safe-applied-to-use-case

August 31, 2022

0.1 SAFE Protocol NLP Experiments

This notebook builds on top of the other notebook where I illustrated the bare SAFE Protocol itself.

This notebook specifically illustrates the application of SAFE to trend detection in document interactions.

0.1.1 Trending Mood Detection For Survey Responses

As described in the report, users respond to a prompt question like "*How do you feel today?*" every day for a month, and an analysis on what responses are trending is carried out every 2 weeks.

When submitting a response to this question, the user picks from a pre-defined list of responses that contains strings like the following

[ ]:
```
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"
}
```

These responses are each encoded as numbers, such that what is actually stored locally on user devices as their responses are integers from `0` to `6` corresponding to each of these answers (`0` corresponds to `I'm feeling joyful!"`, `1` corresponds to `"I'm feeling angry"`, and so on…)

The target computation of the Bayesian approach to secure trend detection is:

$$p(\mathrm{t} = t|D) = \frac{p(D|\mathrm{t} = t|)p(\mathrm{t} = t|)}{p(D)}$$

where t is some term in the keyword set V.

The **keyword set** V for our experimental setting consists of the responses themselves, since what we what to determine is a probability distribution for the various moods given the overall document set contributed by users.

As discussed in their paper, calculating the marginal probablity $p(D)$ would not be privacy-preserving, as this would require access to the dataset D. The authors advise avoiding calculating it by treating it as constant. Then, the posterior likelihood $p(\mathrm{t} = t|D)$ is proportional to $p(D|\mathrm{t} = t|)p(\mathrm{t} = t|)$. Since we are only trying to find trending keywords (in terms of rankings), we do not need the exact value of the posterior and we can simply consider this equation:

$$p(\mathrm{t} = t|D) \propto p(D|\mathrm{t} = t|)p(\mathrm{t} = t|)$$

Defining Our Terms In particular, we want to define the terms on the RHS : the likelihood, $p(D|\mathrm{t} = t|)$, and the prior $p(\mathrm{t} = t|)$.

Defining The Prior For the first run of the protocol, the priors will be defined uniformly– i.e., $\frac{1}{d}$ where d is the number of keywords in V.

For each subsequent run, the priors will be defined according to last round's posterior probability.

Defining the Likelihood Vector Let's suppose there are N users who respond to the prompt question every day for y days. (We assume that users respond every day with some response)

This will mean that each user has a list of y integer responses that represents the option they picked for each day.

This will constitute their **document set** D_i which we wish remains private, where *each integer* (representing a response) is a **document**.

Each document's **primary keyword set** is simply a single-element set with the integer that represents the response they picked. (e.g. the primary keyword set of the document `1` just is `1`).

As described in the paper, what ends up being the user's raw feature vector is a vector of likelihoods,

$$L_i = (p(D_i|\mathrm{t} = t_1), \cdots, p(D_i|\mathrm{t} = t_d))$$

where d is the number of words in the vocabulary V, and each $p(D_i|\mathrm{t} = t)$ is calculated by the fraction of the given keyword in the document set of the user.

For instance, say that a user has the following document set of just 7 responses (for simplicity):

```
u_1 = [0, 1, 2, ,3, 1, 4, 5]
```

They have 7 documents in total. Their feature vector of likelihoods is

```
V = [0, 1, 2, 3, 4, 5, 6]

u_1 = [0, 1, 2, 3, 1, 4, 5]

u_1_feature_vector = [1/7, 2/7, 1/7, 1/7, 1/7,  1/7, 0]
```

Computing The Target Function Treating each user's document set D_i as a random variable for the subset of the overall document set D (i.e., the set of all user responses), then we can treat the aggregation of these instances (i.e., the likelihood of the whole document set) can be represented as the mean. (ibid., 5)

So, the aggregator wants to compute the following for each keyword.

$$p(D|\mathrm{t} = t_1) = \sum_{i=1}^{N} p(D|\mathrm{t} = t_1|)$$

Aggregating the feature vectors for all users can be done securely using the SAFE Protocol.

0.1.2 Test Run : With Raw SAFE version

Generating The Raw Feature Vectors Let's say there are 10 users who respond to the prompt question every day for 21 days, and as analysts we want to consider what responses are trending given this period.

