A Model of Outbound Client Traffic on The Tor Anonymity Network

by

Julian Applebaum

A thesis submitted to the faculty of Wesleyan University in partial fulfillment of the requirements for the Degree of Bachelor of Arts with Departmental Honors in Mathematics and Computer Science

Middletown, Connecticut

April, 2013
Acknowledgements

First, I would like to thank my advisor Norman Danner for being so actively involved in the process of performing this research. I felt truly lucky to have someone ready to put so much time and thought into guiding a student project. I would also like to thank my reading committee, Norman Danner, Emmanuel Kaparakis, and Danny Krizanc, for taking the time to evaluate this work.

To my house mates Arya, Bryce, and Sam, thank you for being so supportive and keeping life fun when things came down to the wire. To my girlfriend Rachel, your love and support throughout all of this has meant the world to me.

Finally, thank you to my parents for giving me love, encouraging my studies, and teaching me to never settle. I wouldn’t be who I am today without your guidance.
Abstract

Tor is a popular low latency anonymity system. It works by routing web traffic through a series of relays operated by volunteers across the globe. Academic research on Tor frequently involves proposing new attacks on the network, creating defenses against these attacks, and designing more efficient methods for routing traffic. For ethical and practical reasons, it is often necessary to perform this research on a simulated version of the live Tor network. ExperimenTor and Shadow are two simulation environments that realistically model many aspects of the extant network. Both of these environments, however, only offer crude models of the outgoing web traffic generated by actual Tor users. In this thesis, we seek to improve on these models by collecting traffic data from the live network, clustering it into groups with similar behavior, then training a Hidden Markov Model on each cluster.
## Contents

Chapter 1. Introduction .................................................................................................................. 1  
1. Onion Encryption ...................................................................................................................... 2  
2. Tor Cells .................................................................................................................................. 3  
3. Circuit Creation ......................................................................................................................... 4  
4. Contributions of this Thesis ....................................................................................................... 4  

Chapter 2. Existing Tor Simulations .............................................................................................. 8  
1. ExperimenTor ............................................................................................................................ 8  
2. Shadow ................................................................................................................................... 10  

Chapter 3. Hidden Markov Models .............................................................................................. 12  
1. Discrete Markov Models .......................................................................................................... 12  
2. Hidden Markov Models ........................................................................................................... 13  
3. The Forward Recursion ........................................................................................................... 13  
4. The Baum-Welch Algorithm .................................................................................................... 15  

Chapter 4. Exploratory Data Analysis Techniques ....................................................................... 17  
1. Distribution Analysis .............................................................................................................. 17  
2. Temporal Analysis .................................................................................................................... 19  

Chapter 5. Clustering .................................................................................................................. 21  
1. Overview .................................................................................................................................. 21  
2. The $k$-Means Algorithm ......................................................................................................... 22  
3. Hierarchical Clustering ............................................................................................................. 24  
4. The HMM Clustering Algorithm ............................................................................................... 26  
5. Estimating $k$ With Monte-Carlo Cross-Validation .................................................................. 30
Chapter 6. Analysis of Training Data

1. Data Collection 31
2. Distribution Analysis 32
3. Temporal Visualizations 34
4. Conclusions and Model Justification 38

Chapter 7. Implementation and Results 39

1. Implementing HMMCluster for Relay Time Series 39
2. Monte-Carlo Cross-Validation and Parameter Selection 40

Chapter 8. Conclusion 46

Bibliography 47
CHAPTER 1

Introduction

When a message is sent over the Internet, it is transmitted from one computer to another via TCP/IP. In TCP/IP, a message is sent piecemeal as a series of packets, each of which consists of a header containing routing information and a payload containing a fixed-size portion of the message. If a user Alice wants to communicate privately with a website bob.com, she may encrypt the payload of these packets to conceal their content from eavesdroppers. While doing so conceals the content of Alice’s message, it does not conceal the fact that Alice is communicating with bob.com. By viewing the routing information in Alice’s packets, an eavesdropper Eve can easily see that Alice is visiting bob.com.

To be fully protected, Alice must conceal not just her message to bob.com, but the fact that she is communicating with bob.com at all. For this reason, many anonymity systems have been developed which offer this type of protection. Anonymity system typically fall into one of two categories: low latency and high latency systems. High latency systems like Babel, Mixmaster, and Mixminion trade off latency for protection against correlation attacks, a class of attack that exploits Alices’s packet timing patterns to statistically infer her connection to bob.com. These systems use techniques such as delaying packets and sending messages out of order in order to distort Alice’s original timing patterns beyond recognition. The transmission delay introduced by these techniques makes high latency systems suitable only for non-interactive activities like sending e-mail. Low latency systems generally do not attempt to modify Alice’s timing patterns. While this leaves such systems vulnerable to correlation attacks, it also offers low enough latency to support interactive activities like web browsing, instant messaging, and video streaming.
Tor is a popular low latency system with an estimated 250,000 daily users [Bauer et al., 2011]. It achieves anonymity by concealing Alice’s message and routing information beneath at least three layers of encryption. Her packet is then sent through at least three volunteer-operated onion routers (OR), each of which “peels off” a layer of encryption. The last router recovers Alice’s message and its original destination, then forwards it to bob.com. Since Eve cannot decrypt the header of Alice’s original packet, she can no longer associate Alice with bob.com.

1. Onion Encryption

To route web traffic through the Tor network, the user connects to an onion proxy (OP) via SOCKS, a session layer protocol for communicating with proxy servers. The onion proxy is generally run by the user, and it is responsible for connecting to the network, building circuits, and transparently sending and receiving Alice’s web traffic.

At startup, the OP first downloads the consensus document, a list of all known ORs and their health statistics, from one of Tor’s directory servers, a set of trusted servers tasked with monitoring the global network. With this list, the OP may then choose a set of relays and build a circuit. All Tor circuits consist of at least three ORs: a guard relay, one or more intermediate ORs, and an exit relay. A guard relay knows Alice’s IP address by virtue of being the first OR in the circuit. As the last OR in the circuit, an exit relay sees Alice’s decrypted packets. To onion encrypt the packets, the OP must negotiate symmetric onion keys with every relay in the circuit. Let $R_1, R_2, \ldots R_n$ be these relays, and $k_1, k_2, \ldots k_n$ be the corresponding keys. Alice’s packet $p$ is encrypted at the OP as $(E_{k_1} \circ E_{k_2} \circ \ldots E_{k_n})(p)$. As it moves through the circuit, $R_i$ applies $D_{k_i}$, removing a layer of encryption. When the cell leaves the exit relay, all $n$ layers have been removed. A packet $p'$ sent by bob.com to Alice is encrypted in the reverse manner: $R_n$ encrypts with $k_n$, $R_{n-1}$ with $k_{n-1}$, and all relays through $R_1$ until the OP receives $(E_{k_1} \circ E_{k_2} \circ \ldots E_{k_n})(p')$. The OP then removes all $n$ layers of encryption and delivers the packet to Alice.
1. INTRODUCTION

<table>
<thead>
<tr>
<th>Command</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>CREATE</td>
<td>Open a new circuit. Payload contains onion key challenge.</td>
</tr>
<tr>
<td>CREATED</td>
<td>Acknowledge successful creation of a circuit. Payload contains onion key response.</td>
</tr>
<tr>
<td>RELAY</td>
<td>Send a relay command and a relay payload to a OR.</td>
</tr>
<tr>
<td>DESTROY</td>
<td>Destroy a circuit. Payload contains reason for destroying.</td>
</tr>
</tbody>
</table>

Figure 1. Four Tor cell commands and their respective purposes.

2. Tor Cells

All of the OP’s communication with the network is done via an application level protocol specified in tor-spec.txt [Dingledine and Mathewson, 2013]. The building blocks of this protocol are Tor cells, 514 byte messages used to send data and commands to OPs and ORs. The first 3 bytes of a cell contain a circuit ID, the 4th byte a command, and the remaining 510 a payload. The circuit ID field is needed because Tor can multiplex circuits over one connection; that is, one OP can build multiple circuits through the same OR. The command field contains one of 11 possible Tor commands. Explanations for the four most relevant to our research are given in Figure 1.

