Development of unsupervised learning transformations through supervised learning methods.

Author: Patricia Cortajarena Sauca

Ponente: Carlos Roberto del Blanco Adán

Tutor: Iñigo Cortajarena Sauca

Dep: GTI: Grupo de Tratamiento de Imágenes

Trabajo Fin de Grado. ETSIT UPM. Madrid. January, 2018.

Abstract

Since dimensionality reduction has turned to be an important task in machine learning problems, the aim of this project is the development of a supervised machine learning embedding algorithm that seeks to replicate other algorithms' transformations such as MDS and t-SNE which are non-parametric and non-reusable. Moreover, since we are using a neural net to do so, ours can be re-used, reproduced and taken advantage of in a more efficient and convenient way. This algorithm can also be implemented for nearest neighbours search in a lower dimensional space obtained from the model.

The usage of real data-sets and neural nets are the main tools used for this study. The results obtained show the trade-off between precision, computational time and memory usage. The loss of precision of the model developed is negligible compared to the improvements in time a memory.

Keywords: *Machine learning, unsupervised-learning, embedding, neural nets, dimensionality reduction, nearest neighbours.*

Resumen

Los algoritmos de reducción de dimensiones suponen un paso imprescindible en problemas de aprendizaje automático. Por ello, el objetivo de este proyecto es el desarrollo de un modelo supervisado de reducción de dimensiones que sea capaz de replicar las transformaciones que realizan otros algoritmos no paramétricos y no reutilizables como MDS o t-SNE. Además, al utilizar redes neuronales para llevarlo a cabo, conseguimos que nuestro modelo sea reutilizable, reproducible y sacar partido de ello de una forma eficiente. Asimismo, este modelo podría usarse para la obtención eficiente de vecinos cercanos en un espacio de menor dimensión obtenido a partir del mismo.

Las principales herramientas usadas para la investigación son bases de datos reales y redes neuronales. Los resultados obtenidos muestran una relación directa entre precisión y uso de tiempo y memoria. La pérdida de precisión es imperceptible en comparación con la mejora en tiempo computacional y uso de memoria.

Palabras clave: *Aprendizaje automático, aprendizaje no supervisado, redes neuronales, redución de dimensiones, vecinos cercanos.*

Acknowledgements

Special thanks to my director, Carlos Roberto del Blanco Adán, for his professional guidance, enthusiastic encouragement and priceless help. Thanks to my family for their unconditional support and to my tutor Iñigo Cortajarena Sauca for introducing me to the world of machine learning, for his constructive recommendations and for his valuable technical help on this project.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Working with large datasets and high-dimensional data in nowadays' problems has encouraged the use of dimensionality reduction algorithms which try to preserve as much information as possible even with a reduced number of features used to describe that same dataset. This is, dimensionality reduction algorithms compute new features based on the original ones. Although the number of new features computed are less than the number of original ones, the reduction's objective is to still represent the same information without significant distortion. Thus, time and memory may be saved when facing large implementations.

Taking into account that this turns into a difficult task, numerous approaches have been proposed. Although the different approaches try to achieve more or less the same aim, they differ from one another and we can not reassure which one would suite a specific problem or even if the behaviour of the algorithm throws the results we expected or needed.

The first point to take into account is the existence of parametric and non-parametric algorithms, depending on whether the dimensionality reduction can be expressed in terms of parameters or not, which can be conclusive when deciding what algorithm to use. Secondly, in both of them we can find different models proposed depending on what to optimize, yet not everything is going to be preserved as well as in the original dataset, so we need to prioritize some aspects.

The decision of which of them to implement depends on the previous study of our data, the performance requirements and the later purpose and usage of the embedded data.

Non-parametric algorithms have an important disadvantage: as previously said, they don't have

parameters which represent the dimensionality reduction transformation; thus, when a new sample is provided we need to compute the whole algorithm again to obtain the datapoint's coordinates in the lower dimensional space, instead of computing the transformation on this sample alone. This is computationally expensive and somewhat inconvenient for real world applications.

We hereby propose the study and research an implementation of a machine learning model which can replicate the behaviour of a non-parametric dimensionality reduction algorithm. The main objective is to resolve the latter's main disadvantage and make it capable of being repeatedly used with new examples, omiting the need to compute the whole non parametric transformation every time, saving time and computational resources.

The embedding algorithms used as a baseline for this research are the ones listed below:

- PCA (Principal Component Analysis)
- MDS (Multidimensional Scaling)
- TSNE (T-Stochastic Neighbour Embedding)

To conclude, the document is divided into chapters which summarize the main steps of this research. Chapter number 2 describes the State-of-The-Art. It is followed by chapter 3, which details the main steps of the implementation. Afterwards, chapter 4 describes the main implementation's applications and the corresponding metrics computed to analyze the algorithm's behaviour. The last chapter (chapter 5) comprehends the conclusion and summarizes the main ideas discussed in this paper.

Chapter 2

State-of-The-Art

Two of the most classical dimensionality reduction techniques are know to be PCA [5] and MDS [15]. They try to preserve the true structure of the data even though they are simple to implement and computationally efficient. They rely on linear models to compute an embedding of the original space into a lower dimensional one. A related linear implementation which is based on random projections of the data is described in [1, 14]. Some other well-known techniques are ISOMAP [11] and t-SNE [12], but their implementations often require more memory and computational load, yet the complexity of the algorithms is higher. Nevertheless, these algorithms are capable of reducing the number of features in a non-linear manner, whereas linear models can not faithfully represent the data's structure since the output space's coordinates must always be linear combinations of the original features. ISOMAP computes geodesic distances instead of euclidean dissimilarities while t-SNE's approach is to compute probabilities from an euclidean metric.

Neural networks have become an essential tool in machine learning problems and can certainly be used for dimensionality reduction. Autoencoders [4, 13] consist on a neural net with a decreasing number of neurons per layer and a symmetrical structure whose output's size is the same as the input's. If the output represents the input without any distortion, then middle layer with the fewest number of neurons corresponds to the embedded description of the data. This is represented in figure 2.1.

The following sections (sections 2.1, 2.2 and 2.3) summarize the main ideas of the algorithms on which the research is based.

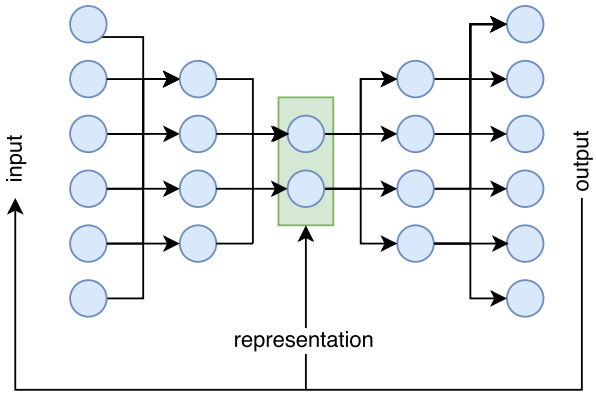

Figure 2.1: Autoencoder neural network.

2.1 PCA: Principal Component Analysis

Principal Component Analysis [5] is based on reducing the number of features by processing the correlations amongst them. The aim is to eliminate these correlations by representing the matrix $\mathbf{X} \in \Re^{mxn}$ (with m being the number of data points and n de number of features) in a lower dimensional orthogonal basis. By omitting the correlation between columns of the matrix $\mathbf{X}$ we are capable of doing away with redundancies.

The model starts by computing de covariance matrix, which results in a $\Re^{nxn}$ symmetric matrix by using the next expression:

$$\text{cov}(\mathbf{X}) = \frac{1}{m-1}\,\mathbf{X}^T\mathbf{X}$$

Since the aim of the PCA is to eliminate the correlations, the covariance matrix of the resulting $\mathbf{Y}$ should be a diagonal matrix with just the variances of the columns.

PCA is often used because of a great advantage: with it, a linear transformation ($\mathbf{Y} = \mathbf{XP}$) can be inferred. As the resulting transformation is basically a change-of-basis (matrix multiplication), it turns to be easy to reuse and computationally simple.

For any symmetric matrix we can find an eigenvalue decomposition to a diagonal matrix, matching exactly our problem (with $\mathbf{X}$ and $\mathbf{Y}$ and their covariances).

$$\mathbf{Y} = \mathbf{XP}$$

$$cov(\mathbf{Y}) = \frac{1}{m-1}\mathbf{Y}^T\mathbf{Y} = \frac{1}{m-1}(\mathbf{XP})^T\mathbf{XP} = \mathbf{P}^T cov(\mathbf{X})\mathbf{P}$$

$$\mathbf{D} = \mathbf{V}^T \mathbf{A} \mathbf{V}$$

$$\mathbf{A} = cov(\mathbf{X}); \mathbf{P} = \mathbf{V}^T; \mathbf{D} = cov(\mathbf{Y})$$

With the previous expressions we get to the point that, by computing the eigenvectors of the covariance matrix $\mathbf{X}$, we can get a linear transformation from space $\mathbf{X}$ to space $\mathbf{Y}$. The matrix of eigenvalues obtained ($\text{cov}(\mathbf{Y})$) are sorted decreasingly and the vectors of the orthogonal basis. Choosing the $\mathbf{N}$ first values of this matrix, being $\mathbf{N}$ the desired output dimension, and computing the corresponding change-of-basis, we obtain the coordinates of the samples in the output space.

2.2 MDS: Multidimensional Scaling

Multidimensional Scaling [15] is a dimensionality reduction technique that tries to create a map which preserves the relative distances between the data points. This lower dimensional embedding aims to keep as much distance information as possible. In order to visualize data, this dimensional map needs to be a one, two or, at most, three dimensional space.

