International Journal of Scientific Research in Computer Science, Engineering and Information Technology

ISSN : 2456-3307 OPEN ACCESS

Available Online at : www.ijsrcseit.com

doi : https://doi.org/10.32628/IJSRCSEIT

IP2VEC Comparing Similarities between IP Addresses

*[1]Sai Viraj, Yashvi Shah[2], Swetha Sunkara[3]

[1]Computer Science Department, New Jersey Institute of Technology, Harrison, USA

[2]Computer Science Department, New Jersey Institute of Technology, New York, USA

[3]Computer Science Department, New Jersey Institute of Technology, Livingston, USA

ARTICLEINFO

Article History:

Accepted: 15 Nov 2023

Published: 30 Nov 2023

Publication Issue

Volume 9, Issue 6

November-December-2023

Page Number

186-195

ABSTRACT

There are numerous cyber-attacks in the network and analyzing IP addresses is used to

1. Unusual access patterns
2. Spike in data transfer
3. Multiple failed attempts
4. Anomaly detections

Keywords: Unusual access patterns, Spike in data transfer, Multiple failed attempts, Anomaly detections

I. INTRODUCTION

How do we compare the similarities?

1. We use Minkowski distance, which is used to measure distance between two points in the multidimensional space.
2. It is widely used in Machine Learning algorithms
3. Here we are using it to calculate the closeness of 2 IP addresses.
4. So when access request is received into the network the controller makes sure that the IP address is similar to its previous IP address in the data set.
5. If it is new then the credentials were tracked and make sure that is from the authorised user.

What we are doing

1. The algorithm we inspired form is Word2vec. Which is used in NLP.
2. We consider the IP address as labels of "text".
3. We used neural network to form vectors of data for easy computation in network space.
4. Then we train the network according the data set.
5. Once it is trained we try to find the similarities in the IP addresses.
6. These relationships are stores as weights

II. EVALUATING THE GRAPHS

1. Categories IP Addresses into types such as Private, Public, Multicast, Broadcast, Link Local, Default Network
2. Utilize a graph-based metric called GRAPH
3. Calculates distances between IP Addresses based on their category.
4. The IP addresses are labeled according to whether they are infected or normal.

ALOGIRTHM :

1. We are using IP2vec algorithm, which is unsupervised machine learning algorithm.
2. We use visualisation technique to show clusters in 2D space.
3. The key features selected for similarity calculation are Source IP Address, Destination IP Address, Destination Port, and Protocol.
4. Flows in the network data are considered as "sentences" for training.
5. when the Source IP Address is chosen as the input word, the context words include the Destination IP Address, Destination Port, and Protocol

Identification of botnets

1. The dataset contains both normal and infected activities
2. The goal is to see if IP2Vec can learn to separate the two based on the behaviors of the hosts.
3. Each IP address is represented in a vector space.
4. t-SNE (t-distributed Stochastic Neighbor Embedding) is used to visualize the learned similarities in a two-dimensional space.
5. It group IP addresses with high similarities together.
6. Output visualisation- IP2Vec, when trained on dataset, successfully identifies and separates infected hosts from normal hosts
7. In configuring IP2Vec, a hidden layer size of 32 neurons is employed

Code:

```
IN[1]!pip install torch
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Requirement already satisfied: torch in /usr/local/lib/python3.8/dist-packages (1.13.0+cu116)
Requirement already satisfied: typing-extensions in /usr/local/lib/python3.8/dist-packages (from torch) (4.4.0)
IN[2] import pandas as pd
import numpy as np
import os
import matplotlib.pyplot as plt
import seaborn as sns
import random
from tqdm import tqdm
import torch as th
from torch.autograd import Variable as V
from torch import nn,optim
import numpy as np
import random
pd.set_option('display.max_columns', None)
%load_ext google.colab.data_table
from google.colab import data_table
from google.colab import drive
drive.mount('/content/drive/')
# os.chdir("Your Location here")
import torch
from torch import nn
device = torch.device("cuda" if
torch.cuda.is_available() else "cpu")
print(device)
data_folder = './data/'
```

Preprocessing

1.a. Load and Preprocess Data

```
In [3]:
df = pd.read_csv(data_folder + 'Darknet.CSV',
on_bad_lines='skip')

print(df.shape)

df.iloc[:5]
(141530, 85)
Warning: Total number of columns (85) exceeds max_
columns (20). Falling back to pandas display.
```

