Modeling Requirements for Simulating Covert Social Groups

Carl A. B. Pearson[1,*], Edo Airoldi[2], Edward Kao[2], Burton Singer[1],
1 Emerging Pathogens Institute, University of Florida, Gainesville, FL, USA
2 Statistics, Harvard University, Cambridge, MA, USA
∗ **E-mail: cap10 {at} ufl {dot} edu**

Abstract

Covert social groups are difficult to observe directly, by definition: members either actively thwart discovery or leverage passive features of their surrounding population to avoid detection. Moreover, increasing active observation would be expected to perturb these social systems. However, these groups never have perfect cover. Their secrecy may make them conspicuously absent from the everyday activity of the background population. Finally there are potential mechanical characterizations – their composition, goals, tactics, etc. – that might predict their actions.

The structure of covert networks is highly variable from one context to another, and they change composition and interaction patterns over time. With any covert network, there are strong constraints on passively and actively obtainable direct empirical data, overwhelming amounts of indirect data, and competing mechanical explanations for what is observed. Studying the detailed history of any one network, or set of them, will not necessarily provide insight about how to effectively detect such networks in the future. This situation recommends simulation of a variety of scenarios with accompanying detection strategies. Because of the censoring issues in the data, simulation requires particular attention to characterizing uncertainty and avoiding overfitting.

Herein, we discuss these issues in terms of network models. Specifically, we review (i) incorporating observation uncertainty, (ii) the need for both back- and foreground populations, (iii) avoiding over-fitting, and (iv) some philosophy relative to implementation.

As part of that philosophical discussion, we walkthrough the exercise of procedurally generating back- and foreground populations, simulating communication among those individuals, and filtering those communications via an observation model. We then address the issues of (i) fitting that model against data, and (ii) analyzing performance of the opposing sides (e.g. Receiver Operator Characteristic). Finally, we also demonstrate a small model extensionand show how it affects the (unextended) detection performance.

Introduction

For investigators ranging from anthropologists to law enforcement, the desire to find, study, and modify the behavior groups operating at the edge of the observable is a principle concern. In the meta-theatre of the ironic, various media outlets have responded to public interest by applying this lens to the activities of intelligence organizations world-wide – themselves desperate to undertake this very task against other clandestine actors. Of course the revealation of these activities has broadly sparked the symmetric concern: being able to operate clandestinely in an age of ubiquitous monitoring. Criminal organizations have long appreciated the value of operating in secret, as have groups subject to State-sponsored abuse or industrial espionage, and time will tell whether or not this spark catches fire in general behavior.

The underlying drive for these opposed efforts is the implacably expanding byte trail. Once, the recorded information of an entire life might have amounted only to a few bytes – e.g., parish records on births and deaths, perhaps including morbidity. Now information era tools produce a near endless supply of 1s and 0s. In some cases, this production is a side effect of organized commercial instinct – Google monetizing search through targeted advertisement, for example. In other cases, individuals are voluntarily advancing their own brand – Instagram "selfies", e.g. However, in many cases this bitwise production is an engineering consequence of the associated technology. E.g., cellular phones transmit constant location data, financial systems are increasing just digital ledgers with almost no physical exchange, and of course any use of the internet produces veritable bit contrails. This rate of production far exceeds the personal processing capability of any practically-sized team of analysts.

Hence, these teams employ computer-based, heuristic filtration to decide which data to record, to review, and to obtain. We avoid saying "algorithmic" at this point: "algorithmic" implies strong, logically inevitable conclusions from inputs. Like "algorithmic" bubble prediction, we side more with Fama than Shiller.

Many such filters are based on social network descriptions. These social networks can be a compact way to represent measured relations in a population. However, these approaches all boil down to measuring properties of mathematical representation, with those properties having a tenuous connection to a mechanistic explanation of those phenomena.

Given the real uncertainty, what these filters call for is testing and validation, but those present their own difficulties. Calling field testing "problematic" seems like a gross understatement; reference "truth" ranges from incomplete to deceptive, and experiments could have dangerous side effects. Even making use of intensely studied historical events is problematic: these offer no way to consider evolutionary behavior and technological innovation (without an additional layer of abstraction that would create an even more complex empirical challenge), even assuming the historical data are more than victor's embellishment.

