Classificação de Textos Baseada em Algoritmo de Compressão LZW

Carlos Vinícius Costa Neves (20180028631)
Rafael Sobral de Morais (20180033515)
Yure Talis Rocha Silva (20170022456)

30/11/2022

1 Introdução

A classificação é um dos problemas mais básicos e conhecidos na área de Aprendizado de Máquina, e visa determinar a qual classe (ou categoria) um indivíduo desconhecido pertence. Em geral, os algoritmos de classificação fazem uso de bases de dados compostas por vários indivíduos previamente rotulados (i.e., cuja classe à qual pertencem é conhecida). Em posse dessa base de dados, efetua-se uma etapa de "treinamento", na qual características relevantes sobre cada categoria são "aprendidas". Dessa forma, ao se deparar com um indíviduo novo, a tarefa de classificá-lo pode ser facilitada fazendo-se uso do conhecimento adquirido durante o treinamento.

É comum que os indivíduos sujeitos a problemas de classificação possam, de forma muito intuitiva, ser representados por vetores de n dimensões. Muitas vezes, classificar tais vetores se resume a observar sua localização no $\mathbb{R}^n$ e compará-la com a de outros indivíduos através, por exemplo, de métricas de proximidade. Contudo, indivíduos em forma de sequências de símbolos de comprimento arbitrário (e.g., textos, imagens, arquivos de áudio) impõem um desafio adicional ao problema de classificação: como identificar, dentro de tais sequências, padrões recorrentes e relevantes de forma a distinguir indivíduos corretamente?

Vários métodos foram elaborados para lidar com o problema de classificar sequências de caracteres. Este trabalho apresenta, especificamente, uma maneira de utilizar técnicas oriundas da Teoria da Informação para auxiliar algoritmos de classificação. Mais especificamente, este trabalho apresenta uma aplicação do algoritmo LZW [4, 5] à resolução de problemas de classificação. Em suma, o método usa o dicionário gerado pelo LZW como um modelo que contém características recorrentes em uma classe de indivíduos. Utilizando-se o mesmo modelo na compressão de um novo indivíduo e observando-se métricas adequadas (e.g., o tamanho do arquivo comprimido, o número de índices do dicionário que foram

utilizados na compressão), pode-se obter indicadores úteis na classificação dos indivíduos.

A implementação do algoritmo LZW já foi descrita em um relatório anterior. Portanto, as próximas seções do trabalho focam diretamente na aplicação do algoritmo LZW como método de classificação. A base de dados utilizada, bem como os testes feitos para aferir a eficácia do método, também serão explicados.

2 Metodologia

Para aplicar o método, uma base de dados foi aleatoriamente particionada em conjuntos de treinamento e de testes. O algoritmo LZW é então utilizado tanto para gerar os modelos de classificação quanto para realizar a etapa de classificação em si. Cada um dos principais componentes do método (e.g., a base de dados usada, as modificações feitas ao LZW) será explicado nas seções seguintes.

2.1 Base de dados

A base de dados utilizada foi proposta em [6], e consiste em um conjunto de obras literárias clássicas. As obras foram separadas por autor, cada um dos quais representa uma classe. Cada obra está contida em um arquivo de texto (`.txt`) próprio. O título de cada um dos arquivos é composto por dois números, o primeiro dos quais representa o autor, e o segundo representa o índice da obra. Os arquivos `8_1.txt` e `8_2.txt`, por exemplo, ambos pertencem à classe 8, sendo de autoria do autor Mark Twain, e contém as obras *Adventures of Huckleberry Finn* e *A Connecticut Yankee in King Arthur's Court*, respectivamente.

Dado um livro que pertence ao conjunto de testes — que nada mais é que um subconjunto da base de dados —, o objetivo é determinar sua autoria com base em seu conteúdo.

2.2 Método de Classificação

Seja $C = \{a_1, a_2, \ldots, a_m\}$ uma classe composta por m arquivos que serão usados para o treinamento. Cada arquivo a_i pode ser pensado como uma sequência de caracteres. A etapa de treinamento se dá pela construção de um novo arquivo $c = a_1 a_2 \ldots a_m$, obtido concatenando-se todos os arquivos em C de forma sequencial. Após isso, o arquivo c é comprimido através do algoritmo LZW, e o dicionario D_c obtido ao fim da compressão é armazenado. O procedimento é feito com todas as classes e seus respectivos arquivos de treino.

Suponha, agora, que deseja-se determinar a classe de um arquivo b que pertence ao conjunto de testes. Novamente, o algoritmo LZW é usado para comprimir o arquivo b. Desta vez, no entanto, a compressão é limitada a usar dicionário D_c de maneira estática. O mesmo arquivo é comprimido utilizando-se o dicionário

de cada uma das classes. O arquivo b é então atribuído ao dicionário que proporcionou a maior razão de compressão.

