universidade de aveiro deti departamento de eletrónica, telecomunicações e informática

Modelo de Cópia para Compressão de Dados e Geração de Automática de Texto

Teoria Algorítmica da Informação

Docente Armando Pinho e Diogo Pratas
2022/2023

Mestrado em Engenharia Informática

Artur Romão, 98470
João Reis, 98474
Pedro Sobral, 98491
Tiago Coelho, 98385

Índice

1. Introdução

O problema proposto pelos docentes da Unidade Curricular Teoria Algorítmica da Informação é a compressão de dados, mais especificamente, estimar a probabilidade do próximo símbolo ser o mesmo que o símbolo previamente previsto, usando um modelo de cópia.

O objetivo deste modelo é prever o próximo caractere que pode aparecer no texto com base nos padrões de caracteres lidos anteriormente.

Para solucionar o problema acima, foram desenvolvidos dois programas, um com a implementação do modelo de cópia, para compressão de dados, *cpm*, e o outro para gerar texto automaticamente, seguindo o modelo de cópia anteriormente implementado, *cpm_gen*.

Este relatório tem o intuito de documentar os passos e decisões tomadas durante a implementação dos programas desenvolvidos para resolver o problema supramencionado, bem como incluir uma análise e discussão dos resultados obtidos, com diferentes parâmetros de entrada.

2. Estruturas de dados

- ***unordered_map***

De maneira a registar a ocorrência de determinada sequência de *k* caracteres para o *Copy Model*, no ficheiro *cpm.cpp*, foi utilizada uma estrutura de dados nativa do C++ que implementa uma *hash table*, ***unordered_map***. O nosso *unordered_map*, ao qual demos o nome de *un_map*, tem a seguinte estrutura: *<string, list<int>>*, sendo que *string* corresponde à ***key***, que neste caso é a **sequência de *k* caracteres** e *list<int>* corresponde aos ***values*** do map, que, neste caso, são os **índices das posições** em que esta sequência de *k* caracteres apareceu anteriormente.

Para o gerador, no ficheiro *cpm_gen.cpp*, o *unordered_map* também foi usado, embora com outra finalidade. A sua estrutura interna é, neste caso, *<string, vector<float>>*, sendo que cada posição do *vector<float>* guarda contadores para cada caractere. Um contador corresponde ao número de vezes que um caractere apareceu logo após aquela sequência de *k* caracteres.

- ***vector***

Esta estrutura foi utilizada para guardar todas as sequências de tamanho *k* presentes no ficheiro. A este ***vector<string>*** demos o nome de ***k_word_read_vector***, que vai ser essencial para nos auxiliar a registar as posições de uma determinada sequência no ficheiro e, consequentemente, as probabilidades associadas ao próximo caractere.

3. Implementação do Modelo de Cópia

O programa *cpm* lê um ficheiro de texto, cujo nome é indicado através da linha de comandos, bem como um parâmetro *alpha*, um valor para o *threshold* e um valor para o *K*. A cada iteração, o programa lê uma palavra de tamanho *K* e armazena essa palavra num vetor (*k_word_read_vector*). O programa verifica se o *unordered_map* contém a palavra atual e, em caso afirmativo, faz uma previsão do próximo caractere. Se não contiver a palavra atual, ela é adicionada ao vetor e ao *unordered_map*, mas a previsão não é feita.

O modelo de previsão do próximo caractere é implementado usando um mapa não ordenado (*unordered_map*), onde a chave é uma string que representa as palavras já lidas e o valor é uma lista de índices que apontam para a posição da respetiva palavra na sequência.

Quando uma previsão é feita, o modelo usa a chave para procurar a lista de índices correspondentes. Em seguida, o modelo percorre a lista de índices e prevê o próximo caractere para cada um deles até atingir o threshold, com base no caractere que apareceu a seguir à sequência com o índice que estamos a analisar. Calcula a probabilidade desse caractere ser o próximo.

Esta probabilidade é calculada com base no número de vezes que a palavra foi prevista corretamente em relação ao número total de vezes que ela apareceu no texto, tal como indicado matematicamente na próxima imagem.

$$P(\text{hit}) \approx \frac{N_h + \alpha}{N_h + N_f + 2\alpha},$$

Fig. 1 - Fórmula matemática para o cálculo da probabilidade

O modelo seleciona o índice com a probabilidade mais elevada e faz a previsão do próximo caractere com base nesse índice.