Generating synthetic data seems like an obvious alternative. It allows for comparison across both detection and masking strategies, consideration of multiple background contexts, forecasting of risks and tradeoffs in a way that allows uncertainties, and in general providing a framework for imaginative assessment. Like all such flexible tools producing quantitative results, it has the subtle downside of the simulator's biases being validated by numerical gospel; if one believes a particular strategy is effective – perhaps even with reasonable agreement for a particular time and situation – there would be a natural tendency to "adjust" scenarios until they indicated the success of strategy. There have been other attempts at such an approach [2].

In the following sections, we lay out the uses and abuses of such a framework. What makes for useful synthetic data sets? What are the appropriate measures for detection strategies on them? We motivate that discussion by inspiration from an over-simplified, network-based model of terrorism – a sub-group of the Salafi jihad networks as described by Sageman *et al.* [**?**] – and community organization and communication. Whether or not that work is an accurate description is not of particular concern. Their qualitative properties are enough of a defensible testbed for modeling the communication patterns of one covert group. We will point out where assumptions can be modified to identify different kinds of groups against a background population, since the behavior and structure of both the background and covert organizations are constantly evolving.

Overview of the Simulation Problem

Detecting a covert group is fundamentally about distinguishing the trace observable activity of that group from the activity of the background, bootstrapping that into guesses about the structure of the group, focusing the observation and distinguishing process based on that structure, and so on iteratively. The purpose of simulation is ultimately to measure performance in the observer-covert group competition.

Given that these aspects must be expressed in the model (with varying degrees of detail), and that this model is used by a team or communicated between researchers, the language for that expression must be both powerful (to capture complexity where necessary), but also comprehensible to communicate what in fact is the meaning of the universe in this model. Because these simulations are fundamentally about individuals and their behavior, we largely frame our discussion in terms of individuals, their internal state, and their interaction with others.

We can of course think in different terms. For example, we are considering a network model of the population and covert group embedded within it, with edges representing relationships between people. We might think of those relationships as being external state and correspondingly represent those relationships as external state in the simulation. E.g., we might represent the population as a graph rather than a bag of individuals, and in the simulation the edges belong to the graph object rather than individuals. As long as we recognize the translation between these perspectives, the choice boils down to what is most pragmatic relative to the science and simulation. If most of the questions answered by the simulation relate to graph measures – e.g., calculating various centrality measures – then the simulation-as-graph rather than simulation-as-individuals probably makes more sense.

Modeling the Covert Groups

Relative to the covert group, the simulation must model how the group members act and interact. Typically, that further entails modeling a few moving pieces: (i) the members internal state (including what other members they know and communicate with), (ii) how that internal state evolves relative to external forces or states, and (iii) what actions those members undertake based on their internal state (possibly in response to some specific external event).

Modeling the Background

Like the covert group, the background population is a (larger) group of individuals that act and interact. They have the same basic modeling requirements as the covert group, but they behave differently. This difference may be as simple as being distinct relative to observation process – e.g., their communications are more likely to be recorded – or it may be more noticeable like having fundamentally more diverse contacts, taking different kinds of actions, etc.

An alternative representation for the background would be to model it as a continuous entity instead of individuals in a network. This could be used to reduce the simulation size. However, this model would still need to create activity data that was consistent (in format) to that created by the covert group, because of the simulation requirements for observation.

Modeling Activity

Individuals create modeled activity based on their internal state. These actions correspond to (i) the observable events of specific interest (e.g., calls, financial transactions), (ii) generally observable events (e.g., travel, work, religious observations), and (iii) interactions that, while not externally observable, play a role in the evolution of internal state in other individuals either directly or by modifying the state of the world. These actions should occur at specific times in the simulation, at whatever resolution is appropriate to the model.

The covert group and background population should be different in their activity; either performing different activities or undertaking them at different rates or on different schedules.

The Observation Model

Activity is understood by the observing entities by an imperfect process. Depending on the situation modeled, that process might simply miss events. Alternatively, the error could be misunderstanding the type or content of the activity, or between what individuals and interaction is occurring. In general, the observing entities should be able to modify what is observed at what rates. Also, the observation process may itself alter the state of the world – i.e., the covert group may have the opportunity to respond to the observation process.

A Language Matching the Simulation Requirements: Scala

Scala [4] is an mixin-inheritance [1], object-oriented, functional language, which also includes an message-passing based model for parallel computing[1]. Indeed, Scala is used under the hood for the network simulation framework NetLogo.

