PROBABILISTIC PASSWORD MODELING: PREDICTING PASSWORDS WITH MACHINE LEARNING

JAY DESTORIES
MENTOR: ELIF YAMANGIL

Abstract

Many systems use passwords as the primary means of authentication. As the length of a password grows, the search space of possible passwords grows exponentially. Despite this, people often fail to create unpredictable passwords. This paper will explore the problem of creating a probabilistic model for describing the distribution of passwords among the set of strings. This will help us gain insight into the relative strength of passwords as well as alternatives to existing methods of password candidate generation for password recovery tools like John the Ripper. This paper will consider methods from the field of natural language processing and evaluate their efficacy in modeling human-generated passwords.

Contents

1. Introduction

Passwords are commonly used by the general public for authentication with computer services like social networks and online banking. The purpose of a password, and authentication in general in the scope of computer systems, is to confirm an identity. For instance, when using an online banking service you provide a password so that you can perform actions on an account, but others cannot. This sense of security relies on a password being something that you can remember, but others cannot discover. The security of a system protected by a password is dependent on that password being difficult to guess. This is helped by the fact that an increase in the length of a password exponentially increases the number of possible passwords. Despite this, people often create unoriginal and easy to guess passwords. In a 2014 leak of approximately 5 million gmail account passwords, 47779 users had used the password "123456"[1]. In fact, there are so many instances of different people using the same passwords that one of the dominant ways to guess passwords in password recovery tools like John the Ripper is to simply guess every password in a list of leaked passwords. It is apparent that there is some underlying thought process in coming up with a password that leads to people creating similar or even identical passwords. This paper will attempt to find that structure in the form of a probabilistic model. It will consider some techniques from the field of natural language processing for inferring the structure of the language. If there is a good way to model the language of passwords, it could be used to generate new password guesses that don't already exist in a wordlist, improving the performance of password recovery tools when wordlists are exhausted. It could also evaluate the strength of a password in terms of the expected number of guesses to find it given a model.

2. To the Community

This paper deals mainly with the technical aspects of guessing passwords. The purpose of this paper is to explore the weaknesses of using passwords for authentication. This is an interesting problem because good passwords by definition do not fit well into a predictive model (a good password is hard to guess). We will consider ways of guessing which passwords a person might use and of evaluating the strength of a password, with the intention not only of demonstrating these techniques, but also of encouraging the use of stronger passwords and additional security measures like two-factor authentication.

3. Probability Modeling

3.1. **N-Gram Language Model.** N-grams make sequence modeling practical by making a Markov assumption: the probability of an element given the previous elements is approximately equal to the probability of that element given the previous n elements. For example, an n-gram of order 2 (bigram) assumes that $P(e_j \mid e_1, ..., e_{j-1}) \approx P(e_j \mid e_{j-1})$. Then, by the chain rule the probability of a sequence

$$\begin{aligned} P(e_1, ..., e_n) &= P(e_1)P(e_2 \mid e_1)P(e_3 \mid e_1, e_2)...P(e_n \mid e_1, ..., e_{n-1}) \\ &\approx P(e_1)P(e_2 \mid e_1)P(e_3 \mid e_2)...P(e_n \mid e_{n-1}) \end{aligned}$$

[1]Password corpora referenced are available at: http://tinyurl.com/pwdcorpora

Using transition probabilities like $P(e_2 \mid e_1), P(e_3 \mid e_2)$ rather that considering all previous elements in the sequence makes it easier to extrapolate to other arbitrary sequences. Consider the password "123456789" in the training corpus for a bigram model describing passwords. Using a bigram captures the idea that numeric character sequences are likely to appear in numeric order and would assign high probabilities to sequences like "123". A bigram model consists of the matrix of transition probabilities from all tokens to all other tokens. To train a bigram model on the training corpus, we want to find the maximum likelihood estimate (MLE)– the matrix which maximizes the probability of the training corpus.

$$\theta_{kj} = P(v_j \mid v_k) : 0 \leq \theta_{kj} \leq 1, \Sigma_j \theta kj = 1 \forall k$$

Then the maximum likelihood estimation for θ_{kj}, $\hat{\theta}_{kj}$ is:

$$\hat{\theta}_{kj} = argmax\{P(c_1, ..., c_n)\}$$

Log is monotonic, so we can we can rewrite this as:

$$\hat{\theta}_{kj} = argmax\{logP(c_1, ..., c_n)\}$$

Then by our bigram assumption:

$$\hat{\theta}_{kj} = argmax\{log\Pi_i P(c_1 \mid c_{i-1})\}$$

$$= argmax\{log\Pi_i \Pi_{kj} \theta_{kj}^{\mathbb{1}(c_{i-1}=v_k \wedge c_i = v_j)}\}$$

$$= argmax\{log\Pi_{kj} \theta_{kj}^{\Sigma_i \mathbb{1}(c_{i-1}=v_k \wedge c_i = v_j)}\}$$

$$= argmax\{log\Pi_{kj} \theta_{kj}^{n_{kj}}\}$$

$$= argmax\{\Sigma_{kj} n_{kj} log\theta_{kj}\}$$

Apply Lagrange multiplier, differentiate and set equal to zero to find the maximum subject to our constraints:

$$\frac{\partial}{\partial \theta_{kj}}[\Sigma_k \Sigma_j n_{kj} log\theta_{kj} - \Sigma_k \lambda_k (\Sigma_j \theta_{kj} - 1)] = 0$$

$$\frac{n_{kj}}{\hat{\theta}_{kj}} - \lambda_k = 0$$

$$\frac{n_{kj}}{\hat{\theta}_{kj}} = \lambda_k$$

$$\hat{\theta}_{kj} = \frac{n_{kj}}{\lambda_k}$$

$$\hat{\theta}_{kj} = \frac{n_{kj}}{\Sigma_j n_{kj}}$$

So to get the MLE estimate $\hat{\theta}_{kj}$, add up all transitions from k to j and divide that by the number of transitions from k to anything.

For passwords, consider all transitions in every sequence in the training corpus, including tokens for the start and end of each password. Count the number of times each character appears at the start of a transition, and the number of times each transition appears. The probability of bigram $P(B \mid A)$ is equal to the number of transitions from A to B divided by the number of transitions starting at A. See the appendix for implementation-specific details.

3.2. **Hidden Markov Model.** Hidden Markov models (HMMs) are similar to n-grams in that they model sequences based on the Markov assumption that we can approximate the likelihood of future events by considering only recent ones. In modeling sequences with n-grams, we considered probabilities of transitions between observable events. In the case of password modeling, those events are the observable characters of a password. In HMMs, we instead consider an unobservable sequence of events which influence the observable events. In the context of natural language processing, one example of applications of Hidden Markov models is in part-of-speech tagging. In that case, the hidden sequence is a sequence of parts of speech, each of which relates to a word. An HMM defines the probability of n-grams of the hidden sequence, as well as the probability of an observation at any given hidden state. To train an HMM for a problem like part-of-speech tagging, humans would manually map parts of speech to words in a training corpus, then use computers to count the number of each n-gram $C_{s_1,...,s_n,b}$, the compute probabilities of each n-gram

$$P(s_1, ..., s_n, b) = \frac{C_{s_1,...,s_n,b}}{\Sigma_a C_{s_1,...,s_n,a}}$$

Computing the emission probabilities (probability of an observed event given a hidden state), we would just count up the number of times each observation appears for a particular hidden state, then divide that by the number of times the hidden state appears. This is a supervised training approach. However, using Hidden Markov models to describe passwords is less intuitive. There is no established way to tag the characters in a password, so there are no labeled training corpora. This means that short of coming up with a system for tagging characters and manually tagging the dataset for this purpose, the only option to train an HMM for password modeling is unsupervised training. This begs the question: why should we use Hidden Markov models for this task? Despite no established tagging system, some patterns come to mind when looking at password lists. Some people create passwords based on words or other common sequences with certain characters substituted. A common example of this is interchangably using "e", "E", and "3", "a", "A", and "4", "l", "L", and "1", etc. An HMM could model this relationship well by grouping characters which are commonly substituted for one another into common hidden states, then modeling their relative popularity as an emission probability from that hidden state. Another common pattern in passwords is the inclusion of a birth year in the password. Consider passwords that follow the pattern "word/common phrase" + "birth year". An HMM may be able to tag the digits in a birth year as individual hidden states. This could let it capture aspects of the data like the fact that numerically higher digits occur more frequently in the tens place for a birth year. This being said, we don't know if the HMM will be able to learn these types of specific details. We will train an HMM on passwords using the Baum Welch algorithm. This approach will use a fixed number of hidden states, randomly initialize our transition and emission probabilities, then use Expectation Maximization to approach a local maximum for the likelihood of the training corpus. A bigram HMM language model for passwords is implemented in the appendix code, however for testing the model we will use NLTK's implementation. NLTK's version of the HMM is more efficicent and will allow for a wider range of tests. For practical applications, it may be useful to optimize our models.

4. Generating Passwords

For each model, we will consider a way of sampling passwords one at a time. We can then evaluate the performance of a model based on how many unique passwords which exist in the test corpus it generates given some fixed number of attempts. For the training corpus, we will use a list of 14,344,391 passwords exposed in a Rockyou leak. For the test corpus, we will use all but the last 5,000 passwords from a list of 3,135,920 passwords exposed in a Gmail leak. We will keep the last 5,000 passwords from the Gmail leak as a held-out corpus[2].

4.1. **Baseline.**

The baseline is based on the default wordlist attack in John the Ripper, a popular password recovery tool. It will yield every password in the training corpus, in order. Once it has exhausted the training corpus, it will incrementally try all passwords given a vocabulary of characters.

4.2. **Bigram Language Model.**

To generate a password from the bigram model, we will set our current character to the start token, then repeatedly do a weighted random selection of a next character given the transition probabilities until transitioning to the end token.

4.3. **Hidden Markov Model.**

Random sampling from an HMM is similar to sampling from an n-gram, except instead of sampling transitions between observations, we sample transitions between hidden states, then for each step also sample an emission for that hidden state. For the purposes of this paper, we can use NLTK's implementation of an HMM, which provides a method to perform random sampling. The main difference between sampling from our HMM implementation and sampling from the NLTK implementation is that the NLTK version specifies a length for the generated sequence, whereas our implementation ends the sequence when it encounters a transition to the end of the sequence, or hits a hard cap on length. This discrepancy may skew the results, but the increased performance allows for more testing.

4.4. **Results.**

[2]Password corpora referenced are available at: http://tinyurl.com/pwdcorpora

Model	Guesses	Training Passwords Used	Correct Guesses
Baseline	100,000	10	4,612
Baseline	100,000	100	6,278
Baseline	100,000	1,000	6,011
Baseline	100,000	14,344,391	81,494
Bigram LM	100,000	10	45
Bigram LM	100,000	100	2,540
Bigram LM	100,000	1,000	4,215
Bigram LM	100,000	14,344,391	3,985
HMM (10 hidden states)	100,000	10	315
HMM (10 hidden states)	100,000	100	1,531
HMM (10 hidden states)	100,000	1,000	1,760
HMM (50 hidden states)	100,000	10	99
HMM (50 hidden states)	100,000	100	1,795
HMM (50 hidden states)	100,000	1,000	2,798
HMM (100 hidden states)	100,000	10	80
HMM (100 hidden states)	100,000	100	1,430
HMM (100 hidden states)	100,000	1,000	2,913

This paper will not consider the efficacy of HMMs trained with more than 1000 passwords, since training time gets too long. This is worth considering as a drawback to using unsupervised learning for password predictor models.

Since the baseline is based on the behavior of a popular password recovery tool, it is unsurprising that it outperforms the language models. The results show that the best precision in a language model was from the bigram model trained on the first 1000 passwords from the training corpus. This doesn't necessarily mean that the HMM shouldn't be used. Since the bigram model and the HMM model the language differently and are sampled differently, it is possible that each may be likely to guess passwords that the other could not. Most encouraging is that all of the models trained on 1000 or fewer inputs were able to guess more passwords than they were given in 100000 tries. This demonstrates that they have the capability to extrapolate from the input, making them potentially useful additions to the baseline for password recovery. Since the strength of a password is based on how difficult it is to guess, this means that these models are useful for evaluating password strength. If we can expect them to guess a given password more quickly than the baseline, then that password is weaker than we may have otherwise thought.

5. Applications

5.1. **Password Strength.** If we conclude that these models are good at guessing passwords, then we could estimate the strength of a password by the expected number of guesses the models would need to make to guess it. We care most about the worst case scenario, so we will say that strength relates to the minimum expected number of guesses from a model.

Given a password *password* and a model M which evaluates $P(password \mid M) = p$, define the Bernoulli random variable C as 1 if we guess *password* and 0 if we do not guess *password* in some trial. The expectation $E(C)$ for a Bernoulli random variable is equal to its probability, p. Therefore the expected number of guesses to

find *password* is n, satisfying:

$$1 = E(\Sigma_{k=1}^{n} C_k)$$

For $E(C) = E(C_1) = ... = E(C_n)$

$$E(\Sigma_{k=1}^{n} C_k) = \Sigma_{k=1}^{n} E(C_k) = nE(C_k) = 1$$

$$\implies n = \frac{1}{E(C_k)} = \frac{1}{p}$$

But p is, in many cases, small enough to cause floating point imprecision, so we deal with $log(p)$ in implementation.

$$n = \frac{1}{exp(log(p))} = exp(-log(p))$$

For the baseline, the order of passwords guesses is known so the guess number for any particular password is known.
See the appendix for a password strength estimate implementation.

5.2. **Password Recovery.** Password recovery tools function by guessing passwords, one at a time, until one is correct. In most cases, this means taking a hashed password and hashing password guesses until the hashes match. The baseline used in this paper is based on the wordlist attack behavior for John the Ripper, a popular password recovery tool. The models discussed in this paper for guessing passwords could easily be applied in password recovery tools. For recovering passwords based on a hash, we would just need to implement the hashing algorithm, then hash each guess and compare it against the original password hash. These models would be useful as additional techniques for guessing passwords if they are likely to guess some probable passwords that existing tools are unlikely to guess. They are therefore potentially useful primarily for guessing passwords which do not occur in wordlists. When given training sets of 10, 100, and 1000 passwords, both the bigram languge model and the HMM are able to extrapolate, correctly guessing many more unique passwords than they were given. This is encouraging. However, in all cases the baseline outperforms these models. This does not mean that the models are not useful, just that they should not replace the baseline technique, but rather be used in parallel.

6. Future Work

Further work in modeling passwords could involve both improving the models described in this paper as well as building new ones. It would be interesting to try modeling passwords with trigrams, four-grams, etc., or with a context-free grammar. An obvious weakness of these models in generating passwords is that there is nothing stopping them from generating the same passwords repeatedly. If we could prune the set of possible generated passwords, it could improve performance by preventing this type of redundancy. If we store every generated password and exclude it from being generated again, the bigram language model's correct guesses goes up from 3,985 to 4,771 when trained on the full training corpus. This exact approach isn't practical because it requires storing every generated password in memory, but it demonstrates the potential improvement with some sort of pruning. Another apparent weakness in this paper is the lack of sufficient computational resources to

properly test all models. The forward backward algorithm used to train the HMM takes too long on consumer grade hardware to test it with large training sets. However, many people or organizations who would be interested in guessing passwords have access to much more powerful hardware, and training could be further sped up with a better optimized implementation, for example by taking advantage of graphics processors to parallelize the task. Therefore, it would be interesting to see how the HMM performs with much larger training sets.

This paper considered the precision of models at guessing passwords. It would also be interesting to know the recall– what portion of the test corpus we are able to guess. This is harder to test and is less relevant when computational resources are scarce. This is similar to the problem of estimating password strength, in that it considers the likelihood of a password being guessed.

7. Conclusion

We've shown that using language models to guess passwords is effective enough to encourage further exploration. Although none could beat the precision of traditional password recovery techniques, all models were able to extrapolate based on an input set of passwords to a set containing a larger number of correct passwords. This paper outlines one method for evaluating the strength of a password. That being said, it provides no guarrantees for the strength of a password. It is most effective at recognizing passwords that are weak.

The important take-away from this paper is how effectively we can guess passwords, even with just traditional methods. Given how much we rely on passwords for critical security, it is important to pick passwords which are harder to guess, but also to keep in mind that regardless of how long or seemingly complex some passwords might be, they may still be vulnerable. Therefore it is important not to rely solely on passwords for security. Other security measures like two-factor authentication improve our chances at keeping systems secure.

References

[1] Bolch, G.; Greiner, S.; Meer, H.; Trivedi, K. (2006) Queueing Networks and Markov Chains *John Wiley & Sons, Inc.*

[2] Ghahramani, Z. (2001) An Introduction to Hidden Markov Models and Bayesian Networks. *International Journal of Pattern Recognition and Artificial Intelligence.*

[3] Rabiner, L.R., & Juang, B.H. (1986) An Introduction to Hidden Markov Models. *IEEE ASSP Magazine*

[4] Rosenfeld, Ronald. Two Decades of Statistical Language Modeling: Where do we go from here? www.cs.cmu.edu/~roni/papers/survey-slm-IEEE-PROC-0004.pdf

[5] Seneta, E. (1981) Non-negative matrices and Markov chains. 2nd rev. ed. *Springer Series in Statistics*

8. Appendix

8.1. Language Model Evaluation.

```
"""
    Test the performance of various password models
"""

import lmgenerator
import hmmgenerator

# basic wordlist attack
def baselineGenerator(training_corpus):
    for pwd in training_corpus:
        yield pwd
    vocabulary = set()
    for pwd in training_corpus:
        for c in pwd:
            vocabulary.update([c])
    vocabulary = list(vocabulary)
    def bruteForce(vocabulary, length):
        if length == 1:
            for c in vocabulary:
                yield c
        else:
            for c in vocabulary:
                for password_end in bruteForce(vocabulary, length - 1):
                    yield c + password_end
    length = 1
    while True:
        for pwd in bruteForce(vocabulary, length):
            yield pwd
        length += 1

def passwordStrength(password, training_corpus, bigramlm, hmm):
    expectations = []
    # baseline
    baseline_guess = 0
    if password in training_corpus:
        baseline_guess = training_corpus.index(password) + 1
    else:
        vocabulary = set()
        for pwd in training_corpus:
            for c in pwd:
                vocabulary.update([c])
        vocabulary = list(vocabulary)
        guessable = True
        for c in password:
            if c not in vocabulary:
                guessable = False
        if not guessable:
            baseline_guess = float('inf')
        else:
            baseline_guess = len(training_corpus)
```

```
51              for pwd_len in xrange(1, len(password)):
52                  baseline_guess += pow(len(vocabulary), pwd_len)
53              def helper(pwd):
54                  if len(pwd) == 1:
55                      return vocabulary.index(pwd[0]) + 1
56                  return pow(len(vocabulary), len(pwd) - 1) * vocabulary.index(pwd[0]) + \
57                          helper(pwd[1:])
58              baseline_guess += helper(password)
59      expectations.append(baseline_guess)
60
61      if bigramlm:
62          expectations.append(bigramlm.ExpectedGuesses(password))
63      if hmm:
64          expectations.append(hmm.ExpectedGuesses(password))
65      return min(expectations)
66
67  # See how many things in test_corpus the generator can guess with some number of
68  # tries
69  def testGenerator(gen, test_corpus, tries):
70      found = 0
71      test_set = set(test_corpus)
72      guesses = set()
73      for i in xrange(tries):
74          guess = gen.next()
75          if not guess in guesses:
76              guesses.update([guess])
77              if guess in test_set:
78                  found += 1
79      return found
80
81  def testCorpora(training_corpus, test_corpus):
82      print "First 5 training passwords: ", training_corpus[:5]
83      print "First 5 test passwords: ", test_corpus[:5]
84
85      tries = 100000
86      baseline = testGenerator(baselineGenerator(training_corpus), test_corpus, tries)
87      print "Baseline wordlist attack -- %d tries: %d." % (tries, baseline)
88      bigramlm = lmgenerator.BigramLM(training_corpus)
89      bigramlmgen = bigramlm.Generator()
90      bigramlmresults = testGenerator(bigramlmgen, test_corpus, tries)
91      print "Bigram LM attack -- %d tries: %d." % (tries, bigramlmresults)
92      hmmlm = hmmgenerator.NLTKHMMLM(training_corpus, 100)
93      hmmgen = hmmlm.Generator()
94      hmmresults = testGenerator(hmmgen, test_corpus, tries)
95      print "HMM attack -- %d tries: %d." % (tries, hmmresults)
96      print "Password strength test:"
97      print "123456:", passwordStrength("123456", training_corpus, bigramlm, hmmlm)
98      print "123456789123:", passwordStrength("123456789123", training_corpus, bigramlm, h
99      print "mingchow:", passwordStrength("mingchow", training_corpus, bigramlm, hmmlm)
100     print "0a0I8DV:", passwordStrength("0a0I8DV", training_corpus, bigramlm, hmmlm)
101     print "AtiK0nAOLP3y:", passwordStrength("AtiK0nAOLP3y", training_corpus, bigramlm, h
102     print "correcthorsebatterystaple:", passwordStrength("correcthorsebatterystaple", \
103                                         training_corpus, bigramlm, hmmlm)
```

```
104
105
106
107 def main():
108     print "#########################################################################"
109     print "Training corpus: rockyou"
110     print "Test corpus: gmail"
111     print "#########################################################################"
112     rockyou_nocount = open('corpora/rockyou_nocount', 'r')
113     training_corpus = [pwd.rstrip() for pwd in rockyou_nocount][:10]
114     gmail_nocount = open('corpora/gmail_nocount', 'r')
115     gmail_corpus = [pwd.rstrip() for pwd in gmail_nocount]
116     test_corpus = gmail_corpus[:-5000]
117     held_out_corpus = gmail_corpus[-5000:]
118     testCorpora(training_corpus, test_corpus)
119
120
121 if __name__ == "__main__":
122     main()
```

8.2. **Bigram Language Model Implementation.**

```
from math import log, exp, isnan
import random
from decimal import *

start_token = "<S>"
end_token = "</S>"

def Preprocess(corpus):
    return [[start_token] + [token for token in pwd] + [end_token] for pwd in corpus]

class BigramLM:
    def __init__(self, training_corpus):
        self.bigram_counts = {}
        self.unigram_counts = {}
        self.Train(training_corpus)

    def Train(self, training_corpus):
        training_set = Preprocess(training_corpus)
        for pwd in training_set:
            for i in xrange(len(pwd) - 1):
                token = pwd[i]
                next_token = pwd[i + 1]
                if not token in self.unigram_counts:
                    self.unigram_counts[token] = 0
                if not token in self.bigram_counts:
                    self.bigram_counts[token] = {}
                if not next_token in self.bigram_counts[token]:
                    self.bigram_counts[token][next_token] = 0
                self.unigram_counts[token] += 1
                self.bigram_counts[token][next_token] += 1

    def GenerateSample(self):
        sample = [start_token]
        while not sample[-1] == end_token:
            selector = random.uniform(0, self.unigram_counts[sample[-1]])
            sum_bc = 0
            for bigram in self.bigram_counts[sample[-1]]:
                sum_bc += self.bigram_counts[sample[-1]][bigram]
                if sum_bc > selector:
                    sample.append(bigram)
                    break
        return ''.join(sample[1:-1])

    # gets the (unsmoothed) probability of a string given the bigramlm
    def StringLogProbability(self, string):
        preprocessed = Preprocess([string])[0]
        logprob = 0
        for i in xrange(1, len(preprocessed)):
            unigram = preprocessed[i - 1]
            bigram = preprocessed[i]
            if unigram in self.unigram_counts and \
               unigram in self.bigram_counts and \
```

```
53                 bigram in self.bigram_counts[unigram]:
54                  logprob += log(self.bigram_counts[unigram][bigram]) - \
55                             log(self.unigram_counts[unigram])
56             else:
57                 logprob = float('-inf')
58         return logprob
59
60     def ExpectedGuesses(self, string):
61         logprob = self.StringLogProbability(string)
62         try:
63             expectation = Decimal(-logprob).exp()
64             return expectation if not isnan(expectation) else float('inf')
65         except:
66             return float('inf')
67
68     def Generator(self):
69         while True:
70             yield self.GenerateSample()
71
72     def SimplePrunedGenerator(self):
73         tries = set()
74         while True:
75             pwd = self.GenerateSample()
76             if not pwd in tries:
77                 tries.update([pwd])
78                 yield pwd
```

