IMPROVED BRUTE FORCING METHODS WITH MARKOV MODELS

JUSTIN HIEMSTRA

Abstract. Bruteforcing methods have been used with limited success in guessing passwords longer than 6 characters, because as the length of a password increases, the number of possible combinations grows exponentially. However, traditional bruteforcing methods rely on the fact that passwords are distributed uniformly amongst the set of all possible passwords. This assumption is flawed because when people pick passwords, they do so based on observable patterns. These patterns effectively lower password entropy and make passwords subjectible to probabilistic analysis. Various Markov Models applied to bruteforcing have shown success at guessing passwords with higher frequency than traditional bruteforce methods, and this paper seeks to explore these models and their relative effectiveness. The models discussed are Markov Chains (without memory, with memory, and indexed) and Hidden Markov Models.

Contents

1. Introduction and Background Information 2
1.1. Some Basic Probability and Set Theory 4
2. Markov Chains 4
2.1. Standard Markov Chains 4
2.2. Markov Chains with Memory 12
2.3. Indexed Markov Chains 13
3. Hidden Markov Models 15
3.1. Hidden Markov Models 15
3.2. Viterbi Algorithm 19
3.3. Forward/Backward and Baum-Welch Algorithms 22
4. Application with Concluding Comments 31
4.1. Application of Standard Markov Chains 32
4.2. Application of Markov Chains with Memory 32
4.3. Application of Indexed Markov Chains 33
4.4. Application of Hidden Markov Models 35

Date: April 12, 2019.

This document is a senior thesis submitted to the Mathematics and Statistics Department of Haverford College in partial fulfillment of the requirements for a major in Mathematics.

Acknowledgments 36
References 37
Appendices 38
A. Python Code 38
A.1. Python code used to approximate standard Markov chain transition matrix of a password set 38
A.2. Python code to approximate standard initial distribution vector of a password set 41
A.3. Python code to approximate transition matrix for Markov chain with memory 2 43
A.4. Python code to approximate initial distribution vector for a Markov chain with memory 46
A.5. Python code to approximate transition matrices for an indexed Markov chain 48
A.6. Python code to calculate P(password) given data set for standard Markov chains 52
A.7. Python code to calculate P(password) given data set and Markov chain with memory 54
A.8. Python code to calculate P(password) given data set and indexed Markov chain 56

1. Introduction and Background Information

Passwords have been used for thousands of years as a way of authenticating users and granting access – just think about the most famous tale of the Arabian Nights, where woodworker Ali Baba famously discovers the command "open sesame" to unlock the treasure of the 40 thieves. Needless to say, passwords have evolved tremendously in the last century since the advent of the modern computer. Now they can be stored, hashed and salted, cracked, and randomly generated, and much more all with little more than a few keystrokes. While this has helped to revolutionize computer security, it has also led to attacks on security that less than several decades ago would have seemed like the inventions of the most bold of science fiction writers. If Ali Baba had been able to guess 1 million passwords a second, his tale would seem much less compelling. Nevertheless, even an 8 digit password with the option of both lowercase and capital letters, numbers, and special characters (which for this example we will say are `!@#$%&*()^`) has $72^8 = 722,204,136,308,736$ possible variations. If we were to guess these passwords at an arbitrary, although seemingly fast rate of 1 million guesses per second, it would still take nearly 18 years to guarantee the correct password was found. If we were to guess these passwords at a rate of 8 billion per second (which is entirely feasible using common computers for a fast hashing

algorithm), then this time drops to a mere 25 hours. Even so, with a mere increase in the length of the password to 12 characters with the same rules, keeping a guess rate of 8 billion guesses per second, the time jumps to an astronomical 76929 years. It goes without saying that this is too long for most of us to wait.

These numbers are making a critical and extremely flawed assumption, though – namely that people's passwords are uniformly distributed amongst the space of all potential passwords. Of course this is intuitively false, as any person could tell that "Kittycat123!" is much more likely to be a password than "0k@#Hckpdz%5," even though both passwords are 12 characters. It is also intuitive that the first option is much more memorable than the second, which is one possible reason most people would refrain from picking the second password. In any case, humans have shown themselves time and time again that they are the weakest link in security and are inherently bad at picking good passwords while being inherently good at picking bad passwords. With this in mind, it should be possible to derive a mathematical approach that explains why we intuitively know that "Kittycat123!" is a bad password and "0k@#Hckpdz%5" is probably not bad, and vice versa, why it makes more sense to guess passwords like "Kittycat123!" first. Armed with this logic, it should be possible to drastically reduce the number of passwords worth guessing, effectually introducing a time-space tradeoff wherein we sacrifice the guarantee of getting all passwords over a very long period of time for a high probability that we get many passwords relatively quickly.

So how, then, can we mathematically represent the intuitive differences between different types of passwords? To begin exploring this question, we must first understand some password trends. A 2004 study done at the University of Cambridge sought to answer some basic questions about password generation in relation to the security-memorability tradeoff amongst a sample of first year students. What they found was that password memorability was one of the most significant and determining factors that went into password selection. Amongst their other results, they found that special character use was almost non-existent, and many passwords were some form of permutation or direct use of common dictionary words[11]. Additional, more recent research by Microsoft developer Troy Hunt confirms this [3]. This final point lies at the heart of this thesis, because it allows us to make the fairly good assumption that many passwords look like words[1], much like "Kittycat123!," which is especially helpful because there are already mathematical models that help describe the characteristics of words. One such model is the **Markov Model**.

[1]Note that here, *word* refers to the natural definition of word and not the mathematical or linguistic definitions

1.1. **Some Basic Probability and Set Theory.** Before continuing, it will be helpful to discuss some basic probability and set theory that will be used at various times in this paper.

Definition 1.1 (Discrete Random Variable[4])**.** A random variable as it is used in this paper is a map from the state space to a specific outcome in the state space (which is defined in section 2.1). It is said to be discrete if it takes on at most countably many values (including finitely many).

Definition 1.2 (Stochastic Process[12])**.** A stochastic process is a family of random variables $\{X_T\}$ indexed by $1 \leq t \leq T$.

Put simply, this is a sequence of random variables that can take on values from the state space.

Definition 1.3 (Conditional Probability[4])**.** Let A, B be two sets. Then $\mathbb{P}(A|B)$, read "the probability of A given B", is defined

$$\mathbb{P}(A|B) = \frac{\mathbb{P}(A \cap B)}{\mathbb{P}(B)}$$

Theorem 1.4 (Associative Laws for Sets[4])**.** *Let A, B, C be sets. Then $A \cap (B \cap C) = (A \cap B) \cap C$ and $A \cup (B \cup C) = (A \cup B) \cup C$*

Theorem 1.5 (Distributive Laws for Sets[4])**.** *Let A, B, C be sets. Then $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$ and $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$.*

Theorem 1.6 (Law of Total Probability[5])**.** *Let A be a set and let B_i for $1 \leq i \leq T$ be a collection of subsets that partition the set B (ie for $1 \leq i \neq j \leq T$, $B_i \cap B_j = \emptyset$ and $\cup_{i=1}^{T}(B_i) = B$). Then*

$$\mathbb{P}(A) = \sum_{i=1}^{T} \mathbb{P}(A \cap B_i) = \sum_{i=1}^{T} \mathbb{P}(A|B_i)\mathbb{P}(B_i).$$

2. Markov Chains

2.1. **Standard Markov Chains.** Markov Models, which are a type of stochastic or random process, are frequently used in linguistics. Under the umbrella term *Markov Model*, there exist several different forms of Markov processes. In our quest to find the best way of guessing passwords, we consider one of the most rudimentary; the *Markov Chain*. The idea is fairly simple, but first we need some solid definitions. Once definitions are out of the way, we'll start with simple examples that should allow us to extrapolate to analyzing real password sets.

To begin, we need to consider how to mathematically describe passwords, which we can begin to do by thinking about the place where passwords are most often created: websites. When creating an account on most websites,

the user is prompted to enter a password, and in a majority of cases the website prescribes a set of rules for password creation. For example, FaceBook may require that your passwords are at least eight characters long, contain uppercase and lowercase letters, a number, and a special symbol. We can describe these different requirements (i.e. lowercase, uppercase, number, symbol) as a single parameter that dictates the creation of passwords, and for now we hold off adding password length. This is important to consider, because trying to guess FaceBook passwords by guessing only passwords with lowercase letters would be a waste of time. We continue, then, to define more thoroughly by defining a *state space*:

Definition 2.1 (State Space)**.** Let $\{X_T\}$ be a discrete stochastic process and let $\Sigma = \{\sigma_1, ..., \sigma_m\}$ be the set of values $\{X_T\}$ can take. Then we define Σ as a *state space*.

It should be noted that the elements in a state space, the states, are thought of as transitioning from one to another, and that the stochastic process $\{X_T\}$ at time t must occupy some state, so that $X_t = \sigma_i$ for $\sigma_i \in \Sigma$.

Example 2.2. If we consider $X = \{X_1 = x_1, ..., X_T = x_T\}$, a sequence of random variables representing some choice of characters in a password along with a state space $\Sigma = \{a, b, c, 1, 2, 3\}$, then for $T = 4$, we may observe the sequence $X = \{X_1 = a, X_2 = b, X_3 = 1, X_4 = 2\}$, which is just the password $ab12$. Moreover, any given state space Σ can have all the needed components for password creation, including upper case and lower case letters, along with numbers and special characters.

In the discussion of state spaces thus far, it has been mentioned only that states transition from one to another, without mention of how exactly these transitions are decided. This decision making process occurs according to some set of governing probabilites, which are given in the form of a transition probability matrix T_Σ.

Definition 2.3 (Transition Probability Matrix)**.** Let $\{X_T\}$ be a stochastic process and let $1 \leq t \leq T - 1$. Let T_Σ be a matrix such that $T_\Sigma = \{a_{ij} = \mathbb{P}(X_{t+1} = \sigma_j | X_t = \sigma_i)\}$. Then we say T_Σ is a transition probability matrix. Also note that notationally T_Σ can be written $T_\Sigma = \{a_{ij} = \mathbb{P}(\sigma_i \to \sigma_j)\}$.

We are now almost ready to rigorously define a Markov Chain, but we must first define what it means for a stochastic process to have the Markov Property.

Definition 2.4 (Markov Property)**.** Let $X = \{X_1, ..., X_T\}$ be a sequence of random variables. If for any positive integers $1 \leq k \leq t \leq T$ and for any choice of states $X_k = x_k, ..., X_t = x_t$ it is true that

$$\mathbb{P}(X_t = x_t | X_{t-1} = x_{t-1}, ..., X_k = x_k) = \mathbb{P}(X_t = x_t | X_{t-1} = x_{t-1}),$$

whenever the conditional probabilities are defined, i.e. whenever the conditioning events have positive probabilities[2], then the stochastic process $\{X_T\}$ is said to possess the Markov property.

Definition 2.5 (Markov chain)**.** Let $\{X_T\}$ be a stochastic process that possess the Markov property and that is parameterized by $\Sigma, T_\Sigma, \vec{\Pi}$, where Σ is the state space, T_Σ is the transition probability matrix, and $\vec{\Pi}$ is the initial distribution vector described by $\vec{\Pi} = \{\pi_i = \mathbb{P}(X_1 = \sigma_i)\}$. Then we say $\{X_T\}$ is a Markov chain. [6]

A Markov chain can be used either to analyze or generate a finite sequence of observations $X = \{X_1 = x_1, ..., X_T = x_T\}$. If we are analyzing an already-observed sequence of states, we are interested in the overall probability of observing that sequence.

Claim 2.6. Given a Markov chain parameterized by $\Sigma, T_\Sigma, \vec{\Pi}$, the total probability of observing a sequence of states $X = \{X_1 = x_1, ..., X_T = x_T\}$ is calculated as follows:

$$\pi_1 \prod_{t=1}^{T-1} a_{t,t+1}$$

where π_1 is the probability $\mathbb{P}(X_1 = x_1)$ and $a_{t,t+1} = \mathbb{P}(X_{t+1} = x_{t+1} | X_t = x_t)$.

Proof. Let $X = \{X_1 = x_1, ..., X_T = x_T\}$ be a sequence of states that adheres to the Markov chain parameterized by $\Sigma, T_\Sigma, \vec{\Pi}$. By the definition of conditional probability this can be rewritten as

$$\mathbb{P}(X_T = x_T | X_1 = x_1, ..., X_{T-1} = x_{T-1}) \mathbb{P}(X_1 = x_1, ..., X_{T-1} = x_{T-1}).$$

However, by the Markov property this is just

$$\mathbb{P}(X_T = x_T | X_{T-1} = x_{T-1}) \mathbb{P}(X_1 = x_1, ..., X_{T-1} = x_{T-1}).$$

By the parameters of the Markov chain, $\mathbb{P}(X_T = x_T | X_{T-1} = x_{T-1}) = a_{T-1,T}$, so this can be further rewritten as

$$a_{T-1,T} \mathbb{P}(X_1 = x_1, ..., X_{T-1} = x_{T-1}).$$

Because X is finite of length T, it follows that this process can be repeated until the following is obtained:

[2]We must include this last part to be safe: consider a case in which the probability of transitioning from $X_{t-1} = x_{t-1}$ to $X_t = x_t$ is 0, but the probability of transitioning from $X_t = x_t$ to $X_{t+1} = x_{t+1}$ is greater than 0. Then using the definition of conditional probability, we can write $\mathbb{P}(X_{t+1} = x_{t+1} | X_t = x_t, X_{t-1} = x_{t-1}, ..., X_k = x_k) = \frac{\mathbb{P}(X_{t+1}=x_{t+1}, X_t=x_t, X_{t-1}=x_{t-1}, ..., X_k=x_k)}{\mathbb{P}(X_t=x_t | X_{t-1}=x_{t-1}, ..., X_k=x_k)\mathbb{P}(X_{t-1}=x_{t-1}, ..., X_k=x_k)}$, and then by the Markov Property, we get $= \frac{\mathbb{P}(X_{t+1}=x_{t+1}, X_t=x_t, X_{t-1}=x_{t-1}, ..., X_k=x_k)}{\mathbb{P}(X_t=x_t | X_{t-1}=x_{t-1})\mathbb{P}(X_{t-1}=x_{t-1}, ..., X_k=x_k)} = \frac{\mathbb{P}(X_{t+1}=x_{t+1}, X_t=x_t, X_{t-1}=x_{t-1}, ..., X_k=x_k)}{0 \times \mathbb{P}(X_{t-1}=x_{t-1}, ..., X_k=x_k)}$ which is undefined.

$$a_{T-1,T}a_{T-2,T-1}...a_{1,2}\mathbb{P}(X_1 = x_1) = \mathbb{P}(X_1 = x_1)\prod_{t=1}^{T-1} a_{t,t+1}.$$

Then by $\vec{\Pi}$,

$$\mathbb{P}(X_1 = x_1, ..., X_T = x_T) = \pi_1 \prod_{t=1}^{T-1} a_{t,t+1}.$$

□

One more definition will help more concretely define the type of Markov chain that describes passwords:

Definition 2.7 (Reversible Markov Chain)**.** A Markov chain is said to be reversible if there is a probability distribution over states π such that

$$\pi_i\mathbb{P}(X_{t+1} = \sigma_j|X_t = \sigma_i) = \pi_j\mathbb{P}(X_{t+1} = \sigma_i|X_t = \sigma_j).$$

This condition is also known as *detailed balance*, and in terms of passwords, it essentially says that given two characters, say a, b, $\mathbb{P}(a \to b) = \mathbb{P}(b \to a)$, and for a Markov chain with only one transition matrix, this results in a symmetric matrix. It should be clear, however, that passwords likely do not follow this pattern, as one would expect $\mathbb{P}(a \to b) > \mathbb{P}(b \to a)$. In the example construction of a Markov chain for a fabricated password set listed later in this section, this is exactly the case.

Simply put, a Markov chain gives us information about transitioning from a current state to a new state, which letters in a word do. In order to apply Markov chains, though, we first need to know all the probabilities of transitioning from one state to another. To estimate these probabilities, we use observeable data, which in the case of passwords could be a *large* list of real-world passwords. The process of using data to find these transition probabilities is what "training" a Markov chain refers to, but this will be more rigorously defined later. In the language of passwords, if our current "state" is the letter b, a Markov chain "trained" with some data set will tell us the breakdown of probabilities for what the next most likely characters are according to the training set (i.e. the large list of real-world passwords). Markov chains are *memoryless*, however, which means that the only factor affecting the likelihood of a transition is the current state and not anything preceding it. Unfortunately this is somewhat problematic. For example, if we considered the characters in the word "abracadabra," where moving from each letter to the next is our transition, then given that our current state is d, it is clear that we are at the sixth transition (assuming that we don't consider being at the first letter a transition). Most english speakers could identify the whole word given just the string "abracad," but a Markov chain would be unable to consider the path leading to d or "abraca," offering us

instead the probability that d will transition to some other letter based on the distribution of letters after d in our training set. This happens because english words don't entirely exhibit the Markov property - intuitively, and using an abreviated notation from the definition 2.2, we cannot say that $\mathbb{P}(a|abracad) = \mathbb{P}(a|d)$ because we cannot ignore the extra information encoded in the start of the word.

