Recognize a Song with Shazam Algorithm

I. INTRODUCTION

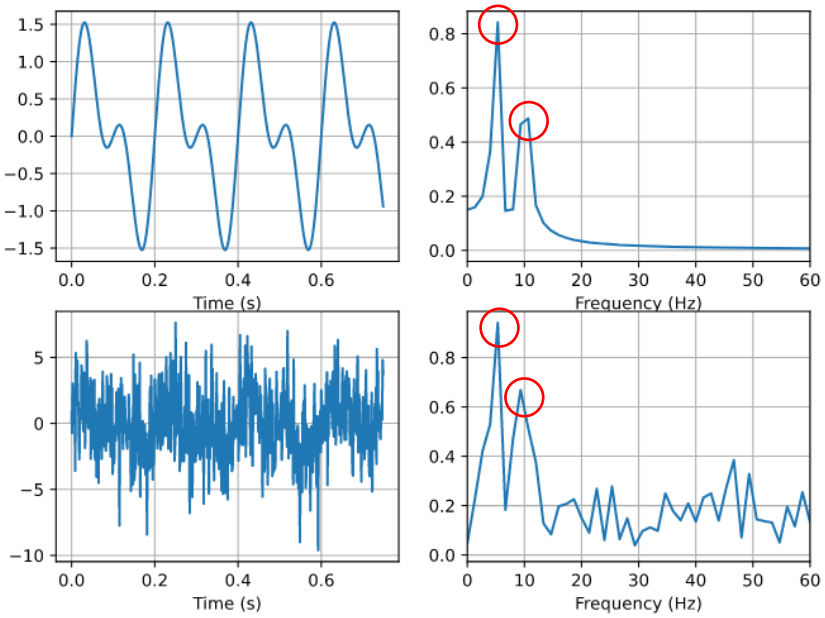
「Shazam 音樂辨識」是現在許多人手機中的必備軟體，只要聆聽幾秒鐘的音樂片段，它就能從巨大的音樂庫找到對應的歌曲，2018 年被 Apple 買下來以後[1]，更讓它愈來愈普及於眾人的生活，這使我們好奇背後的原理，因而希望在期末專題實作其高效率的聲音比對技術。