Why do these traits make Scala particularly effective for the simulation requirements described above?

First, the covert group members and the background population share behavior (in the real and simulation sense), but are not necessarily simple extensions, which makes a mixin inheritance object system a natural fit for the model subjects.

Second, many interactions in an organization can be thought of as management directing labor to complete a task; in a functional paradigm, functions are themselves treated as objects. This translates natural to sending a function object from one simulated individual (in a leadership position) to another (in a subordinate position). Loosely adopting some of the other functional paradigms, as Scala does – emphasis on immutability, principally – provides practical advantages to concurrent programming, which is necessary for large scale simulations. Additionally, the functional emphasis bakes in other programming constructs to do with collection processing (e.g., iteration, transformation, filtering) that are less convenient in earlier languages.

Finally, the message-passing framework is an exact analogy to an individual / agent / actor - based model. Having this baked into the language makes it a feature more like control branches (e.g., `if`, `else`, `for`) or collections, than a more complicated syntax available only to advanced programmers (and often only via library support) rather than more phenomena- and model-focused scientists.

Practical Application

The practical application of these ideas is – for a simple simulation – as straightforward as stated.

Using the Scala actor framework, we simply define how each agent in the system responds to events, including the observing entities, and how they generate new events. We accomplish the first by appropriately defining `r`eceive and the handling of different `c`ases of messages. We accomplish the second by having actors know about other actors, and having their internal state determine if they send events to these actors at any particular time.

In considering the salafi network (see fig. 1), we will observe a highly constrained set of activity: only communication, with all of that communication having been reduced to `Good` or `Bad`. The communication connections between individuals will be static, therefore we may set the social network at the outset of the simulation.

In the code, we could define different kinds of actors – the covert leader, covert followers, background individuals – as different classes of simulation object. However, we instead take advantage of the context switching in the newer Scala actor framework. By approaching the simulation this way, we lay a good foundation for observing the dynamics of individuals – transitioning from background

[1]The actor model is transitioning from the initial implementation to that developed by the Akka project [3]. We base our discussion on the Akka model. The most important difference is the notion of contexts and context switching: with context switching, an actor alters how it responds to events. This is a useful abstraction for representing evolving individual state.

Figure 1. the salafi network

to covert member to covert leader – as well as the dynamics of relationship and group formation and dissolution.

One syntactical aside for the code the below sections: the Scala collections (like `Set`) define methods for adding and removing elements in terms of the natural mathematical operators (`+`, `-`) and for similar operations with groups of elements (`++`, `--`). Similarly, the names for message senders (`sender`), changing context (`context become ...`), et cetera are all quite clear. Given the informative, mixed-natural-and-mathematical-like syntax, we prefer actual code snippets to pseudo-code to illustrate simulation steps.

The Covert Group

First, we define the covert group `Receive` context:

```
def plotter(
  // a plotter has superiors, collaborators, and subordinates
  // these groups are initially empty
  superiors:Set[Plotter] = Set.empty,
  collaborators:Set[Plotter] = Set.empty,
  subordinates:Set[Plotter] = Set.empty) : Receive = {
    // ... to-be-defined behavior
    case _ => // ignore unmatched messages
}
```

We need to define how they adopt and evolve this context. Since we are not studying the organizational dynamics with this simulation, we can be phenomenologically loose for this part, though this approach means we can add monitoring that activity as part of future detection scenarios with minimal coding. The simulation runner will send a `Radicalize` message to the initial cabal, then a `Collaboration` message to the same group:

```
case class Radicalize(id:Long)
case class Collaboration(id:Long, group:Set[Plotter])

def receive = {
  case Radicalize(_) =>
    context become plotter()
  case _ => // ignore other messages
}

def plotter(...) => {
  case Collaboration(_, group) =>
    context become plotter(collaborators = collaborators ++ (group - self))
  // ...
}
```

These leaders then need to recruit subordinates. In this particular simulation, we are using exactly the network from Sageman et al., so we will have the simulation runner inform each plotter whom they are recruiting with a `Subordinates` message. The leads will then send `Recruit` messages to those individuals and they will adopt the plotter context.

```
case class Subordinates(id:Long, recruits:Set[Plotter])
case class Recruit(id:Long)
case class RecruitAck(id:Long) // acknowledge recruitment

def plotter(...) = {
  case Subordinates(_, recruits) =>
    recruits foreach { recruit =>
      recruit ! Recruit(0)
    }
  case RecruitAck(_) =>
    context become plotter(superiors, collaborators, subordinates + sender)
  // ... other cases
}

def receive = {
  case Recruit(id) =>
    context become plotter(superiors = Set(sender))
    sender ! RecruitAck(id)
  // ... other cases
}
```