While passwords and regular words do not necessarily exhibit the Markov property, we can still gain insight by treating them as though they do. For example, it is intuitive that vowels and consonants follow certain distributions where given a current state of "vowel," there is a high probability that the next state will be a consonant. In fact, Markov himself showed this by applying a Markov model to the text of *Eugene Onegin* in his native language, Russian[7]. If nothing else, letters in words *do* follow certain distribution probabilities, even if a simple Markov chain isn't the best at fleshing out the most accurate distribution. Nevertheless, we must start somewhere.

A constructed example will aid in showing how we can actually apply a Markov chain to passwords. Let's say that some site, *hackedsite.com*, enforces users to create passwords using elements drawn from Σ, and that we have obtained a list of real passwords from *hackedsite.com*. Perhaps this list of passwords looks like:

- $ababab$
- $12ab$
- $ab123$
- $a13b13c$
- $abcabc$
- $a1c2c3$
- $123abc$
- 112133223[3]
- $abc131$
- $a1b2c3$
- $b2a133$
- $a1b12c$

While these passwords are pretty arbitrary and therefore difficult to glean real information from, they provide a nice point to start thinking about passwords. For example, after a bit of head scratching it shouldn't be too difficult to recognize a few basic patterns. First, it is immediately noticeable that the term abc appears in many of the passwords[4]. The term 123 appears several times as well. These are broader patterns, but even on a character by character basis, patterns can be found. For example, the most common character after 1 is 2. However, 3 is also somewhat common to see after 1. The most common character after a is b, but another common character is 1. So on and so on. By now it should be becoming clear how we intend to use a Markov chain.

[3]Fun fact: this password is actually a DeBruijn sequence - it encodes all possible transitions from one number to another in a minimal string without repetition, if you wrap back to the beginning after the final 3!

[4]abc is fairly easy to remember, so it's not hard to believe someone would include it in a real password.

The first thing to establish is the state space with which we are working, which is precisely the characters $a, b, c, 1, 2, 3$. Elements in this state space transition from one to another, and can be visualized by the following:

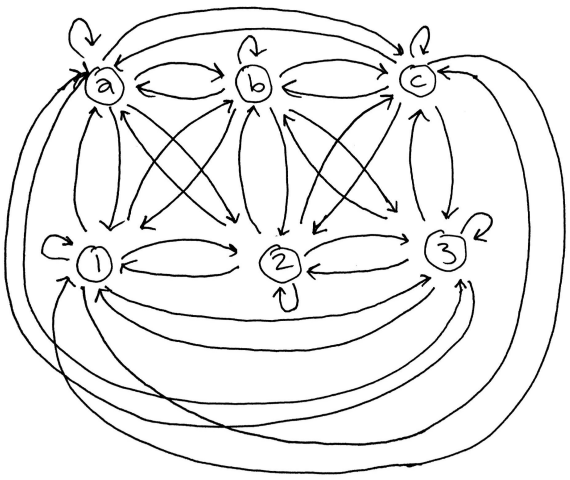

FIGURE 1. A decision making graph for $S = \{a, b, c, 1, 2, 3\}$

Figure 1 is intentionally left unweighted because it would be very crowded if the edges of the graph were labeled with their weights. More importantly, these weights, which are the probabilities of a given state transitioning to another (and in the case of the graph are assigned to the edges between two vertices, representing a transition from one vertex to the other), have not yet been estimated: they comprise the transition probability matrix T.

Now, assuming there existed another site *securesite.com* possessing the same state space Σ as *hackedsite.com* with a list of secret passwords, we can start using the information from *hackedsite.com* to assist us in guessing passwords, assuming the second site's users had similar internal methods for picking passwords[5]. Let's begin by assuming we want to start guessing passwords of length 6. The first thing we could do is to try guessing every possible password of length 6 from our state space Σ. Doing so would require $6^6 = 46,656$ guesses which is easy to do on a computer, but still more work than desired (and which, in a realistic setting of a state space closer to 94 characters, would be *much* larger). Instead, it makes sense to optimize which passwords we guess first by using the data we have to train our decision making. We begin to start training our model, i.e. using data to abstract the

[5]This is a fairly good assumption if we train our model with a large, realistic data set.

general relationships that will make up the parameters $\vec{\Pi}$ and T. To estimate $\vec{\Pi}$ we tally up all the first characters. From this we observe that most of the passwords (8/12 or 67%) from *hackedsite.com* begin with a, and so we can infer that passwords from *securesite.com* might also begin with a a large percentage of the time. It makes sense, then, to start guessing passwords that begin with a. From here, we want to choose the next best letter to try, so we tally up all transitions that occur in the list of passwords to find which transitions are most likely. Note that the last letter of a password does not make a transition, so we only consider character transitions up to that point. For clarity, the number of times each transition takes place is placed in a matrix A:

$$A = \begin{array}{c} \\ a \\ b \\ c \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{cccccc} a & b & c & 1 & 2 & 3 \end{array} \\ \begin{pmatrix} 0 & 9 & 0 & 4 & 0 & 0 \\ 2 & 0 & 4 & 3 & 2 & 0 \\ 1 & 0 & 0 & 1 & 1 & 2 \\ 0 & 2 & 1 & 1 & 5 & 5 \\ 2 & 0 & 3 & 1 & 1 & 3 \\ 1 & 1 & 1 & 2 & 1 & 2 \end{pmatrix} \end{array}$$

This matrix gives the overall number of times a particular transition is made, where rows represent the current state, the columns represent the state to which we are transitioning. To put this in terms of probability (the weights of the edges of the graph in Figure 1), we sum the entries in each row, and then each entry is divided by the sum of its row. Our new matrix becomes our transition matrix:

$$T_\Sigma = \begin{array}{c} \\ a \\ b \\ c \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{cccccc} a & b & c & 1 & 2 & 3 \end{array} \\ \begin{pmatrix} 0 & 0.69 & 0 & 0.31 & 0 & 0 \\ 0.18 & 0 & 0.37 & 0.27 & 0.18 & 0 \\ 0.2 & 0 & 0 & 0.2 & 0.2 & 0.4 \\ 0 & 0.14 & 0.07 & 0.07 & 0.36 & 0.36 \\ 0.2 & 0 & 0.3 & 0.1 & 0.1 & 0.3 \\ 0.125 & 0.125 & 0.125 & 0.25 & 0.125 & 0.25 \end{pmatrix} \end{array}$$

We now have a transition matrix as discussed in definition 2.2. Moreover, this matrix is *learned* from data provided by *hackedsite.com* and will hopefully allow us to make better guesses in the future.

Now that we have trained our transition matrix T using data, we return to our initial goal, which is to optimize the number of passwords we can guess in a limited amount of time. Since longer passwords created from larger character spaces will require exponentially more guesses to the point that it is infeasible to try all combinations, we want to make only the most

sensible guesses with the recognition that some passwords may not be found. However, the goal is to correctly guess as many as possible within some sort of time constraint. Again, since we decided to begin guessing passwords that start with a, our new transition matrix tells us that a is most likely to transition to b, which is most likely to transition to c, etc.. If we follow through this process, we end up with several passwords that at first glance appear equally probable:

- $abc312$
- $abc313$
- $abc331$
- $abc333$

If we actually use the values from our transition probability matrix to calculate the probability of these passwords, however, we see that the first two passwords are about 1.5 times more likely. We can also see this by looking at our transitions on what is called a weighted tree, which is similar in function to the weighted graph discussed earlier. This time, however, each "branch" in the tree represents a transition, and where there are more than one "branch," we must make a decision as to where we will transition. The weights on the tree define the conditional probabilities from T and direct us in our decision making (we should generally pick the most probable decision):

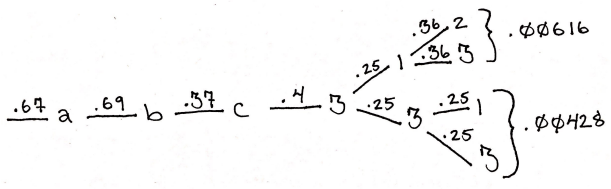

FIGURE 2. A weighted decision tree for our Markov chain

It was stated that the password combinations we obtained using T looked equally probable, but that they were in fact not. The unequal probabilities happen because even though transitioning either to 1 or 3 when we are at $abc3$ is equally probable, the subsequent transitions are not equally probable - a transition from 1 to a 2 or 3 occurs with a probability of 0.36, while a transition from a 3 to either a 3 or a 1 has a probability of 0.25. In fact, when we try to start picking some of the next most probable passwords we run into a problem - at which points in our decision tree do we decide to pick the second most probable option? Should we veer off immediately, say by transitioning from $a \to 1$? Or should we choose to branch out closer to the end, say by taking the path $abc31b$, where b represents the second most probable outcome after 3?

Unfortunately, finding the most probable strings now becomes a matter of following every branch of the decision tree to the end and then comparing the probabilities of the resulting strings with each other. While this is easily done for our example using a computer, in the case of a large character space with longer passwords, this turns into having to compute billions and billions of probabilities and storing them in memory. Our hope had been that using a Markov chain would return the most likely passwords first without using up lots of time and physical resources. What would be need to continue this path, other than more computing power, is an efficient algorithm for computing the n most probable strings generated by a Markov chain. As is, a Markov chain with parameters estimated from large data sets will be used to estimate the probabilities of observing various passwords in section 4, which we will use to affirm our intuitive notion that passwords like "Kittycat123!" are in fact more likely to be observed in real life. Additionally, by estimating the probability of observing a password in real life something can also be said about the potential security of a password, with the idea being that a "less likely" password is less likely to be guessed in a targeted attack that uses one of the methods described in this paper.

2.2. **Markov Chains with Memory.** As was mentioned in 2.1, Markov chains are generally considered memoryless, and this is a problem for us because we intuitively know that words, which are most often used to create passwords, do not exhibit the Markov property. Thankfully, there are forms of Markov chains that introduce a form of memory.

Definition 2.8. We say a Markov chain has "memory of length m" if it satisfies the property

$$\begin{aligned}&\mathbb{P}(X_t = x_t | X_{t-1} = x_{t-1}, X_{t-2} = x_{t-2}, ..., X_1 = x_1)\\ =&\mathbb{P}(X_t = x_t | X_{t-1} = x_{t-1}, X_{t-2} = x_{t-2}, ..., X_{t-m} = x_{t-m}) \texttt{ for } t > m\end{aligned}$$

As before, we are interested in calculating the probability of a given password, assuming password creation can be reasonably modeled by a Markov chain with memory (also referred to as a *higher order* Markov Chain in some places). Then let X be a sequence of states $X = \{X_1 = x_1, ..., X_T = x_T\}$ that can be modeled by a Markov chain with memory m, and define $k = T - m$. By conditional probability this can be rewritten as

$$\mathbb{P}(X_T = x_T | X_{T-1} = x_{T-1}, ..., X_1 = x_1)\mathbb{P}(X_{T-1} = x_{T-1}, ..., X_1 = x_1).$$

Because the Markov chain has memory m, this can be rewritten as

$$\mathbb{P}(X_T = x_T | X_{T-1} = x_{T-1}, ..., X_k = x_k)\mathbb{P}(X_{T-1} = x_{T-1}, ..., X_1 = x_1).$$

It follows that this process can be repeated until the following, written in shorthand to be concise where $X_i = x_i$ is just abbreviated X_i, is obtained:

$$\mathbb{P}(X_T|X_{T-1},...,X_k)\mathbb{P}(X_{T-1}|X_{T-2},...,X_{k-1})...\mathbb{P}(X_m,X_{m-1},...,X_1)$$

To calculate these values, it is necessary to know the probabilities of transitioning to each state given every possible permutation of preceding m states, or $\mathbb{P}(X_i = x_i|X_{i-1} = x_{i-1},...,X_{i-m} = x_{i-m})$, which for a state space with n characters, requires knowing n^m different probabilities. To more easily understand this, we can use a change of variables where we define each permutation $\hat{X}_j = (X_{i-1} = x_{i-1},...,X_{i-m} = x_{i-m})$. We can now think of the $\hat{x}$'s as being new states that transition to states in Σ.

Example 2.9. Consider the sequence *abcde*, and assume it is being modeled by a Markov chain with memory $m = 2$. Then the $\hat{x}$'s are every possible permutation from $\Sigma = \{a, b, c, d, e\}$ of length 2, and the observed $\hat{X}$'s are $\hat{X}_1 = ab, \hat{X}_2 = bc, \hat{X}_3 = cd$, where each $\hat{x}$ can be thought of its own new character. Moreover, the transitions are $ab \to c, bc \to d, cd \to e$. Also note that despite the fact that in a regular Markov chain this sequence would have 4 transitions, for a Markov chain of length 2 it has only 3. In general, there are $T - m$ total transitions.

As has been noted, for a Markov chain with state space Σ of size n and with memory m, there are n^m $\hat{x}$'s that need to be calculated. Effectually, this causes the transition matrix T to grow in size as well. Whereas with a regular Markov chain T was $n \times n$, the transition matrix T for a Markov chain with memory will be $n^m \times n$. However, similar to regular Markov chains, a value a_{ij} in the matrix describes the probability of transitioning from $\hat{x}_i$ to an element in the state space σ_j. Because of this overall increase in complexity, Markov chains with memory being modeled on computers are generally restricted to lower values for memory. As will be discussed in section 4, a Markov chain with $m = 2$ for a full 94 character state space is sufficient to cause significant calculation wait times.[6] It should also be noted that $\vec{\Pi}$ will change slightly, as now the first "character" is a tuple, and so $\hat{\vec{\Pi}}$ must be redefined as $\hat{\vec{\Pi}} = \{\hat{\pi}_i = \mathbb{P}(\hat{X}_1 = \hat{x}_i) = \mathbb{P}(X_m = x_m,...,X_1 = x_1)\}$

Putting together this information, we can calculate the probability of a sequence $X = \{X_1 = x_1,...,X_T = x_T\}$ the same way we do for regular Markov chains:

$$\hat{\pi}_i \prod^{T-m} a_{t,t+1}.$$

2.3. **Indexed Markov Chains.** There is yet another component of Markov chains that has been assumed to be true, but that upon inspection of passwords seems unlikely. Namely, this is the assumption that the state transition

[6]At least this was true for how the author implemented this. Perhaps there is a more clever way to do so.

matrix T is the same regardless of time. When we are considering passwords though, we can observe trends that contradict this. For example, when a password contains a capital letter, overwhelmingly that capital letter is the first letter in the word. Also, when it contains a number or special character, that number or character is usually found at the very end of the password [3]. This leads us to think that perhaps it is faulty to assume T does not change with time since it does not appear that the same distribution governs letters and numbers at the beginning and at the end of a password. To continue improving our simple Markov chain, we adapt it to make use of different transition matrices based on different time indices as we transition from character to character. Overall the Markov chain operates the same, except for each index $t = 2, 3, 4...$ a different transition matrix, trained from data, is used. Visually this can be thought of using the following diagram (borrowed from [10]):

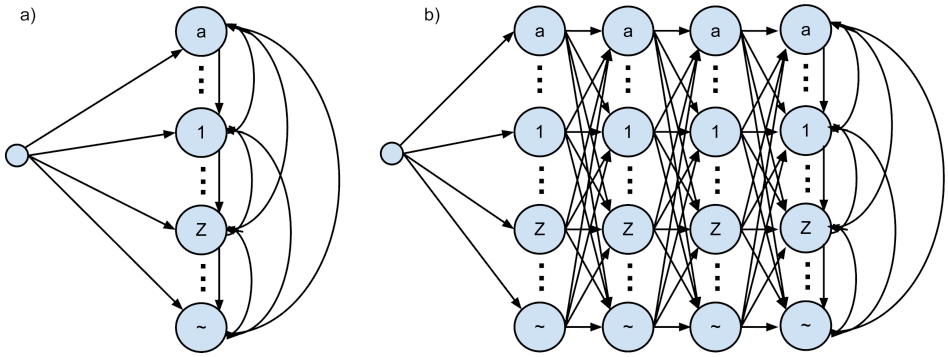

FIGURE 3. Regular vs Indexed Markov Chain

On the left is a regular Markov chain, and on the right an indexed Markov chain. Notably, the extra layers in the picture of the indexed Markov chain represent separate T matrices for each index.

