Contents

[Security Tools Lab 1 1](#_Toc85562224)

[Project 4 – Passwords 1](#_Toc85562225)

[1. Introduction 3](#_Toc85562226)

[2. Bigram Language Model 4](#_Toc85562227)

[2.1 N-gram Language Model 4](#_Toc85562228)

[2.2 Model Implementation 5](#_Toc85562229)

[3. Data Collection 7](#_Toc85562230)

[3.1 Dashlane 8](#_Toc85562231)

[3.2 Avast 9](#_Toc85562232)

[3.3 Passwords Generator (passwordsgenerator.net) 9](#_Toc85562233)

[3.4 Lastpass 10](#_Toc85562234)

[3.5 Roboform 10](#_Toc85562235)

[3.6 Zoho 10](#_Toc85562236)

[4. Password Strength 11](#_Toc85562237)

[5. Limitations and Extensibility 13](#_Toc85562238)

[6. Conclusion 13](#_Toc85562239)

## Introduction

In the context of electronic systems and digital media, passwords are a common and standard feature of the authentication process. Passwords serve to confirm the identity of a user, before granting said user his corresponding access and rights. This control process therefore relies on information asymmetry - users only know their own passwords, or possess enough information to make reliable guesses of their passwords. Other users should not have the ability to reliably guess their passwords.

In light of that end objective, password policies have been influenced with regards to dimensions such as length and usage of lower-case, upper-case and symbols. However, people often create passwords that are relatively easy to guess, structured around various bits of their personal information. For example, passwords involving their nicknames and ending them with year of birth (e.g., johnny1988). Furthermore, examining password dumps like rockyou.txt and others, we will notice many instances of the same passwords or close variations of being used by multiple strangers. This indicates that there exists a commonality in thought process amongst people that leads to such an outcome.

Randomness after all does not come naturally to humans as a component of our thought process. To that end, password generators are offered as a solution. They purport a higher degree of randomness and look to alleviate the cognitive burden places on users, having to conjure a new password each time.

In this paper we will examine the degree to which passwords generated by certain password generators are easy to guess, and consequently their strength. A good password should be very difficult to guess within a reasonable amount of time and information provided.

It would be possible to guess these passwords if the generators followed some kind of deterministic structure in the generation process. We will attempt to find that structure, should it exist, using a probabilistic model. We will evaluate password strength of each generator by utilizing the metric of number of correct guesses made on an out-of-sample test corpus.

## Bigram Language Model

### N-gram Language Model

A language model learns to predict the probability of a sequence of words or characters. Language models have the ability to model the rules of a language as a probability and are used effectively at a number of NLP related tasks like speech recognition and machine translation.

An N-gram language model predicts the probability of a given N-gram within any sequence of words or characters in the language. An N-gram is a sequence of N tokens which could be characters or words given our implementation case. For our probabilistic modelling, we will use a character-level Bigram Language model.

For example, the password: “abcedfg”.

At a character level, a unigram (or 1-gram) is a one-character sequence. For the above password, the possible unigrams would be:

1. a
2. b
3. c
4. d
5. e
6. f
7. g

Therefore, a bigram (or 2-gram) is a two-word sequence of words, and from the earlier sentence, all the possible bigrams would be as follows:

1. ab
2. bc
3. cd
4. de
5. ef
6. fg

A good N-gram model can predict *P* (e|h) – what is the probability an element, e, occurring given the history of previous elements h – where the history contains n-1 elements. We estimate this probability to construct an N-gram model.

Probability is computed by:

1. Chain rule of probability
2. Simplification assumption

The chain rule of probability:

*P*(e1,...,en) = *P*(e1) . *P*(e2 | e1) . *P*(e3 | e1 e2) . *P*(e4 | e1 e2 e3) ..... *P*(en | e1...en-1)

This allows us to compute the joint probability of a sequence by using the conditional probability of an element given the previous element.

To calculate the sequential conditional probabilities with complex conditions, we make a simplification assumption. This assumption is known as the 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.

A bigram model would assume that:

*P*(ek | e1,...,ek-1) = *P*(ek | ek-1)

By this assumption, we would be looking to predict the current n-gram from the previous n-gram.

Using transition probabilities like P(e2|e1) and P(e3|e2) rather that considering all previous elements in the sequence makes it easier to extrapolate to other arbitrary sequences. As in the example password above, “abcdefg”, using a bigram captures the idea that alphabetical character sequences are likely to appear in order and would assign higher probabilities to sequences like "abc".

