random-D-safe-run

August 31, 2022

1 Test Runs of SAFE With Random `[-D, D]` Interval

1.1 Introduction

This notebook showcases an implementation of the proposed protocol for randomizing the D value used in each round of a trend detection protocol.

This protocol is described in Section 4.0.4 of the report.

1.1.1 Testing The Shared PRNG

Assuming that users use the same PRNG (the one in `random`) and a shared seed, they can generate a shared random `D` value for each round of the SAFE protocol.

Below we use an adapted PRNG from Python's documentation to generate the random floats. See this script for details.

Let's see it generate a few float values.

[ ]:
```
from secure_SAFE_utils import SharedPRNG
from itertools import repeat

shared_secret_seed = 1153

prng = SharedPRNG(shared_secret_seed)

for _ in repeat(None, 15):
    print(prng.random())
```

```
11613.49268699761
245416.82755214526
14548830.783622317
3821455.7910910887
49780359.35642068
471097.0168719898
6434314.925774718
2836508.010135213
52878698.22020412
88718462.36848637
18970195.149385847
115456656.76885584
```

```
50972.905916520736
1021033.1478310596
1748152.7667528219
```

Using The PRNG Within SAFE Let's first see what the default value of D is

[ ]:
```
from MoodAppUser import MoodAppUser

MoodAppUser.set_D_value()
print(MoodAppUser.D_Value)
```

```
100.0
```

Let's now set it randomly using the `shared_secret_seed` from above

[ ]:
```
MoodAppUser.set_D_value(value_is_random=True, shared_seed=shared_secret_seed)
print(MoodAppUser.D_Value)
```

```
11613.49268699761
```

Notice how this value is the *same* as the first value the other instance of the `SharedPRNG` generated above!

This is because they use the same seed.

If we call it again:

[ ]:
```
MoodAppUser.set_D_value(value_is_random=True, shared_seed=shared_secret_seed)
print(MoodAppUser.D_Value)
```

```
245416.82755214526
```

We get the second value.

Trial Run With Randomized D Value Let's try replicate the result we had in this notebook with random D values.

The code is essentially the same, except we will set the D value randomly each time.

[ ]:
```
import random as r
import numpy as np
from pprint import pprint
from collections import Counter

r.seed(123) # as in the use-case application

round_and_D_values = {}

POSSIBLE_RESPONSES = {
    0: "I'm feeling joyful!",
    1: "I'm feeling angry",
```

```
    2: "I'm feeling disgusted",
    3: "I'm feeling fearful",
    4: "I'm feeling sad...",
    5: "I'm feeling surprised!",
    6: "I'm feeling neutral",
}
KEYWORDS = [0, 1, 2, 3, 4, 5, 6] # corresponding to our 7-emotion taxonomy

NO_OF_DAYS_TRACKED = 21
NO_OF_USERS = 10
```

[ ]:
```
# DOCUMENT SET AND USER GENERATION

random_document_sets = [r.choices(KEYWORDS, k = NO_OF_DAYS_TRACKED) for _ in␣
 ↪repeat(None,   NO_OF_USERS)]

users = [MoodAppUser(i, document_set) for i, document_set in␣
 ↪enumerate(random_document_sets)]

# PRIORS

no_of_keywords = len(KEYWORDS)
priors_for_keywords = np.round([(1 / no_of_keywords) for _ in repeat(None,␣
 ↪no_of_keywords)], 4)

# TARGET POSTERIOR

target_aggregation = np.sum(list([user.feature_vector for user in users]), axis␣
 ↪= 0)

target_posterior = np.multiply(target_aggregation, priors_for_keywords)

print("Target Posterior For Initial Round:\n")
pprint(target_posterior)
```

```
Target Posterior For Initial Round:

array([0.26539388, 0.27219592, 0.21093469, 0.25181838, 0.13605509,
       0.18372653, 0.10886122])
```

[ ]:
```
# SHARE_GENERATION

MoodAppUser.set_D_value(value_is_random=True, shared_seed=shared_secret_seed)

for i in range(NO_OF_USERS):
    users[i].generate_shares_to_send(NO_OF_USERS, no_of_keywords)
    users[i].calculate_Nth_share()
```

```
# SANITY CHECK
assert np.all(np.subtract(users[0].feature_vector, np.sum(list(users[0].
 ↪shares_to_send.values()), axis = 0))  == users[0].Nth_share)

print(MoodAppUser.D_Value)
```

```
454650.9619881974
```

[ ]:
```
# SHARE_DISTRIBUTION

for user in range(NO_OF_USERS):
    other_user_ids = [id for id in range(NO_OF_USERS) if id != user]
    for other_user in other_user_ids:
        share_to_send = users[user].get_share_for_user(other_user)
        users[other_user].receive_share(share_to_send)
    # NOTE - as noted below, we can generate the obfuscated vector here if we␣
 ↪want to be more efficient.

# SANITY CHECK
assert np.all(users[0].shares_to_send[1] == users[1].shares_received[0])
```

[ ]:
```
print(users[0].shares_to_send[1])
```

```
[-407727.4204 -101461.5862 -356932.0088   66861.2633  421120.0703
 -159405.0515  133518.1092]
```

[ ]:
```
# OBFUSCATED FEATURE VECTOR CALCULATION AND SECURE POSTERIOR CALCULATION

for user in range(NO_OF_USERS):
    users[user].generate_obfuscated_feature_v()

masked_vector_result = np.sum(list([user.obfuscated_feature_v for user in␣
 ↪users]), axis = 0)

# rounding because precision beyond 4.d.p is unlikely to be consequential :
target_aggregation = np.round(target_aggregation, 4)
masked_vector_result = np.round(masked_vector_result, 4)
assert np.array_equal(target_aggregation, masked_vector_result)

print(f"Sum of raw vectors: \n\n {target_aggregation}")
print(f"Sum of masked vectors: \n\n {masked_vector_result}\n")
```

```
Sum of raw vectors:

 [1.8572 1.9048 1.4761 1.7622 0.9521 1.2857 0.7618]
Sum of masked vectors:
```

```
[1.8572 1.9048 1.4761 1.7622 0.9521 1.2857 0.7618]
```

Success!