Recognize a Song with Shazam Algorithm

1. INTRODUCTION

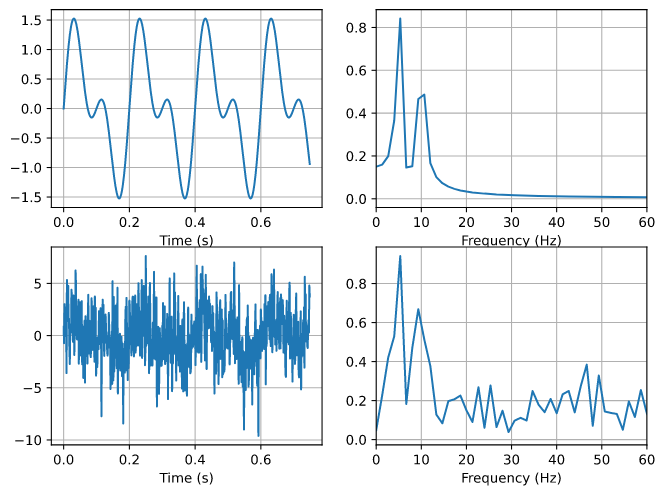
「Shazam音樂辨識」是現在許多人手機中的必備軟體，只要聆聽幾秒鐘的音樂片段，它就能從巨大的音樂庫找到對應的歌曲，2018年被Apple買下來以後[1]，更讓它愈來愈普及於眾人的生活，這使我們好奇背後的原理，因而希望在期末專題實作其高效率的聲音比對技術。

1. SHAZAM ALGORITHM

根據Shazam論文：“*An Industrial-Strength Audio Search Algorithm*[2]”，Shazam有三項主要技術：

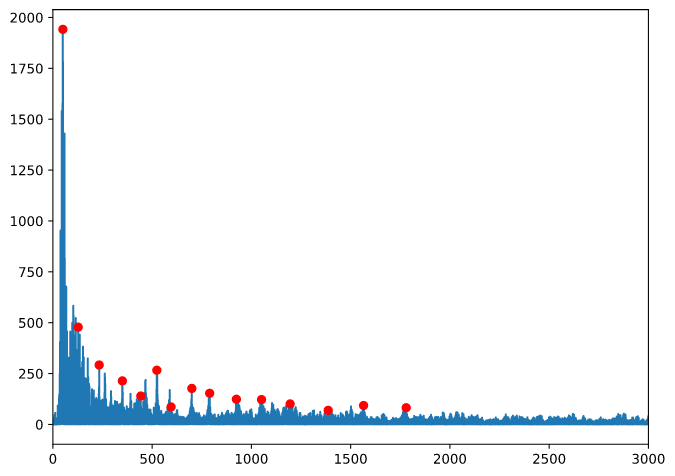
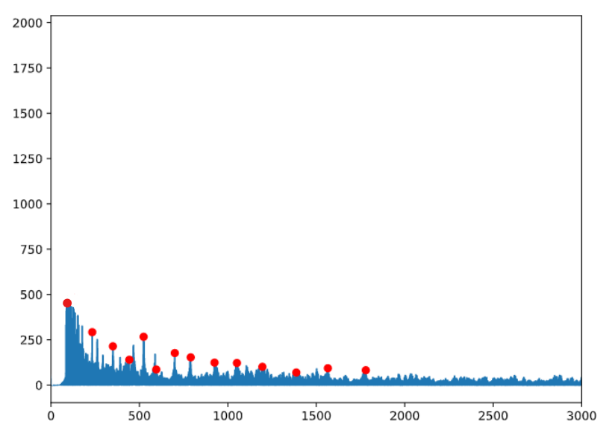
* + 1. 聲音特徵提取（Robust Constellations）

在Shazam Algorithm中，一首歌曲先被切成一段一段的等寬window，每一段的聲音經過Short Time Fourier Transform（STFT）後，頻譜中的多個「頻率峰值」即是這段的聲音特徵，當聲音遭到環境噪音破壞時，由於一首歌在同一時間的dominant frequency不變，即使波形遭到破壞，仍然能還原出Short Time Peak Frequency（STPF），從而有效降低環境影響。



