universidade de aveiro deti departamento de electrónica, telecomunicações e informática

Lab work n◦2

Teoria Algorítmica da Informação

Trabalho realizado por:

- Isaac dos Anjos Nº Mec. 78191
- Lucas Barros Nº Mec. 83895
- Pedro Cavadas Nº Mec. 85090

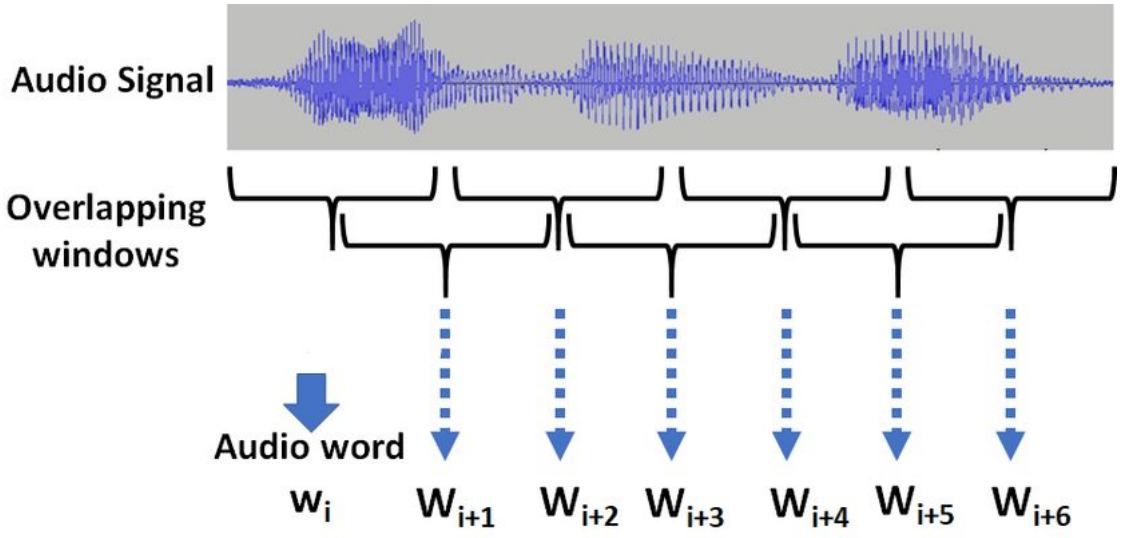
Introdução

O principal objetivo deste trabalho é desenvolver e testar um sistema de identificação de música automática. Tal é feito utilizando pequenas amostras de música (por volta de 10 segundos) para fazer a query. Para comparar com a amostra, existe um base de dados que tem várias músicas guardadas, e por cada amostra, compara-se com as músicas conhecidas. A música de que resultar um erro/distância menor, será o resultado. Todo este processo, será explicado com maior detalhe numa fase posterior deste relatório.

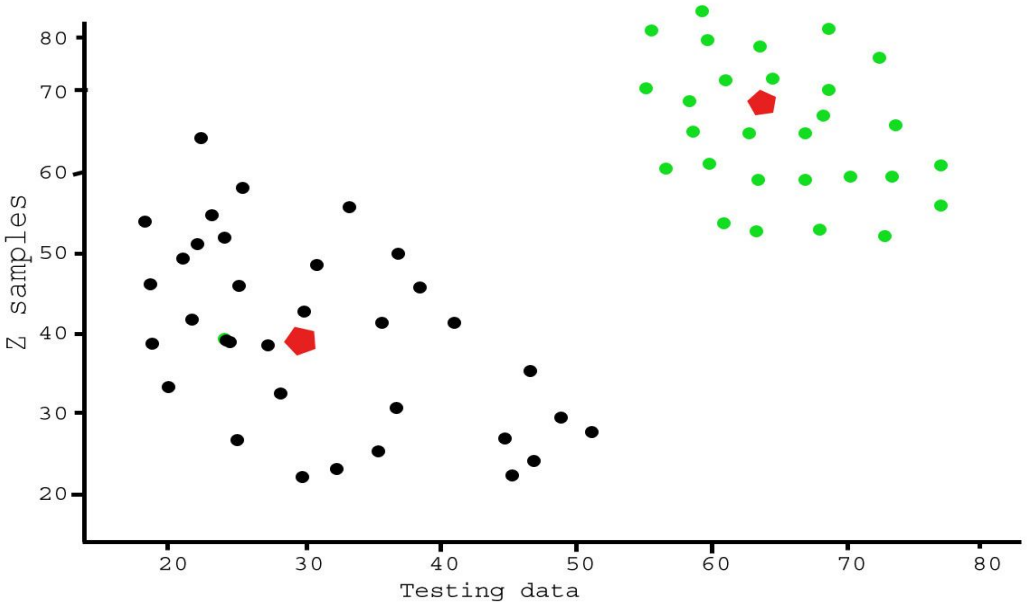
Este relatório está dividido em ... (é preciso escrever esta shit first).

