Determining Password Strength Using Machine Learning

Stephen Hamilton
Virginia Polytechnic Institute and State University
925 Prices Fork Rd
schamilton@vt.edu

1. Background

Passwords are the latchkey to our online lives. If someone got a hold of our passwords, we would be scrambling to change the passwords to prevent that person from accessing our data. An attacker could do anything with your passwords - impersonate you, steal and sell your identity, read your emails, perform administrative actions, or even access sensitive information. Needless to say, passwords are critical. However, most sign up forms show only a simple password strength checker. This checker uses a simple scoring system, adding to the score the longer the password is. It also adds a higher weight score if special characters or numbers are added. The problem with this approach is that a password as simple as "Password123!@#" would be shown as "strong" to the user. Despite this password being exceptionally weak to a common dictionary attack. [4] Password crackers using dictionaries would quickly try "Password" as a password, and then append a sequence of common numbers and special characters. These crackers intend to catch behavior such as this, where a common password is used, but "spiced up" a little by adding numbers and special characters.

Passwords are stored as hashes. This is a one-way encryption that verifies that the user has entered their password. When logging in, the user's entered password is hashed with the same algorithm used to hash the password during sign up. If the two hashes match, the user entered their password, and are authenticated. However, if a collection of password hashes are uncovered, rogue actors could attempt to brute force passwords that match in a sea of hashes. If a hash matches, they store the password that generated that hash, effectively stealing your password. Sony's recent data breach in 2023 resulted in thousands of passwords being exposed [2], some belonging to more than just Sony employees and customers. However, having a strong password makes it either extremely difficult or impossible for password crackers to get that hash. Having a password that is difficult to guess by a machine is a secure password, and that is what this project aims to assist with.

2. Introduction

The goal of this project is to learn about how to develop machine learning models, while developing a useful application in the process. This project focuses on determining the strength of a given password by classifying it in one of 5 categories: very weak, weak, average, strong, very strong. If a user entered "1234" or "Password", it should show as "very weak". However, if a user entered something like "correct h0rs3-BATTERY-st@pl#", it would be classified as "strong". This would be more advanced than the basic password strength indicators in many account creation forms.

While this would be overkill for a basic indicator used in an account creation dialog, this could be a useful standalone tool for users validating their security. If a user needs to have high security standards, they need very strong passwords. Having a tool such as this in their toolbox would be helpful in determining whether a password is secure or not.

3. Approach

Trying to quantify a password is a challenge. Usually, machine learning models expect a fixed number of inputs. However, due to the nature of passwords, they could be any length, so trying to determine strength is difficult.

Initially, an attempt was made to use text vectorization to classify the password. Unfortunately, this requires a common character to be used to split the password into multiple segments. This is usually used on English text, so a space would be used. Passwords do not have a standard character that could be used to separate it into multiple segments, so standard text vectorization would not work.

Another attempt was made to try encoding passwords into their UTF-8 equivalent, visualized in Figure 1. The password string was converted into a byte array of UTF-8 characters. A maximum length was set. If the array was larger than the max length, it would simply be truncated. If the array is shorter, the remaining space is filled with NULL bytes.

However, this approach also had problems. Instead of the model being trained on characters, it was being trained

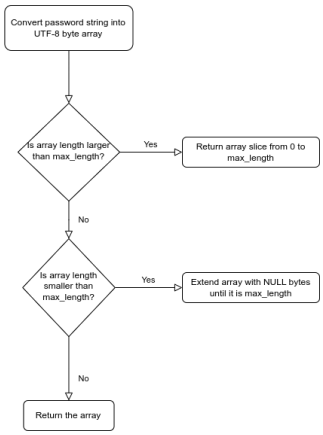

Figure 1. Password padding technique

```
Prediction:
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 140ms/step
[[1.20288588e-01 1.57832201e-05 7.55284667e-01 1.56907863e-05
  1.24395207e-01]
 [1.20288588e-01 1.57832201e-05 7.55284667e-01 1.56908009e-05
  1.24395214e-01]
 [1.20288588e-01 1.57832201e-05 7.55284667e-01 1.56907863e-05
  1.24395207e-01]]
```

Figure 2. Attempt to use NULL terminators as padding

on NULL characters. No matter what password was put in, the percentage chance of it being an ”average” password remained unchanged. Trying to change the maximum length resulted in minimal changes to the results.

The result of this flawed approach is shown in Figure 2. The passwords fed into the model were ”1234”, ”Password”, and ”Th!$ i$ my SUPER ultr@ m3g5 CR4ZY P5SSW0R8 64”. The results shown is a two-dimensional array. The array has three elements, one for each password. Each nested array has 5 elements, one for the probability of the password belonging to that category. You can see that the percentage chance is very slightly different between the three passwords. However, the percentage chance is very distinct across the 5 categories. Index 2 always has the highest chance, which belongs to the ”average” category.

```
[[3.8254493e-01 3.5657367e-01 2.3165177e-01 1.2264378e-09 2.9229665e-02]
 [3.8254493e-01 3.5657367e-01 2.3165177e-01 1.2264378e-09 2.9229665e-02]
 [3.8254493e-01 3.5657367e-01 2.3165177e-01 1.2264378e-09 2.9229665e-02]]
[0, 0, 0]
```

Figure 3. Padding technique results, with 5 epochs and Adam optimization

```
[[1.1704436e-01 1.9519146e-06 7.5265145e-01 1.5921278e-06 1.3030070e-01]
 [1.1704436e-01 1.9519146e-06 7.5265145e-01 1.5921278e-06 1.3030070e-01]
 [1.1704436e-01 1.9519146e-06 7.5265145e-01 1.5921278e-06 1.3030067e-01]]
```

Figure 4. Padding technique results, with 1 epoch and Adam optimization

4. Experiments and Results

The adequacy of the model was measured by first looking at the accuracy metric produced by Keras after training. If accuracy was near or below fifty percent, the run was considered bad and immediately discarded. A tweak to the model would then be made and another run made to see how prediction accuracy reacts. Tweaks could be anywhere from restructuring the model to adjusting parameters such as maximum password length, number of epochs, and size of the dataset. Another factor is how much training data is in use. If there is not a lot of training data, more needed to be used, otherwise the model would not account for different types of passwords.

Initially, text vectorization was used, but proved to be nonfunctional, as it expected the input to be fixed. This likely could have been fixed by padding with some dummy character, but was not applied before trying the more successful padding technique.

The sparse categorical entropy loss function was used in every experiment as it is great for organizing data into multiple categories. [5]

As shown by Figure 3, using the padding technique with additional epochs resulted in the ”very weak” category always being selected. This was tested with the three passwords ”1234”, ”Password”, and ”Th!$ i$ my SUPER ultr@ m3g5 CR4ZY P5SSW0R8 64”. As evidenced by the percentage output, this was not an acceptable model, as the first two passwords are expected to be ”very weak”, but the third is expected to at least be ”strong”.

Figure 4 is still using the padding technique but only one epoch and a smaller dataset. This was an initial test to see how padding would work, but it resulted in the ”average” category being selected for the three passwords shown above. Multiple tests were done after this to attempt to resolve this issue of grouping all predictions into one category. These tests tried increasing epoch, changing the maximum password length, increasing the dataset size, and using different optimization strategies.

Increasing epoch only resulted in all the passwords being grouped into a different category. Changing the maximum password length only changed the percentages by a small fraction. Increasing the dataset size had the same ef-

PSCD	PWLDS
0 (weak)	0 (very_weak)
1 (medium)	2 (average)
2 (strong)	4 (very_strong)

Table 1. Mapping PSCD categories to PWLDS categories

fect as increasing epoch size - passwords were grouped into a different category. Using different optimization strategies seemed to have little to no effect. When only using the PSCD dataset, accuracy was around seventy to eighty percent. However, combining both PSCD and PWLDS datasets resulted in accuracy dipping to as low as forty percent.

5. Datasets

The datasets used are the Password Strength Classifier Dataset (PSCD) [1] and the Password Weakness and Level Dataset (PWLDS) [3].

5.1. Password Strength Classifier Dataset

Initially, the PSCD dataset was used for testing. This dataset needed some pre-processing before it could be properly used. There were several bad lines that seemed to have some extra data associated that were not just password and strength. A Python script was written and placed in the same directory as the dataset. This script ran and created another dataset with the same data, just with the bad lines cleaned up. Also, this script unified the headers to have the same names as the headers from the PWLDS dataset. Furthermore, the script also converted these categories to match the categories coming from PWLDS.

The PSCD categories were 0 (weak), 1 (medium), and 2 (strong). However, the PWLDS categories were 0 (very_weak), 1 (weak), 2 (average), 3 (strong), and 4 (very_strong). The PWLDS categories were decided on as they provide more feedback to the end-user about the strength of their password. Having five categories rather than three would make it easier for the user to see how bad or how good their password strength is.

The PSCD categories were mapped to PWLDS categories using Table 1:

These passwords were collected by using passwords cracked during the 000webhost leak. This contains approximately six-hundred seventy thousand passwords for analysis. Most of them are classified as ”medium”, but adding the PWLDS dataset will help balance this.

The PSCD dataset was inserted into the repository by downloading it and uploading it with Git Large File Storage.

5.2. Password Weakness and Level Dataset

PWLDS contains over four million passwords, designed to help researchers develop password strength tools. The strength levels are indicated with their names in Table 2 This dataset does not use real-world data. Rather, it uses generated passwords, using varying settings for each category of strength. Weaker passwords are shorter and less complicated, while stronger passwords are longer and contain differing symbols, numbers, and other unicode characters. However, most of these unicode characters are skipped when training the algorithm, as this reduces the complexity when storing bytes into an array.

Category Index	Category Name
0	very_weak
1	weak
2	average
3	strong
4	very_strong

Table 2. PWLDS category indices to human-readable string

The PWLDS dataset was inserted into the repository by adding it as a Git Submodule, since the dataset is hosted on GitHub.

6. Availability

The source code, along with a copy of this document, can be found on the following GitHub page: https://github.com/Stephen-Hamilton-C/mlpass

7. Reproducibility

7.1. Setup

Clone the code from the GitHub repository and make sure to follow the README document included to get started with the code.

Remember to create a Python virtual environment, by running the following command:

```
$ python3 -m venv venv
```

Then, you can enter this virtual environment by running this command on Windows:

```
$ .\venv\Scripts\activate.bat
```

If you are running macOS or Linux, run this command instead:

```
$ source venv/bin/activate
```

Install all required packages to run this project:

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

You can exit the virtual environment by running the 'deactivate' command.

Index	Category
0	very_weak
1	weak
2	average
3	strong
4	very_strong

Table 3. Category index mapped to a human-readable category

7.2. Code

The main.py file in the mlpass directory is not used. This file was originally intended to be the frontend for this application, but time ran out on experimentation before it could be completed.

All of the relevant code is found in mlpass/ml.py. This is where the model parameters can be changed and tweaked. To test the results, run the ml script with the following command while in the root directory of the repository:

```
$ python3 mlpass/ml.py
```

The max_length variable can be changed in the ml.py script to change the maximum password length. The train method on machine takes in a single integer. This integer is the number of epochs to use when training. When the model is done training, the code will save the model to a file with the save method. Change the load_path parameter to change the name of the model when it is saved. If a model with that name already exists, you can put it into the load_path parameter and skip the training and saving step. If you try to train the model after it has already been loaded from a file, it will output a warning and then skip training.

The datasets are combined into a single Pandas DataFrame, shuffled, and then split in half. The first half is the training data, and the second half is the validation data. More data could be loaded in by changing the load_data function defined at the top of the file.

Predicting results can be achieved by running the predict method on machine. It takes in a list of passwords to evaluate and returns a list of category indices. Use Table 3 to decipher these indices.

8. Conclusion

This project was overall unsuccessful. With more time, it may have been possible to explore different model structures. Trying different layers, different techniques, and adjusting hyperparameters would all have varying levels of impact on the machine learning model.

This is certainly possible, as large corporations such as Twitter have machine learning algorithms that classify passwords. However, in its current state, the model simply struggles with classifying passwords in more than one category. Perhaps a smarter split of the data is necessary - ensuring there are equal counts of passwords in each category would help the algorithm make proper evaluations, rather than have it biased toward a single category.

Using string vectorization is tricky, as it expects a common character to be used to split the password into multiple segments. There are not any good characters to use that are commonly used in all passwords. Perhaps something as simple as the letter 'a', but there is no certainty that it will be in all passwords. Using fixed-length inputs is also likely not a good solution, as this can result in the model training off of NULL bytes, rather than off of the actual content of the password. This is what likely resulted in shorter passwords getting scored higher than they deserved.

The results were expected to return very_weak for the first two passwords, as they were intentionally made to be so. The third password was intended to at least be categorized as strong, but was instead categorized in the same place as the weak passwords. This was far too low of a categorization for the strong password, and too high of a categorization for the weak passwords. Adjusting hyperparemeters often only changed which category the passwords were all put into.

The two datasets have lots of passwords to work with - one dataset has more than the other. However, combining the two tanked accuracy to around forty percent, which hints at something being wrong with the structure of the model.

Overall, this was an excellent learning experience and could be improved upon, given more time. The source code is available on GitHub for tinkering and fixing, so perhaps this could be remedied in due time. However, as the model stands in its current state, it simply cannot correctly categorize passwords.

References

[1] Bhavik Bansal. Password strength classifier dataset, 2019. https://www.kaggle.com/datasets/bhavikbb/password-strength-classifier-dataset/data.

[2] Matthew Zeitlin Charlie Warzel. It gets worse: The newest sony data breach exposes thousands of passwords. *IGN*, 2023.

[3] infinitode. Password weakness and level dataset, 2024. https://github.com/infinitode/pwlds/tree/1d9d9fd713cf640e18e5af3b7fd5a2ef782b22cc.

[4] Kaspersky. What is a dictionary attack? 2024.

[5] TensorFlow. tf.keras.losses.sparsecategoricalcrossentropy, 2024. https://www.tensorflow.org/api_docs/python/tf/keras/losses/SparseCategoricalCrossentropy.