（圖一）：環境噪音對PF（Peak Frequency）影響[3]

除了環境問題，我們也發現，Shazam於2000年提出的這套演算法，也能很有效地解決不同麥克風的頻率響應（frequency response）問題，當使用者用同一台手機進行錄音時，會有一些PF遭到消除或是位移，但Shazam取多個進行比較，避免掉特定頻率消失的問題，因此能精確比對出歌曲，我們將在底下做更多討論。



（圖二）：頻率響應對PF（Peak Frequency）影響

在實作上，我們將一首歌切成一段一段長度為*W*的sampling windows，window每次取值向後位移*W*/2，值得注意的是，當這首歌無法被W整除時，我們會在後面補0來維持每一個window等寬。對每個window做STFT並提取STPF，得到許多特徵參數： (*t*1, *f*11), (*t*1, *f*12), …, (*t*1, *f*1N), (*t*2, *f*21), (*t*2, *f*22), …, (*tT*, *fT*1), …, (*tT*, *fTN*)，其中*N*是一段短時頻譜中PF最多的數量（由我們設定），且為了避免*f*11*f*12, …, *f*1N過於接近，我們限制兩個peak之間至少需距離*D*，因此對於每個*t*i，PF的數量有可能少於*N*。



STFT of *t*1

(*t*1, *f*11)

*f*

*t*



(*t*1, *f*12)

(*t*1, *f*14)

(*t*1, *f*13)

*t*1

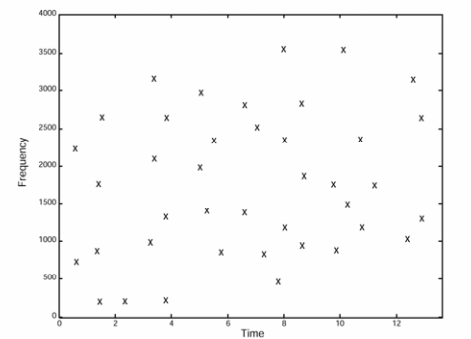
*t*2

…

*t*3

（圖三）：聲音特徵提取圖解

將這些點集合起來，我們可以得到一個time domain (*ti*)對應peak frequency (*fij*)的點集合，以原論文的圖所示，Shazam認為這樣的特徵很像星空點綴圖，因而稱之為constellations。



（圖四）：robust constellations[2]

* + 1. Hash Function (Fast Combinatorial Hashing)

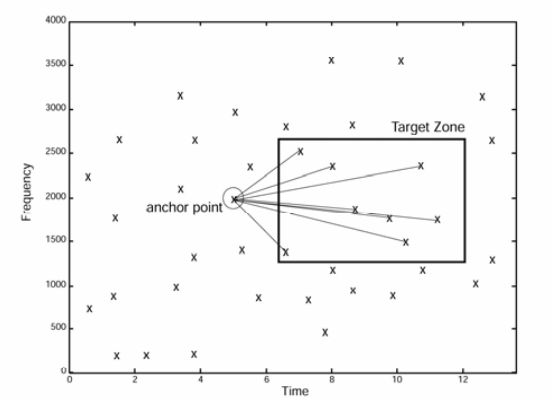
在上一小節中，有了時間對應PF的點集合以後，還不能拿來進行比對，很顯然一段錄音得到的PF，無法完美對應上原曲，正如上（圖二）所示，當低頻因錄音品質低劣而消失時，第一個PF消失、而第二個產生了偏移；此外，許多歌在不同的橋段可能會有相似的頻率分布，例如一段木吉他的C大調順階和弦、或是電吉他的solo，頻率分布很有可能雷同，都導致直接比對上的準確率降低。Shazam提出的解法是：把問題的複雜度提高。比起直接比較PF，不如比較PF在時間上的變化，在原始論文裡，他們提出一個hash function：

