****



**Institute of Technology, Nirma University**

**Innovative Assignment Mathematical Foundation for Computer Science**

**ENTROPY ANALYSIS OF**

**PASSWORD STRENGTH**

Prepared By

Yatrik Prajapati (25BAM505)

Om Patel (25BCE518)

**1. Introduction**

In the landscape of modern computer science, cybersecurity stands as a paramount and ever-evolving challenge. Our digital lives—from personal banking and private communications to corporate and national infrastructure—are secured by a simple, ancient mechanism: the password. This fundamental layer of digital security is often the only barrier standing between sensitive data and malicious actors. Yet, the methods by which we create and validate these critical gatekeepers are often archaic, relying on simplistic rules that fail to capture the true, mathematical essence of security.

## **The Core Problem: Predictability**

The primary vulnerability of any password system is not its length or complexity, but its predictability. Hackers and security adversaries do not guess passwords randomly. They operate with intelligent, data-driven strategies. Techniques like dictionary attacks leverage massive lists of common words, phrases, and (most importantly) previously leaked passwords. More advanced brute-force attacks don't just try "aaa," "aab," "aac"; they use sophisticated character patterns and statistical models based on common human choices.

The uncomfortable truth is that humans are terrible at creating randomness. We are creatures of habit. We use:

* Familiar names: john, lisa
* Sequential numbers: 123456, 98765
* Common keyboard patterns: qwerty, asdfg
* Simple substitutions: P@ssword1! (This password meets many "strong" criteria—uppercase, lowercase, symbol, number—yet it is one of the first patterns a hacker will try).

This human tendency for patterns is what makes leaked password databases like RockYou.txt so valuable to attackers. They provide a statistical blueprint of our collective password-creation psychology, allowing attackers to refine their guessing algorithms to be incredibly efficient.

**## The Flaw in Our Current Defenses**

Current security policies attempt to fight this predictability with arbitrary rules. A system that demands "one uppercase letter, one number, and one symbol" is a blunt instrument. It forces complexity but does not guarantee unpredictability. As seen with P@ssword1!, users often satisfy these rules in the most predictable way possible, checking boxes without adding any meaningful security.

This highlights a major gap: we lack a standardized, quantitative method to measure password strength. We need to move beyond "does it have a symbol?" and ask, "How much *information* does this password contain? How much uncertainty or randomness does it *actually* represent?"

## **A Mathematical Solution: Shannon Entropy**

This project provides a direct solution to this problem by applying a foundational concept from applied mathematics and information theory: Shannon Entropy. Developed by Claude Shannon, the "father of the information age," entropy is a concrete, numerical measure of uncertainty or randomness within a set of data.

Instead of just looking at one password, we will analyze an entire dataset of millions of leaked passwords. By calculating the entropy of this dataset, we can answer critical questions:

* On average, how many "bits" of information does a character in a password from this list *actually* contain?
* How skewed is the distribution of characters? (Are 'a', 'b', 'c', '1', '2', '3' used exponentially more than 'Z', 'X', '!', '%'?)

This project bridges the gap between theoretical computer science and practical cybersecurity. We will build a tool to analyze a password file and produce a single, definitive entropy score. This score is not a guess or a rule of thumb; it is a mathematical calculation that definitively measures the randomness of the password set. This provides a formal foundation for understanding password strength and building smarter, data-driven security policies for the future.

**2. About the Project**

**2.1 Project Definition and Objectives**

This project is an **empirical investigation** into the quantitative nature of password security. It moves beyond the theoretical realm of "what makes a good password" and enters the practical, data-driven world of "what makes a *leaked* password database so weak?" At its core, this project is the design and implementation of a **diagnostic tool**—a Python-based analysis pipeline that ingests raw password data and produces a definitive, mathematical measure of its predictability.

The primary objective is to **quantify the randomness (or lack thereof) of real-world passwords** using the formal concept of **Shannon Entropy**. Instead of relying on anecdotal evidence, we will provide two key outputs:

1. **A Quantitative Metric:** A single, precise entropy score (measured in bits) that represents the average "information" or "surprise" contained in each character of the password set.
2. **A Visual Proof:** A frequency distribution graph that visually demonstrates *why* the entropy score is what it is, highlighting the extreme overuse of a few predictable characters.

This project, therefore, serves as a bridge between the abstract mathematical theory of information and the practical, high-stakes field of cybersecurity.

