Hafez: an Interactive Poetry Generation System

Marjan Ghazvininejad*, **Xing Shi***, **Jay Priyadarshi, and Kevin Knight**
†Information Sciences Institute & Computer Science Department
University of Southern California
{ghazvini,xingshi, jpriyada, knight}@isi.edu

Abstract

Hafez is an automatic poetry generation system that integrates a Recurrent Neural Network (RNN) with a Finite State Acceptor (FSA). It generates sonnets given arbitrary topics. Furthermore, Hafez enables user to revise and polish generated poems by adjusting various style configurations. Experiments demonstrate that such "polish" mechanisms consider the user's intention and lead to a better poem. For evaluation, we build a web interface where users can rate the quality of each poem between 1 to 5 stars. We also speed up the whole system by a factor of 10 via vocabulary pruning and GPU computation, so that adequate feedback can be collected at a fast pace. Based on such feedback, the system learns to adjust its parameters to improve poetry quality.

1 Introduction

Automated poetry generation is attracting increasing research effort. People approach the problem by using grammatical and semantic templates (Oliveira, 2009, 2012) or treating the generation task as a translation/summarization task (Zhou et al., 2009; He et al., 2012; Yan et al., 2013; Zhang and Lapata, 2014; Yi et al., 2016; Wang et al., 2016; Ghazvininejad et al., 2016). However, such poetry generation systems have to face the following challenges:

1. **Difficulty of evaluating poetry quality.** Automatic evaluation methods, like BLEU, cannot judge the rhythm, meter, creativity or syntactic/semantic coherence, and furthermore, there is no test data in most cases. Subjective evaluation requires evaluators to have relatively high literary training, so systems will receive limited feedback during the development phase.[1]

2. **Inability to adjust the generated poem.** When a poet composes a poem, he/she usually needs to revise and polish the draft from different aspects (e.g., word choice, sentiment, alliteration, etc.) for several iterations until satisfaction. This is a crucial step for poetry creation. However, given a user-supplied topic or phrase, most existing automated systems can only generate different poems by using different random seeds, providing no other support for the user to polish the generated poem in a desired direction.

3. **Slow generation speed.** Generating a poem may require a heavy search procedure. For example, the system of Ghazvininejad et al. (2016) needs 20 seconds for a four-line poem. Such slow speed is a serious bottleneck for a smooth user experience, and prevents the large-scale collection of feedback for system tuning.

This work is based on our previous poetry generation system called Hafez (Ghazvininejad et al., 2016), which generates poems in three steps: (1) search for related rhyme words given user-supplied topic, (2) create a finite-state acceptor (FSA) that incorporates the rhyme words and controls meter, and (3) use a recurrent neural network (RNN) to decode the poem guided by the FSA. We address the above-mentioned challenges with the following approaches:

*: Equal contributions

[1]The Dartmouth Turing Tests in the Creative Arts (bit.ly/20WGLF3), in which human experts are employed to judge the generation quality, is held only once a year.

Topic: Presidential elections

To hear the sound of communist aggression!
I never thought about an exit poll,
At a new Republican convention,
On the other side of gun control.

Table 1: One poem generated in a 15-minute human/computer interactive poetry contest.

1. We build a web interface[2] for our poem generation system, and for each generated poem, the user can rate its quality from 1-star to 5-stars. Our logging system collects poems, related parameters, and user feedback. Such crowd-sourcing enables us to obtain large amounts of feedback in a cheap and efficient way. Once we collect enough feedback, the system learns to find a better set of parameters and updates the system continuously.

2. We add additional weights during decoding to control the style of generated poem, including the extent of words repetition, alliteration, word length, cursing, sentiment, and concreteness.

3. We increase speed by pre-calculation, preloading model parameters, and pruning the vocabulary. We also parallelize the computation of FSA expansion, weight merging, and beam search, and we port them into a GPU. Overall, we can generate a four-line poem within 2 seconds, ten times faster than our previous CPU-based system.

With the web interface's style control and fast generation speed, people can generate creative poems within a short timespan. Table 1 shows one of the poems generated in a poetry mini-competition where 7 people are asked to use Hafez to generate poems within 15 minutes. We also conduct experiments on Amazon Mechanical Turk, which show: first, through style-control interaction, 71% users can find a better poem than the poem generated by the default configuration. Second, based on users' evaluation results, the system learns a new configuration which generates better poems.

