Development of unsupervised learning transformations through supervised learning methods.

Author: Patricia Cortajarena Sauca

Ponente: Carlos Roberto del Blanco Adán

Tutor: Iñigo Cortajarena Sauca

Trabajo Fin de Grado

ETSIT UPM

Madrid. January, 2018

Abstract

The aim of this project is

Acknowledgements

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 means 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 in huge implementations are saved.

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 would suite for 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.

So 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 reduced data.

The non-parametric algorithms have an important disadvantage: as it is said, they don't have parameters which represent the dimensionality reduction; thus, when a new datapoint is provided we need to compute the whole algorithm again to obtain the datapoint's representation in the lower space, instead of computing only the new data.

We hereby propose the study and research of an implementation which replicates the behaviour of a non-parametric dimensionality reduction algorithm. The main objective is to solve the non-parametric's main disadvantage and make the new algorithm capable of being repeatedly used with new datapoint examples, omiting the need to compute the whole dimensionality reduction algorithm every time.
The dimensionality reduction algorithms used to base the 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 resume 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 development of the implementation. Afterwards, chapter 4 describes the main implementation's applications and the corresponding metrics computed to analyse the algorithm's behaviour. The last chapter (chapter 6) comprehends the conclusion and resumes the main ideas acquired.

Chapter 2

State-of-The-Art

The most classical dimensionality reduction techniques are know to be PCA [5] and MDS [14]. They preserve the true structure of the data even though they are simple to implement and efficiently computable. They rely on linear models to compute data mining on near lower dimensional spaces. A related linear implementation which is based on random projectations of the data is described in [1, 13]. Some other well-known techniques are ISOMAP [10] and t-SNE [11], but their implementations often require more memory and computational time, yet the complexity of the algorithms is higher. Nevertheless, this algorithms are capable of reducing the number of features in non-linear data, where linear models can not faithfully reprensent the data's structure. ISOMAP computes geodesic distances instead of taking into account only euclidean distances while t-SNE's approach calculates probabilities from euclidean distances.

Neural networks have become an essencial tool in machine learning problems and it is certainly used in dimensionality reduction. The autoencoders [4, 12] consist on a neuralnet with decreasing number of neurons per layer and then symmetrically more layers are added with the same corresponding increasing neurons per layer. If the output represents without distorsion the input to the net, the middle layer with the fewest number of neurons corresponds to the reduced description of the data. This is represented in figure 2.1.

The next sections (sections 2.1, 2.2 and 2.3) resume the main ideas of the algorithms in which the research is based.

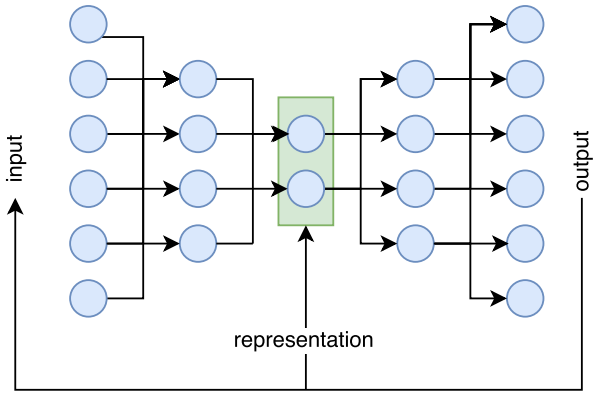

Figure 2.1: Autoencoder neural network.

2.1 PCA: Principal Component Analysis

Principal Component Analysis [5] algorithm is based on reducing the number of features by processing the correlations between the features of the datapoints. The aim is to eliminate this correlations by transforming the matrix $\mathbf{X} \in \Re^{mxn}$ (with m being the number of data points and n de number of features) into an orthogonal basis. By omiting 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. We obtain it by using the next expression:

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

Because the aim of the PCA is to eliminate the correlations, the covariance matrix of the result $\mathbf{Y}$ should be a diagonal matrix with just the variances of the columns.
PCA is famous because of a great advantage: we can find a linear transformation ($\mathbf{Y} = \mathbf{XP}$), which makes this a parametric model, easy to reuse and quite computationally simple.
For symmetric matrices ($\mathbf{X}$) we can find eigenvalue decomposition with a diagonal matrix ($\mathbf{Y}$), matching exactly with our linear problem with $\mathbf{X}$ and $\mathbf{Y}$.

