

largeVis: An Implementation of the LargeVis Algorithm

Amos Elberg

2016-05-28

This Vingette provides an overview of the largeVis package.

Introduction

The `largeVis` package offers four functions for visualizing high-dimensional datasets and finding approximate nearest neighbors, based on the `LargeVis` algorithm presented in Tang et al. (2016):

1. `randomProjectionTreeSearch`, a method for finding approximate nearest neighbors.
2. `projectKNNs`, which takes as input a weighted nearest-neighbor graph and estimates a projection into a low-dimensional space.
3. `vis`, which combines `randomProjectionTreeSearch`, `buildEdgeMatrix`, and `projectKNNs`, along with additional code to implement the `LargeVis` algorithm.
4. `manifoldMap`, which produces a plot for visualizing embeddings of images.

See the original paper for a detailed description of the algorithm.

Data Preparation

For input to `largeVis`, data should be scaled, NA's, Infs and NULL removed, and transposed from the R-standard so that examples are columns and features are rows. Duplicates should be removed as well.

If there are NA's, Infs, or NULLs in the input, `randomProjectionTreeSearch` will definitely fail.

If the numerical range covered by the data is large, this can cause errors in or before the `buildEdgeMatrix` function. This is because the algorithm requires calculating $\exp(\|\vec{x}_i - \vec{x}_j\|^2)$ in the high-dimensional space, which will overflow if the distance between any nearest neighbors exceeds about 26.

If there are duplicates in the input data, while the implementation tries to filter duplicates, it is likely to lead to problems. If the number of duplicates is large, this can cause the random projection tree search to fail. If the number is small, the algorithm may identify a sufficient number of neighbors, but an error may then occur during `buildEdgeMatrix`, or stochastic gradient descent.

Memory Consumption

The algorithm is necessarily memory-intensive for large datasets. `neighborsToVectors`, `distance`, and `buildEdgeMatrix` are available as separate functions to facilitate memory-efficient handling of large datasets, because the high-dimensional dataset is not needed after distances have been calculated. In this case, the workflow is:

```
neighbors <- randomProjectionTreeSearch(largeDataset)
neighborIndices <- neighborsToVectors(neighbors)
rm(neighbors)
distances <- distance(neighborIndices$i,
                      neighborIndices$j,
                      largeDataset)
```

```

rm(largeDataset)
wij <- buildEdgeMatrix(i = neighborIndices$i,
                        j = neighborIndices$j,
                        d = distances)
rm(distances, neighborIndices)
coords <- projectKNNs(wij$wij)

```

In testing, this method reduced peak RAM requirements by more than 70%.

Overview of Functions and Hyperparameters

`randomProjectionTreeSearch`

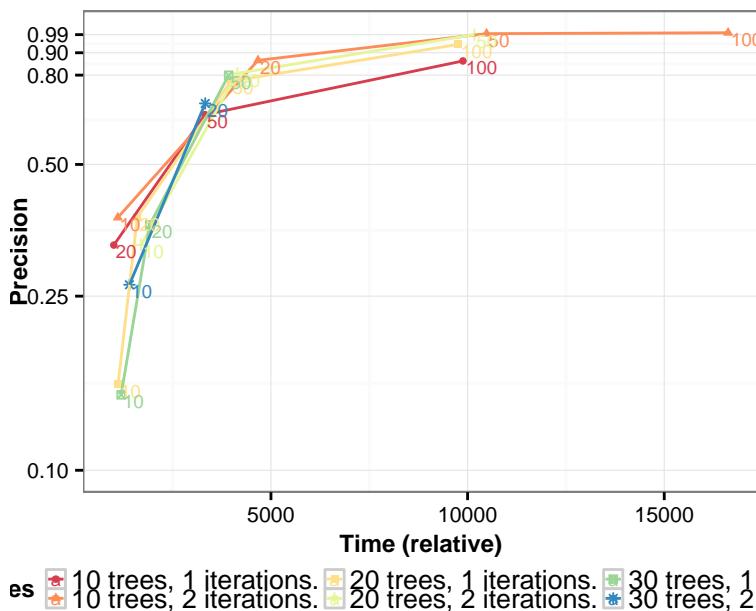
This function uses a two-phase algorithm to find approximate nearest neighbors. In the first phase, the algorithm creates `n_trees` binary trees dividing the space into leaves of at most `tree_threshold` nodes. A node's candidate nearest neighbors are the union of all nodes with which it shared a leaf on any of the trees. In the second phase, for each node, the algorithm looks at the candidate nearest neighbors for that node, as well as each of those nodes' candidate nearest neighbors. The logic of the algorithm is that a node's neighbors' neighbors are likely to be the node's own neighbors. In each iteration, the closest K candidate neighbors for each node are kept.

