Probabilistic ANN: The Swiss Army Knife of GenAI

Vincent Granville, Ph.D.
vincentg@MLTechniques.com
www.MLTechniques.com

Version 1.0, February 2024

1 Motivation and architecture

With the increasing popularity of RAG, LLM, and ANN-based fast vector search including in real time, it was time for me to figure out how I could improve the existing technology. In this context, ANN stands for approximated nearest neighbors, a better alternative to K-NN.

What I came up with covers a large number of applications: matching embeddings to prompts or user queries, data synthetization, GenAI model evaluation, measuring the similarity or distance between two datasets, detection of extremes in arbitrary dimensions, finding the envelop of a dataset, or classification (supervised and unsupervised). The technique was first introduced in the context of NoGAN tabular data synthetization, see here. The goal is to find a good approximation to the solution, as quickly as possible. You can think of it as a gradient descent method, with very steep descent at the beginning, leading to a satisfactory solution in a fraction of the time most sophisticated algorithms require. Speed is further increased by using absolute differences rather than square roots of sums of squares. Also, there is no gradient and no neural networks: thus, no math beyond random numbers.

The following architecture and algorithm are common to all applications. You have two sets of multivariate vectors, S and T respectively with n and m elements, each elements being a vector with d components:

$$S = \{x_1, \dots, x_n\}$$
$$T = \{y_1, \dots, y_m\}.$$

For each element x_k in S, you want to find the closest neighbor $y_{\sigma(k)}$ in T. Thus, the problem consists of finding the function σ_0 that minimizes the loss function $L(\sigma)$ defined by

$$L(\sigma) = \sum_{k=1}^{n} \|x_k - y_{\sigma(k)}\|.$$

The minimum is over all integer functions σ defined on $\{1, \dots, n\}$ with values in $\{1, \dots, m\}$. There are m^n such functions. The one minimizing $L(\sigma)$ is denoted as σ_0. It might not be unique, but this is unimportant. In some cases, we are interested in maximizing $L(\sigma)$ instead, which is identical to minimizing $-L(\sigma)$. And frequently, to be admissible as a solution, a function σ must satisfy $x_k \neq y_{\sigma(k)}$ for $1 \leq k \leq n$.

The oldest application in recent times, also the origin for the abbreviation ANN, is the K-NN problem, or K nearest neighbors. In this case, S consists of K copies of T. As we shall see, my algorithm results in a different solution, with a variable number of neighbors per observation, rather than the fixed value K. Also, when $K = 1$, the trivial solution is $\sigma(k) = k$ for $1 \leq k \leq n$. That is, the closest neighbor to x_k is x_k itself. Thus the aforementioned constraint $x_k \neq y_{\sigma(k)}$ to eliminate this solution.

An ancient version dating back to 1890 is the assignment problem. It was solved in polynomial time in 1957, using the Hungarian algorithm [Wiki]. These days, we want something much faster than even quadratic time. My method will provide a good approximation much faster than quadratic if you stop early. Brute force would solve this problem in $n \times m$ steps, by finding the closest $y_{\sigma(k)}$ to each x_k separately. Note that unlike in the original assignment problem, here the function σ does not need to be a permutation, allowing for faster, one-to-many neighbor allocation.

The solution can be an excellent starting point for an exact search, or used as a final, good enough result. The algorithm processes the data set S a number of times. Each completed visit of S is called an epoch. In a given epoch, for each observation x_k (with $1 \leq k \leq n$), a potential new neighbor $y_{\sigma'(k)}$ is randomly selected. If

$$\|x_k - y_{\sigma'(k)}\| < (1 - \tau) \cdot \|x_k - y_{\sigma(k)}\|,$$

then $y_{\sigma'(k)}$ becomes the new, closest neighbor to x_k, replacing the old neighbor $y_{\sigma(k)}$. In this case, $\sigma(k) \leftarrow \sigma'(k)$. Otherwise, $\sigma(k)$ is unchanged, but $y_{\sigma'(k)}$ is flagged as unsuitable neighbor in the list of potential neighbors to x_k. For each x_k, the list of unsuitable neighbors starts empty and grows very slowly, at least at the beginning. The parameter τ is called the temperature. The default value is zero, but positive values that decay over time may lead to an accelerated schedule. Negative values always underperform, but it makes the loss function goes up and down, with oscillations of decreasing amplitude over time, behaving very much like the loss function in stochastic gradient descent and deep neural networks.

