CS4028 CA

Hariss Ali Gills

October 27, 2024

Contents

Chapter 1

Tasks 1 - 3

1.1 Brute-force Cracking

For this task, the most challenging part was definitely generating the set of shortlex strings. After remembering the definition from Modeling and Problem Solving for Computing course, the shortlex order is simply a total ordering of the lexicographical order at a certain length [10]. To compute the lexicographical order of a set at a certain length, find the order on the Cartesian product of the ordered set - in this case ASCII lowercase and digits [5]. While writing this report, I came across the use of generators in Python to save some memory when returning so many strings [2].

As for the algorithm itself, the output list was first initialized with None values. This was done to ensure that the indices match up i.e The input list hash would have the same index as the output plain text password. The function keeps running until it finds all of the passwords hence the stop condition is when the output list has no `None` values. Instead of iterating through the list of hashes which would generate the same strings a lot of times, the loop uses the list of shortlex strings. Currently, a limitation is when the input list contains non-unique hashes.

1.2 Dictionary Cracking

The second task felt relatively straightforward. The first step was to hash all of the values in the dictionary text file. A dictionary was used with the keys are the hashed passwords since it made the cracking algorithm more readable. With a list the indices would have to match the line and in turn the plain text password which would be too many read operations. Next, we check if the password was already hashed. if so, we skip hashing it. This check actually makes the speed slower, but as per the assessments' assumption "you should assume that computing hashes is more expensive than searching lists." Hence this check was kept. An improvement for speed could be to hash the file in chunks instead of the whole thing at once.

The function dictionary_cracking uses the same indices trick as before to match the input list. This time the hash list is looped, and is enumerated to get the index. Also, in this algorithm, it's possible that the returned list may have some None values.

1.3 Dictionary Cracking with Salts

By this third task, it was annoying to format the hash strings constantly, so the hashes were read form a hashes.json file. Python annoyingly has no built in way to store tuples from json so Python encodes tuples to json lists because that is the closest thing in JSON to a tuple [9]. So, each item was converted to a tuple before finally calling dictionary_cracking_with_salt. Another change was to the hash_file function to accept salt strings. By default, an empty string is passed.

The function almost works identically to dictionary_cracking. It was tempting to merge the functions together to handle both cases, but as the assessment document implies that a different input be provided, so it was decided against doing that. The main difference is that we recalculate the hash dictionary every time a new hash value is iterated over. An improvement is to check if this salt was already used and to reuse the dictionary.

Chapter 2

Task 4 - Cracking Using N-gram Markov Chains

2.1 Goals

With passwords being the most used form of authentication method [7], NLP techniques are employed into popular cracking tools. Tools like John the Ripper offers users to train bi-gram Markov chains while OMEN offers a variable n-gram to train the chains [4]. The goal was to get similar percentage of passwords cracked compared to this paper [6] using the RockYou dataset [3].

2.2 Methodology

2.2.1 Cross Validation

Just like in the "Analysis of Password datasets" [6], 5-fold cross validation was deployed but only on the RockYou dataset. To achieve this, the `split_into_folds` function was implemented.

In Cross Validation, a dataset is split into a learning segments and training segments. Ideally, each data point should be used so k-fold Cross Validation is commonly used. In this process, the data is randomly divided into k folds of roughly similar size. A distinct fold of the data is then kept out for validation during each of the k training and validation iterations [8].

2.2.2 The Dataset

A data breach that occurred at RockYou in December, 2009 exposed more than 32 million user accounts. This happened as a result of failing to patch a ten-year-old SQL vulnerability and keeping user data in an unencrypted database, including user passwords in plain text rather than using a cryptographic hash. The extent of the breach was miscommunicated by RockYou, and users were not notified of it [1].

An issue with the RockYou dataset, was that the text was encoded in UTF-8 but had some latin-1 characters. To resolve this, the `.txt` file was re-encoded to UTF-8. Another common issue with password datasets is that the lists are ordered by frequency, but none of the frequency values are provided which could improve the accuracy of the probability counts for Markov chains.

2.2.3 Training the Chain

Unlike the work by Potasznik et al. [6], which computes probabilities similar to OMEN. Here, n-gram lengths go from 1 to n-1. In this case, we generate a up to 3-gram Markov chain per each test fold. This process is described below:

1. `chain_folds`: Take in a list of folds and count the occurrences of a distinct n-gram per password a with right delimiter ("€" was chosen arbitrarily)
2. `find_token_occurence`: Use a special data structure to make counting simpler by avoiding `KeyErrors` with pythons `defaultdict`. The Markov chain is a dictionary of dictionary of floats in which the first key in the previous tokens, second key is the next token, the float value is the probability of that next token.
3. `find_ngram`: To find the n-grams, text processing per password is done. A left delimiter ("£" was chosen arbitrarily) is added in case the index is too small. A list of tuples which contain the previous token and the next token is returned.
4. `calculate_probabilities`: Finally, iterate through each previous token, sum up all of counts, and assign each next token a probability by using its count divided by the sum.

Optionally, you can pickle the generated Markov chains to use later. For a detailed listing refer to `markov/train_chain.py`

2.2.4 Cracking Using the Generated Passwords

Cracking is a simpler process. We track the success rates and elapsed times per fold. Like the paper, a million passwords with a maximum length of a hundred are generated, but we only count the uniquely generated passwords. This process is described below:

1. `cracking_with_markov_chains`: To crack a list of hashes, a count of unique passwords is kept. If a password generated by the Markov chain is found, it is added to the plain text list with the same index as the hashed list to maintain order.
2. `generate_password`: The password is initialized with left delimiters until the max password length or the right delimiter is generated.
3. `gen_character`: Each character is found by using Python's `random.choices()` with the keys as the n-grams and the probabilities as weights. In the worst case, we try to shorten the n-gram and try recursively.