	Flow ID	Src IP	Src Port	Dst IP	Dst Port	Protocol	Timestamp	Fl Durat
0	10.152.152.11-216.58.220.99-57158-443-6	10.152.152.11	57158	216.58.220.99	443	6	24/07/2015 04:09:48 PM	
1	10.152.152.11-216.58.220.99-57159-443-6	10.152.152.11	57159	216.58.220.99	443	6	24/07/2015 04:09:48 PM	
2	10.152.152.11-216.58.220.99-57160-443-6	10.152.152.11	57160	216.58.220.99	443	6	24/07/2015 04:09:48 PM	
3	10.152.152.11-74.125.136.120-49134-443-6	10.152.152.11	49134	74.125.136.120	443	6	24/07/2015 04:09:48 PM	
4	10.152.152.11-173.194.65.127-34697-19305-6	10.152.152.11	34697	173.194.65.127	19305	6	24/07/2015 04:09:45 PM	10778

```
IN[4]  print('Total Number of flows:',len(df['Flow ID'].unique()))
print('Obs per flow, 10 largets:\n', df['Flow ID'].value_counts().value_counts()[:10])

df['src-dst'] = df['Src IP']+'-'+df['Dst IP']
print('Number of unique flows per Flow ID:\n', df.groupby(['Flow ID'])['src-dst'].nunique().value_counts())

df['src-dst-dst port-protocol'] = df['Src IP']+'-'+df['Dst IP']+'-'+df['Dst Port'].astype(str)+'-'+df['Protocol'].astype(str)
print('Number of unique flow charactaristics per Flow ID:\n', df.groupby(['Flow ID'])['src-dst-dst port-protocol'].nunique().value_counts())

filtered_df = df[['Src IP', 'Dst IP', 'Dst Port', 'Protocol', 'Timestamp']].drop_duplicates()
filtered_df['Dst Port'] = filtered_df['Dst Port'].astype(str)
filtered_df['Protocol'] = filtered_df['Protocol'].astype(str)
filtered_df.iloc[:5]
```

```
Total Number of flows: 77568
Obs per flow, 10 largets:
 1    49363
2     17841
4      5995
3      1917
6       844
8       616
10      197
5       156
12      114
16       53
Name: Flow ID, dtype: int64
Number of unique flows per Flow ID:
 1    77568
Name: src-dst, dtype: int64
Number of unique flow charactaristics per Flow ID:
 1    77568
Name: src-dst-dst port-protocol, dtype: int64
```

Out[4]:

	Src IP	Dst IP	Dst Port	Protocol	Timestamp
0	10.152.152.11	216.58.220.99	443	6	24/07/2015 04:09:48 PM
3	10.152.152.11	74.125.136.120	443	6	24/07/2015 04:09:48 PM
4	10.152.152.11	173.194.65.127	19305	6	24/07/2015 04:09:45 PM
5	10.152.152.11	173.194.65.127	443	6	24/07/2015 04:10:00 PM
6	173.194.33.97	10.152.152.11	56254	6	24/07/2015 04:09:45 PM

IN[5] 1.b. Create Utility Functions for Generation

```
In [6]:
def gen_sameples_and_vocab_from_a_flow(df, flatten=True, source_ip = 'Src IP', dest_ip = 'Dst IP', proto = 'Protocol', dest_port = 'Dst Port'):
    """
    Create pairs from a Flow dataset and vocabulary
    """
    df = df.reset_index(drop=True)
    pairs = [(
        (df[source_ip][j],df[dest_ip][j]),
        (df[source_ip][j],df[dest_port][j]),
        (df[source_ip][j],df[proto][j]),
        (df[dest_port][j], df[dest_ip][j]),
        (df[proto][j], df[dest_ip][j])
    ) for j in range(len(df))]

if flatten:
pairs = [item for sublist in pairs for item in sublist]
vocab = set(df[source_ip].tolist() + df[dest_ip].tolist() + df[dest_port].tolist() + df[proto].tolist())
    vocab_size = len(vocab)
    word_to_ind = {word:idx for idx,word in enumerate(vocab)}
```

```
  ind_to_word = {idx:word for idx,word in
enumerate(vocab)}
flatten=True, source_ip = 'Src IP', dest_ip = 'Dst IP',
proto = 'Protocol', dest_port = 'Dst Port'):
```

Create pairs from a Flow dataset and vocabulary
Output:

```
  Vocabulary size: 20472
  First five elements:
   ['27627', '23.36.90.113', '51947', '42788', '1797']
  First five nested pairs:
]: [('10.152.152.11', '216.58.220.99'),
    ('10.152.152.11', '443'),
    ('10.152.152.11', '6'),
    ('443', '216.58.220.99'),
    ('6', '216.58.220.99'),
    ('10.152.152.11', '74.125.136.120'),
    ('10.152.152.11', '443'),
    ('10.152.152.11', '6'),
    ('443', '74.125.136.120'),
    ('6', '74.125.136.120')]
```