The authors of Tang et al. (2016) suggest that a single iteration of the second phase is generally sufficient to obtain satisfactory performance.

The chart below illustrates the trade-off between performance and accuracy for the nearest-neighbor search, using various hyperparameters. The data was produced using the `benchmark.R` script in the `inst/` directory. The test data is the 1-million vector, 128-feature SIFT Dataset, as per Erik Bernhardsson's ANN Benchmark github.

e vs. Precision 1–Million Vector SIFT Dataset, (K =

Labelled by Tree Threshold



es 10 trees, 1 iterations. 20 trees, 1 iterations. 30 trees, 1 iterations.
 10 trees, 2 iterations. 20 trees, 2 iterations. 30 trees, 2 iterations.

If `randomProjectionTreeSearch` fails to find the desired number of neighbors, usually the best result is obtained by increasing the tree threshold. If `randomProjectionTreeSearch` fails with an error that no

neighbors were found for some nodes, and the tree threshold is already reasonable, this may be an indication that duplicates remain in the input data.

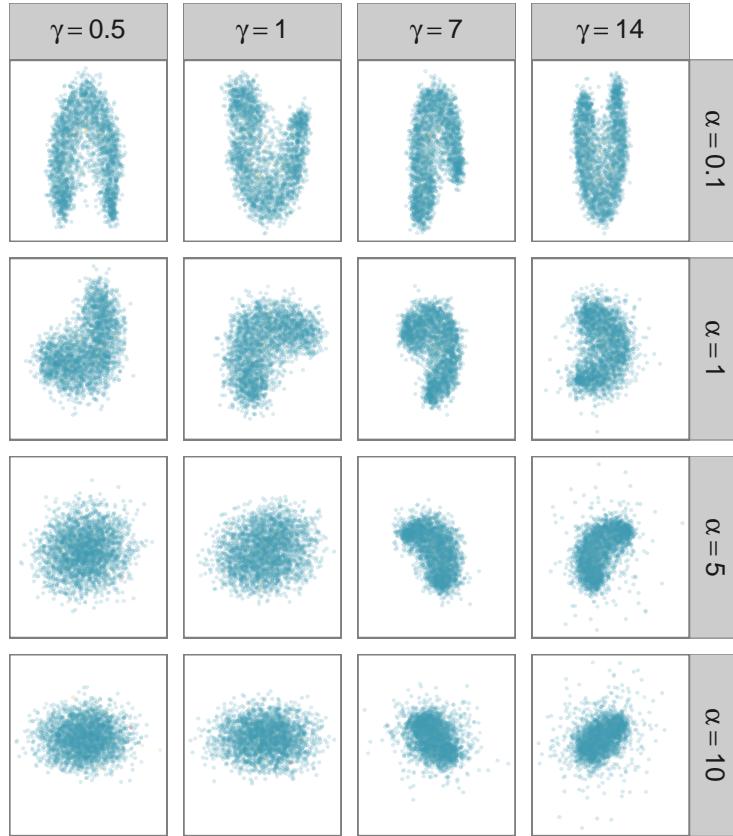
projectKNNs

This function takes as its input a `Matrix::sparseMatrix`, of connections between nodes. The matrix must be symmetric. A non-zero cell implies that node i is a nearest neighbor of node j , vice-versa, or both. Non-zero values represent the strength of the connection relative to other nearest neighbors of the two nodes.

