AI Generated vs Human Written Text

Mestrado em Engenharia Informática

Universidade de Aveiro

DETI - Aveiro, Portugal

TAI - Teoria Algorítmica da Informação

Guilherme Antunes [103600], Gonçalo Silva [103668], Pedro Rasinhas [103541]

v2024-05-08

Índice

1 Introdução

O recente aparecimento em massa de novas ferramentas de geração de texto por meio de inteligência artificial trouxe consigo desafios e oportunidades significativas quer para a maneira como procuramos por informação, quer para a maneira como a escrevemos. Um desses desafios prende-se quanto à questão da autoria dos textos, particularmente no contexto de modelos de linguagem avançados como o ChatGPT. Distinguir texto escrito por humanos e texto escrito por modelos de linguagem está a revelar-se uma tarefa crucial em diversos campos, incluindo a deteção de plágio e a moderação de conteúdo.

As abordagens mais comuns para resolver o problema de classificação mencionado acima envolvem operações de extração e seleção de características a partir dos dados de treino. O desafio consiste em escolher o menor conjunto possível de características que tenham capacidade discriminatória suficiente para permitir a resolução do problema.

No contexto da Unidade Curricular de Teoria Algorítmica da Informação foi-nos proposto para este projeto a resolução do problema de classificação de texto humano e não humano tendo por base a utilização expressa da compressão de dados para solucionar o problema sem ter necessidade de recorrer aos mecanismos de extração e seleção de características. Assim, propomo-nos a implementar um programa que seja capaz de efetuar a distinção entre texto escrito por mão humana e texto gerado por ChatGPT e outros modelos de linguagem do género e a estudar de que maneira varia a precisão da classificação mediante a variação de alguns parâmetros e características.

Como sugerido no enunciado do projeto, iremos utilizar modelos de contexto finito, mais especificamente cadeias de Markov, para lidar com os padrões existentes nos dados de treino e contabilizar a frequência de cada caractere após a ocorrência de cada padrão.

2 Metodologia

O projeto é composto pelo programa principal was_chatted.cpp e por outros 2 programas de apoio, benchmark.cpp e dataset.py. Enquanto o was_chatted.cpp implementa o objeto principal do projeto, o benchmark.cpp implementa uma variação do *was_chatted* dedicada à avaliação da precisão do programa e dos tempos que cada operação leva. Por sua vez o dataset.py é responsável por, partindo de um *dataset* maior, separar os exemplos das diferentes classes, selecionar uma parte para *dataset* de teste e normalizar os exemplos para cada uma das classes.

2.1 Organização

Relativamente à organização dos ficheiros do projeto podemos encontrar na pasta ***src/*** todos os ficheiros relacionados com código desenvolvido. É nesta pasta que encontramos o código fonte do ***was_chatted.cpp***, do ***benchmark.cpp***, e das *hashtables* utilizadas e testadas no projeto. Na pasta ***dataset/*** está presente todo o material necessário para gerar o *dataset*, incluindo o dataset.py, e o próprio *dataset* gerado e na pasta ***report*** podemos encontrar este relatório que descreve o trabalho desenvolvido. Também está disponível um ficheiro ***README.md*** que descreve como correr os programas, o ***build.sh*** que permite compilar todo o projeto e gerar o *dataset*, e a pasta ***bin/*** onde estão os executáveis do projeto.

```
group_2/
├── bin
│   ├── benchmark
│   └── was_chatted
├── build.sh
├── dataset
│   ├── AI_Human.csv
│   ├── dataset.py
│   ├── dataset_url.txt
│   ├── test
│   │   ├── test_ai_0.txt
│   │   ├── test_human_0.txt
│   │   └── ...
│   ├── test_tiny
│   │   ├── test_ai_0.txt
│   │   ├── test_human_0.txt
│   │   └── ...
│   └── train
│       ├── train_ai.txt
│       └── train_human.txt
├── README.md
├── report
│   └── TAI_report2.pdf
└── src
    ├── benchmark.cpp
    ├── flat_hash_map.hpp
    ├── progress_bar.cpp
    ├── progress_bar.hpp
    ├── table.cpp
    ├── table.hpp
    └── was_chatted.cpp
```