NS TABLE

```
corpus = pairs
voc_size = vocab_size
print(voc_size)
corpus[:5]
20472
Out[6]:
[('10.152.152.11', '216.58.220.99'),
 ('10.152.152.11', '443'),
 ('10.152.152.11', '6'),
 ('443', '216.58.220.99'),
 ('6', '216.58.220.99')]

In[7]  t = 60
power_parameter = 0.75
ns_table_size = 100000

def make_ns_table_ips(pairs, t=60,
power_parameter=0.75, ns_table_size=100000):
  """
  pairs - pairs of entities. Probably
(ip_source,ip_destination) in our real data.
  t - threshold variable from the paper.
  """
  from collections import Counter
  import operator

  flat_list_of_pairs = [item for sublist in pairs for item
in sublist]
  counter = Counter(flat_list_of_pairs)
increasingly_sorted_counter =
dict(sorted(counter.items(),
key=operator.itemgetter(1)))
  increasingly_sorted_counter_keys =
list(increasingly_sorted_counter.keys())
  increasingly_sorted_counter_values =
np.array(list(increasingly_sorted_counter.values()))

increasingly_sorted_counter_values[increasingly_sort
ed_counter_values<t] = 0
  increasingly_sorted_counter_thresholded = {i : j for i,
j in zip(increasingly_sorted_counter_keys,
increasingly_sorted_counter_values)}
  decreasingly_sorted_counter_thresholded =
dict(sorted(increasingly_sorted_counter_thresholded.i
tems(), key=operator.itemgetter(1),reverse=True))

  # convert to frequencies rather than counts
  # S =
np.sum(np.array(list(decreasingly_sorted_counter_thr
esholded.values())))
  S = 1

decreasingly_sorted_counter_thresholded_freq_value
s = [decreasingly_sorted_counter_thresholded[k]/S for
k in decreasingly_sorted_counter_thresholded.keys()]
  decreasingly_sorted_counter_thresholded_keys =
list(decreasingly_sorted_counter_thresholded.keys())

  # Create negative_sampling table
  # The negative sampling probabilities are
proportional to the frequencies
  # to the power of a constant (typically 0.75).
  freqs_sorted =
[(decreasingly_sorted_counter_thresholded_keys[j],
decreasingly_sorted_counter_thresholded_freq_value
```

```
s[j]) for j in
range(len(decreasingly_sorted_counter_thresholded_
keys))]
  ns_table = {}
  sum_freq = 0
  for w, freq in freqs_sorted:
ns_freq = freq ** power_parameter
      ns_table[w] = ns_freq
      sum_freq += ns_freq

  # Convert the negative sampling probabilities to
integers, in order to make
  # sampling a bit faster and easier.
  # We return a list of tuples consisting of:
  # - the word
  # - its frequency in the training data
  # - the number of positions reserved for this word in
the negative sampling table
  scaler = ns_table_size / sum_freq
  results = [(w, freq, int(round(ns_table[w]*scaler))) for
w, freq in freqs_sorted]
  return results

IN[8]
   ns_table = make_ns_table_ips(pairs)

print('Sum:', np.sum([x[2] for x in ns_table]))
ns_table[:5]
Sum: 99901
Output[8]:
[('10.152.152.11', 160497.0, 10787),
 ('6', 110630.0, 8160),
 ('17', 54310.0, 4786),
 ('53', 23346.0, 2541),
 ('443', 23026.0, 2514)]
```