The `LargeVis` algorithm, explained in detail in Tang et al. (2016), estimates the embedding by sampling from the identified nearest-neighbor connections. For each edge, the algorithm also samples M non-nearest neighbor negative samples. M , along with γ and α , control the visualization. α controls the desired distance between nearest neighbors. γ controls the relative strength of the attractive force between nearest neighbors and repulsive force between non-neighbors.

The following grid illustrates the effect of the α and γ hyperparameters, using the `wiki` dataset which is included with the package:

Effect of α vs. γ



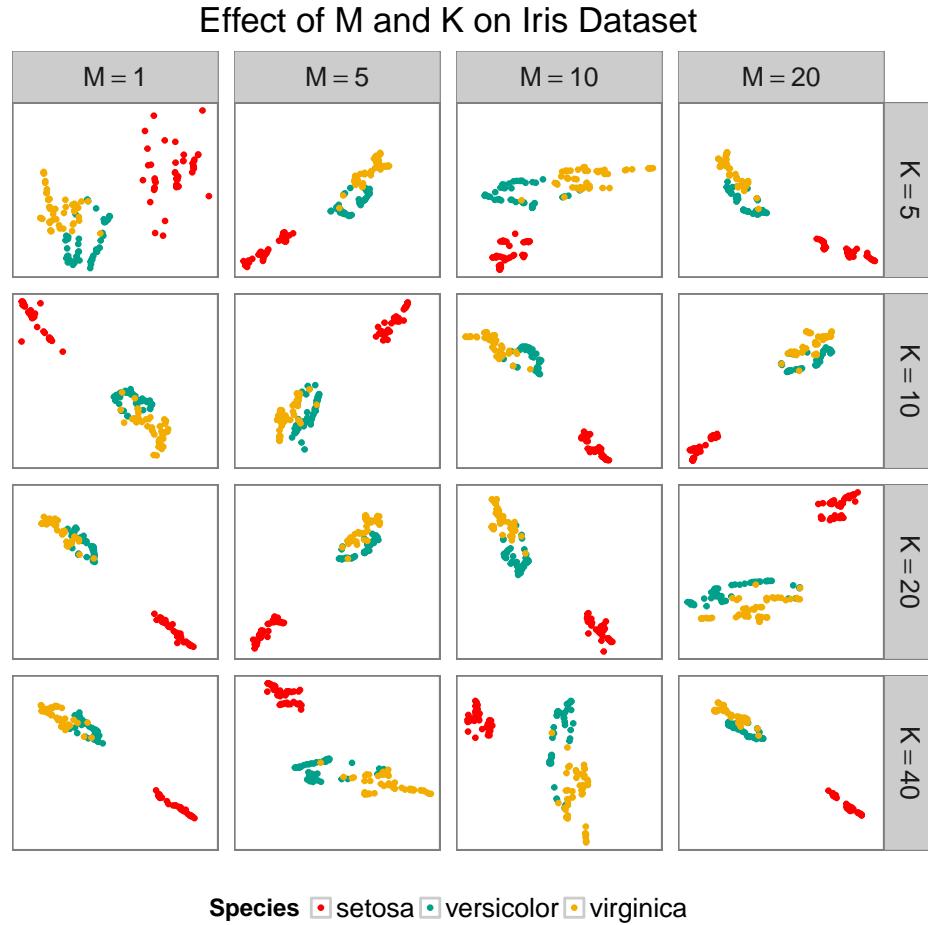
The additional hyperparameters ρ and $\min-\rho$ control the starting and final learning rate for the stochastic gradient descent process.

The algorithm can treat positive edge weights in two different ways. The authors of Tang et al. (2016) suggest that edge weights should be used to generate a weighted sampling. However, the algorithm for taking a weighted sample runs in $O(n \log n)$. Alternatively, the edge-weights can be applied to the gradients. This is controlled by the `weight_pos_samples` parameter.

vis

The `vis` function combines `randomProjectionTreeSearch` and `projectKNNs`, along with additional logic for calculating edge weights, to implement the complete LargeVis algorithm.

The following chart illustrates the effect of the `M` and `K` parameters, using the `iris` dataset.