2.2 Argumentos da Linha de Comandos (was_chatted)

Nome	Flag	Descrição	Type
Alpha	-a	Parametro de “smoothing” para o cálculo de probabilidades	Double
Modo Iterativo	-i	Define se o programa deve iniciar em modo iterativo	Bool
Directory	-d	Define se o programa deve iniciar em modo de avaliação de um diretório e o caminho até ele	String
Ficheiro	-f	Define se o programa deve iniciar em modo de avaliação individual de um ficheiro e o caminho até ele	String
Human Collection	-h	Define o caminho até à coleção de textos escritos por humanos	String
ChatGPT Collection	-c	Define o caminho até à coleção de textos gerados por modelos de linguagem	String
Tamanho do contexto finito	-n	Tamanho dos padrões a armazenar no modelo de contexto finito	Positive Integer

3 Dataset

Para alimentar o modelo procurámos *datasets* online que cumprissem os requisitos de ter um volume considerável, na ordem dos megabytes, e que apresentasse dados relativamente relevantes para o problema. Também procurámos textos em inglês, para podermos limitar o modelo aos caracteres ASCII, e que fossem relativamente recentes, uma vez que a escrita antiga já apresenta diferenças relativamente à mais recente.

Cumprindo estes requisitos optámos por um *dataset* com cerca de 1.1GB de dados para ambas as classes com cerca de 60% desses dados relativos à classe Humano e os outros 40% à classe AI/ChatGPT.

A fim de normalizar os dados limitamos o número de casos da classe com maioria ao mesmo número de casos da classe minoritária, retiramos aleatoriamente 10% dos elementos da classe minoritária para servir de conjunto de teste dessa classe, damos *shuffle* aos textos da classe maioritária e retiramos a quantidade necessária de ficheiros para igualar o melhor possível a quantidade de informação presente em ambas as classes, recorrendo ao tamanho em bytes que cada uma das classes ocupa no conjunto de treino. Por fim, entre os ficheiros que não foram selecionados para o conjunto de treino da classe maioritária selecionamos uma quantidade equivalente a 10% do tamanho do conjunto minoritário para igualar o número de casos no conjunto de testes de ambas as classes.

A partir do conjunto de teste, selecionamos 10% dele para construir outro conjunto de teste mais pequeno que é mais útil para testar modelos que ocupem mais memória. Uma vez que o *dataset* é bastante extenso estes 10% da classe de teste continuaram a ser uma boa amostra para testar a precisão do modelo.

Estão também implementadas, através de macros, variações no tamanho do conjunto de treino e no tamanho dos ficheiros do conjunto de teste para possibilitar um estudo mais variado das capacidades do modelo.

Na construção do conjunto de treino os diferentes ficheiros são separados pelo caractere *'\0'* para que o *was_chatted* possa saber que quando encontra este caractere deve "resetar" todo o padrão que já trazia guardado do ficheiro anterior,

uma vez que na realidade nunca foi escrito o texto de um ficheiro antes do texto do ficheiro seguinte, evitando assim a consideração indevida de padrões.

4 Implementação

Como já foi referido anteriormente o objetivo deste programa *was_chatted.cpp* é conseguir classificar textos como humanos ou gerados por inteligência artificial utilizando a compressão de dados como métrica única para efetuar essa distinção. Utilizando modelos de contexto finito conseguimos determinar a probabilidade condicionada de uma letra ocorrer após um determinado padrão. Assim criamos 2 tabelas, uma para os padrões humanos e outra para os padrões gerados por modelos. Utilizando estas tabelas determinamos a quantidade de informação que, utilizando uma de cada vez, irá minimizar a quantidade de bits necessária para comprimir o ficheiro que queremos determinar se é escrito por humanos. A compressão que ocupar menor espaço em memória determinará o veredito do modelo, caso a compressão menor seja a que utilizou a tabela de padrões humanos considera-se que o texto é humano, caso seja a tabela de padrões gerados por modelos de linguagem considera-se que o texto é não humano.

4.1 Tabelas

Cada tabela tem a função de implementar um modelo de contexto finito Markov. Para isso utilizamos estruturas do tipo *hash_map* em que a chave de cada par corresponde ao padrão verificado. Em ambas as tabelas todos os padrões terão o mesmo tamanho, introduzido via terminal no parâmetro -n, por exemplo, se tivermos um alfabeto que aceita qualquer caractere ASCII e definirmos um -n 4 então um padrão que pode ser considerado e guardado como chave é o padrão “the “ ou o padrão “Ever”.