A bigram model will therefore consist of the matrix of transition probabilities from some arbitrary token to all other tokens. We use count to find the probabilities.

### Model Implementation

We being with tokenizing out input passwords into individual character elements. We achieve this with the preprocess function.

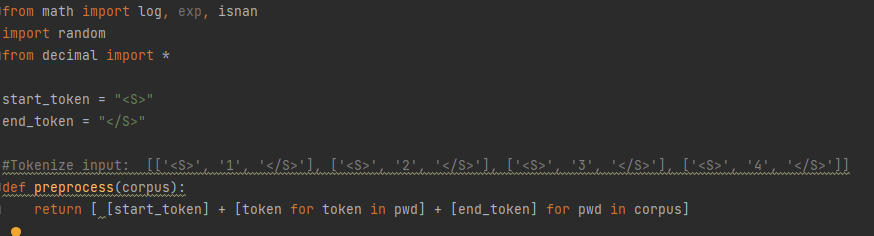


Figure 1: preprocess()

The function will take in a list of passwords, for example ['DPtkUwFt', 'CkYmNTwx'] and output the same list as [['<S>', 'D', 'P', 't', 'k', 'U', 'w', 'F', 't', '</S>'], ['<S>', 'C', 'k', 'Y', 'm', 'N', 'T', 'w', 'x', '</S>']], where the start and end of each password is denoted by the tokens <S> and </S>.



Figure 2: Train()

As above, in the function def Train after preprocessing is complete, we proceed to find all the possible Bigrams, dictionary of Bigrams, and Unigrams along with their corresponding frequencies. To that end, we use the dictionaries self.unigram\_counts and self.bigram\_counts to store the frequencies of each token and the conditional frequency of next\_token occurring given that token has occurred in self.bigram\_counts[token][next\_token].

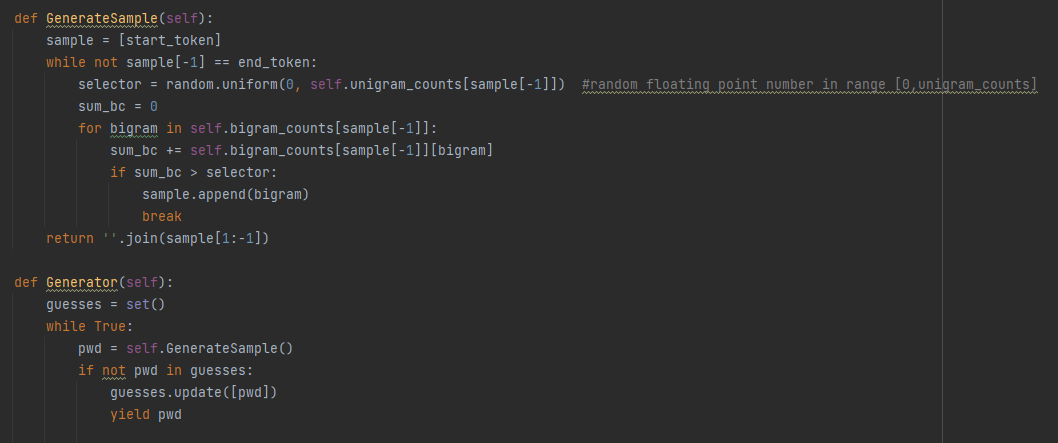


Figure 3: Generating guesses

The GenerateSample function calculates the transition probabilities *P*(ek | ek-1), like *P*(e2|e1). It will begin generating guesses using a selector - a random floating-point number in the range from 0 to the unigram count of the present token, where the present token is not the end\_token.

The Generator function will check if a generated guess is new and unique against the previous set of guesses before yielding it.

The full code can be referenced within bigram.py.

## Data Collection

For each of the password generators we generated password of lengths 8, 12 and 20. For each password length variation, we sampled 10,000 passwords each under different password policy conditions.

The python package Selenium was the cornerstone in our approach to automated data collection. It allows us to automate repeated actions and commands by interacting with the web browser. Here we use it to configure the password policies of each generator accordingly, capture password generated, generate a new password and so forth.

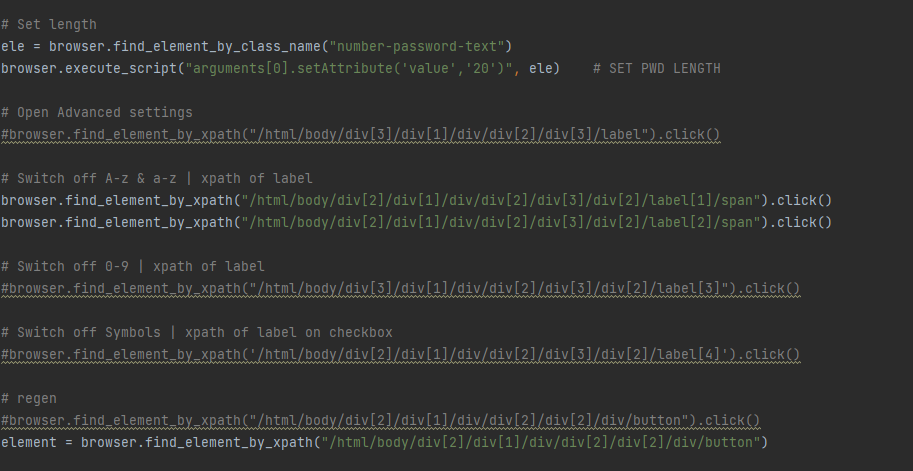


Figure 4: Selenium example

This is achieved by identifying the web page elements useful to us and their relevant tags, be it class, id or xpath. Through python we can then send commands such as clicks or alter attribute values.

As much as possible we look to keep the password generation policies consistent across different generators. With the exception of Zoho, we were able to do with the remaining five. For each of these five generators (Avast, Lastpass, Dashlane, Passwordgenerator.net, Roboform), we ran selenium scripts for each length bin (8,12,20) in batches of five. A single batch of five would comprise of the following variations:

1. Letters, Digits, Symbols
2. Letters, Digits
3. Letters, Symbols
4. Letters
5. Symbols and Digits

For consistency as well as to track practical expectations with the real-world password requirements, we conformed to mixed case for alphabets.



Figure 5: Copy-paste loop

Thereafter, we capture and re-generate the next new password using a for loop. The corresponding python scripts used for each generator is attached for reference.

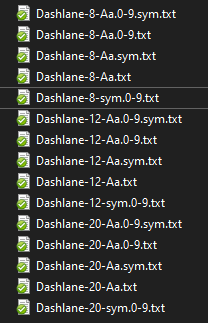
### Dashlane

We used the following controls for the Dashlane password generator for generation:

1. Length (8,12,20)
2. Letters (mixed case)
3. Digits
4. Symbols

|  |  |  |
| --- | --- | --- |
| **Filename** | **Length** | **Variation** |
| Dashlane-8-Aa.txt | 8 | Only letters |
| Dashlane-8-Aa.sym.txt | 8 | Letters and symbols |
| Dashlane-8-Aa.0-9.txt | 8 | Letters and digits |
| Dashlane-8-Aa.0-9.sym.txt | 8 | Letters, digits and symbols |
| Dashlane-8-sym.0-9.txt | 8 | Symbols and digits |

The above table illustrates the nomenclature used for naming our files. This logic would extend to apply to password sets of length 12 and 20 as the screenshot below:



Each file contains 10,000 passwords. This would bring our total passwords in the Dashlane corpus to 150,000 passwords.

### 3.2 Avast

We follow the style of password generation in the earlier case here.

|  |  |  |
| --- | --- | --- |
| **Filename** | **Length** | **Variation** |
| Avast-8-Aa.txt | 8 | Only letters |
| Avast-8-Aa.sym.txt | 8 | Letters and symbols |
| Avast-8-Aa.0-9.txt | 8 | Letters and digits |
| Avast-8-Aa.0-9.sym.txt | 8 | Letters, digits and symbols |
| Avast-8-sym.0-9.txt | 8 | Symbols and digits |

Each file contains 10,000 passwords. This would bring our total passwords in the Avast corpus to 150,000 passwords.

### 3.3 Passwords Generator (passwordsgenerator.net)

|  |  |  |
| --- | --- | --- |
| **Filename** | **Length** | **Variation** |
| PwdGen-8-Aa.txt | 8 | Only letters |
| PwdGen-8-Aa.sym.txt | 8 | Letters and symbols |
| PwdGen-8-Aa.0-9.txt | 8 | Letters and digits |
| PwdGen-8-Aa.0-9.sym.txt | 8 | Letters, digits and symbols |
| PwdGen-8-sym.0-9.txt | 8 | Symbols and digits |

Each file contains 10,000 passwords. This would bring our total passwords in the corpus to 150,000 passwords.

