Jon Lesser
ANLP
December 13, 2006

Final Project: Improving RSVP with NLP

Directory Structure

src

– Contains my two python scripts, two pickle files generated by “rsvp.py”, “huckfinn.txt”, and the urwid folder, which is a library I use for display.

Writeup

– Contains this writeup as well as my presentation slides in PDF format.

Running the application

Rsvp.py generates two pickles that are read by flasher.py. To see a demo, use the following command:

```
$ python flasher.py –v True –b 700
```

The two arguments are optional. –v True tells the application to display some extra numbers. –b 700 tells the application to 700 milliseconds as the base time.

Summary

Rapid serial visual presentation (RSVP) is a technique for displaying words or images sequentially.[1] For my project, I researched existing usage of RSVP for small screen, mobile devices, and then built a working RSVP application. My application parses a text file, tokenizes the file, time weights the tokens, and then displays the tokens in a terminal.

A standard RSVP system displays each text chunk for an equal amount of time. My application code uses Natural Language Processing techniques to modulate the time each text chunk appears. I believe the time modulated display of text chunks is more pleasant to read than a standard RSVP system.

Concept Development and Scope Refinement

A Technology Review article originally seeded the idea for this project. The article discussed the iCue product, an RSVP system for cell phones that integrates with a proprietary digital bookstore.[2] Technology Review compared the iCue to a tachistoscope[3], but failed to discuss relevant work under the RSVP keyword. In searching

[1] K. I. Forster, “Visual perception of rapidly presented word sequences of varying complexity,” Percept. Psychophys. 8, 215 – 221 (1970).
[2] See http://www.i-cue.co.uk
[3] See http://en.wikipedia.org/wiki/Tachistoscope

for relevant academics articles, I was pleasantly surprised to find that several researchers have already done work with RSVP applied specifically to mobile devices.[4][5][6]

My original idea was to map the cadence of reading aloud to RSVP. The literature I read seemed to cast doubt that such a mapping would work well. However, I think it would be very interesting to analyze recordings of people reading aloud and then build models that capture their cadence. These models could then be used to modulate presentation time in an RSVP system in such a way as to simulate a specific person's reading style. Audio books read by a single voice actor seems like a natural choice for this type of experiment. Given the complexities of audio signal processing and my limited time, I chose to investigate textual NLP techniques with the hope of achieving results similar to those I envisioned in my original idea.

The fact that Python programs can be run on many mobile phones, including my Nokia 7610, led me to consider using my own phone as a development platform. After all, one of the primary benefits of RSVP is its efficiency for text display on small screens. My own survey of RSVP software for mobile devices turned up an open-source Python package called GutenFlash.[7] GutenFlash was intended for use on linux-based pocket PCs and utilizes the pyGTK library for its user interface. The convenience of testing on my laptop, rather than having to continually push code to my mobile device, convinced me to use gutenFlash for this project.

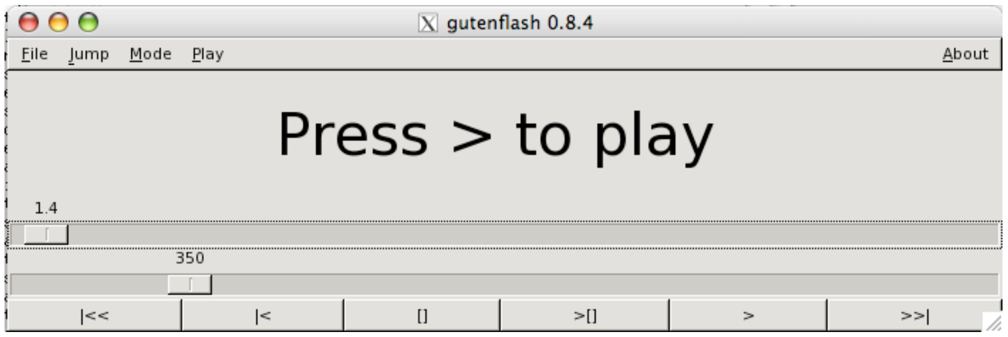
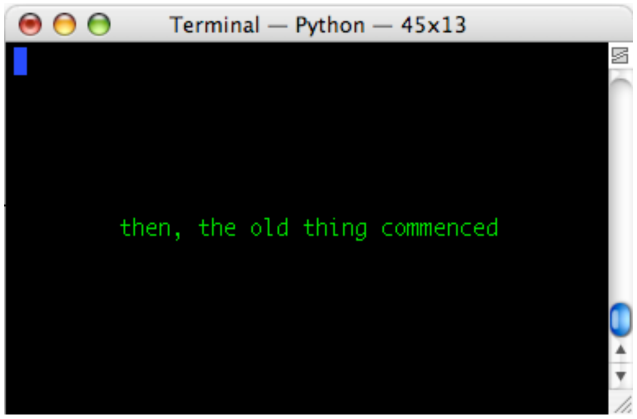


Figure 1. gutenFlash interface. Top, horizontal scroll bar represents location within the text. The bottom, horizontal scroll bar allow you to adjust the display speed in words per minute.