Hash( (*ti*, *fij*), (*tm*, *fmn*) ) = 32-bit integer *V*

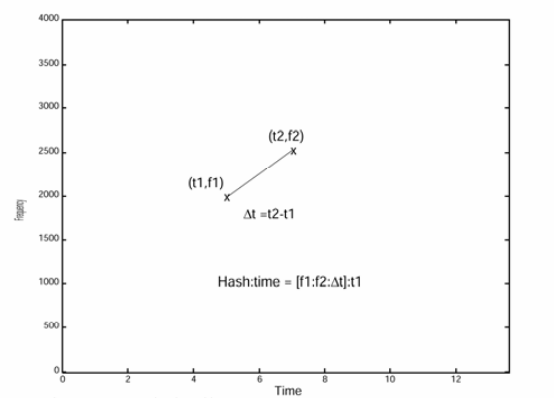
*V* = | *fij* | *fmn* | *tm* - *ti*

　10bit 10bit 12bit

比起10 bits的PF資訊，將每個點和另外一個點組成pair，提升成32 bits的資訊量，提高了精確度（ **The specificity of the hash would be about a million times greater.**[2]）



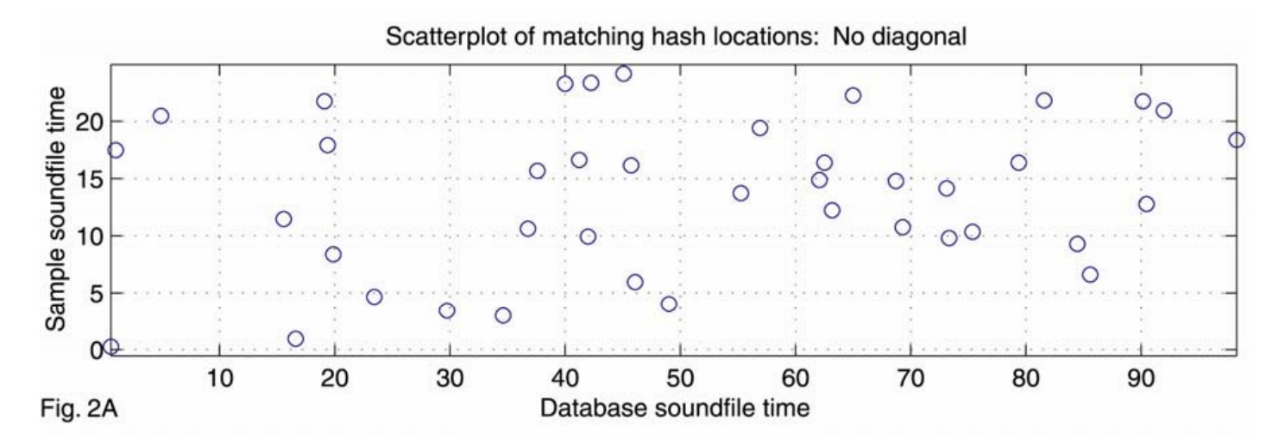
（圖五）：points pair[2]



（圖六）：hash[2]

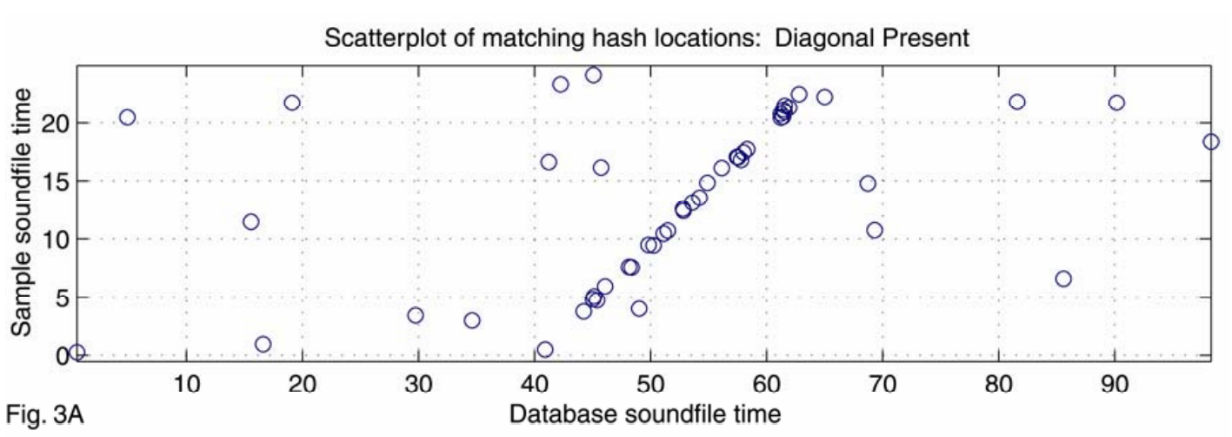
* + 1. 排名 (Searching and Scoring)