Após o cálculo da probabilidade, é então calculado o número de bits necessários para armazenar as previsões feitas pelo modelo, calculando a entropia, ou seja, aplicando a seguinte fórmula: −log2 P(hit).

Além disso, o tamanho do mapa tem um tamanho máximo de cinco (5) posições, ou seja, se o mapa estiver cheio e for adicionada uma nova posição, a posição mais antiga é removida. Isso ajuda a reduzir o uso de memória e o tempo de processamento do modelo.

Existe também um *threshold* para determinar quando uma previsão deve ser interrompida (*default* = 4). Este valor corresponde ao número de previsões erradas feitas pelo modelo, isto é, o número de *fails*.

4. Implementação do Modelo de Geração de Texto

Por forma a implementar o modelo de geração de texto, foi tido em conta o trabalho desenvolvido no Modelo de Cópia e as diversas informações prestadas durante as aulas. Para podermos gerar texto, usamos um ficheiro que serve como exemplo, para que o modelo seja capaz de aprender a imitar o estilo e a estrutura do texto.

Primeiramente, começa-se por ler o ficheiro, guardar as diversas sequências de *k* chars como strings num vector e as diferentes letras do ficheiro. Tendo estas duas informações, podemos então guardar num *unordered_map*, cada sequência e um vector que corresponde a um contador de cada char diferente que prossegue a sequência (Exemplo para sequência “TAA”: TAA 0 2 0 1).

Segundamente, temos todas as informações necessárias para podermos começar a gerar texto, pelo que começamos por selecionar a última sequência presente no vector e vamos obter do *unordered_map* as diferentes contagens das letras. Com estas contagens, calculamos a sua soma e as diferentes probabilidades de cada letra, obtendo assim cada probabilidade para prever a letra que sucede a sequência. Posteriormente, foram usadas estas 3 diferentes estruturas para gerar a próxima letra tendo em conta as probabilidades:

- **std::random_device**: classe usada para gerar números aleatórios não-determinísticos. Usa fontes de entropia do sistema operacional para gerar números aleatórios mais seguros e imprevisíveis do que outros geradores.

- **std::mt19937**: classe que é um gerador de números pseudo-aleatórios. O construtor da classe recebe um objeto std::random_device como parâmetro para inicializar o gerador com uma seed aleatória.

- **std::discrete_distribution<>d(probabilities.begin(), probabilities.end())**: classe que cria uma distribuição discreta, usada para gerar valores aleatórios de acordo com uma distribuição de probabilidade específica.

Por último, tendo gerado a próxima letra, atualizamos as probabilidades e, adicionamos ao vector de sequências, no caso de ser uma nova sequência gerada. Continuamos a gerar letras até atingirmos o valor passado como argumento ou no caso de não ser passado nenhum valor, usamos o default (500).

Sempre que a sequência que estamos a querer gerar a próxima char, não possuir probabilidades para cada char (quando é uma sequência nova gerada), são usadas probabilidades iguais de gerar qualquer um dos caracteres.

5. Resultados

5.1 Resultados do Modelo de Cópia

Para fazer uma análise mais profunda aos nossos resultados, fizemos alguns testes para para vários valores de K, de número de bits, tempos de execução e valores para o threshold. Os resultados estão indicados na tabela abaixo, com um alpha constante (0.25).

K	threshold	número total de bits	Número médio de bits por caractere	Tempo de Execução (s)
3	1	12 039 490	188 117	284.384
3	2	16 936 976	498 146	286.712
3	3	19 824 226	566 406	576.906
3	4	21 914 914	755 687	668.772
3	5	23 445 456	901 748	1 120.15
3	6	24 661 762	986 470	1 131.69
4	1	11 929 505	46 599.6	1 102.26
4	2	16 814 472	113 611	1 035.09
4	3	19 645 968	178 600	860.253
4	4	21 774 724	259 223	871.168
4	5	23 245 904	309 945	934.325
4	6	24 491 070	318 066	932.489
5	1	11 716 111	11 441.5	1 354.35
5	2	16 400 453	26 886	1 184.39
5	3	19 102 394	39 224.6	1 112.45
5	4	21 092 422	54 928.2	1 106.25
5	5	22 550 544	67 315.1	1 085.75
5	6	23 753 250	83 344.7	1 051.42
6	1	11 520 625	2 812.65	3 434.68
6	2	16 021 671	6 905.89	2 403.93
6	3	18 611 610	1 0635.2	1 964.05
6	4	20 464 040	14 250.7	1 686.39

6	5	21 864 236	17 732.6	1 568.49
6	6	22 973 684	21 838.1	1 444.94

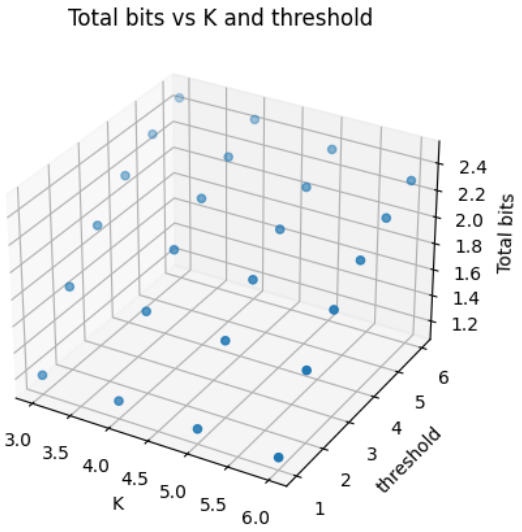

Fig. 2 - Comparação entre K, threshold, total bits (10^7)

Com este gráfico conseguimos visualizar de forma fácil, a relação entre estas 3 variáveis, denotando-se que quanto menor o valor da variável *threshold* menor o número total de bits, para valores baixos de threshold o K não é um fator muito determinante ficando a ser para valores maiores do threshold.

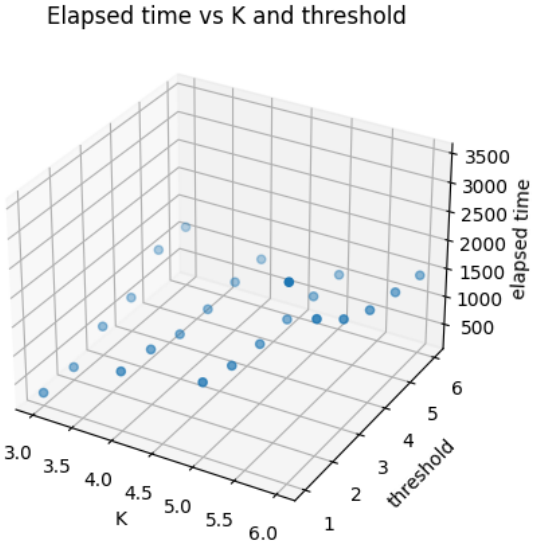

Fig.3 - Comparação entre K, threshold e tempo de execução

Após analisar o gráfico da figura 2, podemos concluir que o tempo de execução é proporcional ao aumento do k e do *threshold*, pois o modelo precisa de

considerar mais sequências e realizar mais iterações para encontrar correspondências adequadas. Ou seja, aumentar o tamanho da sequência que o modelo analisa a cada passo leva a um aumento do número de possíveis sequências antes de encontrar uma correspondência adequada e aumentar o valor do *threshold*, significa aumentar o número de "*fails*" permitidos antes da correspondência ser descartada, levando a um aumento de iterações.

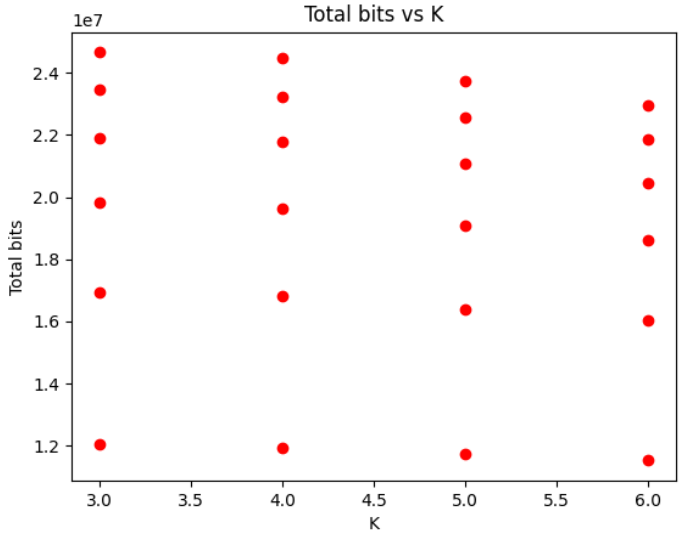

Fig.4 - Comparação entre o número total de bits e o valor de K

Neste gráfico, para cada valor de K, temos vários pontos que correspondem aos diferentes *thresholds*. Analisando a tendência à medida que o K aumenta, podemos facilmente inferir que o número total de bits diminui. Assim, podemos concluir que um maior valor de K nos permite comprimir mais informação em menos espaço.

