source{d} | blog

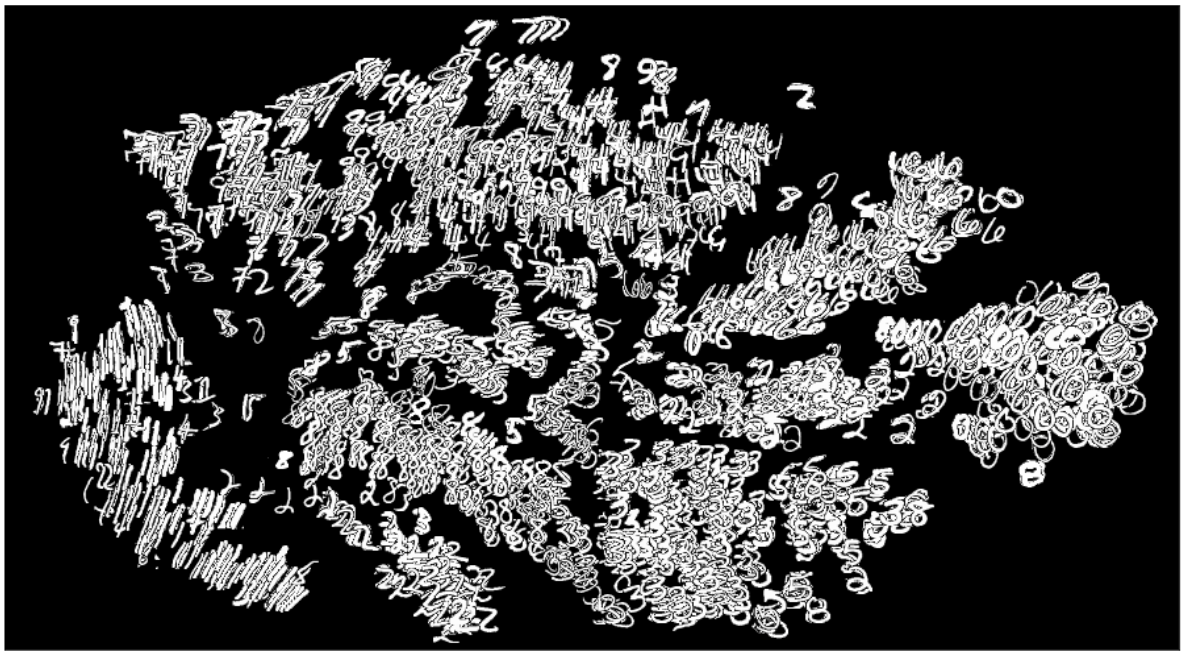
Building the first **AI** that **understands code**

Take me to **sourced.tech**

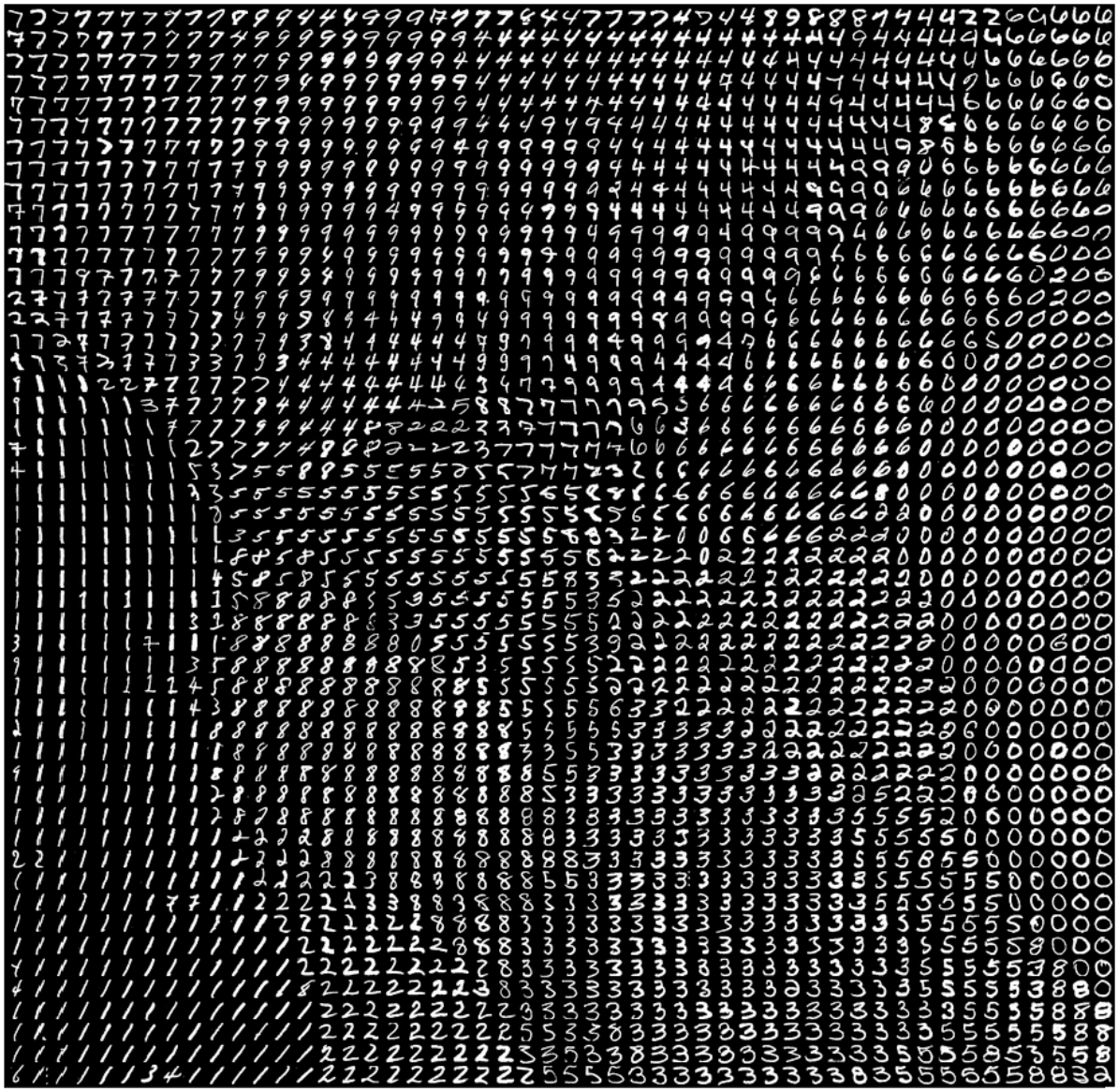
Jonker-Volgenant Algorithm + t-SNE = Super Powers

by **Vadim Markovtsev** 14 March 2017

BEFORE



AFTER



Intrigued? Then… first things first!

t-SNE

t-SNE is the very popular algorithm to extremely reduce the dimensionality of your data in order to visually present it. It is capable of mapping hundreds of dimensions to just 2 while preserving important data relationships, that is, when closer samples in the original space are closer in the reduced space. t-SNE works quite well for small and moderately sized real-world datasets and does not require much tuning of its hyperparameters. In other words, if you've got less than 100,000 points, you will apply that magic black box thing and get a beautiful scatter plot in return.

Here is a classic example from computer vision. There is a well known dataset named **"MNIST"** by Yann LeCun (one of the inventors of **Backpropagation** method of

training neural networks - the core of modern deep learning) et. al. It is often used as the default dataset for evaluating machine learning ideas and is widely employed in academia. MNIST is 70,000 greyscale images of size 28x28. Each is the scan of a handwritten digit $\in [0, 9]$. There is a way to obtain an **"infinite"** MNIST dataset but I shouldn't diverge.

Thus each MNIST sample contains $28 \cdot 28 = 784$ features and can be represented by a 784-dimensional vector. Vectors are linear and we lose the locality relationships between individual pixels in this interpretation but it is still helpful. If you try to imagine how our dataset looks like in 784D, you will go nuts unless you are a trained mathematician. Ordinary humans can consume visual information only in 3D, 2D or 1D. We may implicitly add another dimension, time, but usually nobody says that a computer display is 3D just because it changes the picture with 100Hz frequency. Thus it would be nice to have a way to *map* samples in 784 dimensions to 2. Sounds impossible? It is, in the general case. This is where **Dirichlet's principle** works: you are doomed to have collisions, whatever mapping algorithm you choose.

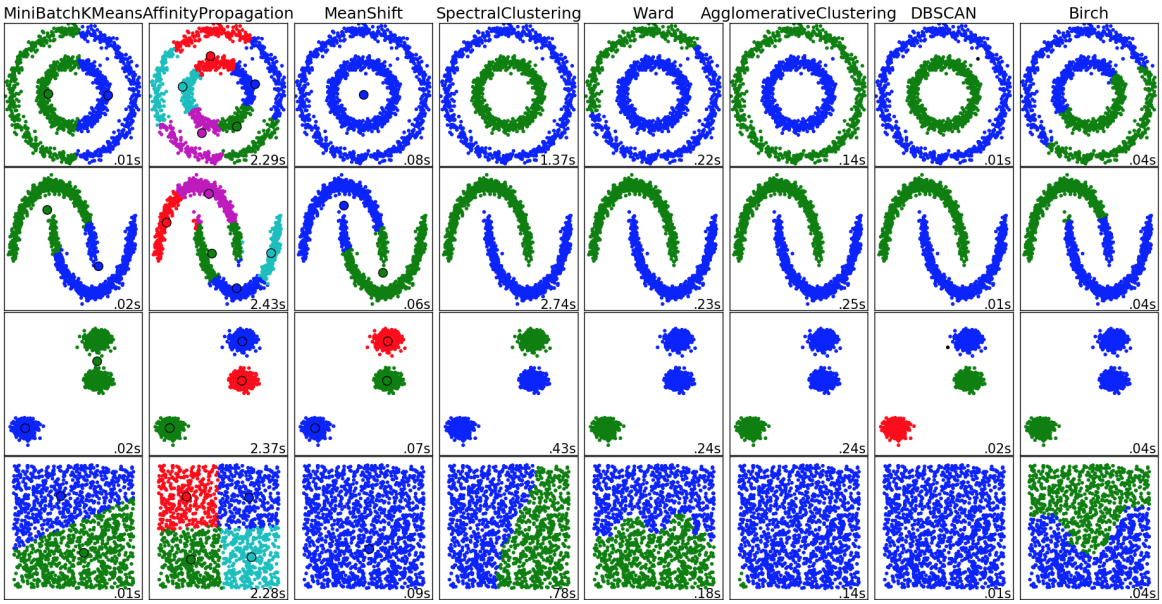
*3D -> 2D projection illusion in **Shadowmatic***