The plots are only generated to discuss the results compared to the paper. The success rates and elapsed times per fold are plotted. A plot of the mean and standard deviation of the success rates per fold is also available. The `seaborn`, `matplotlib`, `numpy` libraries need to be installed to graph the plots.

2.3 Results

The folds were run on a charging laptop the following specs:

- CPU: AMD Ryzen 5 7640U w/ Radeon 760M Graphics (12) @ 4.971GHz
- Memory: 32 GB DDR4
- OS: Arch Linux version 6.11.4-arch2-1

As the charts suggest, generating a million unique passwords takes about mean values of 66.95 seconds with a 27.61% success rate.

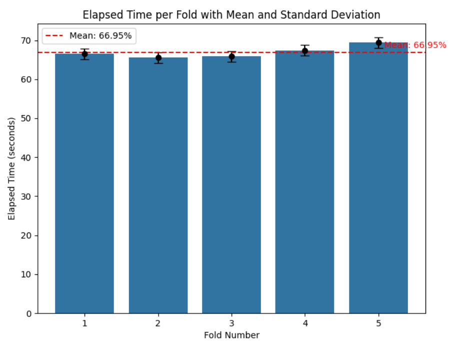

Figure 2.1: Elapsed Ti per Fold

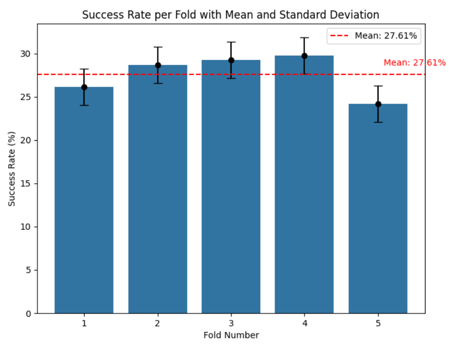

Figure 2.2: Success Rate per Fold

2.4 Conclusion

This is a significant improvement to the percentages shown in the paper [6]. Although there might be be a slight improvement due to the n-gram lengths, this is largely due to the fact that one large dataset, using different dictionaries, was used in the paper. This is because it seems like password datasets seem to have a vast difference in password structure across dictionaries.

Unfortunately, due to time constraints and the other datasets being quite challenging to find, only the RockYou dataset was used. A better designed experiment could be conducted by using the large dataset.

2.5 Lessons Learnt

This was a very fascinating mini-project that was inspired by the CS4051 NLP course's word sequences lecture. I found it quite challenging to find good datasets and often missed the frequency counts which could have improved the results. Thus, I focused on getting one dataset to run well. I did manage to find hashmob which had interesting wordlists. Moreover, I also wanted to use `argparse`, so that each task could have been run independently.

Appendix A

Appendix

The below section is from the `README.md`

A.1 CS4048 CA - Cracking using N-gram Markov Chains

This project implements a password-cracking algorithm based on Markov Chains. The goal is to generate password guesses that closely resemble real-world password patterns by analyzing a training set of known passwords. Using Markov Chains, the model learns the statistical likelihood of character sequences in passwords, enabling it to generate highly probable password candidates that can be used for password cracking.

A.1.1 Installation

The project has been tested on python version 3.12.7. Hence, it is suggest to use that version.

External packages **are** required to plot.

To plot, use the package manager pip to install the dependecies.

```
pip install -r requirements.txt
```

A.1.2 Usage

By default, all of the tasks and imports related to plotting are commented out. You can comment out the distinct tasks within main.py to run specific tasks.

```
python main.py
```

Bibliography

[1] Nik Cubrilovic. RockYou Hack: From Bad To Worse — TechCrunch — techcrunch.com. https://techcrunch.com/2009/12/14/rockyou-hack-security-myspace-facebook-passwords/. [Accessed 24-10-2024].

[2] Python Docs. Generator Objects — docs.python.org. https://docs.python.org/3/c-api/gen.html. [Accessed 19-10-2024].

[3] Brannon Dorsey. Rock You. https://github.com/brannondorsey/naive-hashcat/releases/download/data/rockyou.txt. [Accessed 25-10-2024].

[4] Markus Duermuth, Fabian Angelstorf, Claude Castelluccia, Daniele Perito, and Abdelberi Chaabane. OMEN: Faster Password Guessing Using an Ordered Markov Enumerator. In *International Symposium on Engineering Secure Software and Systems*, milan, Italy, March 2015.

[5] math24. Lexicographic Orders — math24.net. https://math24.net/lexicographic-orders.html. [Accessed 19-10-2024].

[6] Richard Johnson Max Potasznik, Abhinav Narain. cs.umd.edu. https://www.cs.umd.edu/~rbjohns8/projects/pwd.pdf. [Accessed 24-10-2024].

[7] Maria Papathanasaki, Leandros Maglaras, and Nick Ayres. Modern authentication methods: A comprehensive survey. *AI, Computer Science and Robotics Technology*, Jun 2022.

[8] Payam Refaeilzadeh, Lei Tang, and Huan Liu. *Cross-Validation*, pages 532–538. Springer US, Boston, MA, 2009.

[9] wikipedia. JSON - Wikipedia — en.wikipedia.org. https://en.wikipedia.org/wiki/JSON#Data_types.2C_syntax_and_example. [Accessed 19-10-2024].

[10] wikipedia. Shortlex order - Wikipedia — en.wikipedia.org. https://en.wikipedia.org/wiki/Shortlex_order. [Accessed 19-10-2024].