Generating Topical Poetry

Marjan Ghazvininejad†, Xing Shi†, Yejin Choi‡, and Kevin Knight†
†Information Sciences Institute & Computer Science Department
University of Southern California
{ghazvini,xingshi,knight}@isi.edu
‡Computer Science & Engineering, University of Washington
yejin@cs.washington.edu

Abstract

We describe Hafez, a program that generates any number of distinct poems on a user-supplied topic. Poems obey rhythmic and rhyme constraints. We describe the poetry-generation algorithm, give experimental data concerning its parameters, and show its generality with respect to language and poetic form.

1 Introduction

Automatic algorithms are starting to generate interesting, creative text, as evidenced by recent distinguishability tests that ask whether a given story, poem, or song was written by a human or a computer. In this paper, we describe Hafez, a program that generates any number of distinct poems on a user-supplied topic. Figure 1 shows an overview of the system, which sets out these tasks:

- Vocabulary. We select a specific, large vocabulary of words for use in our generator, and we compute stress patterns for each word.
- Related words. Given a user-supplied topic, we compute a large set of related words.
- Rhyme words. From the set of related words, we select pairs of rhyming words to end lines.
- Finite-state acceptor (FSA). We build an FSA with a path for every conceivable sequence of vocabulary words that obeys formal rhythm constraints, with chosen rhyme words in place.
- Path extraction. We select a fluent path through the FSA, using a recurrent neural network (RNN) for scoring.

Figure 1: Overview of Hafez converting a user-supplied topic word (wedding) into a four-line iambic pentameter stanza.

---

1For example, in the 2016 Dartmouth test bit.ly/20WGLF3, no automatic sonnet-writing system passed indistinguishability, though ours was selected as the best of the submitted systems.
Sections 3-7 describe how we address these tasks. After this, we show results of Hafez generating 14-line classical sonnets with rhyme scheme ABAB CDCD EF EF GG, written in iambic pentameter (ten syllables per line with alternating stress: “da-DUM da-DUM da-DUM . . .”). We then show experiments on Hafez’s parameters and conclude by showing the generality of the approach with respect to language and poetic form.

2 Prior Work

Automated poem generation has been a popular but challenging research topic (Manurung et al., 2000; Gervas, 2001; Diaz-Agudo et al., 2002; Manurung, 2003; Wong and Chun, 2008; Jiang and Zhou, 2008; Netzer et al., 2009). Recent work attempts to solve this problem by applying grammatical and semantic templates (Oliveira, 2009; Oliveira, 2012), or by modeling the task as statistical machine translation, in which each line is a “translation” of the previous line (Zhou et al., 2009; He et al., 2012). Yan et al. (2013) proposes a method based on summarization techniques for poem generation, retrieving candidate sentences from a large corpus of poems based on a user’s query and clustering the constituent terms, summarizing each cluster into a line of a poem. Greene et al. (2010) use unsupervised learning to estimate the stress patterns of words in a poetry corpus, then use these in a finite-state network to generate short English love poems.

Several deep learning methods have recently been proposed for generating poems. Zhang and Lapata (2014) use an RNN model to generate 4-line Chinese poems. They force the decoder to rhyme the second and fourth lines, trusting the RNN to control rhythm. Yi et al. (2016) also propose an attention-based bidirectional RNN model for generating 4-line Chinese poems. The only such work which tries to generate longer poems is from Wang et al. (2016), who use an attention-based LSTM model for generation iambic poems. They train on a small dataset and do not use an explicit system for constraining rhythm and rhyme in the poem.

Novel contributions of our work are:

- We combine finite-state machinery with deep learning, guaranteeing formal correctness of our poems, while gaining coherence of long-distance RNNs.
- By using words related to the user’s topic as rhyme words, we design a system that can generate poems with topical coherence. This allows us to generate longer topical poems.
- We extend our method to other poetry formats and languages.

