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

Since ancient times, passwords have been used as a means for authentication. Given the rise of personal computers and the internet, passwords are now commonly used by the general public for authentication with services like social networks or 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". 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. Classifying the Language of Passwords

2.1. Regular Languages.

2.2. Context-Free Languages.

2.3. Context-Sensitive and Recursively Enumerable Languages.

3. Probability Modeling

3.1. N-Gram Language Model.

3.2. Hidden Markov Model.

3.3. Context-Free Grammar.

4. Generating Passwords

4.1. Baseline.

4.2. N-Gram Language Model.

4.3. Hidden Markov Model.

4.4. Context-Free Grammar.

5. Applications

5.1. Password Strength.

5.2. Password Recovery.

6. Conclusion

References

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

[2] 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

7. Appendix

7.1. Language Model Evaluation.

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

import lmgenerator

# basic wordlist attack
def baselineGenerator(training_corpus):
    for pwd in training_corpus:
        yield pwd
    while True:
        yield ""

# See how many things in test_corpus the generator can guess with some number of
# tries
def testGenerator(gen, test_corpus, tries):
    found = 0
    test_set = set(test_corpus)
    guesses = set()
    for i in xrange(tries):
        guess = gen.next()
        if not guess in guesses:
            guesses.update([guess])
            if guess in test_set:
                found += 1
    return found

def testCorpora(training_corpus, test_corpus):
    print "First 5 training passwords: ", training_corpus[:5]
    print "First 5 test passwords: ", test_corpus[:5]

    tries = 100000
    baseline = testGenerator(baselineGenerator(training_corpus), test_corpus, tries)
    print "Baseline wordlist attack -- %d tries: %d." % (tries, baseline)
    bigramlmgen = lmgenerator.SimplePrunedBigramLMGenerator(training_corpus)
    bigramlm = testGenerator(bigramlmgen, test_corpus, tries)
    print "Bigram LM attack -- %d tries: %d." % (tries, bigramlm)


def main():
    print "######################################################################"
    print "Training corpus: rockyou"
    print "Test corpus: gmail"
    print "######################################################################"
    rockyou_nocount = open('corpora/rockyou_nocount', 'r')
    training_corpus = [pwd.rstrip() for pwd in rockyou_nocount]
    gmail_nocount = open('corpora/gmail_nocount', 'r')
    gmail_corpus = [pwd.rstrip() for pwd in gmail_nocount]
    test_corpus = gmail_corpus[:-5000]
    held_out_corpus = gmail_corpus[-5000:]
```

```
    testCorpora(training_corpus, test_corpus)


if __name__ == "__main__":
    main()
```

7.2. **N-Gram Language Model Implementation.**

```
from math import log, exp
import random

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):
        self.bigram_counts = {}
        self.unigram_counts = {}

    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):


def BigramLMGenerator(training_corpus):
```

```
    lm = BigramLM()
    lm.Train(training_corpus)
    while True:
        yield lm.GenerateSample()

def SimplePrunedBigramLMGenerator(training_corpus):
    tries = set()
    gen = BigramLMGenerator(training_corpus)
    while True:
        pwd = gen.next()
        if not pwd in tries:
            tries.update([pwd])
            yield pwd
```

7.3. **Hidden Markov Model Implementation.**

```
from math import log, exp, log1p
import random
from memoize import memoize

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):
        return log1p(exp(lp1 - lp2)) + lp2 if lp2 > lp1 else log1p(exp(lp2 - lp1)) + lp1
    return reduce(adderhelper, lps)


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

# gets a count-length array of random probabilities summing to s
def RandomPartition(count, s):
    if count is 1:
        return [s]
    split_prob = (random.random() * .4 + .2) * s
    split_count = len(count) / 2
    return RandomPartition(split_count, split_prob) + \
           RandomPartition(count - split_count, s - split_prob)