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

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

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

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

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 eigenvalues matrix obtained ($\text{cov}(\mathbf{Y})$) are sorted decreasingly and are the orthogonal basis values. Choosing the $\mathbf{N}$ first values of this matrix, being $\mathbf{N}$ the desired output dimension, and computing the corresponding eigenvectors, we obtain our reduced dimensionally points, as a basis transformation of our datapoints from the original dataset, by the new basis coordinates.

2.2 MDS: Multidimensional Scaling

Multidimensional Scaling [14] in dimensionality reduction tries to create a map which displays the relative distances between the data points. This focuses on getting a lower dimensional map, keeping as much distance information as possible. In case of visualizing data, this dimensional map needs to be a one, two or, at most, three dimensional space.
MDS calculates a metric or non-metric solution depending on the data provided, which has to be a *proximity* matrix. This *proximity* matrix quantifies how close the datapoints are. 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 explained in the previous section. The *proximity* matrix is always symmetric and somehow describes the relations between the features, so basically we can treat our problem as a variation of the PCA algorithm.
Metric MDS performs the same steps as in PCA, with the modification of being a distance matrix the one computed in this problem.
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 meassure 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, so MDS turns to be non-parametric.

2.2.1 Stress metric

As in every data problem, we need a metric which shows how well the performance given a particular dataset is. In Multidimensional Scaling we compute the *stress* measure that compares the predicted distances with the original ones. Note that obviously this depends on the number of dimensions we want to keep, yet if we lower the number of dimensions, the stress will get higher, because we are representing the same distances relations in a lower dimensional space.

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

d_{ij} represents the original distance and d'_{ij} is the predicted distance 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 lower, relating zero stress to the perfect performance of the MDS algorithm.

2.3 TSNE: T-Stochastic Neighbour Embedding

T-Stochastic Neighbour Embedding [11] is our non-linear example of dimensionality reduction. It relies on Stochastic Neighbour Embedding (SNE). The main reason why this algorithm is used is because it is capable of representing both the local and the global structure of the original data. As said, 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. That means we measure the similarity of two points x_i, x_j as the probability $p_{i|j}$ of considering the second one as a neighbour of the first. The probability in the original and in the low dimensional space is computated as seen:

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

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

This expressions represent gaussian distributions centered in each datapoint x_i, where σ_i is the example's variance. The conditional probability represents whether the point x_j is considered a datapoint's neighbour or not. 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 appropiate than in a sparse region. Having decided which value to use for each of the datapoints, 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 $q_{i|j}$ with all the datapoints and consequently obtain the algorithm's behaviour by analysing how many neighbours have been mantained 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 diferent values of the variance depending on the point and the "crowding problem". T-SNE tries to solve this limitations as in the next section is described.

2.3.2 t-SNE

Although SNE is capable of showing good visualizations, t-SNE was proposed as a modified algorithm which tried to make its behaviour more accurate and easy to compute.
First of all, instead of minimizing the sum of the different KL divergences along all the datapoints, another way of computing the cost is presented: we are trying to minimize a single KL divergence between a joint probability P and the same in the low dimensional space, Q. With this symmetric aproach, we ommit the need to obtain the variance value for each datapoint and the cost function is much easier to compute and optimize, so does the gradient.

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

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

Secondly, t-SNE handles with the problem known as "crowding problem" 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 the mutually equidistant points when reducing from high-dimensional space to low-dimensional space. This task tends to be quite difficult because the area available in a lower space is less than in the higher one. In the end, the points tend to crush together in the center of the map, so the result is the impossibility of representing the true distances from the original dataset.
The t-Student distribution is considered a heavy-tailed distribution because it allows to represent a moderate distance as a much larger distance in the map without any inconvenience. Therefore, 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 related 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 this 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 development of a supervised learning tool which learns the embedding derived by an unsupervised learning method, which could be used in real machine learning applications.

The two main supervised learning algorithms used in today's applications are especially Neural Networks and Decision Trees. To know which of them or even if both could suit our problem we study their main characteristics and differences.
First of all, decision trees are known to be faster and easier to train, and their results are very interpretable. On the other hand, neuralnets are slower and less interpretable. Neural networks are prone to overfitting if the size of the given dataset is not large enough or if the parameters learnt are not the best in the algorithm's performance, while decision trees are less prone to overfit if pruned (eliminating the tree's branches which do not contribute correctly to the classification). What really makes the difference from one another is the fact that neuralnets are capable of having two output values or the number of values required as outputs whether the decision trees only show one output per performance. If we wanted to extract more than one value we would be required to have one diferent tree for each output. That increases computational complexity as the number of output dimensions increase.

