on latent space of classifiers with and without triplet regularization; 10 % of data labeled.