Another mechanism to accelerate the convergence at the beginning (what we are interested in) is as follows. At the start of each epoch, sort S in reverse order based on distance to nearest neighbors in T, obtained so far. In a given epoch, do not process all observations x_k, but only a fraction of them, for instance the top 50% with the largest NN distances.

Figure 1 illustrates the convergence. The power function $\varphi(t) = \alpha + \beta t^{-\gamma}$ provides an excellent fit. Here $\varphi(t)$ is the average nearest neighbor distance at time t. The time represents the number of steps performed so far, on a dataset with $n = m = 200$. Interestingly, $\gamma \approx 0.50$, but on some datasets, I was able to get faster convergence, with $\gamma \approx 0.80$. The coefficient α represents the average NN distance at the limit, if you were to do an exact search. In other words, $\alpha \approx L(\sigma_0)/n$. If you are only interested in α, you can get a good approximation in a fraction of the time it takes to compute the exact NN distances. To do it even faster by interpolating the curve fitting function based on the first few hundred measurements only, see Figure 4.5 and explanations in this article.

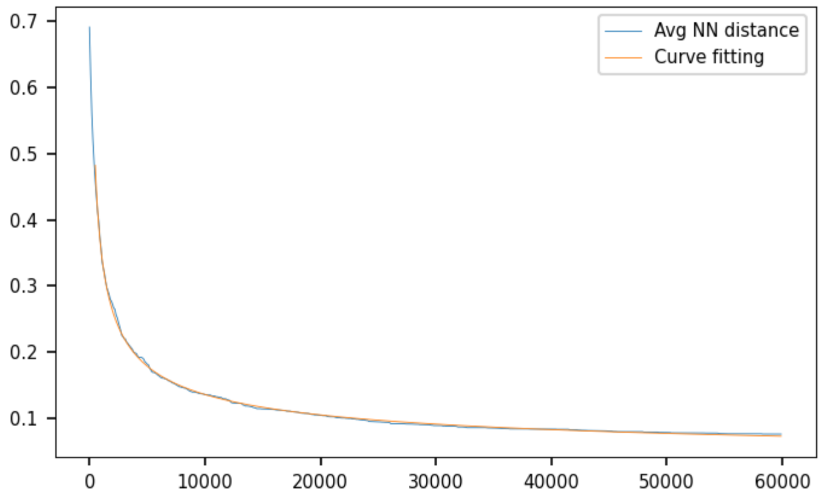

Figure 1: Average NN distance over time, with probabilistic ANN

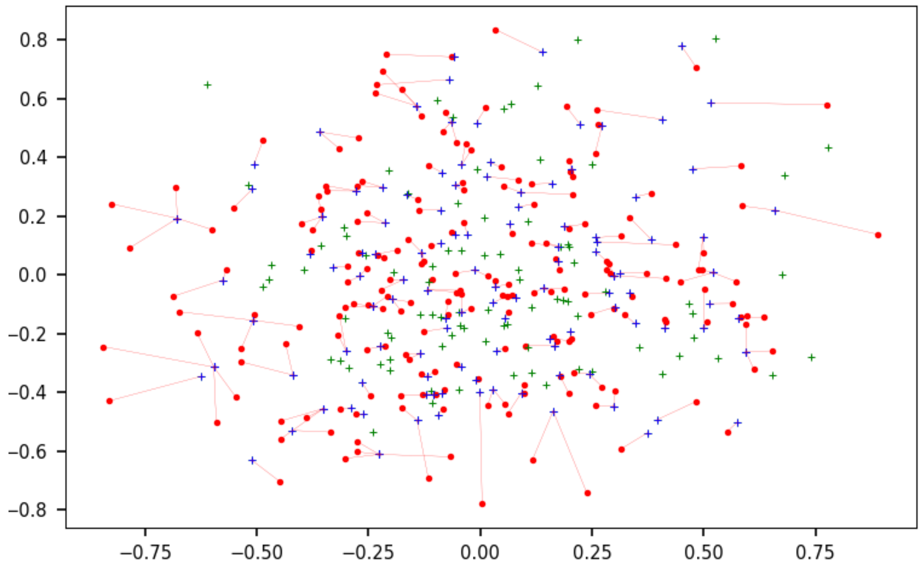

Figure 2: Approximate NNs from T (blue) to points in S (red) after a few thousand steps

Figure 2 shows the dataset used in Figure 1, with red segments linking points in S (red) to their closest neighbor in T (blue) obtained at the current iteration. View video here showing how the approximate nearest neighbors get more and more accurate over time, from beginning to end.

2 Applications

The methodology presented here is useful in several contexts. Now, I describe how to leverage my algorithm in various applications, ranging from traditional to GenAI and LLM.

2.1 Embeddings and large language models