3 Vocabulary

To generate a line of iambic pentameter poetry, we arrange words to form a sequence of ten syllables alternating between stressed and unstressed. For example:

```
010 1 0 10 101
```

Attending on his golden pilgrimage

Following Ghazvinniejad and Knight (2015), we refer to unstressed syllables with 0 and stressed syllables with 1, so that the form of a Shakespearean sonnet is 

\[
((01)^5)^7\]

To get stress patterns for individual words, we use CMU pronunciation dictionary,\(^2\) collapsing primary and secondary stresses. For example:

```
CAFETERIA K AE2 F AH0 T IH1 R IY0 AH0
```

becomes

```
CAFETERIA 10100
```

The first two columns of Table 1 show other examples. From the 125,074 CMU dictionary word types, we can actually only use words whose stress pattern matches the iambic pattern (alternating 1s and 0s). However, we make an exception for words that end in ...100 (such as spatula). To mimic how human poets employ such words, we convert all “...100” patterns to “...101”. This leaves us with a 106,019 word types.

Words with multiple syllable-stress patterns present a challenge. For example, our program may use the word record in a “...10...” context, but if it is a verb in that context, a human reader will pronounce it as “01”, breaking the intended rhythm. To guarantee that our poems scan properly, we eject all ambiguous words from our vocabulary. This problem is especially acute with monosyllabic words, as most have a stress that depends on context. Greene et al. (2010) apply the EM algorithm to align

\(^2\)http://www.speech.cs.cmu.edu/cgi-bin/cmudict
human-written sonnets with assumed meter, extracting \( P(0|\text{word}) \) and \( P(1|\text{word}) \) probabilities. Using their method, we eject all monosyllabic words except those with \( P(0|\text{word}) > 0.9 \) or \( P(1|\text{word}) > 0.9 \). A consequence is that our poetry generator avoids the words to, it, in, and is, which actually forces the system into novel territory. This yields 16,139 monosyllabic and 87,282 multisyllabic words.

Because our fluency module (Section 7) is restricted to 20,000 word types, we further pare down our vocabulary by removing words that are not found in the 20k-most-frequent list derived from the song lyrics corpus we use for fluency. After this step, our final vocabulary contains 14,368 words (4833 monosyllabic and 9535 multisyllabic).

### 4 Topically Related Words and Phrases

After we receive a user-supplied topic, the first step in our poem generation algorithm is to build a scored list of 1000 words/phrases that are related to that topic. For example:

- **User-supplied input topic:** colonel
- **Output:** colonel (1.00), lieutenant_colonel (0.77), brigadier_general (0.73), commander (0.67) ... army (0.55) ...

This problem is different from finding synonyms or hypernyms in WordNet (Miller, 1995). For example, while Banerjee and Pedersen (2003) use WordNet to assign a 1.0 similarity score between car and automobile, they only give a 0.3 similarity between car and gasoline.

A second method is to use pointwise mutual information (PMI). Let \( t \) be the topic/phrase, and let \( w \) be a candidate related word. We collect a set of sentences \( S \) that contain \( t \), and sort candidates by

\[
P(w) \text{ in general text} \qquad \text{Proportion of sentences in } S \text{ containing } w
\]

Table 2 shows that PMI has a tendency to assign a high score to low frequency words (Bouma, 2009; Role and Nadif, 2011; Damani, 2013).

A third method is word2vec (Mikolov et al., 2013a), which provides distributed word representations. We train a continuous-bag-of-words model\(^3\) with window size 8 and 40 and word vector dimension 200. We score candidate related words/phrases with cosine to topic-word vector. We find that a larger window size works best (Pennington et al., 2014; Levy and Goldberg, 2014).

Table 2 shows examples. The training corpus for word2vec has a crucial effect on the quality of the related words. We train word2vec models on the English Gigaword corpus,\(^4\) a song lyrics corpus, and the first billion characters from Wikipedia.\(^5\) The Gigaword corpus produces related words that are too newsy, while the song lyrics corpus does not cover enough topics. Hence, we train on Wikipedia. To obtain related phrases as well as words, we apply the method of Mikolov et al. (2013b) to the Wikipedia corpus, which replaces collocations like Los Angeles with single tokens like Los_Angles. Word2vec then builds vectors for phrases as well as words. When the user supplies a multi-word topic, we use its phrase vector if available. Otherwise, we create the vector topic by element wise addition of its words’ vectors.