1) GENERATION OF TARGET TEXT PAIRS

```
   K=6
     unflattened_list = [[x[0]]*x[2] for x in ns_table]
flattened_list = np.array([item for sublist in
unflattened_list for item in sublist])
false_entities = np.random.choice(flattened_list,
size=k)
                                                  In [10]:
pairs_batch = pairs[:16]
pairs_batch

Output[10]:
[('10.152.152.11', '216.58.220.99'),
 ('10.152.152.11', '443'),
 ('10.152.152.11', '6'),
 ('443', '216.58.220.99'),
 ('6', '216.58.220.99'),
 ('10.152.152.11', '74.125.136.120'),
 ('10.152.152.11', '443'),
 ('10.152.152.11', '6'),
 ('443', '74.125.136.120'),
 ('6', '74.125.136.120'),
 ('10.152.152.11', '173.194.65.127'),
 ('10.152.152.11', '19305'),
 ('10.152.152.11', '6'),
 ('19305', '173.194.65.127'),
 ('6', '173.194.65.127'),
 ('10.152.152.11', '173.194.65.127')]
class ContextGenerator:
  def __init__(self, ns_table, k=20):

    # Cerate a repeated list of negative samples, using
ns_table
    unflattened_list = [[x[0]]*x[2] for x in ns_table]
    flattened_list = np.array([item for sublist in
unflattened_list for item in sublist])
    self.ns_table = flattened_list
    self.k = k

  def create_batch(self, true_pairs):
```

```
    """
    Create a batch of examples to be entered later to the ip2vec model.
    - pairs_batch: a list of true pairs, containing pairs of entities like described in
      the ip2vec paper [(ip1,ip2)(ip1,destination_protocol2),...].
    - ns_table: A list of different entities from the data, with their empirical
      probabilities, and with the number of times they are to repeat in the table
      that we will train on.

    stages:
    1. Given desired number of examples:
      - Sample alpha in [1, int(theta*<batch_size>)] to make the number of true
        pairs in the batch. The rest of the sample [int(0.25*<batch_size>), (<batch_size>-1)]
        will be the false pairs to be generated.
      - Draw <alpha> number of obs from pairs
    Draw (<batch_size>-<alpha>) ip addresses from ns_table.
      - Choose with replacement ip addresses from the true pairs and assign them with
        new destinations.
    # unpack from self
    ns_table = self.ns_table
    k = self.k
    batch_size = len(true_pairs)
    # define resulting_batch
    resulting_batch = []
    # add positive pairs to batch
    for pair in true_pairs:
      resulting_batch.append([pair[0],pair[1], 1])
    # keep only sources
    sources = [x[0] for x in true_pairs]
    # Draw batch_size-alpha false entities
    false_entities = np.random.choice(flattened_list, size=(k*batch_size))
# add negative pairs to batch
    for j in range(len(false_entities)):
      resulting_batch.append([np.random.choice(sources), false_entities[j], 0])
    return np.array(resulting_batch)
  context_generator = ContextGenerator(ns_table=ns_table)
context_generator.create_batch(true_pairs=pairs[:5])
```