Luckily, the following two assumptions stand:

1. The original high-dimensional space is *sparse*, that is, samples are most likely not distributed uniformly in it.

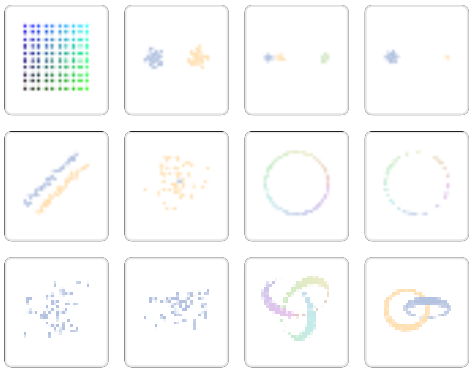
2. We do not have to find the exact mapping, especially given the fact that it does not exist. We can rather solve a different problem which has a guaranteed precise solution which approximates what we would like to see. This resembles how JPEG compression works: we never get the pixel-to-pixel identical result, but the image looks *very* similar to its origin.

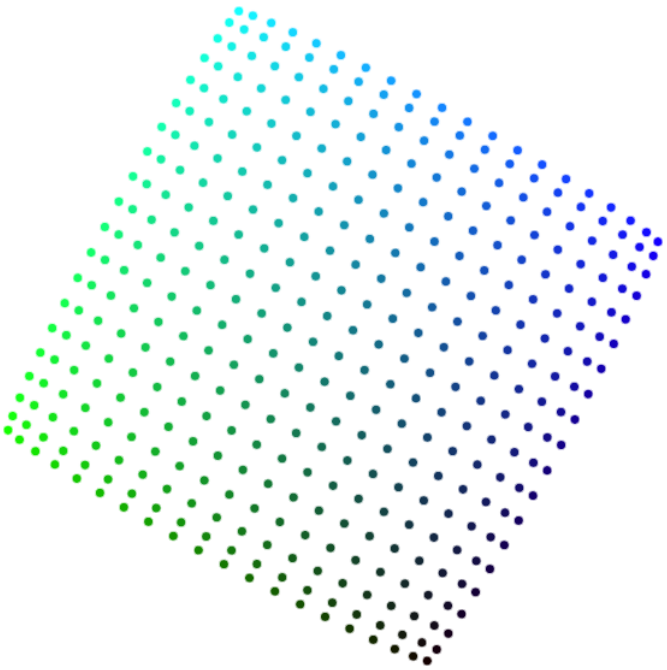
The question is, what is the best approximate problem in (2). Unfortunately, there is no “best”. The quality of dimensionality reduction is subjective and depends on your ultimate goal. The root of the confusion is the same as in determining the perfect clustering: it depends.



*Different clustering algorithms from **sklearn***

t-SNE is one of a series of possible dimensionality reduction algorithms which are called embedding algorithms. The core idea is to preserve the similarity relations as much as possible. Play with it yourself:





A square grid with equal spacing between points. Try convergence at different sizes.

Step 1,010

Points Per Side 20

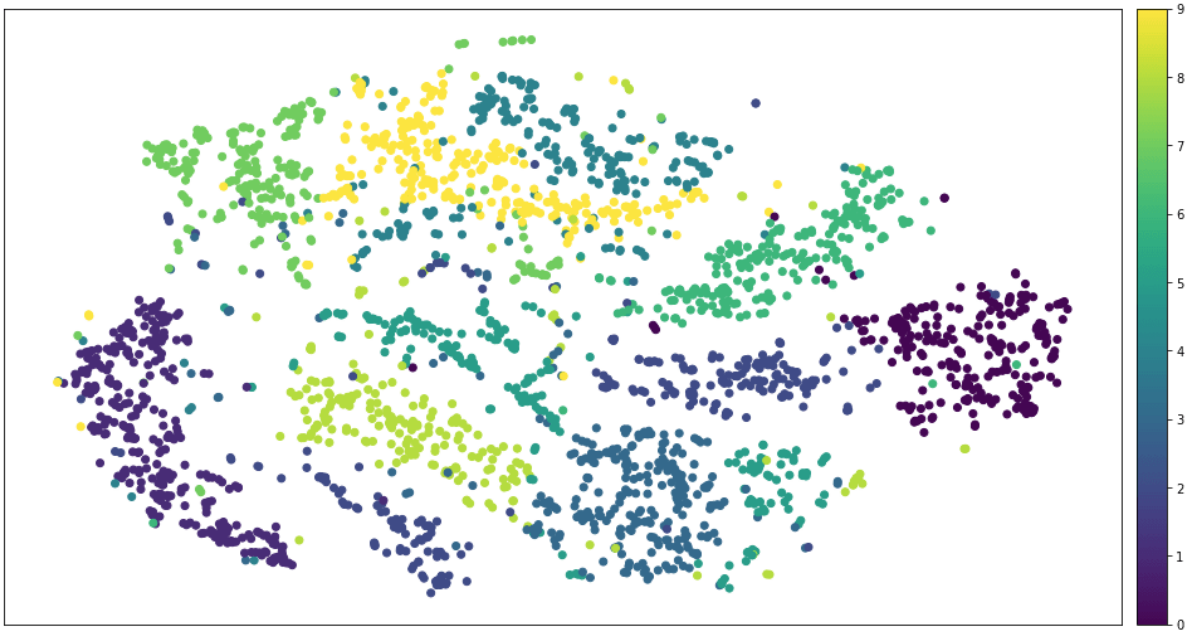
Perplexity 10

Epsilon 5

MARTIN WATTENBERG Google Brain
FERNANDA VIÉGAS Google Brain
IAN JOHNSON Google Cloud
Oct. 13 2016
Citation: Wattenberg, et al., 2016