The payload of a RELAY cell contains additional header fields, a relay subcommand, and a data payload. The data payload contains Alice’s onion encrypted message. Figure 2 shows these fields. The “Recognized” and digest fields are used by an OR to determine if the payload is intact and fully decrypted. “Recognized” is set to 0 if the payload is fully decrypted. Digest is set to the first four bytes of the running digest of all data either bound for the OR or originating at the OR. It is used to ensure the integrity of the encrypted payload. If the ‘‘Recognized’’ field is 0 and the digest has the correct value, then the OR will forward Alice’s packet to bob.com. The Length field denotes the length in bytes of the data payload. Payloads shorter than 503 bytes are NUL-padded. The subcommand field may contain one of 15 possible subcommands. The five most relevant ones to our research are listed in Figure 3.
1. INTRODUCTION

<table>
<thead>
<tr>
<th>Command</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>RELAY_BEGIN</td>
<td>Open a new anonymized TCP connection. Payload contains destination address and port.</td>
</tr>
<tr>
<td>RELAY_DATA</td>
<td>Send data. Payload contains onion encrypted TCP packet.</td>
</tr>
<tr>
<td>RELAY_CONNECTED</td>
<td>Acknowledgement for RELAY_BEGIN. Payload contains info about the TCP connection.</td>
</tr>
<tr>
<td>RELAY_EXTEND</td>
<td>Extend the circuit by adding an OR. Payload unspecified.</td>
</tr>
<tr>
<td>RELAY_EXTENDED</td>
<td>Acknowledgment for RELAY_EXTEND. Payload unspecified.</td>
</tr>
</tbody>
</table>

Figure 2. Format of a RELAY cell payload.

Figure 3. Five RELAY commands and their respective purposes.

3. Circuit Creation

Figure 4 demonstrates the process of Alice creating a 3 hop circuit, then anonymously communicating with bob.com. To initiate the circuit, Alice sends a CREATE cell to R1, and R1 acknowledges with CREATED. Alice next sends a relay cell with the RELAY_EXTEND command, prompting R1 to build the next step of the circuit with R2. Once Alice is notified that her RELAY_EXTEND was successful with a RELAY_EXTENDED, she sends a second RELAY_EXTEND to R2. When this is acknowledged, Alice sends a RELAY_BEGIN command to open a TCP connection between R3 and bob.com. With this connection established, Alice may now anonymously send and receive packets through the circuit via her SOCKS connection to the OP.

4. Contributions of this Thesis

Tor is widely researched in the academic community. Areas of study include devising new attacks on the network, building defenses against these attacks, and creating new traffic routing algorithms to improve latency and bandwidth. For both ethical and practical reasons, it is often necessary to do this research on a simulation of the extant network. For researching attacks, it is unethical to attempt attacks on real Tor users.
For researching routing algorithms, it is not possible to witness the whole-network effects of a change without deploying experimental code to every relay on the network, which is both undesirable and likely impossible in practice.

ExperimenTor [Bauer et al., 2011] and Shadow [Jansen and Hopper, 2012] are two freely available Tor simulation platforms that have been used in academic research. Both of these platforms allows a researcher to run many Tor relays and clients simultaneously within a simulated Internet. They each offer sophisticated models of network factors like packet latency, traffic congestion, and bandwidth limitations. Both also make a
thoughtful attempt at “scaling down” the global distribution of Tor relays and clients to a computationally manageable size. A weak point of both platforms, however, is their models of the traffic generated by network clients. Both only provide simple simulations of World Wide Web traffic.

Without realistic network traffic, OPs and ORs in the simulated network are not operating under the same conditions that they would be in real-world use. This fact compromises the validity of any conclusions drawn from research done in the simulation. To realize the full benefits of a simulated environment, a better model is needed.

In this thesis, we focus on the problem of modeling the outbound traffic sent from an OP into the network. Using a guard relay that we control, we recorded roughly one day’s worth of timing data for outbound RELAY, CREATE, and DESTROY cells. Using this data, we formed a set of windowed time series, in which each observation is the number of RELAY cells occurring within a 5 second window. Since Tor nodes are unaware of the data transmitted in a client’s RELAY DATA cells, we believe that these cell counts are the most relevant possible characterization of client behavior. The only aspects of client traffic that a relay can reasonably react to are its volume and its timing dynamics.

Our goal is to create a generative model of these series, i.e. one capable of producing new series with similar dynamics on demand. Based on our intuitions about Internet client behavior, we believe that a Hidden Markov Model is an appropriate model of this data. A Hidden Markov Model (HMM) is a stochastic process that probabilistically transitions between a finite set of states. At each state, it emits a symbol; in our case, a windowed cell count. Intuitively, a HMM is an appropriate model of client behavior; one can envision a client transitioning between states such as “uploading file” and “sending email”.

To train a HMM on our dataset, we first generate exploratory visualizations to aid our understanding of its behaviors. We then use HMMCluster, a statistical sequence clustering algorithm proposed by Smyth [1997], to cluster the series into similar groups and train a multi-part HMM on them. Qualitative analyses of our model indicate that,
though there is room for improvement, the HMM we have created serves as faithful
model of the client traffic patterns in our dataset.
CHAPTER 2

Existing Tor Simulations

1. ExperimenTor

ExperimenTor [Bauer et al., 2011] is a Tor simulation toolkit built on top of the ModelNet network emulation platform [Vahdat et al., 2002]. At least two computers on the same intranet are required to run ExperimentTor - one or more FreeBSD core routers to simulate the network, and one or more Linux edge nodes to run processes like ORs and OPs, BitTorrent clients, and HTTP servers. As an extension to ModelNet, ExperimenTor inherits the ability to scale to an arbitrary size computing cluster, yielding the potential for a very large simulated Tor networks. ExperimenTor is a real time simulation, and is capable of running the Tor software completely unmodified.

To date, ExperimenTor has been used in two research projects on Tor. AlSabah et al. [2011b] use ExperimenTor to test new techniques for congestion and flow control. In a separate paper, AlSabah et al. [2011a] also use ExperimentTor to evaluate a new approach to multi-path circuit construction and stream splitting.

In ModelNet, a network topology is specified in a Graph Modeling Language file. Edges in the graph, representing network links, may be annotated with values for bandwidth, latency, and packet drop rate. For very large topologies, ModelNet may optionally “distill” the network graph into a simpler form that is less computationally taxing to simulate.

Client processes like ORs run in virtual nodes, each of which represents one machine connected to the virtual network. Depending on how well-provisioned it is, one edge node may run many distinct virtual nodes. Each virtual node is assigned a unique IP address in the 10.0.0.0/8 address space. To ensure that outgoing TCP packets are correctly marked with their associated virtual address, system calls for socket creation
and name resolution - `bind`, `connect`, `gethostbyaddr`, and `gethostbyname`, for example - are intercepted at runtime.

To properly simulate effects like cross traffic and congestion, ModelNet models network links as *pipes*. A pipe is a packet queue with a “leaky bucket” at its entrance. The leaky bucket analogy stems from the way incoming packets are handled. Packets accumulate at the beginning of the pipe and “leak” into it at a constant rate. As long as the average packet rate stays below the pipe’s bandwidth rating, no packets are dropped. If the bandwidth rating is exceeded, however, the entrance “overflows” and packets are lost. Since pipes are shared between virtual nodes, a burst of packets from one node can thus cause another’s packets to be dropped shortly afterward. This behavior is consistent with that of a real network.

ExperimenTor automates the process of building a functional Tor network within ModelNet. Using the included configuration tool, a researcher may use the live Tor consensus document to generate a set of Tor relays and clients. The configuration tool maps router bandwidth measurements in the consensus to virtual nodes. To assign reasonable latencies to network links, the tool samples from the King network latency dataset [Gummadi et al., 2002]. At the beginning of an experiment, private directory servers are created, and their public/private keys are distributed to all Tor routers and clients in the virtual network. With routers, clients, and directory authorities instantiated, the virtual network then functions in the same manner as the live one.