In large language models, embeddings are lists of tokens attached to a keyword. For instance, Table 1 is based on my specialized XLLM that answers research questions about statistics and probability: see here. The table features embeddings, for 7 one-token keywords. For each keyword, the top row shows the top 10 tokens. The bottom one shows the importance of each token: the numbers represent the normalized pointwise multual information (PMI) between a token and a keyword. This metric measures the strength of the association between a token and a keyword. Here, I use variable-length embeddings [1] to reduce the size of the embedding tables.

To measure the similarity between two words, first compute the dot product (the $\bullet$ product) [Wiki] between the two word embeddings. Tokens with a PMI of zero for either word (that is, absent for one of the two words) are ignored in the computations. Then, compute the norm $\|\cdot\|$ of each word. The norm is the square root of the sum of squared PMIs. For instance, based on Table 1:

$$\text{normal} \bullet \text{Gaussian} = 0.43903 \times 0.00858 + 0.05885 \times 0.01164 = 0.004452$$
$$\text{binomial} \bullet \text{Gaussian} = 0.11796 \times 0.00858 + 0.01117 \times 0.01164 = 0.001142$$

$$\|\text{normal}\| = 0.49678, \quad \|\text{Gaussian}\| = 0.05853, \quad \|\text{binomial}\| = 0.13998.$$

There are many different ways to define the similarity between two words. The cosine similarity [Wiki] is one of them. It normalizes the dot products, but does not capture magintudes. It is computed as follows:

$$\rho(\text{normal}, \text{Gaussian}) = \frac{\text{normal} \bullet \text{Gaussian}}{\|\text{normal}\| \cdot \|\text{Gaussian}\|} = 0.15311,$$
$$\rho(\text{binomial}, \text{Gaussian}) = \frac{\text{binomial} \bullet \text{Gaussian}}{\|\text{normal}\| \cdot \|\text{Gaussian}\|} = 0.13940.$$

Whether using the dot product or cosine similarity, “normal” is closer to “Gaussian” than “binomial”. The distance may then be defined as $1 - \rho$. The goal, given two sets of embeddings S and T, is to find, for each embedding in S, its closest neighbor in T. For instance, S may consist of the top 1000 standardized user queries with associated embeddings (stored in cache for fast real-time retrieval), and T maybe the full list of embeddings based on crawling and/or parsing your entire repository.

