

Determining Password Security using ML

Password strength can often be determined using a set of simple rules, like checking for length, the inclusion of uppercase/lowercase letters, digits, special characters, etc., with if-else statements.

However, applying machine learning (ML) offers several advantages over rule-based approaches:

1. Learning from Data Patterns:

- **Rule-based systems** require manually defining rules like:
 - If the password length is greater than 8, give it a medium score.
 - If it contains uppercase, lowercase, numbers, and symbols, consider it strong.
- With **machine learning**, instead of manually coding these rules, the model learns from historical password data. It can capture more complex patterns that might not be obvious. For example, ML can account for common weak passwords like "password123" even though it fits many rule-based strong criteria.

2. Adaptability:

- Password policies evolve over time, and new trends in weak passwords emerge (e.g., passwords like "letmein2024"). With **rule-based** systems, you need to update the rules manually to adapt to new trends.
- An ML model can automatically adapt as it is retrained on new password datasets, learning to recognize newly popular weak passwords that follow existing patterns.

3. Handling Edge Cases:

- Rules often overlook edge cases or uncommon password patterns. For example, a password like "abc123XYZ!" might seem strong by rule-based criteria (contains uppercase, lowercase, digits, and special characters), but it's easy to guess due to its simplicity.
- ML models can consider a wide range of features from the training data and might penalize patterns like these because they are common and weak.

4. Feature Combination:

- **Rule-based systems** handle individual features independently (e.g., length, symbols, etc.), but they often can't account for how features combine in complex ways. For example, a 12-character password may not necessarily be strong if it's entirely lowercase letters.
- ML models naturally handle feature combinations. They can learn that a 12-character lowercase password is not as strong as a shorter, more complex password.

5. Scalability and Fine-Tuning:

- Manually setting and fine-tuning rules becomes increasingly difficult as you aim to balance security and usability. You may end up either blocking legitimate strong passwords or allowing weak ones.
- **ML models** can be fine-tuned by adjusting hyperparameters and retraining with more data, ensuring a more accurate classification over time without manually tweaking each rule.

6. Generalization to Unseen Data:

- With rules, you have to predefine every possible case that could lead to password strength evaluation. If someone uses an unexpected password format, the rules might fail.
- **ML models generalize better** to unseen data, meaning they can recognize new, previously unseen patterns in passwords and correctly classify them as strong or weak.

7. Continuous Improvement:

- Rule-based systems are static unless updated manually. They will always work the way they were programmed, regardless of changes in the environment or trends.
- ML models can be improved by retraining with more or better data. They can incorporate feedback over time, improving accuracy as more examples are provided.

When Rule-Based Systems Make Sense:

- If you're working with a small-scale problem where you can easily define the rules (e.g., enforcing a corporate password policy), then **rule-based systems** are efficient and straightforward.
- If password security is being evaluated in highly controlled environments where the rules are well understood and don't change often, a rule-based approach can work perfectly.

When to Use Machine Learning:

- **Large-Scale Systems:** When you're dealing with large datasets (e.g., millions of passwords) and need to generalize across multiple user behaviors.

- **Complex Evaluations:** If password security needs to go beyond basic rules, considering password structures, historical data on compromised passwords, and patterns, ML provides a flexible and adaptable approach.
- **Dynamic Systems:** For applications like evaluating user-generated passwords in real-time on platforms with evolving trends, ML can be much more effective.

While rule-based approaches are quick and effective for well-defined, simple password policies, machine learning allows for more flexible, data-driven, and adaptive systems that can capture nuances in password security that may not be obvious or easily coded with if-else statements. It also ensures that the system can evolve and generalize better as new data becomes available.

Program to Implement Password Strength Determination

Python libraries required to be installed for the following code:

pandas

sklearn

xgboost

To install use the command: -

```
$ pip install pandas sklearn xgboost
```

We already have pandas and sklearn installed in our environment, so to install xgboost we use the command:

```
$ pip install xgboost
```

```
kali@kali ~  
File Actions Edit View Help  
(kali@kali)~  
$ pip install xgboost  
Defaulting to user installation because normal site-packages is not writeable  
Collecting xgboost  
  Downloading xgboost-2.1.1-py3-none-manylinux_2_28_x86_64.whl.metadata (2.1 kB)  
Requirement already satisfied: numpy in /usr/lib/python3/dist-packages (from xgboost) (1.26.4)  
Collecting nvidia-nccl-cu12 (from xgboost)  
  Downloading nvidia_nccl_cu12-2.23.4-py3-none-manylinux2014_x86_64.whl.metadata (1.8 kB)  
Requirement already satisfied: scipy in /usr/lib/python3/dist-packages (from xgboost) (1.12.0)  
Downloading xgboost-2.1.1-py3-none-manylinux_2_28_x86_64.whl (153.9 MB)  
153.9/153.9 MB 11.3 MB/s eta 0:00:00  
Downloading nvidia_nccl_cu12-2.23.4-py3-none-manylinux2014_x86_64.whl (199.0 MB)  
199.0/199.0 MB 19.7 MB/s eta 0:00:00  
Installing collected packages: nvidia-nccl-cu12, xgboost  
Successfully installed nvidia-nccl-cu12-2.23.4 xgboost-2.1.1  
(kali@kali)~  
$
```