Bauer et al. [2011] report running a simulated network of 1000 relays with one ModelNet core node, one edge node, and a 1 Gbps network switch. Since Tor has an estimated 250,000 clients daily, however, the problem of how to best “scale down” the distribution of clients remains an open question. For relays, ExperimenTor’s configuration tool maintains the same proportion of guard, middle, and exit nodes, while scaling all node bandwidths by a common factor. While this is a reasonable approach, Bauer et al. [2011] note that its effects on the simulated network’s “realism” have not been thoroughly studied.
2. EXISTING TOR SIMULATIONS

2. Shadow

Shadow [Jansen and Hopper, 2012] is a network emulation platform built for running Tor simulations. Unlike ModelNet, Shadow is easily run on one well-provisioned computer. Shadow is a discrete event simulator, meaning that all events in a network experiment occur in virtual time. Client applications are encapsulated by “plugin” libraries that implement a standard interface for communicating with the virtual network. Shadow may run multiple instances of a client application by dynamically managing each one’s memory in a strategy similar to kernel context switching.

Shadow has been used in three Tor research projects. Jansen and Hopper [2012] use Shadow to experiment with a recently integrated replacement for Tor’s round-robin circuit scheduling algorithm. Jansen et al. [2012a] use Shadow and Experimentor to validate the accuracy of a Tor network model. Jansen et al. [2012b] use Shadow to test three algorithms for throttling bulk transfers.

In Shadow, a network topology is specified as a linked set of geographic clusters. Each cluster is given values for upstream/downstream bandwidth and packet loss rate. Links between clusters are given values for latency and jitter (variation in packet delay). Client applications like Tor relays and web servers may then be attached to each cluster. Geographic clustering is used to scale the Internet down to a manageable size for simulation. Jansen et al. [2012a] argue that this is appropriate because geographic clustering most closely resembles the structure of the live Internet.

Shadow includes implementations of a bulk downloader, a simple web server, and a simulated web browser. Like Experimentor, Shadow’s virtual network is built on “pipes” with leaky buckets at their entrance [Jansen and Hopper, 2012]. These pipes are shared amongst client applications, resulting in realistic congestion and cross traffic effects.

To communicate with Shadow’s virtual network, a client application is encapsulated by a Shadow plugin. A plugin is a library, written independently of the client, that implements a set of callbacks defined by Shadow. These callbacks allow Shadow to
perform tasks such as creating new application instances, assigning IP addresses, and sending/receiving packets in a uniform manner across plugins. To integrate correctly into the simulation, a plugin’s encapsulated application must be single-threaded and non-blocking.

In addition to implementing a set of callbacks, a plugin must register pointers to all of its application’s variable state with Shadow. This step is what allows Shadow to run multiple instances of an application simultaneously. When the simulation needs to communicate with a running instance, that instance’s variable state is loaded into the appropriate memory locations and control is passed to it. Once communication is finished, the application relinquishes control and its variable state is moved to another, inactive memory location. Shadow may then load another instance’s state and transfer control.

Shadow runs Tor by wrapping it with the Scallion plugin. To register Tor’s variable state with Shadow, Scallion uses standard binary utilities like `objcopy`, `readelf`, and `nm` to scan, rename, and globalize Tor symbols. Registration code for these globalized symbols is then generated before compilation and included in Scallion. To further adapt Tor without modifying its source code, Scallion uses function interposition to replace selected pieces of functionality with appropriate substitutes at runtime. Using the latter technique, Scallion intercepts all of Tor’s socket system calls with its own virtual socket implementation. The virtual socket library packages outgoing data into packet objects, which are then pushed to the appropriate queue in the simulated network.

Shadow comes with a global network topology created by Jansen et al. [2012a] in their Tor modeling study. Values for each country’s upstream/downstream bandwidth and packet loss are taken from the Ookla Net Index dataset [Ookla]. Latency and jitter estimates are derived from measurements on the iPlane latency estimation service [iPlane]. In addition, Shadow includes four scaled-down Tor router/client distributions derived using Jansen et al. [2012a]’s best fit sample algorithm. These distributions range from 20 routers and 200 clients to 250 relays and 2500 clients [Jansen, 2012]. The former can run with less than 4 GiB RAM, while the latter requires nearly 64 GiB.
CHAPTER 3

Hidden Markov Models

A Hidden Markov Model, or HMM, is a type of stochastic process well-suited to general-purpose time series modeling. A HMM consists of a set of states that probabilistically transition between each other. Each time a state is visited, a symbol is drawn from a probability distribution and emitted. HMMs are widely used in the field of speech recognition to model the audio patterns of spoken words and syllables. Due to their generality, HMMs have also seen use in many other areas, including genetics, neurobiology, signal processing, and image analysis [Pyle, 2003]. In this chapter, we explore the structure and theory of HMMs. Our discussion is drawn from Rabiner’s 1989 tutorial on HMMS.

1. Discrete Markov Models

To understand HMMs, we first consider a simpler process, the discrete Markov Model. A Discrete Markov Model $M$ consists of a set of $m$ states $Q_1, Q_2, \ldots, Q_m$, an $m \times m$ matrix of transition probabilities $A$, and a length $m$ initial state distribution $\pi$. $A_{ij}$ is the probability of transitioning from state $Q_i$ to $Q_j$, and $\pi_i$ is the probability of choosing $Q_i$ as the first state. To generate a length $T$ time series from $M$, first choose a state $Q_i$ with probability $\pi_i$ and emit a symbol unique to that state. Randomly transition to another state $Q_j$ with probability $A_{ij}$ and emit another unique symbol. Continue transitioning in the same manner until $T$ symbols have been emitted.

Note that, by our definition of $A$, for any adjacent pair of states $Q_t, Q_{t+1}$

$$P(Q_{t+1}|Q_t, Q_{t-1}, \ldots, Q_0) = P(Q_{t+1}|Q_t)$$
The probability of the next state in the sequence is only dependent on the current one. This fact is called the Markov Property, and it has important implications in the theory of HMMs.

2. Hidden Markov Models

DMMs are very limited because they can only emit discrete symbols. To model numerical time series, a more flexible model is needed. HMMs generalize DMMs by adding a layer of indirection when emitting a symbol. Let \( B \) be a length \( m \) list of probability distributions. Upon visiting state \( Q \), instead of emitting a discrete symbol, we emit a symbol drawn from the distribution \( B_Q \). The power of HMMs comes from the fact that \( B_Q \) can be any distribution over any set. By using a distribution defined on the real numbers, a HMM becomes a powerful tool for modeling numerical series like ours. Since each emitted symbol is now the result of two random choices — transitioning to \( Q_i \) and drawing from \( B_{Q_i} \) — a HMM is a doubly stochastic process. Since we can no longer directly see what state the model is in, we say that the states are “hidden”. A HMM \( \lambda \) is notated as a three-tuple \( (A, B, \pi) \). We wish to model client traffic with Gaussian emission distributions, so we limit our discussion to these types of HMMs.

3. The Forward Recursion

One of the fundamental problems of HMMs is, given a HMM \( \lambda \) and length \( T \) time series \( O_1, O_2, \ldots, O_T \), what is \( P(O|\lambda) \), the probability of \( \lambda \) emitting \( O \)? To approach this problem, first consider the task of finding \( P(O|Q_0 \ldots Q_T) \), the probability of \( \lambda \) emitting \( O \) while traversing the state sequence \( Q_0 \ldots Q_T \):

\[
P(O|Q_0 \ldots Q_T) = \prod_{t=1}^{T} P(O_t|Q_t) = \prod_{t=1}^{T} B_{Q_t}(O_t)
\]
Marginalizing over all possible state sequences, we derive \( P(O|\lambda) \):

\[
P(O|\lambda) = \sum_{Q_0 \ldots Q_T} P(O|Q_0 \ldots Q_T)P(Q_0 \ldots Q_T)
\]

The problem with this naïve approach is that it requires \( 2T \times m^T \) calculations to compute — \( 2T \) per unique state sequence, iterated over \( m^T \) possible sequences. The exponential component makes it unfeasible even for relatively small \( m \) and \( T \).