Taking into account the characteristics and differences between neuralnets and decision trees, the main supervised learning algorithm tested in our development are neural networks, yet it is a powerful and efficient tool which can learn from easy functions up to complicated non-linear models. 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 the parameters learned in the fitting process.

Multidimensional Scaling and T-Stochastic Neighbour Embedding are both non-parametric dimensionality reduction algorithms. As it was explained in the introduction (Chapter 1), this algorithms can not be expressed in terms of parameters, so this means once the algorithm is computed, if we want to add a new datapoint to the dataset and then obtain its reduced embedded vector, it would be necessary to compute everything again so that that the total ouput includes the new datapoint's output too. Thus, this means a lot of computational work.

Going in depth into the problem, that means if we have a set of **m** datapoints with **n** features each, and we want to represent those points in a two-dimensional space, we look forward to using a neuralnet (supervised algorithm) which learns that embedding from **n** dimensions into two, replicating any of the previously mentioned dimensionality reduction algorithms (unsupervised non-parametric algorithm).
The neural network needs to be trained at least once, so we need to compute the dimensionality reduction algorithm once (step 1, figure 3.1) to input the dataset into the neuralnet and set the output as the reduced values just computed. Having the neuralnet trained (step 2, figure 3.1), as soon as we get a new datapoint, instead of recalculating the MDS or t-SNE algorithm, we can reuse the neuralnet's parameters to obtain the predictions (step 3, figure 3.1) which will be the dimensionally reduced value from the new examples.

With this implementation we would be able to speed up processes, reduce memory and also computational times.

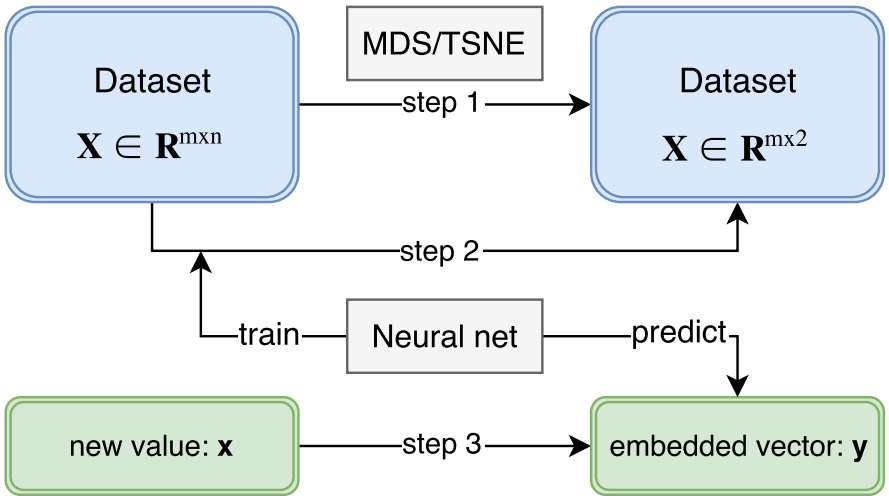

Figure 3.1: Dimensionality reduction algorithm learned by a Neural Network

3.1 Cross Validation

Cross validation is a technique to evaluate the predictive models to get to know if the model is capturing the general behaviour of the data to suit new examples, and moreover estimate how accurate the model is performing.

The easiest way to know whether a model is underfitting or overfitting is dividing the dataset into train datapoints and test datapoints, choosing for example 80% to train the model and 20% to test the behaviour with datapoints which have not been used to adjust the model. Although this is a good first approach, we might be fitting our model to the test points without noticing. This means we might be choosing the model's parameters to fit the test set, and still don't capture the general behaviour of the problem.

A more accurate approach would be cross validation. It consists on dividing the data into a number of folds usually depending on the dataset. Choosing between three to five folds would be the best decision. The algorithm is then trained the same number of times as folds, choosing in each iteration one of the folds to test and the rest of them to train. As we have trained and tested the model with disjoint values, we are not fitting our model to certain values. We use a losses metric to cuantify this result and to conclude we calculate a mean of all of them to obtain a unique value to represent the total behaviour. A schema is shown below to represent what cross validation is doing.