Open the nano editor to write down the program script. Use command nano followed by filename to create.

```
$ nano Password_Security_Check1.py
```

```
(kali㉿kali)-[~]  
$ nano Password_Security_Check1.py
```

Now start writing the code. This is available in this GitHub repository [https://github.com/avnishnaithani/pythonforcybersecurity/blob/main/Domain%207/Password Security Check1.py](https://github.com/avnishnaithani/pythonforcybersecurity/blob/main/Domain%207/Password%20Security%20Check1.py)

And the dataset can be downloaded from the same repository

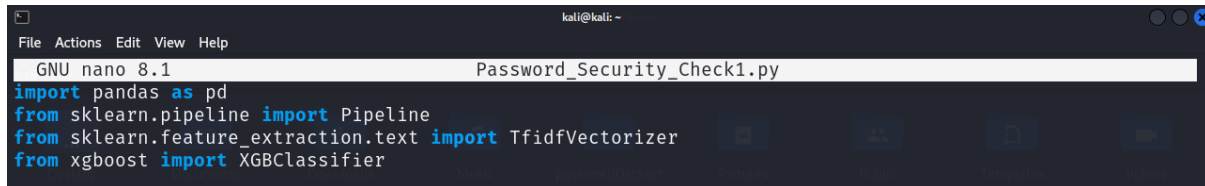
<https://github.com/avnishnaithani/pythonforcybersecurity/blob/main/Domain%20/passwordDataset.csv>

Screenshot of Dataset parameters/features

[illegible]

We break down the code into the following steps: -

Step 1: Importing Libraries



```

File Actions Edit View Help
GNU nano 8.1 Password_Security_Check1.py
import pandas as pd
from sklearn.pipeline import Pipeline
from sklearn.feature_extraction.text import TfidfVectorizer
from xgboost import XGBClassifier
  
```


pandas: This is a data manipulation library used to handle and manipulate tabular data, specifically reading the password dataset and splitting it.

Pipeline: This comes from scikit-learn and helps in chaining multiple steps (like feature extraction and classifier training) into one unified process. It streamlines the process of transforming the data and fitting the model.

TfidfVectorizer: A feature extraction technique that converts text into numerical values based on word importance (Term Frequency-Inverse Document Frequency). In this case, it turns passwords into numerical feature vectors for machine learning models.

XGBClassifier: An implementation of the XGBoost (Extreme Gradient Boosting) algorithm used as a classifier, which is particularly efficient for supervised learning tasks like password strength prediction.

Step 2: Loading and Shuffling the Dataset



```

# Load and shuffle the dataset
df = pd.read_csv("passwordDataset.csv", dtype={"password": "str", "strength": "int"}, index_col=None)
df = df.sample(frac=1)
  
```

Loading the CSV: The `pd.read_csv` function loads the password dataset from a CSV file into a pandas DataFrame. The file should contain two columns: one for passwords (`password`) and one for their corresponding strength scores (`strength`).

- `dtype={"password": "str", "strength": "int"}:` Ensures that the passwords are treated as strings and the strength values as integers.
- `index_col=None:` Tells pandas not to use any column as the index; it will assign default row numbers.

Shuffling the Dataset: The `sample(frac=1)` randomly shuffles the rows in the dataset (with `frac=1`, the entire dataset is shuffled). This ensures that the model doesn't learn based on any inherent order of the data, such as all weak passwords being at the top.

Step 3: Splitting the Dataset into Training and Test Sets

```
# Split into training and testing sets
l = len(df.index)
train_df = df.head(int(l * 0.8))
test_df = df.tail(int(l * 0.2))
```

Get Length of Dataset: `l = len(df.index)` calculates the total number of rows in the dataset, which will be used to split the data.

80% Training Data: `df.head(int(l * 0.8))` takes the first 80% of the rows from the shuffled dataset to use as training data.

20% Testing Data: `df.tail(int(l * 0.2))` takes the last 20% of the rows from the shuffled dataset to use as testing data.

Step 4: Separating Features (Passwords) and Labels (Strength)

```
# Separate labels and features
y_train = train_df.pop("strength").values
y_test = test_df.pop("strength").values
X_train = train_df.values.flatten()
X_test = test_df.values.flatten()
```

Extract Labels (Strength):

- `train_df.pop("strength")` removes the strength column from the training DataFrame and returns it. This gives us the target labels (`y_train`) that we want to predict.

