COSC 73: Computational Linguistics

Fall 2014, Dartmouth College

Twitter Bot Final Project

---

pome

[*sic*]

Authors:
Ben Packer
Byrne Hollander

Professor:
Sravana Reddy

November 25, 2014

Contents

1 Description of the motivating problem

1.1 Goal

The goal of the project was the make a Twitter bot capable of composing poems made up of Tweets related to a particular subject. The result should be a combination of existing Tweets into a couplet, where the two lines have the same number of syllables and rhyme. The program interface is used entirely through Twitter a Twitter user may tweet `@PomeSic -query ''query''`, where the "query" is in quotes and specified by a flag. Ideally, the poem that our program composed should both satisfy the basic poem requirements and be a small representation of what the Twitter sphere thinks about the chosen topic at the given time. This last specification, that the poem should not only be coherent but reflective of the Twitter discourse about a particular subject, has led us to prioritize algorithms that create a more native Twitter feel, including the use of Twitter vocabulary and Twitter-specific context similarity.

1.2 The Poem Composition Process

Since the goal of the project is to compose a functional poem of existing Tweets, we did not have to come up with a mechanism of generating text from scratch. Instead, our approach to the problem was to get existing Tweets that could be combined to make poems, and then modify the Tweets so that they met our poetic specifications. The overall process of selecting Tweets and then modifying them is presented below:

1. Find "similar" pairs of Tweets
2. Normalize the Tweets.
3. Make the Tweets rhyme.
4. Make the Tweets the same number of syllables.
5. "De-normalize" the Tweets.

1.3 Description of Our Program

First, our program checks Twitter to see if a new request has tweeted @PomeSic. Then, it checks the tweets arguments to see if it has enough information in the proper format to compose a poem. Next, it composes the poem, and tweets it back at the user. The interaction with Twitter is handled through the Tweepy API[1]. We use the `api.mentions` method to get tweets @PomeSic, the `api.search` method to find Tweets about the user-specified subject, and the `api.update_status` to Tweet the poem back at the user. Those parts are simple and uninteresting, so lets turn to creating the poem.

[1] `http://docs.tweepy.org/en/v2.3.0/`

2 A brief survey of related research, citations to the implemented algorithms, and description of similar existing products

2.1 Brief survey of related research

Lorem ipsum dolor sit amet, consectetuer adipiscing elit.

2.2 Citations to the implemented algorithms

2.2.1 "Similarity" - *word2vec*

Whenever we use the word "similarity" in this paper to describe the "similarity" between two words, the metric we have decided to use is the cosine similarity of context vectors using a skip-gram context model. We used the freely available software *word2vec*[2] in order to compute the context vectors, and the Python package Gensims *word2vec* interface[3] for computing the cosine similarity. Although we initially encountered long loading times (around 3.5 minutes) when trying to load the context vectors into Gensim, after we recomputed the context vectors and stored them in binary, the loading time went down to a little under 30 seconds. While *word2vec* gives the user the option to use either a skip-gram model or a continuous bag-of-words model, we decided to use a skip-gram model based on the *Efficient Estimation of Word Representations in Vector Space* paper[4]. In the paper, the authors demonstrate that the skip-gram model outperforms a continuous bag of words model on its semantic disambiguation abilities, while a continuous bag of words model better reproduces syntactic similarity. Since the syntactic requirements of Twitter are loose already and we are primarily interested in this as a measure of semantic similarity, we decided to use the skip-gram model.

[2] `http://code.google.com/p/word2vec/`
[3] `http://radimrehurek.com/gensim/models/word2vec.html`
[4] `http://arxiv.org/pdf/1301.3781.pdf`

2.2.2 Find "Similar" Pairs of Tweets

Our first idea was to treat all of the Tweets we downloaded as sections of one corpus, and then to make bag of words vectors for each Tweet and compare their similarity. However, we quickly realized that this algorithm was functionally equivalent to a word overlap algorithm, in that it would rank tweets by how many words in the Tweet overlapped. Not only did this metric seem silly, as it constantly produced pairs of Tweets like:

tweet1: `Riches celebrity houses are incredible!!! RT!! (url)`
tweet2: `incredible celebrity houses (url)`

we realized that a poem is more interesting when its words do not overlap, but they are similar. As a result, we came up with the following easy metric for string similarity:

`similarity(tweet1, tweet2)` = mean of all mean similarities between word1, word2, for word1 in tweet1, for word2 in tweet2

with an additional penalty for words being the same. Specifically, if two words are the same, we treat their cosine similarity as 0 (even though it is actually 1) in order to discourage Tweet pairs from having too many overlapping words. It is inevitable that Tweets will have at least one overlapping word because they all came from a single query, but this will ensure that their overlap is limited.