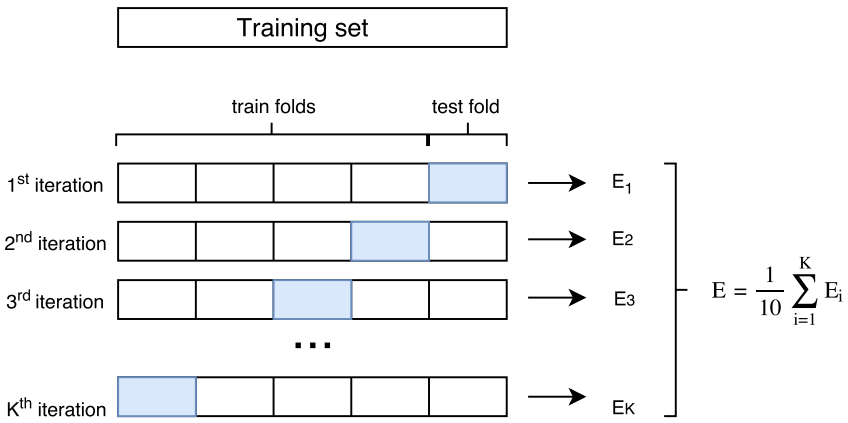

Figure 3.2: Cross validation using K-Folds.

Chapter 4

Applications

4.1 Application 1: Replicating t-SNE with Neuralnet

The first implementation to be described is the model which replicates the non-parametric dimensionality reduction algorithm. In this concrete example, T-Stochastic Neighbour Embedding is used as the problem to carry out. The main objective is to assess whether the supervised neuralnet is capable of replicating the previous algorithm.

4.1.1 Dataset

The dataset *"load-digits"* is the data used to develop this example and is extracted from the Scikit-Learn datasets [8]. It consists on 1797 instances of handwritten digits between 0 to 9 from diferent writers with 64 features each. This features display the values of grayscale intensity for each pixel in the image. This means our images are 8x8 pixel images which are represented by 64 total features containing intensities from 0 (completely black) to 16 (pure white) as show in figure 4.1. To translate this into a dataset, each image is represented in a row with its corresponding 64 values and including a last column with the label of the number represented in the image.

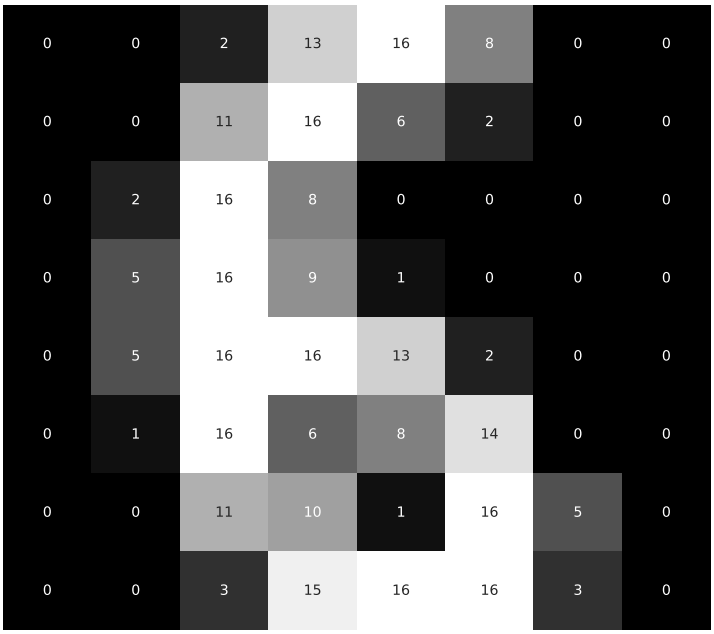

Figure 4.1: Example dataset point representing a handwriten digit.

4.1.2 Algorithm: t-SNE and Neuralnet

This section comprehends the next points to take into account: the computation of the dimensionality reduction algorithm and its implementation in the neuralnet model.

As a basis to the problem, t-SNE is computed to all the datapoints. Previously, standardizating features in the original data by removing the mean and scaling to unit variance is computed for each of the 64 variables, which achieves better results afterwards. Due to the dataset composed by 1797 instances with 64 features each, a two dimensional space reduction is calculated, looking forward to visualizating the data.

Once reduced data is obtained, both the original and the just calculated points are divided into two corresponding folds, one being a subsample for training the neuralnet and the other to test the model. Therefore, overfitting is avoided by using completely different data to train than to test if the neuralnet is capturing the general behaviour of the dimensionality reduction algorithm.

To continue, the next to keep in mind is the neuralnet's construction. There are several types of neuralnets depending on how it's built, mainly if they are composed by a few layers (1-2 layers) or more layers, and the number of neurons per layer, obtaining shallow or deep and wide or

narrow neuralnets correspondingly. Depending on the net computed, the results gathered will differ from one another, thus on section 4.1.3 the diferences between them are shown.