It will help further in understanding this concept to introduce a *time homogeneous Markov chain*:

Definition 2.10. A Markov chain is said to be time-homogeneous (or stationary) if for all $2 \leq t \leq T-1$ it is true that

$$\mathbb{P}(X_{t+1} = \sigma_j | X_t = \sigma_i) = \mathbb{P}(X_t = \sigma_j | X_{t-1} = \sigma_i).$$

It follows then that a Markov chain with different transition matrices for different indices is *not* time-homogeneous.

There are several different ways to implement an indexed Markov chain, as will be discussed in section 4. One method is train the transition matrices on a set of sequences of a fixed length (eg passwords with 8 characters),

which will more accurately describe sequences of that length. Another is to predetermine how many transition matrices will be created and then use sequences of different lengths to train those matrices. In general, this should cause the matrices to more accurately reflect different trends about average-lengthed passwords, but the problem is that transition matrices for higher indices will have less data with which to be trained because passwords are an average of only 9.6 characters [3].

3. Hidden Markov Models

So far we have looked into Markov chains in hopes of finding an efficient algorithm that can tell us which character sequences are more worthwhile to guess as potential passwords over others. While a simple Markov chain showed some promise, it presented several issues in that finding the n most probable character sequences was a difficult problem that required as much work as guessing all possible passwords (at least, we haven't found a more efficient way to do this yet - which does not mean there *isn't* an elegant solution). As such, we now turn to another form of Markov model known as a Hidden Markov Model (HMM for short). Hidden Markov models are used very often in linguistics because they're exceptional at modeling sequential data (much like characters in a password) and because they can be used to flesh out certain properties that might be hidden within data (which sounds pretty good when we're thinking about passwords). One notable field in which HMMs have found great success, just like Markov chains, is linguistics, where they're used in many different predictive speech systems for tagging parts of speech. While it is not totally obvious yet how an HMM might be applied to the overall goal of this thesis, they seem closely enough related to the topic that they seem like a worthwhile approach.

3.1. **Hidden Markov Models.**

A hidden Markov model can be visualized nicely using what is called a *Trellis diagram*, which shows an ordered sequence of nodes. Before a rigorous definition of hidden Markov models, consider first a case similar to that discussed in 2.1, where we have a set of states capable of transitioning between each other. This time, however, the states are not the sole object of focus because they're actually "hidden", or unobservable. Instead, the states give off or emit certain observations, which is the part of the system we can see. Such a system can be visualized in this trellis graph:

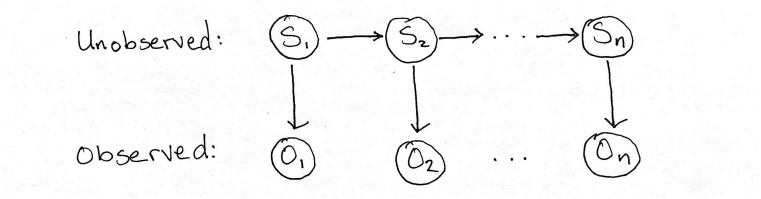

FIGURE 4. Trellis diagram for hidden states $s_1, ..., s_n$ and emitted observations $o_1, ..., o_n$

Here, we see a sequence of states that remain unobserved and a sequence of observations that give us some information about the hidden states. To be precise, we can use the probabilities that certain states emit various observations to actually discover quite a bit about the hidden sequence of states. Now let's be a little more precise about what an HMM really is.

To define a Hidden Markov model, we need several things. First, we need to know that whatever we are modeling transitions between a finite number of states, and that each state gives some sort of emission(s) which can be observed, and which come from a finite set of possible emissions. This part of the definition is unavoidably vague because these states and emissions can take a multitude of forms and really must be evaluated on a case-by-case basis. For example, states could consist of whether it is raining or sunny, and emissions might be whether a person feels safe or unsafe driving to the grocery store given the weather (this example, or variations of it are used quite often [9][8]). We also need some way to know the probabilities of any state transitioning to another, as well as the probability of observing any particular emission given any hidden state. These will later be defined in terms of transition and emission matrices, and it will necessarily be true that each entry in the matrix is non-negative and each row sums to 1 (i.e. the matrix must be stochastic).. Finally, we will need to know the distribution of initial states, that is, the probability that any given hidden state is the first to occur.

Hidden Markov models also assume the Markov property discussed in section 2.1, although the property must be slightly ammended to account for the added complexity of HMMs.

Definition 3.1 (Markov Property for Hidden Markov Models)**.** Let $\Sigma = \{\sigma_1, ..., \sigma_m\}$ be a state space and let $E = \{\epsilon_1, ..., \epsilon_n\}$ be a set of emissions. Let $O, S = (O_1 = o_1, ..., O_T = o_T, S_1 = s_1, ..., S_T = s_T)$ be a sequence of hidden states and emissions with $s_i \in \Sigma$ and $o_j \in E$. Then if for O, S it is true that

$$\begin{aligned}&\mathbb{P}(S_t = s_t | S_{t-1} = s_{t-1}, ..., S_1 = s_1, O_{t-1} = o_{t-1}, ..., O_1 = o_1)\\ =&\mathbb{P}(S_T = s_t | S_{t-1} = s_{t-1})\end{aligned}$$

and

$$\begin{aligned}&\mathbb{P}(O_t = o_t | S_t = s_t, ..., S_1 = s_1, O_{t-1} = o_{t-1}, ..., O_1 = o_1)\\ =&\mathbb{P}(O_t = o_t | S_t = s_t),\end{aligned}$$

we say O, S possesses the Markov Property for Hidden Markov Models.

Definition 3.2. [8] A Hidden Markov Model can be characterized by five components:

1. m - the total number of possible hidden states that can exist. Define the set of all possible hidden states as $\Sigma = \{\sigma_1, ..., \sigma_m\}$.
2. n - the total number of possible emissions that can be emitted from hidden states in Σ. Define the set of all emissions as $E = \{\epsilon_1, ..., \epsilon_n\}$.
3. $T_\Sigma = \{a_{ij} = \mathbb{P}(\sigma_i \to \sigma_j)\}$ for $1 \leq i, j \leq m$ - the transition likelihood matrix describing the likelihood that state σ_i transitions to σ_j. Note that this information is stored in a matrix for ease of use and readability, but T_Σ is not treated like a matrix, i.e. only single entries are considered from it at a time. Also note that because the model is assumed to possess the Markov property, the probability of transitioning from one state to another depends solely on the two states, and not on any preceding sequence of states[7].
4. $T_E = \{b_{ij} = \mathbb{P}(\epsilon_j | \sigma_i)\}$ for $1 \leq i \leq m$ and $1 \leq j \leq n$ - the emission likelihood matrix describing the likelihood that a hidden state σ_i emits the observation ϵ_j. Again, this matrix is for ease of use and readability. Furthermore, it is true that the likelihood of any emission ϵ_j is observed is dependent solely on the current hidden state σ_i.
5. $\vec{\Pi} = \{\pi_i = \mathbb{P}(S_1 = \sigma_i)\}$ where S_1 is the first hidden state in the sequence generated by the model for $1 \leq i \leq m$.

Notationally, let $\lambda = \{\vec{\Pi}, T_\Sigma, T_E\}$ represent the Hidden Markov Model. The parameters m, n need not be included in this notation because they are implicit in the definitions of T_Σ and T_E.

These five parameters are used to generate a sequence of random variables O representing observations such that $O = \{O_1 = o_1, ..., O_T = o_T\}$ where $o_i \in E$ and T represents the length of the sequence, along with a sequence of random variables S representing hidden states such that $S = \{S_1 = s_1, ..., S_T = s_T\}$ with $s_i \in \Sigma$ according to the following procedure:

[7]Also note that we are assuming for simplicity's sake that the process is time-homogeneous

a. Pick a value s_1 from Σ for S_1 according to the initial distrubution vector $\vec{\Pi}$.
b. Choose an emission o_1 from E for O_1 according to the emission probability matrix T_E.
c. Transition to a new state $S_2 = s_2$ according to the transition probability matrix T_Σ.
d. Repeat steps b, c, advancing the index by one each time until $O_T = o_T$ is reached.

Now that we know what an HMM is, we are interested in finding the probability of any coupling of sequences O, S given $\lambda = \{\vec{\Pi}, T_\Sigma, T_E\}$.

Claim 3.3 ([8]). Let $\lambda = \{T_\Sigma, T_E, \Pi\}$ define a Hidden Markov Model. Then the probability of a given pair of observation and hidden state sequences O, S, each of length T, is calculated as

$$\mathbb{P}(O,S) = \mathbb{P}(S_1 = s_1)\mathbb{P}(O_1 = o_1|S_1 = s_1)\prod_{k=2}^{T}\big(\mathbb{P}(S_k = s_k|S_{k-1} = s_{k-1})\mathbb{P}(O_k = o_k|S_k = s_k)\big).$$

Proof. Consider that $\mathbb{P}(O,S) = \mathbb{P}(O_1 = o_1, ..., O_T = o_T, S_1 = s_1, ..., S_T = s_T)$. For the sake of notational ease (and space), we will write only the random variable O_i or S_j instead of $O_i = o_i$ or $S_j = s_j$ in this proof. Then $\mathbb{P}(O,S) = \mathbb{P}(O_1, ..., O_T, S_1, ..., S_T)$. The first step in the proof is to recognize that the Markov property allow and the definition of conditional probability allow us to rewrite this:

$$\begin{aligned}
&\mathbb{P}(O_1, ..., O_T, S_1, ..., S_T) \\
&\qquad = \mathbb{P}(O_T|S_T, ..., S_1, O_{T-1}, ..., O_1)\mathbb{P}(S_T, ..., S_1 O_{T-1}, ..., O_1) \\
&\qquad = \mathbb{P}(O_T|S_T)\mathbb{P}(S_T, ..., S_1, O_{T-1}, ..., O_1) \\
&\qquad = \mathbb{P}(O_T|S_T)\mathbb{P}(S_T|S_{T-1}, ..., S_1, O_{T-1}, ..., O_1)\mathbb{P}(S_{T-1}, ..., S_1, O_{T-1}, ..., O_1) \\
&\qquad = \mathbb{P}(S_{T-1}, ..., S_1, O_{T-1}, ..., O_1)\mathbb{P}(S_T|S_{T-1})\mathbb{P}(O_T|S_T).^{8}
\end{aligned}$$

Again, the Markov property allows us to further break up this sequence:

$$\mathbb{P}(S_{T-2}, ..., S_1, O_{T-2}, ..., O_1)\mathbb{P}(S_{T-1}|S_{T-2})\mathbb{P}(O_{T-1}|S_{T-1})\mathbb{P}(S_T|S_{T-1})\mathbb{P}(O_T|S_T).$$

It follows that this process can be repeated until we are left with

$$\mathbb{P}(O,S) = \mathbb{P}(O_1, S_1)\mathbb{P}(S_2|S_1)\mathbb{P}(O_2|S_2)...\mathbb{P}(S_T|S_{T-1})\mathbb{P}(O_T|S_T).$$

However this is just

$$\mathbb{P}(S_1)\mathbb{P}(O_1|S_1)\prod_{k=2}^{T}\big(\mathbb{P}(S_k|S_{k-1})\mathbb{P}(O_k|S_k)\big),$$

which aside from notation is identical to what was stated in claim 3.2.

□

We can immediately ask ourselves a few questions about HMMs:

i) How can an HMM be used in the context of passwords?
ii) How do we know the state transition and observation emission probabilities?
iii) Given a sequence of observed emissions, how do we find the most probable sequence of underlying states?
iv) If we don't know much about our system, how do we know what parameters to choose so that we can approximate the true underlying model, given that we can't observe the hidden states directly (we can't even be guaranteed we know how many or of what type they are)?

The first of these questions has no concrete answer, because an HMM could potentially be applied to passwords in a variety of ways. For example, hidden states could be different types of words or sequences commonly used in passwords, like names, nouns, places, etc. with associated observations like "Alice," "banana," and "Paris." Hidden states may also represent different sites where passwords are created, while the observations are actually passwords generated at that site (for example, it's not hard to imagine that passwords associated with a botanical site are botanical in nature). Only trial and error will show which applications of an HMM to passwords may be most useful. The second question in contrast is fairly simple - we get an approximation of these values the same way we found the similar matrix of transition probabilities for a Markov chain, by using large data sets. The third and fourth questions require significantly more work, and answering them is the natural next step in trying to see how an HMM might be useful in pursuit of this thesis's goal.

3.2. **Viterbi Algorithm.** We now know how to talk about HMMs but we are still left with unanswered questions as discussed in the previous section. The first of these questions we'll tackle is being able to find the most probable sequence of hidden states given a sequence of emissions. At first glance, it may be tempting to simply calculate the probability of all possible sequences of hidden states given a sequence of observations, but this is an unwieldy beast. Since our sequence of hidden states has length T and there are m possible hidden states, there are m^t total sequences for which we must calculate the probability. While it's hard to say exactly how difficult comparing the likelihood of all sequences might be without explicitly defining the states and the length of a sequence, it's fairly easy to see how m^T can grow very quickly, even for small m. Instead, we use the *Viterbi Algorithm* to optimize this process. The motivation is fairly straightforward; instead of calculating the probability of all sequences up to a point, we care only about the most probable sequence up to a point. By doing this, we greatly reduce the number of sequences that require our attention.

Claim 3.4. Given $\lambda = \{T_\Sigma, T_E, \Pi\}$ with m hidden states and some generated sequence of emissions $O = \{O_1 = o_1, ..., O_T = o_T\}$, define

$$x_{1j} = \mathbb{P}(S_1 = \sigma_j)\mathbb{P}(O_1 = o_1 | S_1 = \sigma_j)$$

where $\sigma_j \in \Sigma$ and $o_1 \in E$, and define

$$x_{ij} = max\{x_{i-1,1}\mathbb{P}(\sigma_1 \to \sigma_j)\mathbb{P}(O_i = o_i|S_i = \sigma_j), ..., x_{i-1,m}\mathbb{P}(\sigma_m \to \sigma_j)\mathbb{P}(O_i = o_i|S_i = \sigma_j)\}$$

for $2 \leq i \leq n$ and $1 \leq j \leq m$. Further define $\mathbb{S}_i$ as the hidden state that maximizes $\{x_{i1}, ..., x_{im}\}$ for $1 \leq i \leq n$ (i.e. $\mathbb{S}_i = argmax\{x_{i1}, ..., x_{im}\}$). Then the generated sequence $\mathbb{S} = \mathbb{S}_1, ..., \mathbb{S}_n$ is the sequence that maximizes the overall likelihood of S given O, λ.

The following sketch of the proof that the Viterbi algorithm produces what it claims to can be used to gain a good intuition as to why it is true.[9] Consider the last hidden state and its associated observation in O, S. For the last observation $O_T = o_T$, we know that there are m different possible hidden states $\sigma_j \in \Sigma$ for $1 \leq j \leq m$ that might have emitted o_T, and for each of these potential states there is some path terminating in σ_j that maximizes probability up to that point. By studying both the probability of the best path up to any given σ_j with the likelihood that σ_j emits the observation o_T, we can find the overall most probable path, or the sequence $S_1 = s_1, ..., S_T = s_T$ such that given $O_1 = o_1, ..., O_T = o_T$, the probability $\mathbb{P}(S|O)$ is maximized. Pictorally, this looks like.[10]

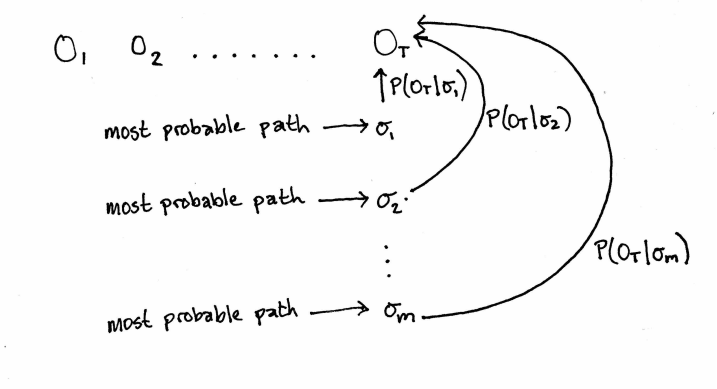

FIGURE 5. Pictoral representation of the best path up to $O_T = o_T$

Hidden within this picture is an important detail that needs to be proven before this can be taken as true.

[9] A full proof of the Viterbi algorithm's correctness can be found in [2]

[10] The following pictoral representations were inspired by and adapted from [9]

Claim 3.5. The sequence $S_1 = s_1, ..., S_{t+1} = \sigma_i$ that maximizes $\mathbb{P}(S_1 = s_1, ..., S_{t+1} = \sigma_i | O_1 = o_1, ..., O_{t+1} = o_{t+1})$ for some σ_i comes from the sequence $S_1 = s_1, ..., S_t = \sigma_j$ that maximizes $\mathbb{P}(S_1 = s_1, ..., S_t = \sigma_j | O_1 = o_1, ..., O_t = o_t)$ for some σ_j[11].