### 5 Choosing Rhyme Words

We next fill in the right-hand edge of our poem by selecting pairs of rhyming words/phrases and assigning them to lines. In a Shakespearean sonnet with rhyme scheme ABAB CDCD EFEF GG, there are seven pairs of rhyme words to decide on.

#### 5.1 Strict Rhyme

The strict definition of English rhyme is that the sounds of two words must match from the last stressed vowel onwards. In a masculine rhyme,
the last syllable is stressed; in a feminine rhyme, the penultimate syllable is stressed. We collect phoneme and stress information from the CMU pronunciation dictionary. We pre-compute strict rhyme classes for words (see Table 1) and hash the vocabulary into those classes.

5.2 Slant Rhyme

In practice, human poets do not always use strict rhymes. To give ourselves more flexibility in choosing rhyme pairs, we allow for slant (or half) rhymes. By inspecting human rhyming data, we develop this operational definition of slant rhyme:

1. Let s1 and s2 be two potentially-rhyming phoneme sequences.
2. Replace ER with UHR in both sequences.
3. Let v1 and v2 be the last stressed vowels in s1 and s2.
4. Let w1 and w2 be last vowels in s1 and s2.
5. Let s1 = a1 v1 x1 w1 c1. Likewise, let s2 = a2 v2 x2 w2 c2.
6. Output NO under any of these circumstances:
   (a) v1 \neq v2, (b) w1 \neq w2, (c) c1 \neq c2, (d) a1 \neq NULL and a2 \neq NULL and a1 = a2.
7. If x1 and x2 are single phonemes:
   (a) If x1 \sim x2, then output YES.\footnote{x \sim y if phonemes x and y are similar. Two phonemes are similar if their pairwise score according to (Hirjee and Brown, 2010) is greater than -0.6. This includes 98 pairs, such as L/R, S/SH, and OY/UH.}
   (b) Otherwise, output NO.
8. If x1 and x2 contain different numbers of vowels, output NO.
9. Let p1 and q1 be the first and last phonemes of x1. Let p2 and q2 be the same for x2.
10. If (p1 = p2) and (q1 \sim q2), output YES.
11. If (p1 \sim p2) and (q1 = q1), output YES.
12. Otherwise, output NO.

Words whose last syllable is stressed do not participate in slant rhymes.

Example slant rhymes taken from our generated poems include Viking/fighting, snoopy/spooky, baby/crazy and comic/ironic. We pre-compute a coarse version of slant rhyme classes (Table 1) with the pattern “v_i * w_i c_i”. If two words hash to the same coarse class, then we subsequently accept or reject depending on the similarity of the intermediate phonemes.

5.3 Non-Topical Rhyming Words

For rare topics, we may not have enough related words to locate seven rhyming pairs. For example, we generate 1000 related words for the topic Viking, but only 32 of them are found in our 14,368-word vocabulary. To give a chance for all topical words/phrases to be used as rhyme words, for each strict rhyme class, we add the most common word in our song lyric corpus to the list of related words. In addition, we add words from popular rhyme pairs (like do/you and go/know) to the list of related words with a low topic similarity score.

5.4 Rhyme word selection

We first hash all related words/phrases into rhyme classes. Each collision generates a candidate rhyme pair (s1, s2), which we score with the maximum of cosine(s1, topic) and cosine(s2, topic). So that we can generate many different sonnets on the same topic, we choose rhyme pairs randomly with probability proportional to their score. After choosing a pair (s1, s2), we remove it, along with any other candidate pair that contains s1 or s2. Because a poem’s beginning and ending are more important, we assign the first rhyme pair to the last two lines of the sonnet.

\footnote{http://slate.me/OhTKCA}
then assign other pairs from beginning of the sonnet towards the end.

