A System for the Automatic Identification of Rhymes in English Text

Bradley Buda, University of Michigan

EECS 595 Final Project

December 10th, 2004

1 Acknowledgments

The author would like to thank Nick Shawver, B.A. in Music from Depauw University, for his guidance and suggestions in developing this system.

2 Abstract

This paper presents a system for identifying rhyming words in a block of text. The system is designed for finding rhymes in hip-hop (rap) music lyrics, but is general enough to work for any block of text containing rhymes. The system runs on Windows or Linux using the .NET Framework. It relies on the CMU Pronouncing Dictionary to find the constituent sounds for the supplied words, then uses a series of custom algorithms to find patterns that indicate rhymes. The system is capable of finding rhymes that span multiple words. Experimental results indicate that the system finds nearly all the intentional rhymes in a lyric, but fails in that it also finds a large number of unintentional rhymes. This paper describes the design of the system, describes the algorithms used in the system, gives experimental results, and presents an analysis of the system's successes and shortcomings and suggestions for future work.

3 Overview

3.1 Motivation

Music, especially hip-hop music, relies on a regular pattern of rhymes to match the spoken lyrics to the accompanying 'beat'. When music is heard by a human listener, it is usually easy to pick out the rhymes in the lyrics because they occur at semi-regular intervals and are spoken in some cadence. A human observer can usually pick out rhymes in written lyrics as well, sometimes with more difficulty.

This project seeks to determine whether or not a machine observer can pick out rhymes in written lyrics. In order to answer this question, a software system was developed which takes as input the lyrics to a song, and outputs candidate rhymes, along with a 'score' indicating the likelihood of each rhyme. The system was then tested on several sample song lyrics to determine its effectiveness against a human analysis.

3.2 Design Principles

The rhyme identification system was developed as an object-oriented software system, using C#, and runs on the Windows or Linux platform.

The system was developed with practical success, rather than a strong theoretical base, as the primary motivation. Thus, the system was developed in an iterative fashion; each algorithm was written, tested, rewritten, and sometimes discarded until the developer found the results satisfactory. Therefore, the algorithms presented in this paper are often intuitive, but do not always reflect some theoretical basis.

3.3 Using the System

The rhyme identification system is invoked from the command line. The user is prompted for a filename which contains the lyrics to be analyzed. The lyrics file must be in a very specific format: ASCII text with a single space between each word, and no line breaks or other extraneous punctuation or whitespace. It would not be difficult to extend the system to allow more free form input.

The system requires one file, dict.bin, which is a binary encoding of the CMU Pronouncing Dictionary. This file was generated by the system from the plain text file and is included in the source code distribution (see 4.1).

The system loads the plaintext lyrics and the binary dictionary and performs a lookup of each word in the plaintext. If any of the supplied words cannot be found in the dictionary, the rhyme identifier will fail.

Once the phonemes (sounds) used in the supplied lyrics are identified, the system executes a number of custom algorithms to identify the most likely rhymes in the text (see 4.3-4.6). The settings used by these algorithms can be changed by editing the Config.cs file and recompiling the code (see 8). The system then outputs these rhymes and prompts for another file.

The system can be aborted at any time by pressing Control-C.

4 Algorithms

4.1 CMU Dictionary Lookup

The CMU Pronouncing Dictionary, available at http://www.speech.cs.cmu.edu/cgi-bin/cmudict , is a machine-readable dictionary of over 125,000 English words. For each word,the dictionary provides one or more pronunciations. A pronunciation consists of a string of phonemes with (optional) stresses. A phoneme is one of 39 basic pronounceable sounds (see 7). Of the 39 phonemes, 15 are "vowels" with stresses. A stress can be one of "0", "1", or "2", where 0 indicates no stress, 1 indicates secondary stress, and 2 indicates primary stress. In words with more than one syllable, the stress markers indicate how much each vowel in emphasized in pronunciation.