Once the net is stablished, the standardized original data from the training subsample turns to be the net's input in the fitting process. The output will always be a two variable output, corresponding to the two values which represent the dimension reduction. At this point, the model also uses the training subsample from the reduced points originated in the t-SNE's computation as the output to obtain and to learn from.
The neuralnet seeks to get the same output as the dimensionality reduction algorithm, that is why the metric used to minimize the error tends to be a distance metric. It tries to reduce the distance between the values from t-SNE and the values being learned by the model. In this case the distance metric which obtains better results corresponds to the mean of the distances between corresponding points, but other metrics can also be used such as mean loss (mean of the square of the difference between points).

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 reduced data replicating what t-SNE would compute but avoiding the need to recalculate the whole algorithm.

Algorithm 1 Neuralnet replicating process

procedure REPLICATING PROCEDURE
data obtaining
    X ← standardized input matrix
    X_tsne ← compute t-SNE from ***X***
    X_train; X_test; X_tsne_train; X_tsne_test; ← train and test data computation
neuralnet model construction
fit neuralnet model
    X_train ← neuralnet's input
    X_tsne_train ← neuralnet's output
    distance error ← neuralnet's error metric
predict neuralnet model
    64 feature input (new example) → neuralnet → predict output = dimension reduction

4.1.3 Metrics

With this first implementation we look forward to obtain reduced data in less computational time using the most accurate neuralnet model depending on its construction. Depending on the type of neuralnet used, times and errors are calculated in the following tables (tables 4.1 and 4.2).

The most important point by now is that this measurements are done with examples from the test subsample which have not been used by the neuralnet to compute its fitting process, thus they indicate the general behaviour of the model.

Table 4.1 includes the model construction and the errors computed. The model construction depends on the layers and the number of neurons per layer and on the dropout applied for each layer. Those two values are declared with arrays as in the table is shown. As a resume, the type of the neuralnet built is summarized in deep or shallow and wide or narrow nets.
The out of sample error computed is again a distance error which evaluates the difference between what the t-SNE computed and what the model is predicting. The error variable then corresponds to the mean of the Euclidean distances between the two algorithms. Percentages are also included to reckon whether the total error compared to the range of values acquired by the algorithms stand in a good proportion.

	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

Neuralnet's background consists on matrix scalar multiplications of vectors with learnt parameters and activation functions, hyperbolic tangent in this problem, which are very efficient

computationally operations.

Table 4.2 shows the huge difference between the computational time (in seconds) of the dimensionality reduction algorithm and the neuralnet's development for the same number of points, refering to the whole dataset (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 4.2: Computational times

4.1.4 Conclusions

Consequently and given the results in the previous tables, we can reassure that, without considerating the neuralnet's construction, all of them will certainly be quicker in obtaining the results. If we look for accuracy, a wide neuralnet is better at replicating the t-SNE algorithm. In particular, the model 6 proposed suits the best for the problem presented.

The next two figures represent the results obtained. The first one matches the predictions from the neural network (in model 6) and the second one represents the same points from the t-SNE algorithm.