*Adapted from **How to Use t-SNE Effectively***

Those are artificial examples - cool but not enough. The majority of real-world datasets resemble a cloud with local clusters. For example, MNIST looks like this:



MNIST after applying t-SNE

We can clearly see how similar digits tend to attract each other.

Linear Programming

Now let us make a steep turn and review what is **Linear Programming** (LP). Sorry but it's not a new design pattern, a Javascript framework or a startup. It is math:

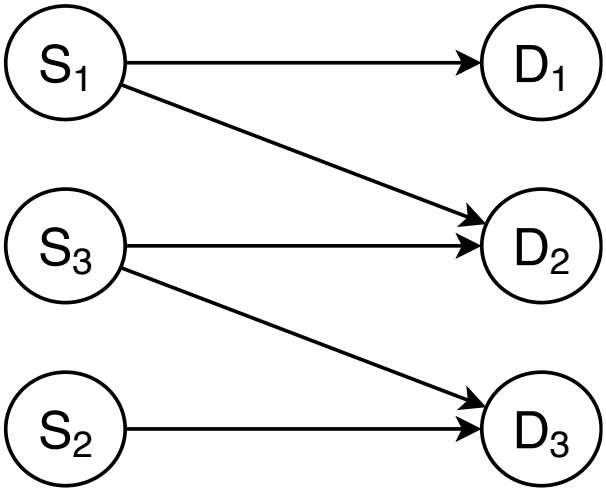
$$\arg\min \vec{c} \cdot \vec{x} \tag{1}$$

$$A \cdot \vec{x} \leq \vec{b} \tag{2}$$

$$\vec{x} \geq 0 \tag{3}$$

We minimize the scalar product of $\vec{c}$ and $\vec{x}$ given the set of linear inequations depending of $\vec{x}$ and the requirement that all its coordinates are not negative. LP is a well-studied topic in convex optimization theory, it is known to have **weakly-polynomial** solutions which typically run in $O(n^3)$ time where n is the number of variables (problem's dimensionality). There often are approximate algorithms which run in linear time. Those algorithms deal with matrix multiplications and can be parallelized efficiently. A programmer's heaven!

Amazingly many problems can be tracked down to LP. For example, let's take the **transportation problem**.



Transportation Problem: supplies and demands.

There is a number of different supplies and demands, which may be not equal. Every demand needs a fixed amount of supplies. Every supply is limited and is connected with some of the demands. The core of the problem is that every edge $S_i D_j$ has it's

own "cost" c_{ij} so we need to find the supply scheme which minimizes the sum of those costs. Formally,

$$\arg\min \sum_{i=1}^{S} \sum_{j=1}^{D} x_{ij} c_{ij} \tag{4}$$

$$x_{ij} \geq 0, \tag{5}$$

$$\sum_{j=1}^{D} x_{ij} \leq w_{S_i}, \tag{6}$$

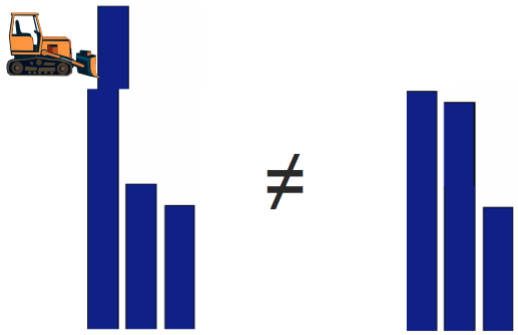
$$\sum_{i=1}^{S} x_{ij} \leq w_{D_j}, \tag{7}$$

$$\sum_{i=1}^{S} \sum_{j=1}^{D} x_{ij} = \min \left(\sum_{i=1}^{S} w_{S_i}, \sum_{j=1}^{D} w_{D_j} \right). \tag{8}$$

The last condition means that either we run out of supplies or there is no more demand. If $\sum_{i=1}^{S} w_{S_i} = \sum_{j=1}^{D} w_{D_j}$, 8 can be normalized and simplified as