Modelo Matemático

Para gerar o banco de dados, são calculados codebooks das músicas. O processo de geração dos codebooks está representado na imagem abaixo.



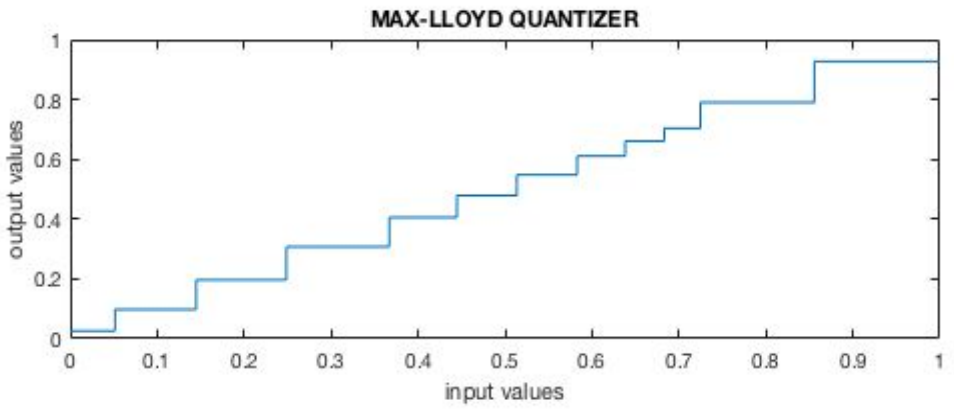
Descrevendo a imagem acima, tendo um ficheiro de áudio original, tal ficheiro é partido em subconjuntos com overlapping (caso contrário o excerto de música a descobrir poderia se encontrar entre dois subconjuntos). No entanto, guardar os subconjuntos sem compressão seria inadequado pois guardar os áudios com overlapping, ocuparia mais espaço do que guardar os áudios originais. Tendo isso em consideração, recorremos ao algoritmo de k-means. A imagem seguinte, representa o algoritmo de k-means.



O K-means é um algoritmo que a partir de um conjunto de dados, divide esses dados em k clusters, cada cluster com um centróide que representa o ponto médio do cluster. Adaptando ao codebook do nosso áudio, cada subconjunto, possui muitos dados. Recorrendo ao algoritmo de K-means, são calculados K centróides reduzindo assim o número de dados a guardar por cada subconjunto. No fim, são apenas guardados os K centróides

De seguida, para classificar então um segmento de áudio, esse segmento é dividido da mesma forma que os originais foram divididos em *audio words* e depois para cada *codebook* na base de dados, para cada *audio word* do segmento é calculado o centróide do *codebook* mais próximo e acumula-se um erro que é dado pela distância (quadrada para diminuir o tempo de processamento) entre essa centróide e *audio word*. No final, o *codebook* que tiver um menor erro associado é a classe selecionada.

Para simular uma situação real em que pode ocorrer ruído, recorremos a um algoritmo (uniform scalar quantization), para comprimir ou seja, reduzir o número de bits utilizados para representar cada ficheiro de áudio. O procedimento deste algoritmo está representado na imagem seguinte.



O que mostra a imagem é um conjunto de input values [0, 0.1, 0.2, ... 1] ser mapeado num conjunto de output values [0, 0.2, ..., 1] com metade do conjunto dos valores possíveis. Um mapeamento destes, iria resultar em menos 1 bit necessário para representar a informação. No exemplo acima, havendo 11 valores possíveis de input, sería necessário log2(11) = 3.46 = 4 bits para a representação. No conjunto de output, havendo 6 valores possíveis, seria necessário log2(6) = 2.58 = 3 bits para a representação. Os valores de input sem mapeio direto para valores de output, por exemplo 0.1, são mapeados para o valor mais próximo deste.

Para calcular o ruído introduzido na amostra, calculou-se o SNR(Signal-to-Noise ratio) que é dado pela seguinte fórmula:

$$\mathrm{SNR} = 10 \log_{10} \frac{E_s}{E_n} \quad (\mathrm{dB}),$$

em que $$E_s = \sum_k x_k^2$$

e $$E_n = \sum_k (x_k - \tilde{x}_k)^2$$

Este fator tem como propósito avaliar a robustez da solução.

Descrição da Solução

A nossa solução para classificar um segmento de música, potencialmente com ruído, está dividida em 2 fases:

- na primeira fase gera-se um conjunto de *codebooks* com base nos ficheiros de áudio originais, gerando assim uma base de dados que será usada para então classificar o segmento de música. O código para esta fase encontra-se nos ficheiros *wavcb*.
- Já na segunda fase calcula-se o erro entre o segmento de música e cada uma das músicas da base de dados. Este processo é feito com recurso aos *codebooks* gerados na fase anterior e o código para esta fase encontra-se nos ficheiros *wavfind* (a classe *Codebook* é também utilizada nesta fase, o que implica que os seus ficheiros são também utilizados).