After a few hours of experimenting with gutenFlash, I began investigating alternative display methods. The mechanisms with which gutenFlash tokenizes text and calculates

[4] Öquist, G. and Goldstein, M. (2003). Towards an improved readability on mobile devices: evaluating adaptive rapid serial visual presentation. Interacting with Computers, 15(4), 539-558.
[5] Öquist, G., Bjo¨rk, S., and Goldstein, M. (2002). Utilizing Gaze Detection to Simulate the Affordances of Paper in the Rapid Serial Visual Presentation Format. F. Paterno` (Ed.): Mobile HCI 2002, LNCS 2411, pp. 378–382, 2002.
[6] Castelhano, M.S. and Muter, P. (2001). Optimizing the reading of electronic text using rapid serial visual presentation. Behavior & Information Technology, 2001, 20(4), 237-247.
[7] GutenFlash was written by Scott Scriven and released under the GNU General Public License.

display timings turned out to be deeply woven into the application logic. In the end, I switched to a terminal-based solution based on Curses. This solution simply centers text chunks within a resizable terminal window.



```
then, the old thing commenced
```

Figure 2. I used the Urwin library as a wrapper for the Python Curses library. (Ironic text is actually from Huck Finn.)

Application Logic Analysis

With a simple interface in place, I moved onto the coding the application logic. The following diagram shows the steps I implemented.

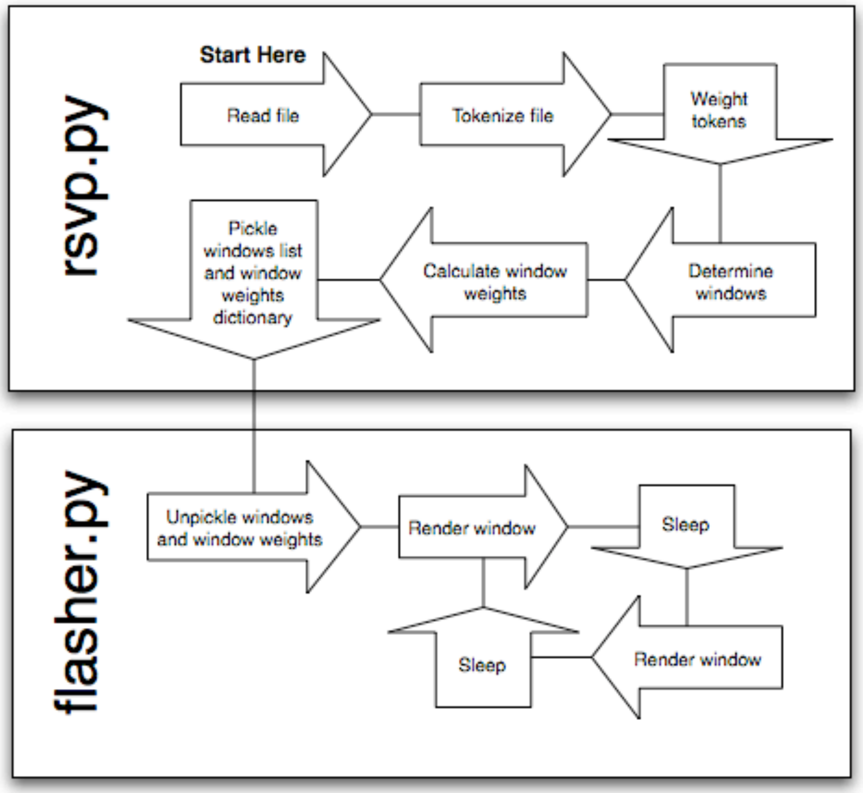


Figure 3. Process flow diagram

At the core of my design are two data structures: *rawTokens*, a list for storing the tokens of the input text in proper sequence, and *tokenWeights*, a dictionary keyed with entries from the *rawTokens* list, used to store weights.[8] With this two-structure method, you have word sequence maintained with the list along with the speed and space benefits of dictionaries. The downside is that there is only one weight for all like tokens. For example, all instances of the token "horseshoe" must carry the same weight because there can be only one dictionary entry keyed to "horseshoe" (Note that "horseshoe," would be a different token and could carry a different weight).