[2]A live demo at : http://52.24.230.241/poem/advance/

2 System Description

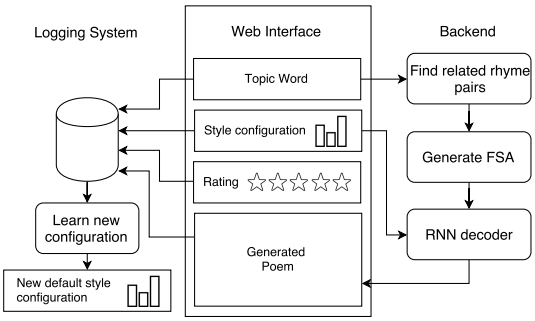


Figure 1: Overview of Hafez

Figure 1 shows an overview of Hafez. In the web interface, a user can input topic words or phrases and adjust the style configuration. This information is then sent to our backend server, which is primarily based on our previously-described work (Ghazvininejad et al., 2016). First, the backend will use the topic words/phrases to find related rhyme word pairs by using a word2vec model and a pre-calculated rhyme-type dictionary. Given these rhyme word pairs, an FSA that encodes all valid word sequences is generated, where a valid word sequence follows certain type of meter and puts the rhyme word at the end of each line. This FSA, together with the user-supplied style configuration, is then used to guide the Recurrent Neural Network (RNN) decoder to generate the rest of the poem. User can rate the generated poem using a 5-star system. Finally, the tuple (topic, style configuration, generated poem, star-rating) is pushed to the logging system. Once a while, a module will analysis the loggings, learn a better style configuration and update it as the new default style configuration.

2.1 Example in Action

Figure 2 provides an example in action. The user has input the topic word "love" and left the style configuration as default. After they click the "Generate" button, a four-line poem is generated and displayed. The user may not be satisfied with current generation, and may decide to add more positive sentiment and encourage a little bit of the alliteration. After they move the corresponding slider bars and click the "Re-generate with the same rhyme words" button, a new poem is returned. This poem has more positive sentiment ("Giving me a very happy ending" vs. "I never really need a happy ending") and more alliteration

("sing a song" and "like the loving").

2.2 Style Control

During the RNN's beam search, each beam cell records the current FSA state s. Its succeeding state is denoted as s_{suc}. All the words over all the succeeding states forms a vocabulary V_{suc}. To expand the beam state b, we need to calculate a score for each word in V_{suc}:

$$score(w, b) = score(b) + \log P_{RNN}(w) + \sum_i w_i * f_i(w); \forall w \in V_{suc} \quad (1)$$

where $\log P_{RNN}(w)$ is the log-probability of word w calculated by RNN. $score(b)$ is the accumulated score of already generated words in beam state b . $f_i(w)$ is ith feature function and w_i is the corresponding weight.

To control the style, we design the following 8 features:

1. Encourage/discourage words. User can input words that they want in the poem, or out. $f(w) = I(w, V_{enc/dis})$ where $I(w, V) = 1$ if w is in the word list V, otherwise $I(w, V) = 0$. $w_{enc} = 5$ and $w_{dis} = -5$.

2. Curse words. We pre-build a curse-word list V_{curse}, and $f(w) = I(w, V_{curse})$.

3. Repetition. To control the extent of repeated words in the poem. For each beam, we record the current generated words $V_{history}$, and $f(w) = I(w, V_{history})$.

4. Alliteration. To controls how often adjacent *non-function* words start with the same consonant sound. In the beam cell, we also record the previous generated word w_{t-1}, and $f(w_t) = 1$ if w_t and w_{t-1} shares the same first consonant sound, otherwise it equals 0.

5. Word length. To control a preference for longer words in the generated poem. $f(w) = length(w)^2$.

6. Topical words. For each user-supplied topic words, we generate a list of related words $V_{topical}$. $f(w) = I(w, V_{topical})$.

7. Sentiment. We pre-build a word list together with its sentiment scores based on SentiWordNet (Baccianella et al., 2010). $f(w)$ equals to $w's$ sentiment score.

8. Concrete words. We pre-build a word list together with a score to reflect its concreteness based on Brysbaert et al. (2014). $f(w)$ equals to w's concreteness score.

2.3 Speedup

To find the related rhyme word pairs, we employ a word2vec model. Given a topic word or phrase $w_t \in V$, we find related words w_r based on the cosine distance:

$$w_r = \underset{w_r \in V' \subseteq V}{\text{argmax}}\ cosine(e_{w_r}, e_{w_t}) \quad (2)$$

where e_w is the embedding of word w. Then we calculate the rhyme type of each related word w_r to find rhyme pairs.