Proof. Assume otherwise. Then the sequence $S_1 = s_1, ..., S_{t+1} = \sigma_i$ that maximizes $\mathbb{P}(S_1 = s_1, ..., S_{t+1} = \sigma_i | O_1 = o_1, ..., O_{t+1} = o_{t+1})$ comes from something other than the sequence $S_1 = s_1, ..., S_t = \sigma_j$ that maximizes $\mathbb{P}(S_1 = s_1, ..., S_t = \sigma_j | O_1 = o_1, ..., O_t = o_t)$. Namely, there is some sequence $S'_1 = s'_1, ..., S'_t = \sigma_j$ such that $\mathbb{P}(S'_1 = s'_1, ..., S'_t = \sigma_j | O_1 = o_1, ..., O_t = o_t)$ is not maximized, but such that $\mathbb{P}(S'_1 = s'_1, ..., S'_t = \sigma_j, S_{t+1} = \sigma_i | O_1 = o_1, ..., O_{t+1} = o_{t+1})$ is maximized. By construction $\mathbb{P}(S'_1 = s'_1, ..., S'_t = \sigma_j, S_{t+1} = \sigma_i | O_1 = o_1, ..., O_{t+1} = o_{t+1}) = \mathbb{P}(S'_1 = s'_1, ..., S'_t = \sigma_j | O_1 = o_1, ..., O_t = o_t)\mathbb{P}(\sigma_j \to \sigma_i)\mathbb{P}(O_{t+1} = o_{t+1} | S_{t+1} = \sigma_i)$. However, this cannot be maximum because there exists a sequence $S_1 = s_1, ..., S_t = \sigma_j$ such that $\mathbb{P}(S_1 = s_1, ..., S_t = \sigma_i | O_1 = o_1, ..., O_t = o_t) > \mathbb{P}(S'_1 = s'_1, ..., S'_t = \sigma_j | O_1 = o_1, ..., O_t = o_t)$, which implies that $\mathbb{P}(S'_1 = s'_1, ..., S'_t = \sigma_j | O_1 = o_1, ..., O_t = o_t)\mathbb{P}(\sigma_j \to \sigma_i)\mathbb{P}(O_{t+1} = o_{t+1} | S_{t+1} = \sigma_i) < \mathbb{P}(S_1 = s_1, ..., S_t = \sigma_j | O_1 = o_1, ..., O_t = o_t)\mathbb{P}(\sigma_j \to \sigma_i)\mathbb{P}(O_{t+1} = o_{t+1} | S_{t+1} = \sigma_i)$, which is a contradiction. Then it follows that the sequence $S_1 = s_1, ..., S_{t+1} = \sigma_i$ that maximizes $\mathbb{P}(S_1 = s_1, ..., S_{t+1} = \sigma_i | O_1 = o_1, ..., O_{t+1} = o_{t+1})$ must come frome the sequence $S_1 = s_1, ..., S_t = \sigma_j$ that maximizes $\mathbb{P}(S_1 = s_1, ..., S_t = \sigma_j | O_1 = o_1, ..., O_t = o_t)$ for some σ_j □

Now all that remains is to actually find these best paths up to each $\sigma_i \in \Sigma$ that may have emitted o_T. To do this, we step back one transition, so to speak, because we know that each σ_i associated with o_T was transitioned to from some other $\sigma \in \Sigma$ associated with o_{T-1}. By doing this, we hope to find which σ is the most probable previous step, given o_{T-1}. Again, we can use the following picture to understand this:

[11]This can be thought of as the most likely path through o_{t+1} must have its first t entries be the most likely path through o_t

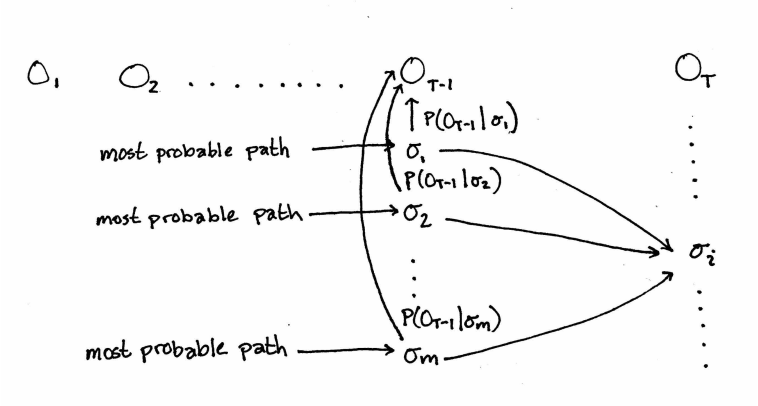

FIGURE 6. Pictoral representation of the best path up to $O_{T-1} = o_{T-1}$.

By iterating this process, we find the best path up to each point. It follows then, that if we store only the most probable path up to each point (whose probability is stored in the $x_{i,j}$'s) and we pick the most probable at each step, we will have maximized the likelihood for S to occur given O. Retranslated into our original language, this whole process can also be shown pictorally:

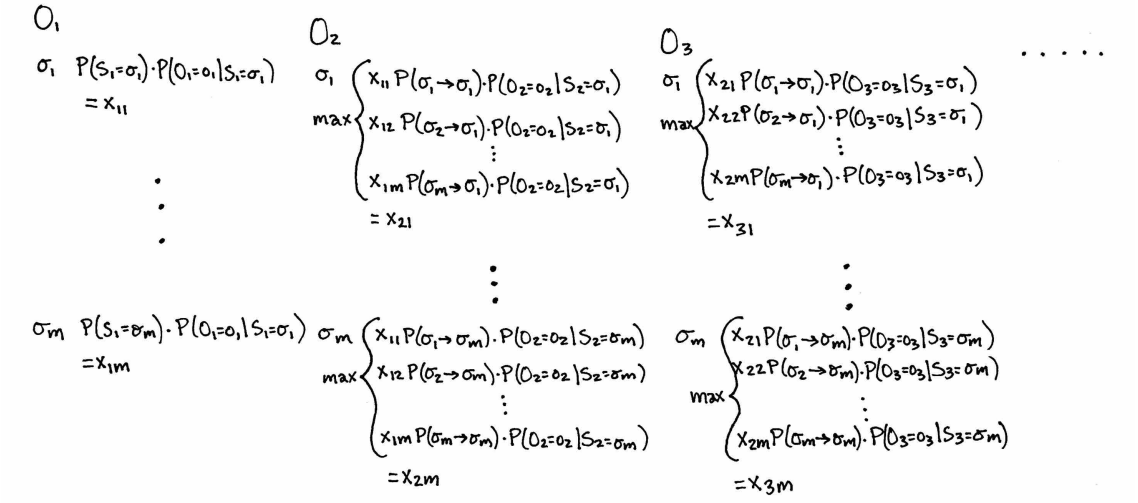

FIGURE 7. Finding the $x_{i,j}$'s.

3.3. **Forward/Backward and Baum-Welch Algorithms.** We are now much better off than before because given a sequence of observed states,

we have a recursive method to find the most probable sequence of hidden states. However, this is only part of a picture that would be completed by answering question *iii*) from 3.1. Namely, if we don't know much about the underlying system, there's no easy way to learn about it because the underlying states are unobserved. To remedy this situation, we can use the *Baum-Welch* algorithm, which can be used to optimize our HMM parameters in a way that tells us more about them. To understand the Baum-Welch algorithm, understanding of two additional algorithms is first necessary.

3.3.1. *Forward Algorithm.* The *Forward Algorithm* (and for that matter the *Backward Algorithm*) are slightly confusing in that their names almost seem to be reversed. In the case of the Forward algorithm, it tells us the joint probability of seeing a sequence of observations prior to the current state being some σ_i, or $\mathbb{P}(O_1 = o_1, ..., O_t = o_t, S_t = \sigma_i)$. Even though the Forward Algorithm looks at events that have already happened, it's named appropriately because the recursive method to find these probabilities iterates *forward in time*, starting with $t = 1$ and ending at $t = T$.

Define $\alpha_i(t) = \mathbb{P}(O_1 = o_1, ..., O_t = o_t, S_t = \sigma_i)$ for $\sigma_i \in \Sigma$. Then the algorithm for finding $\mathbb{P}(O_1 = o_1, ..., O_t = o_t, S_t = \sigma_i)$ as well as $\mathbb{P}(O|\lambda)$ – the total probability of observing O relative to all possible S – is as follows:[8]

Algorithm 1 Forward Algorithm[8]

1. Let $\alpha_i(1) = \pi_i b_{ik}$, for $1 \leq i \leq m$, where b_{ik} is the likelihood that the hidden state s_i emits the observation $o_1 = \epsilon_k \in E$.
2. Compute $\alpha_i(t+1) = b_{ik} \sum_{j=1}^{m} \alpha_j(t) a_{ji}$, for $1 \leq t \leq T-1$ and for $O_{t+1} = \epsilon_k$.
3. Then $\mathbb{P}(O|\lambda) = \sum_{j=i}^{m} \alpha_j(T)$.

Proof. It must be shown that $\alpha_i(t) = \mathbb{P}(O_1 = o_1, ..., O_t = o_t, S_t = \sigma_i)$ for all $1 \leq t \leq T$ and that $\mathbb{P}(O|\lambda) = \sum_{j=1}^{m} \alpha_j(T)$. To do this, we will use induction.

- ◊ *Base case*: By definition $\alpha_i(1) = \pi_i b_{ik}$. Furthermore, $\mathbb{P}(O_1 = o_1, S_1 = \sigma_i) = \mathbb{P}(S_1 = \sigma_i)\mathbb{P}(O_1 = o_1|S_1 = \sigma_i)$, and since $o_1 = \epsilon_k \in E$, it follows that $\mathbb{P}(O_1 = o_1, S_1 = \sigma_i) = \pi_i b_{ik}$. Then $\alpha_i(1) = \mathbb{P}(O_1 = o_1, S_1 = \sigma_i)$.
- ◊ *Inductive Assumption*: Assume $\alpha_j(t) = \mathbb{P}(O_1 = o_1, ..., O_t = o_t, S_t = \sigma_j)$.
- ◊ *Inductive Step*: Consider $\mathbb{P}(O_1 = o_1, ..., O_{t+1} = o_{t+1}, S_{t+1} = \sigma_i)$. Because hidden states are disjoint, then by the distributive law for sets we can rewrite $\mathbb{P}((S_t = \sigma_1 \cup ... \cup S_t = \sigma_m) \cap (O_1 = o_1, ..., O_{t+1} = o_{t+1}, S_{t+1} = \sigma_i))$ which becomes $\mathbb{P}\Big(((O_1 = o_1, ..., O_{t+1} = o_{t+1}, S_{t+1} =$

$\sigma_i) \cap S_t = \sigma_1\big) \cup ... \cup \big((O_1 = o_1, ..., O_{t+1} = o_{t+1}, S_{t+1} = \sigma_i) \cap S_t = \sigma_m\big)\Big)$. But this is just the following sum over j:

$$= \sum_{j=1}^{m} \mathbb{P}(O_1 = o_1, ..., O_t = o_t, S_t = \sigma_j, O_{t+1} = o_{t+1}, S_{t+1} = \sigma_i).$$

By the definition of conditional probability, this tells us that

$$= \sum_{j=1}^{m} \mathbb{P}(O_1 = o_1, ..., O_t = o_t, S_t = \sigma_j) \mathbb{P}(O_{t+1} = o_{t+1}, S_{t+1} = \sigma_i | O_1 = o_1, ..., O_t = o_t, S_t = \sigma_j),$$

and by the Markov property and the inductive assumption for $\alpha_j(t)$, we can rewrite this as

$$= \sum_{j=1}^{m} \alpha_j(t) \mathbb{P}(O_{t+1} = o_{t+1}, S_{t+1} = \sigma_i | S_t = \sigma_j).$$

However, since we know O_{t+1} is dependent on only S_{t+1} and S_{t+1} depends only on S_t, the last term becomes $\mathbb{P}(O_{t+1} = o_{t+1}, S_{t+1} = \sigma_i | S_t = \sigma_j)$. But this is clearly just $\mathbb{P}(O_{t+1} = o_{t+1} | S_{t+1} = \sigma_i) \mathbb{P}(S_{t+1} = \sigma_i | S_t = \sigma_j)$, and since o_{t+1} equals some $\epsilon_k \in E$, this can be rewritten as

$$= b_{ik} \sum_{j=1}^{m} \alpha_j(t) a_{ji}.$$

Then $\mathbb{P}(O_1 = o_1, ..., O_{t+1} = o_{t+1}, S_{t+1} = \sigma_i) = b_{ik} \sum_{j=1}^{m} \alpha_j(t) a_{ji}$ for $1 \leq t \leq T-1$, which is by definition $\alpha_i(t+1)$.

Now it remains to show that $\sum_{j=1}^{m} \alpha_j(T) = \mathbb{P}(O|\lambda)$. To do this, we again use the fact that hidden states are disjoint to write $\sum_{j=1}^{m} \alpha_j(T) = \mathbb{P}\Big(\big((O_1 = o_1, ..., O_T = o_T) \cap S_T = \sigma_1\big) \cup ... \cup \big((O_1 = o_1, ..., O_T = o_T) \cap S_T = \sigma_m\big)\Big) = \mathbb{P}\big((S_T = \sigma_1 \cup ... \cup S_T = \sigma_m) \cap (O_1 = o_1, ..., O_T = o_T)\big) = \mathbb{P}(O)$. Since the HMM is parameterized by λ (ie we are given λ), we write this as $\mathbb{P}(O)$.

□

3.3.2. *Backward Algorithm.* The Backward Algorithm, like the Forward Algorithm, is concerned with a sequence of observations and a single hidden state, although this time, we are interested in calculating the future probability of seeing the observation sequence $o_{t+1}, ..., o_T$ given the starting hidden state of $S_t = \sigma_i$.

Define $\beta_i(t) = \mathbb{P}(O_{t+1} = o_{t+1}, ..., O_T = o_T | S_t = \sigma_i)$ for some $\sigma_i \in \Sigma$. Then the algorithm for finding $\mathbb{P}(O_{t+1} = o_{t+1}, ..., O_T = o_t | S_t = \sigma_i$ is as follows:[8]

Algorithm 2 Backward Algorithm[8]

1. Define $\beta_i(T) = 1$, for $1 \leq i \leq m$
2. Compute $\beta_i(t) = \sum_{j=1}^{m} a_{ij} b_{jk} \beta_j(t+1)$, for $t = T-1, T-2, ..., 1$ where $o_{t+1} = \epsilon_k \in E$ and $1 \leq i \leq m$.

Proof. To prove that the algorithm works, it must be shown that $\beta_i(t) = \mathbb{P}(O_{t+1} = o_{t+1}, ..., O_T = o_T | S_t = \sigma_i)$ where $\sigma_i \in \Sigma$ for all $1 \leq t \leq T-1$. To do this we will use induction.

◊ *Base case*: By definition $\beta_i(T-1) = \sum_{j=1}^{m} a_{ij} b_{jk} \beta_j(T)$ which by definition of $\beta_i(T)$ is just $\sum_{j=1}^{m} a_{ij} b_{jk}$. By writing this out $\sum_{j=1}^{m} \mathbb{P}(S_T = s_T | S_{T-1} = \sigma_j) \mathbb{P}(O_T = \epsilon_k | S_T = \sigma_j)$ we can see that this sum is the sum of all possible previous hidden states transitioning to the current state $S_T = \sigma_j$ and the current observation being $O_T = o_T = \epsilon_k$ for some $\epsilon_k \in E$. Then it follows by total probability that $\beta_i(T-1) = \sum_{j=1}^{m} a_{ij} b_{jk} \beta(T) = \mathbb{P}(O_T = o_t | S_{T-1} = \sigma_i)$.

◊ *Inductive Assumption* Assume that $\beta_j(t+1) = \mathbb{P}(O_{t+2} = o_{t+2}, ..., O_T = o_T | S_{t+1} = \sigma_j)$.