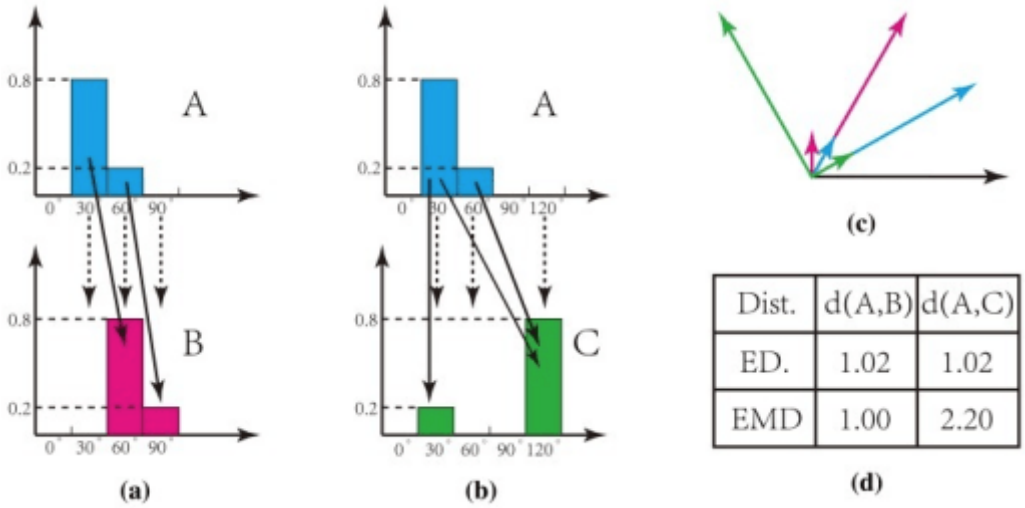
$$\sum_{i=1}^{S} \sum_{j=1}^{D} x_{ij} = \sum_{i=1}^{S} w_{S_i} = \sum_{j=1}^{D} w_{D_j} = 1.$$

Now if we replace "supplies" and "demands" with "dirt", $\min \sum_{i=1}^{S} \sum_{j=1}^{D} x_{ij} c_{ij}$ gives us **Earth Mover's Distance**: the minimal volume of work required to carry dirt from one pile distribution to another. Next time you dig holes in the ground, you know what to do…



Earth Mover's Distance

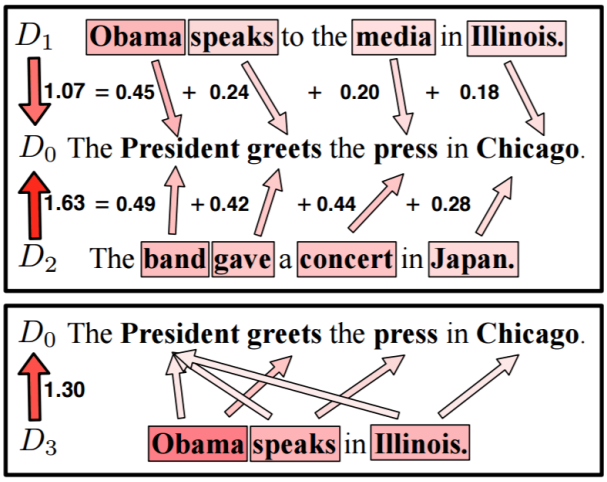
If we replace “supplies” and “demands” with “histograms”, we get the most popular way to compare images in pre-deep learning era (**example paper**). It is better than naive L2 because it captures the spatial difference additionally to the magnitudal one.



Dist.	d(A,B)	d(A,C)
ED.	1.02	1.02
EMD	1.00	2.20

Earth Mover’s Distance is better than Euclidean distance for histogram comparison.

If we replace “supplies” and “demands” with “words”, we get **Word Mover’s Distance**, a good way of comparing meanings of two sentences given word embeddings from **word2vec**.



Word Mover’s Distance.

If we relax the conditions 5-8 by throwing away 8, set $w_{S_i} = w_{D_i} = 1$ and turn inequalities 6 and 7 into equations by adding the symmetric negated inequalities, we get the **Linear Assignment Problem** (LAP):

$$\arg\min \sum_{i=1}^{S} \sum_{j=1}^{D} x_{ij} c_{ij} \tag{9}$$

$$x_{ij} \geq 0, \tag{10}$$

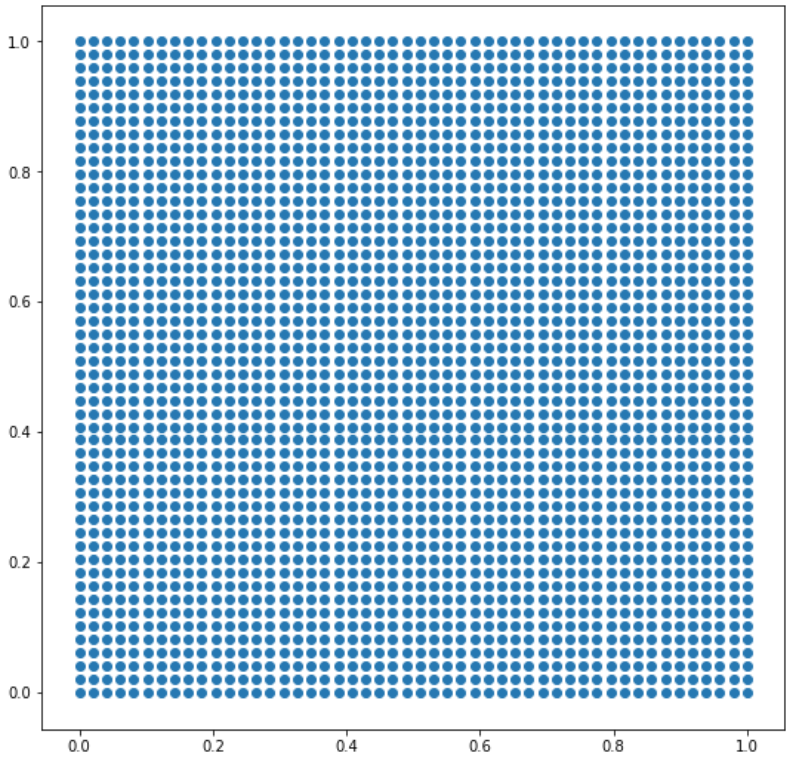
$$\sum_{j=1}^{D} x_{ij} = 1, \tag{11}$$

$$\sum_{i=1}^{S} x_{ij} = 1. \tag{12}$$

Unlike in Transportation Problem, it can be proved that $x_{ij} \in \{0, 1\}$ - the solution is always binary. In other words, either the whole supply goes to some demand, or nothing.