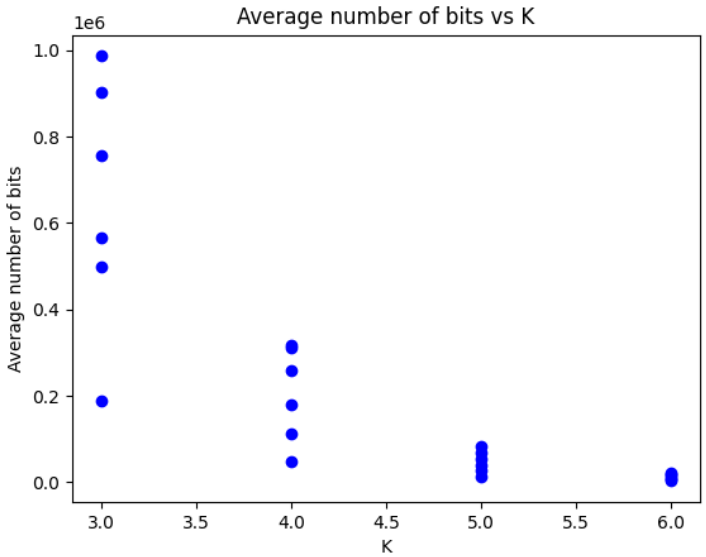

Fig.5 - Comparação entre o número médio de bits por símbolo e o valor de K

Este gráfico permite-nos analisar a evolução do número médio de bits por símbolo, em função de K. Tal como no gráfico anterior, para este gráfico observa-se a mesma tendência: com o aumento do valor de K, o número médio de bits por símbolo baixa. E essa descida é bem mais significativa, se a compararmos com a do gráfico anterior.

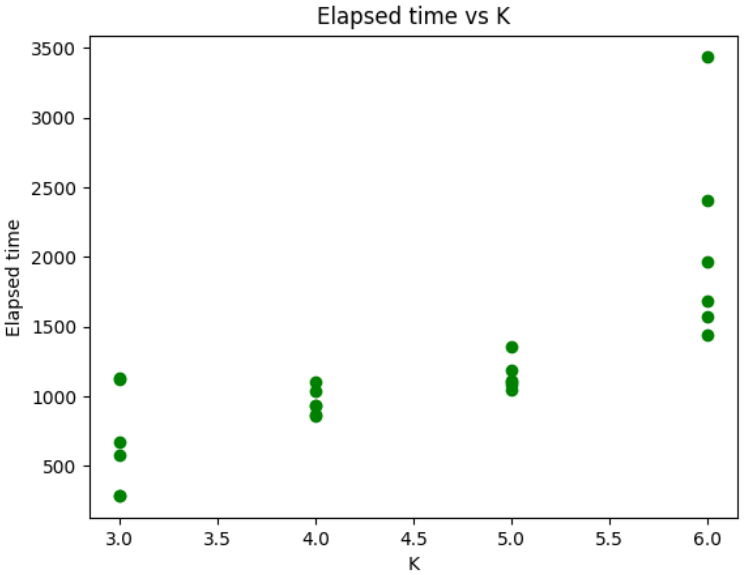

Fig.6 - Comparação entre o tempo de execução e o valor de K

Este gráfico ilustra a variação do tempo de execução com os diferentes valores de K. Ao analisar este gráfico, podemos observar de maneira simples que o tempo de execução aumenta substancialmente para um maior valor de K.