II. SHAZAM ALGORITHM

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

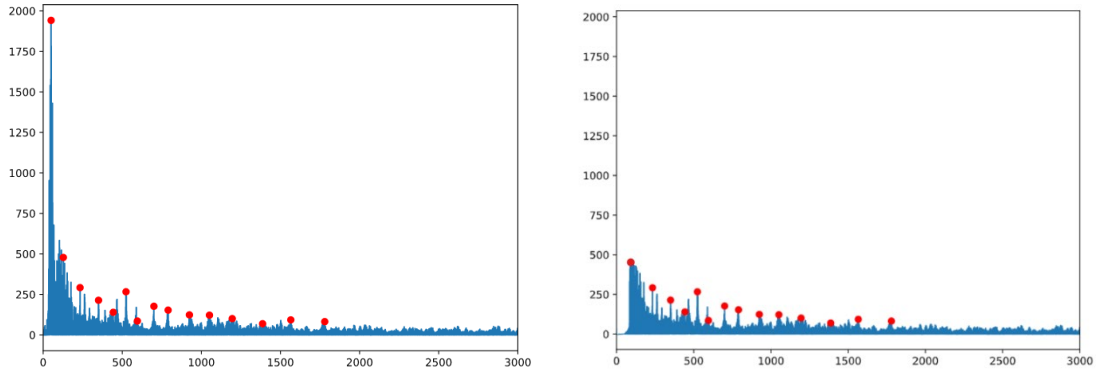
1. 聲音特徵提取（Robust Constellations）

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。



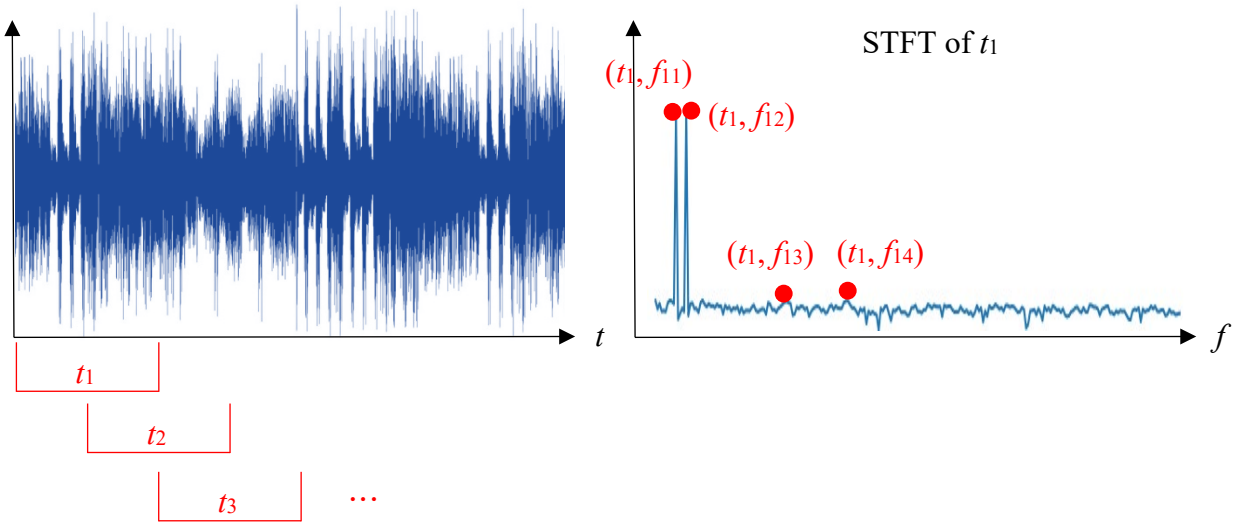
（圖一）：環境噪音對 PF（Peak Frequency）影響[3]

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。



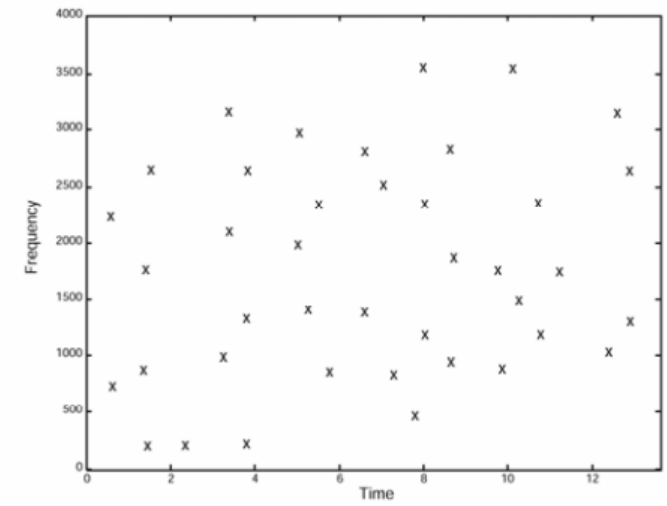
（圖二）：頻率響應對 PF（Peak Frequency）影響

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}), …, (t_T, f_{T1}), …, (t_T, f_{TN})，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。



（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 (t_i)對應 peak frequency (f_{ij})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]

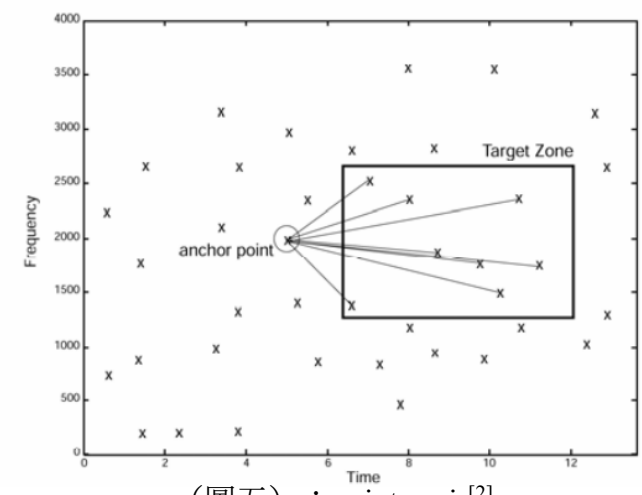
2. Hash Function (Fast Combinatorial Hashing)