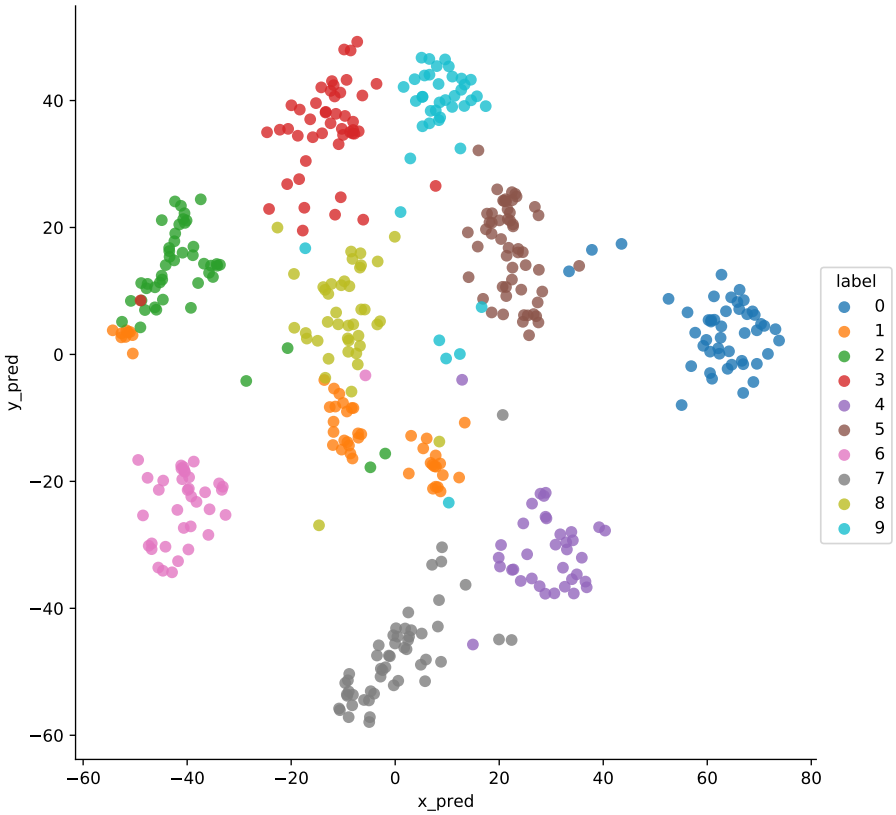

Figure 4.2: Low dimensional space from NeuralNet.

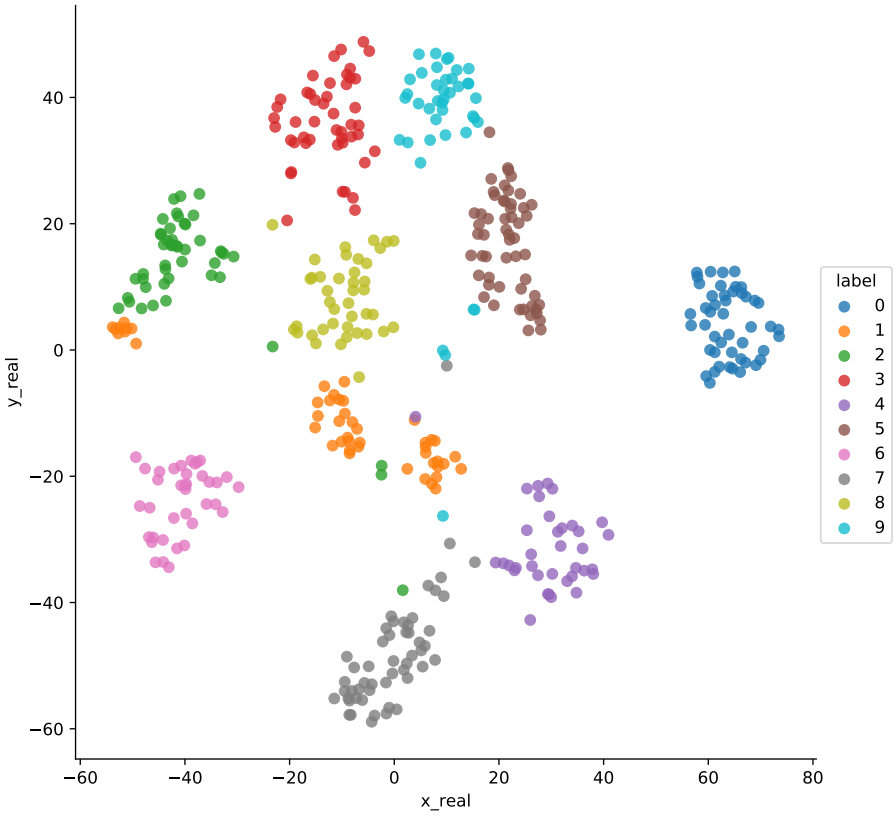

Figure 4.3: Low dimensional space from from t-SNE

4.2 Application 2: Nearest Neighbours calculation

The second application consists on the computation of nearest neighbours in a low-dimensional space. This second implementation focuses on the Multidimensional Scaling algorithm. The aim is to compute the nearest neighbours in the output coordinates from MDS. A loss in precision and recall is involved in the nearest neighbours computation since an error in computation is made when calculating the low dimensional equivalent points. Nevertheless, it allows for a faster, low memory intensive algorithm.

4.2.1 Dataset

The dataset *"forests.csv"* is used for this application and is extracted from UCI Machine Learning Repository [6]. For convenience, a random subsample of 20.000 examples is storaged with 55 features each. These are all numeric variables that describe several characteristics of the forest. The forest type output is represented by a number in range 1 to 7, shaping the last column of the dataset.
The reason why this dataset is chosen for this concrete 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: Low dimensional nearest neighbors obtaining

The main task contemplated in this application is the computation of nearest neighbours. The following research bears with reusability and time and memory usage reduction algorithms.