**## 2.2 The Core Concept: Measuring Uncertainty with Shannon Entropy**

The entire project is built upon the foundational mathematical concept of **Shannon Entropy**. This score, measured in "bits," is not an abstract number. It has a tangible, intuitive meaning: **Entropy is the average number of "yes/no" questions you would need to ask to correctly guess a character if one were pulled at random from the dataset.**

* **Information as "Surprise":** Entropy measures uncertainty. A character that is highly predictable (like the letter 'u' after a 'q' in English) carries very little "information" or "surprise." A character that is highly unpredictable (like a symbol % in the middle of a word) carries a high amount of information.
* **The Role of Data Distribution:** Entropy is not just about *how many* unique characters are used, but about *how evenly* they are distributed. A password set that uses all 90+ characters (uppercase, lowercase, numbers, symbols) but uses the letter 'a' 90% of the time would still have extremely low entropy.

**## 2.3 The Two Faces of Entropy: Our Hypothesis**

Our project is designed to test a specific hypothesis: that real-world password lists are not just weak, but **mathematically and measurably closer to a known, patterned language (like English) than to a truly random string of data.** We can define this hypothesis by contrasting two states:

**## The Low Entropy State (The Reality)**

This state signifies high predictability, order, and pattern. This is what we expect to find in RockYou.txt.

* **Cause:** Human psychology. We rely on common words, names, keyboard patterns, and simple number sequences.
* **Mathematical Signature:** A highly **skewed probability distribution**. A few characters ('a', 'e', 's', '1', '2', '0') have a very high probability $p(x)$ of appearing.
* **The Visual Proof:** The resulting frequency graph is a **"skyline"**—dominated by a few "skyscraper" bars, with the vast majority of other characters (especially symbols and uppercase letters) being almost non-existent.
* **The Result:** A dangerously low entropy score (e.g., 2.5 - 3.5 bits), proving that an attacker needs to ask very few "questions" to guess the next character.

**## The High Entropy State (The Ideal)**

This is the goal of all secure systems. It represents maximum unpredictability, chaos, and a lack of discernible patterns.

* **Cause:** True randomness, such as the output from a cryptographic generator or a well-designed password manager.
* **Mathematical Signature:** A **uniform probability distribution**. Every one of the 90+ possible characters has a roughly equal probability $p(x)$ of appearing.
* **The Visual Proof:** The frequency graph would be a **"flat plateau."** All 90+ bars would be approximately the same height, with no single character being a better "bet" than any other.
* **The Result:** A very high entropy score (e.g., > 5.0 bits). For a set of 94 characters (a standard keyboard), the theoretical maximum entropy would be $\log\_2(94) \approx 6.55 \text{ bits}$.

**## 2.4 The Analysis Pipeline: What We Built**

Our project is a Python script that automates this entire investigation. It functions as a two-stage pipeline:

1. **Stage 1: The Quantitative Analysis Engine**
   * **Input:** A raw .txt password file of any size.
   * **Process:** The script reads the entire file, carefully handling non-standard character encodings (latin-1). It cleans the data (by removing newlines) and then uses an efficient Counter object to create a comprehensive frequency table of every unique character.
   * **Calculation:** It iterates this table, applying the Shannon Entropy formula $H(X) = -\sum\_{i=1}^{n} p(x\_i) \log\_2(p(x\_i))$ to compute the final, precise entropy score.
   * **Output (Metric):** This score is printed to the console.
2. **Stage 2: The Visual Evidence Generator**
   * **Input:** The same character frequency table from Stage 1.
   * **Process:** The script selects the **Top 30** most common characters (a number chosen for graph readability) and prepares them for plotting.
   * **Output (Visualization):** Using matplotlib, the script generates and saves a **bar chart** that provides the visual "proof" to accompany the metric. This chart allows any observer, even a non-technical one, to *see* the password set's predictability.

**3. Working**

This project's "Working" section details the end-to-end execution of our analysis pipeline, from the initial problem statement to the final visual output. The process is implemented as a single, self-contained Python script that programmatically follows the 5-step analysis framework.

**## Step 3.1: Identifying the Problem (The Technical Goal)**

The problem is to quantify the predictability of a given password corpus. From a technical standpoint, the goal is to create a program that can:

1. Read a text file of arbitrary size (potentially gigabytes) containing one password per line.
2. Process this data efficiently without running out of memory or crashing on unknown characters.
3. Apply a consistent, mathematically sound formula to this data.
4. Produce a single numerical score and a human-readable graph that, together, serve as a definitive report on the dataset's randomness.