The forward algorithm improves on this complexity by leveraging the Markov Property to eliminate redundant calculations. Let \( \alpha_t(i) = P(O_1, O_2, \ldots, O_t, Q_i) \) be the joint probability of observing the partial series \( O' = O_1, O_2, \ldots, O_t \) and being in state \( Q_i \) at time \( t \). \( \alpha_t(i) \) is called the *forward variable*, and it is defined inductively with respect to \( t \):

- **Base:** \( \alpha_1(i) = \pi_i B_{Q_i}(O_t) \) \( 1 \leq i \leq m \)
- **Induction:** \( \alpha_{t+1}(j) = \left[ \sum_{i=1}^{m} \alpha_t(i)A_{ij} \right] B_{Q_j}(O_{t+1}) \) \( 1 \leq t \leq T - 1, 1 \leq j \leq m \)
- **Termination:** \( P(O|\lambda) = \sum_{i=1}^{m} \alpha_T(i) \)

In the base case, \( \alpha_1(i) \) is the joint probability of choosing \( Q_i \) from the initial state distribution and emitting \( O_t \) from \( B_{Q_i} \). In the induction step, the summand in the bracketed term is the joint probability of observing \( O' \), being in \( Q_i \) at time \( t \), then transitioning from \( Q_i \) to \( Q_j \). Marginalizing over all possible \( Q_i \) yields the probability of observing \( O' \) and being in state \( Q_j \) at time \( t+1 \), while multiplying by \( B_{Q_j}(O_{t+1}) \) adds the event of emitting \( O_{t+1} \). In the termination step, marginalizing over all the states yields \( P(O|\lambda) \).

The key insight in this definition is that for all values of \( j \) in the induction step, the \( \alpha_t(i) \) term is the same. In a dynamic programming algorithm, it can thus be computed once and re-used \( m \) times at each \( t \) value. This savings is made possible by the Markov Property. Since the transition to \( Q_j \) is conditionally independent of all previous states,
3. HIDDEN MARKOV MODELS

$\alpha_t(i)$ remains separate from the rest of the expression. Computed in ascending order of $t$, each term in the inductive series $\alpha_1(i), \alpha_2(i), \ldots \alpha_T(i)$ now takes $O(m)$ calculations to compute, and the whole series thus $O(mT)$. Since this series must be computed for all $m$ states in the termination step, the total complexity of the forward algorithm is $O(m^2T)$.

4. The Baum-Welch Algorithm

Now that we know how to compute $P(O|\lambda)$, a natural question is how to find the model $\lambda'$ that maximizes this probability. Unfortunately, if we wish to find the globally optimal $\lambda'$, there is no way to do this. Given a non-optimal model $\lambda$, however, we may reestimate its parameters to improve $P(O|\lambda)$ up to a local maximum. The Baum-Welch algorithm is an iterative procedure for reestimating $\lambda$ such that $P(O|\lambda)$ may reach a local optimum.

To understand the Baum-Welch algorithm, we first define the backward variable $\beta_t(i) = P(O_{t+1}, O_{t+2}, \ldots O_T|Q_t)$, the probability of observing $O_{t+1}, O_{t+2}, \ldots O_T$ given that $\lambda$ is in state $Q_t$ at time $t$. $\beta_t(i)$ is defined inductively in a similar manner to $\alpha_t(i)$:

Base: $\beta_T(i) = 1 \quad 1 \leq i \leq m$

Induction: $\beta_t(i) = \sum_{j=1}^{m} A_{ij} B_{Q_j}(O_{t+1}) \beta_{t+1}(j) \quad t = T-1, T-2, \ldots 1; 1 \leq i \leq m$

With the forward and backward variables, we may now define the two central variables in the Baum-Welch algorithm, $\gamma_t(i)$ and $\xi_t(i, j)$. $\gamma_t(i) = P(Q_t|O)$ is the probability of being in state $Q_t$ at time $t$ given $O$. It is defined as

$$\gamma_t(i) = \frac{\alpha_t(i) \beta_t(i)}{P(O)}$$

Intuitively, $\alpha_t(i)$ accounts for all possible sequences through time $t$, and $\beta_t(i)$ all sequences after. Between the two variables, every state is allowed to vary except for $Q_t$.

---

1Rabiner claims that the base case in this definition is defined “arbitrarily”. We propose a more satisfactory reason: since $O_{T+1}$ does not exist, $B_T(i)$ is the probability of observing no output after the HMM has stopped emitting symbols, which must necessarily be 1.
We are ultimately marginalizing over every possible state sequence in which $\lambda$ is in state $Q_t$ at time $t$.

$\xi_t(i,j) = P(Q_t, Q_{t+1}|O)$ is the probability of being in $Q_t$ at time $t$, then transitioning to $Q_{t+1}$. Using the forward and backward variables, $\xi_t(i,j)$ is defined as:

$$\xi_t(i,j) = \frac{\alpha_t(i)A_{ij}B_{Q_j}(O_{t+1})\beta_{t+1}(j)}{P(O)}$$

where $\alpha_t(i)$ and $B_{t+1}(j)$ are used in a similar way to $\gamma_t(i)$.

With $\gamma_t(i)$ and $\xi_t(i,j)$, we may define reestimation formulas for $A$, $B$, and $\pi$:

$$\bar{\pi}_i = \text{Probability of state } i \text{ at time 1} = \gamma_1(i)$$

$$\bar{A}_{ij} = \frac{\text{expected # of transitions } Q_i \rightarrow Q_j}{\text{expected # of transitions away from } Q_i} = \frac{\sum_{t=1}^{T-1} \xi_t(i,j)}{\sum_{t=1}^{T-1} \gamma_t(i)}$$

$$\bar{\mu}_i = \frac{\sum_{t=1}^{T} \gamma_t(i)O_t}{\sum_{t=1}^{T} \gamma_t(i)}$$

$$\bar{\sigma}_i = \sqrt{\frac{\sum_{t=1}^{T} \gamma_t(i)(O_t - \bar{\mu}_i)^2}{\sum_{t=1}^{T} \gamma_t(i)}}$$

In the definition of $\bar{A}_{ij}$, Rabiner uses alternate interpretations of $\xi_t(i,j)$ and $\gamma_t(i)$: $\xi_t(i,j)$ is treated as the fractional number of transitions from $Q_i$ to $Q_j$ at time $t$, and $\gamma_t(i)$ as the fractional number of transitions away from $Q_i$. Summing these fractional transitions yields the expected frequencies of these events over the whole time series. $\bar{\mu}_i$ and $\bar{\sigma}_i$ are special cases of formulas Rabiner presents for reestimating an arbitrary continuous emission distribution. The two presented here are used for single variable Gaussian emissions. $\bar{\mu}_i$ is a weighted average in which each $O_t$ is weighted by the probability of visiting $Q_i$ at time $t$. $\bar{\sigma}_i$ weights each term in the same manner within the sum of squared differences. By iteratively applying the above four formulas, $P(O|\lambda)$ may be optimized up to a local maximum.
CHAPTER 4

Exploratory Data Analysis Techniques

In the data mining process, exploratory analysis is one of the most important steps a researcher must take. Exploratory analysis is the informal process of manipulating and visualizing data in search of intuitive patterns. Despite its informality, exploratory analysis is important because it helps us better understand the data we are trying to model. Pyle [2003] explains,

The most formidable pattern recognition apparatus known to humankind is easily accessible to any data miner – the human brain and mind. So powerful is the human mind that it will perceive patterns even when none exist. The question is how to enable this formidable apparatus to go to work most effectively in data mining.

Exploratory analysis is an effective answer to Pyle’s question. With the correct statistics and visualizations, we may form hypotheses about the data that inform the structure of our model.

1. Distribution Analysis

The dataset we will be analyzing is a set of single variable, natural number-valued time series. For each series $O$, $O_t$ is the number of relay cells observed in time window $t$. Since the difference between two cell counts is meaningful, $O$’s observations are interval data. For a single series, Warner [1998] suggests first examining the distribution of $O$’s observations. The distribution tells us what kinds of observations $O$ contains, how often they occur, and whether they tend to clump around one or more values. Note that since distribution analysis does not assume ordered data, it tells us nothing about
how $O$ changes over time. In spite of this shortcoming, the observation distribution is still a useful, though incomplete characterization of a time series.

The maximum, minimum, mean, median and standard deviation are commonly used descriptive statistics for summarizing a distribution. Below, we address each of these as they pertain to an individual OP’s relay time series.

- **Maximum cells/second.** This helps us assess the activity level of a circuit. A time series with a high maximum implies that at at least one point in time, the client was sending a large amount of outbound data.

- **Minimum cells/second.** Due to the bursty nature of web traffic, this is not a useful statistic for our research. Even a highly active circuit is likely to have idle sections at 0 cells/second.