6 Constructing FSA of Possible Poems

After choosing rhyme words, we create a large finite-state acceptor (FSA) that compactly encodes all word sequences that use these rhyme words and also obey formal sonnet constraints:

- Each sonnet contains 14 lines.
- Lines are in iambic pentameter, with stress pattern (01)$^5$. Following poetic convention, we also use (01)$^5$0, allowing feminine rhyming.
- Each line ends with the chosen rhyme word/phrase for that line.
- Each line is punctuated with comma or period, except for the 4th, 8th, 12th, and 14th lines, which are punctuated with period.

To implement these constraints, we create FSA states that record line number and syllable count. For example, FSA state L2-S3 (Figure 2) signifies “I am in line 2, and I have seen 3 syllables so far”. From each state, we create arcs for each feasible word in the vocabulary. For example, we can move from state L1-S1 to state L1-S3 by consuming any word with stress pattern 10 (such as table or active). When moving between lines (e.g., from L1-S10 to L2-S1), we employ arcs labeled with punctuation marks.

To fix the rhyme words at the end of each line, we delete all arcs pointing to the line-final state, except for the arc labeled with the chosen rhyme word. For speed, we pre-compute the entire FSA; once we receive the topic and choose rhyme words, we only need to carry out the deletion step.

In the resulting FSA, each path is formally a sonnet. However, most of the paths through the FSA are meaningless. One FSA generated from the topic natural language contains $10^{229}$ paths, including this randomly-selected one:

- Of pocket solace ammunition grammar.
- An tile pretenders spreading logical.
- An stories Jackie gallon posing banner.
- An corpses Kato biological ...

Hence, we need a way to search and rank this large space.

7 Path extraction through FSA with RNN

To locate fluent paths, we need a scoring function and a search procedure. For example, we can build a n-gram word language model (LM)—itself a large weighted FSA. Then we can take a weighted intersection of our two FSAs and return the highest-scoring path. While this can be done efficiently with dynamic programming, we find that n-gram models have a limited attention span, yielding poor poetry.

Instead, we use an RNN language model (LM). We collect 94,882 English songs (32m word tokens) as our training corpus,\(^8\) and train\(^9\) a two-layer recurrent network with long short-term memory (LSTM) units (Hochreiter and Schmidhuber, 1997).\(^10\)

When decoding with the LM, we employ a beam

---

\(^8\)http://www.mldb.org/
\(^9\)We use the toolkit: https://github.com/isi-nlp/Zoph_RNN
\(^10\)We use a minibatch of 128, a hidden state size of 1000, and a dropout rate of 0.2. The output vocabulary size is 20,000. The learning rate is initially set as 0.7 and starts to decay by 0.83 once the perplexity on a development set starts to increase. All parameters are initialized within range $[-0.08, +0.08]$, and the gradients are re-scaled when the global norm is larger than 5.
search that is further guided by the FSA. Each beam state $C_{t,i}$ is a tuple of $(h, s, word, score)$, where $h$ is the hidden states of LSTM at step $t$ in $i$th state, and $s$ is the FSA state at step $t$ in $i$th state. The model generates one word at each step.

At the beginning, $h_{0,0}$ is the initial hidden state of LSTM, $s_{0,0}$ is the start state of FSA, $word_{0,0} = $<START>$ and score_{0,0} = 0$. To expand a beam state $C_{t,i}$, we first feed $h_{t,i}$ and $word$ into the LM and get an updated hidden state $h_{next}$. The LM also returns a probability distribution $P(V)$ over the entire vocabulary $V$ for next word. Then, for each succeeding state $s_{suc}$ of $s_{t,i}$ in the FSA and the word $w_{next}$ over each edge from $s_{t,i}$ to $s_{suc}$, we form a new state $(h_{next}, s_{suc}, w_{next}, score_{t,i} + log(P(w_{next})))$ and push it into next beam.

Because we fix the rhyme word at the end of each line, when we expand the beam states immediately before the rhyme word, the FSA states in those beam states have only one succeeding state—LN-S10, where $N = [1, 14]$, and only one succeeding word, the fixed rhyme word. For our beam size $b = 50$, the chance is quite low that in those $b$ words there exists any suitable word to precede that rhyme word. We solve this by generating the whole sonnet in reverse, starting from the final rhyme word. Thus, when we expand the state L1-S8, we can choose from almost every word in vocabulary instead of just $b$ possible words. The price to pay is that at the beginning of each line, we need to hope in those $b$ words there exists some that are suitable to succeed comma or period.