8.3. **Hidden Markov Model Implementation.**

```
from math import log, exp, log1p, isnan
import random
from memoize import memoize
import nltk
from decimal import *

class HMMLM:
    def __init__(self, training_corpus, num_states):
        sequences = [[(c, "") for c in pwd] for pwd in training_corpus]
        symbols = list(set([c for pwd in training_corpus for c in pwd]))
        states = range(num_states)
        trainer = nltk.tag.hmm.HiddenMarkovModelTrainer(states=states, symbols=symbols)
        self.hmm = trainer.train_unsupervised(sequences)

    def Sample(self, range_start, range_end):
        pwd = self.hmm.random_sample(random.Random(), random.randint(range_start, range_
        pwd = "".join([e[0] for e in pwd])
        return pwd

    def StringProbability(self, pwd):
        return self.hmm.log_probability([(c, None) for c in pwd])

    def ExpectedGuesses(self, pwd):
        logprob = self.StringProbability(pwd)
        try:
            expectation = Decimal(-logprob).exp()
            return expectation if not isnan(expectation) else float('inf')
        except:
            return float('inf')

    def Generator(self):
        while True:
            pwd = self.hmm.random_sample(random.Random(), random.randint(4, 18))
            pwd = "".join([e[0] for e in pwd])
            yield pwd


start_token = "<S>"
end_token = "</S>"
wildcard_token = "<*>"

# reduce floating point imprecision in adding probabilities in log space
def SumLogProbs(lps):
    # ln(e^lp1 + e^lp2) == ln(e^lp2 (e^(lp1 - lp2) + 1)) = ln(e^(lp1 - lp2) + 1) + lp2
    def adderhelper(lp1, lp2):
        if lp1 == float('-inf') and lp2 == float('-inf'):
            return float('-inf')
        return log1p(exp(lp1 - lp2)) + lp2 if lp2 > lp1 else log1p(exp(lp2 - lp1)) + lp1
    return reduce(adderhelper, lps)


def Preprocess(corpus):
```

```
53      return [[start_token] + [token for token in pwd] + [end_token] for pwd in corpus]
54
55  # gets a count-length array of random probabilities summing to s
56  def RandomPartition(count, s):
57      if count is 1:
58          return [s]
59      split_prob = (random.random() * .4 + .2) * s
60      split_count = count / 2
61      return RandomPartition(split_count, split_prob) + \
62             RandomPartition(count - split_count, s - split_prob)
63
64  # gets an array of log probabilities [p1, p2, ...] where e^p1 + e^p2 + ... = 1
65  def RandomLogProbs(count):
66      total = 4000000000
67      partition = RandomPartition(count, total)
68      return [log(p) - log(total) for p in partition]
69
70
71  class HMMLM:
72      def __init__(self, training_corpus, state_count):
73          vocabulary = set()
74          for pwd in training_corpus:
75              vocabulary.update([c for c in pwd])
76          self.o_vocabulary = set(vocabulary)
77          self.states = range(state_count)
78          self.transition_probability = {state1: {state2: prob for (state2, prob) in \
79              zip(self.states + [end_token], RandomLogProbs(state_count + 1))} \
80              for state1 in self.states}
81          self.transition_probability[start_token] = {state: prob for (state, prob) in \
82              zip(self.states, RandomLogProbs(state_count))}
83          self.emission_probability = {state: {symbol: prob for (symbol, prob) in \
84              zip(vocabulary, RandomLogProbs(len(vocabulary)))} for state in self.states}
85          self.emission_probability[start_token] = {start_token: 0}
86          self.emission_probability[end_token] = {end_token: 0}
87          self.ForwardBackward(training_corpus)
88
89
90      def ForwardMatrix(self, pwd):
91          bp = [{state: None for state in self.states} for c in pwd]
92
93          # initialization
94          bp.insert(0, {start_token: 0})
95          bp.append({end_token: None})
96          ppwd = [start_token] + [c for c in pwd] + [end_token]
97
98          # recursion
99          for i in xrange(1, len(pwd) + 2):
100             bp[i] = {state: SumLogProbs(map(lambda p: bp[i - 1][p] + \
101                                 self.transition_probability[p][state] + \
102                                 self.emission_probability[state][ppwd[i]], bp[i - 1])) \
103                     for state in bp[i]}
104
105         return bp
```

```
106
107
108     def BackwardMatrix(self, pwd):
109         bp = [{state: None for state in self.states} for c in pwd]
110
111         # initialization
112         bp.append({end_token: 0})
113         bp.insert(0, {start_token: None})
114         ppwd = [start_token] + [c for c in pwd] + [end_token]
115
116         # recursion
117         for i in reversed(xrange(0, len(pwd) + 1)):
118             bp[i] = {state: SumLogProbs(map(lambda n: bp[i + 1][n] + \
119                             self.transition_probability[state][n] + \
120                             self.emission_probability[n][ppwd[i + 1]], bp[i + 1])) \
121                      for state in bp[i]}
122
123         return bp
124
125     def TransitionMatrix(self, alpha, beta, pwd):
126         length = len(pwd) + 2
127         ppwd = [start_token] + [c for c in pwd] + [end_token]
128         m = [{i: {} for i in self.states + [start_token]} for t in xrange(length - 1)]
129         for t in xrange(length - 1):
130             for start in self.states + [start_token]:
131                 for end in self.states + [end_token]:
132                     if start not in alpha[t] or end not in beta[t + 1]:
133                         m[t][start][end] = float('-inf')
134                     else:
135                         m[t][start][end] = alpha[t][start] + \