Por sua vez, cada valor associado a uma chave será uma nova estrutura *hash_map* constituída por uma chave do tipo *char* e um valor do tipo *unsigned int*. Esta nova estrutura será responsável por armazenar os caracteres que ocorrem

após o padrão da chave anterior e por armazenar a quantidade de vezes que o caractere ocorre após esse padrão.

Cada tabela, correspondente à *struct Table*, tem também uma função *memorySize()* que permite monitorar o tamanho que a estrutura está a ocupar em memória no momento.

4.2 Cálculo de Probabilidades, Quantidade de Memória e Precisão

Para determinar a classificação do(s) ficheiro(s) que inserimos na linha de comandos calculamos a probabilidade condicionada do caractere que estamos a ler ocorrer no tipo de escrita representado pela tabela que estamos a utilizar. Essa probabilidade é dada pela fórmula da Fig. 1.

$$P(y|x) \approx (N(y|x) + \alpha) \; / \sum N(@|x) + \alpha|\sum|$$

Figura 1 - Fórmula de cálculo da probabilidade condicionada de y dado x

Dado pela fórmula da Fig. 1, em que x representa o padrão verificado antes do da ocorrência do caractere y, @ representa todos os caracteres que ocorrem depois de x, α é o parâmetro de suavização e $|\sum|$ o tamanho do alfabeto, o valor resultante desta operação corresponde à probabilidade do símbolo x ocorrer após o padrão y.

Após sabermos a probabilidade de uma letra y aparecer depois de uma sequência x interessa-nos saber a quantidade de memória que nos irá custar comprimir esse símbolo. Para isso aplicamos o logaritmo de base 2 à probabilidade calculada, definindo assim que a quantidade de memória necessária para comprimir todo o ficheiro se pode definir como o somatório negativo dos logaritmos de base 2 das probabilidades condicionadas dos caracteres presentes no ficheiro. Esta fórmula pode encontra-se na Figura 2.

$$Memória \approx -\sum log2(P(y|x))$$

Figura 2 - Fórmula de cálculo da quantidade de memória necessária para comprimir um ficheiro

Para calcularmos a precisão do modelo implementado utilizamos o programa *benchmark* que recebe um diretório cujos ficheiros nele contidos serão todos utilizados para avaliar a precisão do modelo. Quando um ficheiro gerado por inteligência artificial é detetado como tal incrementamos o contador de *hits*, caso contrário incrementamos o contador de *misses*, o mesmo para os casos em que um documento escrito por uma pessoa é detetado como tal ou não. Posto isto, o cálculo da precisão é dado pela fórmula da Figura 3.

$$Precisão \approx Hits/(Hits + Misses)$$

Figura 3 - Fórmula de cálculo da quantidade da precisão do modelo

4.3 Escolhas e Detalhes de Implementação

Uma vez que estamos a trabalhar com dados que facilmente escalam em termos de memória ocupada é importante manter o mínimo de memória ocupada e maximizar a utilidade daquela que não temos como poupar. Para isso vários tipos de dados foram escolhidos tendo em vista a melhor gestão da memória. São casos o buffer de 20 MB implementado para ler os ficheiros de treino, que apesar de ser mais rápido ler todo o ficheiro para memória podemos estar a lidar com documentos na ordem dos gigabytes, que lidos inteiramente ao mesmo tempo para memória podem causar uma má operação do computador que corre o programa; a utilização de *size_t* em detrimento da classe *string* e o *hashing* dos padrões que permite uma redução de memória em 4x, uma vez que uma *string* ocupa 32 bytes e *size_t* apenas ocupa 8 bytes e permite armazenar os padrões em formato de *hash* abdicando em troca uma quantidade de tempo praticamente impercetível para fazer essa conversão; a utilização de *uint* ao invés de *int* que, apesar de não representar um ganho de memória, permite alcançar quantidades maiores abdicando apenas da possibilidade de representar um número negativo que a classe *int* tem mas que para nós não tem qualquer interesse uma vez que não existem contagens negativas; a já referida normalização dos *datasets* de treino feita ao nível dos bytes; a também já referida utilização do caractere *'\0'* para sinalizar que o padrão deve levar reset impedindo

assim a contabilização de padrões que nunca ocorreram; o ignorar de todos os caracteres que não sejam antecedidos por um padrão de tamanho n e a utilização da flat_hash_map em detrimento de qualquer outra opção, justificada pela comparação presente nos resultados.