- `test_df.pop("strength")` does the same for the testing DataFrame (`y_test`).

Extract Features (Passwords):

- `train_df.values.flatten()` converts the remaining password column into a 1D array for training.
- Similarly, `test_df.values.flatten()` creates a 1D array of passwords for testing.

At this point:

- `X_train` and `X_test` contain the password strings (the features).
- `y_train` and `y_test` contain the corresponding strength values (the labels).

Step 5: Tokenizing Passwords into Characters

```
# Custom tokenizer function to split passwords into characters
def character_tokens(input_string):
    return [x for x in input_string]
```

This function takes a password as input and returns a list of its individual characters.

Example: If `input_string = "password"`, it returns `['p', 'a', 's', 's', 'w', 'o', 'r', 'd']`.

This is important because passwords are sequences of characters, and we want to represent them at the character level for feature extraction (rather than treating the whole password as a single unit).

Step 6: Building the Machine Learning Pipeline

```
# Build the pipeline with TfidfVectorizer and XGBClassifier
password_clf = Pipeline([
    ("vect", TfidfVectorizer(tokenizer=character_tokens, token_pattern=None)), # Set token_pattern None
    ("clf", XGBClassifier())
])
```


Pipeline: A Pipeline chains together multiple processing steps. In this case, it consists of:

TfidfVectorizer: Converts the passwords into numerical vectors, where each character is treated as a token. We specify:

- `tokenizer=character_tokens`: The custom tokenizer to break passwords into characters.
- `token_pattern=None`: Since we're using a custom tokenizer, the default regex pattern is not needed.

XGBClassifier: An XGBoost classifier that uses the vectorized password data to predict the password's strength.

Step 7: Training the Model

```
# Train the classifier
password_clf.fit(X_train, y_train)
```

The `fit()` function trains the model on the training data:

- `X_train`: The vectorized password data.
- `y_train`: The corresponding strength labels.

The model learns to predict the password strength based on the patterns in the character sequences.

Step 8: Evaluating the Model

```
# Evaluate the classifier
score = password_clf.score(X_test, y_test)
print(f"Model accuracy: {score}")
```

The `score()` method evaluates the trained model's accuracy on the test data:

- `X_test`: Vectorized passwords from the test set.
- `y_test`: The actual strength values from the test set.

It prints the accuracy of the model (i.e., how well it correctly predicted password strength for unseen data).

Step 9: Making Predictions

```
# Test predictions
entered_password = input("Enter password to determine its strength:")
predictions = password_clf.predict([entered_password])
print(f"Predictions: {predictions}")

if predictions==[0]:
    print("Password Strength: Weak")
elif predictions==[1]:
    print("Password Strength: Medium")
elif predictions==[2]:
    print("Password Strength: Strong")
```

`predict()`: This method predicts the strength of new passwords:

- The pipeline first vectorizes the passwords (`common_password` and `strong_computer_generated_password`).
- Then the trained `XGBClassifier` predicts their strength.

Convert file to executable form using the command:

```
$ chmod +x Password_Security_Check1.py
```

```
(kali㉿kali)-[~]
$ chmod +x Password_Security_Check1.py
```

Finally execute the program using:

```
$ python3 Password_Security_Check1.py
```

```
(kali㉿kali)-[~]
$ python3 Password_Security_Check1.py
```

We get the following output:

```
(kali㉿kali)-[~]
$ python3 Password_Security_Check1.py
Model accuracy: 0.9796829616134163
Enter password to determine its strength:abcde
Predictions: [0]
Password Strength: Weak

(kali㉿kali)-[~]
$ python3 Password_Security_Check1.py
Model accuracy: 0.9802877687098195
Enter password to determine its strength:john12345
Predictions: [1]
Password Strength: Medium
```

```
(kali㉿kali)-[~]
$ python3 Password_Security_Check1.py
Model accuracy: 0.979921897750267
Enter password to determine its strength:John123@
Predictions: [1]
Password Strength: Medium

(kali㉿kali)-[~]
$
```

Here we can see examples of weak and medium strength passwords.

Notice that John123@ is a medium strength password. But, according to rule-based systems, this will be classified as strong as it satisfies all the traditional rules – At least 1 capital letter, small letter, number and special character.

Below is an example of how to convert it to a strong password:

```
(kali㉿kali)-[~]
$ python3 Password_Security_Check1.py
Model accuracy: 0.9808179082634569
Enter password to determine its strength:*123John789%
Predictions: [2]
Password Strength: Strong

(kali㉿kali)-[~]
$
```

So, this is where Machine Learning models trained on password data has the advantage over traditional methods.

This approach uses character-level tokenization and applies machine learning, enabling it to capture complex password patterns rather than relying on predefined rules.