Outra métrica que também foi usada é o número de índices em D_c que foram de fato utilizados durante a compressão. Neste caso, o arquivo b é atribuído à classe cujo dicionário teve o maior número de índices utilizados.

2.3 Modificações feitas no LZW

O código utilizado para implementar o algoritmo LZW manteve-se, em sua maioria, inalterado com relação ao trabalho anterior. A modificação mais importante foi a adição dos métodos `SaveModel()` e `LoadModel()`. O primeiro método é responsável por armazenar em um arquivo o dicionário (ou modelo) resultante de uma compressão. Os dicionários são armazenados na forma de um arquivo binário como ilustrado no trecho de código a seguir, no qual cada índice e seu respectivo código são armazenados. O segundo método, por sua vez, lê um dicionário armazenado em um arquivo para posterior uso de forma estática durante uma compressão.

```
1  for (auto it = dict.begin(); it != dict.end(); ++it)
2  {
3    // Armazena o índice do código
4    model_file.write((char*) &(it->second), sizeof(int));
5
6    // Armazena o tamanho do código e, finalmente, o código
7    int str_length = it->first.size();
8    model_file.write((char*) &(str_length), sizeof(int));
9    model_file.write((char*) it->first.data(), it->first.size());
10 }
```

Além disso, já que nesta aplicação não é necessário armazenar o resultado da compressão, optou-se por desativar a escrita de arquivos, que foi substuída por uma variável que conta quantos *bytes* foram escritos, conforme ilustrado no trecho de código a seguir.

```
1  // output_file.write((const char*) &byte_to_write, sizeof(char));
2  bytes_written++;
```

Por fim, um *container* do tipo `set`, denominado `used_codewords`, foi utilizado para realizar a contagem de índices usados durante a compressão. Sempre que um código do dicionário é usado, ele é inserido em `used_codewords`. Dessa forma, as duas métricas para classificação são exibidas ao fim da compressão como ilustrado a seguir.

```
1  cout << "Compressed file size: " << bytes_written << endl;
2  cout << "Used codewords: " << used_codewords.size() << endl;
```

3 Resultados e discussão

O algoritmo LZW, bem como a base de dados e o método classificador podem ser encontrados no seguinte *link*: `https://github.com/cvneves/ITI`. O algoritmo LZW foi implementado na linguagem C++ e compilado através da ferramenta *CMake*. Nenhuma biblioteca ou código adicional foi utilizada para escrever ou ler os arquivos. Para particionar a base de dados em conjuntos de treinamento e de teste, foi utilizado o método de validação cruzada. Mais especificamente, 20% dos indivíduos de cada classe foram usados para teste, enquanto os 80% restantes foram usados na etapa de treinamento. Durante a etapa de treinamento, os dicionários associados a cada classe foram gerados para $K = 9, \ldots, 16$. Após isso, durante a etapa de testes, calculou-se a acurácia obtida para cada valor de K. Os resultados obtidos encontram-se nas figuras 1 e 2, que mostram, respectivamente, a acurácia e os tempos de execução em função de K.

Para gerar os conjuntos de treinamento e de testes, gerar os modelos, realizar os testes e plotar os resultados, foi utilizado um *shell script*. O *script*, que faz uso da ferramenta *gnuplot* para gerar os gráficos, pode ser executado através da seguinte sequência de comandos.

```
# para compilar o algoritmo LZW
mkdir build && cd build && cmake .. && cd ..
# para fazer os testes
bash guten.sh
# para visualizar os resultados
evince plot_accuracy.ps
```

Na Figura 1, pode-se perceber que, independente da métrica utilizada, o fator que mais impacta a acurácia do método é o valor de K. Isso faz sentido, já que esse valor dita o número de padrões que o dicionário que é usado como modelo é capaz de armazenar. Embora o comportamento do método em ambos os casos seja semelhante, percebe-se que o uso do número de índices do dicionário como métrica de distância proveu os melhores resultados para valores mais elevados de K.

A Figura 2, por sua vez, divide os tempos de execução em dois tipos: (i) os tempos necessários para gerar os modelos e (ii) os tempos necessários para fazer os testes (i.e., comprimir cada arquivo de teste usando cada um dos dicionários). Assim como na Figura 1, isso é feito para cada valor de K. Os tempos mencionados foram medidos com a escrita do arquivo no disco ativada e desativada. Como esperado, maiores valores de K impactam negativamente no tempo tanto da etapa de treinamento quanto na etapa de testes. Além disso, pode-se perceber que mesmo que mínima, a desativação da escrita do arquivo comprimido causa uma redução nos tempos de execução. O impacto provavelmente seria mais perceptível em bases de dados maiores, ou em arquivos mais volumosos.