◊ *Inductive Step* Now consider $\mathbb{P}(O_{t+1} = o_{t+1}, ..., O_T = o_T | S_t = \sigma_i)$. By the conditional probability law of total probability, we can write this

$$\sum_{j=1}^{m} \mathbb{P}(O_{t+1} = o_{t+1}, ..., O_T = o_T | S_t = \sigma_i, S_{t+1} = \sigma_j) \mathbb{P}(S_{t+1} = \sigma_j | S_t = \sigma_i),$$

which is just

$$\sum_{j=1}^{m} \mathbb{P}(O_{t+1} = o_{t+1}, ..., O_T = o_T | S_t = \sigma_i, S_{t+1} = \sigma_j) a_{ij}.$$

We now use the fact that

$$\sum_{j=1}^{m} \mathbb{P}(O_{t+1} = o_{t+1}, ..., O_T = o_T | S_t = \sigma_i, S_{t+1} = \sigma_j) a_{ij}$$

$$= \sum_{j=1}^{m} \mathbb{P}(O_{t+1} = o_{t+1} \cap (O_{t+2} = o_{t+2}, ..., O_T = o_T) | S_t = \sigma_i, S_{t+1} = \sigma_j) a_{ij}$$

$$= \sum_{j=1}^{m} \mathbb{P}(O_{t+1} = o_{t+1} | S_t = \sigma_i, S_{t+1} = \sigma_j) \mathbb{P}(O_{t+2} = o_{t+2}, ..., O_T = o_T | S_t = \sigma_i, S_{t+1} = \sigma_j) a_{ij}.$$

Then by the Markov property,

$$= \sum_{j=1}^{m} \mathbb{P}(O_{t+1} = o_{t+1} | S_{t+1} = \sigma_j) \mathbb{P}(O_{t+2} = o_{t+2}, ..., O_T = o_T | S_{t+1} = \sigma_j) a_{ij},$$

and by our assumption that $\beta_j(t+1) = \mathbb{P}(O_{t+2} = o_{t+2}, ..., O_T = o_T | S_{t+1} = \sigma_j)$ and because $o_{t+1} = \epsilon_k \in E$, this is just $\sum_{j=1}^{m} a_{ij} b_{jk} \beta_j(t+1)$. But this is by definition $\beta_i(t)$, and so it is true that $\beta_i(t) = \sum_{j=1}^{m} a_{ij} b_{jk} \beta_j(t+1)$

□

3.3.3. *Baum-Welch Algorithm.* The Forward and Backward algorithms aren't entirely useful on their own, but when combined they give important insight that is used in the Baum-Welch algorithm. To begin, we will define two variables whose purposes will become evident shortly: $\gamma_i(t) = \mathbb{P}(S_t = \sigma_i | O)$ and $\zeta_{ij}(t) = \mathbb{P}(S_t = \sigma_i, S_{t+1} = \sigma_j | O)$. Immediately we are interested in relating these to known values in our HMM.

Claim 3.6. The variable $\gamma_i(t)$ is related to the Forward and Backward variables according to the following:

$$\gamma_i(t) = \frac{\alpha_i(t)\beta_i(t)}{\sum_{i=1}^{m} \alpha_i(t)\beta_i(t)}$$

Proof. Consider $\gamma_i(t) = \mathbb{P}(S_t = \sigma_i | O)$. By the definition of conditional probability, this can be rewritten as

$$\gamma_i(t) = \frac{\mathbb{P}(O_1 = o_1, ..., O_t = o_t, S_t = \sigma_i, O_{t+1} = o_{t+1}, ..., O_T = o_T)}{\mathbb{P}(O)}.$$

Again, using the laws of conditional probability, we can further rewrite this as

$$\gamma_i(t) = \frac{\mathbb{P}(O_1 = o_1, ..., O_t = o_t, S_t = \sigma_i) \mathbb{P}(O_{t+1} = o_{t+1}, ..., O_T = o_T | O_1 = o_1, ..., O_t = o_t, S_t = \sigma_i)}{\mathbb{P}(O)}.$$

Again by the Markov property for HMMs, this can be further reduced to

$$\gamma_i(t) = \frac{\mathbb{P}(O_1 = o_1, ..., O_t = o_t, S_t = \sigma_i) \mathbb{P}(O_{t+1} = o_{t+1}, ..., O_T = o_T | S_t = \sigma_i)}{\mathbb{P}(O)},$$

which by definition is just

$$\frac{\alpha_i(t)\beta_i(t)}{\mathbb{P}(O)}.$$

Now consider the sum $\sum_{i=1}^{m} \alpha_i(t)\beta_i(t)$. As we just showed, this is equivalent to the sum

$$\sum_{i=1}^{m} \mathbb{P}(O_1 = o_1, ..., O_t = o_t, S_t = \sigma_i, O_{t+1} = o_{t+1}, ..., O_T = o_T).$$

But this is just

$$\sum_{i=1}^{m} \mathbb{P}\big((O_1 = o_1, ..., O_T = o_T) \cap (S_t = \sigma_i)\big),$$

which can be rewritten using the conditional probability law of total probability as

$$\begin{aligned}&\sum_{i=1}^{m} \mathbb{P}(O_1 = o_1, ..., O_T = o_T | S_t = \sigma_i)\mathbb{P}(S_t = \sigma_i)\\ =&\mathbb{P}(O_1 = o_1, ..., O_T = o_T).\end{aligned}$$

Hence,

$$\gamma_i(t) = \mathbb{P}(S_t = \sigma_i | O) = \frac{\alpha_i(t)\beta_i(t)}{\sum_{i=1}^{m} \alpha_i(t)\beta_i(t)}.$$

□

Claim 3.7. The variable $\zeta_{ij}(t)$ is related to the Forward and Backward variables according to the following:

$$\zeta_{ij}(t) = \frac{\alpha_i(t)a_{ij}b_{jk}\beta_j(t+1)}{\mathbb{P}(O)}$$

where $b_{ik}$is the likelihood that $o_{t+1} = \epsilon_k$ given $S_{t+1} = \sigma_i$.

Proof. Consider $\zeta_{ij}(t) = \mathbb{P}(S_t = \sigma_i, S_{t+1} = \sigma_j | O)$. By the definition of conditional probability, this can be rewritten as

$$\zeta_{ij}(t) = \frac{\mathbb{P}(O_1 = o_1, ..., O_t = o_t, S_t = \sigma_i, S_{t+1} = \sigma_j, O_{t+1} = o_{t+1}, ..., O_T = o_T)}{\mathbb{P}(O)}.$$

Again, by the laws of conditional probability this can be further rewritten as

$$\frac{\mathbb{P}(O_1 = o_1, ..., O_t = o_t, S_t = \sigma_i)\mathbb{P}(S_{t+1} = \sigma_j, O_{t+1} = o_{t+1}, ..., O_T = o_T | O_1 = o_1, ..., O_t = o_t, S_t = \sigma_i)}{\mathbb{P}(O)},$$

which by the Markov property is just

$$\frac{\alpha_i(t)\mathbb{P}(S_{t+1} = \sigma_j, O_{t+1} = o_{t+1}, ..., O_T = o_T | S_t = \sigma_i)}{\mathbb{P}(O)}.$$

This can be further rewritten (switching to our shorthand so it fits better in the margins):

$$
\begin{aligned}
=&\frac{\alpha_i(t)\mathbb{P}(S_{t+1}=\sigma_j,O_{t+1}|S_t=\sigma_i)\mathbb{P}(O_{t+2},...,O_T|S_{t+1}=\sigma_j,S_t=\sigma_i,O_{t+1})}{\mathbb{P}(O)}\\
=&\frac{\alpha_i(t)\mathbb{P}(S_{t+1}=\sigma_j,O_{t+1}|S_t=\sigma_i)\mathbb{P}(O_{t+2},...,O_T|S_{t+1}=\sigma_j)}{\mathbb{P}(O)}\\
=&\frac{\alpha_i(t)\mathbb{P}(S_{t+1}=\sigma_j|S_t=\sigma_i)\mathbb{P}(O_{t+1}|S_t=\sigma_i,S_{t+1}=\sigma_j)\mathbb{P}(O_{t+2},...,O_T|S_{t+1}=\sigma_j)}{\mathbb{P}(O)}\\
=&\frac{\alpha_i(t)a_{ij}\mathbb{P}(O_{t+1}|S_{t+1}=\sigma_j)\beta_j(t+1)}{\mathbb{P}(O)}\\
=&\frac{\alpha_i(t)a_{ij}b_{jk}\beta_j(t+1)}{\mathbb{P}(O)}.
\end{aligned}
$$

□

Finally, it will be beneficial to relate $\gamma_i(t)$ to $\zeta_{ij}(t)$.

Claim 3.8.

$$\gamma_i(t)=\sum_{j=1}^{m}\zeta_{ij}(t).$$

Proof. Consider that by definition $\zeta_{ij}(t)=\mathbb{P}(S_t=\sigma_i,S_{t+1}=\sigma_j|O)$. Already, this almost looks like $\gamma_i(t)=\mathbb{P}(S_t=\sigma_i|O)$, aside from the $S_{t+1}=\sigma_j$ term. As we have done previously, we will try summing over j to get rid of this term:

$$
\begin{aligned}
\sum_{j=1}^{m}\zeta_{ij}(t)=&\sum_{j=1}^{m}\mathbb{P}(S_t=\sigma_i,S_{t+1}=\sigma_j|O)\\
=&\sum_{j=1}^{m}\frac{\mathbb{P}(O_1=o_1,...,O_T=o_T,S_t=\sigma_i,S_{t+1}=\sigma_j)}{\mathbb{P}(O)}.
\end{aligned}
$$

By the conditional probability law for total probability:

$$
\begin{aligned}
=&\sum_{j=1}^{m}\frac{\mathbb{P}(O_1=o_1,...,O_T=o_T,S_t=\sigma_i|S_{t+1}=\sigma_j)\mathbb{P}(S_{t+1}=\sigma_j)}{\mathbb{P}(O)}\\
=&\frac{\mathbb{P}(O_1=o_1,...,O_T=o_T,S_t=\sigma_i)}{\mathbb{P}(O)}\\
=&\mathbb{P}(S_t=\sigma_i|O)
\end{aligned}
$$

Hence,

$$\gamma_i(t) = \sum_{j=1}^{m} \zeta_{ij}(t).$$

□

We now move on to tackling *iv*) from section 3.1. Up until now, everything has been done with the assumption that we are given a Hidden Markov Model $\lambda = \{T_\Sigma, T_E, \vec{\Pi}\}$. However, unless we have created our own model from scratch, the underlying HMM describing real world events is likely hidden from us, and we must find some way to reasonably discover or approximate the true HMM. Namely, our goal is to find some $\bar{\lambda} = \{\bar{T_\sigma}, \bar{T_E}, \bar{\vec{\Pi}}\}$ with pre-chosen values n, m (which we must pick based on available knowledge of the underlying system) such that given a sequence of observations $O = \{O_1 = o_1, ..., O_T = o_T\}$, the likelihood of O is maximized. In other words, we seek to find the model $\bar{\lambda} = \{\bar{T_\sigma}, \bar{T_E}, \bar{\vec{\Pi}}\}$ that most closely approximates the true underlying Hidden Markov Model. This is done heuristically using the Baum-Welch algorithm:

Algorithm 3 Baum-Welch Algorithm[8][1]

1. Given a sequence of observations O, choose a reasonable model $\lambda = \{T_\Sigma, T_E, \vec{\Pi}\}$ to describe the underlying HMM.
2. Define a maximum number of iterations, l_{max} and set a counter variable $l = 1$.
3. Define $\bar{\lambda} = \{\bar{T_\sigma}, \bar{T_E}, \bar{\vec{\Pi}}\}$ as follows:
   a) Let $\bar{\vec{\Pi}} = \{\pi_i = \gamma_i(1)\}$ for $1 \leq i \leq m$.
   b) Let $\bar{T_\Sigma} = \left\{ a_{ij} = \frac{\sum_{t=1}^{T-1} \zeta_{ij}(t)}{\sum_{t=1}^{T-1} \gamma_i(t)} \right\}$.
   c) Let $\bar{T_E} = \left\{ b_{ij} = \frac{\sum_{t=1}^{T} \gamma_i(t) \text{ s.t. } O_t = \epsilon_j}{\sum_{t=1}^{T} \gamma_i(t)} \right\}$.
4. Let $l = l + 1$ and
   - if $l < l_{max}$ and $\mathbb{P}(O|\bar{\lambda}) > \mathbb{P}(O|\lambda)$, let $\lambda = \bar{\lambda}$. Return to 3.
   - else terminate.

Baum et al showed that it is always true that either $\mathbb{P}(\bar{\lambda}) = \mathbb{P}(\lambda)$ or that $\mathbb{P}(\bar{\lambda}) > \mathbb{P}(\lambda)$, and so either our new approximation of the HMM is better than the previous, or it is at least a local maximum.[8, pg. 256] To justify this, consider the fact that we want to find a better estimate for $T_\Sigma, T_E, \vec{\Pi}$ that takes into account the actual sequence of observations we can access,

as naturally a model estimate that uses more of the available information will be better. In the case of reestimating $\vec{\Pi}$ with the values of $\gamma_i(1)$, the justification is clear because $\gamma_i(1) = \mathbb{P}(S_1 = \sigma_i|O)$, which is essentially adding an awareness of O to our initial definition of $\vec{\Pi}$.

To justify why $\bar{T_\Sigma}$ is a better estimate for T_Σ, we will take entries a_{ij} in T_Σ and relate them to $\zeta_{ij}(t)$ and $\gamma_i(t)$. First, though, note that $a_{ij} = \mathbb{P}(S_{t+1} = \sigma_j|S_t = \sigma_i)$ for all $1 \leq t \leq T-1$ because our model is time independent. Then we can rewrite

$$\begin{aligned}
a_{ij} &= a_{ij}\frac{\sum_{t=1}^{T-1}\mathbb{P}(S_t = \sigma_i)}{\sum_{t=1}^{T-1}\mathbb{P}(S_t = \sigma_i)} \\
&= \frac{a_{ij}\mathbb{P}(S_1 = \sigma_i) + ... + a_{ij}\mathbb{P}(S_{T-1} = \sigma_i)}{\sum_{t=1}^{T-1}\mathbb{P}(S_t = \sigma_i)} \\
&= \frac{\mathbb{P}(S_2 = \sigma_j|S_1 = \sigma_i)\mathbb{P}(S_1 = \sigma_i) + ... + \mathbb{P}(S_T = \sigma_j|S_{T-1} = \sigma_i)\mathbb{P}(S_{T-1} = \sigma_i)}{\sum_{t=1}^{T-1}\mathbb{P}(S_t = \sigma_i)} \text{ (by time ind.)} \\
&= \frac{\mathbb{P}(S_1 = \sigma_i, S_2 = \sigma_j) + ... + \mathbb{P}(S_{T-1} = \sigma_i, S_T = \sigma_j)}{\sum_{t=1}^{T-1}\mathbb{P}(S_t = \sigma_i)} \text{ (by conditional probability)} \\
&= \frac{\sum_{t=1}^{T-1}\mathbb{P}(S_t = \sigma_i, S_{t+1} = \sigma_j)}{\sum_{t=1}^{T-1}\mathbb{P}(S_t = \sigma_i)}
\end{aligned}$$

However, note that by definition, $\zeta_{ij}(t) = \mathbb{P}(S_t = \sigma_i, S_{t+1} = \sigma_j|O)$ and $\gamma_i(t) = \mathbb{P}(S_t = \sigma_i|O)$, so if we want our estimation of a_{ij} to take into account the sequence of observations O, it makes sense to replace the previous sum with

$$\frac{\sum_{t=1}^{T-1}\zeta_{ij}(t)}{\sum_{t=1}^{T-1}\gamma_i(t)}.$$

Finally, to justify why $\bar{T_E}$ is a better estimate for T_E, we consider a similar explanation:

$$\begin{aligned}
b_{ij} &= b_{ij}\frac{\sum_{t=1}^{T}\mathbb{P}(S_t = \sigma_i)}{\sum_{t=1}^{T}\mathbb{P}(S_t = \sigma_i)} \\
&= \frac{b_{ij}\mathbb{P}(S_1 = \sigma_i) + ... + b_{ij}\mathbb{P}(S_T = \sigma_i)}{\sum_{t=1}^{T}\mathbb{P}(S_t = \sigma_i)} \\
&= \frac{\mathbb{P}(O_1 = \epsilon_j|S_1 = \sigma_i)\mathbb{P}(S_1 = \sigma_i) + ... + \mathbb{P}(O_T = \epsilon_j|S_T = \sigma_i)\mathbb{P}(S_T = \sigma_i)}{\sum_{t=1}^{T}\mathbb{P}(S_t = \sigma_i)} \\
&= \frac{\sum_{t=1}^{T}\mathbb{P}(O_t = \epsilon_j|S_t = \sigma_i)\mathbb{P}(S_t = \sigma_i)}{\sum_{t=1}^{T}\mathbb{P}(S_t = \sigma_i)}
\end{aligned}$$

However, it is not true for all observations in O that $O_t = \epsilon_j$, which means that some entries in this sum are 0, and so they may be omitted. The moving along we get

$$= \frac{\sum_{t=1 \text{ s.t. } O_t=\epsilon_j}^{T} \mathbb{P}(O_t = \epsilon_j | S_t = \sigma_i)\mathbb{P}(S_t = \sigma_i)}{\sum_{t=1}^{T} \mathbb{P}(S_t = \sigma_i)}$$
$$= \frac{\sum_{t=1 \text{ s.t. } O_t=\epsilon_j}^{T} \mathbb{P}(O_t = \epsilon_j, S_t = \sigma_i)}{\sum_{t=1}^{T} \mathbb{P}(S_t = \sigma_i)},$$

and since we are only summing over indeces where $O_t = \epsilon_j$, this is just

$$\frac{\sum_{t=1 \text{ s.t. } O_t=\epsilon_j}^{T} \mathbb{P}(S_t = \sigma_i)}{\sum_{t=1}^{T} \mathbb{P}(S_t = \sigma_i)}.$$

Again, by definition $\gamma_i(t) = \mathbb{P}(S_t = \sigma_i | O)$, which we take to be a better approximation because it encodes more given information, so we can rewrite this sum simply as

$$\frac{\sum_{t=1 \text{ s.t. } O_t=\epsilon_j}^{T} \gamma_i(t)}{\sum_{t=1}^{T} \gamma_i(t)}.$$

It should be noted that the better the initial guess for λ is, the more likely $\bar{\lambda}$ is a good approximation of the the underlying HMM.