At this point, we have all the necessary messages to construct the given network. Those messages, based on the labels in Sageman et al. are covered in Appendix **??**. However, we still need to model how the plotters actually plot. We will assume that all plotting (i) originates with leadership, (ii) when intermediates receive direction, they work with their `collaborators` as well as delegating work, (iii) that directions occasionally result in feedback to higher-ups, and finally (iv) all of the "work" and direction is captured as passing `Bad` messages.

The simulation runner sends `Time(t:Long)` messages to all actors in the simulation to indicate the march of time (waiting for them to complete all their business for that step and reply with `TimeAck(t)`). In general, during each time step a leader may initiate a new `Bad` message. Other members of the covert group circulate `Bad` messages subsequent to receiving a `Bad` message themselves, with different probabilities for sending messages to superiors, collaborators, and subordinates. Those probabilities do not depend on the origin of the received message (though this could be a useful mechanical tweak to the model).

Though those probabilities could be coded into the actors as part of their initialization, if they are set and adjusted by events then the actors (i) more accurrately reflect real behavior and (ii) the evolution of individuals occurs via a consistent simulation mechanism. The former provides a lever for expert insight and empirical study. The latter makes it easier to produce real-alike data – i.e., simulation data that has the form of real observations.

In our simplified example, we do not consider how "real" events would translate into adjusting a covert member's behavior. Instead, the simulation runner sends a message, giving us a wedge for later adjustment without frontloading too much model complexity. This represents the final step (in our simulation) for how the plotters evolve in state, so the following repeats the previous snippets with some condensing modifications to the code

```
object SimActor {
// the companion object, where we define common elements for SimActors
  case class Radicalize(id:Long) // become a plotter
  case class Collaboration(id:Long, group:Set[Plotter])
                               // link existing plotters
  case class Subordinates(id:Long, recruits:Set[Plotter])
                               // tell a plotter to recruit subordinates
  case class Recruit(id:Long)  // tell an actor to join plotters
  case class RecruitAck(id:Long) // acknowledge recruitment
  type Plotter = SimActor
  type Probability = Double
}

class SimActor extends Actor {

  def plotter(
    superiors:Set[Plotter] = Set.empty,
    collaborators:Set[Plotter] = Set.empty,
    subordinates:Set[Plotter] = Set.empty,
    initiate:Probability = 0.0,  // prob. of starting plot comms
    report:Probability = 0.0,    // prob. of reporting to superiors
    collaborate:Probability = 0.0, // prob. of working w/ peers
    delegate:Probability = 0.0   // prob. of delegating work
    ) = {
      def update(
        newSuperiors:Set[Plotter] = superiors,
        newCollaborators:Set[Plotter] = collaborators,
        newSubordinates:Set[Plotter] = subordinates,
        newInitiate:Probability = initiate,
        newReport:Probability = report,
        newCollaborate:Probability = collaborate,
        newDelegate:Probability = delegate
        ) = {
          context become plotter(newSuperiors, newCollaborators, newSubordinates,
            newInitiate, newReport, newCollaborate, newDelegate)
      }

      {
        case Subordinates(_, recruits) =>
          recruits foreach { recruit =>
            recruit ! Recruit(0)
          }
        case RecruitAck(_) => update(newSubordinates = subordinates + sender)
        // ... other cases
      }
  }
}
```

References

[1] Gilad Bracha and William Cook. Mixin-based inheritance. In *ACM SIGPLAN Notices*, volume 25, pages 303–311. ACM, 1990.

[2] Kathleen M Carley. Dynamic network analysis. In *Dynamic social network modeling and analysis: Workshop summary and papers*, pages 133–145. Comittee on Human Factors, National Research Council, 2003.

[3] Typesafe Inc. Akka, 2013.

[4] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. An overview of the scala programming language. Technical report, Citeseer, 2004.

A Messages to Construct Salafi Network

```
// ...TODO figure out these messages
val cabal = Set(``A'',``B'',``C'');
cabal foreach { _ ! Radicalize(...) }
cabal foreach { _ ! Collaboration(..., cabal) }
```