The plaintext dictionary takes up 3.5 MB on disk, and is regularly structured so it can be read by either a human or a machine. However, the initial load and indexing of the plaintext dictionary into the rhyming system can take as long as 45 minutes (on the development machine, a 300 MHz Pentium II). Therefore, the dictionary was converted to a binary format more readable by the program. This file, "dict.bin", takes 2.9 MB on disk and loads in approximately 5 seconds. Looking up one of the 125,000 words in the dictionary takes under 100 ms.

A dictionary lookup simply takes a word, converts it into uppercase (the dictionary is not case-sensitive), and returns a list of stressed phonemes in that word, or a 'null' if the word is not in the dictionary. This information is fetched for each word in the supplied lyrics and sent to the syllable-finding algorithm for the next step.

4.2 Transforming Phonemes to Syllables

While a phoneme is a fundamental unit of pronunciation, a non-vowel phoneme cannot be pronounced by itself. In this project, a syllable is defined as a list of phonemes containing a single vowel phoneme. These syllables are the units which are actually involved in rhymes (see 4.3), so they must be found before rhymes can be found.

Syllables are found on a per-word basis. For each word inputted into the rhyming system, the number of syllables corresponds to the number of vowels. The trick is determining which consonant phonemes bind to which syllables.

The rhyming system employs a nearest neighbor system to carry out this binding. The system is not exact, but is correct in the majority of cases and represents a trade off between design complexity and exactness. Even in the cases where the algorithm incorrectly finds syllables, rhymes can still be determined accurately: see 4.3 and 4.5.

The algorithm for grouping phonemes into syllables:

```
for each vowel phoneme in the word:
      label the phoneme with a unique integer, starting a zero
for each consonant phoneme in the word:
      find the closest vowel phonemes to this consonant
            if there is a single closest phoneme, copy its label to this
            phoneme
            if there are two closest phonemes, copy the label of the phoneme
            with a higher stress
            if there are two closest phonemes and with the same stress, copy
            the label of the phoneme on the right (this is arbitrary but a
            tiebreaker is needed, and this seems to work better than copying
            the label on the left).
```

This algorithm is deterministic and will always find a labeling for each phoneme. The list of phonemes that represents each syllable is then converted into a new data structure, the 'Syllable' class. This structure represents the syllable as having four elements: the phoneme that represents the vowel, the list of phonemes that represent the consonants before the vowel (the *prefix*), the list of phonemes that represent the consonants after the vowel (the *suffix*), and the stress on the vowel (remember that consonants never have a stress, but vowels always do).

4.3 Determining if Two Syllables Rhyme

Determining whether or not two words, or even two syllables, rhyme is not a binary question – some words clearly rhyme, other clearly do not rhyme, but there is a large gray area of words that may or may not rhyme. Thus, instead of a Boolean function for determining if two syllables rhyme, the "Rhyme" function developed for this system returns a floating-point 'score', ranging from 0 to 1 inclusive, that determines the degree to which two syllables rhyme.

If two syllables have a rhyming score of 0, they share no information in common (different vowels, different stresses, and no phonemes shared in either the prefix or the suffix). The words "freeway" and "capstone" would have a rhyming score of 0 – the vowels in each syllable are different, the prefix and suffix sounds are different, and the emphasis is on different syllables in each word. Conversely, a score of 1 can only by obtained by two syllables that are in fact identical. This does not mean that only two identical words can rhyme – they must simply be pronounced in the same,way, such as "red" and "read" (the past-tense of "read").

Many pairs of syllables have a rhyme score that is neither 0 nor 1. The algorithm for determining whether or not two syllables rhyme is presented below:

```
initialize intermediate and possible to 0.
add PointsVowelStress points to possible
if syllable 'a' and syllable 'b' have the same stress:
      add PointsVowelStress points to intermediate
add PointsVowelSound points to possible
if syllable 'a' and syllable 'b' have the same vowel phoneme:
      add PointsVowelSound points to intermediate
let prefix_distance equal the “edit distance” (see below) between the prefix of
a and the prefix of b
let prefix_max equal the total number of phonemes in the prefix of a + the
total number of phonemes in the prefix of b
let prefix_similarity equal prefix_distance / prefix_max
add prefix_similarity * PointsPrefix to intermediate
add PointsPrefix to Possible
compute suffix_distance, suffix_max, suffix_similarity in the same way
add suffix similarity * PointsSuffix to intermediate
add PointsSuffix to Possible
calculate the final score by dividing intermediate by Possible
```