# gets an array of log probabilities [p1, p2, ...] where e^p1 + e^p2 + ... = 1
def RandomLogProbs(count):
    total = 4000000000
    partition = RandomPartition(count, total)
    return [log(p) - log(total) for p in partition]


class BigramHMM:
    def __init__(self, vocabulary, state_count):
        self.o_vocabulary = set(vocabulary)
```

```
        self.states = range(state_count)
        self.start_probability = {state: prob for (state, prob) in zip(self.states, Rand
        self.transition_probability = {state1: {state2: prob for (state2, prob) in (self
        self.end_probability = {state: prob for (state, prob) in zip(self.states, Randon
        self.emission_probability = {state: {symbol: prob for (symbol, prob) in zip(voca

    @memoize
    def ForwardMatrix(pwd):
        bp = [{state: None for state in self.states} for c in pwd]

        # initialization
        bp[0] = {state: self.start_probability[state] + self.emission_probability[state]

        # recursion
        for i in xrange(1, len(pwd)):
            bp[i] = {state: SumLogProbs(map(lambda p: bp[i - 1][p] + self.transition_pro

        return bp


    @memoize
    def BackwardMatrix(pwd):
        bp = [{state: None for state in self.states} for c in pwd]

        # initialization
        bp[len(pwd) - 1] = {state: self.end_probability[state] for state in self.states}

        # recursion
        for i in reversed(xrange(0, len(pwd) - 1)):
            bp[i] = {state: SumLogProbs(map(lambda n: bp[i + 1][n] + self.transition_pro

        return bp


    @memoize
    def ForwardProbability(step, state, pwd):
        matrix = self.ForwardMatrix(pwd)
        if state == wildcard_token:
            return SumLogProbs(matrix[step].values())
        return matrix[step][state]
        """
        if step is 0:
            return self.start_probability[end_state]

        bp = [{state: None for state in self.states} for c in pwd]

        # initialization
        bp[0] = {state: self.start_probability[state] + self.emission_probability[state]

        # recursion
        for i in xrange(1, step - 1):
            bp[i] = {state: sum(map(lambda p: bp[i - 1][p] + self.transition_probability
```

```
        # termination
        if end_state == wildcard_token:
            return sum(map(lambda state: sum(map(lambda p: bp[step - 1][p] + self.transi

        return sum(map(lambda p: bp[step - 1][p] + self.transition_probability[p][end_st
    """

    @memoize
    def BackwardProbability(step, state, pwd):
        matrix = self.BackwardMatrix(pwd)
        return matrix[step][state]
        """
        last_step = len(pwd) - 1
        if step == last_step:
            return self.end_probability[start_state]

        bp = [{state: None for state in self.states} for c in pwd]

        # initialization
        bp[last_step] = {state: self.end_probability[state] for state in self.states}

        # recursion
        for i in reversed(xrange(step + 1, last_step - 1)):
            bp[i] = {state: sum(map(lambda n: bp[i + 1][n] + self.transition_probability

        # termination
        return sum(map(lambda n: bp[step + 1][n] + self.transition_probability[start_sta
    """

    @memoize
    def TimeStateProbability(step, state, pwd):
        return self.ForwardProbability(step, state, pwd) + \
               self.BackwardProbability(step, state, pwd) - \
               self.ForwardProbability(len(pwd) - 1, wildcard_token, pwd)

    @memoize
    def StateTransitionProbability(step, state1, state2, pwd):
        return self.ForwardProbability(step, state1, pwd) + \
               self.BackwardProbability(step + 1, state2, pwd) + \
               self.transition_probability[state1][state2] + \
               self.emission_probability[state2][pwd[step + 1]] - \
               self.ForwardProbability(len(pwd) - 1, wildcard_token, pwd)

    def ForwardBackward():
        # for now assume convergence in constant number of iterations
        for i in xrange(10):
            # expectation

            # maximization

            # reset memos
            self.ForwardMatrix.reset()
            self.BackwardMatrix.reset()
```

```
            self.ForwardProbability.reset()
            self.BackwardProbability.reset()
            self.TimeStateProbability.reset()
            self.StateTransitionProbability.reset()
```

7.4. **Context-Free Grammar Implementation.**