We can now use the Viterbi and Baum-Welch algorithms to analyze passwords, which will be discussed in section 4.

4. Application with Concluding Comments

In this section, several large training sets of real-world passwords will be used to train and implement the various Markov chains discussed in this thesis. While building an actual program to use these in bruteforcing is beyond the scope of this paper, the Markov chains will be used to verify our initial intuition that some passwords are more likely to be used by humans than others, which should be reflected by having a higher overall probability. The training sets used include the "rockyou.txt," and "myspace.txt," password sets, along with a conglomerate password set "AntiPublic." Respectively, these contain roughly 32 million, 427 million, and 562 million passwords each.[12]

[12]Although in the Markov chains with memory section, Antipublic was not used because the program for calculating the transition matrix was expected to take over 4 days to run due to the author's very limited computing power on a single core intel i5 processor with only 8 gigabytes of ram

4.1. **Application of Standard Markov Chains.** One way to check that a Markov chain is appropriate for modeling passwords is to try feeding one a few different passwords. In section 1, it was claimed that intuitively, the password "Kittycat123!" is much more likely to be a real password than "0k@#Hckpdz%5." Then assuming passwords can be appropriately modeled by Markov chains, we can calculate the actual probabilities of these and several other passwords. In addition to these two passwords, we will try calculating the probabilities of observing a password the author created according to a common password creation scheme - "P455w0rd123!," along with a password that was randomly generated by the LastPass random password generator with input paramaters "12 characters," "Upper and Lowercase," "Minimum of 3 numbers," and "Special characters" - "z8U73*!mg@5S." Below is a table of probabilities according to the different training sets for a standard Markov chain:

Password Set	"Kittycat123!"	"0k@#Hckpdz%51"	"P455w0rd123!"	"z8U73*!mg@5S"
Rockyou	4.13505573575e-18	2.68083451418e-29	1.67240191453e-22	7.38680466769e-31
Myspace [13]	3.78308960169e-17	0	0	0
Antipublic	2.00339243244e-17	4.08860633509e-30	1.54299616379e-20	6.48580541618e-30

These numbers verify what we had hoped they would – we can see clearly that the password "Kittycat123!" is clearly the most likely by orders of magnitude. Moreover, "P455w0rd123!," which was created using the word "Password" and switching several letters to their digit look-alikes, also has relatively high probability when compared to the random strings. This confirms that the transitions taking place in it must be semi-frequent, which may be explained by the Markov model reflecting some common underlying password creation trends (although this conclusion cannot be outright determined, and in fact comparing "P455W0rd123!" with similar 12-character passwords that end in "123!" show that that Markov model may only be expressing the overall quantity of lowercase, uppercase, numbers and special characters, which are in general the most deterministic feature of a password's security).

4.2. **Application of Markov Chains with Memory.** Again, we hope to use Markov chains, this time implemented with $m = 2$ (due to computational power limitations[14]), to test some of our intuition about passwords.

Password Set	"Kittycat123!"	"0k@#Hckpdz%5"	"P455w0rd123!"	"z8U73*!mg@5S"
Rockyou	4.00211562786e-15	3.84742041879e-28	8.02964469332e-19	0
Myspace	1.5847060339e-13	0	0	0

[14]Here the Antipublic data set is not included because the program used to estimate the transition matrix, slated to run just shy of 4 days, never had an uninterupted chance to terminate.

Once more, these numbers add to the growing body of evidence that our intuitions may have been correct: we see that "Kittycat123!" is by far the most probable, with the pattern-constructed password "P455w0rd123!" coming in at second, the human generated password coming in at third, and the randomly generated password having undefined probability.

We now move on to see whether Markov chains with memory are more capable of telling us that a password like *abracadabra* is more likely than a password like *abracadabrd*. Essentially, we are hoping that adding memory to the Markov chain gives the chain a better chance at picking up on patterns in passwords (because, after all, you have to remember what has already been established to explain a pattern). Several different types of passwords containing some form of clearly identifiable pattern[15] can then be created, and then these passwords can be tested against the pattern applied incorrectly:

$\mathbb{P}(abracadabra) = 1.69594044414e-13$
$\mathbb{P}(abracadabrd) = 1.89324641898e-15$[16]
$\mathbb{P}(abcdefgh) = 2.19587965676e-11$
$\mathbb{P}(fadhebgc) = 3.69930437095e-13$
$\mathbb{P}(123456789) = 2.77509528407e-07$
$\mathbb{P}(918273645) = 9.40000283098e-11$
$\mathbb{P}(a1b2c3d4e5) = 1.98899504362e-17$
$\mathbb{P}(a3b1c5d2e4) = 9.00789930584e-22$

In each and everyone of these cases, what was hoped for is verified: introducing memory into the Markov chain allows the model to more suitably rank passwords based on their exhibition of a logical pattern.

4.3. **Application of Indexed Markov Chains.** Our final implementation of Markov chains helps further confirm our intuition:

Password Set	"Kittycat123!"	"0k@#Hckpdz%51"	"P455w0rd123!"	"z8U73*!mg@5S"
Rockyou	3.92008464899e-16	4.45807250796e-30	2.969286342653-21	1.48144894008e-32
Myspace	0	0	0	0
Antipublic	9.8544938451e-16	1.10863834593e-29	2.88716532716e-18	4.42802425092e-31

Yet again we see that the password we would expect to have the highest probability does in fact have the highest by several orders of magnitude. The pattern-constructed password has the second, the human generated password is third, and the random password is fourth.

[15] Identifying the pattern and its jumbled variant is left as an exercise for the reader.

[16] In this case, replacing the final letter with anything that also commonly occurs after the cluster of letters "*br*", such as the letter *e* will cause the difference between these two passwords to shrink. As the length of memory is increased, one would expect the difference in probability between these two passwords to grow regardless of the character placed at the end.

While nothing can be determined conclusively about these passwords using any of the Markov chains discussed so far, these numbers present a persuasive argument that Markov chains do in fact model some of the underlying password creation trends we had hoped they would. To be more conclusive, one would have to be very careful about the passwords that were compared, controlling for the total number of different types of characters, the likelihood of the different special characters (for example, a "!" is much more likely in a password than a "%" and so any two similarly constructed passwords, one whose only special character is "!" and the other whose only special character is "%", will likely have very different probabilities, even though the underlying password creation technique is the same) and so on.

One application of these various models not discussed so far is checking that a user's password is secure. Since ideally a bruteforce password attack would prioritize more likely passwords, it makes sense to choose passwords with lower probabilities according to these models. Effectually, a password with lower probability should have a better chance of withstanding a bruteforce attack, especially one based on any of the models discussed in this paper.

Now it has also been mentioned that an indexed Markov chain should confirm a trend we observe in passwords – namely that special characters are more likely to occur at the end of passwords. We can then construct passwords to see whether or not this is the case. We will start by testing against the passwords

$$(aaaaaaa!, aaaaaa!a, aaaaa!aa, aaaa!aaa, aaa!aaaa, aa!aaaaa, a!aaaaaa).$$

Note that with these passwords, aside from the first, there are the same number of each type of transition (ie there are $5 \times a \to a$, $1 \times a \to !$, and $1 \times ! \to a$). The reason we include the first password in this set (which has $6 \times a \to a$, $1 \times a \to !$, and no $! \to a$) is because we still want it to be true that "!" appears at the end of the password, so we check this case. Finally, the reason we do not check the password $!aaaaaaa$ is because this changes the value for the initial distribution, and we already assume that "!" is less likely to occur at the beginning of a password from observation. We can now check the assumption (here we use only the largest training set, AntiPublic):

$$\begin{aligned}
\mathbb{P}(aaaaaaa!) &= 6.60864029259e-16 & \mathbb{P}(aaa!aaaa) &= 1.8899437618e-17 \\
\mathbb{P}(aaaaaa!a) &= 5.30792802652e-16 & \mathbb{P}(aa!aaaaa) &= 8.09724163596e-18 \\
\mathbb{P}(aaaaa!aa) &= 3.22422594458e-16 & \mathbb{P}(a!aaaaaa) &= 4.16366090077e-17 \\
\mathbb{P}(aaaa!aaa) &= 3.79525129385e-17
\end{aligned}$$

This trend, which shows that the probability steadily decreases as "!" is moved forward in the password, confirms that indexed Markov chains are better at capturing trends we observe in real passwords than standard Markov

chains, because all of the passwords aside from the *aaaaaaa*!, when analyzed with a standard Markov chain, would show the same probability (which is $2.22404415478e-16$) since they have the same number of each type of transition.

It is worth noting that this trend does not seem to appear as strongly when the same password is tested with different special characters. For example, when the special character "%" is used, we get the following probabilities:

$$\mathbb{P}(aaaaaaa\%) = 4.30348662538e-18 \quad \mathbb{P}(aaa\%aaaa) = 2.40811900998e-18$$
$$\mathbb{P}(aaaaaa\%a) = 5.53865676248e-18 \quad \mathbb{P}(aa\%aaaaa) = 1.64874136141e-18$$
$$\mathbb{P}(aaaaa\%aa) = 3.99697153743e-18 \quad \mathbb{P}(a\%aaaaaa) = 9.09272690323e-19$$
$$\mathbb{P}(aaaa\%aaa) = 1.51732474001e-18$$

This indicates that when people use "%", the character is more evenly spread throughout passwords.

Another interesting observation is the distribution of the "_" symbol in passwords. Since "_" is commonly used to represent a space between words, it should make sense that this symbol should be most frequent neither at the beginning or the end of passwords, but somewhere in the middle. This can be seen using an indexed Markov chain[17]:

$$\mathbb{P}(aaaaaaaaaa\_) = 7.41528683e-23 \quad \mathbb{P}(aaaaa\_aaaaaa) = 1.34840427e-22$$
$$\mathbb{P}(aaaaaaaaa\_a) = 1.88956268e-22 \quad \mathbb{P}(aaaa\_aaaaaaa) = 7.22417620e-23$$
$$\mathbb{P}(aaaaaaaaa\_aa) = 1.39673956e-22 \quad \mathbb{P}(aaa\_aaaaaaaa) = 2.96022951e-23$$
$$\mathbb{P}(aaaaaaaa\_aaa) = 1.18356981e-22 \quad \mathbb{P}(aa\_aaaaaaaaa) = 2.79106433e-23$$
$$\mathbb{P}(aaaaaaa\_aaaa) = 1.37423799e-22 \quad \mathbb{P}(a\_aaaaaaaaaa) = 2.09277982e-23$$
$$\mathbb{P}(aaaaaa\_aaaaa) = 1.33752753e-22$$

Again, in this case we see a moderate bias towards "_" being found towards the middle of passwords

4.4. **Application of Hidden Markov Models.** A coded implementation of HMMs is beyond the scope of this paper for several reasons. First, an HMM could be run on the level of single characters, but it is unlikely that this would reveal new information (although it has the potential to) beyond the already known fact that there are hidden states "lowercase," "uppercase," "numbers" and "special characters" whose observations are the sets that comprise each of these categories. What would be more interesting is running an HMM at a higher level, for example, having it examine the words stop sequences[18] used in passwords. This becomes much more difficult because the HMM will have

[17]Here, longer passwords are used with the assumption that passwords containing a "_" representing a space are made of shorter words, increasing the overall length

[18]A stop sequence is a common string that is used at the end of passwords. For example, the stop sequence used in the previous section for two passwords was "123!"

to be told what valid "words" and stop sequences are, which requires large dictionaries consisting of predetermined values. Moreover, one would expect that a sequence like "P455w0rd" should count as a word, since it represents "Password" with several switched characters, but clearly this sequence will not be found in any dictionary list. It becomes an issue then of how to define the "words" the HMM can examine. Even so, in the case that an HMM were looking at transitions between words and stop sequences, some hypothetical hidden states could be some of the categories discussed in [3], such as place names, people names, dictionary words, email addresses, double words, etc. By running an HMM using the discussed algorithms against a large list of real passwords, one would hope to more precisely identify the underlying methods people use in creating their passwords, which in turn could be reflected in some type of bruteforce algorithm that prioritizes passwords that follow similar creation logic.

Acknowledgments

I'd like to give special thanks to my thesis advisor Rob Manning for working with me over the course of two semesters, along with Lynne Butler for the feedback she gave as my second reader. Thanks also goes out to the entire math department for the time they take to read all math majors' theses.

References

[1] S. R. Dunbar. "Topics in Probability Theory and Stochastic Processes". In: (2017).

[2] G. D. Forney. "The Viterbi Algorithm". In: ().

[3] T. Hunt. *The Science of Password Selection*. 2011. URL: https://www.troyhunt.com/science-of-password-selection/.

[4] J. Kirkwood. *Markov Processes*. CRC Press, 2015.

[5] A. Meyer. *Mathematics for Computer Science*. 2015.

[6] Giuseppe Modica and Laura Poggiolini. *A First Course in Probability and Markov Chains*. Wiley, 2013.

[7] O. Pavlyk. "Centennial of Markov Chains". In: (2013). URL: http://blog.wolfram.com/2013/02/04/centennial-of-markov-chains/.

[8] L. Rabiner. "A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition". In: (1989).

[9] L. Serrano. *A Friendly Introduction to Bayes Theorem and Hidden Markov Models*. 2018. URL: https://www.youtube.com/watch?v=kqSzLo9fenk.

[10] W. Tansey. "Improved Models for Password Guessing". In: (2012).

[11] J. Yan et al. "The memorability and security of passwords - some empirical results". In: (2004).

[12] G. Zitkovic. "Introduction to Stochastic Processes". In: (2010). URL: https://web.ma.utexas.edu/users/gordanz/notes/introduction_to_stochastic_processes.pdf.