word	token 1	token 2	token 3	token 4	token 5	token 6	token 7	token 8	token 9	token 10
hypothesis	alternative	null	statistical	false	test	nested	testing	type	bourget	chinese
	0.05070	0.03925	0.03539	0.03177	0.01885	0.01661	0.01358	0.01056	0.01011	0.01011
test	statistical	wilcoxon	negative	alternative	alpha	fisher	kolmogorov	contingency	type	false
	0.09546	0.05842	0.03206	0.02700	0.02519	0.02456	0.02224	0.02099	0.02066	0.01924
normal	distribution	bivariate	standard	log	multivariate	variate	ratio	trivariate	sum	difference
	0.43903	0.15486	0.10019	0.09719	0.05885	0.05204	0.03569	0.03368	0.03240	0.03074
Gaussian	inverse	joint	increment	multivariate	physicists	spectrum	noisy	distribution	board	polygon
	0.04340	0.02718	0.01164	0.01164	0.01164	0.01006	0.00964	0.00858	0.00832	0.00774
walk	random	self-avoiding	wiener	connective	polya	levy	two-dim	lattice	trajectories	confined
	0.16104	0.10019	0.04138	0.02888	0.01691	0.01491	0.01447	0.01344	0.01004	0.01004
random	walk	variable	number	sequence	independent	set	constant	polya	one-dim	process
	0.16104	0.10245	0.08385	0.06631	0.05068	0.03509	0.03230	0.03028	0.02939	0.02844
binomial	distribution	negative	approximation	integer	beta	multivariate	discrete	trial	rise	infinity
	0.11796	0.06830	0.01455	0.01327	0.01133	0.01117	0.01039	0.00990	0.00944	0.00886

Table 1: Embeddings (one per word) with normalized PMI score attached to each token

When the goal is to compute all nearest neighbors withing T (in this case, $S = T$), the XLLM architecture is especially efficient. It uses a separate embedding table for each top category. Assuming q tables respectively with $N_1, \dots, N_q$ embeddings, standard k-NN over categories bundled together is $O(N^2)$ with $N = N_1 + \cdots + N_q$, versus the much lower $O(N_1^2 + \cdots + N_q^2)$ when the q categories are treated separately. With the ANN algorithm described in section 1, these computing times are significantly reduced. However, with q categories, you must add a little overhead time and memory as there is a top layer for cross-category management. When a category has more than (say) 5000 embeddings, further acceleration is achieved by splitting its table into smaller batches, and compute nearest neighbors on each batch separately. The solid gain in speed usually outweighs the small loss in accuracy. For prompt compression to reduce the size of the input user queries, see here.

2.2 Generating and evaluating synthetic data

My first use of probabilistic ANN (PANN) was for synthesizing tabular data, see here. It led to a faster and better alternative to GAN (generative adversarial networks), and was actually called NoGAN as it does not require neural networks. But it also helps with various related GenAI problems. For instance:

- **Refining existing synthetic data**. Say you have a real dataset T, and and you created a synthetic version of it, the S set. You can generate much more observations than needed in your synthetic data, then only keep the best ones. To do this, only keep in S observations with a nearest neighbor in T that is close enough. In short, you discard synthetic observations that are too far away from any real observation. This simple trick will improve the quality of your synthetic data, if the goal is good enough replication of the underlying distribution in the real data. PANN is particularly handy to solve this problem.
- **Evaluating the quality of synthetic data**. The best metrics to evaluate the faithfulness of synthetic data are typically based on the multivariate empirical cumulative distributions (ECDF), see here. The ECDF is evaluated at various locations z in the feature space, computed both on the synthetic data S, and the real data T. In particular, the Kolmogorov-Smirnov distance is defined as

$$\text{KS}(S,T) = \sup_z |F_\text{s}(z) - F_\text{r}(z)|,$$

where F_s, F_r are the ECDFs, respectively for the synthetic and real data. It involves finding the closest neighbors to each z, both in S and T. Again, the PANN algorithm can help accelerate the computations.

For an alternative to PANN, based on interpolated binary search and radix encoding, see here. Several nearest neighbor search methods are discussed in the article "Comprehensive Guide To Approximate Nearest Neighbors Algorithms", available here.