MDS computes a metric or non-metric solution depending on the input matrix provided, which has to be a *proximity* matrix. This *proximity* matrix quantifies how close the samples are in the original space. On the one hand for metric solutions, this *proximity* matrix has to be a true distance matrix, while on the other, both dissimilarities or correlations are alternatives to the input for the problem's matrix.

MDS algorithm is based on some ideas discussed in the previous section. The *proximity* matrix is always symmetric and somehow describes the relations between the features. This means that we can basically treat our problem as a variation of the PCA algorithm.

Metric MDS performs the same steps as in PCA, being a distance matrix the input to the algorithm.

In non metric MDS, we assume a less strict relation and we compute the observed distances as a function of the real distance plus some measure error. The usable information in this case is going to be the rank order of the previous matrix, which could be the input for the model.

The main difference between PCA and MDS is the fact that, because of the need to compute

a pairwise distance or proximity matrix, there is no linear transformation that suits both the distance computations plus the matrix operations (PCA / change-of-basis), so MDS turns to be non-parametric.

2.2.1 Stress metric

In every machine learning problem, we need a metric or loss which quantifies how well the algorithm is fitting our data. In Multidimensional Scaling we compute the *stress* metric that compares the distances in the output space with the original ones. Note that this obviously depends on the number of dimensions we want to keep; if we lower the number of dimensions, the stress will be higher since we are trying to represent the same distance relations in a smaller number of dimensions.

$$stress = \sqrt{\frac{\sum (d_{ij} - d'_{ij})^2}{\sum d_{ij}^2}}$$

d_{ij} represents the original distance and d'_{ij} is the distance in the output space (based on the MDS model). This last value is either a predicted true distance or a function which represents the non-metric transformation of the data.

Regarding the previous expression, if our prediction stands well for the original data, the stress value should be lower, relating zero stress to the perfect performance of the MDS algorithm.

2.3 TSNE: T-Stochastic Neighbour Embedding

T-Stochastic Neighbour Embedding [12] is an example of a non-linear dimensionality reduction algorithm. It relies on Stochastic Neighbour Embedding (SNE). The main reason why this method is used is because it is capable of representing both the local and the global structure of the original data. This section will be divided in two: the basis SNE and the t-SNE upgrades.

2.3.1 SNE

SNE approaches the dimensionality reduction by converting the Euclidean distances into conditional probabilities as a way of expressing similarities between points. This means that we

measure the similarity of two points x_i, x_j as the probability $p_{i|j}$ of finding the second one as a neighbour of the first. The probability in the original and in the low dimensional space is computed as seen:

$$p_{i|j} = \frac{exp(-||x_i - x_j||^2/2\sigma_i^2)}{\sum_{k \neq i} exp(-||x_i - x_k||^2/2\sigma_i^2)}$$

$$q_{i|j} = \frac{exp(-||y_i - y_j||^2)}{\sum_{k \neq i} exp(-||y_i - y_k||^2)}$$

These expressions represent probabilities under gaussian distributions centered in each datapoint x_i, where σ_i is the sample's variance. The conditional probability represents how likely it is for the point x_j to be considered a sample's neighbour. According to the value of σ_i the probabilities change, so it is chosen depending on the density of the data. In a dense region, a smaller value of σ_i is more appropriate than in a sparse region. Having decided which value to use for each of the samples, we obtain a probability distribution, P_i, that is computed as explained in the next paragraph.

As the point of this metric is to compute similarities as probabilities, we can calculate the mismatch between $p_{i|j}$ and $q_{i|j}$ for all the samples and consequently obtain the algorithm's behaviour by analysing how many neighbours have been kept in the low dimensional map. In terms of conditional probabilities, Kullback-Leibler divergence perfectly suits this need. Summing up all the previous ideas we get to the point of minimizing the cost function described as:

$$C = \sum_i KL(P_i||Q_i) = \sum_i \sum_j p_{i|j} log \frac{p_{i|j}}{q_{i|j}}$$

The limitations of this algorithm are the non symmetric general expression of the Kullback-Leibler divergence, the difficulty to optimize the cost function, the fact that we need to choose different values of the variance depending on the point and the "crowding problem". T-SNE tries to solve these limitations.

2.3.2 t-SNE

Although SNE is capable of showing good visualizations and is widely accepted, t-SNE was proposed as a modification to this algorithm which tried to make it more accurate and lower its computational cost.

Instead of minimizing the sum of the different KL divergences along all the samples, another way of computing the cost is proposed: we are trying to minimize a single KL divergence between a joint probability P and the equivalent one in the low dimensional space, Q. With this symmetric approach, we omit the need to obtain the variances for each sample and the cost function is much easier to compute and optimize, and so is its gradient.

$$C = KL(P||Q) = \sum_i \sum_j p_{ij} log \frac{p_{ij}}{q_{ij}}$$

$$\frac{\partial C}{\partial y_i} = 4 \sum_j (p_{ij} - q_{ij})(y_i - y_j)$$

Moreover, t-SNE handles the problem known as "crowding problem" extremely well by introducing a heavy-tailed distribution, the Student-t distribution, rather than a Gaussian for the low-dimensional space. The "crowding problem" appears when we want to faithfully represent mutually equidistant points while going from a high-dimensional space to low-dimensional one. This task tends to be quite difficult because the area / volume available in a lower dimensional space is considerably smaller than in the high dimensional one. In the end, the points tend to clash in the center of the map, and it becomes hard to represent and preserve the true distances from the original space.

The t-Student distribution is a heavy-tailed distribution and it allows for a more faithful representation of distances. The joint probabilities are now then computed as follows:

$$q_{ij} = \frac{(1 + ||y_i - y_j||^2)^{-1}}{\sum_{k \neq l} (1 + ||y_k - y_l||^2)^{-1}}$$

The reason why this particular distribution was chosen is because it is closely connected to the Gaussian distribution, as the t-Student is an infinite mixture of Gaussians.

To conclude, the gradient of the Kullback-Leibler divergence taking into account these changes in the Q low-dimensional space stands for the next expression:

$$\frac{\partial C}{\partial y_i} = 4 \sum_j (p_{ij} - q_{ij})(y_i - y_j)(1 + ||y_i - y_j||^2)^{-1}$$

Chapter 3

Framework

We hereby propose the implementation of a supervised learning tool which learns the embedding derived by an unsupervised learning method, which could be used in real machine learning applications.

Two main supervised learning algorithms used in today's applications are Neural Networks and Decision Trees. To know which of them or even if both could suit our problem we studied their main characteristics and differences.

First of all, decision trees are believed to be faster and easier to train, and their results can be very interpretable. On the other hand, neural nets are slower to train (not so with GPUs) and less interpretable (usually considered black-boxy). Neural networks are prone to over-fitting if the size of the given data-set is not large enough or if the parameters used for regularization aren't the best for a particular use case, while decision trees are less prone to over-fitting if correctly pruned / regularized (eliminating some of the tree's branches and restricting tree depth can help to mitigate these issues). What really makes the difference from one another is the fact that neural nets are capable of having two or more output variables (when considering general regression) while the regression trees can only handle one output. Whenever decision trees need to be used for multi-output regression, multiple trees are needed, which increases computational complexity and performance assessment, since each of the trees / tree models may have different hyper-parameters.

Taking these differences into account, the supervised learning algorithm chosen for this research is neural networks, since it is a flexible and powerful tool that can learn smooth mathematical

functions but also complex non-linear and possibly noisy relationships. The advantage of our framework relies on the fact that, as it's said, due to being a supervised algorithm, it is possible to reuse it after the training process since the parameters and weights of the net can be stored.

Multidimensional Scaling and T-Stochastic Neighbour Embedding are both non-parametric dimensionality reduction algorithms. As it was discussed in the introduction (Chapter 1), these algorithms are not parametric and non-reusable; once the embedding is computed, if we wanted to add a new sample to the set and obtain its embedded vector, we would need to compute everything again in order to include the new sample, which may lead to different results and implies a huge amount of computation. This is impractical and not suitable for real life applications.

Having said that, we have a set of **m** samples with **n** features each, and we want to represent those points in a lower-dimensional space, for which we are willing to use a neural net (supervised algorithm) which can learn that embedding, replicating any of the previously mentioned dimensionality reduction algorithms (unsupervised, non-parametric and gradient descent based).

The neural network needs to be trained at least once, so we need to first compute the dimensionality reduction algorithm (step 1, figure 3.1) to feed the results into the neural net with its output being the reduced vectors computed by the latter. Having the neural net trained and its weights stored (step 2, figure 3.1), as we get a new sample, instead of recalculating the MDS or the t-SNE from scratch, we can reuse the neural net's parameters to obtain a prediction (step 3, figure 3.1) which is the lower dimension vector of this sample.

With this implementation we are able to speed up the process, reduce memory usage (since we only need to store the neural net's weights) and also reduce computational time considerably.

3.1 Cross Validation

Cross validation is a technique used to evaluate predictive models and its aim is to estimate if the model is capturing the general or underlying structure of the data and can suit new unseen examples, and also estimate how accurate the model is only taking into account these unseen samples (in-sample performance is useless since the training set can be easily over-fitted with a complex enough model).

The easiest way to know whether a model is under-fitting or over-fitting is dividing the dataset into training set and test set, choosing for example 80% to train the model and 20% to test its behaviour in samples that have not been seen yet. Although this is a good first approach, we might be fitting our model to the test samples by trial and error without noticing it; this is, we might be choosing the model's parameters to fit the test set, and still not capturing the overall underlying function of the data-set.

