Implementing a Random Passphrase Generator using Markov Chains

Alan Matthew

Abstract

This document presents the design and implementation of a Markov Phrase Generator, a simple yet effective tool for generating passphrases using Markov chains based on a given training text. The project demonstrates the application of Markov chains in text generation by creating a transition matrix from the training data, which is then used to generate new phrases.

Contents

1 Introduction

The Markov Phrase Generator is a passphrase generation tool that leverages the statistical model of Markov chains. By analyzing a corpus of text, it creates a transition matrix that models the likelihood of sequences of words, allowing for the generation of new, semantically plausible phrases.

One application of this tool is in the context of cryptocurrency wallets, where secure and memorable passphrases are essential for authorizing transactions and recovering accounts. The Markov Phrase Generator aims to address the challenge of creating such passphrases by offering a balance between security and memorability by leveraging the linguistic patterns found in a chosen corpus of text.

2 Data Collection and Preprocessing

The training data for the Markov Phrase Generator project was sourced from a collection of verses from Kanye West's songs, available on the Kaggle dataset titled Kanye West Verses. This dataset comprises various lines from Kanye West's discography, providing a rich corpus of contemporary language use and stylistic phrasing. The choice of this dataset was motivated by its diversity in vocabulary and syntax, enabling the generation of creative and memorable passphrases. Furthermore, with regards to security, the dataset's distinct style and content can contribute to the uniqueness and unpredictability of the generated passphrases, such that the probability of the tool generating the same phrase twice is extremely low.

Prior to usage, we need to preprocess the data to ensure that the Markov chain model can effectively capture the linguistic patterns in the training text. This involves cleaning and normalizing the text by removing non-alphabetic characters and converting all words to lowercase. The following Rust function, `clean_word`, accomplishes this task efficiently.

```rust
fn clean_word(word: &str) -> String {
  word.chars()
      .filter(|c| c.is_ascii_alphabetic())
      .collect::<String>()
      .to_lowercase()
}
```

Listing 1: Preprocessing Function

3 Problem Statement

The problem of generating secure and memorable passphrases for cryptocurrency wallets can be formally stated as a network problem in the context of Markov chains. This problem is formalized as follows: given a network which consists of states represented by pairs of words from the training corpus, and the transitions between these states are determined by the subsequent word that follows a given pair, how can we navigate this network (or graph) to construct a sequence of words (passphrase) that is both unpredictable and semantically coherent.

4 Assumptions and Limitations

The Markov Phrase Generator operates under several assumptions:

- **Linguistic Coherence**: It assumes that a sequence of words generated based on the statistical patterns in the training corpus will be semantically coherent and meaningful to users.

- **Security through Randomness**: The model presumes that by generating passphrases based on a probabilistic method, the outputs will be sufficiently random to ensure security against brute-force or dictionary attacks.

However, there are limitations to this approach:

- **Predictability**: The reliance on a fixed corpus, especially one with a distinctive style like Kanye West's lyrics, may introduce patterns that could potentially be exploited by sophisticated attackers.
- **Memory vs. Security Trade-off**: While aiming for memorable passphrases, there's a risk that the generated phrases might become too predictable or not secure enough for cryptographic purposes.
- **Cultural Sensitivity**: The content of the training data may reflect the biases or explicit themes of the source material, which might not be appropriate for all users.

5 Core Concepts

5.1 Transition Matrix Creation

The transition matrix is a pivotal component in the Markov Phrase Generator. It records the probability of transitioning from one pair of words to another within the corpus. This matrix is constructed by parsing through each triplet of consecutive words in the training text, treating the first two words as the key and the third word as a possible continuation. The Rust function `create_transition_matrix` accomplishes this task efficiently.

```rust
fn create_transition_matrix(words: Vec<String>) -> HashMap<(String, String), Vec<String>> {
    let mut transition_matrix = HashMap::new();
    for window in words.windows(3) {
        if let [w0, w1, w2] = &window {
            let entry = transition_matrix
                .entry((w0.clone(), w1.clone()))
                .or_insert_with(Vec::new);
            entry.push(w2.clone());
        }
    }
    transition_matrix
}
```

Listing 2: Creating the Transition Matrix

This function iterates over each triplet (or window of 3 words) in the training data. For each triplet, it maps the first two words to a list of possible third words, effectively capturing the essence of a Markov chain's transition probabilities in a discrete and manageable form.