- **Mean cells/second ($\mu$).** The mean is a measure of central tendency. For normally distributed observations, it tells us where the bulk of the corresponding Gaussian curve’s mass lies. $\mu$ must be interpreted very cautiously in our setting because, due to their burstiness, our observations are not always normal. $\mu$ still serves as a useful proxy of the overall activity level present in a circuit. A value near 0 is evidence of an idle client, while a higher $\mu$ suggests at least one period of active communication.

- **Median cells/second.** The median is a robust measure of central tendency. It is robust because its value is not influenced by outliers, and its meaning is the same regardless of the observations’ distribution. For a length $T$ time series $O$, the median is found by sorting $O$’s observations in ascending order of value. If $T$ is odd, then the median is the $\lceil T/2 \rceil$-th value. If $n$ is even, it is the mean of the $T/2$-th and $T/2 + 1$-th values.

- **Standard deviation of cells/second ($\sigma$).** The standard deviation $\sigma$ is a measure of dispersion, telling us how much a set of observations tend to differ from their mean value. For normally distributed observations, it characterizes the shape of the corresponding Gaussian. A high standard deviation produces a low, flat
curve, while a low one produces a sharp peak at \( \mu \). For a time series \( O \), \( \sigma \) is defined as the square root of \( O \)'s variance:

\[
\sigma = \sqrt{\frac{1}{n} \sum_{o \in O} (o - \mu)^2}
\]

A series with \( \mu \) and \( \sigma \) near zero rarely, if ever, peaks above 0, suggesting an idle circuit. A high \( \sigma \) suggests a “peaky” series corresponding to bursts of data above the mean rate.

To visualize a distribution, we use a frequency histogram. A frequency histogram is a bar chart in which each each bar represents the number of observations falling within a fixed range of values. A frequency histogram is constructed by binning \( O \)'s observations by their value into a set of evenly sized intervals. Each interval \([j, k]\) is then plotted as a bar on the graph with height equal to the number of items in \([j, k]\).

2. Temporal Analysis

Distribution analysis is a general purpose method that applies to any type of interval data. To address \( O \)'s temporal dynamics, however, time series–specific methods are needed.

One of the most useful visualizations of a time series is the time plot. Time is placed on the \( x \)-axis, observation values on the \( y \)-axis, and a line is drawn through the points. A time plot allows us to visually search for peaks, trends, and patterns in an individual series. An example time plot is shown in the top half of Figure 1.

To test for the existence of recurring patterns in \( O \), we may used the lagged autocorrelation. To understand the lagged autocorrelation, first consider the sample Pearson correlation \( r \) between two length \( n \) series \( O \) and \( O' \):

\[
r_{OO'} = \frac{\sum_{i=1}^{T} (O_i - \mu_X)(O'_i - \mu_Y)}{(T - 1)\sigma_O \sigma'_{O}}
\]

\( r_{OO'} \) assesses the strength of linear association between \( O \) and \( O' \). It ranges from -1 to 1. A value of 1, or perfect positive correlation, indicates a very strong linear relationship.
between $O$ and $O'$. A value of -1, or perfect negative correlation, indicates an equally strong but negated linear relationship. 0 indicates the lack of any linear relationship. Cohen [1995] notes $|r| \geq .7$ is generally evidence of strong relationship. He also notes that visual inspection of $O$’s time plot is necessary to confirm inferences drawn from $r$.

Given a time series $O$, let $O^k$ be the result of lagging $O$ by $k$ time units, such that $O^k_t = O_{t-k}$. The lag $k$ autocorrelation of $O$ is $r_{OO^k}$, the Pearson correlation of $O$ and $O^k$. The value of $r_{OO^k}$ tells us how strongly $O_t$ is predicted by $O_{t-k}$. $|r| \geq .7$ suggests a repeating pattern in the size $k$ time window. The top graph in Figure 1 shows an example time series displaying highly cyclic behavior. The bottom graph is called a correlogram, and it plots the series’ autocorrelation at varying $k$ values. Note how in the bottom graph, the autocorrelation peaks when $k$ is a multiple of 4. This provides strong evidence that the series has a repetitive component in the 4 unit time window.
CHAPTER 5

Clustering

1. Overview

Clustering is the process of partitioning a set of data points into two or more groups such that elements in the same group are qualitatively similar, while elements in separate groups are distinct. In the machine learning literature, clustering is classified as an unsupervised learning method because it does not make use of pre-classified data points. Formally, given a set of input vectors \( X = \{x_1, x_2, \ldots, x_n\} \), clustering creates a \( k \)-partition \( C = \{c_1, c_2, \ldots, c_k\} \) obeying three properties [Xu and Wunsch, 2005]:

- \( c_i \neq \emptyset \) (no cluster may be empty)
- \( \bigcup_{i=1}^{k} c_i = X \) (no orphan data)
- \( c_i \cap c_j = \emptyset \), \( i, j = 1, 2, \ldots, k, i \neq j \) (no overlap between partitions)

Clustering has a wide range of applications in data mining. In market research, clustering is used to segment customers into meaningful subgroups for targeted advertising [Gan et al., 2007]. In bioinformatics, it is used to group gene expression data into groups with similar expression patterns. Examples abound in other fields.

To cluster a dataset, two components are typically needed: a dissimilarity or similarity function, and a clustering algorithm. A dissimilarity function \( d : X \times X \rightarrow \mathbb{R} \) is a function that expresses how dissimilar two data points are. \( d \) must obey two axioms:

- \( d(x_i, x_j) = d(x_j, x_i) \) \( \forall i, j \) (Symmetry)
- \( d(x_i, x_j) \geq 0 \) \( \forall i, j \) (Positivity)

If \( d \) also satisfies the two axioms below, it is referred to as a distance metric.

- \( d(x_i, x_j) \leq d(x_i, x_k) + d(x_k, x_j) \) (Triangle Inequality)
- \( d(x_i, x_j) = 0 \) iff \( x_i = x_j \) (Reflexivity)
Euclidean distance between vectors is a commonly used example of a distance metric.

A similarity function \( s : X \times X \to \mathbb{R} \) must obey the same Symmetry and Positivity axioms. To qualify as a metric, it must obey these additional axioms:

- \( s(x_i, x_j) s(x_j, x_k) \leq [s(x_i, x_j) + s(x_j, x_k)] s(x_i, x_k) \)
- \( s(x_i, x_j) = 1 \) iff \( x_i = x_j \) (Reflexivity)

2. The k-Means Algorithm

The k-means algorithm is one of the simplest and most widely used clustering algorithms [Gan et al., 2007]. It was first described by MacQueen [1967]. Given a \( n \)-dimensional dataset \( X \subseteq \mathbb{R}^n \) and a fixed number \( k \), k-means partitions \( X \) into \( k \) clusters. k-means begins by first randomly\(^1\) partitioning \( X \) into \( k \) initial clusters. Each cluster \( C \) has a centroid or cluster prototype \( \mu(C) \) that represents \( C \)'s center. This is typically the \( n \)-dimensional vector whose \( j \)th coordinate is the mean of the \( j \)th coordinate of the vectors in \( C \).

The partition is iteratively refined as follows: first, for each data point \( x \in X \), the distance between \( x \) and \( u_C \) is calculated for each cluster. Each \( x \) is then reassigned to the cluster whose centroid it is closest to. The centroids are then recomputed for any clusters that have changed. This process is repeated until either the clusters stop changing, a maximum number of iterations is achieved, or the change in the error function

\[
E = \sum_{i=1}^{k} \sum_{x \in C_i} d(x, \mu(C))
\]

is less than a threshold \( \epsilon \) [Gan et al., 2007].

\( E \) is an example of a cluster quality metric. A low value of \( E \) implies that \( X \) has been partitioned into compact and well-separated clusters. Conversely, a high value of \( E \) indicates that one or more clusters contain inappropriately heterogeneous data. In general, cluster quality metrics give us a concrete way to assess how effectively a given algorithm and dissimilarity function partitions a data set. A wide variety of metrics

\(^1\)The initial partition may be created in other ways, but it is typically done randomly.
5. CLUSTERING

fun kmeans(X: dataset, k: int, ϵ: real)
begin
    clusters ← randPartition(X, k)
    terminate ← false
    while not terminate
        centroids ← centroids for clusters
        curErr ← E(clusters)
        for x in X
            C = the cluster that x is currently in
            C’ = arg min\(_{1≤i≤k} d(x, c_i)\)
            if C ≠ C’
                Move x to C
                didChange = true
            else
                didChange = false
            endif
        endf
        terminate ← didChange and curErr − E(clusters) > ϵ
    endw
    return clusters
end

Figure 1. Pseudocode for the k-means algorithm.

exist for different algorithms and data types. Gan et al. [2007] provides an extensive survey of available metrics.