136                                            self.transition_probability[start][end] + \
137                                            self.emission_probability[end][ppwd[t + 1]] +
138                                            beta[t + 1][end] - alpha[length - 1][end_toke
139         return m
140
141
142     def TimeStateMatrix(self, alpha, beta, pwd):
143         length = len(pwd) + 2
144         m = [{} for t in xrange(length)]
145         for t in xrange(length):
146             for j in self.states + [start_token, end_token]:
147                 if j not in alpha[t] or j not in beta[t]:
148                     m[t][j] = float('-inf')
149                 else:
150                     m[t][j] = alpha[t][j] + beta[t][j] - alpha[length - 1][end_token]
151         return m
152
153
154     def ForwardBackward(self, corpus):
155         max_length = max([len(pwd) for pwd in corpus]) + 2
156         last_prob = None
157         x = 0
158         while True:
```

```
159             print "iteration:", x
160             log_prob = 0
161
162             ksi_all = [{i: {j: float('-inf') for j in self.states + [end_token]} \
163                 for i in self.states + [start_token]} for t in xrange(max_length - 1)]
164             gamma_all = [{j: {} for j in self.states + [start_token, end_token]} \
165                 for t in xrange(max_length)]
166
167             for pwd in corpus:
168                 alpha = self.ForwardMatrix(pwd)
169                 beta = self.BackwardMatrix(pwd)
170                 ksi = self.TransitionMatrix(alpha, beta, pwd)
171                 gamma = self.TimeStateMatrix(alpha, beta, pwd)
172                 length = len(pwd) + 2
173                 log_prob += alpha[length - 1][end_token]
174                 for t in xrange(length - 1):
175                     for start in self.states + [start_token]:
176                         for end in self.states + [end_token]:
177                             ksi_all[t][start][end] = SumLogProbs([ksi_all[t][start][end]
178                                                                   ksi[t][start][end]])
179                 ppwd = [start_token] + [c for c in pwd] + [end_token]
180                 for t in xrange(length):
181                     for j in self.states + [start_token, end_token]:
182                         if ppwd[t] not in gamma_all[t][j]:
183                             gamma_all[t][j][ppwd[t]] = float('-inf')
184                         gamma_all[t][j][ppwd[t]] = SumLogProbs([gamma_all[t][j][ppwd[t]]
185                                                                 gamma[t][j]])
186
187             # re-estimate transition probabilities using ksi_all
188             for start in self.states + [start_token]:
189                 transition_den = float('-inf')
190                 for end in self.states + [end_token]:
191                     for t in xrange(max_length - 1):
192                         transition_den = SumLogProbs([transition_den, ksi_all[t][start][
193                 for end in self.states + [end_token]:
194                     transition_num = float('-inf')
195                     for t in xrange(length - 1):
196                         transition_num = SumLogProbs([transition_num, ksi_all[t][start][
197                     self.transition_probability[start][end] = transition_num - transitio
198
199             # re-estimate emission probabilities using gamma_all
200             for j in self.states + [start_token, end_token]:
201                 emission_den = float('-inf')
202                 for t in xrange(max_length):
203                     emission_den = SumLogProbs([emission_den] + gamma_all[t][j].values()
204                 for e in self.emission_probability[j]:
205                     emission_num = float('-inf')
206                     for t in xrange(max_length):
207                         if e in gamma_all[t][j]:
208                             emission_num = SumLogProbs([emission_num, gamma_all[t][j][e]
209                     self.emission_probability[j][e] = emission_num - emission_den
210
211             print "log prob:", log_prob
```

```
212             if last_prob is not None and log_prob - last_prob < .0001:
213                 break
214             last_prob = log_prob
215             x += 1
216 
217 
218     def GenerateSample(self):
219         state = start_token
220         sample = []
221         while len(sample) < 18:
222             transitions = [(next_state, \
223                 Decimal(self.transition_probability[state][next_state]).exp()) \
224                 for next_state in self.transition_probability[state]]
225             selector = random.uniform(0, 1)
226             sum_p = 0
227             for (next_state, dec_prob) in transitions:
228                 sum_p += dec_prob
229                 if sum_p >= selector:
230                     state = next_state
231                     break
232             if state == end_token:
233                 break
234             emissions = [(e, Decimal(self.emission_probability[state][e]).exp()) \
235                         for e in self.emission_probability[state]]
236             selector = random.uniform(0, 1)
237             sum_p = 0
238             for (e, dec_prob) in emissions:
239                 sum_p += dec_prob
240                 if sum_p >= selector:
241                     sample.append(e)
242                     break
243         print ''.join(sample)
244         return ''.join(sample)
245 
246     def Generator(self):
247         while True:
248             yield self.GenerateSample()
```