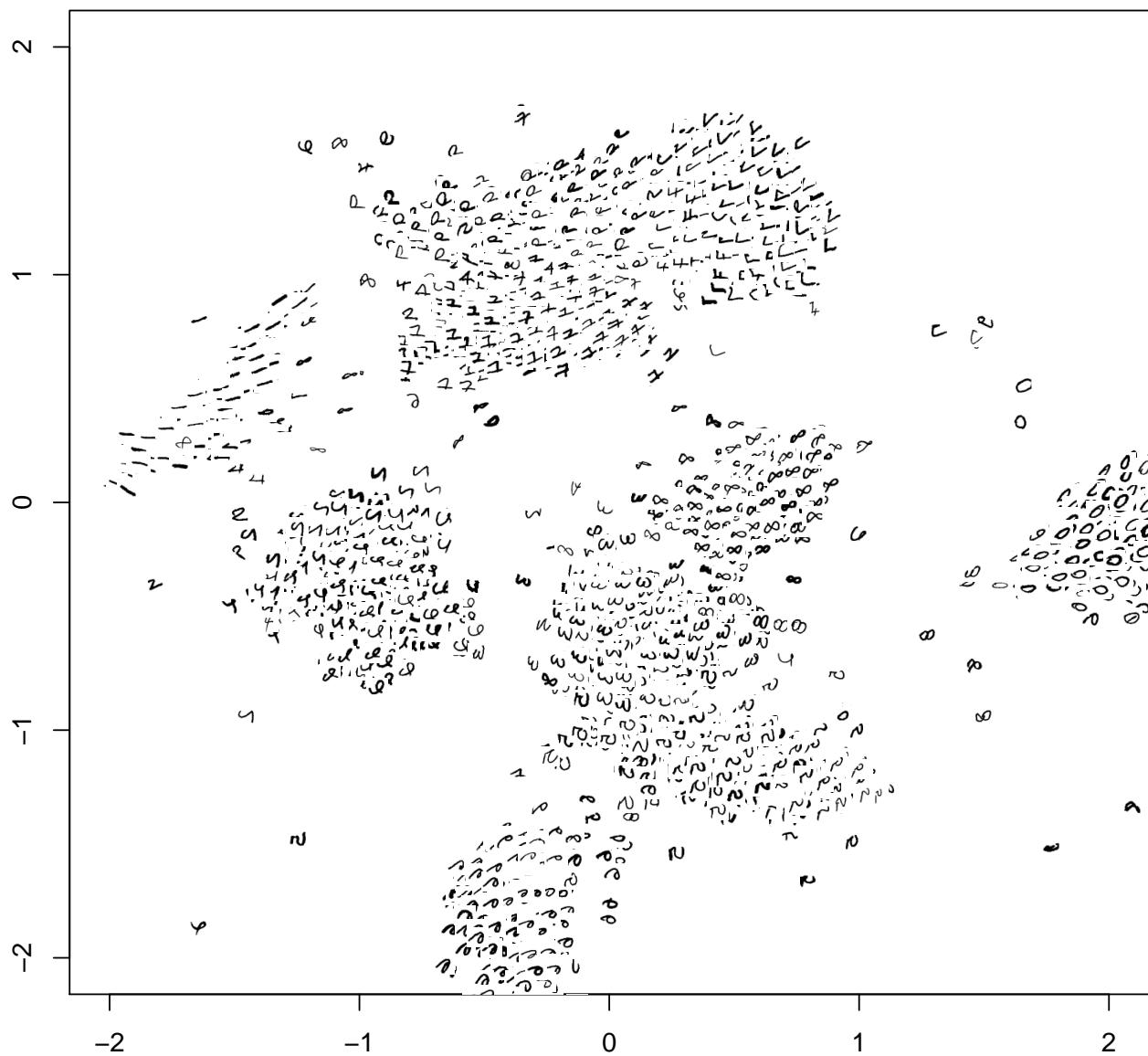


manifoldMap

The `manifoldMap` function is useful when the examples being clustered are themselves images. Given a coordinate matrix (as generated by `projectKNNs` or `vis`) and an array of `N` images, the function samples `n` images and plots them at the coordinates given in the matrix. If the `transparency` parameter is a number between 0 and 1, then the function adds to each image an alpha channel where the value per pixel is proportional to `transparency*` the image content.