This algorithm basically calculates the similarity between two syllables in each of their four categories, then computes a weighted average of those four similarities. For the vowel sound and stress, the similarity is binary – either two syllables have the same vowel sound / stress, or they do not. This could be improved by observing that some vowels sound more alike than others, and perhaps building a “similarity matrix” between vowel sounds, however that approach was not taken for this project. The similarity between the two prefixes and suffixes is not binary, but is again 'fuzzy' between 0 and 1 – here the edit distance is used (see 4.4).

The calculation of the weighted average is very important to this project, and deserves further discussion. Each component is given a weight, indicated by the variables PointsVowelSound, PointsVowelStress, PointsPrefix, and PointsSuffix. These are some of the system parameters (see 8) that are user configurable. The initial values are given in section 8, however these numbers represent the author's educated guesses, and can probably be improved (see 6.2).

4.4 Edit Distance

The “edit distance”, also known as the Levenshtein distance, is calculated as described in http://www.merriampark.com/ld.htm . The edit distance between two strings is the minimum number of characters insertions and deletions needed to convert the first string to the second (or vice-versa); for example, the edit distance between “ABC” and “ACD” is two: first we transform ABC to AC by deleting 'B', and then we transform AC to ACD by adding D. The maximum edit distance between two strings is the sum of the lengths of the strings – if the two strings have no characters in common, then every character in the first string must be deleted, and every character in the second string must be added. This observation motivates the method used to calculate prefix_similarity and suffix_similarity.

4.5 Using Syllable Pairs to Find Word Rhymes

Now that we have a method for determining the likelihood that two syllables rhyme, we can apply that method to pairs of syllables across the input text to find the most likely rhyming candidates. It is not obvious which pairs of syllables are most likely to rhyme, so as a first approximation we are going to look at all possible pairs of syllables in the provided text. We 'flatten' the list of syllables so the entire text is represented as single list of syllables; however, the mappings from the original words to their syllables are preserved for later use. This flattening process has the advantage that rhymes which span across words can be found by the system through the same method as finding single word rhymes.

This idea of comparing every pair of syllables can be improved by observing that rhyming words must have some proximity to each other in text; i.e. a rhyme usually does not span more than a few syllables. Thus, we restrict the search for a syllable that rhymes with the current syllable to only those other syllables withing some configurable 'radius'.

Once finding candidate pairs of syllables, the algorithm next tries to 'extend' a rhyme by looking at syllables to the left and right of the rhyming pair and searching for longer rhymes. The range of syllables that potentially rhyme is extended according to a configurable threshold. This algorithm returns these 'rhyme ranges' and their average (per syllable) rhyme score for post-processing. A summary of the algorithm:

```
initialize pair_list as an empty list
for each syllable in the master list of all syllables in the text
      find all syllables within SearchMaxDistance syllables after this one
      create a pair consisting of this syllable and each 'close' syllable and
      add to pair_list
for each pair of syllables in pair_list
      remove the pair from pair_list if the rhyme score between the two
      syllables is less than SearchBreadth
for each pair of syllables left in pair_list
      create a “rhyme range” object for this pair:
            extend the rhyme left (earlier in the text) for each neighboring
            syllable pair with a rhyme score greater than SearchDepth until the
            threshold is not met or we hit the beginning of the text
            extend the rhyme right using the same method
            compute the average score across all the rhyming syllables in the
            range
for each rhyme range object
      if the average score does not exceed SearchQuality, discard the rhyme
      range
return the remaining rhyme ranges for post-processing
```

Like the earlier algorithm, this step relies on several user-configurable constants: SearchMaxDistance, SearchBreadth, SearchDepth, and SearchQuality. The values chosen by the author for these constants are give in Section 8, but it is the author's belief that these numbers could be improved, see 6.2.