I can think of at least two alternative methods to store tokens and weights. The first would be to use a conditional frequency distribution. In this case, the key would still be the token, but multiple weights, perhaps representing the token in difference contexts, could be stored. The actual weight used could be a specific sample, a random sample, or an average of all the samples. A second alternative method would be to use a list of tuples. In this case, each element of the list would store both the token and a weight allowing for the finest possible weighting granularity.

The method I implemented puts a lot of emphasis on tokenizing because the more diverse your token set, the more control you can have over weighting. As it turned out, I was not very happy with my tokenizing method. While I wrote regular expressions to pull out things like proper names, I could not give special weighting those things because I could not distinguish them in the list returned by the tokenize function. A more sophisticated regular expression tokenizer could be written to add "tags" to the tokens it finds thus allowing weighting based on a sort of token type such as date, name, et cetera.

Interesting problems present themselves when you start to think about weighting not just tokens, but specific sequences of tokens. My approach was to focus on weighting individual tokens, and then when building windows with multiple tokens I summed the weights of all the tokens in the window and added in a constant weight for each token.[9] Consequently, I am not paying any attention to the wealth of weighting intelligence one could gather from the sequencing of tokens. An alternative approach to my own might start by parsing text directly into windows and then running a weighting analysis on the window. On the one hand, the larger unit of analysis seems likely to lead to better weighting schemes. However, I opted against this approach because it seemed like a logical first step of window analysis would be to break it into tokens anyways.

[8] For this project, weights are represented as milliseconds added to the display time of a given token. For example, a heavily weighted token may receive a weight of 500, which would equate to an additional half second of display.

[9] For this project, a window is the sequence of tokens that is displayed at one time. I define a variable character limit for the window and then concatenate tokens until that character limit is reached. 25 proved to be a nice window size. Alternative to windows include showing individual tokens, sentences clauses, or sentences.

Feature Analysis

It was wisely suggested to me to investigate n-grams for this project. Not so wisely, I did not implement any sort of n-gram analysis due to time constraints, though I gave it quite a bit of thought. There are two important things one can do with n-grams: weight and group. One would likely want to decrease the weighting around terms that frequently appear in sequence and conversely increase the weighting of rare sequences. For grouping, one would likely want to display common n-grams together in a window context. Groups could be made either by combining tokens or developing logic to make sure a given set of tokens appear in the same window.

The following table describes the features I actually implemented:

Feature	**Description**
Syllables Weighting	Token weight increased by 50 for each syllable over one. For example, the weight of a token with three syllables would be increased by 100.
Punctuation Weighting	If a token ended with one of my enumerated punctuation marks, it received an additional weight of 250.
Segmentation Weighting	If a token was equivalent to one of my enumerated segmentation words, it received an additional weight of 150.
Frequency Weighting	Tokens receive weighting by the following formula: `log( Term Frequency )^2` In effect, infrequent words receive more weight than frequent words.

A useful reference for evaluating the weight applied to tokens is typical reading speed. On average, people read 250 to 350 words per minute. That means on average people spend 170 to 240 milliseconds on each word. For this project, I chose to use absolute weights. I found the absolute values easier to interpret and subsequently easier to adjust. If I were developing my system further, I would convert the weights to relative values that could better adapt to changes display speed.

Evaluation of individual features is very difficult because only the sum of the feature weights is recorded in the weight dictionaries. So while I am happy with the net effect of all the weighting, it is not clear how I can optimize the individual features given my weighting implementation. The feature that gives the largest weight is punctuation. This feels right based on my own reading experience during which I spend the most time lingering around punctuation.

It is not clear to me whether incorporating statistics gathered from a larger corpus would benefit token weighting for a given document. For example, in the frequency feature I am only comparing the frequency of terms within a given document. With my current approach, a frequent token in the given document will receive relatively little weight, regardless of how infrequently that token may appear in a larger training corpus. I believe my current approach yield the correct outcome in this case because you are likely to recognize readily frequent tokens in a given document, even if they are infrequent outside of that document. However, in the case of infrequent tokens within a given document, I have no idea if those tokens represent rare words that a reader may need a little more time with. Here is does seems to make sense to incorporate data from outside the document to help determine the appropriate weight for a token. An n-gram feature would face a situation similar to the one I just described.

Conclusions and future work

I consider this project to be a moderate success. I was able write some code that runs (at least on my machine) and spend a lot of time thinking about what I think is an interesting problem. At the moment, I do not have any intentions to carry on with this project.

There are a number of worthwhile avenues for future work in improving RSVP with NLP. I have identified a number of such avenues throughout this report. I will add that without a more robust testing framework it will be difficult to assess future work just as it was difficult to assess my own results.