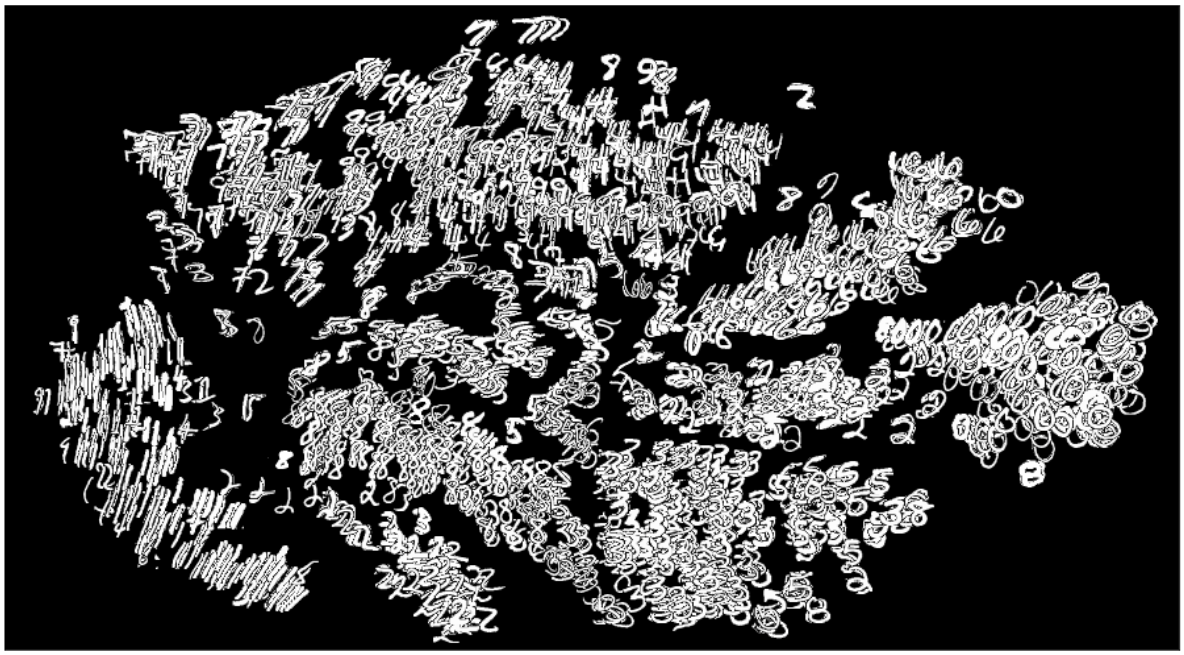
t-SNE LAP

As we saw in the first section, t-SNE (or any other embedding) produces a scatter plot. While it is perfectly suitable for dataset exploration tasks, sometimes we need to map every sample in the original scatter plot to a node in the regular grid. E.g. source{d} needs this mapping to… you will see why soon.



The Regular Grid.

We can draw MNIST digits instead of dots after t-SNE, this is how it looks like:



MNIST digits after t-SNE.

Not very clear. This where LAP arises: we could define the cost matrix as the pairwise euclidean distances between t-SNE samples and grid nodes, set the grid square equal to the dataset size $||S|| = ||D||$ and eventually solve our problem. But how? No algorithms were presented so far.

JONKER-VOLGENANT ALGORITHM

It turns out that there are tons of general-purpose linear optimization algorithms, starting from the **simplex method** and ending with very sophisticated solvers. Algorithms which are specialized for the specific conditions usually converge remarkably faster, though they may have some limitations.