Because we train on song lyrics, our LM tends to generate repeating words, like never ever ever ever ever. To solve this problem, we apply a penalty to those words that already generated in previous steps during the beam search.

To create a poem that fits well with the predetermined rhyme words at the end of each line, the LM model tends to choose “safe” words that are frequent and suitable for any topic, such as pronouns, adverbs, and articles. During decoding, we apply a reward on all topically related words (generated in Section 4) in the non-rhyming portion of the poem.

Finally, to further encourage the system to follow the topic, we train an encoder-decoder sequence-to-sequence model (Sutskever et al., 2014). For training, we select song lyric rhyme words and assemble them in reverse order (encoder side), and we pair this with the entire reversed lyric (decoder side). At generation time, we put all the selected rhyme words on the source side, and let the model to generate the poem conditioned on those rhyme words. In this way, when the model tries to generate the last line of the poem, it already knows all fourteen rhyme words, thus possessing better knowledge of the requested topic. We refer to generating poems using the RNN LM as the “generation model” and to this model as the “translation model”.

8 Results and Analysis

Sample outputs produced by our best system are shown in Figures 3 and 4. We find that they generally stay on topic and are fairly creative. If we request a poem on the topic Vietnam, we may see the phrase Honky Tonkin Resolution; a different topic leads the system to rhyme Dirty Harry with Bloody Mary. In this section, we present experiments we used to select among different versions of our poem generator.

The first experiment tests the effect of encouraging topical words in the body of the poem, via a direct per-word bonus. For 40 different topics, we generate 2 sonnets with and without encouragement, using the same set of rhyme words. Then we ask 23 human judges to choose the better sonnet. Each judge compares sonnets for 10 different topics. Table 3 shows that using topical words increases the
Love at First Sight
An early morning on a rainy night,
Relax and make the other people happy,
Or maybe get a little out of sight,
And wander down the streets of Cincinnati.

Girlfriend
Another party started getting heavy.
And never had a little bit of Bobby,
Or something going by the name of Eddie,
And got a finger on the trigger sloppy.

Noodles
The people wanna drink spaghetti alla,
And maybe eat a lot of other crackers,
Or sit around and talk about the salsa,
A little bit of nothing really matters.

Civil War
Creating new entire revolution,
An endless nation on eternal war,
United as a peaceful resolution,
Or not exist together any more.

Figure 4: Sample stanzas generated from different topic phrases.

<table>
<thead>
<tr>
<th>Preference</th>
<th>Encourages</th>
<th>Does Not Encourage</th>
<th>Cannot Decide</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sonnets</td>
<td>54%</td>
<td>18%</td>
<td>28%</td>
</tr>
</tbody>
</table>

Table 3: Users prefer the system that encourages the use of related words in the body (non-rhyme) portion of the poem. 40 poems are tested with 23 judges.

We find that on average, each sonnet copies only 1.2 5-grams from the training data. If we relax the repeated-word penalty and the iambic meter, this number increases to 7.9 and 10.6 copied 5-grams, respectively. Considering the lack of copying, we find the RNN-generated grammar to be quite good. The most serious—and surprisingly common—grammatical error is the wrong use of a and an, which we fix in a post-processing step.

9 Other Languages and Formats
To show the generality of our approach, we modify our system to generate Spanish-language poetry from a Spanish topic. We use these resources:

- A song lyric corpus for training our RNN. We download 97,775 Spanish song lyrics from LyricWikia,\(^1\) which amounts to 20m word tokens and 219k word types.
- A Spanish Wikipedia dump\(^2\) consisting of 885m word tokens, on which we run word2vec to find words and phrases related to the topic.