5 Outras funcionalidades

Para tornar o programa mais útil, eficiente e mais amigo do utilizador comum adicionámos, para além da análise de um ficheiro único, a possibilidade de analisar um diretório completo utilizando a *flag* -d para especificar o caminho e retornando 0's e 1's mediante se o ficheiro impresso é gerado por ChatGPT ou não e também adicionámos a possibilidade de enviar ficheiros 1 a 1 de modo interativo. Estas funções irão permitir ao utilizador aproveitar todos os dados de treino para mais do que 1 ficheiro, sem ter de treinar o modelo todo de novo cada vez que quer saber se um documento foi ou não escrito por humanos.

6 Resultados

6.1 Hash Map

Durante uma primeira implementação do *was_chatted.cpp* estava a ser utilizada como *hash_map* a classe *unordered_map*, comumente utilizada em programas *C++*. No entanto com o aumento da quantidade de informação que está inerente ao aumento do tamanho dos contextos a guardar surgiu a necessidade de procurar soluções mais eficientes, quer em termos de velocidade quer em termos de memória. Para isso foi efetuada uma pesquisa de quais são os *hash_maps* mais eficientes conhecidos pela comunidade online tendo chegado a mais 3 implementações para além da *unordered_map*:

- Google Sparse Hash Map
- Skarupke Flat Hash Map
- TSL Hopscotch Map

Para averiguar qual delas seria mais útil para o nosso caso de uso colocámos as 3 à prova e tirámos tempos relativamente ao tempo que levavam a inserir toda a informação relativa a contextos de ordem 2 para *datasets* de treino cerca de 700 MB no total, o tempo que levavam a analisar todos os ficheiros de teste e a aceder aos dados presentes na hash_map e analisámos também a quantidade de memória consumida por cada uma.

	unordered_map	flat_hash_map	sparse_hash_map	hopscotch_map
Tempo de Inserção (ms)	47002	20069	47142	18603
Tempo de Leitura (ms)	60488	35011	500246	29386
Memory Size (KB)	916	597	1343	1448

Tabela 1 - Desempenho das estruturas de *hash*

Como é possível observar pela Tabela 1 a *flat_hash_map* e a *hopscotch_map* foram as que obtiveram melhores resultados temporais, tendo ficado ambas perto uma da outra nesse aspeto. No entanto verificamos também que a *hopscotch_map* ocupa muito mais memória do que qualquer uma das outras.

Para tentar perceber se o gasto extra de memória valia a pena, isto é, se de facto a velocidade da *hopscotch_map* justificava o dispender de tanta memória decidimos testar ambas as *hash_maps* para contextos de ordem maior.

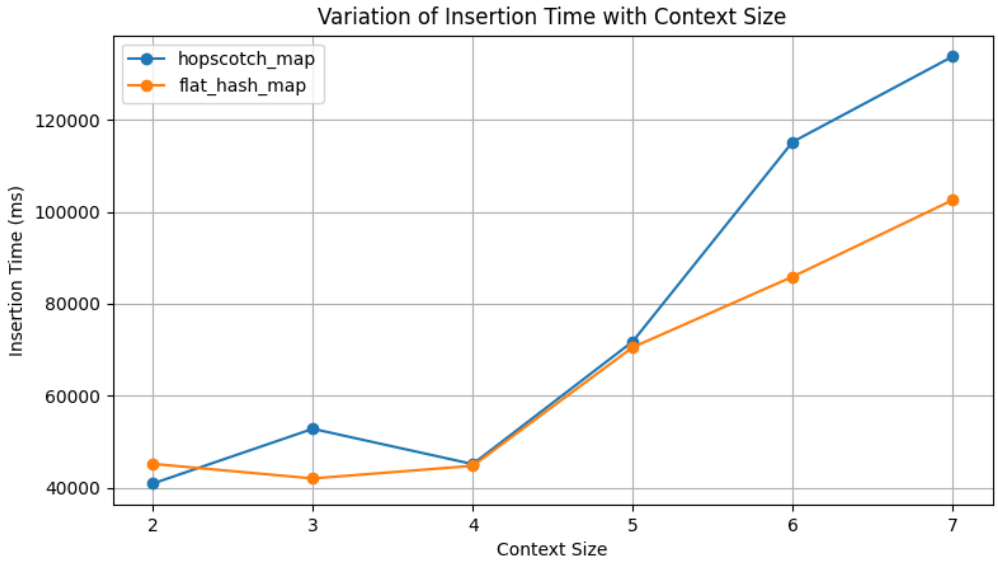

Figura 4 - Variação do tempo de inserção em função do aumento do tamanho do contexto para diferentes hash_maps