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：

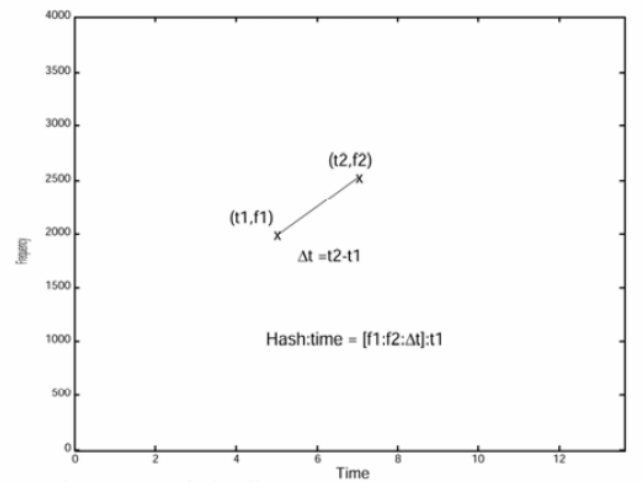
$$\text{Hash}\left((t_i, f_{ij}), (t_m, f_{mn}) \right) = \text{32-bit integer } V$$

$$V = \underset{}{|} \underset{10\text{bit}}{f_{ij}} \underset{}{|} \underset{10\text{bit}}{f_{mn}} \underset{}{|} \underset{12\text{bit}}{t_m - t_i}$$

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]）



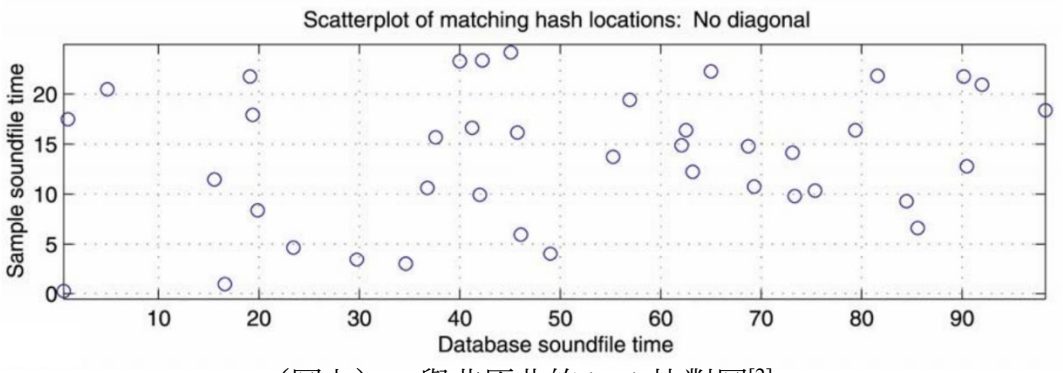
（圖五）：points pair[2]



（圖六）：hash[2]

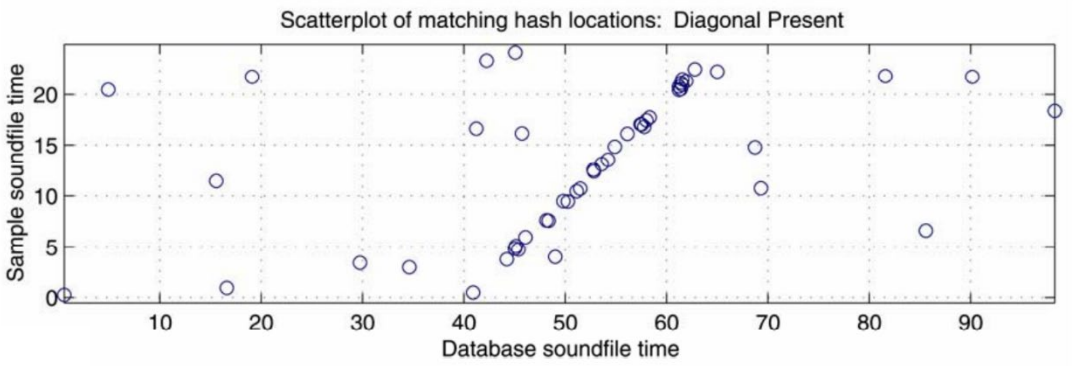
3. 排名 (Searching and Scoring)

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 $\Delta t = t(D) - t(S)$會是一樣的，因此找到最大的 Δt 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 Δt 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]

III. 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。

```
more STRONGLY: 819 points at time 2
那種朋友: 252 points at time 256
天黑黑: 228 points at time 231
ハニージェットコースター: 201 points at time 188
怪物: 199 points at time 187

The song is more STRONGLY
```

（圖九）：程式輸出

2. 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：

（表二）：我們的預設參數

參數名稱	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。

3. 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。

IV. 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*", 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

V. 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