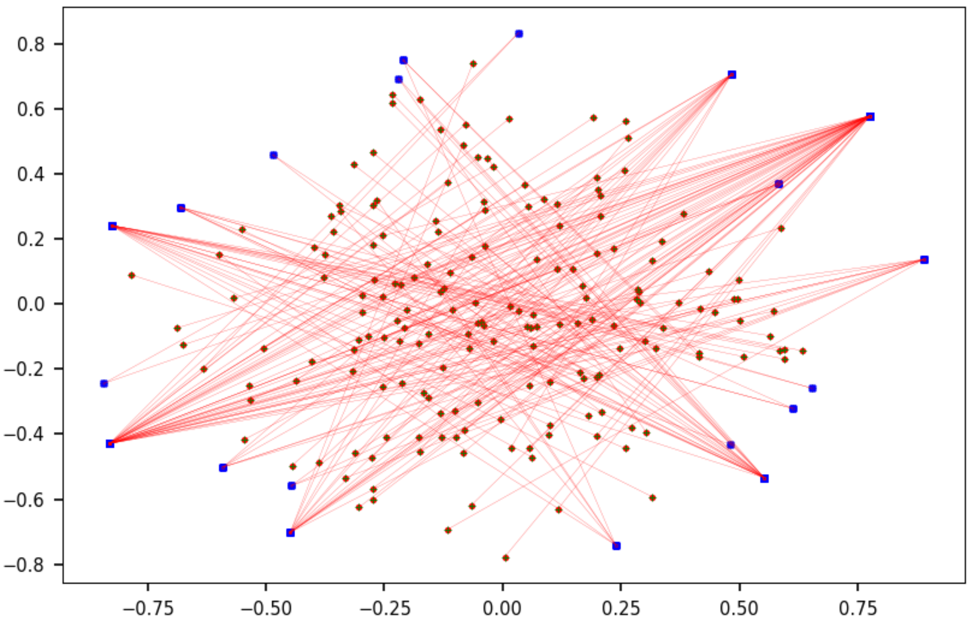

Figure 3: Extreme points in blue ($S = T$) obtained by maximizing the loss $L(\sigma)$

2.3 Clustering, dataset comparisons, outlier detection

Nearest neighbor (NN) methods were first used in classification algorithms, with $S = T$: the ancestor of PANN is K-NN. To adapt PANN to $K > 1$, proceed as follows. If you want the three approximate nearest neighbors ($K = 3$) to observation x_{k_1} in S, keep two extra copies of x_{k_1}, say x_{k_2} and x_{k_3}, in S. At any time in the iterative algorithm, flag any nearest neighbor assigned to one of the copies, say x_{k_2}, as nonassignable to the two other copies, in this case x_{k_1} and x_{k_3}. To do so, add the nearest neighbor in question (its index in T) to the lists hask[k_1] and hash[k_3] in line 138 in the Python code in section 3. You also need to set optimize='speed' in line 117 so that hash is active. In the code, the nearest neighbor to x_{k_1} in T, is y_j with $j =$arr_NN[k_1]. Its index is j.

For classification purposes, a new point x_{k_1} in S, outside the training set T, gets assigned using majority vote, by looking at the class assigned to its nearest neighbors in T. For clustering (unsupervised classification) the same rule applies, but there is no class label: the end result is a clustering structure that groups points in unlabeled clusters based on proximity.

Beyond classification, PANN is also helpful to find similar datasets in a database. For example, images or soundtracks. Each dataset hast its feature vector, consisting not necessarily of pixels or waves, but instead, where vector components are summary statistics about the image or soundtrack. This is an extension of what I discussed in section 2.2. It also applies to comparing time series, where vectors consist of autocorrelations of various lags, with one vector per time series. Finally, if you maximize rather than minimize the loss function, you can detect extreme points as opposed to nearest neighbors: see Figure 3.

3 Python code

See section 2.3 for details about hash, arr_NN and the parameter optimize. The sort_x_by_NNdist_to_y function is used together with K=int(0.5*N) in line 80 to accelerate the speed of the algorithm, by processing only the top K observations in S at each epoch, sorted by nearest distance to T. An epoch (same meaning as in neural networks) is a full run of S. However here, only K observations out of N are processed during an epoch. Yet, due to re-sorting S at the end of each epoch, the K observations change at each epoch. Thus over time, all observations are used. To not use this feature, set K=N.