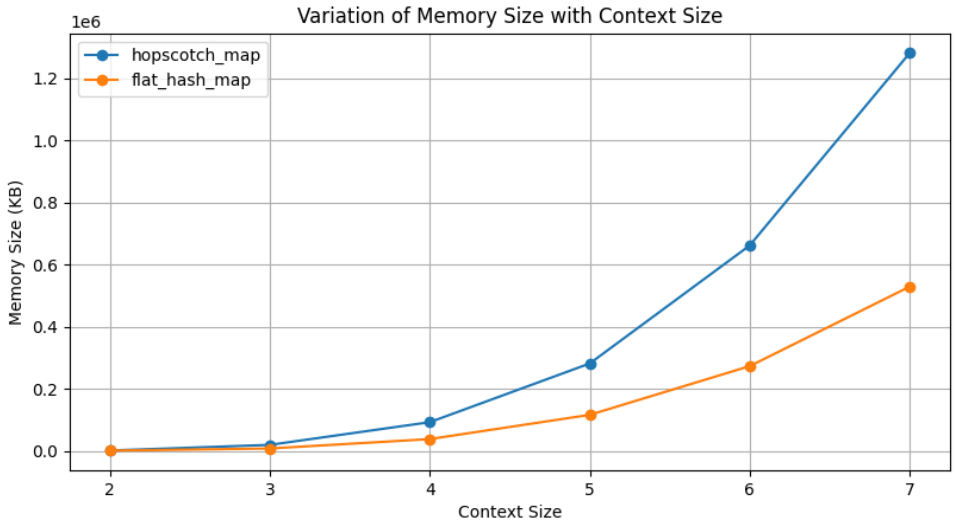

Figura 5 - Variação da memória ocupada em função do aumento do tamanho do contexto para diferentes hash_maps

Como é possível observar pela Fig. 5 a *flat_hash_map* usa consistentemente menos memória que a *hopscotch_map*, e pela Fig. 4 conseguimos observar que a partir de contextos de ordem 6 a *flat_hash_map* é significativamente mais rápida em termos de operações de inserção, o que levou a que a nossa escolha recaísse sobre esta uma vez que ocupa menos memória e é mais rápida que as concorrentes. Acabámos por ignorar o tempo de leitura uma vez que este era praticamente insignificante quando falamos em ficheiros isolados. Os testes de velocidade de leitura feitos na Tabela 1 foram efetuados para cerca de 30 mil ficheiros.

6.2 Precisão do Modelo

A fim de avaliar a precisão e utilidade do modelo implementado foram executados vários testes em diferentes condições ao modelo.

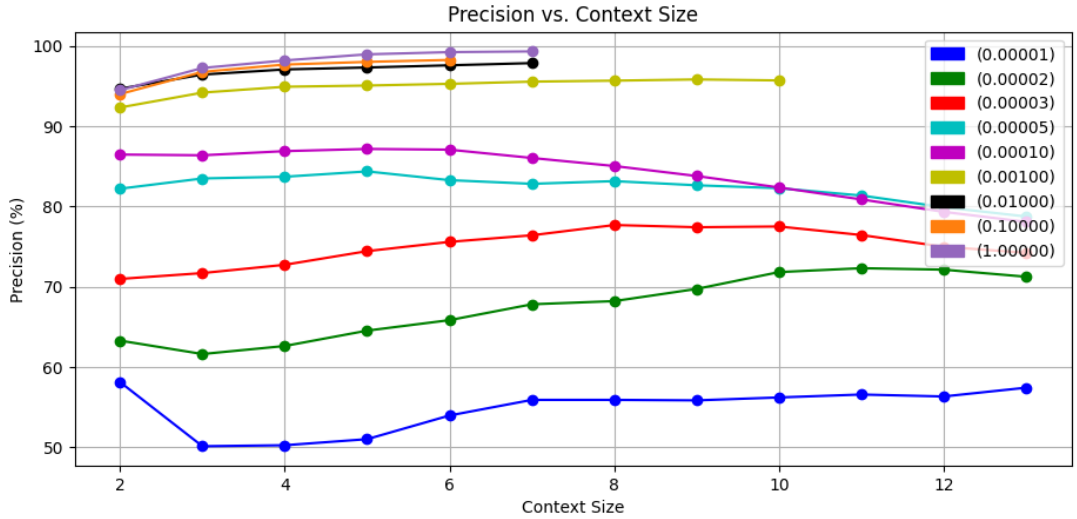

Figura 6 - Variação da precisão do modelo em função do aumento do tamanho do contexto para diferentes datasets de treino