A more robust approach would be to use cross validation, which consists in dividing the data into a number of disjoint folds; some usual choices are three, five and ten folds, where this choice depends only on the size of the dataset (smaller datasets usually require larger number of folds). The algorithm is then trained repeatedly, choosing every time one of the folds to test and the rest of them to train. As we have trained and tested the model with disjoint samples, we are not fitting our model to certain subsets of the dataset. We use an error metric to the model's performance in all test / validation folds and compute the mean and variance of this error metric along all folds. The mean represents the expected performance in a future live test (always considering that the underlying problem or dynamics don't change, an issue that may happen in real live situations) and the standard deviation of the error throughout all folds is a measure of model variance. High deviations may imply that the model is over-fitted (overly complex) and that we may need to regularize it more. The best cross validation performance usually comes from a correct balance between bias and variance for our model, which comes from a correct choice of hyper-parameters. The following schema represents the cross validation procedure.

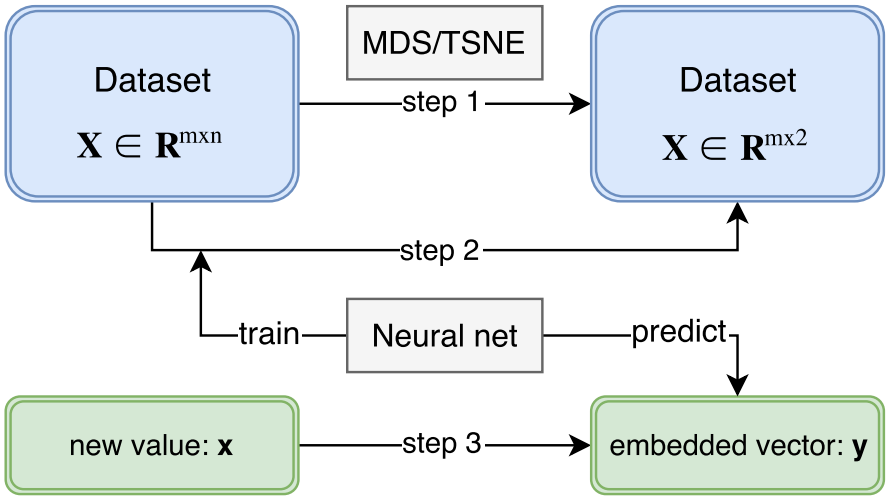

Figure 3.1: Dimensionality reduction algorithm learned by a Neural Network

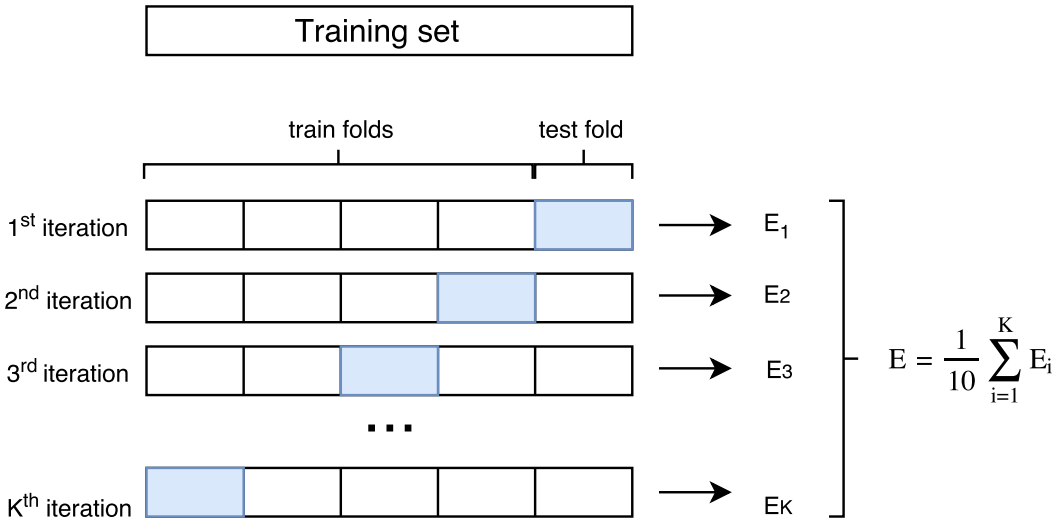

Figure 3.2: Cross validation using K-Folds.

Chapter 4

Applications

4.1 Application 1: Replicating t-SNE/MDS with neural net

The first implementation to be discussed is the base model to replicate a general non-parametric dimensionality reduction algorithm. For this particular use case, T-Stochastic Neighbour Embedding and MDS are used as the algorithms to carry out transformations with. The main objective is to assess whether the supervised neural net is capable of replicating the previous algorithms.

4.1.1 Dataset

The dataset *"load-digits"* is the data used to test this example and is obtained from the Scikit-Learn datasets [9]. It consists of 1797 instances of handwritten digits from 0 to 9 from different writers with 64 features each. This features represent the values of gray-scale intensity for each pixel in the image. Our images are 8x8 pixel images represented by a total of 64 features with a color resolution (brightness in case of grey-scale) ranging from 0 (completely black) to 16 (pure white) as show in figure 4.1. To translate this into a training set, each image is flattened as a row with its corresponding 64 pixel features; a last column with the label of the number represented in the image is concatenated to the table.

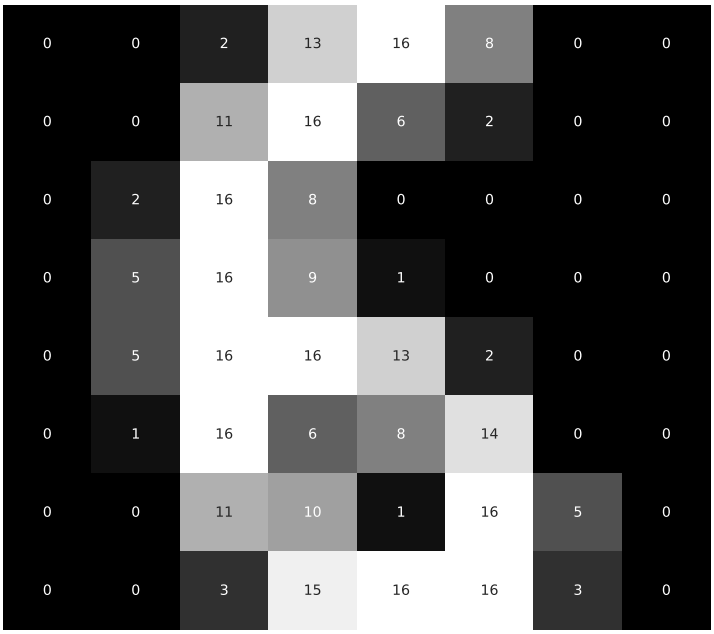

Figure 4.1: Example dataset point representing a handwritten digit.

4.1.2 Algorithm: t-SNE/MDS and neural net

This section discusses the computation of the space reduction algorithm and the supervised replicator implementation using a neural net model.

As a starting point for the problem, the embedding model is computed for all the data points beforehand. Standardization is carried out for all features in the original data by removing the mean and scaling to unit variance. This is done for each of the 64 variables, since the neural net is sensible to scales. We don't do the scaling inside cross validation since our only aim is to check whether the neural net can learn the space mapping or not, so this is completely different to usual machine learning problems; our output here is the sample vectors / embeddings derived from the unsupervised model. Due to the data set containing 1797 instances with 64 features each, a two dimensional space output is chosen, in order to be able to visualize the results.

Once the embeddings are obtained, both the original and the compressed samples are divided into two corresponding folds, one for training the neural net and the other one to test whether the model can correctly locate unseen points where the unsupervised model had placed them. Therefore, by doing this, we are actually able to check if the neural net is capturing the general behaviour of the dimensional reduction algorithm and can replicate it further on.

To continue, the next to consider is the neural net's structure. There are several kinds of neural nets depending on their topology. The main structure parameters for a neural net with no convolution or pooling (simple perceptron-like structure) are the number of layers and the number of neurons per layer. Depending on these a neural network can be shallow or deep (number of hidden layers) and wide or narrow (neurons per layer). Depending on the kind of structure used, the results obtained will differ from one another. In section 4.1.3 the differences between them are shown.

Once the net is built, the standardized original data from the training subset will constitute the net's input in the training process. The output is, in our case, a two variable output, and corresponds to the coordinates of all embeddings.

The neural net aims to get the same output as the dimensionality reduction algorithm, that is why the loss function used for the neural net's gradient descent is a custom euclidean distance error metric (although this is not the only possible loss function). This way, the neural net tries to minimize the distance between the true position obtained directly from t-SNE or MDS and the position being predicted by the model. In this case the distance metric that throws the best overall results corresponds to the mean of the euclidean distances between true and predicted locations. Other metrics were also tested, such as mean loss (mean of the square of the difference between points) but results were not as good. The following equation describes the loss function used (where n represents samples and j dimensions):

$$e = \frac{\sum^{n} \sqrt{\sum^{j} (\hat{y}_j - y_j)^2}}{n}$$

To conclude, as the net has been trained to obtain two output values depending on the 64 feature standardized values in the input, it is possible to predict, taking into account the new input, the embedded data replicating what the non-parametric embedding model would compute, but avoiding the need to recalculate the whole algorithm.