當使用者錄一段聲音以後，我們將這段sample *S*與資料庫中其他的data *D*所擁有的hash進行比較，把*S*的時間作為縱軸、*D*的時間作為橫軸，標記出具有相同hash的位置。



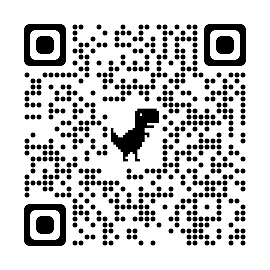
（圖七）：與非原曲的hash比對圖[2]

此演算法的一大假設在於，*S*和*D*並沒有播放速度的差異，因此Shazam提出，若*S*與*D*能夠完美對應，應該形成斜率為1的斜直線，對該直線上各點而言，時間差Δ*t* = *t*(*D*) - *t*(*S*)會是一樣的，因此找到最大的Δ*t*集合，代表這個片段有最佳的對應關係，於是算出各首歌最大的Δ*t*集合內的點數量，就能知道哪首*D*最貼切*S*，從而給出答案。



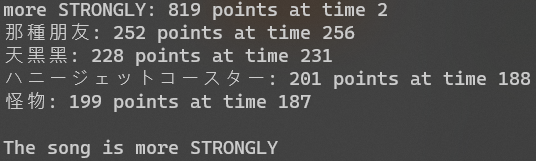
（圖八）：與原曲的hash比對圖[2]

1. 實作
   * 1. 專案連結

我們在參考原始paper[2]與其他資料[3][4]以後，用python實作出一套Shazam聽歌辨識程式，連結位於

<https://github.com/Xuan-Yi/DSP-Final-Project>

使用方式請參照README.md所示，環境設置好以後即可使用main.py來做辨識，輸出應如下（圖九）所示，包含前五名最有可能的答案，以及各自在哪個時間對應到最多hash value，讓使用者得知這片段的頻率分布與哪些地方相近，並在最後給出辨識答案。



（圖九）：程式輸出

* + 1. 參數調整與performance

Shazam原始論文並未提到有哪些特定參數可以做調整，然而我們根據它的理論，在實作的同時，把一些可能影響辨識成功率的因素整理成參數表，如下（表一）所示：

（表一）：參數表

|  |  |
| --- | --- |
| 參數名稱 | 描述 |
| swnd\_length | 在做STFT時，一段window的長度是多少 |
| peaks\_num | 對於每個window，在取PF時，最多取幾個 |
| spread\_distance | 在取PF時，對於兩個PF之間的最短距離至少要是多少，為了避免像48.49.50.51.52(Hz)這種PF過度集中的狀況 |
| freq\_bits | 進行hash時，PF資訊應該要以幾bits進行儲存（原論文是10 bits[2]，詳見I.2小節） |
| fanout\_size | 在建hash table時，對於每一點，它後面幾個window的點會跟它湊為一對 |

為了要測試performance，我們準備了50首歌，涵蓋中文、日文，以及男女聲、合唱、樂團，同時有純音樂、搖滾、流行樂……各種曲風。在實驗中，我們將筆電架高在書桌上進行播放，故意讓它產生殘響（reverb）效果，並且用手機開啟後錄製6秒（44.1 kHz, 128kbps），藉此模擬一般使用者在一個空間聽到歌曲後，拿出手機錄音的情境。