### 3.4 Lastpass

|  |  |  |
| --- | --- | --- |
| **Filename** | **Length** | **Variation** |
| Lastpass-8-Aa.txt | 8 | Only letters |
| Lastpass-8-Aa.sym.txt | 8 | Letters and symbols |
| Lastpass-8-Aa.0-9.txt | 8 | Letters and digits |
| Lastpass-8-Aa.0-9.sym.txt | 8 | Letters, digits and symbols |
| Lastpass-8-sym.0-9.txt | 8 | Symbols and digits |

Each file contains 10,000 passwords. This would bring our total passwords in the Lastpass corpus to 150,000 passwords.

### 3.5 Roboform

|  |  |  |
| --- | --- | --- |
| **Filename** | **Length** | **Variation** |
| Roboform-8-Aa.txt | 8 | Only letters |
| Roboform-8-Aa.sym.txt | 8 | Letters and symbols |
| Roboform-8-Aa.0-9.txt | 8 | Letters and digits |
| Roboform-8-Aa.0-9.sym.txt | 8 | Letters, digits and symbols |
| Roboform-8-sym.0-9.txt | 8 | Symbols and digits |

Each file contains 10,000 passwords. This would bring our total passwords in the Roboform corpus to 150,000 passwords.

### 3.6 Zoho

|  |  |  |
| --- | --- | --- |
| **Filename** | **Length** | **Variation** |
| Zoho-8-Aa.txt | 8 | Only letters |
| Zoho-8-Aa.sym.txt | 8 | Letters and symbols |
| Zoho-8-Aa.0-9.txt | 8 | Letters and digits |
| Zoho-8-Aa.0-9.sym.txt | 8 | Letters, digits and symbols |

For Zoho, we are required to keep alphabets in any generation mix, to which we can either add digits or symbols or both. We are unable to generate variations of symbols and digits or either individually. Additionally, we elected to not use the start with alphabet option. Therefore, for this corpus we only have 120,000 passwords.

For each password generator vendor, we merged the respective individual files into a single larger corpus. This was achieved using script merged.py.

1. Dashlane – merged\_dashlane.txt
2. Avast – merged\_avast.txt
3. Password Generator – merged\_pwdgen.txt
4. Roboform – merged\_roboform.txt
5. Zoho – merged\_zoho.txt

We use the merged corpus in an 80/20 training and test corpus split for our modelling and subsequent testing.

## Password Strength

We will evaluate the password strength of each generator based on how many unique passwords guesses our model makes which exist in the test corpus, for a fixed number of attempts.

Firstly, to validate our model, we will apply it over a common password dump, rockyou.txt[[1]](#footnote-1).

It comprises of 14,344,391 passwords and we will split our training and test corpus, 80% and 20% respectively. The number of guesses the model could make was limited to 100,000 tries and 1,000,000 tries, ceteris paribus. We opted to not exceed a million guesses due to computational time taken.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Corpus** | **Guess attempts** | **Size of Training Corpus** | **Size of Test Corpus** | **Correct Guesses** |
| Rockyou.txt | 100,000 | 5000 | 14,339,391 | 5,535 |
| Rockyou.txt | 1,000,000 | 5000 | 14,339,391 | 34,317 |
| Rockyou.txt | 100,000 | 10,000 | 14,334,391 | 5,584 |
| Rockyou.txt | 1,000,000 | 10,000 | 14,334,391 | 34,770 |
| Rockyou.txt | 100,000 | 11,475,513 | 2,868,878 | 1,089 |
| Rockyou.txt | 1,000,000 | 11,475,513 | 2,868,878 | 7,144 |

The last two rows reflect the proportion split of 80% training and 20% test corpus as we plan to implement with the passwords generated from each password manager. Additionally, to mimic the sizes of samples we collected for each generator, we will constrict the size of training corpus to 5,000 and 10,000 respectively. The results above speak to the efficacy of the model; on the most constrained set of inputs of 5000 sized training corpus and 100,000 tries for guesses it was able to make 5,535 correct guesses – which is more that the training corpus size. We expect this to bode well for smaller sampling sizes.