5.2 Markov Chain for Passphrase Generation

The core of passphrase generation lies in simulating a Markov chain process, where the next word is chosen based on the transition matrix's probabilities. The `markov_chain` function implements this idea by selecting the next word in the sequence with the help of the transition matrix created earlier.

```rust
fn markov_chain(
    transition_matrix: &HashMap<(String, String), Vec<String>>,
    length: usize,
    w0: String,
```

```
    w1: String,
    w2: String,
) -> Vec<String> {
    let mut rng = rand::thread_rng();
    let mut result = vec![w2.clone()];
    let (mut _w0, mut w1, mut w2) = (w0, w1, w2);
    for _ in 0..length - 1 {
        if let Some(next_words) = transition_matrix.get(&(w1.clone(), w2.clone())) {
            let next_word = next_words[rng.gen_range(0..next_words.len())].clone();
            result.push(next_word.clone());
            _w0 = w1;
            w1 = w2;
            w2 = next_word;
        }
    }
    result
}
```

Listing 3: Markov Chain Function

This function starts with an initial seed (the last word of the starting triplet) and generates a sequence of words by randomly selecting from the list of possible continuations for the current state (the last two words). This randomness introduces variety in the output, making each generated passphrase unique.

5.3 Generating the Passphrase

The `generate_passphrase` function orchestrates the passphrase generation process, beginning with reading and cleaning the training data, followed by constructing the transition matrix, and finally generating the passphrase using the Markov chain.

```
fn generate_passphrase(words: Vec<String>, length: usize) -> String {
    let clean_words = words
        .into_iter()
        .map(|w| clean_word(&w))
        .collect::<Vec<String>>();
    let transition_matrix = create_transition_matrix(clean_words.clone());
    let start_index = rand::thread_rng().gen_range(0..clean_words.len() - 3);
    let chain = markov_chain(
        &transition_matrix,
        length,
        clean_words[start_index].clone(),
        clean_words[start_index + 1].clone(),
        clean_words[start_index + 2].clone(),
    );
    chain.join(" ")
}
```

Listing 4: Generating the Passphrase

This function showcases the end-to-end process of generating a passphrase. It starts by cleaning the input words to ensure consistency, creates a transition matrix to capture the linguistic structure of the input text, and then employs the Markov chain to generate a sequence of words that form the passphrase. The choice of starting words (seed) for the Markov chain is randomized to ensure diversity in the generated passphrases.

6 Installation and Running the Project

6.1 Installation

To get started with the Markov Phrase Generator, you first need to set up Rust and Cargo on your system. Follow these steps:

1. **Open a Terminal or Command Prompt.**
2. **Install Rustup:** Rustup is the installer for the Rust programming language. It also installs Rust and Cargo. Execute the following command:
   - On Unix-based systems (Linux, macOS):

     ```
     curl -proto '=https' -tlsv1.2 -sSf https://sh.rustup.rs | sh
     ```
3. **Follow the on-screen instructions:** The installer will guide you through the setup. It's recommended to proceed with the default options.
4. **Restart your Terminal or Command Prompt:** This step ensures that your environment variables are updated.
5. **Verify Installation:** Confirm that Rust and Cargo are installed by running:

   ```
   rustc -version
   ```

   This command should return the installed Rust version.

6.2 Running the Project

With Rust and Cargo ready, you can now run the Markov Phrase Generator.

1. **Clone the Project:** Clone the project repository from GitHub using the following command:

   ```
   git clone https://github.com/yokurang/passphrase-generator.git
   ```
2. **Navigate to the Project Directory:** Change to the project directory:

   ```
   cd passphrase-generator
   ```
3. **Build the Project:** Compile the project using Cargo:

   ```
   cargo build
   ```
4. **Run the Project:** Launch the executable, specifying the passphrase length:

   ```
   cargo run - 12
   ```

This command generates a 12-word passphrase, leveraging the linguistic patterns found in `kanye_verses.txt`.

7 Conlusion

In conclusion, we were able to implement a Markov Phrase Generator that effectively generates passphrases based on a given training corpus. The project demonstrates the application of Markov chains in text generation, offering a practical tool for enhancing security for cryptocurrency wallets. While mindful of its assumptions and limitations, this project underscores the potential of applying natural language processing techniques in the realm of cybersecurity.