The parameter decay in line 151 offers a different acceleration mechanism. The default value is zero. A positive value provides a boost at the beginning, where it is most needed. A negative value always yields lower performance, yet the resulting loss function goes up and down (rather than only down), in a way similar to stochastic gradient descent in deep neural networks: there is no benefit to it, other than educational.

Acceleration mechanisms offer a modest boost, about 10% in my tests. But I haven't investigated them thoroughly. If you remove them, it will reduce the length of the code and make it easier to understand. There is also a significant amount of code to produce the visualizations and the video. If you remove this, the code will be significantly shorter. The code is also on GitHub, here, with a shorter version here.

```
1  # Probabilistic ANN, can be used for clustering / classification
2
3  import numpy as np
4  import matplotlib.pyplot as plt
5  from scipy.optimize import curve_fit
6  import matplotlib as mpl
7  from PIL import Image
8  import moviepy.video.io.ImageSequenceClip
9
10
11 #--- [1] Parameters and functions for visualizations
12
13 def save_image(fname,frame):
14
15    # back-up function in case of problems with plt.savefig
16    global fixedSize
17
18    plt.savefig(fname, bbox_inches='tight')
19    # make sure each image has same size and size is multiple of 2
20    # required to produce a viewable video
21    im = Image.open(fname)
22    if frame == 0:
23       # fixedSize determined once for all in the first frame
24       width, height = im.size
25       width=2*int(width/2)
26       height=2*int(height/2)
27       fixedSize=(width,height)
28    im = im.resize(fixedSize)
29    im.save(fname,"PNG")
30    return()
31
```

```
def plot_frame():

   plt.scatter(x[:,0], x[:,1], color='red', s = 2.5)
   z = []

   for k in range(N):

      neighbor = arr_NN[k]
      x_values = (x[k,0], y[neighbor,0])
      y_values = (x[k,1], y[neighbor,1])
      plt.plot(x_values,y_values,color='red',linewidth=0.1,marker=".",markersize=0.1)
      z_obs = (y[neighbor,0], y[neighbor,1])
      z.append(z_obs)

   z = np.array(z)
   plt.scatter(y[:,0], y[:,1], s=10, marker = '+', linewidths=0.5, color='green')
   plt.scatter(z[:,0], z[:,1], s=10, marker = '+', linewidths=0.5, color='blue')
   return()

mpl.rcParams['axes.linewidth'] = 0.5
plt.rcParams['xtick.labelsize'] = 7
plt.rcParams['ytick.labelsize'] = 7


#--- [2] Create data, initial list of NN, and hash

def sort_x_by_NNdist_to_y(x, y, arr_NN):

   NNdist = {}
   x_tmp = np.copy(x)
   arr_NN_tmp = np.copy(arr_NN)
   for k in range(N):
      neighbor = arr_NN_tmp[k]
      NNdist[k] = np.sum(abs(x_tmp[k] - y[neighbor]))
   NNdist = dict(sorted(NNdist.items(), key=lambda item: item[1],reverse=True ))

   k = 0
   for key in NNdist:
      arr_NN[k] = arr_NN_tmp[key]
      x[k] = x_tmp[key]
      k += 1
   return(x, arr_NN)

seed = 57
np.random.seed(seed)
eps = 0.00000000001

N = 200       # number of points in x[]
K = int(0.5 * N) # sort x[] by NN distance every K iterations
M = 200       # number of points in y[]

niter = 10000
mean = [0, 0]
cov = [(0.1, 0),(0, 0.1)]
x = np.random.multivariate_normal(mean, cov, size=N)
y = np.random.multivariate_normal(mean, cov, size=M)
# y = np.copy(x)
np.random.shuffle(x)
np.random.shuffle(y)

arr_NN = np.zeros(N)
arr_NN = arr_NN.astype(int)
hash = {}
sum_dist = 0

for k in range(N):
```

```python