4.1.3 Metrics

The aim of this first implementation is not only to be able to reuse the embedding function but also to make if extremely CPU efficient to compute the location (reduced vector) of new samples and also provide a memory efficient way of doing this (we only need to store the neural

Algorithm 1 neural net replicating process

1: **procedure** REPLICATING PROCEDURE
2: *data obtaining*
3: $\boldsymbol{X} \leftarrow$ standardized input matrix
4: $\boldsymbol{X\_emb} \leftarrow$ compute t-SNE/MDS from $\boldsymbol{X}$
5: ***X_train; X_test; X_emb_train; X_emb_test;*** $\leftarrow$ train and test data computation
6: *neural net model construction*
7: *fit neural net model*
8: $\boldsymbol{X\_train} \leftarrow$ neural net's input
9: $\boldsymbol{X\_emb\_train} \leftarrow$ neural net's output
10: ***distance error*** $\leftarrow$ neural net's error metric
11: *predict neural net model*
12: *64 feature input (new example)* $\rightarrow$ neural net $\rightarrow$ predict output = dimension reduction

net weights).

To do this, several neural net structures were tested. Since the transformation or mapping function derived by the unsupervised method seems to be pretty smooth (complex but with low noise levels), low regularization (no $L2$) with no dropout threw the best results. For all neural net structures, transformation times and errors were computed in the following tables (tables 4.1, 4.2, 4.3 and 4.4).

Recall that these measurements are performed over samples from the test set which are unseen for the model (non present during training), and thus they indicate the general behaviour of the model.

Tables 4.1 and 4.2 include the models' structures and the errors obtained for both of the embedding algorithms. As stated before, the model's structure depends on the total number of hidden layers, the number of neurons per layer and on the dropout applied to each layer (in the end no dropout was used as discussed before). The neural net structure is inherent in the length of the array in the table, where the length represents the number of layers. To summarize, the types of neural nets tested are divided into deep or shallow and wide or narrow nets.

The out of sample error computed is again an euclidean distance error and matches the loss function exactly. As stated before, it evaluates the difference between the position computed by t-SNE or MDS and what the model is predicting. The error value corresponds to the mean

of these Euclidean distances. Percentages are also included to reckon whether the total error compared to the range of values acquired by the algorithms stand for a decent figure.

	dropout	layers	type	error	%error_X	%error_Y
model1	[0, 0, 0, 0]	[32, 16, 16, 8]	deep - narrow	6.335186	4.976027	5.882600
model2	[0, 0]	[32, 32]	shallow - narrow	5.810866	4.564196	5.395738
model3	[0, 0]	[32, 16]	shallow - narrow	6.276359	4.929821	5.827976
model4	[0, 0, 0]	[128, 64, 64]	deep - wide	3.379963	2.654821	3.138498
model5	[0, 0, 0, 0]	[128, 64, 64, 32]	deep - wide	3.481527	2.734596	3.232807
model6	[0, 0, 0]	[128, 128, 128]	shallow - wide	3.239071	2.544157	3.007672
model7	[0, 0]	[128, 64]	shallow - wide	4.151196	3.260593	3.854634
model8	[0]	[1000]	shallow - wider	5.510108	4.327962	5.116465

Table 4.1: Comparison between model's errors obtained out of sample (t-SNE)

	dropout	layers	type	error	%error_X	%error_Y
model1	[0, 0, 0, 0]	[32, 16, 16, 8]	deep - narrow	1.539260	3.164186	2.495089
model2	[0, 0]	[32, 32]	shallow - narrow	1.434994	2.949851	2.326077
model3	[0, 0]	[32, 16]	shallow - narrow	1.401265	2.880517	2.271405
model4	[0, 0, 0]	[128, 64, 64]	deep - wide	1.328322	2.730571	2.153166
model5	[0, 0, 0, 0]	[128, 64, 64, 32]	deep - wide	1.238499	2.545926	2.007566
model6	[0, 0, 0]	[128, 128, 128]	shallow - wide	1.227050	2.522390	1.989007
model7	[0, 0]	[128, 64]	shallow - wide	1.257435	2.584852	2.038261
model8	[0]	[1000]	shallow - wider	1.557930	3.202565	2.525353

Table 4.2: Comparison between model's errors obtained out of sample (MDS)

Taking an in depth look at computation times and considering that neural nets' whole transformation consists in matrix scalar multiplications and parametric activation functions that can be also vectorized (hyperbolic tangent in our case), this allows for an extremely efficient computation of inferred coordinates (low CPU usage, high speed).

Tables 4.3 and 4.4 show the huge difference between the computational time (in seconds) of the dimensional reduction algorithm and the neural net's development for the same number of

points, referring to the whole data set (1787 points in total).

	times
tsne	49.568863
model1	0.139451
model2	0.117794
model3	0.122860
model4	0.181596
model5	0.221698
model6	0.217583
model7	0.207650
model8	0.226093

Table 4.3: Computational times (t-SNE)

	times
unsup	28.104550
model1	0.067040
model2	0.082692
model3	0.087886
model4	0.089499
model5	0.105532
model6	0.133000
model7	0.126438
model8	0.184579

Table 4.4: Computational times (MDS)

4.1.4 Conclusion

After all and given the results shown in the previous tables, we can assure that, even ignoring the neural nets' diverse structures, all of them are certainly quicker in obtaining the embeddings. If we look for accuracy, a wide neural net is better at replicating both the t-SNE and MDS algorithms. In particular, the model 6 suits the best all problems presented.

The next four figures show the results obtained. The first and third ones represent the predictions from the neural networks of the location of all samples in the out-of-sample test for the best of the models in both problems (model 6), and the second and fourth ones represent the same points from the embedding algorithms (t-SNE and MDS). As it can be clearly seen, the neural net is able to faithfully replicate the position of new points and performs extremely well in terms of preserving the local structure of the dataset.

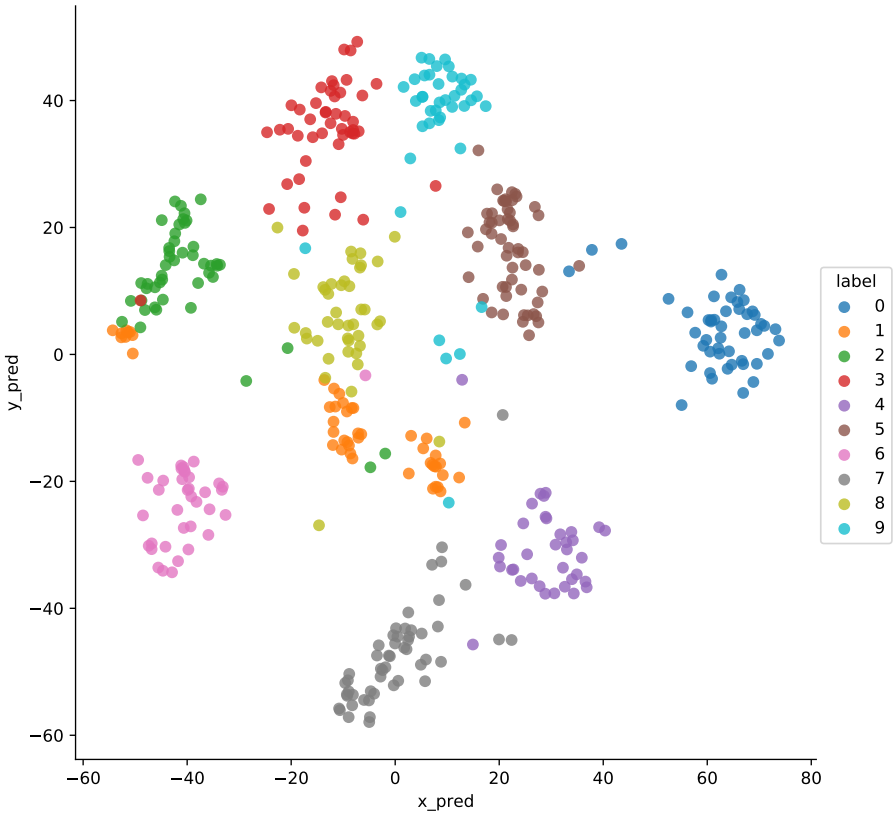

Figure 4.2: Low dimensional space from neural net trained for t-SNE.