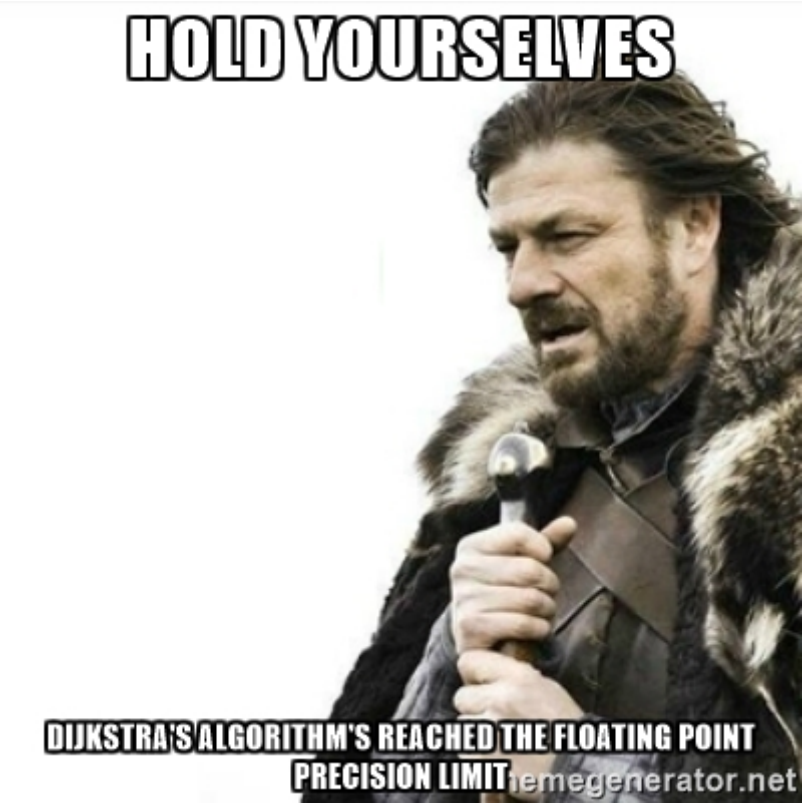
Hungarian algorithm is one of those specialized solvers invented in 1950-s. It's complexity is $O(n^3)$. It is rather simple to understand and to implement, thus the popular choice in a lot of projects. For example, it has recently become the part of **scipy**. Unfortunately, it performs slow on bigger problem sizes; scipy's variant is particularly **very** slow. I waited an hour for it to finish on 2500 MNIST samples and yet Python was still digesting the victim.

Jonker-Volgenant algorithm is an improved approach developed in 1987. It's core is still the shortest augmenting path traversal and it's complexity is still cubic, but it uses some smart heuristics and tricks to dramatically reduce the computational

load. The performance of many other LAP algorithms including JV was extensively studied in **2000's Discrete Applied Mathematics paper**. The conclusion was that:

> *JV has a good and stable average performance for all the (problem − Vadim) classes, and it is the best algorithm for the uniform random ... and for the single−depot class.*

There is a caveat with the JV algorithm though. It is loosely tolerant to the difference between any pair of cost elements in the cost matrix. For example, if there are two very close costs appearing in the same graph where we search for the shortest path using Dijkstra's algorithm, it can potentially loop forever. If you take a closer look at Dijkstra's algorithm, you will eventually discover that when it reaches the floating point precision limit, terrible things may happen. The common workaround is to multiply the cost matrix by some big number.



Anyway, the most exciting thing about JV for a lazy engineer like me is that there is an existing Python 2 package which wraps the **ancient** JV C implementation: **pyLAPJV**. That C code was written by **Roy Jonker** in 1996 for MagicLogic Optimization Inc. - he is the company's president. If you read this, Roy, please share your paper under CC-BY-something, everybody wants to read it! Besides from being

abandonware, pyLAPJV has a minor problem with the output which I resolved in **PR #2**. The C code is reliable, but it does not leverage any threads or SIMD instructions. Of course, we saw that JV is sequential in it's nature and cannot be easily parallelized, however, I managed to speed it up 2x after optimizing the hottest block - augmenting row reduction - with **AVX2**. The result is the new Python 3 package **src-d/lapjv** which we open sourced under MIT license.

Augmenting row reduction phase at its core is finding the minimum and the second minimum array elements. Sounds easy as it is, the unoptimized C version takes about 20 lines of code. AVX version is 4 times bigger: we record minimums in each lane of the SIMD vector, perform **blending** and cast other dark SIMD magic spells I learned while I was writing Samsung's **libSoundFeatureExtraction**.

```
template <typename idx, typename cost>
__attribute__((always_inline)) inline
std::tuple<cost, cost, idx, idx> find_umins(
    idx dim, idx i, const cost *assigncost, const cost *v) {
  cost umin = assigncost[i * dim] - v[0];
  idx j1 = 0;
  idx j2 = -1;
  cost usubmin = std::numeric_limits<cost>::max();
  for (idx j = 1; j < dim; j++) {
    cost h = assigncost[i * dim + j] - v[j];
    if (h < usubmin) {
      if (h >= umin) {
        usubmin = h;
        j2 = j;
      } else {
        usubmin = umin;
        umin = h;
        j2 = j1;
        j1 = j;
      }
    }
  }
  return std::make_tuple(umin, usubmin, j1, j2);
}
```