   # nearest neighbor to x[k] can't be identical to x[k]
   dist = 0

   while dist < eps:
     neighbor = int(np.random.randint(0, M))
     dist = np.sum(abs(x[k] - y[neighbor]))

   arr_NN[k] = neighbor
   sum_dist += np.sum(abs(x[k] - y[neighbor]))
   hash[k] = (-1,)

x, arr_NN = sort_x_by_NNdist_to_y(x, y, arr_NN)
low = sum_dist


#--- [3] Main part

mode   = 'minDist' # options: 'minDist' or 'maxDist'
optimize = 'speed' # options: 'speed' or 'memory'
video  = False   # True if you want to produce a video
decay  = 0.0

history_val = []
history_arg = []
flist = []
swaps = 0
steps = 0
frame = 0

for iter in range(niter):

   k = iter % K
   j = -1
   while j in hash[k] and len(hash[k]) <= N:
      # if optimized for memory, there is always only iter in this loop
      steps += 1
      j = np.random.randint(0, M) # potential new neighbor y[j], to x[k]

   if optimize == 'speed':
      hash[k] = (*hash[k], j)

   if len(hash[k]) <= N:

      # if optimized for memory, then len(hash[k]) <= N, always
      old_neighbor = arr_NN[k]
      new_neighbor = j
      old_dist = np.sum(abs(x[k] - y[old_neighbor]))
      new_dist = np.sum(abs(x[k] - y[new_neighbor]))
      if mode == 'minDist':
         ratio = new_dist/(old_dist + eps)
      else:
         ratio = old_dist/(new_dist + eps)
      if ratio < 1-decay/np.log(2+iter) and new_dist > eps:
         swaps += 1
         arr_NN[k] = new_neighbor
         sum_dist += new_dist - old_dist
         if sum_dist < low:
            low = sum_dist

         if video and swaps % 4 == 0:

            fname='ann_frame'+str(frame)+'.png'
            flist.append(fname)
            plot_frame()

            # save image: width must be a multiple of 2 pixels, all with same size
            # use save_image(fname,frame) in case of problems with plt.savefig
            plt.savefig(fname, dpi = 200)
            plt.close()
            frame += 1

    if iter % K == K-1:
       x, arr_NN = sort_x_by_NNdist_to_y(x, y, arr_NN)

    if iter % 100 == 0:
       print("%6d %6d %6d %8.4f %8.4f"
             % (iter, swaps, steps, low/N, sum_dist/N))
       history_val.append(sum_dist/N)
       history_arg.append(steps) # try replacing steps by iter


history_val = np.array(history_val)
history_arg = np.array(history_arg)

if video:
   clip = moviepy.video.io.ImageSequenceClip.ImageSequenceClip(flist, fps=6)
   clip.write_videofile('ann.mp4')


#--- [4] Visualizations (other than the video)

plot_frame()
plt.show()

#- curve fitting for average NN distance (Y-axis) over time (X-axis)

# works only with mode == 'minDist'

def objective(x, a, b, c):
   return(a + b*(x**c))

# ignore first offset iterations, where fitting is poor
offset = 5

x = history_arg[offset:]
y = history_val[offset:]

# param_bounds to set bounts on curve fitting parameters
if mode == 'minDist':
   param_bounds=([0,0,-1],[np.inf,np.infty,0])
else:
   param_bounds=([0,0,0],[np.inf,np.infty,1])

param, cov = curve_fit(objective, x, y, bounds = param_bounds)
a, b, c = param
# is c = -1/2 the theoretical value, assuming a = 0?
print("\n",a, b, c)

y_fit = objective(x, a, b, c)
## plt.plot(x, y, linewidth=0.4)
plt.plot(history_arg, history_val, linewidth=0.4)
plt.plot(x, y_fit, linewidth=0.4)
plt.legend(['Avg NN distance','Curve fitting'],fontsize = 7)
plt.show()
```

References

[1] Johnathan Chiu, Andi Gu, and Matt Zhou. Variable length embeddings. *Preprint*, pages 1–12, 2023. arXiv:2305.09967 [Link]. 3