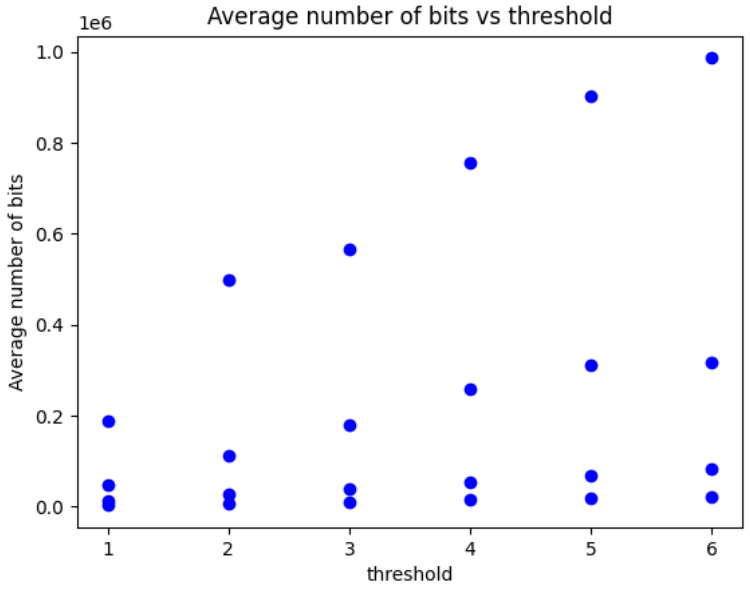

Fig.7 - Comparação entre o número médio de bits por símbolo e o valor de *threshold*

O objetivo deste gráfico é analisar a evolução do número médio de bits em função do valor do *threshold* (temos várias correspondências para cada valor do *threshold*, porque cada ponto corresponde também a um valor de K e, como já constatamos na análise da Figura 5, o número médio de bits por símbolo diminui com o aumento do valor de K). Pela análise do gráfico, é notório que com o aumento do *threshold*, o número médio de bits por símbolo também aumenta.

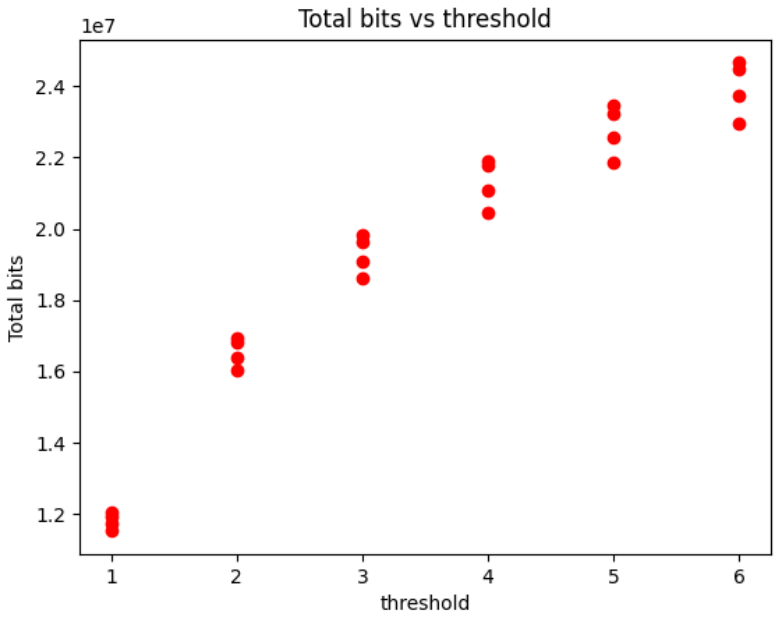

Fig.8 - Comparação entre o número total de bits e o valor de *threshold*

Este gráfico tem como intuito avaliar como o threshold interfere no número total de bits.

Analisando este último gráfico, concluímos que quanto maior for o valor do threshold, maior será no número de bits necessários para representar a sequência.

5.2 Resultados do Modelo de Geração de Texto

Relativamente aos resultados do modelo gerador, com o seguinte texto de entrada:

GAATTCTAGGCTTTCTTTGAAGAGGTAGTAATCTGTAGCCCTCACCTAGGA
CTACAAGGTCATTTTTTAAAAAATAGCTAAGAAAACACATGTCTGGCATGTTTATCT
CAGGCCATCGTTCTTGGCCTTCTAGAGAGTTAATGTCTACTATGTCACTTCATCA
GGGAGGGGTAGTTAAGCTTGAAAAATCTTTCTATGACATGACTGTGTCCTGCACA
TATTAAAAACTGGCCGAGTGAACACACCACCGACAGGCCATGTTTGGAGCCAGT
GTTTTTGCTGAAAGTCAGACAATTCTCCTTCCCCGTCGTGGAGGGCGGAGAAGA