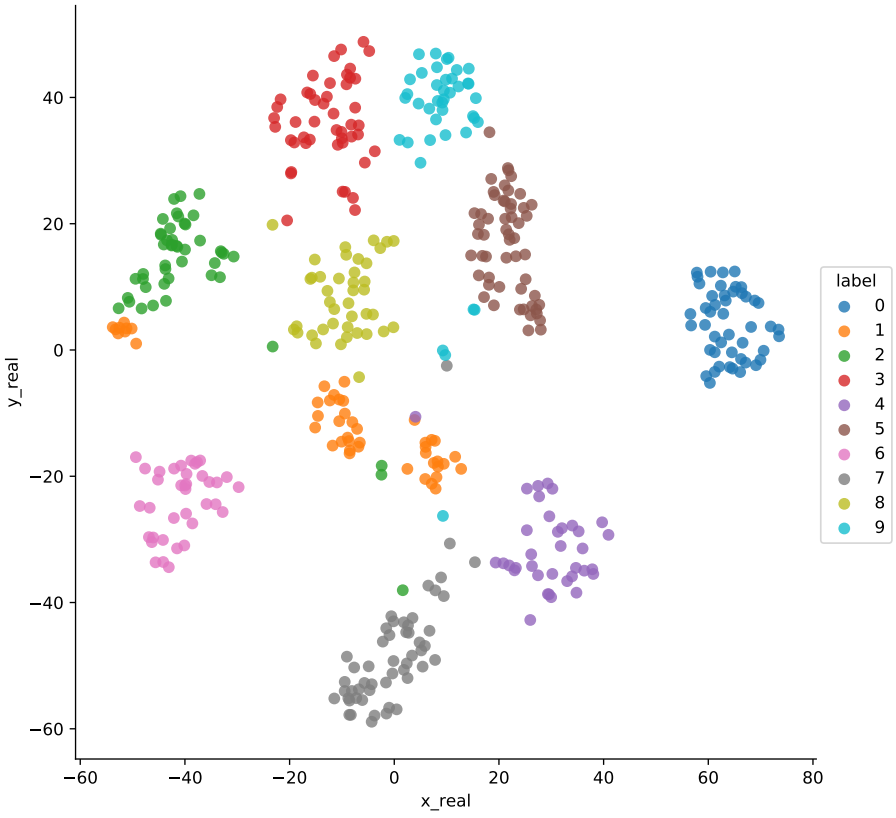

Figure 4.3: Low dimensional space from t-SNE

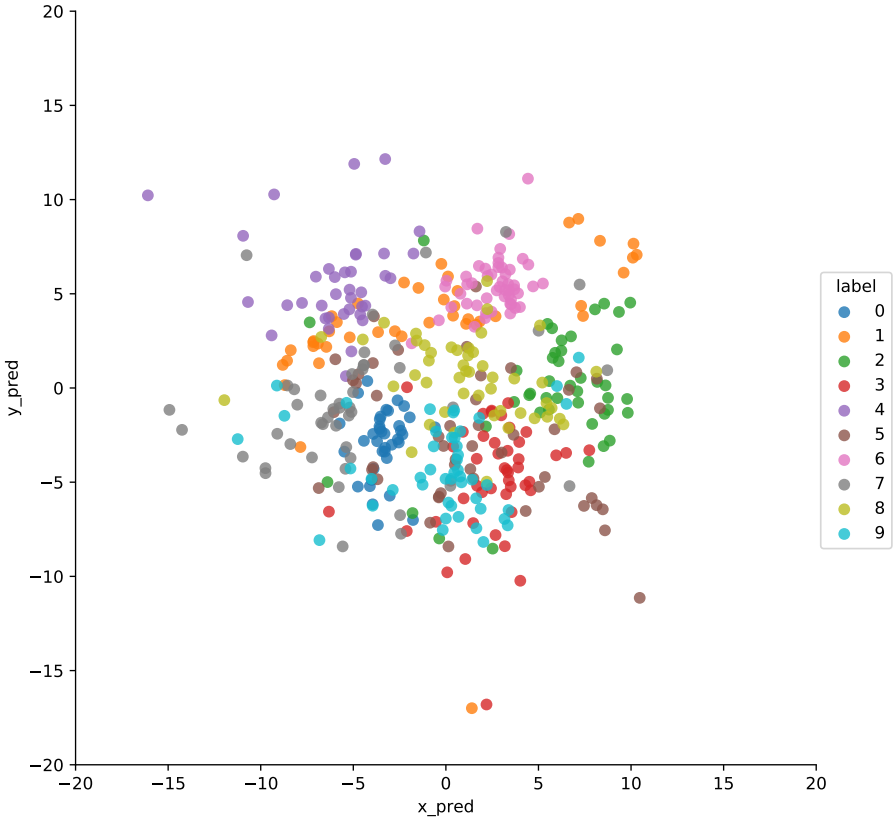

Figure 4.4: Low dimensional space from neural net trained for MDS.

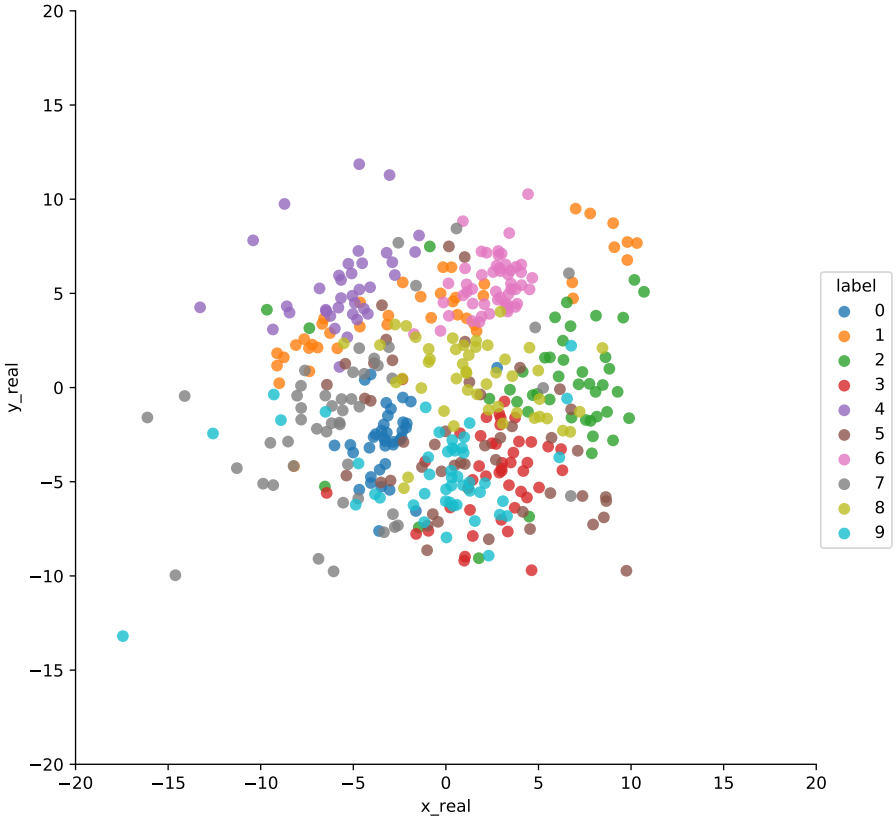

Figure 4.5: Low dimensional space from MDS

4.1.5 Code

By running the following python 3 code these results can be easily reproduced. For better performance, a server with multiple cores (and possibly a GPU) may be used, since all tests have been parallelized using python 3's **joblib** package.

```
import matplotlib
matplotlib.use("Agg")
import pandas as pd
import numpy as np
import time

import seaborn as sns
import matplotlib.pyplot as plt

from scipy import sparse
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split

from sklearn.preprocessing import StandardScaler
from sklearn.manifold import TSNE, MDS
from sklearn.pipeline import make_pipeline

from keras.wrappers.scikit_learn import KerasRegressor
from keras.models import Sequential
from keras.layers import Dense, Activation, Dropout
from keras import optimizers
import keras.backend as K

# distance loss


def dist_loss(y, y_):
    return K.mean(K.sqrt(K.sum(K.square(y - y_), axis=1)))


def dist_error(y_true, y_pred):
    return np.mean(np.sqrt(np.sum((y_true - y_pred) ** 2, axis=1)))

# percentage error


def percentage_error(err, X_div):
    maximo = X_div.max(axis=0)
    minimo = X_div.min(axis=0)
    rango = maximo - minimo
    return err / rango * 100

# neural net model


def reg(shape, loss_func, layers, dropout,
        output_shape, lr, act_function='tanh'):

    model = Sequential()
    for i, (layer, drop) in enumerate(zip(layers, dropout)):

        if i == 0:
            model.add(Dense(layer, input_shape=(
                shape,), activation=act_function))
        else:
            model.add(Dense(layer, activation=act_function))

        model.add(Dropout(drop))

    model.add(Dense(output_shape, activation='linear'))
```

```
    ada = optimizers.Adagrad(lr=lr)

    model.compile(optimizer=ada, loss=loss_func)
    return model

# neural net replicating


def app(X_unsupervised, dic, X, y):

    resultados = {key: [] for key in dic}
    tiempos = {key: [] for key in dic}

    X_train, X_test, X_unsuper_train, X_unsuper_test, y_train, y_test = train_test_split(
        X, X_unsupervised, y, random_state=0)

    min_error = 1000

    for model_name, model_params in dic.items():
        nn = lambda: reg(
            X.shape[1], loss_func=dist_loss,
            layers=model_params['layers'], dropout=model_params['dropout'],
            output_shape=2, lr=0.1, act_function='sigmoid')

        neuralmodel = KerasRegressor(nn, epochs=200, batch_size=10)

        super_pipe = make_pipeline(
            StandardScaler(),
            neuralmodel)

        super_pipe.fit(X_train, X_unsuper_train)

        # compute times of predicting all the training set
        start = time.time()
        super_pipe.predict(X)
        stop = time.time()
        tiempos[model_name] = stop - start

        # predictions of the test set
        predictions = super_pipe.predict(X_test)

        error = dist_error(X_unsuper_test[:, :2], predictions[:, :2])

        # select minimun error to export only those predictions
        if min_error > error:
            min_error = error
            result_predic = predictions
            result_unsuper_test = X_unsuper_test
            result_y = y_test

        # calculate percentage error
        perc = percentage_error(error, X_unsuper_test)

        res = [error]
        for e in perc:
            res.append(e)

        resultados[model_name] += res

    return resultados, tiempos, result_predic, result_unsuper_test, result_y

if __name__ == '__main__':

    dataset = load_digits()

    X, y = dataset['data'], dataset['target']

    tsne = TSNE(n_components=2, verbose=10)
    md = MDS(n_components=2, random_state=5, n_jobs=1,
             verbose=10, n_init=1)

    # pipelines
    unsupervised_pipeline = make_pipeline(
```

```
    StandardScaler(),
    md
)

start = time.time()
X_unsup = unsupervised_pipeline.fit_transform(X)
end = time.time()

time_unsup = end - start

# iterate models
models = {
    "model1": {
        "layers": [32, 16, 16, 8],
        "dropout": [0, 0, 0, 0],
        "type": 'deep - narrow'
    },
    "model2": {
        "layers": [32, 32],
        "dropout": [0, 0],
        "type": 'shallow - narrow'
    },
    "model3": {
        "layers": [32, 16],
        "dropout": [0, 0],
        "type": 'shallow - narrow'
    },
    "model4": {
        "layers": [128, 64, 64],
        "dropout": [0, 0, 0],
        "type": 'deep - wide'
    },
    "model5": {
        "layers": [128, 64, 64, 32],
        "dropout": [0, 0, 0, 0],
        "type": 'deep - wide'
    },
    "model6": {
        "layers": [128, 128, 128],
        "dropout": [0, 0, 0],
        "type": 'shallow - wide'
    },
    "model7": {
        "layers": [128, 64],
        "dropout": [0, 0],
        "type": 'shallow - wide'
    },
    "model8": {
        "layers": [1000],
        "dropout": [0],
        "type": 'shallow - wider'
    }
}

modelsdf = pd.DataFrame(models).T

# obtain results
results, time, predictions, real_data, y_label = app(X_unsup, models, X, y)

# error metrics
resultsdf = pd.DataFrame(
    results, index=['error', '%error_X', '%error_Y']).T
totalmetrics = pd.concat([modelsdf, resultsdf], axis=1)
# totalmetrics.to_latex(buf='../text/figures/app1metricserror.tex')

# times
time_unsup = pd.DataFrame([time_unsup], columns=['times'], index=['unsup'])
timesdf = pd.DataFrame([time], index=['times']).T
totaltimes = pd.concat([time_unsup, timesdf])
# totaltimes.to_latex(buf='../text/figures/app1metricstime.tex')

# save plot figures
predictdf = pd.DataFrame(predictions, columns=[
```

```
                        'x_pred', 'y_pred']).assign(label=y_label)
realdf = pd.DataFrame(real_data, columns=[
                      'x_real', 'y_real']).assign(label=y_label)

fig, ax = plt.subplots(1, 1, figsize=(10, 7))
sns.lmplot(x='x_pred', y='y_pred', data=predictdf,
           hue='label', fit_reg=False, size=7)
plt.xlim(-20, 20)
plt.ylim(-20, 20)
plt.savefig('../figures/app1plotpredictionsmds.pdf', bbox_inches='tight')

sns.lmplot(x='x_real', y='y_real', data=realdf,
           hue='label', fit_reg=False, size=7)
plt.xlim(-20, 20)
plt.ylim(-20, 20)
plt.savefig('../figures/app1plotrealmds.pdf', bbox_inches='tight')

# example of number representation
fig, ax = plt.subplots(1, 1, figsize=(9, 8))
heatmap(X[550].reshape(8, 8), cmap='gray', annot=True, ax=ax, cbar=False)
plt.axis('off')
plt.savefig('../text/figures/exampledigit.pdf', bbox_inches='tight')
```