**## Step 3.2: Designing the Mathematical Requirement (The Formula)**

The entire project's analytical power comes from the **Shannon Entropy formula**. This is the core mathematical requirement.

$$H(X) = -\sum\_{i=1}^{n} p(x\_i) \log\_2(p(x\_i))$$

Here is a detailed breakdown of how we translate this formula into a working algorithm:

* **$H(X)$ (The Output):** This is the final entropy score in "bits." It's the single variable our entire calculation aims to solve for. We initialize it as entropy = 0.0.
* **$\sum$ (The Process):** This is the "summation" symbol. In programming, this translates to a for loop. We must loop through *every unique character* (i=1 to n) found in the dataset, calculate its part of the formula, and add it to our entropy variable.
* **$p(x\_i)$ (The Probability):** This is the probability of any single character $x\_i$ appearing. We calculate this using a two-step "preprocessing" phase:
  1. **Count:** We must first count the *total number of all characters* in the file. This becomes our denominator, text\_length.
  2. **Frequency:** We also count the *specific frequency of each unique character* (e.g., 'a' appears 50,000 times). This is the count for that character.
  3. **Calculate:** For each character in our loop, we calculate probability = count / text\_length.
* **$\log\_2(p(x\_i))$ (The "Information Content"):** This is the "magic" of the formula. We use Python's math.log2() function.
  1. This logarithm measures "surprisal." A very common character (high probability $p(x)$) will have a $\log\_2$ value close to 0, contributing very little to the total entropy.
  2. A very rare character (low probability $p(x)$) will have a large negative $\log\_2$ value. When multiplied by the negative sign in the formula, it contributes a large *positive* amount to the total entropy.

In summary, our program works by first building a complete probability model of the entire dataset ($p(x\_i)$ for all $x\_i$) and then iterating through that model to calculate the total weighted-average "surprisal," which is the Shannon Entropy.

**## Step 3.3: Data Sourcing & Preprocessing (The Implementation)**

This is the first practical stage of the Python script's execution.

1. **Sourcing:** The script is hard-coded to look for rockyou.txt (though this can be easily changed). This file is chosen as a "worst-case scenario" benchmark, as it is one of the largest and most widely-used leaked lists in security research.
2. **Reading the File:** The open() command is used with a critical parameter: encoding='latin-1'. This is a crucial technical choice. Standard utf-8 encoding will **fail** because it cannot process the unusual, non-standard, and sometimes corrupt byte sequences found in real-world password dumps. latin-1 is a "safer" encoding that maps every possible byte value to a character, ensuring no data is lost and the program doesn't crash.
3. **Data Cleaning:** The *only* cleaning step is to remove newline characters (\n) using password\_data.replace('\n', ''). This is vital because the "Enter" keypress (a newline) that separates passwords is not part of the password itself. Including it in our analysis would incorrectly skew the statistics.

**## Step 3.4: Performing the Analysis (The Code in Action)**

This is the core logic of the script, where the mathematics from Step 3.2 is executed.

1. **Total Character Count:** The script first calculates text\_length = len(text). This is our denominator for all probability calculations.
2. **Character Frequency Counting:** Instead of a slow, manual for loop, the script uses a highly-optimized Python tool: collections.Counter(text). This single command instantly builds a dictionary-like object (char\_counts) that maps every unique character to its precise frequency (e.m., {'a': 54820, 'b': 12015, ...}). This is the most efficient way to get all our numerators.
3. **The Entropy Loop:** The script then iterates through the char\_counts object. The main() loop looks like this:

Python

entropy = 0.0

for char, count in char\_counts.items():

# Step 1: Calculate p(x\_i)

probability = count / text\_length

# Step 2: Calculate p(x\_i) \* log2(p(x\_i))

# We subtract because the formula has a minus sign

entropy -= probability \* math.log2(probability)

1. **Final Output:** After the loop completes, the entropy variable holds the final score. The script prints this value, formatted to 4 decimal places, providing a precise numerical result.

**## Step 3.5: Visualization (Generating the Proof)**

The final stage of the script's operation is to provide visual evidence for the numerical result.

1. **Finding Key Data Points:** A chart with all 90+ unique characters would be unreadable. Therefore, the script uses the char\_counts.most\_common(30) function. This intelligently sorts all characters by frequency and selects only the "Top 30." This is a deliberate choice for **data visualization clarity**.
2. **Preparing for Plotting:** The matplotlib library requires separate lists for labels (the x-axis) and values (the y-axis). The script uses zip(\*common\_chars) to "unzip" the list of pairs into two distinct lists: characters and counts.
3. **Building the Graph:** The script executes a series of matplotlib commands:
   * plt.figure(figsize=(15, 7)): Creates a large, wide canvas so the labels don't overlap.
   * plt.bar(characters, counts, ...): Generates the actual bar chart.
   * plt.title(...), plt.xlabel(...), plt.ylabel(...): Sets the labels for the graph.
   * plt.xticks(rotation=60): Rotates the x-axis labels (the characters) so they are readable.
4. **Displaying the Result:** Finally, plt.savefig(...) saves a high-quality PNG of the graph to the project folder, and plt.show() displays it to the user. This graph provides the undeniable, visual proof of *why* the entropy is low: the bars are extremely skewed, showing a massive over-reliance on a few characters.