Os ficheiros que implementam os vários algoritmos necessários à resolução do problema encontram-se nas pastas *include/wav* (*headers*) e *src/wav* (implementação). Além destes ficheiros existem também ficheiros que implementam uma *main* e que se

encontram na pasta *src*. Estes últimos fazem recurso (incluem) dos ficheiros de implementação com o mesmo nome (ou seja, o *src/wavcmp.cpp* inclui o ficheiro *include/wav/wavcmp.hpp* e víncula o ficheiro *src/wav/wavcmp.cpp*), com a exceção do *wavcp.cpp*, que apenas copia um ficheiro *wav*, e do *wavfind.cpp* que utiliza os *codebooks* e portanto inclui também o ficheiro *include/wav/wavcb.hpp* e víncula o *src/wav/wavcb.cpp*.

src/wavcp.cpp

Este ficheiro contém o código que gera uma cópia de um dado ficheiro de áudio (opcionalmente com um *offset* e com um tamanho máximo, criando assim um segmento do ficheiro original).

src/wavhist.cpp

Este ficheiro contém o código que mostra um histograma de um dado ficheiro de áudio e de um dado canal desse ficheiro (opcionalmente mostr também uma média dos histogramas de todos os canais)

src/wavquant.cpp

Este ficheiro contém o código que gera uma versão quantizada de um dado ficheiro de áudio. Caso a versão quantizada seja de 8 ou menos *bits* o ficheiro é guardado em formato *wav* com 8 *bits* por *sample*. Caso contrário o ficheiro é guardado em formato *wav* com 16 *bits* por *sample*.

src/wavcmp.cpp

Este ficheiro contém o código que calcula e mostra o "*signal-to-noise ratio*" de um certo ficheiro de áudio em relação ao ficheiro original.

src/wavcb.cpp

Este ficheiro contém o código responsável por gerar um *codebook* para cada ficheiro áudio indicado. Os *codebooks* gerados são armazenados em formato binário. Este processo é feito em múltiplas *threads*, mais especificamente, uma para cada ficheiro de aúdio

src/wavfind.cpp

Este ficheiro contém o código que compara cada um dos *codebooks* na base de dados com um ficheiro de amostra e indica qual o *codebook* correspondente a essa amostra. Isto significa que o ficheiro de áudio original correspondente a esse *codebook* é o ficheiro de áudio do qual a amostra (potencialmente, com ruído) é retirada.

Resultados

Para testar o funcionamento do nosso algoritmo criamos então uma base de dados de codebooks com base em 7 sons diferentes, para isto utilizamos os seguintes parâmetros:

- *Vector size*: 8820
- *Overlap factor*: 4410
- *Cluster size*: 100
- *Max iterations*: 100

Este processo demorou cerca de 1 minutos e 4 segundos. No entanto, este tempo é altamente dependente da máquina utilizada, sendo que a máquina utilizada neste caso possui um processador de alta performance de última geração o que implica que na maioria das máquinas (de uso pessoal) deverá demorar mais tempo.

Em seguida realizou-se uma cópia de 6 segundos de cada um dos ficheiros usados na base de dados e uma cópia de 3 desses ficheiros de apenas 1 segundo (ou seja, 10 cópias no total). De seguida para cada uma dessas cópias foi feita quantização, para introdução de ruído, de 8, 4 e 3 bits (ou seja, no final ficou-se com 40 ficheiros diferentes).

Finalmente, cada um dos ficheiros gerados foram submetidos para classificação, ou seja, 40 ficheiros diferentes foram submetidos para classificação. Desses 40, 34 foram classificados corretamente, sendo que as cópias dos originais, e as suas versões de 8 bits foram todas classificadas corretamente. Isto significa que apenas nas cópias de 4 e 3 bits houve erros de classificação (1 erro em 10 classificações nas de 4 bits e 5 erros em 10 nas de 3 bits).

Estes resultados provam-se muito satisfatórios, uma vez que mesmo com segmentos curtos, foi necessário introduzir uma enorme quantidade de ruído (tornando a música quase imperceptivel ao ouvido humano) para que o classificador começasse a classificar incorretamente os ficheiros de áudio.

Nota: devido ao excesso de *prints* foi criado um *script* em *Python3* (como executar está explícito no ficheiro *README*) que reproduz os resultados obtidos.

Conclusão

Com este trabalho, pudemos concluir que o método utilizado para classificação prova-se ser bastante eficaz nessa tarefa, pelo menos para os parâmetros testados. E apesar do método necessitar algum poder computacional para a geração dos *codebooks* (de notar que a geração de *codebooks* é independente e por isso não é necessário gerar para todas as músicas sempre que se insere uma nova, o que é em si uma grande vantagem) o processo de classificação é relativamente rápido e por isso, concluímos que é um método que poderá ser utilizado no dia a dia e até de forma comercial.

Referências

https://github.com/premake/premake-core