First, let's define our constants and import what we need

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

r.seed(123) # for reproducibility

# as from above:
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
```

Next, let's create the 10 users and give them document sets that consists of a random selection of responses.

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

The Distribution Of Randomly Generated Document Sets **Note**: according to the documentation on `random.choices()` (which is the method we invoked to form our random document sets)

> If neither weights nor cum_weights are specified, selections are made with equal probability.

Since we did not specify any arguments for the `weights` nor `cum_weights` parameters, we should except out document sets to have a roughly **flat distribution**.

This is just worth bearing in mind when calculate the posterior. It is not a problem since we are merely illustrating the protocol's application

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

     print("Here are some users and their random document sets:\n")
     for i in range(3):
         print(f"User {users[i].id} and their random document set:\n{users[i].
      ↪document_set}\n")
```

```
Here are some users and their random document sets:

User 0 and their random document set:
[0, 0, 2, 0, 6, 0, 3, 2, 5, 1, 2, 2, 1, 0, 3, 0, 4, 0, 2, 3, 6]

User 1 and their random document set:
[0, 0, 5, 0, 6, 4, 1, 5, 5, 2, 5, 1, 4, 3, 5, 2, 2, 5, 3, 4, 4]

User 2 and their random document set:
[4, 6, 3, 4, 2, 0, 5, 1, 5, 6, 4, 1, 2, 2, 0, 3, 4, 2, 1, 0, 4]
```

We can also take a look at their raw feature vectors.

```
[ ]: print("Here are some users and their raw feature vectors:\n")
     for i in range(3):
         print(f"User {users[i].id} and their raw feature vector:\n{users[i].
      ↪feature_vector} of length {len(users[i].feature_vector)}\n")
```

```
Here are some users and their raw feature vectors:

User 0 and their raw feature vector:
[0.3333 0.2381 0.0952 0.1429 0.0476 0.0952 0.0476] of length 7

User 1 and their raw feature vector:
[0.1429 0.2857 0.0476 0.1905 0.0952 0.1429 0.0952] of length 7

User 2 and their raw feature vector:
[0.2381 0.0952 0.0952 0.1905 0.1429 0.0952 0.1429] of length 7
```

Performing Secure Aggregation of Feature Vectors Now that we have users and their raw feature vectors, let's compute the target function.

Since this is the first run of the protocol, we begin with uniform priors.

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

Next, let's calculate using the raw feature vectors what the target value is:

```
[ ]: 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])
```

We want to get the same result running SAFE. Let's see if that happens:

Share Generation First, let's have each user generate the N shares.

Users first want to generate $N-1$ shares to send to other users.

They will also generate their N^{th} share (to keep) using these.

First, we set the D value for the round, and then users will draw shares from these.

```
[ ]: MoodAppUser.set_D_value()

     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)
```

Share Distribution Now we wish to distribute the shares among the users.

User 0 will send the shares they generated for User 1 and User 2 to each respectively, and so on.

```
[ ]: 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])
```

0.1.3 Calculate Obfuscated Feature Vector

Having distributed the shares, each user can now calculate their obfuscated feature vector.

They do this by adding the sum of received shares to their Nth share.

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

     # NOTE - share distribution and obfuscated vector generation can be done in the␣
      ↪same loop
```

Secure Aggregation & Posterior Calculation Let's now see if aggregating the obfuscated feature vectors results in the same result as aggregating the raw feature vectors. If so, our protocol has been implemented correctly.

```
[ ]: 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]
```

Great! And now we calculate the posterior

```
[ ]: masked_posterior = np.multiply(masked_vector_result, priors_for_keywords)

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

     print(f"Raw Posterior value: \n\n {target_posterior}")
     print(f"Masked Posterior value: \n\n {masked_posterior}\n")
```

```
Raw Posterior value:

 [0.2654 0.2722 0.2109 0.2518 0.1361 0.1837 0.1089]
Masked Posterior value:

 [0.2654 0.2722 0.2109 0.2518 0.1361 0.1837 0.1089]
```

Success!

0.1.4 Detecting Trends In Interactions Over Some Static List Of Resources

A neat feature of the above protocol is that it can not only be used to detect trends in user responses to the prompt question, but also to detect trends for **interactions with a static list of resources**.

We simply map the list of resources accessed by a user to numbers just as we did with the possible questions.

For instance, say we had a static list of 3 resources, 3 active users and collected their inputs for 5 days. Then for a given user we'd have something like:

```
[ ]: # as from above:
     RESOURCES = [0, 1, 2]

     example_document_set = {
         0: [0, 1, 2],
         1: [0, 1, 2],
         2: [0, 1],
         3: [],
         4: [2]
     }

     KEYWORDS = [0, 1, 2] # corresponding to our keywords.
```

The protocol can then be run in a similar fashion, so long as we **flatten the document set into a list representation**. So, we'd be able to compute a probability distribution (posterior) for which resources are 'trending' given the overall document set.

0.1.5 Drawbacks of Approach:

One issue that should be addressed is how to account for the fact that users may not use the app every day. How should we encode and handle non-responses?

```
[ ]: # as from above:
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",
    -1: None # no response for that day.
}

u_1 = MoodAppUser(0, [0, 1, 2, 3, 1, 4, 5, -1]) # 8 days of collected dataw
```

We could just include `-1` as a keyword in the response. However, as with any dataset with missing values, the utility of our analysis can be adversely impacted if there are too many non-responses