Regarding the first point, because of MDS being non-parametric, it is uncapable of obtaining new embedded points without computing again all the operations. This means if we have already output the low-dimensional map of points from the dataset, it would not be posible to obtain a new point in this map unless the algorithm is recomputed. If we based our nearest neighbours search in a previous dimensionality reduction algorithm such as MDS, the only results available would be from the points just computed by MDS.

Nearest neighbours brute force models [2] are known to be eficiently poor when a raw dataset is provided, both because their memory tends to be big and thus the computational time required

also increases.

Nearest neighbours brute force algorithm computes the euclidean distances between each point and the rest of the data and then sortens them according to their distances, from lowest to highest, for each of the examples in the dataset. This means the computational complexity corresponds to $O(n^2)$. Since the euclidean distances is computed as:

$$dist(p,q) = \sqrt{(q_1 - p_1)^2 + (q_2 - p_2)^2 + ... + (q_n - p_n)^2} = \sqrt{\sum_{i=1}^{n}(q_i - p_i)^2}$$

The brute force algorithm's complexity in terms of dimensions computed, d, is $O(n^2 * d)$.

As for the memory and taking into account the previous ideas of how to compute neighbours, this model requires the original matrix to be stored which means the memory complexity also depends on the number of points and their dimensions and is resumed in a $O(n * d)$ model.

The next model is a new solution proposed to solve the problems just mentioned.

The algorithm involves two phases: one fitting process and its later usage for new points. This is why the original dataset is splitted into two folds, one for the training and the second one to test out of sample performance.

The training phase aims to obtain all the 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 tranformation 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. Instead, in this proposal, and as seen in the first application (4.1), the training consists on faithfully learning the mapping of a dimensionality reduction algorithm from a high-dimensional space to a low-dimensional space by training a neuralnet to do so. In this case, the model saves the neuralnet's parameters which replicate the MDS, instead of the exact transformations from it. Then, when a new out of sample point is included, there is no need to compute the whole MDS algorithm to be able to transform the data because we have a neuralnet that can approximate the embedding operation.

Accordingly, the approximate nearest neighbours model designed will have its computational time and memory complexity reduced by the relation between the original (d) and the low-dimensional space's dimensions (k).

- Computational time complexity: $O(n^2 * k)$

– Memory complexity: $O(n * k)$

To conclude, when a new sample is requested to find the 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 embedding learnt from MDS. Afterwards, the nearest neighbours are computed comparing the new example with the rest of the low dimensional samples, achieving a faster and low memory intensive algorithm.

Algorithm 2 Low dimensional nearest neighbours calculation

```
procedure FITTING PROCESS
    X ← imputed and standardized dataset
    X_mds ← compute MDS from X
neuralnet model construction
fit neural net model
save reduced matrix from training neural net
```