To speed up this step, we carefully optimize the computation with these methods:

1. Pre-load all parameters into RAM. As we are aiming to accept arbitrary topics, the vocabulary V of word2vec model is very large (1.8M words and phrases). Pre-loading such a large number of parameters saves 3-4 seconds.

2. Pre-calculate the rhyme types for all words $w \in V'$. During runtime, we use this dictionary to lookup the rhyme type.

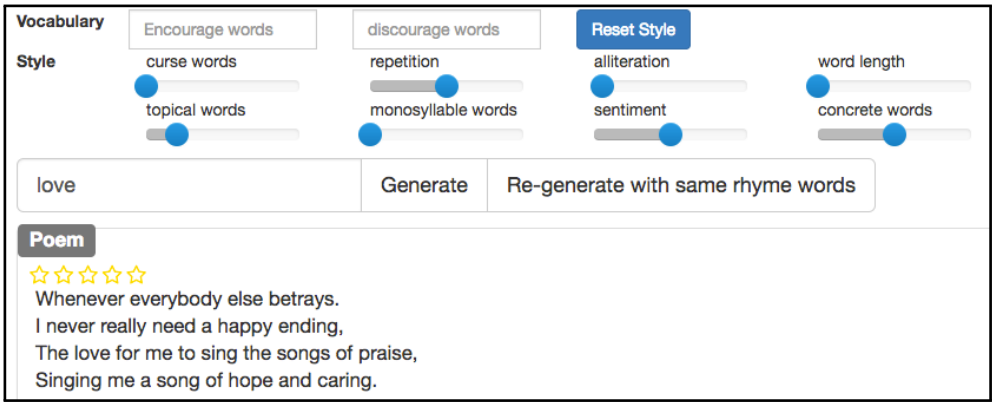
3. Shrink V'. As every rhyme word/phrase pairs must be in the target vocabulary V_{RNN} of the RNN, we further shrink $V' = V \cap V_{RNN}$.

To speedup the RNN decoding step, we use GPU processing for all forward-propagation computations. For beam search, we port to GPU the two most time-consuming parts, calculating scores with Equation 1 and finding the top words based the score:

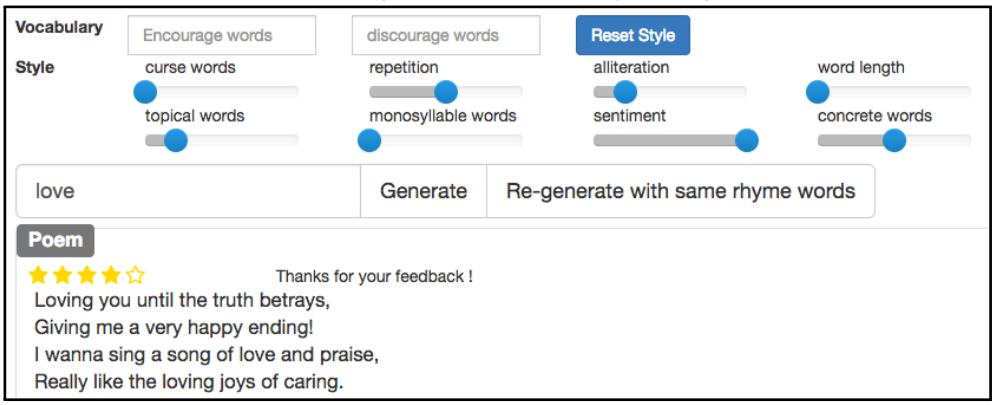
1. We warp all the computation needed in Equation 1 into a single large GPU kernel launch.

2. With beam size B, to find the top k words, instead of using a heap sort on CPU with complexity $\mathcal{O}(B|V_{suc}|logk)$, we do a global sort on GPU with complexity $\mathcal{O}(B|V_{suc}|log(B|V_{suc}|))$ in one kernel launch. Even though the complexity increases, the computation time in practice reduces quite a bit.

Finally, our system can generate a 4-line poem within 2 seconds, which is 10 times faster than previous CPU-based version.



(a) Poem generated with default style settings



(b) Poem generated with user adjusted style settings

Figure 2: A poem generated with (a) default style configuration and (b) user-adjusted style configuration.

2.4 Learn a New Style Configuration

Except the fixed weight for encourage/discourage words, all other 7 weights form our style configuration space:

$$W = \{w_i | i = 1..7\} \tag{3}$$

and we denote the default configuration as W_d, intuitively selected.