4.2 Application 2: Approximate Nearest Neighbor search

The second application consists in the computation of nearest neighbours in a low-dimensional space. This second implementation focuses on the Multidimensional Scaling algorithm, since this particular method's objective is to preserve distances and, consequently, local structure (in terms of neighbors) of the data. The aim is to compute the nearest neighbours in the output space and check how accurate this computation is compared to a brute force approach. Since this computation is performed in a lower dimensional space, it provides a real application with significant improvements in speed and memory usage, since we are only storing a reduced number of columns and performing euclidean distance computations using these dimensions.

4.2.1 Dataset

The dataset *"forests.csv"* is used for this application and is obtained from the UCI Machine Learning Repository [7]. For convenience, a random subsample of 20.000 examples is selected with 55 features each. All features are numeric variables that describe several characteristics and measurements of each forest. The forest class is represented by a number in a range of 1 to 7, shaping the last column of the dataset.

The reason why this dataset is chosen for this particular application relies on the need to compute a substantial amount of examples in the training phase to obtain a reliable model and to be able to asses time and memory performance improvements and also recall/precision

degradation.

4.2.2 Algorithm: Approximate nearest neighbors using MDS embeddings

As it has just been stated, the main task contemplated in this application is the computation of nearest neighbours. The following tests bear with re-usability and time / memory usage reduction.

Regarding the first point, because of MDS being non-parametric, there is no way to obtain the embeddings for new samples. This means if we have already obtained the low-dimensional map of points from the dataset, it would be impossible to obtain the coordinates of a new sample unless the algorithm is recomputed again from scratch. If we based our nearest neighbours search in the output space from the MDS algorithm it's true that we would gain the benefits of making computations in a

Nearest neighbours brute force approach [2] is known to have poor scalability and both low CPU / memory efficiency since it requires to compute euclidean distances between each point and all of the rest and then sort them according to their distances, from lowest to highest. This means that its computational complexity order corresponds to the expression $O(n^2)$, where n is the number of samples of the query.

Since euclidean distances are computed as:

$$dist(p,q) = \sqrt{(q_1 - p_1)^2 + (q_2 - p_2)^2 + ... + (q_n - p_n)^2} = \sqrt{\sum_{j=1}^{d}(q_j - p_j)^2}$$

Therefore, if we increase the number of samples n, the amount of pairwise distance computations needed increases as n^2. The brute force algorithm's complexity in terms of number of dimensions and sample is consequently $O(n^2 * d)$.

As for memory usage (complexity) and taking into account the previous ideas of how neighbours are computed, this model requires the original matrix to be stored which means the memory complexity also depends on the number of points and their dimensions, which corresponds to the following expression: $O(n * d)$. Our experimental approach tries to deal with these inconveniences by learning an unsupervised algorithm's map.

The algorithm involves two phases: one fitting process and afterwards its usage (transformation) for new points. This is why the original dataset is split into two subsets, one for training and the second one to test out-of-sample performance.

The training phase aims to obtain all parameters or transformations which will be later used to transform the data. During the fitting process, the MDS computes the embeddings (vectors of all samples in the embedding space) directly by using gradient descent on its loss function (stress). This is why we would not be able to obtain the transformation of a new sample without having to fit the whole dataset again (which requires computational time), thus we would not be able to step into the process of computing nearest neighbours in the low-dimensional space unless we are querying already known samples.

Instead, in this proposal, and relative to the first application (4.1), the training consists in learning the mapping of a dimensionality reduction algorithm from a high-dimensional space to a low-dimensional space by training a neural net to do so. In this case, the model just saves the neural net's parameters which replicate the MDS, instead of the exact transformations from it, and stores in memory only the embeddings (lower memory and size) of the training set. Then, when a new out of sample sample is queried, there is no need to compute the whole MDS algorithm to be able to transform the data because we have a neural net that can approximate the embedding operation.

Accordingly, the approximate nearest neighbours model designed will have its computational time and memory complexity reduced by the ratio between the original (d) and the low-dimensional space's dimensions (k). We shall also take into account that the higher the level of compression, the more the recall / precision will be degraded.

- Computational time complexity: $O(N^2 * K)$
- Memory complexity: $O(N * K)$

To conclude with, the overall procedure can be summarized in the following manner: when a new sample is queried (in order to find its k-nearest neighbours), it is firstly transformed by the neural net from d dimensions to k, corresponding to the low-dimensional space, by predicting with the parameters learned from MDS. This parameters are learned in cross-validation to ensure that test set samples are unseen. Finally, the K nearest neighbours are computed comparing in the low dimensional space. The objective is to check whether these whole procedure is faster

than a brute force approach. Memory usage is lower and this is a fact, since we only need to store a matrix with a reduced shape.

Algorithm 2 Low dimensional nearest neighbours calculation

1: **procedure** FITTING PROCESS
2: $\boldsymbol{X} \leftarrow$ imputed and standardized dataset
3: $\boldsymbol{X\_mds} \leftarrow$ compute MDS from $\boldsymbol{X}$
4: *neural net model construction*
5: *fit neural net model*
6: *save reduced matrix from training neural net*

Algorithm 3 Low dimensional nearest neighbours calculation

1: **procedure** NN CALCULATION FOR NEW SAMPLE
2: $\boldsymbol{x} \leftarrow$ new sample
3: $\boldsymbol{x\_reduced} \leftarrow$ predicted output from neural net with x (reduction)
4: *nearest neighbours calculation with brute force algorithm in the low dimensional space*

4.2.3 Metrics

With this second implementation, the main metrics to look at correspond to precision and computational measurements, such as time and memory usage. As the objective is to compute nearest neighbours, precision is described as the percentage of nearest neighbours preserved in comparison to a brute force approach in the original space. In the case of time and memory, the measurements are strictly related to the algorithm's performance, thus we can compute how much time they require to find these neighbours and the memory needed.

The following tables relate the model used to compute the neural net with the previous metrics, depending on the dimensions of the low-dimensional space chosen to lower the number of features.