TTCTATCTGGAGAAGGGTCCTCTGAAGCTCACATCTGGCATTTGGAATGAATAAT
CTCTTCAATGGCCAGGCAC

Obtivemos o seguinte **resultado**:

AATCTGGCATTCCTAAGAGTAGCTGGCCGACAGGCCATTTAAGGAGCCGA
GTCATGTTTTTGGAGCTGGCATTCCCGAGTTTGGAGTTCTATTCTGTTAAAAACA
TATGTGGCTAAATATCACCGTCCTCATTTGACATTTCTTTCTATTTGGCCAGTTATT
TCTATCTAAGAAGGGCCAGTCTTCTCATGTTTTAAGGCCATTTGGAGTCCTGTCT
AAAGGCCGACATCTATGTCTGTTTCTGCATTCTGGCCATTTGGCCGAGCCACATT
TTGCATGAATATTTCACATTTGGAGAAAATGTCCTCTATGTGTGTTCAGGTAAGAT
TCACATTTCTAGAGCCAGGTAGTTTGGCCGAGTTTTCCTCACATTCTTGAACACA
ATATGAAAAACATTCTGGAGTCCTAAGGCCGAGTTATGAAGATTCTGGCCGTCTC
ATGTGTTTTCTAGTTATTCTATGTTATCGTCCTCTATGACATTCTATCTCTGGCCAT
GAAGAG

Embora o modelo não gere um resultado 100% correto, dá para ver algumas semelhanças e dá para confirmar a sua consistência.

Com o seguinte texto de entrada do famoso escritor português José Saramago:

Algumas vezes este romancista, confundido nas malhas da ficção que ia tecendo, chegou a imaginar-se transportado na fantástica jangada de pedra em que transformara a Península Ibérica, flutuando sobre o mar atlântico, a caminho do Sul e da utopia. A peculiaridade da alegoria era transparente: embora prolongando algumas semelhanças com os motivos do mais comum dos emigrantes, que parte para outras terras e busca a vida, prevalecia, neste caso, uma diferença assaz substancial, a de também comigo viajarem, em tão inaudita migração, o meu próprio País, todo ele, e, sem que aos espanhóis tivesse pedido antes a devida licença, portanto sem procuração nem autorização, a Espanha.

Obtivemos o seguinte **resultado**:

A prolonfundo, que pedra Ibérizaís, a a vido, ficença, que tão que ca via ia, umale, ca co, em da vido, tra fantorizaínsul em pedras te: em te o atlânterevido meu próis audido, em dado a imas tra audido meu ale, em ca asobrem, terevial, ficomigrangado nem da País, comumasobrenís, tão se chegormale, tranterra jarevida jansforta, que tra a do meu atlântão, tras tra audida vida te rolonfundo, tras em da traça Paça ficença, que de comigo atlânte rocuransforida te cominhóis mar-sem t

Aqui conseguimos ter uma perceção mais óbvia do que o modelo consegue gerar corretamente, sendo visível que palavras mais pequenas como “que”, “de”, “comigo”, o modelo gera corretamente.

6. Conclusão

Em conclusão, o desenvolvimento e implementação dos modelos de cópia e de geração de texto apresentados neste relatório foram bem sucedidos na solução do problema proposto de compressão de dados, sendo ambos baseados na mesma abordagem.

Foram utilizadas estruturas de dados eficientes e algoritmos apropriados para a implementação dos modelos, como uma estrutura de dados nativa do C++, *unordered_map*, que implementa uma hash table, para armazenar as sequências de caracteres e os índices correspondentes no ficheiro.

O modelo de cópia usa essa estrutura para prever o próximo caractere com base nas sequências anteriores. O modelo de geração de texto usa a mesma estrutura para armazenar as contagens de cada caractere que aparece depois de uma determinada sequência de caracteres.

Ambos os modelos foram desenvolvidos com a possibilidade de ajustar vários parâmetros, incluindo o tamanho da sequência (K), um valor limiar (*threshold*) e um parâmetro de suavização (alpha). Os resultados mostraram que a mudança desses parâmetros afeta significativamente o desempenho do modelo.

O modelo de cópia mostrou-se capaz de prever o próximo caractere com base nos padrões de caracteres lidos anteriormente, enquanto o modelo de geração de texto foi capaz de gerar textos coerentes seguindo o padrão do texto de entrada.