2.2.3 Normalize the Tweets

Although we tried to construct the bot without relying too heavily on normalization, it became a necessity for Phonetic reasons, which will be elaborated more on later. There are several different approaches to normalization that we found in the literature. First, many authors treat the problem as one of machine translation, in which the goal is to match the words used in Twitter to the words used in Standard

English. We found a dictionary here[5] that had been constructed using by comparing contexts of similar words and then ranking the contextually similar words by string similarity, as described here[6]. However, the use of this dictionary had severe limitations, as most possible representations of a word were not present. For example, although there was an entry comparing "`ggoooooddd`" to "`good`", there was no similar entry for "`ggoood`". As a result of the small fraction of out-of-vocabulary words represented in this dictionary, we attempted to reconstruct the normalization scheme used to create that dictionary, which was not too difficult. We finally settled on a six-step normalization process:

1. Remove punctuation
2. If it is a number, return a textual representation of the number for our phonetic dictionary
3. If the word is in the dictionary, it is already normalized; return it
4. Use the *word2vec* context vectors to get words that occur in similar contexts, filter those results to only contextual similar words that are "normal" already, and then select
5. among those words based on string edit distance
6. If everything else has failed, simply spell-check it using edit distance
7. If everything fails, return the word un-normalized

This normalization scheme was not without its flaws, however. It had the least success on common abbreviations like "`lol`" or news stations like "`ABC`", once correcting the latter to "`feud`". However, this was the best that we could do given our limited time and resources. In the future, a major improvement to this program would be the creation of a method for getting the phonemic representation of abbreviations ("`Ay Bee See`") so that we could use them more effectively in the poem not normalized. Furthermore, the default spell checking feature was used less often than we

[5]`https://sites.google.com/a/student.unimelb.edu.au/hanb/research`
[6]`http://aclweb.org/anthology/D/D12/D12-1039.pdf`

had hoped due to the very large frequency of OOV words. For example, when trying to use the context similarity method to find the normalized version of "`goodd`", the following contextually similar possibilities were generated:

```
(u'goood', 0.8947709798812866)
(u'nicee', 0.7713996171951294)
(u'sweeet', 0.7375402450561523)
(u'gewd', 0.7344779968261719)
(u'gooood', 0.7263408899307251)
(u'guud', 0.726222813129425)
(u'siick', 0.7210522294044495)
(u'greatt', 0.7182121276855469)
(u'baad', 0.7177780270576477)
(u'amazingg', 0.7133066654205322)
```

unfortunately none of which are already in Standard English. It appears that on Twitter the word "`good`" is used so infrequently compared to its misspelled counterparts that this method achieve very little in this case. Although the spell checker did a fine job here, the limits to the context method should be noted.

2.2.4 Make the Tweets Rhyme

Making two tweets rhyme is as simple as making their two final words rhyme. In order to make their two final words rhyme, our algorithm considers two options: first, making the last word of the first tweet rhyme with the last word of the second, or second, making the last word of the second tweet rhyme with the last word of the first. This was simple enough - in order to make word x rhyme with word y, we generated all possible rhymes with word y, and ranked them by contextual similarity to word x. After doing the inverse operation (making word y rhyme with word x) we selected among the two options based on the contextual similarity between the original word and its replacement. Unfortunately due to the limits of our phonetic abilities, to be elaborated on below, we also looked for candidate replacements among words present in our phonetic dictionary.

Another decision in this category is how to determine if two words rhymed. Initially, we decided that if the final two phonemes of a word were the same, the words rhymed. However, this method decided that the words "`wants`" and "`expects`" rhymed because of their similar last two phonemes. As a result, we decided that word a rhymes with word b if all phonemes from the final stressed vowel sound of a until the end of the word are the same as the phonemes of the final stressed vowel sound of b until the end of the word, which produced much better results.

2.2.5 Making the Tweets the Same Number of Syllables

In order to make the tweets the same number of syllables, we repeatedly replaced a word in the first or second tweet by a similar word with one more or one less syllable until the number of syllables in the tweets matched. Here is the pseudocode:

```
Suppose tweet1 has 4 syllables and tweet2 has 6
While tweet1 and tweet2 are not the same number of syllables:
    Cost of subtracting 1 syllable from tweet2 = largest cosine similarity
        between a word in tweet2 and a substitution with 1 less syllable
    Cost of adding 1 syllable to tweet1 = largest cosine similarity between a
        word in tweet1 and substitution with 1 more syllables
    Perform less costly operation (whichever has highest cosine similarity)
End
```

2.2.6 De-normalize the tweets

Finally, in order to make the poem have that Twitter feeling to it, we undo all normalizations that we can. This means that if a word was normalized because it needed to be a candidate for either rhyming changes or syllable substitutions, but it was not replaced with a different word, we replace it with its pre-normalized form.

2.3 Description of similar existing products

Several similar Twitter bots exist now. One of them is @Pentametron which "looks for poetry in everday musings." It retweets tweets that are written in perfect iambic pentameter and even posts "unintentional sonnets" on a website associated with the account. There is also an accidental haiku bot (@accidental575) that finds tweets in the 5-7-5 haiku form and tweets them formatted and with attribution.

3 User feedback

Phasellus viverra nulla ut metus varius laoreet. Quisque rutrum.

4 Analysis of the program's shortcomings and thoughts on future development

4.1 Analysis of the program's shortcomings

Despite many attempts to install several different code packages that would allow us to get phonemic realizations of out of vocabulary words, we were unable to install one. We tried to install Sequitur G2P[7], but there was a problem with a deprecated version of a Numpy API that was not fixed by upgrading Numpy. The only suggestions on the internet were to uncomment the line of C code that specified which Numpy API version to use, but that did not work. We tried to install DirecTL+[8] but could not install one of the dependencies STLport.

4.2 Thoughts on future development

Lorem ipsum dolor sit amet, consectetuer adipiscing elit.

[7] http://www-i6.informatik.rwth-aachen.de/web/Software/g2p.html
[8] https://code.google.com/p/directl-p/

5 Project Links

Link to Github page	`http://github.com/byrnehollander/pomesic`
Link to Twitter page	`http://www.twitter.com/pomesic`