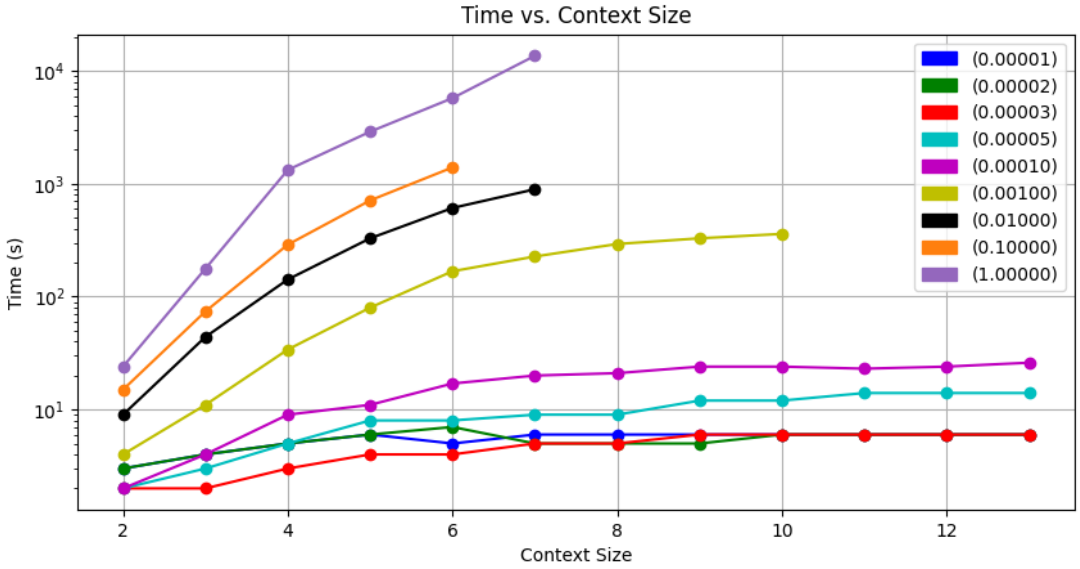

Figura 7 - Variação do tempo de leitura em função do aumento do tamanho do contexto para diferentes datasets de treino