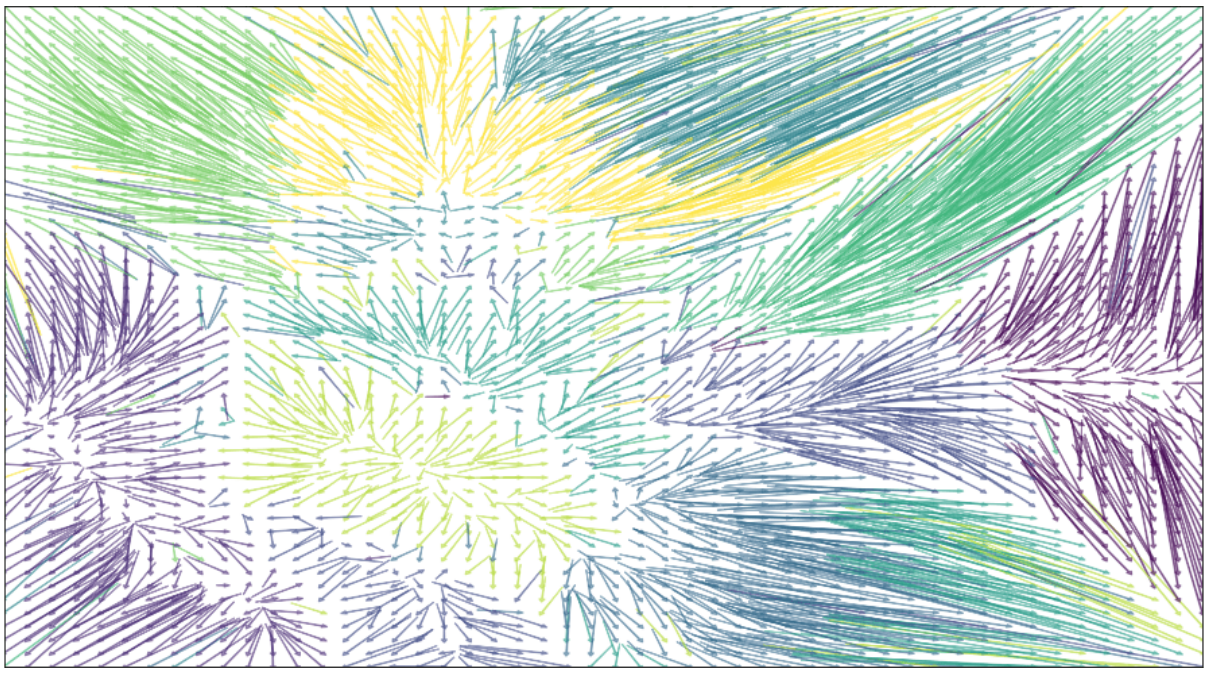
Finding two consecutive minimums, plain C++.

```
template <typename idx>
__attribute__((always_inline)) inline
std::tuple<float, float, idx, idx> find_umins(
    idx dim, idx i, const float *assigncost, const float *v) {
  __m256i idxvec = _mm256_setr_epi32(0, 1, 2, 3, 4, 5, 6, 7);
  __m256i j1vec = _mm256_set1_epi32(-1), j2vec = _mm256_set1_epi32(-1
  __m256 uminvec = _mm256_set1_ps(std::numeric_limits<float>::max()),
         usubminvec = _mm256_set1_ps(std::numeric_limits<float>::max(
```

```
for (idx j = 0; j < dim - 7; j += 8) {
  __m256 acvec = _mm256_loadu_ps(assigncost + i * dim + j);
  __m256 vvec = _mm256_loadu_ps(v + j);
  __m256 h = _mm256_sub_ps(acvec, vvec);
  __m256 cmp = _mm256_cmp_ps(h, uminvec, _CMP_LE_OQ);
  usubminvec = _mm256_blendv_ps(usubminvec, uminvec, cmp);
  j2vec = _mm256_blendv_epi8(
      j2vec, j1vec, reinterpret_cast<__m256i>(cmp));
  uminvec = _mm256_blendv_ps(uminvec, h, cmp);
  j1vec = _mm256_blendv_epi8(
      j1vec, idxvec, reinterpret_cast<__m256i>(cmp));
  cmp = _mm256_andnot_ps(cmp, _mm256_cmp_ps(h, usubminvec, _CMP_LT_
  usubminvec = _mm256_blendv_ps(usubminvec, h, cmp);
  j2vec = _mm256_blendv_epi8(
      j2vec, idxvec, reinterpret_cast<__m256i>(cmp));
  idxvec = _mm256_add_epi32(idxvec, _mm256_set1_epi32(8));
}
float uminmem[8], usubminmem[8];
int32_t j1mem[8], j2mem[8];
 mm256_storeu_ps(uminmem, uminvec);
```

Finding two consecutive minimums, optimized code with AVX2 intrinsics.

lapjv maps 2500 MNIST samples in 5 seconds on my laptop and finally we see the precious result:



Linear Assignment Problem solution for MNIST after t-SNE.

NOTEBOOK

I used the following **Jupyter** notebook (**link**) to prepare this post:

In [1]:
```
%pylab inline
import pickle
from sklearn.datasets import fetch_mldata
from sklearn.decomposition import PCA
from sklearn.manifold import TSNE
from sklearn.utils import shuffle
```

```
Populating the interactive namespace from numpy and m
atplotlib
```

Load the MNIST dataset.

In [2]:
```
mnist = fetch_mldata('MNIST original', data_home=".")
```

In [3]:
```
mnist["data"].shape
```

Out[3]:
```
(70000, 784)
```

Take random 2500 images - we will project them to 50 x 50 grid.

In [4]:
```
size = 50
N = size * size
data, target = shuffle(mnist["data"],
mnist["target"], random_state=777, n_samples=N)
```

Conclusion

There is an efficient way to map t-SNE-embedded samples to the regular grid. It is based on solving the Linear Assignment problem using Jonker-Volgenant algorithm implemented in **src-d/lapjv**. This algorithm scales up to 10,000 samples.

Vadim Markovtsev

A former system programmer, Vadim is now a machine learning engineer in love with deep neural networks.

BACK TO TOP

SHARE