Pseudocode for k-means is presented in figure 1. k-means’ most desirable characteristic is its speed. Each iteration requires O(|X|kn) calculations [Phillips, 2002]. Since k and n are typically small constants, it may run in essentially linear time with respect to |X|. This is significantly faster than most other clustering algorithms, making k-means well suited for large datasets. k-means does have well-documented drawbacks, though:

- It only works well for clusters in which the data points tend to clump into hyperspheres, the n-dimensional generalization of spheres.
- There is no efficient and universal method for choosing the correct k or creating the initial partition. For a given k, a poor initial partition may result in misplaced data points, yielding a sub-optimal E metric. If k is too low, then
qualitatively distinct data points may be forced into the same cluster; if it is too high, then similar points may be incorrectly separated from one another.  
- Since every object is forced into a cluster, it is sensitive to outliers. PAM, a variation on \( k \)-means, addresses this by using the median vector instead of the mean as the cluster prototype [Kaufman and Rousseeuw, 1990]. Since the median is a robust statistic, this prevents the cluster centroids from being adversely affected by outliers.

3. Hierarchical Clustering

Unlike \( k \)-means, which creates a single partition on \( X \), hierarchical, agglomerative clustering builds a nested tree of partitions. The algorithm begins by dividing \( X \) into \( |X| \) singleton clusters – the leaves of the tree in Figure 2. At each iterative step, the two closest clusters are merged. The algorithm continues to merge clusters until there are only two left. To derive a \( k \) partition, the tree is cut at the appropriate level.

To determine which clusters to merge at each step, a dissimilarity function \( D \) defined on clusters instead of data points is needed. Two such measures are single linkage and complete linkage [Gan et al., 2007]. Given two clusters \( C_1 \) and \( C_2 \), single linkage defines the dissimilarity to be the pairwise dissimilarity between the two closest objects in \( C_1 \) and \( C_2 \):

\[
D_s = \min \{ s(x_1, x_2) : x_1 \in C_1, x_2 \in C_2 \}
\]

For this reason, it is also referred to as the nearest neighbor method. In complete linkage, \( D \) is the pairwise distance between the furthest two elements in the two clusters:

\[
D_c = \max \{ d(x_1, x_2) : x_1 \in C_1, x_2 \in C_2 \}
\]

Gan et al. surveys other available cluster dissimilarity functions.

Hierarchical, agglomerative clustering’s greatest advantage is that \( k \), the number of clusters, does not need to be known a priori. Instead, the researcher may examine the tree of partitions and choose the optimal level to cut at [Xu and Wunsch, 2005]. The optimal level may be found by visualizing the tree in a similar manner to Figure 2.
5. CLUSTERING

Figure 2. Illustration of agglomerative vs. divisive clustering.

```
fun agglomerative(X: dataset)
begin
  n ← |X|
  partitions ← []
  P ← {\{x\} : x ∈ X}
  while |P| > 2
    partitions ← P::partitions
    i, j = arg min_{1≤i,j≤n,i\neq j} D(P_i, P_j)
    P ← (P - \{P_i, P_j\}) ∪ \{P_i, P_j\}
    n ← n - 1
  endw
return partitions
end
```

Figure 3. Pseudocode for hierarchical, agglomerative clustering.
or by calculating a cluster quality measure such as $E$ at each level. A criticism of classical hierarchical algorithms is that they are sensitive to outlier data points. This is because once a data point is assigned to a cluster, it is not reconsidered at higher levels of the tree. If a data point is misplaced at an early stage of the algorithm, it will not be corrected. Classical hierarchical algorithms also run in at best $O(N^2)$ time, since at the bottom of the tree $|X|$ singleton clusters must be pairwise compared to one another. CURE, ROCK, Chameleon, and BIRCH are variants of agglomerative clustering developed in the late 1990’s and early 2000’s that seek to address some of the classical algorithm’s weak points [Xu and Wunsch, 2005]. They improve upon its complexity and add robustness against outliers via techniques such as random sampling and alternative representations of the data’s hierarchical structure.

4. The HMM Clustering Algorithm

The methods discussed thus far are normally used to cluster vector data. To model Tor clients, however, we need to cluster a set of time series. The most challenging aspect of clustering time series is defining a meaningful dissimilarity function between two series. Qualitatively, two time series may be considered “close” if their observation distributions are similar and they display similar temporal patterns. Using descriptive statistics such as those previously discussed, the observation distributions of two series may be meaningfully compared. Comparing temporal dynamics, however, is a more difficult problem. For a set of uniform length series, the Pearson correlation is a reasonable method of comparison; however, it is not defined on variable length series. To compare variable length series, more advanced methods are needed.

Xu and Wunsch [2005] survey the three most common approaches to clustering variable length time series. Sequence similarity algorithms use a dissimilarity function defined directly on two series, then cluster with a standard procedure like $k$-means or the agglomerative algorithm. Levenshtein/edit distance is a dissimilarity function commonly used for this purpose [Sankoff and Kruskal, 1983]. Indirect clustering algorithms summarize the characteristics of a time series in a fixed-length feature vector $v$ which
may then be used as input to a standard clustering procedure. Methods for constructing \( v \) vary based on which qualitative aspects of a time series the researcher is most interested in. *Statistical sequence clustering* partitions and models a set of time series by treating it as the output of a set of stochastic processes. Since these methods directly address the issue of modeling, they are particularly well suited to our research.

Smyth [1997] presents a statistical sequence clustering algorithm, HMMCluster, for partitioning and modeling a set of time series using HMMs. Given a set of series \( X \) and constants \( k \) and \( m \), HMMCluster yields both a \( k \)-partition of \( X \) and a \( k \times m \) state “composite” HMM faithful to \( X \)’s cluster structure. HMMCluster proceed as follows:

- first, a HMM is trained on each individual series.
- The series are then clustered using a dissimilarity function based on log-likelihood values. HMMs are next trained on the clusters and combined into one composite model, which is finally trained on the entire dataset \( X \).

Pseudocode for HMMCluster is presented in Figure 4. To train a \( m \) state HMM \( \lambda_i \) on a time series \( x_i \), \( \lambda \)'s parameters are initialized in a “default” manner with the function \( \text{initHMM} \). Pseudocode for \( \text{initHMM} \) is presented in Figure 5. \( \text{initHMM} \) first assigns \( \lambda \) a uniform transition matrix. It then uses \( k \)-means to cluster \( x_i \)'s observations into \( m \) clusters \( c_1, c_2, \ldots, c_m \), each of which covers a slice of \( x \)'s emission distribution. Each state \( Q_i \) is assigned a Gaussian emission distribution with mean \( \mu(c_i) \) and standard deviation \( \sigma(c_i) \). \( \lambda \) is then reestimated on \( x_i \) with the Baum-Welch algorithm. The dissimilarity between two series is \( x_i, x_j \) is now defined in terms of log-likelihoods:

\[
 d_L(x_i, x_j) = -\frac{1}{2}(L(x_i|\lambda_j) + L(x_j|\lambda_i))
\]

\( X \) is partitioned into \( k \) clusters using \( d_l \) as the dissimilarity function. Smyth suggests using the agglomerative algorithm with complete linkage, but notes that any applicable clustering algorithm is a valid choice. A new \( m \) state HMM \( \lambda_C \) is then trained on each cluster \( C \) with \( \text{initHMM} \). Smyth does not specify how to generalize \( \text{initHMM} \) from a single sequence to a cluster, so we assume that at the “default” initialization step, the series in \( C \) are concatenated into one joint series whose observations are then clustered.
fun HMMCluster(S: list of sequence, m: int, k: int)
begin
    Z ← Matrix(|S|, |S|) // the distance matrix
    M ← {} // the initial HMMs, one per sequence
    for s in S
        λ ← initHMM(s, m)
        M ← M ∪ {λ}
    endf
    for i in |M|
        for j in |M|
            D_{ij} ← 1/2(L(S_i|M_j) + L(S_j|M_i))
        endf
    endf
    P ← cluster(S, Z, k)
    M' ← []
    for c in P
        s' ← join(c)
        λ ← initHMM(s', m)
        M' ← λ::M'
    endf
    λ' ← compositeHMM(M')
    λ' ← BaumWelch(λ')
    return λ'
end

Figure 4. Pseudocode for Smyth [1997]’s HMM training algorithm.