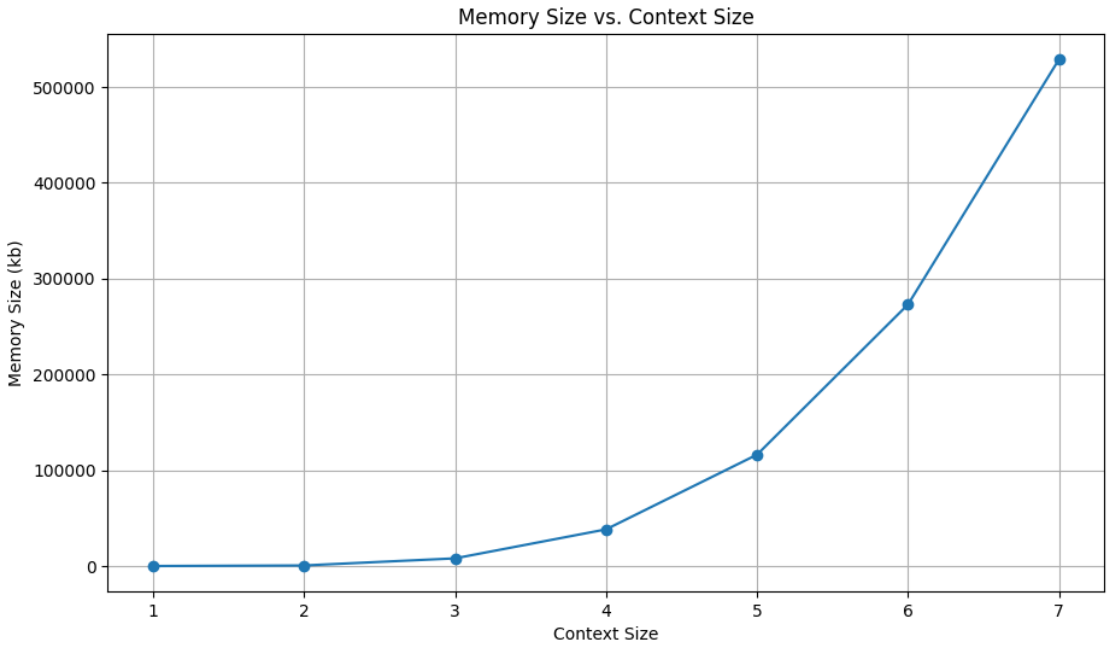

Figura 8 - Variação da memória ocupada em função do aumento do tamanho do contexto para o dataset de treino completo

Na figura 6 podemos verificar que a maior precisão alcançada foi para modelos de contexto de ordem 7, alcançando uma precisão de cerca de 99.3% de precisão, tendo sido essa precisão sobre cerca de 3 mil ficheiros de teste com uma média de 2KB de informação por ficheiro. Possível e expectavelmente o modelo alcançaria valores de precisão mais altos para ordens superiores, no entanto como é visível na Figura 7, o tempo de leitura associado à testagem do conjunto de teste para ordem 7 já foi extremamente elevado inviabilizando um teste semelhante para ordens superiores.

Relativamente aos outros modelos presentes nas figuras apresentadas podemos verificar que os modelos que utilizaram menos informação para proceder ao treino, nomeadamente aqueles que apenas usaram 0.00001, 0.00002 e 0.00003 do dataset de 700MB, verificam um aumento da precisão com o aumento da ordem do modelo de contexto à semelhança dos modelos que usam mais informação. Isto deve-se ao facto de, no caso dos modelos bem treinados, mais informação recolhida irá naturalmente manifestar-se numa melhoria dos resultados de classificação. No caso dos modelos com menos treino, teorizamos que o aumento da precisão se deve ao facto de a precisão já ser tão baixa para contextos de baixa ordem que é difícil fazer pior em ordens superiores. Já os modelos que perdem precisão mais cedo e

que se encontram numa posição mais intermédia quanto à quantidade de treino recebido acabam por revelar que de facto o treino não foi suficiente e que aumentar o tamanho dos contextos irá mostrar que os dados de treino não são suficientes para garantir um aumento da precisão, esperando-se assim que quantos mais dados de treino maiores ordens de contextos os modelos conseguiram avaliar com maior precisão.

Na figura 8 podemos observar o crescimento aparentemente exponencial da memória ocupada pelo treino do dataset completo.

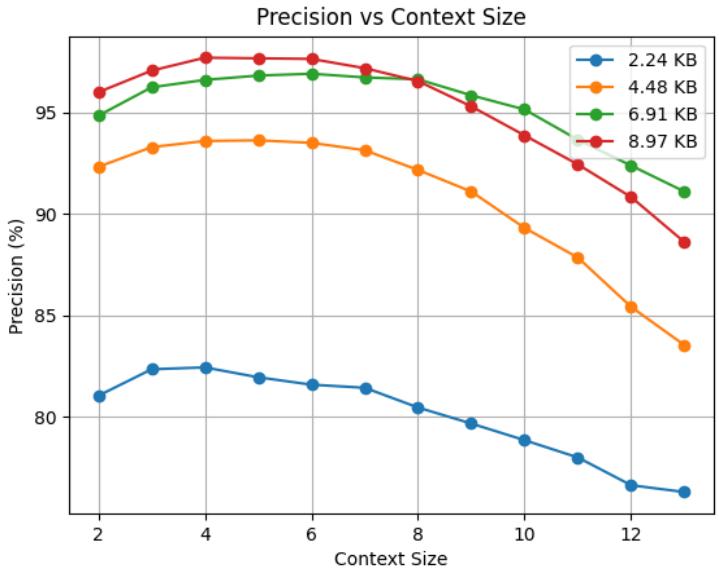

Figura 9 - Variação da precisão em função do aumento do tamanho do contexto para tamanhos de ficheiros de teste diferentes e dataset 0.00001x o tamanho total