Users will usually start with W_d to generate their first poem p_d, and if they later adjust the style configuration and click "Re-generate with same rhyme words" button, the new poem p_i will change the style accordingly, but still with the same rhyme words as p_d. In the logging system, a unique hash h_r is recorded to distinguish different rhyme word sets in the poem.

After proper processing, our logging system organizes the data points as a dictionary $D = \{h_r : [(p_d, W_d, r_d), (p_1, W_1, r_1), ..., (p_n, W_n, r_n)]\}$. r_d represents user's star rating for the poem with default setting(p_d) while $r_i \in \{1, ..., n\}$ represents user's star rating for $p_1, ..., p_n$.

To learn a new style configuration W_{new}, we construct our training data $D_{train} = \{h_r : [(\Delta W_1, \Delta r_1), ..., (\Delta W_n, \Delta r_n)]\}$, where $\Delta W_i = W_i - W_d$ and $\Delta r_i = r_i - r_d$. Then, we fit a quadratic regression between the rating change Δr and each weight change $\Delta w_j \in \Delta W$ independently:

$$\Delta r = a(\Delta w_j)^2 + b\Delta w_j + c \tag{4}$$

and the new weight $w_{j,new}$ is

$$w_{j,new} = w_{j,d} + \underset{\Delta w_j}{\operatorname{argmax}}\, \Delta r \tag{5}$$

2.5 Alexa Skill

To feel beauty, rhythm, and emotion in poetry, it is better to listen to a spoken version. With this in mind, we also publish our system as an Amazon Alexa Skill ("Custom Poetry"), so that users can ask Alexa to compose and recite a poem on any topic. Table 2 shows a sample conversation between a user and Alexa.

3 Experiments

We design an Amazon Mechanical Turk task to explore the effect of style options. In this task Turk-

User: Alexa, start Custom Poetry **Alexa**: Welcome to Custom Poetry. Please give me a topic. **User**: Mountains **Alexa**: A hundred thousand Morrison formation, An ancient crown of gold or mountain chains, Mountains from the land of elevation, A northern storm across the hills and plains. **Alexa**: Feel free to give me another topic. **User**: Lakes **Alexa**: ... **Alexa**: Feel free to give me another topic. **User**: Stop **Alexa**: Thank you for trying Custom Poetry. Have a nice day!

Table 2: Spoken onversation between a user and Alexa.

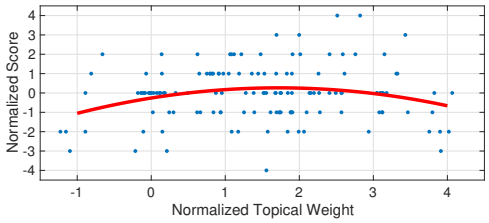
ers first use Hafez to generate a *default poem* on an arbitrary topic with the default style configuration, and rate it. Next, they are asked to adjust the style configurations to re-generate at least five different *adjusted poems* with the same rhyme words, and rate them as well. Improving the quality of *adjusted poems* over the *default poem* is not required for finishing the task, but it is encouraged. For each task, Turkers can select the best generated poem, and if subsequent human judges (domain experts) rank that poem as "great", a bonus reward will be assigned to that Turker. We gathered data from 62 completed HITs (Human Intelligence Tasks) for this task.

3.1 Human-Computer Collaboration

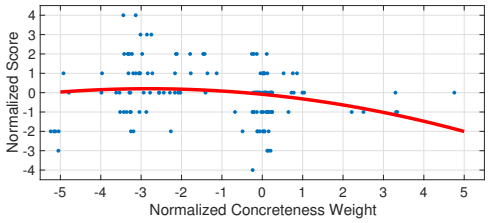
This experiment tests whether human collaboration can help Hafez generate better poems.

In only 10% of the HITs, the reported best poem was generated by the default style options, i.e., the *default poem*. Additionally, in 71% of the HITs, users assign a higher star rating to at least one of the *adjust poems* than the *default poem*. On average the best poems got $+1.4$ more stars compared to the default one.