4.6 Post Processing

This method tends to find a large number of rhymes, several of which must be discarded. Two rules are applied to determine whether or not we should discard a rhyme:

- First, we discard all duplicate rhymes in the list – because of the way rhyme ranges are found, it is possible that multiple syllable rhymes can be repeated in the list.
- Second, we look at each rhyme to make sure the final syllable for each half of the rhyme ends at the end of a word. Because of the way English is spoken, with pauses between each word, we do not have a rhyme that ends 'inside' a word. We shorten rhymes until then end within a word, or discard them if it is not possible to end them within a word.

After these 'cleanup' steps, we look at the mappings from syllables to words that we saved earlier, and return to the user the actual English words that rhyme.

5 Results

5.1 Experimental Results

The original intent of this project was to apply this code to the extremely large Original Hip-Hop Lyrics Archive, at http://www.ohhla.com . This archive contains over 11,000 user-submitted song lyrics encompassing 42 MB of text. This is the highest-quality source of data available on rap lyrics, but it has a number of problems that made it impossible to use this data in this paper.

First, the format of the text files is not consistent. While most of the files have a common header indicating the song name, artist, and album, not all authors obey this header. A much larger problem is the way that sound effects, the name of the current rapper, the use of choruses or 'hooks', and other information about the song is embedded by the author within the lyrics. The many authors of these lyrics have chosen myriad systems for actually typing the body of the lyrics, which are usually discernible to a human reader, but very difficult for a computer to process. The author still believes that this corpus could be a valuable source of test data for this or other projects, but the text processing required to standardize their format is a project in and of itself.

Therefore, in order to evaluate the success of this program, the author hand-typed five lyrics files for various hip hop songs that have interesting rhyming patterns and structures. The disadvantage to this approach is that the rhyming system was tuned by the author to perform well on these examples; however, the author believes that the examples are broad enough in scope to be representative of the larger corpus of hip hop lyrics available.

The success of the system was measured for each file as the overlap between the rhymes observed by the author in a file, and the rhymes the system found in a file. In general, there are three cases:

- The author observes a rhyme in a file, and the system finds that rhyme as well. This is a success.
- The author observes a rhyme in a file, and the system does not detect that rhyme. This is a false negative.
- The system detects a rhyme in a file that was not observed by the author. This is a false positive. False positives can occur for two reasons – either the words do not rhyme, or the words do rhyme, but the rhyme is unintentional or incidental and does not contribute to the rhyming structure of the song.

On to the actual results. The table below presents each of the tested lyrics, the size of the lyrics file (measured in words and in syllables), the number of successes, false negatives, and false positives.

Song	*# Words*	*# Syllables*	*Successes*	*False Negatives*	*False Positives*
Sugarhill Gang - “Rapper's Delight” (fourth verse)	237	309	12	3	57
Run-D.M.C. - “It's Tricky” (entire song w/o choruses)	235	306	25	3	77
Beastie Boys - “Remote Control” (first verse)	85	112	6	4	13
Beastie Boys - “Intergalactic” (entire song w/o choruses)	374	465	40	7	87
MC Paul Barman - “The Joy of Your World” (first verse)	121	151	9	4	12

5.2 Shortcomings

Looking at even this small amount of data reveals the strengths and weaknesses of the system. The system is quite good at finding the rhymes that are in the song, but it the majority of the rhymes it finds are false positives. While the false positives were not explicitly classified, a cursory analysis reveals that most of them were “unintentional rhymes” -- words that rhymed, but did not contribute to the rhyming structure of the song.

There are a number of ways the system could be fixed to minimize this problem. First, the various configurable parameters could be adjusted by a smarter process than educated guessing. More importantly, the system should factor in more information about the lyrics. Line breaks and punctuation are disregarded by the system, but these provide clues as to where the rhymes will naturally occur in a song. Also, most of the actual rhymes occur at a regular period, starting at syllable zero. If this period can be found for a song, rhymes that are aperiodic can be discarded.