In HMMCluster’s last step, all k HMMs are combined into one composite model λ’. Let $A^1 \ldots A^k$ be the transition matrices of $\lambda_{C^1} \ldots \lambda_{C^k}$. $A^1 \ldots A^k$ are placed into the block
fun initHMM(s: sequence, m: int)
begin
    λ ← HMM()
    C ← kmeans(s, m)
    λ.B ← ([μ_c, σ_c) : c ∈ C]
    for i in [1, m]
        for j in [1, m]
            λ.A_{ij} ← 1/m
    endf
    endf
    return λ
end

Figure 5. Pseudocode for Smyth [1997]’s “default” HMM initialization.

A’s block diagonal structure isolates the λ_{C_i}’s from one another. Any time series emitted by λ’ is the product of one distinct λ_{C_i}.

The composite initial state distribution π’ is constructed as follows. Let w(C_i) = |C_i|/|X| be the weight of cluster C_i in the composite HMM, and π^i be the initial state distribution of model λ_{C_i}. Then π’ = [w(C_1)π_1^1, ... w(C_1)*π_m^1, ... w(C_k)π_1^k, ... w(C_k)π_m^k].

The net effect of this construction is that λ’ behaves as though it were k separate HMMs, each of which has probability w(C_i) of being chosen to emit a series. An ancillary advantage of this approach is that evaluating the log–likelihood of λ’ on a set of sequences is much more straightforward than it would be given k individual models.

To evaluate HMMCluster experimentally, Smyth generated 20 length 200 series from 2 HMMs with similar dynamics. He then used HMMCluster to train a 2 HMM model
on the set of series. The HMMs yielded by Smyth’s algorithm were very accurate — no transition probability was off by greater than .02 from that of the corresponding generator HMM, and $\pi$ and $B$ were both similarly close to their ground truths.

5. Estimating $k$ With Monte-Carlo Cross-Validation

Smyth presents Monte-Carlo Cross-Validation (MCCV), a method for estimating the optimal $k$ for HMMCluster. For each potential $k$, $X$ is randomly partitioned into a training set $X_{\text{test}}$ of size $\beta * |X|$ and a testing set $X_{\text{test}} = X \setminus X_{\text{train}}$. A model $\lambda'$ with $k$ clusters is then trained on $X_{\text{train}}$ with HMMCluster, and the log-likelihood $L(X_{\text{test}}|\lambda)$ recorded. This process is repeated over $n$ trials, and at the end the mean log—likelihood is computed for each $k$. By plotting a bar chart of $k$ vs mean log-likelihood, one can see which $k$ achieved the highest score. Ideally, a large peak occurs at $X$’s true $k$ value, and in Smyth’s experiments this was indeed the case.
CHAPTER 6

Analysis of Training Data

1. Data Collection

The training dataset for our model was collected from Feb 21, 2013 at 1:03 am through Feb 22 at 11:49 am, for a total of 22 hours, 46 minutes. To collect the data, we modified version 0.2.35 of Tor’s OR software to log CREATE, RELAY, and DESTROY cells. A pseudonym for the sender’s address and the circuit ID were included in the log statements to facilitate separating individual circuits.

In the collection period, our guard router received 104,181 CREATE cells. Of these cells, 18,424 (17.68%) resulted in complete and valid circuits. We consider a circuit to be complete if it contains at least one CREATE, one DESTROY, three inbound, and three outbound RELAY cells. The latter requirement stems from the fact that at least three RELAY cells in each direction are needed to build a three hop circuit that connects to a service outside of the Tor network. We consider a circuit to be valid if all of its RELAY cells are logged after its last CREATE cell and before its first DESTROY cell. Since a circuit begins with CREATE and ends with DESTROY, the meaning of a RELAY cell outside of this window is undefined. 48,371,742 relay cells were received in total. 95% of these came from the top two most active clients. These same two clients only accounted for 18% of CREATES, however. This indicates that the majority of the traffic received was concentrated in a relatively small number of circuits. A likely explanation of this behavior is that the two clients in question were uploading one or more very large files (eg. a movie) to other network users.

For each client, a representative time series was constructed by grouping RELAY cells into 5 second, non-overlapping windows. For each 5 second time point, the corresponding observation is the number of RELAY cells received in that window.
2. Distribution Analysis

To examine observation distributions over the whole data set, we plot histograms of the mean, median, minimum, maximum, and standard deviation of each series’ observations. We also include a histogram of circuit lengths in seconds, and a histogram of instantaneous (i.e. single window) cell counts. All of these graphs are displayed in Figure 1. Due to large exponential drop-offs, a log scale is used on the y-axis for every chart. All graphs were produced in Python using Matplotlib [Hunter, 2007].

In the means cells/window histogram, the tallest bar is the leftmost one representing means from 0-10 cells/window. This tells us that the majority of clients spent most of their time sending comparatively little data. The thick tail on the right, however, confirms that there are also many more active clients as well. The two outliers to the right may potentially be attributed to bulk uploaders. Since the y-axis is already on a log scale, the graph’s seemingly exponential shape may indicate a doubly exponential dropoff in the mean cells/window distribution.

Median cells/window is very similar to mean cells/window. Since the mean and median are both measures of central tendency, this similarity is evidence that both graphs provide an accurate depiction of typical client activity levels. If these graphs were shaped very differently, we would not be able to draw conclusions from either one without further investigation.

The minimum cells/window chart indicates that, even for a series with a very high mean, there are almost always sections with little to no activity. The vast majority of minimums are less than 4 cells/window, suggesting we should expect very low “valleys” and flat sections in our time plots.

Maximum cells/window has a similar shape to the mean and median, but it ranges over much higher values. This tells us that the peak data rate of a client may lie well above its average. Due to this, we expect large, distinct spikes on our time plots.

The standard deviation of cells/window show that most of our series have observations well beyond their means. This is consistent with the small minimums and large
maxima we have seen. Like the other graphs, it may exhibit a doubly exponential dropoff. The highest bar covers the range $\sigma = [0, 5]$. We suspect that this bar represents the same majority of inactive clients that the previous four histograms suggest in their leftmost bar.
Instantaneous cell count yields less of a doubly exponential trend than the previous five graphs. It shows that even at the single-window level, lower activity levels are still the most commonly observed.

The circuit length histogram shows that the bulk of our circuits are less than 33:20 long, but some outliers stay open as long as 18 hours. Exceptionally long circuits can occur if the OP process is left running for an extended period. In this case, the OP leaves previously created circuits open for later reuse indefinitely.

3. Temporal Visualizations

To view how our series change over time, we use two different visualizations: a horizon graph, and a luminance plot. A horizon graph displays multiple time series simultaneously by layering them on top of one another in a nested manner [Heer et al., 2009]. The darkest portions of a horizon graph represent points in time at which multiple series are observing the same minimum value.

Figure 2 shows a horizon graph for 1000 randomly selected time series. Its high, jagged peaks are consistent with the high maximums and standard deviations shown in our histograms. The dark peak near \( t = 0 \) tells us that many of the series in this sample exhibit a peak of \( \geq 500 \) cells/second at their beginning. As time progresses forward, the color of the graph becomes progressively lighter. This behavior is consistent with that of the circuit length histogram. The further forward in time the we are, the fewer circuits there are that are still open.

The horizon graph’s greatest virtue is that each time series is the same physical shape as it would be in a single time plot. The principle drawback, though, is that it does not let us easily distinguish individual series from one another. A visualization less susceptible to this problem is the luminance graph. A luminance graph displays each time series as a series of colored boxes. Each box’s luminance is determined by the series’ observation at that point in time. Bright colors correspond to high values, and dark ones to low values.
Figure 3 shows a luminance graph for the same 1000 time series as the horizon graph. For the purpose of demonstration, the graph is significantly zoomed in, and hence not all of the series are shown. The series are sorted in ascending order of circuit length along the $y$-axis. Observation values greater than 700 are clipped. We can now clearly distinguish highly active series, such as the brightly speckled ones toward the top, from inactive ones with nearly uniform color. We can also see how often or sporadically a given series tends to peak by looking for isolated bright points. Series with similar color patterns suggest a natural cluster structure on the dataset.