For each of the password generators, we will merge all the individual files into a combined larger corpus and randomly shuffle the elements before conducting the split between training and test corpus. This approach should allow the model to pick up deterministic characteristics if they are significantly present.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Corpus** | **Guess attempts** | **Size of Training Corpus** | **Size of Test Corpus** | **Correct Guesses** |
| merged\_dash.txt | 100,000 | 120,000 | 30,000 | 0 |
| merged\_dash.txt | 1,000,000 | 120,000 | 30,000 | 0 |
| merged\_avast.txt | 100,000 | 120,000 | 30,000 | 0 |
| merged\_avast.txt | 1,000,000 | 120,000 | 30,000 | 0 |
| merged\_pwdgen.txt | 100,000 | 120,000 | 30,000 | 0 |
| merged\_pwdgen.txt | 1,000,000 | 120,000 | 30,000 | 0 |
| merged\_lastpass.txt | 100,000 | 120,000 | 30,000 | 0 |
| merged\_lastpass.txt | 1,000,000 | 120,000 | 30,000 | 0 |
| merged\_zoho.txt | 100,000 | 96,000 | 24,000 | 0 |
| merged\_zoho.txt | 1,000,000 | 96,000 | 24,000 | 0 |
| merged\_robo.txt | 100,000 | 120,000 | 30,000 | 0 |
| merged\_robo.txt | 1,000,000 | 120,000 | 30,000 | 0 |

The model was unable to make any correct guesses in any of the password generator corpus. Given these password managers use pseudo-random algorithms to generate their passwords, this is not that surprising.

To offer a sense of this model’s efficacy, let us consider an eight-character password formulated from a universe of 26 upper-case letters, 26 lower-case letters, 10 digits and 18 symbols. This constitutes a total of 80 possible characters to choose from. For a random password, there are 80 possibilities for each character. Which means an eight-character password has 808 possibilities (1,677,721,600,000,000), over a quadrillion.

To combat the possibility that our test corpus might be a constrain due to its size, we repeated the experiment utilising a 20/80 split, in the way of training and test corpus respectively.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Corpus** | **Guess attempts** | **Size of Training Corpus** | **Size of Test Corpus** | **Correct Guesses** |
| merged\_dash.txt | 100,000 | 30,000 | 120,000 | 0 |
| merged\_dash.txt | 1,000,000 | 30,000 | 120,000 | 0 |
| merged\_avast.txt | 100,000 | 30,000 | 120,000 | 0 |
| merged\_avast.txt | 1,000,000 | 30,000 | 120,000 | 0 |
| merged\_pwdgen.txt | 100,000 | 30,000 | 120,000 | 0 |
| merged\_pwdgen.txt | 1,000,000 | 30,000 | 120,000 | 0 |
| merged\_lastpass.txt | 100,000 | 30,000 | 120,000 | 0 |
| merged\_lastpass.txt | 1,000,000 | 30,000 | 120,000 | 0 |
| merged\_zoho.txt | 100,000 | 24,000 | 96,000 | 0 |
| merged\_zoho.txt | 1,000,000 | 24,000 | 96,000 | 0 |
| merged\_robo.txt | 100,000 | 30,000 | 120,000 | 0 |
| merged\_robo.txt | 1,000,000 | 30,000 | 120,000 | 0 |

## Limitations and Extensibility

An immediate and obvious limitation is the sample size for each type of password generator. Perhaps with a larger sample size and a larger number of allowed guesses, some correct guesses of the test corpus could have been made. We should consider the nature of randomness. While the model’s generated guesses did not occur within our test corpus, it could very be possible that at some point they very well might have as per the infinite monkey theorem.

This leads us to next limitation, which is one of computational power. With more powerful hardware training could be done at a much quicker pace over larger training corpus. Additionally, with a dedicated graphics card we could even consider deep learning approaches like neural networks which possibly have a higher efficacy that statistical models like the one we employed.

A problem with n-gram language models is that if we increase the n in n-grams it becomes computation-intensive. If we decrease the n, then long-term dependencies are not taken into consideration. Perhaps a trigram (3-gram) model would have yielded better results.

## Conclusion

On the subject of evaluating password strength, we have outlined one of many methods for doing so – couched in the contrapositive of identifying weak passwords. We should note that instead of coming up with our own passwords, which usually will result in some weakness or subconscious deterministic behaviour, we should opt to use password generators instead. This should be in the least, the base requirement. Where possible we should also actively utilise two-factor or multi-factor authentication mechanisms. As the defence-in-depth principle goes, the more bulwarks we have, the less vulnerable we tend to be.

1. See rockyou.txt in attached files. [↑](#footnote-ref-1)