Appendices

A. Python Code

A.1. **Python code used to approximate standard Markov chain transition matrix of a password set.** Please be forewarned that the author of this thesis is not a "coder," and as such the following code may not adhere to professional written code standards. The following code was also not cleaned to remove commented-out sections that were not used. Further note that the formatting has caused the title of each index to be found at the bottom of the page.

```
"""
Finding the Transition matrices for my password databases

"""
import numpy as np
import random as rm
import csv
from collections import Counter
def markov(x):
      with open(x, "r") as database:
            charspace =
["A","B","C","D","E","F","G","H",'I','J','K','L','M','N','O','P','Q','R','S','T','
U','V','W','X','Y','Z','a','b','c','d','e','f','g','h','i','j','k','l','m','n','o'
,'p','q','r','s','t','u','v','w','x','y','z','1','2','3','4','5','6','7','8','9','
0','!','@','#','$','%','^','&','*','(',')','-','_','+','=','.','[',']','{','}','|'
,'`','~',';',':',"'",'"','<',',','>','/','?',' ']

            #print charspace
            # The following commented out section could be used to build the
character space beyond what is included here.
            """
            charspace2 = []
            T = []
            Pi = []
            for i in database:
                  for j in i:
                        if str(j) not in charspace and str(j) not in charspace2:
                              charspace2.append(str(j))

            charspace = charspace + charspace2
            print charspace
            size_of_char = 0
            for i in charspace:
                  size_of_char += 1
            print size_of_char
            #print charspace
            """
            trans = [] #matrix that defines state transitions
            #temporary matrix used to build trans
            for i in range(93):
                  temptrans = []
                  counter = 0
                  while counter <=93:
                        temptrans.append(str(charspace[i])+str(charspace[counter]))
                        counter +=1
                  trans.append(temptrans)

            """
            Now we want to get T
            """

            T = []
            for i in range(94):
                  tempT = []
                  for j in range(94):
                        tempT.append(0)
                  T.append(tempT) #T is now a 93x93 matrix
```

```
            #print T
            #Now to add values to T
            progress = 0
            for i in database:
                  #print i
                  #Now check that i has no forbidden characters.
                  counter = 0
                  for j in i:
                        if str(j) not in charspace:
                              #print j
                              counter +=1
                  if counter >1: #each line has a hidden character used to signal
next line that's not in charspace. We need counter>1
                        continue
                  else:
                        counter = 0
                        for j in i[0:-2]:
                              #check the pair j,j+1. Add a 1 to the "j"th column of
T
                              T[charspace.index(str(j))][charspace.index(i[counter
+ 1])] +=1
                              counter +=1
                  progress +=1
                  print "Progress = " + str(float(progress)/float(562000000) * 100)
+ "for Antipublic"

            # T is still just a matrix of "total transitions" and we need to turn
them into probabilities
            counti = 0
            #Lets find the sum of the rows to get a denominator
            sumofrow = 0

            for i in T:
                  sumofrow=0
                  for j in i:
                        sumofrow = int(j) + sumofrow
                  #print sumofrow
                  countj = 0
                  for j in i:
                        if sumofrow >= 1:
                              T[counti][countj] = float(j)/float(sumofrow) #To find
percentages. Note counti and countj are needed because i,j cannot be indeces
                              #print T[counti][countj]
                              countj +=1
                        elif sumofrow == 0:
                              T[counti][countj] = 0 #To find percentages. Note
counti and countj are needed because i,j cannot be indeces
                              #print T[counti][countj]
                              countj +=1
                  counti +=1

            #Double check to make sure the rows sum to 1
            counti = 0
            for i in T:
                  counti = 0
                  print sum(T[counti])
                  counti+=1
```

```
            with open(x+"_T.csv", "wb") as f:
                  writer = csv.writer(f)
                  writer.writerows(T)
            print "Done with T for " + x


markov("anti_stripped_562_mil.txt")
```

A.2. **Python code to approximate standard initial distribution vector of a password set.**

```
"""
Finding Pi for txt files

"""

import numpy as np
import random as rm
import csv
from collections import Counter
def markov(x):
      with open(x, "r") as database:
            charspace =
["A","B","C","D","E","F","G","H",'I','J','K','L','M','N','O','P','Q','R','S','T','
U','V','W','X','Y','Z','a','b','c','d','e','f','g','h','i','j','k','l','m','n','o'
,'p','q','r','s','t','u','v','w','x','y','z','1','2','3','4','5','6','7','8','9','
0','!','@','#','$','%','^','&','*','(',')','-','_','+','=','.','[',']','{','}','|'
,'`','~',';',':',"'",'"','<',',','>','/','?',' ']


            Pi = []
            for i in charspace:
                  Pi.append(0)



##############################################################################
########################################
            numofpswd = 0 #Our denominator will be the total number of passwords
            for i in database:

                  #Now check that i has no forbidden characters.

                  counterpi = 0
                  for j in i:
                        if str(j) not in charspace:

                              counterpi +=1

                  if counterpi >1: #each line has a hidden character used to signal
next line that's not in charspace. We need counter>1
                        continue
                  elif i[0] not in charspace:

                        if len(i)>1: #Some lines appear to be blank? Not sure
what's going on here.
                              Pi[charspace.index(i[1])] += 1
                              numofpswd +=1
                  else:
                        Pi[charspace.index(i[0])] += 1
                        numofpswd +=1



            counterpi2 = 0
            for i in Pi:
                  Pi[counterpi2] = float(i)/float(numofpswd)
                  counterpi2 +=1
```

```
            #print sum(Pi) #Pi sums to 1, it's probably good
            #print Pi
            with open(x+"_Pi.csv", "w") as f:
                  for j in Pi:
                        f.write(str(j))
                        f.write(",")
                  f.close()
            print "Done with Pi for " + x


#markov("rockyou.txt")
#markov("myspace.txt")
markov("anti_stripped_562_mil.txt")
```

A.3. **Python code to approximate transition matrix for Markov chain with memory 2.**

```
"""
Finding T matrix for Markov Chain with memory = 2
"""

import numpy as np
import random as rm
import csv
from collections import Counter

def markov_mem_T(x):
      with open(x, "r") as database:
            charspace =
["A","B","C","D","E","F","G","H",'I','J','K','L','M','N','O','P','Q','R','S','T','
U','V','W','X','Y','Z','a','b','c','d','e','f','g','h','i','j','k','l','m','n','o'
,'p','q','r','s','t','u','v','w','x','y','z','1','2','3','4','5','6','7','8','9','
0','!','@','#','$','%','^','&','*','(',')','-','_','+','=','.','[',']','{','}','|'
,'`','~',';',':',"'",'"','<',',','>','/','?',' ']

            memory_matrix = [] #This will eventually be the matrix with our
couplets

            for i in charspace:
                  for j in charspace:
                        memory_matrix.append(i+j) #This creates said matrix.

            trans = []
            for i in range(8836):
                  trans.append([])
            for i in trans:
                  for j in range(94):
                        i.append(0)


            """
            trans = [] #we need to build a 94^2x94 matrix
            for i in range(8836):
                  trans.append([])


            empty_94 = [] #we create an empty matrix with 94 spots, one for each
character
            for i in range(94):
                  empty_94.append(0)
            counter = 0
            for i in trans: #fill all the rows with 94 empty spots
                  trans[counter] = empty_94
                  counter +=1
                  """

            progress = 0
            for i in database:
                  counter = 0
                  for j in i:
                        if str(j) not in charspace:
                                    #print j
                              counter +=1
                  if counter >1: #each line has a hidden character used to signal
next line that's not in charspace. We need counter>1
```

```
                        continue


                elif len(i)>3:
                        counter = 0
                        length = len(i)
                        temp_mem = []
                        while counter < length - 2: #-2 bewcause we need to ignore
the new line character
                                temp_mem.append(i[counter]+i[counter+1]) #We now have
a matrix that has all our clusters
                                counter += 1
                        #print temp_mem
                        temp_mem_length = 0
                        for j in temp_mem:
                                temp_mem_length +=1
                        #print temp_mem_length
                        counter = 0
                        for j in temp_mem:

                                if counter < temp_mem_length -1:
                                        #print j
                                        #print matrix.index(i)
                                        #print temp_mem[counter +1][1]
                                        #print charspace.index(temp_mem[counter + 1]
[1])
                                        trans[memory_matrix.index(j)]
[charspace.index(temp_mem[counter + 1][1])] = trans[memory_matrix.index(j)]
[charspace.index(temp_mem[counter + 1][1])] +1

                                counter +=1
                        progress +=1
                        print str(float(progress)/float(562000000) * 100) +"% if you're
doing antipublic"
                print trans

                #Now we need to sum the rows and then divide each entry by the sum
                counti = 0
                for i in trans:
                        sumofrow = 0
                        for j in i:
                                sumofrow = sumofrow +int(j)
                        countj = 0
                        for j in i:
                                if sumofrow >= 1:
                                        trans[counti][countj] = float(j)/float(sumofrow) #To
find percentages. Note counti and countj are needed because i,j cannot be indeces
                                        #print T[counti][countj]
                                        countj +=1
                                elif sumofrow == 0:
                                        trans[counti][countj] = 0 #To find percentages. Note
counti and countj are needed because i,j cannot be indeces
                                        #print T[counti][countj]
                                        countj +=1
                        counti +=1
                #double check that each row sums to 1
                counti = 0
                for i in trans:
```

```
                        continue


                elif len(i)>3:
                        counter = 0
                        length = len(i)
                        temp_mem = []
                        while counter < length - 2: #-2 bewcause we need to ignore
the new line character
                                temp_mem.append(i[counter]+i[counter+1]) #We now have
a matrix that has all our clusters
                                counter += 1
                        #print temp_mem
                        temp_mem_length = 0
                        for j in temp_mem:
                                temp_mem_length +=1
                        #print temp_mem_length
                        counter = 0
                        for j in temp_mem:

                                if counter < temp_mem_length -1:
                                        #print j
                                        #print matrix.index(i)
                                        #print temp_mem[counter +1][1]
                                        #print charspace.index(temp_mem[counter + 1]
[1])
                                        trans[memory_matrix.index(j)]
[charspace.index(temp_mem[counter + 1][1])] = trans[memory_matrix.index(j)]
[charspace.index(temp_mem[counter + 1][1])] +1

                                counter +=1
                        progress +=1
                        print str(float(progress)/float(562000000) * 100) +"% if you're
doing antipublic"
            print trans

            #Now we need to sum the rows and then divide each entry by the sum
            counti = 0
            for i in trans:
                  sumofrow = 0
                  for j in i:
                        sumofrow = sumofrow +int(j)
                  countj = 0
                  for j in i:
                        if sumofrow >= 1:
                              trans[counti][countj] = float(j)/float(sumofrow) #To
find percentages. Note counti and countj are needed because i,j cannot be indeces
                              #print T[counti][countj]
                              countj +=1
                        elif sumofrow == 0:
                              trans[counti][countj] = 0 #To find percentages. Note
counti and countj are needed because i,j cannot be indeces
                              #print T[counti][countj]
                              countj +=1
                  counti +=1
            #double check that each row sums to 1
            counti = 0
            for i in trans:
```

A.4. **Python code to approximate initial distribution vector for a Markov chain with memory.**

```
"""
Finding the T Matrix for a Markov Chain with memory = 2

"""
import numpy as np
import random as rm
import csv
from collections import Counter

def markov_mem_2(x):
      with open(x, "r") as database:
            charspace =
["A","B","C","D","E","F","G","H",'I','J','K','L','M','N','O','P','Q','R','S','T','
U','V','W','X','Y','Z','a','b','c','d','e','f','g','h','i','j','k','l','m','n','o'
,'p','q','r','s','t','u','v','w','x','y','z','1','2','3','4','5','6','7','8','9','
0','!','@','#','$','%','^','&','*','(',')','-','_','+','=','.','[',']','{','}','|'
,'`','~',';',':',"'",'"','<',',','>','/','?',' ']
            memory_matrix = [] #This will eventually be the matrix with our
couplets
            for i in charspace:
                  for j in charspace:
                        memory_matrix.append(i+j) #This creates said matrix.


            Pi = []
            for i in memory_matrix:
                  Pi.append(0) #Create a bunch of empty spaces - one for each entry
in memory_matrix




################################################################################
##########################################
            numofpswd = 0 #Our denominator will be the total number of passwords
            counter = 0
            progress = 0
            for i in database:

                  #Now check that i has no forbidden characters.

                  counterpi = 0
                  for j in i:
                        if str(j) not in charspace:
                              counterpi +=1
                  if counterpi >1: #each line has a hidden character used to signal
next line that's not in charspace. We need counter>1
                        counter +=1
                        continue
                  elif i[0] not in charspace:
                        if len(i)>3: #Some lines appear to be blank? Not sure
what's going on here.
                              temp_mem = i[1]+i[2]
                              print temp_mem
                              Pi[memory_matrix.index(temp_mem)] += 1
                              numofpswd +=1
                  elif len(i) >2:
```

```
                        temp_mem = i[0]+i[1]

                        Pi[memory_matrix.index(temp_mem)] += 1
                        numofpswd +=1
                  counter +=1
                  #print counter
                  progress +=1
                  print "Progress = " +str(float(progress)/float(562000000) * 100)
+ "for antipublic "

            #print numofpswd
            #print Pi

            counterpi2 = 0
            for i in Pi:
                  Pi[counterpi2] = float(i)/float(numofpswd)
                  counterpi2 +=1

            #print sum(Pi) #Pi sums to 1, it's probably good
            #print Pi
            with open(x+"_mem_Pi.csv", "w") as f:
                  for j in Pi:
                        f.write(str(j))
                        f.write(",")
                  f.close()
            print "Done with Pi for " + x









#markov_mem_2("rockyou.txt")
markov_mem_2("anti_stripped_562_mil.txt")
```

A.5. Python code to approximate transition matrices for an indexed Markov chain.

```
"""
Finding indexed Transition matrices
"""

import numpy as np
import random as rm
import csv
from collections import Counter
def markov_indexed_T(x):
      with open(x, "r") as database:
            charspace =
["A","B","C","D","E","F","G","H",'I','J','K','L','M','N','O','P','Q','R','S','T','
U','V','W','X','Y','Z','a','b','c','d','e','f','g','h','i','j','k','l','m','n','o'
,'p','q','r','s','t','u','v','w','x','y','z','1','2','3','4','5','6','7','8','9','
0','!','@','#','$','%','^','&','*','(',')','-','_','+','=','.','[',']','{','}','|'
,'`','~',';',':',"'",'"','<',',','>','/','?',' ']
            max_length = 0
            trans1 = trans2 = trans3 = trans4 = trans5 = trans6 = trans7 = trans8 =
trans9 = trans10 = trans11 = trans12 = trans13 = trans14 = trans15 = trans16 =
trans17 = trans18 = trans19 = trans20 = [] #we will only consider 20 transition
matrices... after that, lets just pick trans20 to reuse

            list_of_trans = [trans1,trans2,trans3,trans4,trans5,trans6,trans7,
trans8,trans9,trans10,trans11,trans12,trans13,trans14,trans15,trans16,trans17,tran
s18,trans19,trans20]
            list_of_trans2 = []
            """
            #Use the following if you want to base the various transition matrices
on the longest password in the set -- in rockyou the longest was over 280
characters, too long to actually use!
            for i in database:
                  counter = 0
                  for j in i:
                        counter +=1
                  if counter > max_length:
                        print i
                        max_length = counter #longest password, but we don't want
to do this because its like 286 characters long
                        """
            #print list_of_trans[0]
            list_of_trans2 = []
            counter = 0
            for i in range(20):
                  list_of_trans2.append([])
            for i in list_of_trans2:
                  for j in range(94):
                        list_of_trans2[counter].append([])
                  counter +=1
            counter = 0
            for i in list_of_trans2:
                  counter2 = 0
                  for j in i:
                        for k in range(94):
                              list_of_trans2[counter][counter2].append(0)
                        counter2 +=1
                  counter +=1
            counter = 0
            for i in list_of_trans2[0][0]:
```

```
                counter +=1
            print counter
            list_of_trans2[0][0][0] +=1
            print list_of_trans2[0]


################################################################################
#####

            progress = 0
            for i in database:

                  #print i
                  #Now check that i has no forbidden characters.
                  counter = 0
                  for j in i:
                        if str(j) not in charspace:
                              #print j
                              counter +=1
                  if counter >1: #each line has a hidden character used to signal
next line that's not in charspace. We need counter>1
                        continue

                  elif 2<len(i):

                        counter = 0
                        for j in i[0:-2]:
                              if counter<20:
                                    list_of_trans2[counter]
[charspace.index(i[counter])][charspace.index(i[counter+1])] += 1


                              elif counter >19:

                                    list_of_trans2[19][charspace.index(i[counter])]
[charspace.index(i[counter+1])] += 1
                              counter += 1
                        progress +=1
                  print str(float(progress)/float(562000000) * 100) +"% if you're
doing rockyou"



            for i in list_of_trans2:
                  counti = 0
                  #Lets find the sum of the rows to get a denominator
                  sumofrow = 0
                  for j in i:
                        sumofrow=0
                        for k in j:
                              sumofrow = int(k) + sumofrow
                        #print sumofrow
                        countj = 0
                        for k in j:
                              if sumofrow >= 1:
                                    i[counti][countj] = float(k)/float(sumofrow)
```

```
#To find percentages. Note counti and countj are needed because i,j cannot be
indeces
                                    #print T[counti][countj]
                                    countj +=1
                               elif sumofrow == 0:
                                    i[counti][countj] = 0 #To find percentages.
Note counti and countj are needed because i,j cannot be indeces
                                    #print T[counti][countj]
                                    countj +=1
                          counti +=1



                with open(x+"_index_1.csv", "wb") as f:
                      writer = csv.writer(f)
                      writer.writerows(list_of_trans2[0])
                with open(x+"_index_2.csv", "wb") as f:
                      writer = csv.writer(f)
                      writer.writerows(list_of_trans2[1])
                with open(x+"_index_3.csv", "wb") as f:
                      writer = csv.writer(f)
                      writer.writerows(list_of_trans2[2])
                with open(x+"_index_4.csv", "wb") as f:
                      writer = csv.writer(f)
                      writer.writerows(list_of_trans2[3])
                with open(x+"_index_5.csv", "wb") as f:
                      writer = csv.writer(f)
                      writer.writerows(list_of_trans2[4])
                with open(x+"_index_6.csv", "wb") as f:
                      writer = csv.writer(f)
                      writer.writerows(list_of_trans2[5])
                with open(x+"_index_7.csv", "wb") as f:
                      writer = csv.writer(f)
                      writer.writerows(list_of_trans2[6])
                with open(x+"_index_8.csv", "wb") as f:
                      writer = csv.writer(f)
                      writer.writerows(list_of_trans2[7])
                with open(x+"_index_9.csv", "wb") as f:
                      writer = csv.writer(f)
                      writer.writerows(list_of_trans2[8])
                with open(x+"_index_10.csv", "wb") as f:
                      writer = csv.writer(f)
                      writer.writerows(list_of_trans2[9])
                with open(x+"_index_11.csv", "wb") as f:
                      writer = csv.writer(f)
                      writer.writerows(list_of_trans2[10])
                with open(x+"_index_12.csv", "wb") as f:
                      writer = csv.writer(f)
                      writer.writerows(list_of_trans2[11])
                with open(x+"_index_13.csv", "wb") as f:
                      writer = csv.writer(f)
                      writer.writerows(list_of_trans2[12])
                with open(x+"_index_14.csv", "wb") as f:
                      writer = csv.writer(f)
                      writer.writerows(list_of_trans2[13])
                with open(x+"_index_15.csv", "wb") as f:
                      writer = csv.writer(f)
                      writer.writerows(list_of_trans2[14])
```

```
        with open(x+"_index_16.csv", "wb") as f:
              writer = csv.writer(f)
              writer.writerows(list_of_trans2[15])
        with open(x+"_index_17.csv", "wb") as f:
              writer = csv.writer(f)
              writer.writerows(list_of_trans2[16])
        with open(x+"_index_18.csv", "wb") as f:
              writer = csv.writer(f)
              writer.writerows(list_of_trans2[17])
        with open(x+"_index_19.csv", "wb") as f:
              writer = csv.writer(f)
              writer.writerows(list_of_trans2[18])
        with open(x+"_index_20.csv", "wb") as f:
              writer = csv.writer(f)
              writer.writerows(list_of_trans2[19])



        print "Done with indexed T" + x









#markov_indexed_T("rockyou.txt")
#markov_indexed_T("myspace.txt")
markov_indexed_T("anti_stripped_562_mil.txt")
```