The approximate algorithm hereby proposed can also be compared to other approximate nearest neighbours algorithms such as KD-Tree [8]. KD-Tree is a binary tree that creates divisions of the space using splitting planes which are orthogonal to the axis. Each node has a corresponding region called cell and the root's cell has to include all the samples in the dataset. For each region we have a hyper-plane that divides the subspace in two regions. This is recursively

	npreserved	t_model	t_brute	mem_model	mem_brute
model1	0.565350	0.771945	1.074873	112112	3080112
model2	0.571843	0.785027	1.084613	112112	3080112
model3	0.570347	0.800511	1.067814	112112	3080112
model4	0.567003	0.755689	1.082235	112112	3080112
model5	0.564130	0.796335	1.038609	112112	3080112
model6	0.564723	0.808626	1.021288	112112	3080112
model7	0.569093	0.767841	1.096126	112112	3080112
model8	0.572430	0.922942	0.888819	112112	3080112

Table 4.5: Nearest neighbours calculation in four dimensions

	npreserved	t_model	t_brute	mem_model	mem_brute
model1	0.728400	0.993689	1.204910	224112	3080112
model2	0.780377	0.969519	1.218967	224112	3080112
model3	0.779553	0.971488	1.219617	224112	3080112
model4	0.760667	0.974497	1.152932	224112	3080112
model5	0.761370	0.978987	1.146578	224112	3080112
model6	0.756977	0.990972	1.081272	224112	3080112
model7	0.771057	0.972610	1.216437	224112	3080112
model8	0.771207	1.006541	0.947692	224112	3080112

Table 4.6: Nearest neighbours calculation in eight dimensions

	npreserved	t_model	t_brute	mem_model	mem_brute
model1	0.669377	0.952179	1.175327	504112	3080112
model2	0.855583	0.953779	1.175554	504112	3080112
model3	0.837507	0.950562	1.197267	504112	3080112
model4	0.848860	0.961237	1.126327	504112	3080112
model5	0.844383	0.955949	1.114451	504112	3080112
model6	0.835173	0.966225	1.045986	504112	3080112
model7	0.855930	0.955785	1.174047	504112	3080112
model8	0.853597	0.945284	0.928504	504112	3080112

Table 4.7: Nearest neighbours calculation in eighteen dimensions

repeated until there are no points left or a minimum of samples per leaf is reached. This model works well for datasets with a moderate / small number of dimensions but has no memory improvements. Its computational complexity is $O(NlogN)$. However, when the number of dimensions increase, the behaviour tends to get worse. For small datasets, query times may be even worse than brute force, but its scalability is significantly better. Table 4.8 corresponds to the first dataset of the application.

	npreserved	t_kdtree	t_brute	mem_kdtree	mem_brute
model1	1.0	2.643447	1.293515	6335358	3080112
model2	1.0	2.656460	1.287590	6335358	3080112
model3	1.0	2.633159	1.281591	6335358	3080112
model4	1.0	2.661036	1.283720	6335358	3080112
model5	1.0	2.630308	1.291127	6335358	3080112
model6	1.0	2.671045	1.281855	6335358	3080112
model7	1.0	2.639825	1.279909	6335358	3080112
model8	1.0	2.632470	1.287737	6335358	3080112

Table 4.8: KD-Tree computing nearest neighbours

Although the total amount of neighbours are preserved by KD-tree, both the computational time and memory used by the model are high, two times compared to brute force algorithm. The next two tables compare the approximate model and the kd-tree approximate nearest neighbours computation with a new dataset that has 4000 instances and 281 features per sample.

	npreserved	t_model	t_brute	mem_model	mem_brute
model1	0.30045	0.152567	0.206344	1200112	6744112
model2	0.43732	0.140289	0.172093	1200112	6744112
model3	0.38892	0.139835	0.178780	1200112	6744112
model4	0.46497	0.132087	0.177754	1200112	6744112
model5	0.41835	0.132285	0.173796	1200112	6744112
model6	0.49443	0.133642	0.174198	1200112	6744112
model7	0.50684	0.136121	0.174210	1200112	6744112
model8	0.59107	0.136980	0.184086	1200112	6744112

Table 4.9: Nearest neighbours calculation in 100 dimensions from new dataset

The computational time KD-Tree requires is twelve times the computational time from the

	npreserved	t_kdtree	t_brute	mem_kdtree	mem_brute
model1	0.99992	2.211247	0.383505	13808037	6744112
model2	0.99992	2.088237	0.442901	13808037	6744112
model3	0.99992	1.973441	0.446145	13808037	6744112
model4	0.99992	1.844881	0.388873	13808037	6744112
model5	0.99992	2.136054	0.422400	13808037	6744112
model6	0.99992	1.878483	0.433052	13808037	6744112
model7	0.99992	2.173580	0.405178	13808037	6744112
model8	0.99992	1.897629	0.438579	13808037	6744112

Table 4.10: KD-Tree computing nearest neighbours from new dataset

approximate model in the best of the cases. The memory KD-Tree needs to compute its operations is nearly also twelve times more than for the model designed in this experiment.

4.2.4 Conclusion

Computing nearest neighbours in nowadays applications has become an important issue. The need to report them live (for instance for advertisement recommendation) demands new state-of-the-art algorithms like the model previously proposed. It consists in lowering the sample's dimensions with a neural net which has learned how to replicate a MDS, and then, nearest neighbours are computed in the low-dimensional space.

As seen in the results studied before, there is a clear relation between precision obtained and time and memory usage. We can sacrifice a bit of recall and get huge improvements in memory usage, which can reduce application costs considerably.

Although KD-Tree also uses the idea of creating a non linear kernel, regarding precision, time and memory usage this gets higher when a complex dataset is provided, and it has no benefits regarding memory usage.

The computational complexities for the three algorithms studied are the next ones (taking into account dimensions computed):

- Brute force algorithm: $O(N^2 * D)$

– KD-Tree algorithm: $O(NlogN * D)$

– Approximate nearest neighbours algorithm: $O(N^2 * D^*) = O(N^2 * \frac{D}{3})$

The following figure illustrates computational complexities for an increasing number of dimensions:

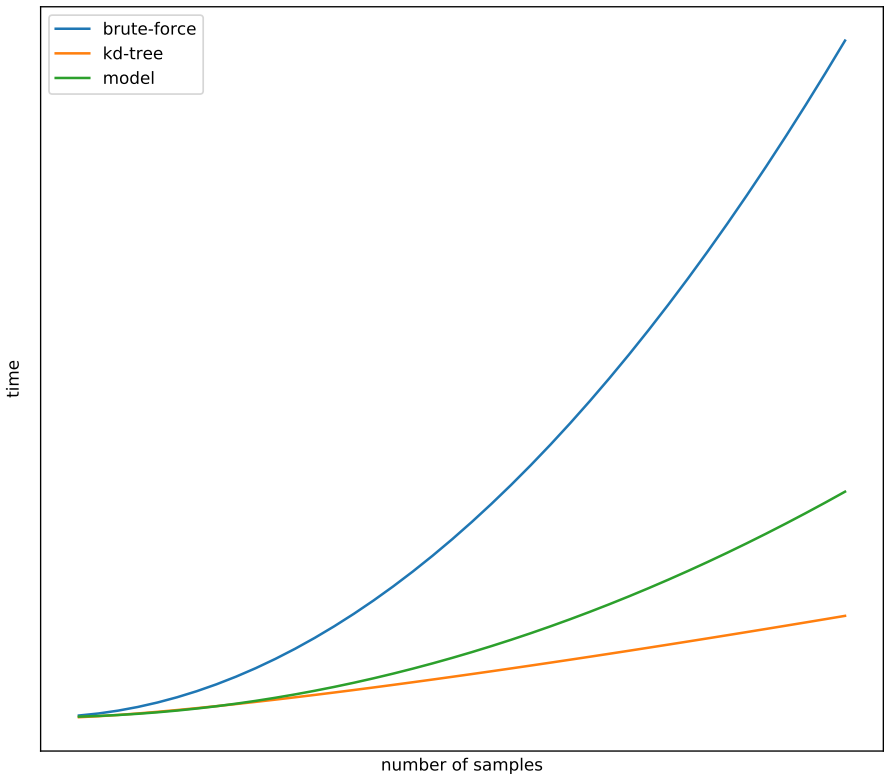

Figure 4.6: Algorithms' complexities.