The function can plot both color and greyscale images.

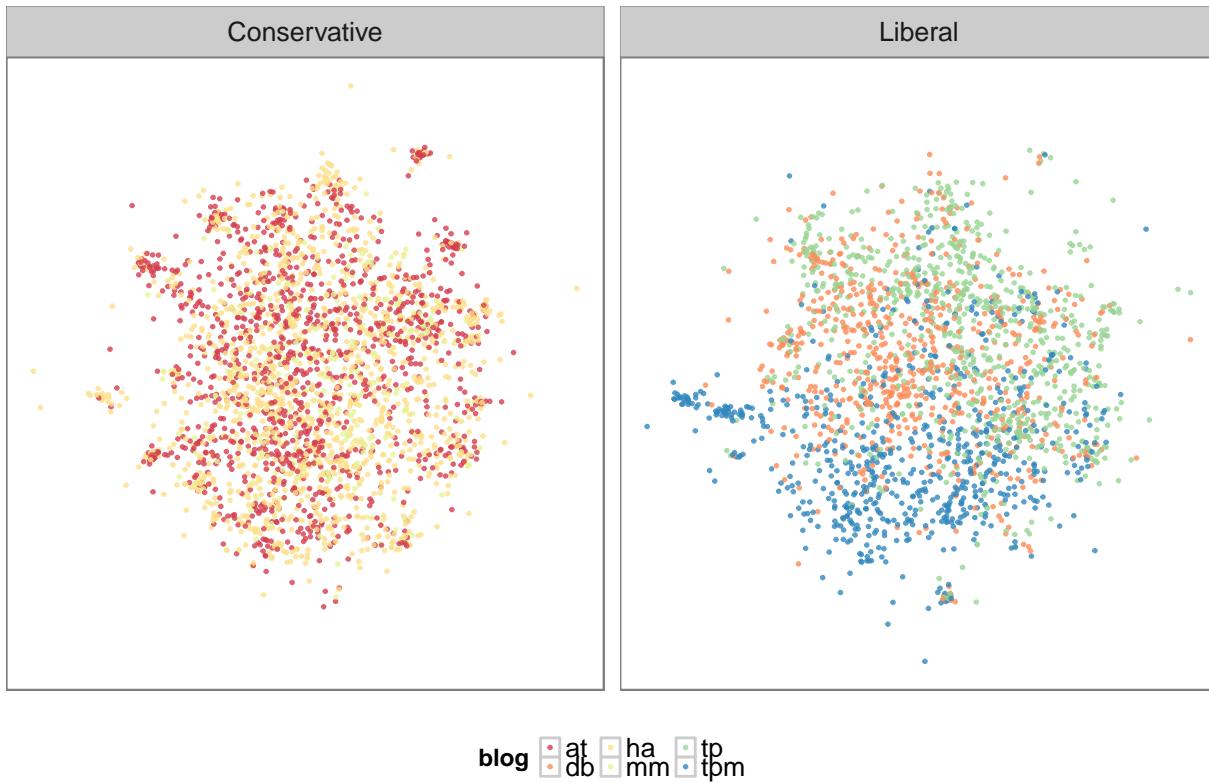
The following plot illustrates this by plotting 5000 images sampled from the MNIST dataset at positions generated by `vis`:



Support for Sparse Matrices

largeVis supports sparse matrices. Besides facilitating very large datasets, this makes it practicable to visualize term-document-matrices. The workflow can be something like this:

Visualization of a tf–idf Matrix



Distance Methods

The original `LargeVis` paper used Euclidean distances exclusively. The `largeVis` package offers a choice among Euclidean and Cosine distance measures.

Bibliography

Tang, Jian, Jingzhou Liu, Ming Zhang, and Qiaozhu Mei. 2016. “Visualization Large-Scale and High-Dimensional Data.” *CoRR* abs/1602.00370. <http://arxiv.org/abs/1602.00370>.