Memory

June 7, 2023

[1]:
```
from tqdm import tqdm
import os
import pandas as pd
import numpy as np
from sklearn.svm import OneClassSVM
import plotly.graph_objects as go
from plotly.subplots import make_subplots
import plotly.express as px
from sklearn.preprocessing import RobustScaler
from collections import Counter
from matplotlib import pyplot as plt
plt.rcParams["figure.figsize"] = (10,10)


from sklearn.decomposition import PCA

from he_svm import preprocess_a_sample, he_svm, preprocess_a_sample_encrypted
import glob
import json
```

[2]:
```
healthy_csvs = ['data/TriaxalBearings/Healthy bearing data/Healthy with pulley.
 ↪csv']

LEN_SAMPLES = 500

train_samples = []
for f in healthy_csvs:
    df = pd.read_csv(f)
    df = df.iloc[:, 1:]
    dfs = df.groupby(np.arange(len(df))//LEN_SAMPLES)
    [train_samples.append(t[1]) for t in list(dfs)[:-1]]
```

[3]:
```
len(train_samples)
```

[3]: 237

[4]:
```
len(train_samples[0])
```

[4]:
```
500
```

1 Train a SVM

[5]:
```
def preprocess_a_sample(df, windows):
    final_sample = []

    for column in df.columns:
        signal = df.loc[:, column]

        signal_fft = np.abs(np.fft.rfft(signal))**2
        len_windows = int(len(signal_fft) / windows) - 1

        for i in range(windows):
            if i == windows-1:
                final_sample.append(np.mean(signal_fft[i*len_windows:]))
            else:
                final_sample.append(np.mean(signal_fft[i*len_windows:
 ↪(i+1)*len_windows]))

    return np.array(final_sample)
```

[6]:
```
windows = 8
```

[7]:
```
preprocessed_samples_nominal = np.array([preprocess_a_sample(sample, windows)␣
 ↪for sample in train_samples])

n = int(len(preprocessed_samples_nominal) * 0.8)
preprocessed_samples_train = preprocessed_samples_nominal[:n]
preprocessed_samples_test = preprocessed_samples_nominal[n:]

svm = OneClassSVM(nu=0.05, kernel='poly', gamma='scale', degree=2)
svm.fit(preprocessed_samples_train)
svm.gamma_value = 1 / ((windows*3) * preprocessed_samples_train.var())  # to␣
 ↪put gamma value in svm
```

2 Memory occupation

[8]:
```
TRANSFER_SPEED = (1 * 1000 * 1000 * 1000) / 8  # 1 Gbit/s
```

[9]:
```
from linetimer import CodeTimer
import tenseal as ts
np.set_printoptions(precision=3, suppress=True)

poly_modulus_degree=2**14
coeff_mod_bit_sizes=[60] + [40]*7 + [60]
```

```
# Setup TenSEAL context
context = ts.context(
            ts.SCHEME_TYPE.CKKS,
            poly_modulus_degree=poly_modulus_degree,
            coeff_mod_bit_sizes=coeff_mod_bit_sizes
          )
context.generate_galois_keys()
context.global_scale = 2**40

sk = context.secret_key()

context.make_context_public()

with open('context', 'wb') as f:
    f.write(context.serialize(save_public_key=False))

file_stats = os.stat('context')

print(f'Context size in MegaBytes is {file_stats.st_size / (1024 * 1024)}')
print(f'Transfer time: {file_stats.st_size / TRANSFER_SPEED}')

os.remove('context')
```

```
Context size in MegaBytes is 394.594425201416
Transfer time: 3.310097952
```

[10]:
```
%load_ext memory_profiler
```

[11]:
```
def fun(sample, context, windows, svm):
    x_enc_preprocessed = preprocess_a_sample_encrypted(sample, context,␣
 ↪windows, None)
    x_enc_predicted = he_svm(x_enc_preprocessed, svm, windows)
    return x_enc_predicted
```

[12]:
```
for f in ['data/TriaxalBearings/1.3mm-bearing-faults/1.3outer-200watt.csv']:
    df = pd.read_csv(f)
    df = df.iloc[:, 1:]
    dfs = df.groupby(np.arange(len(df))//LEN_SAMPLES)
    anomalous_samples = [t[1] for t in list(dfs)[:-1]]

    for sample in anomalous_samples[:]:
        print(sample)
        print(f"Sample length: 3 * {len(sample)}")

        df = sample
```

```
        X = df.loc[:, ' X-axis']
        Y = df.loc[:, ' Y-axis']
        Z = df.loc[:, ' Z-axis']

        with CodeTimer('Encryption'):
            enc_X = ts.ckks_vector(context, X)
            enc_Y = ts.ckks_vector(context, Y)
            enc_Z = ts.ckks_vector(context, Z)

        encrypted_sample = {'X': str(enc_X.serialize()), 'Y': str(enc_Y.
↪serialize()), 'Z': str(enc_Z.serialize())}

        with open('sample', 'w') as f:
            json.dump(encrypted_sample, f)


        file_stats = os.stat('sample')

        print(f'A single sample size in MegaBytes is {file_stats.st_size /␣
↪(1024 * 1024)}')
        print(f'Transfer time: {file_stats.st_size / TRANSFER_SPEED}')

        os.remove('sample')
        break
```

```
      X-axis   Y-axis   Z-axis
0    -0.1350   2.0480   1.3973
1    -0.1802   2.0480   1.0481
2    -0.6722   1.7615   0.8155
3    -0.1718   1.8442   0.7816
4    -0.5349   1.3910   0.5537
..       …       …       …
495  -1.1491  -0.3261   0.8738
496   0.1940   0.5144   0.9226
497  -0.5761   0.9639   1.1901
498  -0.5527   1.0113   0.7272
499  -0.8701   0.2551   0.8635

[500 rows x 3 columns]
Sample length: 3 * 500
Code block 'Encryption' took: 50.99307 ms
A single sample size in MegaBytes is 16.929019927978516
Transfer time: 0.142010912
```

[13]:
```
# Importing the library
import psutil
```

```
print('RAM Used (GB):', psutil.virtual_memory()[3]/1000000000)
```

```
RAM Used (GB): 12.663652352
```

[14]:
```
%memit res=fun(sample, context, windows, svm)
```

```
peak memory: 2767.68 MiB, increment: 1478.13 MiB
```

[15]:
```
print('RAM Used (GB):', psutil.virtual_memory()[3]/1000000000)
```

```
RAM Used (GB): 13.940957184
```

[16]:
```
res
```

[16]:
```
array([<tenseal.tensors.ckksvector.CKKSVector object at 0x7fa6347bebb0>],
      dtype=object)
```

[17]:
```
with open('res', 'w') as f:
    encrypted_result = {'X': str(res[0].serialize())}
    json.dump(encrypted_result, f)


file_stats = os.stat('res')

print(f'A single result in MegaBytes is {file_stats.st_size / (1024 * 1024)}')
print(f'Transfer time: {file_stats.st_size / TRANSFER_SPEED}')
```

```
A single result in MegaBytes is 1.5826778411865234
Transfer time: 0.013276464
```

[18]:
```
os.remove('res')
```

[ ]:

[ ]: