AI Assignment #8: Password Strength Classifier

Harrison Downs | B01256060 | CSCI 6385

1: Overview

In the domain of cybersecurity, one of the most common vulnerabilities of an organization is weak or default passwords. Attackers will often employ spraying attacks, in which they rapidly attempt to log in to multiple accounts using the most common passwords. Furthermore, online resources such as the rockyou dataset that contains millions of leaked passwords, or www.routerpasswords.com that stores default credentials for network devices, can be used to ascertain the most likely password in various scenarios.

Organizations can reduce their attack surface by implementing good password policies. Many such policies include a required password length, use of symbols and numbers, or even a mandatory password change over set intervals. However, how can we be sure that we are configuring password parameters in the right way? Is password strength something that can be optimized for?

As an individual with a focus in cybersecurity and a growing interest in AI, I decided to implement a machine learning agent to dynamically classify passwords' strength. I used a large password classification dataset for training and testing purposes, while I used the scikit-learn Python module to create a logistic regression agent that predicts whether a password is weak, medium, or strong. Using this tool, I aim to ascertain the features that make a password truly difficult to break.

2: Methodology

2.1: Dataset

As the saying goes, an AI agent is only as good as its training data. Because my focus was on creating a password classifier, I needed a large dataset of passwords and their respective strength classifications. Ultimately, I chose to use Bhavik Bansal's Password Strength Classifier Dataset, and in order to discuss this, I first need to touch on the 2015 000webhost breach and PARS.

In 2015, 000webhost suffered a massive data breach that affected approximately 13.5 million users; the leaked data included usernames, email addresses, and passwords. Quickly after the attack occurred, this data circulated around the internet, and ultimately became

accessible to the public. These days, it's included in many security tools such as SecLists as real-life examples of login credentials.

Bansal used the 000webhost data to craft his dataset, and used PARS (Password Analysis and Research System) to evaluate each password's strength on a scale of 0-2, with 0 being weak, 1 being medium, and 2 being strong. PARS is an open-source, modular, and scalable research platform from Georgia Tech that is used extensively by the password research community. In other words, Bansal's use of PARS in his dataset ensures that each password's strength was determined through valid, peer-reviewed means. Additionally, Bansal pruned off passwords with an ambiguous strength level, decreasing the total amount of passwords from 13.5 million to 0.7 million.

This dataset provided exactly what I needed: real-life examples of passwords and their respective strengths given by a reliable source.

2.2: Feature selection

When creating a password, modern password policies often enforce some sort of static rule, mandating a total length and the inclusion of numbers and/or special characters. With this in mind I included character types as their own features of a given password, as can be shown in Figure 2.2.1.

Fig. 2.2.1: List of features relating to character types

```
# Length and character counts
length = len(password)
uppercase = sum(1 for char in password if char.isupper())
lowercase = sum(1 for char in password if char.islower())
digits = sum(1 for char in password if char.isdigit())
special = sum(1 for char in password if not char.isalnum())
```

Additionally, my agent takes into account a password's Shannon entropy (a measure of its randomness), uniqueness ratio, common patterns, and whether or not it's on a list of the most common passwords. For the remainder of this section, I will go into detail about these three features and how they are implemented in my program.

Shannon entropy, developed by Claude Shannon, is essentially a formula that measures the predictability or randomness of a given sequence of characters. High password randomness corresponds to positive entropy values, while low password randomness (or high predictability) leads to entropy values approaching zero. The Shannon entropy *H* of a password is given by:

$$H = -\sum_{x \in chars} p(x) log_2 p(x)$$

Where *x* denotes each unique character in the password, *p*(*x*) is the probability of *x* appearing in the password, and $\log_2 p(x)$ calculates the overall contribution of *x* to the password's entropy. Below is an example comparing the entropy of two passwords: pw1 = "555555", and pw2 = "2a5e320c9" (Fig. 2.2.2). Thus, I implemented Shannon entropy in my list of features (Fig. 2.2.3).

Figure 2.2.2: Comparison of entropy between passwords pw1 and pw2

pw1: "555555"	**pw2: "2a5e320c9"**
Character distribution: ('5', 6}	Character distribution: {'2': 2, 'a': 1, '5': 1, 'e': 1, '3': 1, '0': 1, 'c': 1, '9': 1}
Probability: $p('5') = 6/6 = 1.0$	Probability: $p('2') = 2/9 = 0.222$ $p('a') = p('5') = \ldots = 1/9 = 0.111$
Entropy: $H = -(1.0 \cdot log_2 1.0) = 0$	Entropy: $H = -(0.222 \cdot log_2 0.222) - 7(0.1 \cdot log_2 0.1)$ $H = -(-0.482) - 7(-0.352) \approx 2.946$
Fully predictable; no randomness at all.	***Harder to predict; higher randomness***

Figure 2.2.3: Implementation of Shannon entropy formula in code

```
# Shannon entropy
char_counts = Counter(password)
total_chars = sum(char_counts.values())
entropy = -sum((count / total_chars) * math.log2(count / total_chars) for count in char_counts.values())
```

This code directly represents the Shannon entropy formula, where:

- ***`char_counts`** uses the **`Counter`** tool from the **`collections`** module to count the occurrence of each unique character in a password and store that data in a dictionary-like object.*
- ***`total_chars`** holds the total length of the password itself.*
- ***`(count / total_chars)`** maps to **p(x)**, calculating the probability of each unique character appearing in the password.*
- ***`math.log2(count / total_chars)`** maps to **$\log_2 p(x)$**, calculating each unique character's contribution to the password's entropy.*
- ***`-sum`**… refers to Σ, adding up the values for all characters in the password, thus leading to the password's overall entropy value **H**.*

Another feature my model accounts for is the presence of common patterns. Specifically, it searches for repeated characters (e.g. "aaa"), common sequences ("abc", "123", "password", "qwerty", "asdf"), and repeated substrings (e.g. "ababab"). I used the **`re`** module to allow for each password to be searched in this manner, and if any of the common patterns are apparent, the model flags that password as containing a common pattern. Finally, the last feature my model checks for is if a given password appears on the most recent list of the most commonly used 100,000 passwords. In total, there are nine features being measured for each password.

2.3: Data preprocessing

Before my logistic regression agent could commence the training process, there was still some preprocessing to be done. With defined features, I used the **`pandas`** module to create a **`features`** dataframe, in which each row is a password from Bansal's dataset, and each column contains a feature. **`labels`** then stored the security classification accompanying each password (Fig. 2.3.1a).

With features and security labels parsed out, I then split the data into training and testing sets, 80% to 20% respectively. Using the **`train_test_split`** module from **`sklearn.model_selection`**, I was able to successfully implement the splitting process, training my model with 80% of the data, while reserving 20% to evaluate its accuracy (Fig. 2.3.1b).

Finally, I standardized the data using the **`StandardScaler`** module from **`sklearn.preprocessing`**. In essence, these lines of code assist the agent in converging to a solution at a faster rate and mitigate feature dominance (Fig. 2.3.1c).

Figure 2.3.1: Data preprocessing code

a) *Data parsing through feature extraction & strength labelling*

```
# Apply feature extraction
features = pd.DataFrame([extract_features(pw) for pw in data['Password']])
labels = data['Strength']
```

b) *Splitting data into training and testing sets*

```
# Split data
X_train, X_test, y_train, y_test = train_test_split(features, labels, test_size=0.2, random_state=7)
```

*Note: **`random_state`** is an arbitrary number used to seed the RNG that randomly assigns data to different groups. Setting it to some constant ensures that the split happens the same way with each execution of the program.*

c) *Standardizing features for higher performance and lower feature domination*

```
# Standardize features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

2.4: Training, evaluation, and completion

Implementing the logistic regression agent itself was relatively straightforward. Using the **`LogisticRegression`** module from **`scilearn.linear_model`**, I simply defined a **`LogisticRegression`** object as **`agent`**, and then had **`agent`** train itself with **`X_train`** and **`y_train`**, which represent the respective passwords and strength labels held in the training set (80% of total data). Then, using the **`accuracy_score`** module from **`sklearn.metrics`**, **`agent`** evaluated itself using **`X_test`** and **`y_test`** (representing the testing data, 20% of the original dataset size) (Fig. 2.4.1a), and reported its overall accuracy as a percentage. With both the data splitting and agent being seeded with a constant, **`agent`**'s accuracy consistently came out to 99.94% (Fig. 2.4.1b).

With **`agent`** fully trained, I implemented various functions including checking the strength of passwords input by the user and reporting feature weights/average values for each strength category. These results are what I will discuss in the next section.

Figure 2.4.1: Implementation of training and evaluation

a) Training and evaluation in the code

```
# Train model
agent = LogisticRegression(random_state=7, max_iter=1000, class_weight='balanced')
agent.fit(X_train, y_train)

# Evaluate
y_pred = agent.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
print(f"Model accuracy: {accuracy * 100:.2f}%\n")
```

b) Accuracy reporting

```
Dataset rows: 669640
Model accuracy: 99.94%
```

3: Analysis and Interpretation of the Data

3.1: Findings

Once my model was functional, I was primarily concerned with two metrics: the weights my model applied to each feature, and the average values of each feature. As I mentioned in the first section, the dataset used in this model is made of passwords which have had their strength classified by a trusted third party resource, so with this data, I hope to reverse engineer the password strength evaluation process and provide some insight into what features truly define strong passwords.

In my logistic regression model, each feature of a given password contributes to the overall probability of whether or not it is a weak (0), medium (1), or strong (2) password. Positive weights correlate the presence of features to a particular class, while negative weights will do the opposite, decreasing the likelihood of a classification if a given feature is present. Weights close to zero suggest that the given feature has little to no impact on the password's classification (Fig. 3.1.1). The average feature values of each classification essentially mirror each corresponding weight (Fig. 3.1.2).

Class 0 (Weak Passwords)

- ***Length (-17.60):***
  A substantial negative weight suggests that short passwords are often considered weak. Shorter passwords provide fewer possible combinations and are easier to guess.

- ***Char. Diversity (Uppercase: -7.54, Lowercase: -7.45, Digits: -7.09, Special: -1.29):***
  These negative weights indicate that passwords lacking these character types are more likely to be weak.

- ***Entropy (-3.35):***
  Low entropy inherently corresponds to high predictability/low randomness in patterns. This is strongly associated with weak passwords.

- ***Uniqueness Ratio (+1.81):***
  Interestingly, a higher uniqueness ratio slightly increases the likelihood of weak classification. This may reflect that very short passwords with unique characters (e.g. "A@1") still fail other strength criteria, namely length.

Class 1 (Medium Passwords)

- ***Length (+1.70):***
  Passwords with moderate length slightly increase the likelihood of medium classification. This suggests that password length alone may not be enough to guarantee a strong classification.

- ***Char. Diversity (Uppercase: +0.52, Lowercase: +0.70, Digits: +0.94, Special: -0.08):***
  These relatively small weights seem to suggest that a reasonable variety of characters contributes to a medium classification.

- ***Entropy (-2.13):***
  Medium passwords tend to have relatively low entropy, though they will likely have higher entropy than weak passwords.

- ***Uniqueness Ratio (+1.42):***
  Similar to weak passwords, a higher uniqueness ratio contributes positively to this class, suggesting that moderately diverse passwords are a hallmark of medium strength.

Class 2 (Strong Passwords)

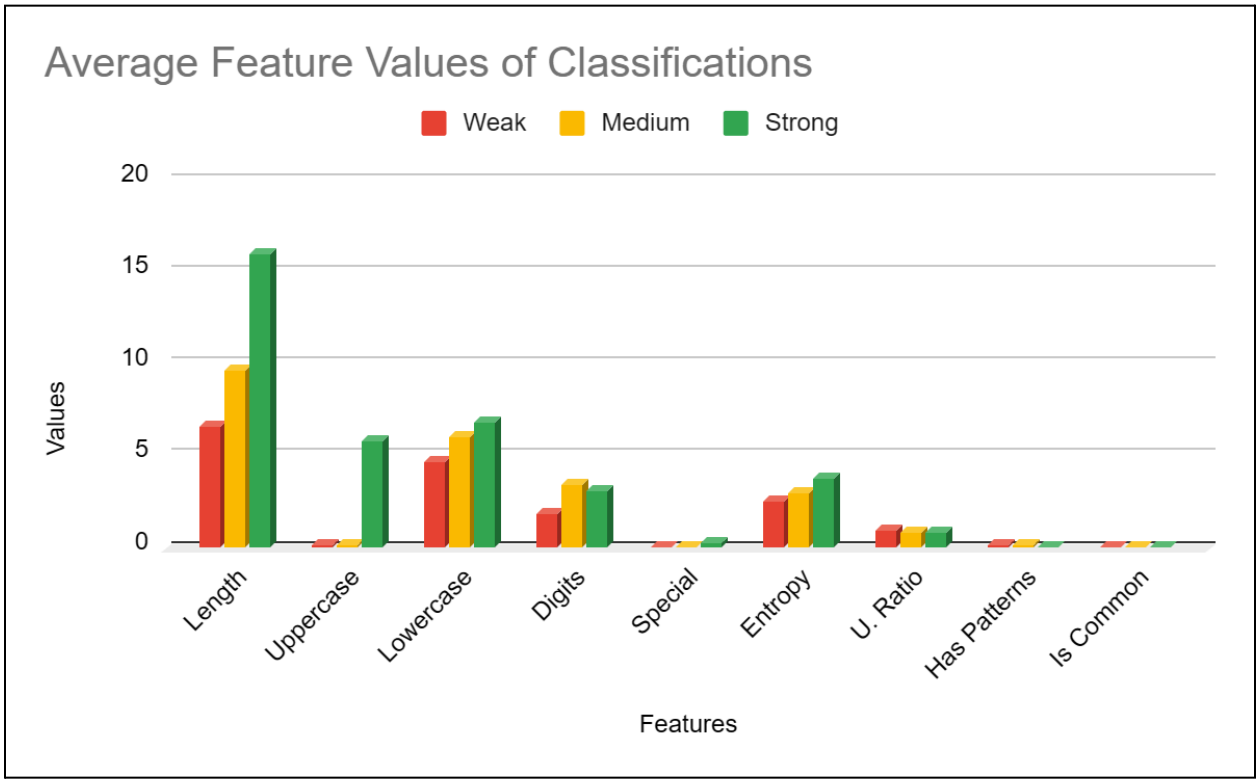
- ***Length (+15.91):***
  An overwhelmingly positive weight for length underscores the critical part it plays in password security. Longer passwords provide exponentially more possible combinations, making them more secure against brute force attacks.

- ***Char. Diversity (Uppercase: +7.02, Lowercase: +6.75, Digits: +6.15, Special: +1.21):***
  There is a strong correlation between strong passwords and character diversity, though not as strong as password length.

- ***Entropy (+5.47):***
  High entropy indicates a high degree of randomness in a password, thus making it more secure against dictionary and guessing attacks.

- ***Uniqueness Ratio (-3.23):***
  Interestingly, there is a negative correlation between a password's strength and its uniqueness ratio. This could reflect how longer passwords have a higher likelihood of repeating characters, thus the uniqueness ratio alone is not an accurate measure of a password's strength.

Figure 3.1.1 (top), 3.1.2 (bottom): Feature Graphs



3.2 Interpretation

Across all classes, length seems to be the dominant factor. Higher length exponentially increases possible password combinations, and this model clearly prioritizes that over anything else. This seems to support the idea that creating passwords as multi-word phrases can be more secure than the traditional word plus appended numbers & symbols approach. For example, given two passwords “Cheeseburger15#” and “ireallyliketoeatcheeseburgers”, my model would categorize the latter as being more likely to be a strong password due to its sheer length despite its lack of character diversity.

Regarding character diversity, my model certainly exhibits a strong positive correlation between diversity and password strength, but to a lesser degree than it does for password length. Overall, this indicates that enhancing an already long password with character variety could produce an even more desirable result. For example, “ireallyliketoeatcheeseburgers” could be turned into “IR3@llyLikeToE@tCh33s3burg3rs!”, or a random string of characters could be appended, enhancing diversity and length simultaneously (“ireallyliketoeatcheeseburgers4e$di*k”).

Passwords with high entropy tend to be strong, while those with lower entropy tend to be weak or medium. However, it was surprising that its influence in the classification process was relatively low (for strong passwords, length was weighted at 15.9, while entropy was only weighted at 5.5).

Lastly, there was an unexpectedly low impact of common pattern/password recognition. Across all classifications, both of these features’ weights never exceeded +-0.05, which is truly minimal. To illustrate this issue, my model classified “baseball” as having medium security, although it is near the top of the list for most common passwords, and is therefore susceptible to dictionary attacks. It appears that the model does not adequately account for this, and views both of these features as negligible to the end result.

4: Reflection

Completing this project and going through the process of developing the code was certainly eye opening. In a way, actually implementing a logistic regression model proved to be straightforward with the power of Scikit. With that resource, I was able to import all of the necessary modules to get my program up and running without needing to manually configure what’s under the hood. This, in turn, allowed me to focus my attention on *how* my agent was being implemented, and what to make of the data it generated.

As I mentioned in the first section, my main focus is currently in cybersecurity, and that industry, like many others, has been heavily influenced by the recent surge in the efficacy and availability of AI. Malicious actors use AI in cases including the creation of personalized phishing emails, voice impersonation, and adversarial machine learning, in which AIs are used to create dynamic, adaptable executable files that can avoid detection and prevention systems (this last use case is a hot topic in AI/cybersecurity research). Contrarily, AI is used in defensive measures including user/file behavior analytics and automated security orchestration and response. Therefore, I chose to incorporate cybersecurity into this project as a way of deepening my knowledge of AI while also exploring ways to apply it to cybersecurity.

While I certainly encountered some programming challenges, these did not hinder my efforts too much. Primarily, my struggles centered around which Python packages to use, and how to actually implement them in the desired way. To help mitigate this issue, I utilized ChatGPT to gain suggestions on which packages to use and to help me understand the syntax those packages used. Scikit, for example, contains a truly vast array of machine learning algorithms, so choosing the right model and implementing it with correct syntax was a task on its own. Ultimately, my program came together, and I can surely say that I can approach future machine learning projects with a bit more ease.

As I worked through this project (and the whole class since the beginning of the semester), I've developed a proficient understanding of various AI concepts, and am certainly interested in learning more about it and how it ties into cybersecurity. I hope this project provided some insight into how we should perceive password security.