4.2.5 Code

```
from sklearn.preprocessing import Imputer
from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import NearestNeighbors
from sklearn.manifold import MDS
from sklearn.pipeline import Pipeline
from sklearn.model_selection import train_test_split
import timeit
# datasets
from utils import load_page, load_bankruptcy
from utils import dist_loss
# numpy
import numpy as np
import pandas as pd
import tensorflow as tf
import keras.backend as K
import matplotlib.pyplot as plt
import sys
import pickle
```

```
import os
import tempfile

import timeit
from joblib import Parallel, delayed

# function to see neighbors preserved


def devidedisin(x, split=None):
    if not split:
        split = int(x.shape[0]/2)
    return np.isin(x[split:], x[0:split])


def nn_preserved(y_true, y_pred, n_true, n_pred):
    conc = np.hstack((y_pred, y_true))
    return np.apply_along_axis(devidedisin, 1, conc, split=n_pred).mean(axis=1)


def compare(X_testing, pipe1, pipe2, k1, k2):

    neighbors1 = pipe1.kneighbors(
        X_testing, n_neighbors=k1, return_distance=False)

    neighbors2 = pipe2.kneighbors(
        X_testing, n_neighbors=k2, return_distance=False)

    timenn = pipe1.get_query_time(
        X_testing, n_neighbors=1000, return_distance=False)

    timereal = pipe2.get_query_time(
        X_testing, n_neighbors=1000, return_distance=False)

    return nn_preserved(neighbors2, neighbors1, k2, k1), timenn, timereal


def get_memory_object(obj):
    fname = tempfile.mktemp()
    pickle.dump(obj, open(fname, 'wb'))
    size = os.path.getsize(fname)
    os.remove(fname)
    return size


def modelsjob(name, params):

    nnreplicator = NNReplicator(
        md, params['layers'], params['dropout'],
        0.1, 'sigmoid', dist_loss, 200, 10)

    pipeline1 = NNPipeline([
        ('imp', Imputer()),
        ('std', StandardScaler()),
        # ('nnrep', nnreplicator),
        ('knn', NearestNeighbors(algorithm='kd_tree', leaf_size=100))])

    # training
    pipeline1.fit(X_train)
    mem_model = get_memory_object(pipeline1)

    # pipeline 2 = TRUE
    pipeline2 = NNPipeline([
        ('imp', Imputer()),
        ('std', StandardScaler()),
        ('knn', NearestNeighbors(algorithm='brute'))])

    # training
    pipeline2.fit(X_train)
    mem_real = pipeline2.get_memory_usage(X_train)

    preserved, time_model, time_real = compare(
        X_test, pipeline1, pipeline2, k1=100, k2=100)
```

```
    return np.mean(preserved), time_model, time_real, mem_model, mem_real


def comp_brute(x):
    return np.square(x)


def comp_kdtree(x):
    return x*np.log(x)


def comp_model(x):
    return np.square(x)/3

# main
if __name__ == '__main__':

    # Set tf to one thread
    # config = tf.ConfigProto(
    #     intra_op_parallelism_threads=1,
    #     inter_op_parallelism_threads=1,
    #     allow_soft_placement=True,
    #     device_count={'CPU': 1})
    # session = tf.Session(config=config)
    # K.set_session(session)

    # dataset
    data = pd.read_csv('../datasets/blog.csv')

    X, y = data.iloc[:, :-1], data.iloc[:, -1]
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25)

    md = MDS(n_components=100, random_state=0, n_jobs=1,
             verbose=10, n_init=1)

    models = {
        "model1": {
            "layers": [32, 16, 16, 8],
            "dropout": [0, 0, 0, 0],
            "type": 'deep - narrow'
        },
        "model2": {
            "layers": [32, 32],
            "dropout": [0, 0],
            "type": 'shallow - narrow'
        },
        "model3": {
            "layers": [32, 16],
            "dropout": [0, 0],
            "type": 'shallow - narrow'
        },
        "model4": {
            "layers": [128, 64, 64],
            "dropout": [0, 0, 0],
            "type": 'deep - wide'
        },
        "model5": {
            "layers": [128, 64, 64, 32],
            "dropout": [0, 0, 0, 0],
            "type": 'deep - wide'
        },
        "model6": {
            "layers": [128, 128, 128],
            "dropout": [0, 0, 0],
            "type": 'shallow - wide'
        },
        "model7": {
            "layers": [128, 64],
            "dropout": [0, 0],
            "type": 'shallow - wide'
        },
        "model8": {
```

```
        "layers": [1000],
        "dropout": [0],
        "type": 'shallow - wider'
    }
}

results = Parallel(n_jobs=1, verbose=10)(
    delayed(modelsjob)(model_name, model_params)
    for model_name, model_params in models.items()
)

resultsdf = pd.DataFrame(
    results,
    columns=['npreserved', 't_model', 't_brute', 'mem_model', 'mem_brute'],
    index=models)
resultsdf.to_latex(buf='../figures/app2kdtreebrute100100blog.tex')

# complexities calculation

indices1 = np.linspace(1, 20, 40)

out1 = comp_brute(indices1)
out2 = comp_kdtree(indices1)
out3 = comp_model(indices1)

fig, ax = plt.subplots(1, 1, figsize=(9, 8))
ax.set_ylabel('time')
ax.set_xlabel('number of samples')
ax.set_yticklabels({})
ax.set_xticklabels({})
plt.tick_params(axis='x', which='both', bottom='off',
                left='off', labelbottom='off')
plt.tick_params(axis='y', which='both', bottom='off',
                left='off', labelbottom='off')
plt.plot(indices1, out1, label="brute-force")
plt.plot(indices1, out2, label="kd-tree")
plt.plot(indices1, out3, label="model")
plt.legend()
plt.savefig('../text/figures/complexities.pdf', bbox_inches='tight')
```

4.3 Python Classes

The following python classes are compatible with the *Scikit-Learn* API and can be used along with *Keras* and *Tensorflow* to replicate any unsupervised algorithm. These classes were also tested for other algorithms such as PCA, Isomap, Locally Linear Embeddings and others, and gave extremely good results in all cases. It allows for a flexible and scalable definition of the neural net structure and can be easily integrated in machine learning pipelines (such as the ones in the Scikit Learn API) and deployed in a real application.

Using these classes one would be able to build an application such as the following: using Tf-IDF representations of random texts we could train a t-SNE and create a 2D visualization of these texts where close texts would have a clear semantic similarity. We could even cluster these texts in this low dimensional space and discover the semantic properties of all zones in it. Then by using a replicator class, we can train a neural net to learn the t-SNE embeddings and be

able to visualize the semantic properties of any given text pasted by the user in the application.

```
from sklearn.base import TransformerMixin
from sklearn.metrics.pairwise import pairwise_distances

from keras.models import Sequential
from keras.layers import Dense, Activation, Dropout
from keras import optimizers
from keras.wrappers.scikit_learn import KerasRegressor

from sklearn.pipeline import Pipeline
from sklearn.neighbors import NearestNeighbors

import sys
import time


class NNReplicator(TransformerMixin):

    def __init__(self,
                 embedder,
                 layers, dropout, lr, act_func, loss_func, epochs, batch_size):

        self.embedder = embedder
        self.layers = layers
        self.dropout = dropout
        self.lr = lr
        self.act_func = act_func
        self.loss_func = loss_func
        self.epochs = epochs
        self.batch_size = batch_size

    def nnConstruct(self, shape):

        model = Sequential()

        for i, (layer, drop) in enumerate(zip(self.layers, self.dropout)):

            if i == 0:
                model.add(Dense(layer, input_shape=(
                    shape,), activation=self.act_func))
            else:
                model.add(Dense(layer, activation=self.act_func))

            model.add(Dropout(drop))

        model.add(Dense(self.embedder.n_components, activation='linear'))
        ada = optimizers.Adagrad(lr=self.lr)

        model.compile(optimizer=ada, loss=self.loss_func)

        self.krObject = KerasRegressor(
            lambda: model, epochs=self.epochs, batch_size=self.batch_size)

    def fit(self, X, y=None):

        shape = X.shape[1]
        self.nnConstruct(shape)

        X_ = self.embedder.fit_transform(X)

        self.krObject.fit(X, X_)
        return self

    def transform(self, X):
        return self.krObject.predict(X)


class NNPipeline(Pipeline):

    def __init__(self, steps, memory=None):
```

```
    super(NNPipeline, self).__init__(
        steps=steps, memory=memory)

    assert isinstance(self.steps[-1][1], NearestNeighbors)

def transform(self, X):
    X_ = X.copy()
    for step in self.steps[:-1]:
        X_ = step[1].transform(X_)
    return X_

def kneighbors(self, X, n_neighbors, return_distance):

    # Finally, run kneighbors method using the last step (NN)
    return self.steps[-1][1].kneighbors(
        X=self.transform(X), n_neighbors=n_neighbors,
        return_distance=return_distance
    )

def get_memory_usage(self, X):

    return sys.getsizeof(self.transform(X))

def get_query_time(self, X, n_neighbors, return_distance, number=100):

    X_ = self.transform(X)
    time_func = lambda: self.steps[-1][1].kneighbors(
        X=X_, n_neighbors=n_neighbors,
        return_distance=return_distance
    )
    return timeit.timeit(time_func, number=number) / number
```

Chapter 5

Conclusions and Future Work

Neural nets are capable of replicating and modeling almost any problem and we hereby proved that also dimensionality reduction models' transformations from algorithms such as MDS and t-SNE can be reproduced by them. Furthermore, these models guarantee a re-usable tool that can be applied alone to computationally speed up processes and lessen memory usage, as a way of visualizing data or as a basis to other applications like nearest neighbours search. As they faithfully represent both dimensionality reduction algorithms, the model could be used as a basis for whatever machine learning problem which requires previous dimensionality reduction.

For the two applications developed, we can reassure that the best neural net model is a wide net, although for replicating other dimensionality reduction algorithms would be important to know which one fits the best.

Bibliography

[1] Ella Bingham and Heikki Mannila. Random projection in dimensionality reduction: Applications to image and text data. 2015. 3

[2] Siena College. Brute-force algorithms. 27

[3] Jan de Leeuw and Patrick Mair. Multidimensional scaling using majorization: SMACOF. *Journal of Statistical Software*, 2009.

[4] Dasika Ratna Deepthi, Sujeet Kuchibholta, and K. Eswaran. Dimensionality reduction and reconstruction using mirroring neural networks and object recognition based on reduced dimension characteristic vector. 3

[5] Tobias Holl. Principal component analysis. *Proseminar Data Mining*. 3, 4

[6] Jieping Ye Lei Yu and Huan Liu. Dimensionality reduction for data mining. techniques, applications and trends.

[7] M. Lichman. UCI machine learning repository, 2013. 26

[8] Songrit Maneewongvatana and David M. Mount. On the efficiency of nearest neighbor searching with data clustered in lower dimensions. 29

[9] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011. 14

[10] Sebastian Ruder. An overview of gradient descent optimization algorithms. 2016.

[11] Joshua B. Tenenbaum, Vin de Silva, and John C. Langford. A global geometric framework for nonlinear dimensionality reduction. *Science vol 290*, 2000. 3

[12] Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-SNE. *Journal of Machine Learning Research*, 2008. 3, 6

[13] Jing Wang, Haibo He, and Danil V. Prokhorov. A folded neural network autoencoder for dimensionality reduction. 2012. 3

[14] Rachel Ward. Dimension reduction via random projections. 2014. 3

[15] Florian Wickelmaier. An introduction to MDS. 2003. 3, 5