*Vector Databases from Embeddings to Applications*

*Let us try to practically implement the search for similar vectors by starting with K-nearest neighbours :*

```
“import numpy as np

import matplotlib.pyplot as plt

from sklearn.neighbors import NearestNeighbors

import time

np.random.seed(42)

# Generate 20 data points with 2 dimensions

X = np.random.rand(20,2)

# Display Embeddings

n = range(len(X))

fig, ax = plt.subplots()

ax.scatter(X[:,0], X[:,1], label='Embeddings')

ax.legend()

for i, txt in enumerate(n):

ax.annotate(txt, (X[i,0], X[i,1]))

neighbours = neigh.kneighbors([[0.45,0.2]], k, return_distance=True)

print(neighbours)

t0 = time.time()

neighbours = neigh.kneighbors([[0.45,0.2]], k, return_distance=True)

t1 = time.time()

query_time = t1-t0

print(f"Runtime: {query_time: .4f} seconds")

def speed_test(count):

    # generate random objects
```

```
    data = np.random.rand(count,2)


    # prepare brute force index
    k=4
    neigh = NearestNeighbors(n_neighbors=k, algorithm='brute', metric='euclidean')
    neigh.fit(data)
    # measure time for a brute force query
    t0 = time.time()
    neighbours = neigh.kneighbors([[0.45,0.2]], k, return_distance=True)
    t1 = time.time()
    total_time = t1-t0
    print (f"Runtime: {total_time: .4f}")
    return total_time
time20k = speed_test(20_000)
# Brute force examples
time200k = speed_test(200_000)
time2m = speed_test(2_000_000)
time20m = speed_test(20_000_000)
time200m = speed_test(200_000_000)”
```

*Thus creating the algorithm for k-nearest neighbour search and finding out the time require to process various amounts of data*

*Now let us continue with the approximate nearest neighbours for the next similarity search algorithm:*

```
“from random import random, randint
from math import floor, log
import networkx as nx
import numpy as np
import matplotlib as mtplt
```

```
from matplotlib import pyplot as plt

from utils import *

vec_num = 40 # Number of vectors (nodes)

dim = 2 ## Dimention. Set to be 2. All the graph plots are for dim 2. If changed, then plots should be commented.

m_nearest_neighbor = 2 # M Nearest Neigbor used in construction of the Navigable Small World (NSW)

vec_pos = np.random.uniform(size=(vec_num, dim))”
```

*Now let us go with implementing HNSW(Hierarchial Navigable Small Worlds):in which the first step is to construct the graph array which is then to be followed by application of HNSW on searching in the vectors:*

```
“GraphArray = construct_HNSW(vec_pos,m_nearest_neighbor)
for layer_i in range(len(GraphArray)-1,-1,-1):
fig, axs = plt.subplots()
print("layer_i = ", layer_i)
if layer_i>0:
pos_layer_0 = nx.get_node_attributes(GraphArray[0],'pos')
nx.draw(GraphArray[0], pos_layer_0, with_labels=True, node_size=120,
node_color=[[0.9,0.9,1]], width=0.0, font_size=6, font_color=(0.65,0.65,0.65), ax = axs)
pos_layer_i = nx.get_node_attributes(GraphArray[layer_i],'pos')
nx.draw(GraphArray[layer_i], pos_layer_i, with_labels=True, node_size=150,
node_color=[[0.7,0.7,1]], width=0.5, font_size=7, ax = axs)
nx.draw(G_query, pos_query, with_labels=True, node_size=200, node_color=[[0.8,0,0]],
width=0.5, font_size=7, font_weight='bold', ax = axs)
nx.draw(G_best, pos_best, with_labels=True, node_size=200, node_color=[[0.85,0.7,0.2]],
width=0.5, font_size=7, font_weight='bold', ax = axs)
plt.show()”
```

*Now let us utilize this constructed graph array for searching:*

```
“(SearchPathGraphArray, EntryGraphArray) = search_HNSW(GraphArray,G_query)

for layer_i in range(len(GraphArray)-1,-1,-1):

   fig, axs = plt.subplots()

   print("layer_i = ", layer_i)

   G_path_layer = SearchPathGraphArray[layer_i]
```

*pos_path = nx.get_node_attributes(G_path_layer,'pos')*

*G_entry = EntryGraphArray[layer_i]*

*pos_entry = nx.get_node_attributes(G_entry,'pos')*

*if layer_i>0:*

*pos_layer_0 = nx.get_node_attributes(GraphArray[0],'pos')*

*nx.draw(GraphArray[0], pos_layer_0, with_labels=True, node_size=120, node_color=[[0.9,0.9,1]], width=0.0, font_size=6, font_color=(0.65,0.65,0.65), ax = axs)*

*pos_layer_i = nx.get_node_attributes(GraphArray[layer_i],'pos')*

*nx.draw(GraphArray[layer_i], pos_layer_i, with_labels=True, node_size=100, node_color=[[0.7,0.7,1]], width=0.5, font_size=6, ax = axs)*

*nx.draw(G_path_layer, pos_path, with_labels=True, node_size=110, node_color=[[0.8,1,0.8]], width=0.5, font_size=6, ax = axs)*

*nx.draw(G_query, pos_query, with_labels=True, node_size=80, node_color=[[0.8,0,0]], width=0.5, font_size=7, ax = axs)*

*nx.draw(G_best, pos_best, with_labels=True, node_size=70, node_color=[[0.85,0.7,0.2]], width=0.5, font_size=7, ax = axs)*

*nx.draw(G_entry, pos_entry, with_labels=True, node_size=80, node_color=[[0.1,0.9,0.1]], width=0.5, font_size=7, ax = axs)*

*plt.show()"*

*As shown in the plot one can understand the relative position of the vector embeddings with that of the query and its semanticity.*

*For the creation of any vector database on the purpose of the project there are four main steps to follow they are: Downloading the sample data, creating an embedding instance of the respective vector database, Create question collection which is to be used in the vector database and then load the sample data and generate vector embeddings which are to be stored in the vector database and now if any query is passed into the vectordb which is converted into vector on which a similarity search is imposed for the retrieval of related information. Note that vector db is used in terms of sparse, hybrid and dense vector embeddings and their relative use of search depends on the computational resource allocated through and the model available to search through.*

*Make a note that vector databases can be used for CRUD operations like:*

*"#Create an object*

*object_uuid = client.data_object.create( data_object={ 'question':"Leonardo da Vinci was born in this country.", 'answer': "Italy", 'category': "Culture" }, class_name="Question" )*

*print(object_uuid)"*

*"#Reading*

*data_object = client.data_object.get_by_id(object_uuid, class_name="Question")*

*json_print(data_object)*

*data_object = client.data_object.get_by_id(*

*object_uuid,*

*class_name='Question',*

*with_vector=True*

*)*

*json_print(data_object)"*

*"#Updating*

*client.data_object.update( uuid=object_uuid, class_name="Question", data_object={ 'answer':"Florence, Italy" })*

*data_object = client.data_object.get_by_id( object_uuid, class_name='Question',) json_print(data_object)"*

*"#Deleting*

*json_print(client.query.aggregate("Question").with_meta_count().do())*

*client.data_object.delete(uuid=object_uuid, class_name="Question")*

*json_print(client.query.aggregate("Question").with_meta_count().do())"*