The principle weakness of the luminance graph in comparison to the horizon graph is that the human eye cannot distinguish relative brightness as easily as it can physical space. This means that in exchange for greater separation between time series, we
sacrifice the resolution of our observations. To gain a thorough understanding of our data, the two visualizations are best used in tandem.

After exploring horizon and luminance graphs on multiple random samples, we produced 6 time plots representative of common attributes seen in the dataset. These plots are shown in Figure 4. Note that in order to preserve details, the series are shown at different scales. The following attributes of our data are highlighted:
Figure 4. 6 time series representative of common patterns seen in the dataset.

- A: Demonstrates tendency toward jagged peaks, shows activity in the 40-100 cell/window range
- B: Shows possibility of very high peaks followed by periods of little to no activity.
- C: Demonstrates a circuit that stays inactive for an extended period, then begins sending data.
- D: Shows a highly active circuit potentially characteristic of a bulk file uploader.
- E: Shows activity in the 5-25 cell/window range.
- F: Shows an inactive circuit. The lone plateau is likely due to cells sent during the circuit creation process. No data is sent afterward.

Note that, while these graphs give us insight into our data, they are not an exhaustive characterization of it. Patterns may exist which were not found in the exploratory process.
4. Conclusions and Model Justification

Exploratory analysis revealed that our time series display a very diverse range of behaviors. In addition, it showed us that despite this diversity, there are common attributes which may lend the data to a natural cluster structure. Many of the series display strong evidence of a client transitioning between multiple distinct states: building a circuit, sending data, falling inactive, then perhaps sending more data at a later time. Though HMMs are not the only process capable of modeling these series, we believe that they are naturally suited to these types of series. Furthermore, due to the diversity of our series, it is clear that one basic HMM is not enough to accurately model client behavior. Training a set of models on clustered series is a natural way to capture all of the distinct patterns present.
CHAPTER 7

Implementation and Results

1. Implementing HMMCluster for Relay Time Series

Since Smyth does not provide an implementation of HMMCluster, we developed one in Python. Python has a rich ecosystem of high performance machine-learning libraries which we were able to take advantage of. Implementations of the forward and Baum-Welch algorithms were provided by The General Hidden Markov Model library [Schliep, 2013], a C/Python library under active development at Rutgers University. Implementations of $k$-means and the agglomerative algorithm were respectively provided by Scikit-learn [Pedregosa et al., 2011] and fastcluster [Müllner]. Python’s multiprocessing API was used to compute pairwise sequence dissimilarities in parallel.

Our implementation of HMMCluster was able to reproduce the results of Smyth [1997]’s experiment perfectly. To further test it, we also modeled data produced from two HMMs with very different dynamics, two HMMs with distant emission distributions, and groups of three HMMs. Accurate parameter estimates were returned in all cases; transition probabilities were rarely off by more than .01, and emissions means and standard deviations were similarly close to their respective ground truths.

Modeling relay time series raised a number of challenges distinct from those addressed by Smyth. Due to the highly diverse nature of our series, it was common for two series to be so dissimilar that $d_L$ evaluated to $\infty$ due to floating point limitations. The complete linkage method is undefined on infinite distances: given two infinitely distant neighbors, which one’s cluster should be merged? To avoid this problem, we performed a log transformation on the series before modeling them. This transformation compressed the range of observation values enough to avoid such minuscule probabilities.
Time series like the ones in Figure 4B and 4C show that a Tor client may sit idle for an extended period before or after sending traffic. Since we are primarily interested in what traffic is sent, not when it is sent, we trim these leading and trailing idle sections before modeling. We conservatively define an idle section to be one in which no window five second window observes more than one RELAY cell. Of the 18,224 series collected, 5109 of them were uniformly idle. We discarded these series, as they hold no value for modeling client traffic.

Surprisingly, after trimming idle sections, 5591 of our series had a standard deviation of 0. A possible explanation of this behavior is that all of the associated clients were consistently pushing against a bandwidth cap, either at our guard node or at an Internet way point. These series are still valuable to us, but due to their lack of variance, Hidden Markov Models are not an appropriate model for them. Figure 1 shows summary histograms for these series. Though it is not pursued here, a natural approach to modeling them would be to fit continuous probability distributions to each of the histograms. Since a series with no variance has no temporal dynamics, these distributions alone would fully capture the nature of these series.

2. Monte-Carlo Cross-Validation and Parameter Selection

After trimming inactive sections and removing zero variance series, 7224 series remained for HMM training. Since we do not know the correct \( k \) or \( m \) values for HMM-Cluster \( a \) priori, we extended Smyth’s MCCV approach to search over both parameters. \( k \) and \( m \) were respectively varied from 4 to 50 and 3 to 10. 22 trial likelihoods were computed for each parameter combination at \( \beta = .5 \). Computing \( d_L \) was by far the most expensive step of the algorithm. Though exact running times were not recorded, performing all 22 trials on a 2.67 Ghz, 16 core server required on the order of two day’s time.

Figure 2 shows a surface plot of the mean cross-validated likelihoods over the parameter space. The dark, horizontal band at \( m = 6 \) suggests that this is the correct number of HMM states to use. The correct \( k \) value, however, is less clear. From \( k = 10 \)
to $k = 30$, the surface has a nearly uniform color. The darkest areas reside toward the highest $k$’s, but we are hesitant to use these due to the risk of over-fitting our model to the data.

The top of Figure 3 displays a bar chart of $k$ vs. mean log-likelihood with $m$ fixed at 6. The whiskers on each bar represent a 95% confidence interval around the corresponding mean. The bottom chart shows the percentage change in log likelihood as $k$ rises. The fact that the top chart’s whiskers overlap so much indicates that none
of the means are statistically significant, and thus we cannot safely conclude that any one $k$ value is superior to another.

Without the ability to infer $k$ from MCCV, we appeal to past research on Tor traffic to choose a sensible value. McCoy et al. [2008] performed a study on traffic data collected at an exit relay, and found that there are 7 types of client traffic typically seen on the network: HTTP, SSL, BitTorrent, Instant Messaging, E-mail, FTP, and TelNet. Based on these findings, we intuitively set $k = 7$ and $m = 6$.

The results of running HMMCluster on all 7224 time series are visualized in Figures 4 and 5. Figure 4 displays encouraging results. All of the distributions are shaped very similarly to the ones we saw in our exploratory analysis. The lower frequency of mean 0 circuits is due to the fact that the HMMs were trained on data in which inactive circuits were removed. In Figure 5, the left hand column contains luminance and horizon graphs for series drawn from each of the seven clusters. Where applicable, the display is limited to a random sample of 100 series. The right hand column contains the same graphs for 100 synthetic, length 100 time series generated by the HMM component corresponding to each cluster.

The luminance plots for synthetic datasets 1, 2, 3, 4, suggest internally consistent clusters, but 5 and 7 do not. The poor internal consistency of these datasets may be explained by the small size of their respective clusters. The synthetic datasets appear
qualitatively similar to their sampled counterparts, indicating that the HMMs have successfully captured their respective cluster’s dynamics. The horizon graphs help us assess the general activity level induced by each model, but due to the fixed length synthetic series we are producing, they are not directly comparable to those of the training data.
Figure 4. Summary histograms for 7224 synthetically generated series.
Figure 5. Comparison of sample data from clusters, and synthetic data from the corresponding HMMs
CHAPTER 8

Conclusion

In this thesis, we set out to create a generative model of outbound client behavior on the Tor network. Due to the lack of statistical significance in MCCV, we were unable to make a purely data-driven decision about our model parameters. Qualitative analyses indicate that in spite of our intuition driven choices, though, we were able to create a competent model of our collected dataset. A logical next step in this research is to evaluate the quality of our model against entirely separate datasets collected on different days. If the model’s behavior is not consistent with these new datasets, more research will be needed in order to extract a truly representative training sample; if it is, however, then we may conclude that our model generates realistic outbound traffic suitable for use in a simulated Tor network.
Bibliography


D. Müllner. fastcluster: Fast hierarchical clustering routines for r and python.