Algorithm 3 Low dimensional nearest neighbours calculation

```
procedure NN CALCULATION FOR NEW SAMPLE
    x ← new sample
    x_reduced ← predicted output from neural net with x (reduction)
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 finally compute nearest neighbours, precision is described as the percentage of nearest neighbours preserved in comparison to bruce force algorithm. In the case of time and memory, the measurements are strictly related to the algorithm's performance, thus we calculate how much time they require to compute the neighbours and the memory the data needs to do so.

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 data's features.

	npreserved	nntime	realtime	mem_model	mem_real
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 4.3: Nearest neighbours calculation in four dimensions

	npreserved	nntime	realtime	mem_model	mem_real
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 4.4: Nearest neighbours calculation in eight dimensions

	npreserved	nntime	realtime	mem_model	mem_real
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 4.5: Nearest neighbours calculation in eighteen dimensions

The approximate model designed can also be compared to other approximate nearest neighbours algorithms such as KD-Tree [7]. KD-Tree is a binary tree that represents a division 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 points from the dataset. For each region we have a hyperplane which divides the points into one side or the other, until there are no points left, which means the partition is over and the tree is constructed. This type of tree achieves a computational complexity of $O(nlogn)$ and works correctly for low-dimensional datasets. When the dimensions get higher, the behaviour tends to get worse. Table 4.6 corresponds to the first dataset of the application.

	npreserved	nntime	realtime	mem_model	mem_real
model1	1.0	2.599825	1.173718	3080112	3080112
model2	1.0	2.633631	1.157942	3080112	3080112
model3	1.0	2.624684	1.168484	3080112	3080112
model4	1.0	2.613410	1.153366	3080112	3080112
model5	1.0	2.614144	1.169271	3080112	3080112
model6	1.0	2.609788	1.184947	3080112	3080112
model7	1.0	2.637135	1.164681	3080112	3080112
model8	1.0	2.612835	1.174181	3080112	3080112

Table 4.6: KD-Tree computing nearest neighbours

We have developed a supervised algorithm, based in neuralnets, which reproduces the ideas showed in the framework's introduction. The inputs expected to run this algorithm are the next ones:

- Number of components for the low-dimensional space
- Dimensionality reduction object, such as PCA, MDS or TSNE in our examples.
- Layers: the number of layers and neurons per layer, taking into account that the last layer has the same neurons as the number of components in the low dimensional space; e.g: layers = 2; neurons = 100, 10; it has three layers, with 100, 10 and the number of components neurons each.
- Dropouts: percentage of neurons from each layer used for training.
- Learning rate for the neural network training.
- Activation function for the neuralnet.
- Losses function used to minimize the cost.
- Number of epochs for each training, being epochs the number of times the whole training set has been passed through the neuralnet.
- Batch size: number of examples used to compute the gradient of the cost function.

The future sections 4.2.4, 4.2.5 and 4.2.6 resume some more ideas about the supervised algorithm, describing the construction of the neuralnet, the fitting process and the dimensionality reduced data obtaining.

4.2.4 Neuralnet construction.

This function is in charge of constructing the neural network taking into account the inputs provided. A neuralnet is defined by the number of layers, number of neurons per layer, activation function, dropouts and loss function. To train the neuralnet we need to specify the number of epochs and batch size.

4.2.5 Fitting process.

The fitting process seeks the parameters or values obtained from a training process given a specific dataset. Depending on whether it is a supervised or non-supervised algorithm, the output of this operations will be either some parameters or directly the remodeled values. For example, as PCA is a parametric transformation, fitting will determine the dimensionality reduction matrix which could afterwards be used to determine the modified data. On the other hand, MDS and TSNE are non-parametric dimensionality reduction algorithms, so the fitting process directly obtains the values from the lower dimension. This parameters and values in the fitting process are stored as an object's own variable until necessary.

We look forward to construct a supervised learning algorithm, based on neuralnets, and we want to obtain the parameters learning from the unsupervised embedding algorithm. The steps to take would be as follows: firstly comes the construction of the neuralnet, taking into account what was said in the previous section plus the input matrix's input. Second of all, we have to store the reduced data from the embedder algorithm. At last, we have the original data and the reduced computed data, so we decided to fit the neuralnet with both datasets and store the output parameters of the neuralnet.

We have been able to turn a unsupervised learning algorithm into a supervised learning algorithm, by replicating the unsupervised algorithm's behaviour.

4.2.6 Dimensionality reduced data obtaining.

We have achieved the task of replicating an unsupervised algorithm's behaviour by a supervised algorithm, characterized for being capable of predicting the output by computing operations between the input and some learned parameters. Even though the unsupervised algorithms have no parameters, a neuralnet does. We have obtained, using those parameters, prediction outputs from the new datapoints, which copy the dimensionality reduction without the need to compute the whole algorithm.

Chapter 5

Metrics

Chapter 6

Conclusion

Chapter 7

EXTRA APPENDIX

We have developed a class named NNReplicator which replicates the behaviour of a neuralnet wrapping an embedder object, as it is represented in figure 3.1. It is written as if it was a Scikit Learn class so that the objects created could fix into a Scikit Learn pipeline, being one of the main future applications.

Once the inputs are understood, the next point is to get to know what is being computed and what methods are included in the class. As we want to include it as a Scikit Learn class, some methods are required; those are the methods fit and transform, which make the class capable of instantiate tranformers, objects that vary the data input into some output.

It is based on the Keras library, which makes easier the task of programming a neuralnet. A sequential model is used to stack one layer after the other. We use: add Dense to include a layer, add Activation to specify which activation function to use and add Dropout to include the porcentage of dropout indicated by the corresponding input.
The best way of implementing the neural net is by designing a for-loop which extracts the inputs from the arrays and automatically computes denses and dropouts.
The default optimizer is Adagrad, and takes as input the learning rate from the object's inicialization. The next step should be the compilation of the whole neural net, using the loss function inputed as the cost to minimize and the model should be completely done.
Despite of having built the neural net correctly, we also need to make it compatible with Scikit Learn. That's why the last step consists on a Keras Wrapper, named Keras Regressor, remodeling the model into a sklearn neuralnet model.

Bibliography

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. 18

[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] M. Lichman. UCI machine learning repository, 2013. 18

[7] Songrit Maneewongvatana and David M. Mount. On the efficiency of nearest neighbor searching with data clustered in lower dimensions. 22

[8] 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. 12

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

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

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

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

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

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

Code