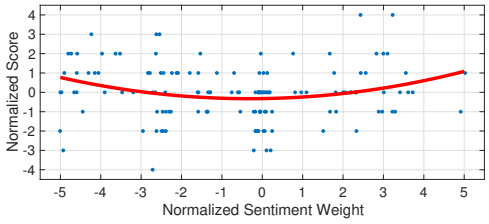
However, poem creators might have a tendency to report a higher ranking for the poems generated through the human/machine collaboration process. To sanity check the results we designed another task and asked 18 users to compare the default and the reported best poems. This experiment sec-



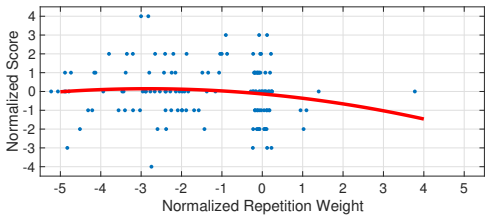
(a)



(b)



(c)



(d)

Figure 3: The distribution of poem star-ratings against normalized topical, concreteness, sentiment and repetition weights. Star ratings are computed as an offset from the version of the poem generated from default settings. We normalized all features weights by calculating their offset from the default values. The solid curve represents a quadratic regression fit to the data. To avoid overlapping marks, we add small amount of random noise to the features weights of data points.

onded the original rankings in 72% of the cases.

3.2 Automatic tuning for quality

We learn new default configurations using the data gathered from Mechanical Turk. As we explained in section 2.4, we examine the effect of different feature weights like repetition and sentiment on star ranking scores. We aim to cancel out the effect of topic and rhyme words on our scoring function.

We achieve this by plotting the score offset from the *default poem* for each topic and set of rhyme words. Figure 3 shows the distribution of scores against topical, concreteness, sentiment and repetition weights. In each plot the zero weight represents the default value. Each plot also shows a quadratic regression curve fit to its data.

In order to alter the style options toward generating better default poems, we re-set each weight to the maximum of each quadratic curve. Hence, the new weights encourage more topical, less concrete, more positive words and less repetition. It is notable that for sentiment, users prefer both more positive and more negative words to the initial neutral setting, but the preference is biased towards positive words.

We update Hafez's default settings based on this analysis. We ask 29 users to compare poems generated on the same topic and rhyme words using both old and new style settings. In 59% of the cases, users prefer the poem generated by the new setting.

We improve the default settings for generating a poem, but this does not mean that the poems cannot be further improved by human collaboration. In most cases, a better poem can be generated by collaboration with the system (changing the style options) for the specific topic and set of rhyme words.

4 Conclusion

We demonstrate Hafez, an interactive poetry generation system. It enables users to generate poems about any topic, and revise their generations through multiple style configurations. We speed up the whole system by vocabulary pruning and GPU computation. Together with an easy-accessible web interface, we collect large numbers of human evaluations in a short timespan, making automatic system tuning possible.

References

Stefano Baccianella, Andrea Esuli, and Fabrizio Sebastiani. 2010. Sentiwordnet 3.0: An enhanced lexical resource for sentiment analysis and opinion mining. In *LREC*.

Marc Brysbaert, Amy Beth Warriner, and Victor Kuperman. 2014. Concreteness ratings for 40 thousand generally known english word lemmas. *Behavior research methods* .

Marjan Ghazvininejad, Xing Shi, Yejin Choi, and Kevin Knight. 2016. Generating topical poetry. In *Proc. EMNLP*.

Jing He, Ming Zhou, and Long Jiang. 2012. Generating Chinese classical poems with statistical machine translation models. In *Proc. AAAI*.

Hugo Oliveira. 2009. Automatic generation of poetry: an overview. In *Proc. 1st Seminar of Art, Music, Creativity and Artificial Intelligence*.

Hugo Oliveira. 2012. PoeTryMe: a versatile platform for poetry generation. *Computational Creativity, Concept Invention, and General Intelligence* 1.

Qixin Wang, Tianyi Luo, Dong Wang, and Chao Xing. 2016. Chinese song iambics generation with neural attention-based model. *arXiv:1604.06274* .

Rui Yan, Han Jiang, Mirella Lapata, Shou-De Lin, Xueqiang Lv, and Xiaoming Li. 2013. I, Poet: Automatic Chinese poetry composition through a generative summarization framework under constrained optimization. In *Proc. IJCAI*.

Xiaoyuan Yi, Ruoyu Li, and Maosong Sun. 2016. Generating chinese classical poems with RNN encoder-decoder. *arXiv:1604.01537* .

Xingxing Zhang and Mirella Lapata. 2014. Chinese poetry generation with recurrent neural networks. In *Proc. EMNLP*.

Ming Zhou, Long Jiang, and Jing He. 2009. Generating Chinese couplets and quatrain using a statistical approach. In *Proc. Pacific Asia Conference on Language, Information and Computation*.