**4. Features**

This project is more than a simple script; it is a complete, self-contained analysis tool. Its features are designed to provide a comprehensive, end-to-end solution, moving from raw data to a final, insightful conclusion.

**## 4.1 Quantitative Analysis Engine**

This is the central feature and primary deliverable of the project. The tool does not provide a vague, qualitative assessment; it provides a hard, **quantitative metric** of randomness.

* **Shannon Entropy Score:** The script's main output is the Shannon Entropy $H(X)$, calculated with high precision and expressed in **bits**. This score is the industry-standard measure of information and uncertainty.
* **From Abstract to Concrete:** This feature successfully transforms the abstract concept of "password strength" into a single, concrete, and comparable number. This allows for objective comparison between different password lists or security policies.
* **Mathematical Purity:** The calculation is a direct, pure implementation of the mathematical formula, not an approximation. It accounts for every single character in the dataset, ensuring the final score is a true and accurate representation of the entire corpus.

**## 4.2 Automatic Data Visualization**

A numerical score is powerful but lacks intuitive impact. This feature provides the essential **visual evidence** to support the quantitative findings, making the results accessible to any audience, regardless of their mathematical background.

* **Top 30 Frequency Chart:** The script automatically generates and displays a bar chart of the **Top 30 most frequent characters**. This is a deliberate design choice for readability—a chart with all 90+ characters would be an unreadable mess.
* **Instant Insight:** This graph instantly reveals the *why* behind the low entropy score. Observers can *see* the **skewed probability distribution** with their own eyes. The "skyscrapers" of 'a', '1', 'e', 's' next to the near-zero bars for symbols (!, @, #) and uppercase letters (Q, Z, X) provide undeniable proof of predictability.
* **Report-Ready Output:** The script automatically saves this visualization as a high-quality character\_distribution.png file, making it a "report-ready" asset that can be directly embedded in presentations or, as in this case, the final project documentation.

**## 4.3 Robustness & Real-World Scalability**

This tool is not a "toy" program; it is designed to be a robust utility capable of handling real-world, messy, and massive datasets.

* **Handles Encoding Errors:** A critical feature is the use of the encoding='latin-1' parameter. Real-world password dumps are not clean utf-8 text; they are binary files containing corrupt, non-standard, and "illegal" byte sequences. This encoding choice ensures the script **will not crash** and will process every single byte of data, providing a complete analysis.
* **Efficient Processing:** The script is optimized for performance. By using collections.Counter, it leverages a highly-optimized, C-backed data structure to perform the frequency count. This is orders of magnitude faster than a naive for loop and a standard dictionary, allowing it to process files with **tens of millions of passwords** in seconds or minutes, not hours.

**## 4.4 Self-Contained & Portable**

The project is designed for simplicity, accessibility, and ease of use.

* **Single-File Solution:** The entire application—data analysis, mathematical calculation, and visualization—is contained within a **single .py script**. There are no complex installations, no databases to configure, and no web servers to run.
* **Minimal Dependencies:** It relies only on standard, ubiquitous Python libraries (math, sys, collections) and two of the most common data science packages (matplotlib, numpy), which are easily installed with a single command (pip install matplotlib numpy).
* **Portability:** This self-contained nature means the tool is highly portable. It can be run on any system with Python installed, whether it's a personal laptop, a university server, or a cloud virtual machine.

**## 4.5 Extensible & Flexible by Design**

While this *case study* is focused on passwords, the *tool* itself is a general-purpose entropy calculator. This flexibility is a powerful feature.

* **Data-Agnostic:** The script does not know what a "password" is. It only knows text. This means it can be immediately repurposed to analyze the entropy of *any* text file. For example:
  + Analyze the entropy of the English language (by feeding it *Moby Dick*).
  + Analyze the entropy of C++ or Python source code (by feeding it .cpp or .py files).
  + Compare the entropy of an unencrypted text file vs. its encrypted, truly random counterpart.
* **Modular Code:** The code is written in a modular fashion with separate functions for calculate\_entropy() and plot\_char\_distribution(). This makes it easy for other developers to import this script as a library into a larger application, such as a full-scale cybersecurity dashboard.
* **6. Source Code Link**
* https://github.com/YourUsername/Password-Entropy-Analyzer