A.6. **Python code to calculate P(password) given data set for standard Markov chains.**

```
"""
Calculate the probability of a password, given a Markov chain whose parameters
were determined by some database
"""

import csv
def calcP(x,y): #x determines the database, y determines the password.

      charspace =
["A","B","C","D","E","F","G","H",'I','J','K','L','M','N','O','P','Q','R','S','T','
U','V','W','X','Y','Z','a','b','c','d','e','f','g','h','i','j','k','l','m','n','o'
,'p','q','r','s','t','u','v','w','x','y','z','1','2','3','4','5','6','7','8','9','
0','!','@','#','$','%','^','&','*','(',')','-','_','+','=','.','[',']','{','}','|'
,'`','~',';',':',"'",'"','<',',','>','/','?',' ']
      pospi = charspace.index(y[0]) #finds where in charspace the first character
of the password is
      #print pospi
      with open(x+"_Pi.csv", 'r') as f: #Open the Pi file for our database
            database = csv.reader(f)
            for row in database:

                  pi = row[pospi] #Grabs the probability of the first state in the
password
            f.close
      #print pi
      counter = 0
      transitions = []
      for i in y:
            if counter < len(y)-1: #get transitions. The last character doesn't
transition
                  t = str(y[counter]) #getting subscripts for a_t,t+1 that will
give us locations in matrix T
                  t1 = str(y[counter +1])
                  transitions.append([charspace.index(t),charspace.index(t1)])
#create a list of lists with our transitions in T
                  #print t + " " + t1
                  counter +=1
      #print transitions

      with open(x+"_T.csv", "r") as f: #Couldn't figure out how to access specific
values of T without loading entire matrix.
            database = csv.reader(f)
            T = []
            for i in database:
                  T.append(i)
            f.close

      prob = float(pi)
      for i in transitions:
            #print T[i[0]][i[1]]
            prob = prob * float(T[i[0]][i[1]])


      print "P("+y+") = " + str(prob)


calcP("rockyou.txt", "Kittycat123!")
calcP("rockyou.txt", "0k@#Hckpdz%5")
```

```
calcP("rockyou.txt", "P455w0rd123!")
calcP("rockyou.txt", "z8U73*!mg@5S")
calcP("myspace.txt", "Kittycat123!")
calcP("myspace.txt", "0k@#Hckpdz%5")
calcP("myspace.txt", "P455w0rd123!")
calcP("myspace.txt", "z8U73*!mg@5S")
calcP("anti_stripped_562_mil.txt", "Kittycat123!")
calcP("anti_stripped_562_mil.txt", "0k@#Hckpdz%5")
calcP("anti_stripped_562_mil.txt", "P455w0rd123!")
calcP("anti_stripped_562_mil.txt", "z8U73*!mg@5S")
```

A.7. Python code to calculate P(password) given data set and Markov chain with memory.

```
"""
calculate probability of password for markov chain with memory = 2
"""

import csv
def calcP_mem(x,y): #x determines the database, y determines the password.

      charspace =
["A","B","C","D","E","F","G","H",'I','J','K','L','M','N','O','P','Q','R','S','T','
U','V','W','X','Y','Z','a','b','c','d','e','f','g','h','i','j','k','l','m','n','o'
,'p','q','r','s','t','u','v','w','x','y','z','1','2','3','4','5','6','7','8','9','
0','!','@','#','$','%','^','&','*','(',')','-','_','+','=','.','[',']','{','}','|'
,'`','~',';',':',"'",'"','<',',','>','/','?',' ']

      memory_matrix = [] #This will eventually be the matrix with our couplets
      for i in charspace:
            for j in charspace:
                  memory_matrix.append(i+j) #This creates said matrix.
      counter = 0
      for i in memory_matrix:
            counter +=1
      #print counter
      first_two = y[0] +y[1]

      pospi = memory_matrix.index(first_two)
      #print pospi

      with open(x+"_mem_Pi.csv", 'r') as f: #Open the Pi file for our database
            database = csv.reader(f)
            for row in database:

                  pi = row[pospi] #Grabs the probability of the first state in the
password
            f.close
      #print pi
      temp_mem = []
      length = len(y)
      counter = 0
      for i in y:
            while counter < length - 1: #-2 bewcause we need to ignore the new line
character
                  temp_mem.append(y[counter]+y[counter+1]) #We now have a matrix
that has all our clusters
                  counter += 1
      #print temp_mem
      length_temp_mem = 0
      for i in temp_mem:
            length_temp_mem +=1
      #print length_temp_mem

      transitions = []
      for i in range(length_temp_mem-1):
            transitions.append([]) #make transitions the right size
      #print transitions
      counter = 0
      while counter < length_temp_mem -1:
            transitions[counter].append(temp_mem[counter])
            transitions[counter].append(temp_mem[counter +1][1])
```

```
            counter +=1
      #print transitions # a matrix with all our transitions
      trans_loc = []
      #for i in transitions:


      with open(x+"_mem_T.csv", "r") as f: #Couldn't figure out how to access
specific values of T without loading entire matrix.
            database = csv.reader(f)
            trans = []
            for i in database:
                  trans.append(i)
            f.close
      #print trans
      #print pi
      #print trans[1][0]
      #print float(pi)* float(trans[27][0])
      prob = float(pi)
      for i in transitions:
            prob = prob * float(trans[memory_matrix.index(i[0])]
[charspace.index(i[1])])

      print "P("+y+") = " + str(prob)

calcP_mem("rockyou.txt", "Kittycat123!")
calcP_mem("rockyou.txt", "0k@#Hckpdz%5")
calcP_mem("rockyou.txt", "P455w0rd123!")
calcP_mem("rockyou.txt", "z8U73*!mg@5S")
calcP_mem("myspace.txt", "Kittycat123!")
calcP_mem("myspace.txt", "0k@#Hckpdz%5")
calcP_mem("myspace.txt", "P455w0rd123!")
calcP_mem("myspace.txt", "z8U73*!mg@5S")
#calcP_mem("anti_stripped_562_mil.txt", "Kittycat123!")
#calcP_mem("anti_stripped_562_mil.txt", "0k@#Hckpdz%5")
#calcP_mem("anti_stripped_562_mil.txt", "P455w0rd123!")
#calcP_mem("anti_stripped_562_mil.txt", "z8U73*!mg@5S")
```

A.8. **Python code to calculate P(password) given data set and indexed Markov chain.**

```
import csv
def calcP(x,y): #x determines the database, y determines the password.

      charspace =
["A","B","C","D","E","F","G","H",'I','J','K','L','M','N','O','P','Q','R','S','T','
U','V','W','X','Y','Z','a','b','c','d','e','f','g','h','i','j','k','l','m','n','o'
,'p','q','r','s','t','u','v','w','x','y','z','1','2','3','4','5','6','7','8','9','
0','!','@','#','$','%','^','&','*','(',')','-','_','+','=','.','[',']','{','}','|'
,'`','~',';',':',"'",'"','<',',','>','/','?',' ']
      pospi = charspace.index(y[0]) #finds where in charspace the first character
of the password is
      #print pospi
      with open(x+"_Pi.csv", 'r') as f: #Open the Pi file for our database
            database = csv.reader(f)
            for row in database:

                  pi = row[pospi] #Grabs the probability of the first state in the
password
            f.close
      counter = 0
      transitions = []
      for i in y:
            if counter < len(y)-1: #get transitions. The last character doesn't
transition
                  t = str(y[counter]) #getting subscripts for a_t,t+1 that will
give us locations in matrix T
                  t1 = str(y[counter +1])
                  transitions.append([charspace.index(t),charspace.index(t1)])
#create a list of lists with our transitions in T
                  #print t + " " + t1
                  counter +=1



      with open(x+"_index_1.csv", "r") as f: #Couldn't figure out how to access
specific values of T without loading entire matrix.
            database = csv.reader(f)
            trans1 = []
            for i in database:
                  trans1.append(i)
            f.close
      with open(x+"_index_2.csv", "r") as f: #Couldn't figure out how to access
specific values of T without loading entire matrix.
            database = csv.reader(f)
            trans2 = []
            for i in database:
                  trans2.append(i)
            f.close
      with open(x+"_index_3.csv", "r") as f: #Couldn't figure out how to access
specific values of T without loading entire matrix.
            database = csv.reader(f)
            trans3 = []
            for i in database:
                  trans3.append(i)
            f.close
      with open(x+"_index_4.csv", "r") as f: #Couldn't figure out how to access
specific values of T without loading entire matrix.
            database = csv.reader(f)
```

```
            trans4 = []
            for i in database:
                  trans4.append(i)
            f.close
      with open(x+"_index_5.csv", "r") as f: #Couldn't figure out how to access
specific values of T without loading entire matrix.
            database = csv.reader(f)
            trans5 = []
            for i in database:
                  trans5.append(i)
            f.close
      with open(x+"_index_6.csv", "r") as f: #Couldn't figure out how to access
specific values of T without loading entire matrix.
            database = csv.reader(f)
            trans6 = []
            for i in database:
                  trans6.append(i)
            f.close
      with open(x+"_index_7.csv", "r") as f: #Couldn't figure out how to access
specific values of T without loading entire matrix.
            database = csv.reader(f)
            trans7 = []
            for i in database:
                  trans7.append(i)
            f.close
      with open(x+"_index_8.csv", "r") as f: #Couldn't figure out how to access
specific values of T without loading entire matrix.
            database = csv.reader(f)
            trans8 = []
            for i in database:
                  trans8.append(i)
            f.close
      with open(x+"_index_9.csv", "r") as f: #Couldn't figure out how to access
specific values of T without loading entire matrix.
            database = csv.reader(f)
            trans9 = []
            for i in database:
                  trans9.append(i)
            f.close
      with open(x+"_index_10.csv", "r") as f: #Couldn't figure out how to access
specific values of T without loading entire matrix.
            database = csv.reader(f)
            trans10 = []
            for i in database:
                  trans10.append(i)
            f.close
      with open(x+"_index_11.csv", "r") as f: #Couldn't figure out how to access
specific values of T without loading entire matrix.
            database = csv.reader(f)
            trans11 = []
            for i in database:
                  trans11.append(i)
            f.close
      with open(x+"_index_12.csv", "r") as f: #Couldn't figure out how to access
specific values of T without loading entire matrix.
            database = csv.reader(f)
            trans12 = []
            for i in database:
```

```
                trans12.append(i)
          f.close
     with open(x+"_index_13.csv", "r") as f: #Couldn't figure out how to access
specific values of T without loading entire matrix.
          database = csv.reader(f)
          trans13 = []
          for i in database:
                trans13.append(i)
          f.close
     with open(x+"_index_14.csv", "r") as f: #Couldn't figure out how to access
specific values of T without loading entire matrix.
          database = csv.reader(f)
          trans14 = []
          for i in database:
                trans14.append(i)
          f.close
     with open(x+"_index_15.csv", "r") as f: #Couldn't figure out how to access
specific values of T without loading entire matrix.
          database = csv.reader(f)
          trans15 = []
          for i in database:
                trans15.append(i)
          f.close
     with open(x+"_index_16.csv", "r") as f: #Couldn't figure out how to access
specific values of T without loading entire matrix.
          database = csv.reader(f)
          trans16 = []
          for i in database:
                trans16.append(i)
          f.close
     with open(x+"_index_17.csv", "r") as f: #Couldn't figure out how to access
specific values of T without loading entire matrix.
          database = csv.reader(f)
          trans17 = []
          for i in database:
                trans17.append(i)
          f.close
     with open(x+"_index_18.csv", "r") as f: #Couldn't figure out how to access
specific values of T without loading entire matrix.
          database = csv.reader(f)
          trans18 = []
          for i in database:
                trans18.append(i)
          f.close
     with open(x+"_index_19.csv", "r") as f: #Couldn't figure out how to access
specific values of T without loading entire matrix.
          database = csv.reader(f)
          trans19 = []
          for i in database:
                trans19.append(i)
          f.close
     with open(x+"_index_20.csv", "r") as f: #Couldn't figure out how to access
specific values of T without loading entire matrix.
          database = csv.reader(f)
          trans20 = []
          for i in database:
                trans20.append(i)
          f.close
```

```
prob = float(pi)

"""
Testing some stuff

h = float(trans1[0][26])
i = float(trans2[26][0])
p = float(pi)
print pi
print h*i *p
"""
counter = 1
for i in transitions:
      if counter == 1:
            prob = prob * float(trans1[i[0]][i[1]])
            counter +=1
      elif counter == 2:
            prob = prob * float(trans2[i[0]][i[1]])
            counter +=1
      elif counter == 3:
            prob = prob * float(trans3[i[0]][i[1]])
            counter +=1
      elif counter == 4:
            prob = prob * float(trans4[i[0]][i[1]])
            counter +=1
      elif counter == 5:
            prob = prob * float(trans5[i[0]][i[1]])
            counter +=1
      elif counter == 6:
            prob = prob * float(trans6[i[0]][i[1]])
            counter +=1
      elif counter == 7:
            prob = prob * float(trans7[i[0]][i[1]])
            counter +=1
      elif counter == 8:
            prob = prob * float(trans8[i[0]][i[1]])
            counter +=1
      elif counter == 9:
            prob = prob * float(trans9[i[0]][i[1]])
            counter +=1
      elif counter == 10:
            prob = prob * float(trans10[i[0]][i[1]])
            counter +=1
      elif counter == 11:
            prob = prob * float(trans11[i[0]][i[1]])
            counter +=1
      elif counter == 12:
            prob = prob * float(trans12[i[0]][i[1]])
            counter +=1
      elif counter == 13:
            prob = prob * float(trans13[i[0]][i[1]])
            counter +=1
      elif counter == 14:
            prob = prob * float(trans14[i[0]][i[1]])
            counter +=1
      elif counter == 15:
            prob = prob * float(trans15[i[0]][i[1]])
            counter +=1
```

```
            elif counter == 16:
                  prob = prob * float(trans16[i[0]][i[1]])
                  counter +=1
            elif counter == 17:
                  prob = prob * float(trans17[i[0]][i[1]])
                  counter +=1
            elif counter == 18:
                  prob = prob * float(trans18[i[0]][i[1]])
                  counter +=1
            elif counter == 19:
                  prob = prob * float(trans19[i[0]][i[1]])
                  counter +=1
            elif counter >= 20:
                  prob = prob * float(trans20[i[0]][i[1]])
                  counter +=1
      print "The probability of " + y +" according to the " + x + " dataset is " +
str(prob)

calcP("rockyou.txt", "Kittycat123!")
calcP("rockyou.txt", "0k@#Hckpdz%5")
calcP("rockyou.txt", "P455w0rd123!")
calcP("rockyou.txt", "z8U73*!mg@5S")
calcP("myspace.txt", "Kittycat123!")
calcP("myspace.txt", "0k@#Hckpdz%5")
calcP("myspace.txt", "P455w0rd123!")
calcP("myspace.txt", "z8U73*!mg@5S")
calcP("anti_stripped_562_mil.txt", "Kittycat123!")
calcP("anti_stripped_562_mil.txt", "0k@#Hckpdz%5")
calcP("anti_stripped_562_mil.txt", "P455w0rd123!")
calcP("anti_stripped_562_mil.txt", "z8U73*!mg@5S")
```