對於每一首歌，我們取用2個6秒片段，一個固定取用前奏、另一個隨機選擇其他片段，可能是樂器solo、副歌、過門、間奏……

綜上所述，錄製的聲音除了空間感濃厚以外，低頻也因筆電喇叭、手機錄音的頻率響應而嚴重不足，加上許多前奏往往過於小聲，導致信噪比偏小，在這樣100個低劣的測資品質下，用不同的參數組合去計算正確率：

|  |  |
| --- | --- |
| 參數名稱  （表二）：我們的預設參數 | 我們的預設值 |
| swnd\_length | 2 (秒) |
| peaks\_num | 15 (個PF) |
| spread\_distance | 200 (samples) |
| freq\_bits | 12 (bits) |
| fanout\_size | 10 (個window) |

在初步測試時，和效能一併考量後，我們選用這套預設參數，獲得的正確率為92%，隨後針對各個參數進行細部研究，底下表格中當我們更改某一變數時，其餘參數都是以上述的預設值做實驗：

（表三）：不同參數的影響

|  |  |  |  |
| --- | --- | --- | --- |
| swnd\_length | 1 | 2 | 3 |
| 正確率 | 100% | 92% | 75% |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| peaks\_num | 5 | 10 | 15 | 20 |
| 正確率 | 58% | 86% | 92% | 96% |

|  |  |  |  |
| --- | --- | --- | --- |
| spread\_distance | 50 | 200 | 400 |
| 正確率 | 74% | 92% | 95% |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| freq\_bits | 6 | 10 | 12 | 16 |
| 正確率 | 3% | 32% | 92% | 100% |

|  |  |  |  |
| --- | --- | --- | --- |
| fanout\_size | 5 | 10 | 15 |
| 正確率 | 92% | 92% | 92% |

因此我們從實驗得知，fanout\_size影響不大，當swnd\_length愈小、peaks\_num愈大、spread\_distance愈大、freq\_bits愈大，正確率能夠提升，直覺來看也就是因為資料的比較更「精細」了，這些參數調整讓歌和歌之間的hash剛好撞到的可能性降低，從而讓正解與類似的歌更能「分得開」，但查詢時間相對的也會提升，就會需要tradeoff。

* + 1. 未來展望

Shazam提及，在一般PC上，給定一段聲音，當資料庫有20,000首歌時，他們有辦法在0.005到0.5秒內辨識出是哪首歌[2]，但我們的程式光是用預設參數從50首歌中找出答案，就需要約2秒的時間，我們推測Shazam快速的原因在於data structure、database query、parallel computing這些方面的improvement，例如hash計算、比對……都是有辦法平行處理的，而我們尚未能完美處理這塊，如今在得到不錯的表現以後，未來期許我們能讓這套程式在效率上也能提升，並運用於我們的生活或其他專案裡。

1. Reference

[1] Wikipedia, Shazam (application). Retrieved from <https://en.wikipedia.org/wiki/Shazam_(application)>

[2] Avery Li-Chun Wang, Shazam Entertainment, Ltd. “[*An Industrial-Strength Audio Search Algorithm*](https://www.ee.columbia.edu/~dpwe/papers/Wang03-shazam.pdf)”, ISMIR 2003, 4th International Conference on Music Information Retrieval, January 2003. Retrieved from <https://www.ee.columbia.edu/~dpwe/papers/Wang03-shazam.pdf>

[3] Michael Strauss, “*How Shazam Works - An explanation in Python*” , 29 January 2021. Retrieved from <https://michaelstrauss.dev/shazam-in-python>

[4] Shazam website: <https://www.shazam.com>

1. 分工

吳宣逸：主題與資料收集、MFCC與PF等特徵參數實作比較與測試（試過用MFCC發現不太行，印出倒頻譜後找不到比PF更好的判別依據）

呂建廷：Shazam主程式實作、錄音測試與參數調整結果分析、報告統整