Output [12]:

```
array([['10.152.152.11', '216.58.220.99', '1'],
       ['10.152.152.11', '443', '1'],
       ['10.152.152.11', '6', '1'],
       ['443', '216.58.220.99', '1'],
       ['6', '216.58.220.99', '1'],
       ['10.152.152.11', '54527', '0'],
       ['10.152.152.11', '6', '0'],
       ['10.152.152.11', '131.202.240.150', '0'],
       ['6', '10000', '0'],
       ['10.152.152.11', '72.21.81.253', '0'],
       ['6', '201.102.63.143', '0'],
       ['10.152.152.11', '17500', '0'],
       ['6', '53', '0'],
       ['10.152.152.11', '173.194.65.155', '0'],
       ['10.152.152.11', '131.202.6.3', '0'],
       ['10.152.152.11', '121.215.39.106', '0'],
       ['10.152.152.11', '54303', '0'],
       ['6', '17', '0'],
       ['10.152.152.11', '5.39.84.13', '0'],
       ['443', '19302', '0'],
       ['443', '173.194.112.109', '0'],
       ['6', '164.215.110.6', '0'],
       ['10.152.152.11', '80', '0'],
       ['10.152.152.11', '24.35.234.148', '0'],
       ['10.152.152.11', '188.122.93.4', '0'],
       ['10.152.152.11', '6', '0'],
       ['10.152.152.11', '10.152.152.11', '0'],
       ['6', '78.46.223.24', '0'],
       ['443', '10.152.152.11', '0'],
       ['6', '6', '0'],
       ['10.152.152.11', '10.152.152.11', '0'],
       ['10.152.152.11', '10.152.152.11', '0'],
       ['10.152.152.11', '131.202.244.5', '0'],
```

```
      ['6', '10.152.152.11', '0'],
      ['10.152.152.11', '185.21.217.78', '0'],
      ['10.152.152.11', '6', '0'],
      ['443', '62.210.36.168', '0'],
      ['443', '443', '0'],
      ['10.152.152.11', '17', '0'],
      ['10.152.152.11', '53', '0'],
      ['443', '10.152.152.10', '0'],
      ['443', '10.152.152.11', '0'],
      ['10.152.152.11', '41.33.142.195', '0'],
      ['443', '89.169.32.27', '0'],

IN[13] batch =
context_generator.create_batch(true_pairs=pairs[:5])
batch[:,2]
Output[13]:
array(['1', '1', '1', '1', '1', '0', '0', '0', '0', '0', '0', '0', '0',
      '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0',
      '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0',
      '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0',
      '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0',
      '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0',
      '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0',
      '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0',
      '0'], dtype='<U21')

batch[:,2].astype(int)

Output14:
array([1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
      0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
      0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
      0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
      0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0])

IN[15] class IP2Vec(nn.Module):
  def __init__(self, vocab_size, word_to_ind,
embedding_dim=128):
    super().__init__()
device = torch.device("cuda" if
torch.cuda.is_available() else "cpu")
    self.device = device
    self.word_to_ind = word_to_ind
    self.vocab_size = vocab_size
    self.embedding_dim = embedding_dim
    self.u_embedding =
nn.Embedding(vocab_size,embedding_dim)
    self.v_embedding =
nn.Embedding(vocab_size,embedding_dim)
    self.log_sigmoid = nn.LogSigmoid()

    init_range= 0.5/embedding_dim
    self.u_embedding.weight.data.uniform_(-
init_range,init_range)
    self.v_embedding.weight.data.uniform_(-0,0)

    # Target word embeddings
    self.w = nn.Embedding(voc_size, embedding_dim)
    # Context embeddings
    self.c = nn.Embedding(voc_size, embedding_dim)

  def get_ip_embeddings(self,ip):
    ip = torch.tensor([word_to_ind[ip]])
def forward(self, targets, contexts): # target,
context,neg
    # batch looks like the following:
    #   array([[a,b,1],
    #          [c,d,1],
    #          [c,m,0],
    #          [c,g,0]])

    # Look up the embeddings for the target words.
    # shape: (batch size, embedding dimension)
    tagets_embeddings = self.u_embedding(targets)
    n_batch, _ = tagets_embeddings.shape
    # View this as a 3-dimensional tensor, with
    # shape (batch size, 1, embedding dimension)
    tagets_embeddings =
tagets_embeddings.view(n_batch, 1,
self.embedding_dim)
    # Look up the embeddings for the positive and
negative context words.
    # shape: (batch size, nbr contexts, emb dim)
    context_embeddings = self.v_embedding(contexts)
```

```
# Transpose the tensor for matrix multiplication
    # shape: (batch size, emb dim, nbr contexts)
    context_embeddings =
context_embeddings.view(n_batch, 1,
self.embedding_dim)
    context_embeddings =
context_embeddings.transpose(1,2)

    # Compute the dot products between target word
embeddings and context
    # embeddings. We express this as a batch matrix
multiplication (bmm).
    # shape: (batch size, 1, nbr contexts)
    dots =
tagets_embeddings.bmm(context_embeddings)

    # View this result as a 2-dimensional tensor.
    # shape: (batch size, nbr contexts)
    dots = dots.view(n_batch, 1)

    return dots
```

MAIN CONTENT OF CODE:

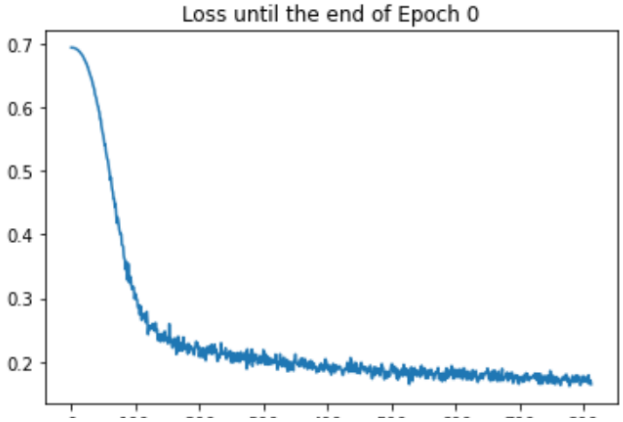
Run

In [18]:

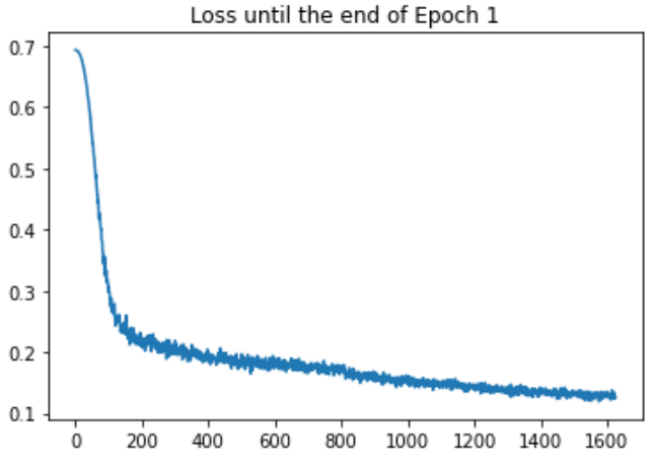
```
model = None

def main():
    global model

    model = IP2Vec(vocab_size,embedding_dim=128,
word_to_ind=word_to_ind)
    trainer = SGNSTrainer(pairs = pairs, model = model,
word_to_ind = word_to_ind, batch_size = 2**9,
ns_table = ns_table, n_epochs = 2, k = 20)

    trainer.train()

main()
```

output:

```
Epoch 1.
iter 200, loss 0.22421549260616302
iter 400, loss 0.19828106462955475
iter 600, loss 0.18850235641002655
iter 800, loss 0.17726364731788635
epoch: 0, loss: 0.0000, time: 1107.83
```




```
Epoch 2.
iter 200, loss 0.15415599942207336
iter 400, loss 0.1420280784368515
iter 600, loss 0.1340135633945465
iter 800, loss 0.12631815671920776
epoch: 1, loss: 0.0000, time: 1081.42
```



model.get_ip_embeddings('24.251.45.232')

```
Out[19]: tensor([[ 3.6288e-03,  4.2481e-04,  7.4446e-04,  1.0061e-03, -2.2955e-03,
                  -3.7387e-03, -2.2634e-03,  3.2091e-03,  2.3584e-03,  2.1584e-03,
                   3.5268e-03,  7.7240e-04, -3.2760e-03, -2.1235e-03,  3.5935e-03,
                   3.0345e-03,  3.2488e-03,  2.3015e-03, -5.9649e-04, -1.2279e-03,
                  -3.7073e-03, -2.1068e-04,  2.8173e-03,  2.0193e-03,  1.9791e-03,
                   8.6873e-04,  1.2119e-03, -1.9423e-03, -2.9486e-03, -5.0157e-04,
                   1.6001e-03, -3.0591e-03, -2.6715e-03,  2.4930e-03, -2.4265e-03,
                   2.0759e-04, -2.0403e-03,  8.3317e-04, -3.6936e-03, -8.4493e-04,
                   1.2296e-03, -1.4259e-03, -1.8743e-04, -8.0446e-04,  1.2302e-03,
                  -2.6887e-03, -1.0998e-03, -1.1958e-03, -1.1340e-03,  1.0223e-03,
                   6.1883e-04,  1.5530e-03, -1.4592e-03,  2.3770e-04,  2.9960e-03,
                   3.6418e-03,  2.5965e-03,  3.3470e-04,  7.6952e-04,  7.2036e-04,
                  -1.0445e-03, -2.8167e-03,  2.7182e-03,  3.1318e-03, -3.8356e-03,
                   1.7991e-03, -1.7574e-03, -6.5373e-04, -1.7340e-03,  2.1261e-03,
                   2.4940e-04, -3.3948e-05, -1.9866e-03, -3.3463e-03,  9.6328e-04,
                  -2.9070e-03, -2.5603e-03,  3.1713e-03, -3.0577e-03,  2.9776e-03,
                   1.1144e-03,  3.2255e-03, -1.0939e-03, -3.8096e-03, -3.5864e-03,
                  -8.2223e-04, -3.0289e-03, -1.0082e-03,  3.6039e-03,  1.1830e-03,
                   2.7311e-03, -1.7661e-03, -3.0120e-03, -3.3092e-03,  1.8226e-03,
                   2.3125e-03,  3.2149e-03,  3.4256e-03, -3.6431e-03, -3.6146e-04,
                  -7.0968e-04,  1.8929e-03, -5.7139e-05, -3.2452e-03,  2.0324e-03,
                   2.0944e-03,  1.5687e-03,  3.3469e-03, -2.7772e-03,  1.6872e-04,
                   1.9490e-03, -1.6738e-03, -1.5882e-03, -2.8948e-03, -2.4592e-03,
                  -3.2565e-04, -3.3082e-03,  9.0032e-04, -1.6008e-03,  3.8568e-03,
                   7.9136e-04, -3.4413e-03, -1.8128e-03,  2.2179e-03,  8.4827e-04
```

