Poetry with Spotify - assignment solution

Luminita Moruz

September 11, 2013

1 Problem formulation

An algorithm should be implemented that takes as input a message, and returns a list of Spotify tracks recreating this message. The following requirements should be incorporated: the message should be UTF-8, no case sensitivity and shorter lists of tracks should be preferred.

2 Assumptions

In my implementation, a few simplifications and assumptions were made:

- **Only exact matches are considered**. Any fragment of the message has to match precisely the full name of a track.
- **The message is split by punctuation marks**. To improve semantics, the message is split in sentences that are treated separately. At the moment, a limited set of punctuation marks are used to split each message.
- **The algorithm gives an output only when a full solution is found**. If only a fraction of the message can be covered by tracks, the algorithm will output that no solution was found.

3 Implementation

Given a message, the first step is to split it in sentences based on punctuation marks. Each resulting sentence is then treated separately.

Let S be a sentence to be analyzed, consisting of n words $w_1, .., w_n$. The goal is to obtain a collection of sub-sentences $[w_1, .., w_{i_1}]$, $[w_{i_1+1}, .., w_{i_2}]$, .. , $[w_{i_k+1}, .., w_n]$, with each such sub-sentence corresponding to a Spotify track, and with k preferable a small number. Two solutions were implemented, denoted BFS and FAST-BFS, both briefly described below.

3.1 BFS

BFS is a version of the best-first search algorithm. First, the algorithm fills a matrix L that stores all the sub-sentences for which a Spotify track was found. More precisely, a line of this matrix $L[i]$ will include a list of indices j_k such that there is a Spotify track corresponding to the sub-sentence $w_i w_{i+1}..w_{j_k}$. For an index i, all the elements of $L[i]$ will be larger or equal to i, and will be sorted in ascending order.

As an example, for a sentence "Roses are very red", we try to find tracks named "Roses", "Roses are", "Roses are very", "Roses are very red", "are", "are very", "are very red", "very", "very red" and "red". If we assume that the only tracks found were "Roses", "Roses are" and "very red", then the matrix L will include only 3 elements: $L[1] = [1, 2], L[3] = [4]$. $L[2]$ and $L[4]$ will be empty since no tracks starting with "are" or "red" were found.

To improve the speed, a dictionary is maintained including all the queries run up to that point. Before running a new query, we check in the dictionary whether the query has already been run before.

The matrix L can be seen as a directed graph where each word in the sentence is a node, and there is an edge from node i to j only if $i <= j$ and there is a Spotify track matching $w_i..w_j$. Seen from this perspective, the problem is reduced to finding paths in the graph from node 1 to node n.

Since it would be too computationally intensive to try to find all the paths in this graph, one alternative is to use a type of best-first search algorithm where we try to visit the most promising paths first, and stop when a solution is found. Here this is accomplished by using a greedy-type of heuristic, where at each step the longest track is chosen. If no solution is reached, the next longest track is considered and so on until all the possibilities are investigated. This idea can be implemented by using a stack.

3.1.1 Analysis

When building the matrix L, the method involves running $n * (n + 1)/2$ queries. While the traversal of the graph could in the worst case involve visiting all the paths, in practice it will be much more efficient.

As long as a solution exists, the algorithm will find it. However, the algorithm is not guaranteed to find an optimal solution. Despite this, for the - admittedly few - examples I tried it, the results looked as expected.

3.2 FAST-BFS

When applying the BFS method described above for a few examples, I noticed that the calculation of the matrix L involved running many unnecessary queries. Following this observation, I implemented a version of the algorithm called FAST-BFS where a query is run only when the investigation of a certain path requires it.

While in the most extreme cases FAST-BFS may be similar in speed to BFS, for virtually all the examples I applied it I obtained substantial speed-ups.

3.3 Other approaches

A completely different approach would be to start by querying a more atypical word or short sub-sentence from the message. We can then examine all the returned tracks to see whether they match a larger portion of the message. Such an approach would limit the number of queries, but would require more space. Another aspect to take into account here is that, as far as I observed, the search function of the API gives only the first 100 hits. This may result in losing potential solutions.

4 Possible improvements

The algorithms described here can be improved in various ways:

- **Better search strategy**. Instead of starting by querying the longest tracks, one can query sub-sentences that have a typical length for the name of a song. This would involve analyzing the name of the songs present in the database, and see what is the distribution of the number of words of the song names.
- **Parallelization**. The queries could be easily run in parallel.
- **Partial solution**. The algorithm should be adapted to output partial solutions. This could involve for example keeping a list with all the successful queries, and then output according to some criteria one of the possible partial solutions.

In addition, other ideas for such an application could include:

- Maintaining a dictionary of the most common abbreviations that can be used when no exact solution is found.
- When various Spotify tracks match a certain query, a criterion to choose a "suitable", rather than a random track, can be implemented.