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 decreasing the number of features needed to describe that same dataset. Thus, time and memory in huge implementations could be saved.
Taking into account that this turns into a difficult task, we can find that numerous approaches have been proposed.
Although they look forward to achieve more or less the same performance, 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 is going to reach the results we expected or needed.
The first point to take into account is the existence of parametric and non parametric algorithms, and 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 our decision of which to implement depends on the previous study of our data, the performance requirements and the later purpose and usage of the reduced data.

We propose the research and then base our study in the next dimensionality reduction algorithms:

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

1.1 PCA: Principal Component Analysis

Principal Component Analysis 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:

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

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 because of some covariance matrix calculation approaches.
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 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 (cov($\mathbf{Y}$)) sorted decreasingly would be 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 would obtain our reduced dimensionally points, as a basis transformation of our datapoints from the original dataset, by the new basis coordinates.

1.2 MDS: Multidimensional Scaling

Multidimensional Scaling in dimensionality reduction tries to create a map which displays the relative distances between the data points. This focuses on getting a one, two or, at most, three dimensional map, keeping as much distance information as possible.
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. In the one hand for metric solutions, this *proximity* matrix has to be a true distance matrix, while in the other, both dissimilarities or correlations could be the input to the problem's matrix.

MDS algorithm is based on some ideas explained in the previous section. As we can see, 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 would be the rank order of the previous matrix, which could be the input for the model.
The main distinction 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.

1.2.1 Stress metric

As in every data problem, we need a metric which shows how well is the performance given a particular dataset. 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 the dimensions lower, the stress will get higher, because we are representing the same distances relations in a lower dimensional space.

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

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.

1.3 TSNE: T-Stochastic Neighbour Embedding

T-Stochastic Neighbour Embedding is our non-linear example of dimensionality reduction. It relies on Stochastic Neighbour Embedding (SNE) which will be explained right below. The main characteristic why this algorithm is used is because it is capable of visualizating 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.

1.3.1 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:

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

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

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 could suit 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.

1.3.2 T-SNE

Although SNE is capable of showing good visualizations, T-SNE was proposed as a modified algorithm which tried to make it's 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 would stand 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 2

Framework

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

The main supervised learning algorithm applied is Neural Networks, yet it is a powerful and efficient tool which can learn from easy functions up to complicated and non-linear models. The advantage 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. However, the aim is to try and learn the values which represent the dimension reduction in a set of datapoints by using that neural net. 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 neural net which learns that embedding from **n** dimensions into two. As soon as we get a new datapoint, instead of recalculating the MDS or T-SNE algorithm, we can use the neural net parameters to obtain the dimensionally reduced value from the new example.

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

We base our programmation in nowadays technologies such as Python in Scikit Learn, Keras and Pandas.

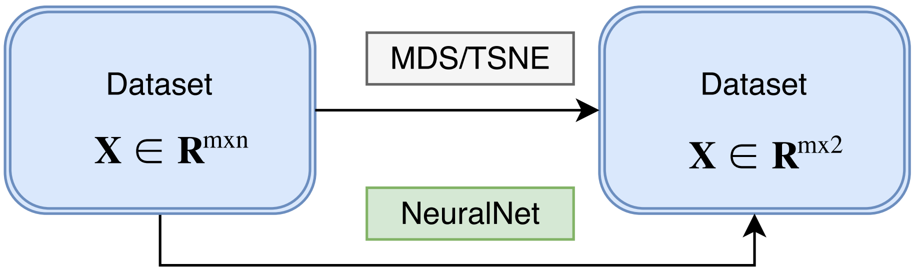
Dataset
$\mathbf{X} \in \mathbf{R}^{mxn}$
MDS/TSNE
NeuralNet
Dataset
$\mathbf{X} \in \mathbf{R}^{mx2}$

Figure 2.1: Dimensionality reduction algorithm learned by a Neural Network

2.1 Class: NNReplicator

We have developed a class named NNReplicator which replicates the behaviour of a neural net wrapping an embedder object, as it is represented in figure 2.1. 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.
The inputs expected in this new class are the next ones:

- n_comp: number of components for the low-dimensional space
- embedder: dimensionality reduction object, such as PCA, MDS or TSNE in our examples
- layers: array with the number of neurons for each layer, taking into account that the last layer has the same neurons as the number of components in the low dimensional space; e.g: [100,10]: it has three layers, with 100, 10 and n_comp neurons each.
- dropout: array with the same size as layers, containing the dropouts for each layer.
- lr: learning rate for the neural network training.
- act_function: activation function for the neural net.
- loss_func: losses function used to minimize the cost.
- epochs: number of epochs for each training
- batch_size: number of examples used to compute the cost function.

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 method fit and transform, which make the class capable of instantiate tranformers. This is explained in sections 2.1.1, 2.1.2 and 2.1.3.

2.1.1 Methods: nnConstruct

This method is in charge of constructing the neural network taking into account the inputs provided. It is based on the Keras library, which makes easier the task of programming a neural net. 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.

2.1.2 Methods: fit

2.1.3 Methods: transform

2.2 Cross Validation

2.3 Metrics

Chapter 3

Applications

Bibliography

Code