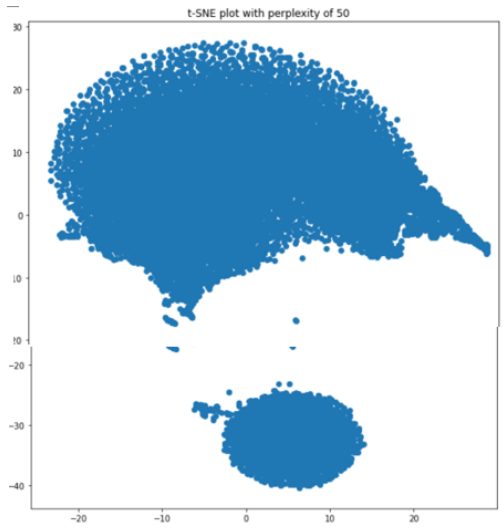
NOW LETS EXECUTE T-SNE

```
from sklearn.manifold import TSNE
from sklearn import preprocessing

def get_embeddings(ip):
  s = model.get_ip_embeddings(ip)
  return
preprocessing.normalize(s.cpu().detach().numpy(),
norm='l2')
entities = [x[0] for x in ns_table]
vectors_of_embeddings =
np.array([get_embeddings(x) for x in
entities]).reshape(len(entities), 128)
print(vectors_of_embeddings.shape)
print('Example of normalized embeddings:')
print(get_embeddings(entities[7]))
for perplexity_ in [25,50,75,100,200]:
  tsne_x = TSNE(n_components=2,
perplexity=perplexity_).fit_transform(vectors_of_emb
eddings)
  plt.figure(figsize=(10,10))
  plt.scatter(tsne_x[:,0], tsne_x[:,1])
  plt.title(f't-SNE plot with perplexity of
{perplexity_}')
  plt.show()
```

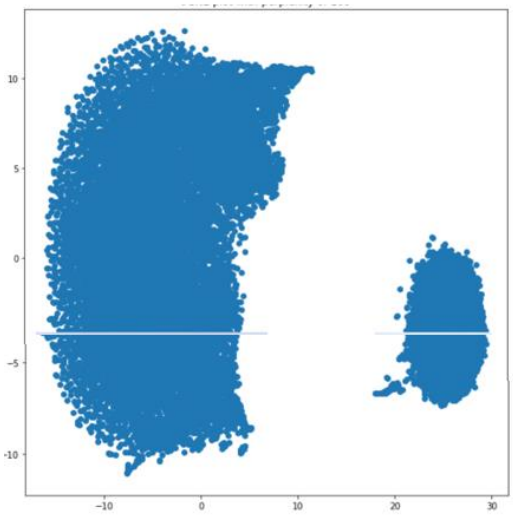
OUTPUT:

```
(20472, 128)
Example of normalized embeddings:
[[ 0.13331725 -0.04424932 -0.05084587  0.05598853 -0.07386814 -0.01834406
   0.00032433  0.11877318 -0.00772842  0.0104785   0.02279551 -0.11802933
  -0.10087359 -0.10665812  0.02402473 -0.07408499  0.10316958 -0.15156011
   0.092866    0.03598131  0.14407785 -0.03101496  0.13439545  0.09424034
   0.11172264 -0.09976584  0.05727379 -0.03441495  0.05452615 -0.10619031
   0.15073523  0.05156993 -0.04899276  0.00111436  0.00363722  0.11491631
  -0.10068344 -0.1436286   0.12844652  0.03747017 -0.01469122  0.08308484
   0.05469527 -0.037278   -0.02932701  0.13444008  0.0589963  -0.12135829
   0.01987013 -0.02921149 -0.06163943 -0.09734007  0.10700258 -0.08055906
   0.14550829  0.05647655 -0.09800148 -0.05283208  0.06877644 -0.08278935
   0.09752644 -0.08756823  0.13855198  0.05669747 -0.05449319 -0.08787786
   0.14284287  0.14113307 -0.14381164  0.05798692 -0.08452989 -0.00755227
   0.11674514  0.12715524 -0.03851837  0.01390668  0.02527453 -0.03840376
   0.06303018  0.09937029  0.10493404  0.02583331 -0.08086807  0.01804575
   0.11840998 -0.04644617 -0.00600361 -0.1349489   0.02522791 -0.1018692
   0.10539316 -0.09836396  0.08073705  0.09759463  0.05748454  0.10983288
   0.01422548 -0.13296157  0.13062069  0.05371841 -0.15207921  0.07815994
  -0.05125256  0.05178965  0.15067126 -0.01867464 -0.07181752 -0.09678625
   0.0308377  -0.12879649  0.09505253  0.06198576  0.1245612  -0.03206175
  -0.03025612 -0.1173816   0.02932527 -0.09626883 -0.11714268 -0.11545634
   0.00223438  0.08426429  0.09453706  0.04174245  0.06236649  0.13340071
```