6 Conclusions

6.1 Summary

The system is certainly a prototype, but the results are encouraging. The algorithms for converting sounds to syllables, calculating rhyme scores, and finding and extending rhymes seem to be a solid basis. However, more needs to be done to post-process the found rhymes and eliminate spurious or accidental rhymes.

A more thorough test on a much larger number of lyrics is needed to confirm these findings, and will probably provide guidance as to more improvements.

6.2 Future Work

First, the rhyme system must be more tolerant of its input – this is a simple text processing task, but was deemed unimportant at this phase. Also, allowing more general input such as line breaks and punctuations gives the system hints as to where the intended rhymes are.

Next, the system needs to more aggressively filter out accidental rhymes. Some methods for doing this were suggested above.

The largest, and perhaps most valuable improvement would be to scientifically select the various constants used in the system (see 8). With a large, labeled set of test cases, the constants for determining syllabic rhymes and for finding larger rhymes could be experimentally determined through machine learning techniques that maximize the effectiveness of the system. Doing this requires much effort in obtaining standardized lyrics and having a human expert identify rhymes, but the author believes that this would pay off tremendously.

Finally, the system could be evaluated for its usefulness in other genres of rhyming text, such as non-hip-hop songs or even rhyming poems. We may find that the types of rhymes used in other genres behave differently than those used by rappers. It is the author's belief, however, that this system provides a solid fundamental start to any system that deals with rhyme detection in written text.

7 Appendix A: List of Phonemes in the CMU Dictionary

This table shows the 39 phonemes used in the CMU Dictionary. The 15 phonemes starting with A, E, I, O, and U are vowels with stressed, the rest are consonants (table from the CMU Dictionary website):

Phoneme	Example	Translation
AA	odd	AA D
AE	at	AE T
AH	hut	HH AH T
AO	ought	AO T
AW	cow	K AW
AY	hide	HH AY D
B	be	B IY
CH	cheese	CH IY Z
D	dee	D IY
DH	thee	DH IY
EH	Ed	EH D
ER	hurt	HH ER T
EY	ate	EY T
F	fee	F IY
G	green	G R IY N
HH	he	HH IY
IH	it	IH T
IY	eat	IY T
JH	gee	JH IY
K	key	K IY
L	lee	L IY
M	me	M IY
N	knee	N IY
NG	ping	P IH NG
OW	oat	OW T
OY	toy	T OY
P	pee	P IY
R	read	R IY D
S	sea	S IY
SH	she	SH IY
T	tea	T IY
TH	theta	TH EY T AH
UH	hood	HH UH D
UW	two	T UW
V	vee	V IY
W	we	W IY
Y	yield	Y IY L D
Z	zee	Z IY
ZH	seizure	S IY ZH ER

8 Appendix B: Configurable Parameters

All of these numbers can be adjusted by editing Constants.cs and recompiling the system. The numbers indicate the author's value for each.

- SearchBreadth = 0.3: Controls how many possible rhyme pairs the searcher will initially explore. A higher value will cause the search to go quicker, but may miss some possible rhymes. Ranges from 0 to 1.
- SearchDepth = 0.5: Controls how long the rhymes found will be. A lower value will result in longer rhymes being explored by the program. Ranges from 0 to 1.
- SearchQuality = 0.75: Controls the final quality of the rhymes kept. A higher value will result in fewer total rhymes returned. Ranges from 0 to 1.
- PointsVowelSound = 5: Controls how important the vowel sound is when determining if two syllables rhyme. Can be any positive integer
- PointsVowelStress = 1: Controls how important the stress of a syllable (its emphasis) is when determining if syllables rhyme. Can be any positive integer.
- PointsSuffix = 7: Controls how important the consonants after the vowel are when determining if two syllables rhyme. Can be any integer greater than 0.
- PointsPrefix = 3: Controls how important the consonants before the vowel are when determining if two syllables rhyme. Can be any integer greater than 0.
- SearchMaxDistance = 25: Controls the maximum length that we will look for rhyming syllables. Can be any integer greater than 0.