Our vocabulary consists of the 20k most frequent lyric words. For each word, we compute its syllable-stress pattern and its rhyme class (see Figure 6). Because Spanish writing is quite phonetic, we can retrieve this from the letter strings of the vocabulary.

\(^1\)http://lyrics.wikia.com/wiki/Category:Language/Spanish
\(^2\)https://dumps.wikimedia.org/eswiki/20160305/eswiki-20160305-pages-meta-current.xml.bz2
Figure 6: Sample word analyses needed to construct Spanish Hafez. \(v\)- and \(-v\) indicate whether the word starts and/or ends with a vowel sound.

For any given vocabulary word:\(^{13}\)

1. We remove silent \(h\), and convert \(y\) into \(i\).
2. We count the number of syllables by isolating vowel groups. In such groups, weak vowels \((i, u)\) attached to strong vowels \((a, e, o)\) do not form separate syllables, unless they are accented \((d\-i\-as versus d\-i\-os)\). Strong clusters are broken into separate syllables \((e.g., c\-a\-er)\).
3. We determine which vowel (and therefore syllable) is stressed. If any vowel is accented, it is stressed. If the word is accent-free, then the second-to-last syllable is stressed, unless the word ends in a consonant other than \(n\) or \(s\), in which case the last syllable is stressed.
4. We form the word’s rhyme class by breaking off a letter suffix starting at the last stressed vowel \((as\ in\ English)\). Weak vowels do not participate \((e.g., t\-i\-enda → -\-en\-da, not -\-en\-da)\). We remove \(h\) from the rhyme, so \(b\-u\-ho\) rhymes with \(c\-o\-n\-ti\-n\-u\-io\). Because rhyming is easier in Spanish than English, we do not need slant rhyme.

Most Spanish poetic formats enforce some number of syllables per line, without meter. However, there are two caveats when counting syllables:

1. Sinalefa merges vowels across word boundaries. Thus, \(l\-a\_o\-b\-r\-a\) is counted as two syllables instead of three, and \(v\-a\_a\_h\-a\-c\-e\) is counted as two syllables instead of four. A line may therefore have more words than syllables.
2. For the last word of a line \((only)\), we count up to its last stressed syllable, then add one. This means that even though iambic meter is not employed, we still need stress patterns to correctly count syllables.

We implement these constraints in the FSA framework, now with separate states for “I have seen M syllables, and the last word ended in a vowel sound” and “I have seen M syllables, and the last word ended in a consonant sound.” Technically speaking, the FSA includes single-state cycles for the Spanish word \(a\), due to sinalefa. Line-ending states can only be reached by words that have their syllable count adjusted as in point 2 above.

Figure 7 shows a sample Spanish output. The format is the classical Spanish \(s\-o\-n\-e\-t\-a\), which consists of 14 eleven-syllable lines under the rhyme scheme ABBA ABBA CDC CDC. This scheme requires us to choose up to four words with the same rhyme.

Overall, we also find Spanish outputs to be fluent, fairly creative, and on topic. Grammatical problems are a bit more common than in our English generator—for example, adjacent words sometimes disagree in number or gender. The RNN generalizations that permit these errors no doubt also permit creative phrasings.

10 Conclusion

We have described Hafez, a poetry generation system that combines hard format constraints with a deep-learning recurrent network. The system uses special techniques, such as rhyme-word choice and encoder-decoder modeling, to keep the poem on topic. We hope that future work will provide more discourse structure and function to automatic poetry, while maintaining the syntax, semantics, and creative phrasings we observe.
Acknowledgments

We would like to thank the anonymous reviewers for their helpful comments. This work was supported by DARPA (W911NF-15-1-0543) and NSF (IIS-1524371).

References


Om P. Damani. 2013. Improving pointwise mutual information (PMI) by incorporating significant co-occurrence. In Proc. ACL.


Omer Levy and Yoav Goldberg. 2014. Dependency-based word embeddings. In Proc. ACL.


Francois Role and Mohamed Nadif. 2011. Handling the impact of low frequency events on co-occurrence based measures of word similarity—a case study of pointwise mutual information. In Knowledge Discovery and Information Retrieval.