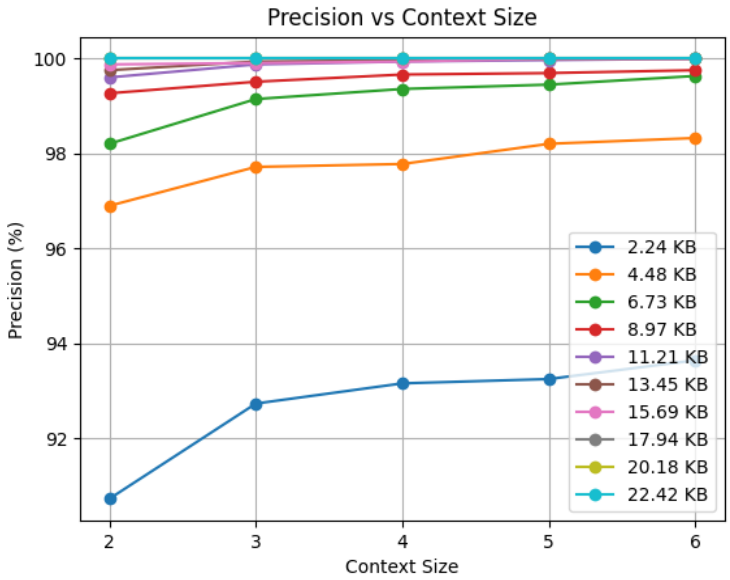

Figura 10 - Variação da precisão em função do aumento do tamanho do contexto para tamanhos de ficheiros de teste diferentes e dataset 0.001x o tamanho total

Estudámos também a influência do tamanho dos ficheiros presentes no conjunto de teste na precisão das classificações. Na figura 9 observamos um decréscimo acentuado da precisão do modelo com o aumento do tamanho dos ficheiros de teste. Mais uma vez observamos que 0.00001 do dataset total não são suficientes para continuar a aumentar a capacidade de classificação para ordens cada vez maiores, apesar de continuarmos com uma percentagem bastante boa.

Já no modelo que utiliza 0.001 do tamanho total do dataset de teste podemos observar que consegues precisões de 100% para tamanhos de ficheiros por volta dos 22.42 KB em todas as ordens de grandeza testadas, o que confirma que num modelo bem treinado um texto mais extenso permite dar mais certezas no resultado da classificação.

7 Conclusão

Os resultados obtidos foram no geral bastante agradáveis uma vez que foram alcançadas consistentemente precisões bastante elevadas para as várias situações a que o modelo foi sujeito. Seria interessante colocar o programa à prova num contexto de mundo real para perceber se a sua eficácia também se aplica a qualquer outro ficheiro puramente escrito por mão humana ou apenas gerado por ChatGPT ou se de alguma maneira existiu algum enviesamento dos dados presentes no *dataset* de treino que não foi detetado até então e que levou a que estes fossem os resultados obtidos. Seria também interessante testar uma implementação semelhante, mas a um nível mais específico como uma frase ou um parágrafo.

Os resultados acabam por permitir-nos concluir que quanto mais contexto é dado a um modelo maior capacidade ele terá de classificar corretamente os testes que lhe são propostos. Concluímos também que são necessários mais dados de treino se queremos classificar ordens maiores e que ficheiros maiores nos dão em geral maiores certezas quanto à natureza do texto contido em cada um.

8 Referências

[1] A. J. Pinho e D. Pratas, “elearning@UA,” [Online]. Available: https://elearning.ua.pt/pluginfile.php/350126/mod_resource/content/8/trab2.pdf. [Acedido em Maio 2024].

[2] “StackExchange,” [Online]. Available: https://softwarerecs.stackexchange.com/questions/49853/high-performance-c-hash-table-implementation. [Acedido em Maio 2024].

[3] “GitHub,” [Online]. Available: https://github.com/skarupke/flat_hash_map. [Acedido em Maio 2024].

[4] “GitHub,” [Online]. Available: https://github.com/Tessil/hopscotch-map. [Acedido em Maio 2024].

[5] “GitHub,” [Online]. Available: https://github.com/Tessil/hopscotch-map. [Acedido em Maio 2024].

[6] “Kaggle,” [Online]. Available: https://www.kaggle.com/datasets/shanegerami/ai-vs-human-text/data. [Acedido em Maio 2024].