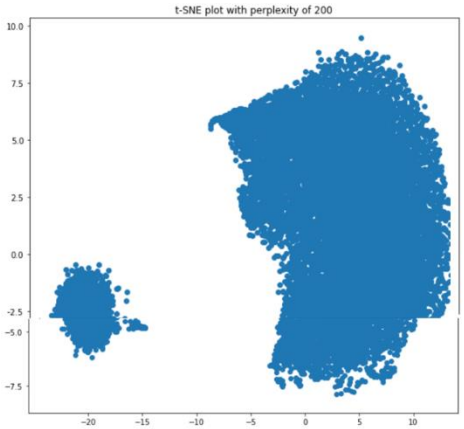
Below img is infected and normal host



Img 1.1

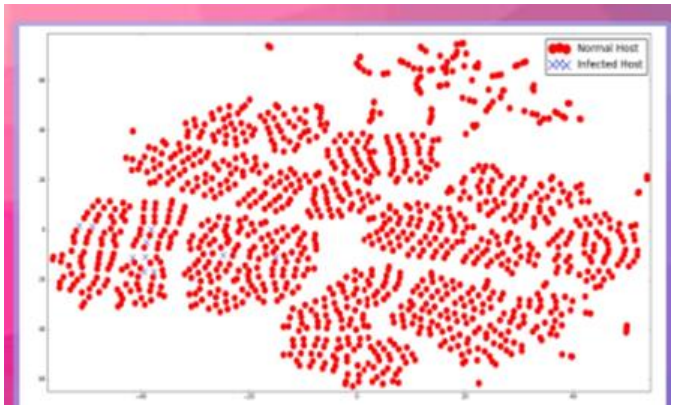


Img 2.2

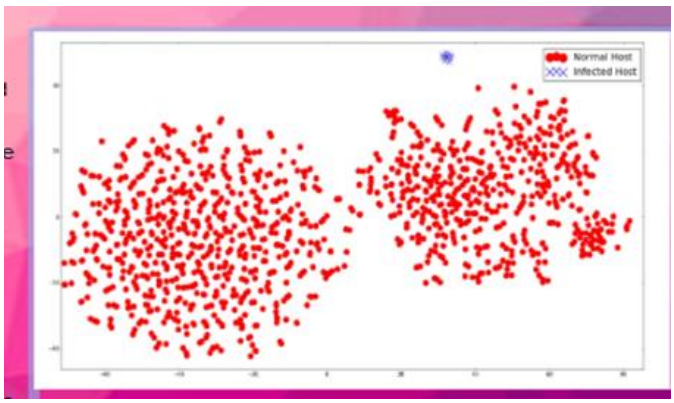


Below img is infected and normal host

Img 1.1



Img 2.2



ADVANTAGES

1. IP2Vec not only learns vector representations for IP Addresses but also for Ports and Protocols, allowing the calculation of similarities between different categorical features within a flow.
2. The model can be customized to include different context features based on the desired target. For example, including features like Bytes for traffic volume estimation.
3. IP2Vec transforms IP Addresses into continuous vectors, enabling the use of these vectors as input for various data mining methods and visualization techniques.

DISADVANTAGES

1. One primary challenge is that the behavior of IP Addresses can change over time, which may pose difficulties in classification settings. For instance, if a host changes from normal to infected, the vector representation may not immediately reflect this change.
2. IP Addresses not present in the training dataset pose a challenge as there are no pre-learned vector representations for them. Solutions include updating the network with new flows or learning a default vector representation for unknown IP Addresses.

REFERENCES

[1]. Aditya Grover (SU: Stanford University)
[2]. Jure Leskovec (SU: Stanford University)106Bryan Perozzi (SBU: Stony Brook University)24
[3]. Rami Al-Rfou (SBU: Stony Brook University)
[4]. Steven Skiena (SBU: Stony Brook University)

Cite this article as :

Sai Viraj, Yashvi Shah, Swetha Sunkara, "IP2VEC Comparing Similarities between IP Addresses", International Journal of Scientific Research in Computer Science, Engineering and Information Technology (IJSRCSEIT), ISSN : 2456-3307, Volume 9, Issue 6, pp.186-195, November-December-2023.