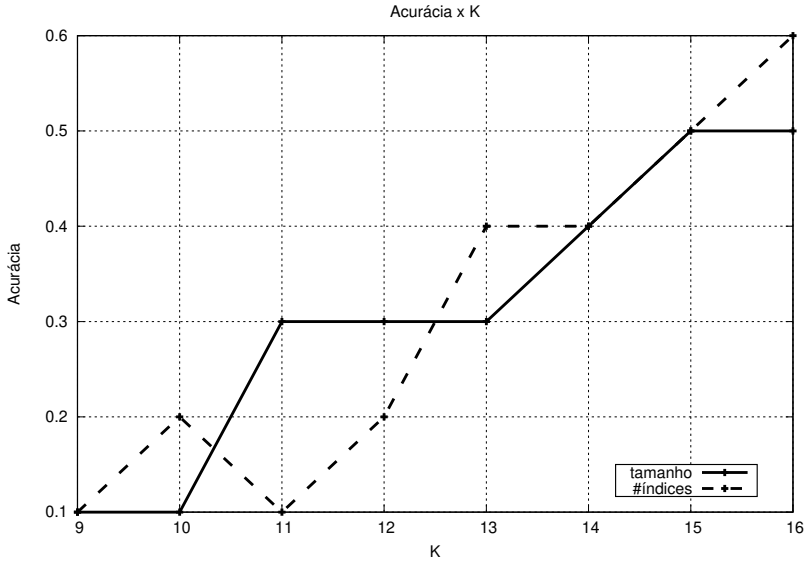

Figura 1: Taxa de acerto em função de K

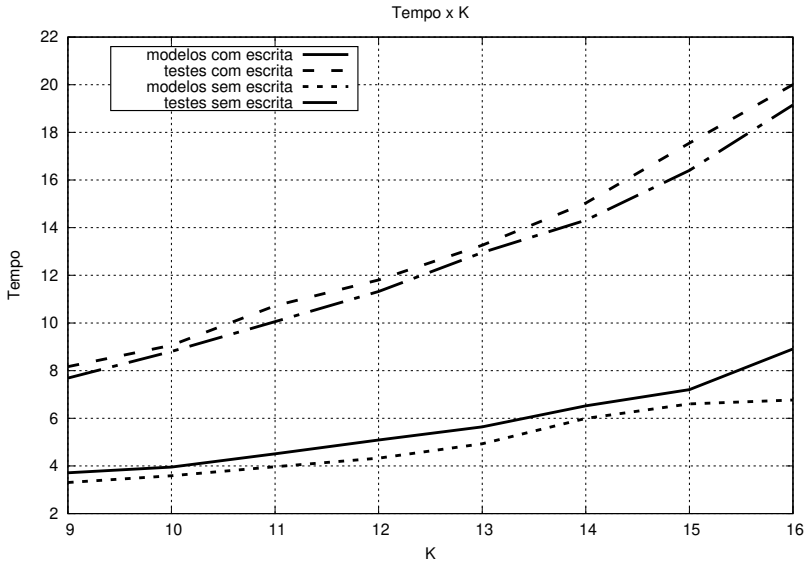

Figura 2: Tempos de execução em função de K

4 Considerações finais

Neste trabalho, uma aplicação do algoritmo LZW à classificação de textos foi mostrada. Embora a acurácia obtida não seja, no geral, competitiva com outros métodos, o método mostrou-se promissor, já que exigiu apenas alterações minimas no algoritmo de compressão. É importante salientar, ainda, que os resultados mostrados foram obtidos sem qualquer tipo de pré-processamento.

Para trabalhos futuros, além das melhorias à implementação do LZW sugeridas no relatório anterior, sugere-se, ainda, o uso de uma base de dados mais ampla. Isso ajudaria a melhor aferir o impacto da desativação da escrita do arquivo comprimido no disco. Seria possível, ainda, utilizar uma validação cruzada *k-fold* e obter uma medida mais confiável da acurácia do método. Sugere-se, aidna, a adição de uma etapa de pré-processamento ao método. Em [3], por exemplo, uma etapa de remoção de pontuação, *tabs*, e quebras de página e linhas é feita, e impacta positivamente na acurácia do método.

Referências

[1] Michael Dipperstein, mar 2015. Disponível em `http://michael.dipperstein.com/lzw/`.

[2] Max Halford, jul 2021. Disponível em `https://maxhalford.github.io/blog/text-classification-by-compression`.

[3] Yuval Marton, Ning Wu, and Lisa Hellerstein. On compression-based text classification. In David E. Losada and Juan M. Fernández-Luna, editors, *Advances in Information Retrieval*, pages 300–314, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[4] Mark Nelson, oct 1989. Disponível em `https://marknelson.us/posts/1989/10/01/lzw-data-compression.html`.

[5] Mark Nelson, nov 2011. Disponível em `https://marknelson.us/posts/2011/11/08/lzw-revisited.html`.

[6] Nitin Thaper. *Using compression for source-based classification of text*. PhD thesis, Massachusetts Institute of Technology, 2001.