Deep-speare: A joint neural model of poetic language, meter and rhyme

Jey Han Lau[1,2] **Trevor Cohn**[2] **Timothy Baldwin**[2]
Julian Brooke[3] **Adam Hammond**[4]

[1] IBM Research Australia
[2] School of Computing and Information Systems, The University of Melbourne
[3] Thomson Reuters
[4] Department of English, University of Toronto

jeyhan.lau@gmail.com, t.cohn@unimelb.edu.au, tb@ldwin.net,
julian.brooke@gmail.com, adam.hammond@utoronto.ca

Abstract

In this paper, we propose a joint architecture that captures language, rhyme and meter for sonnet modelling. We assess the quality of generated poems using crowd and expert judgements. The stress and rhyme models perform very well, as generated poems are largely indistinguishable from human-written poems. Expert evaluation, however, reveals that a vanilla language model captures meter implicitly, and that machine-generated poems still underperform in terms of readability and emotion. Our research shows the importance expert evaluation for poetry generation, and that future research should look beyond rhyme/meter and focus on poetic language.

1 Introduction

With the recent surge of interest in deep learning, one question that is being asked across a number of fronts is: can deep learning techniques be harnessed for creative purposes? Creative applications where such research exists include the composition of music (Humphrey et al., 2013; Sturm et al., 2016; Choi et al., 2016), the design of sculptures (Lehman et al., 2016), and automatic choreography (Crnkovic-Friis and Crnkovic-Friis, 2016). In this paper, we focus on a creative textual task: automatic poetry composition.

A distinguishing feature of poetry is its *aesthetic forms*, e.g. rhyme and rhythm/meter.[1] In this work, we treat the task of poem generation as a constrained language modelling task, such that lines of a given poem rhyme, and each line follows a canonical meter and has a fixed number of stresses. Specifically, we focus on sonnets and generate quatrains in iambic pentameter (e.g. see Figure 1), based on an unsupervised model of language, rhyme and meter trained on a novel corpus of sonnets.

Shall I compare thee to a summer's day?
Thou art more lovely and more temperate:
Rough winds do shake the darling buds of May,
And summer's lease hath all too short a date:

Figure 1: 1st quatrain of Shakespeare's *Sonnet 18*.

Our findings are as follows:

- our proposed stress and rhyme models work very well, generating sonnet quatrains with stress and rhyme patterns that are indistinguishable from human-written poems and rated highly by an expert;
- a vanilla language model trained over our sonnet corpus, surprisingly, captures meter implicitly at human-level performance;
- while crowd workers rate the poems generated by our best model as nearly indistinguishable from published poems by humans, an expert annotator found the machine-generated poems to lack readability and emotion, and our best model to be only comparable to a vanilla language model on these dimensions;
- most work on poetry generation focuses on meter (Greene et al., 2010; Ghazvininejad et al., 2016; Hopkins and Kiela, 2017); our results suggest that future research should look beyond meter and focus on improving readability.

In this, we develop a new annotation framework for the evaluation of machine-generated poems, and release both a novel data of sonnets and the full source code associated with this research.[2]

[1] Noting that there are many notable divergences from this in the work of particular poets (e.g. Walt Whitman) and poetry types (such as free verse or haiku).

[2] https://github.com/jhlau/deepspeare

2 Related Work

Early poetry generation systems were generally rule-based, and based on rhyming/TTS dictionaries and syllable counting (Gervás, 2000; Wu et al., 2009; Netzer et al., 2009; Colton et al., 2012; Toivanen et al., 2013). The earliest attempt at using statistical modelling for poetry generation was Greene et al. (2010), based on a language model paired with a stress model.

Neural networks have dominated recent research. Zhang and Lapata (2014) use a combination of convolutional and recurrent networks for modelling Chinese poetry, which Wang et al. (2016) later simplified by incorporating an attention mechanism and training at the character level. For English poetry, Ghazvininejad et al. (2016) introduced a finite-state acceptor to explicitly model rhythm in conjunction with a recurrent neural language model for generation. Hopkins and Kiela (2017) improve rhythm modelling with a cascade of weighted state transducers, and demonstrate the use of character-level language model for English poetry. A critical difference over our work is that we jointly model both poetry content and forms, and unlike previous work which use dictionaries (Ghazvininejad et al., 2016) or heuristics (Greene et al., 2010) for rhyme, we learn it automatically.

3 Sonnet Structure and Dataset

The sonnet is a poem type popularised by Shakespeare, made up of 14 lines structured as 3 quatrains (4 lines) and a couplet (2 lines);[3] an example quatrain is presented in Figure 1. It follows a number of *aesthetic forms*, of which two are particularly salient: stress and rhyme.

A sonnet line obeys an alternating stress pattern, called the iambic pentameter, e.g.:

S^- S^+ S^- S^+ S^- S^+ S^- S^+ S^- S^+
Shall I compare thee to a summer's day?

where S^- and S^+ denote unstressed and stressed syllables, respectively.

A sonnet also rhymes, with a typical rhyming scheme being *ABAB CDCD EFEF GG*. There are a number of variants, however, mostly seen in the quatrains; e.g. *AABB* or *ABBA* are also common.

We build our sonnet dataset from the latest image of Project Gutenberg.[4] We first create a (generic) poetry document collection using the GutenTag tool (Brooke et al., 2015), based on its inbuilt poetry classifier and rule-based structural tagging of individual poems.

Partition	#Sonnets	#Words
Train	2685	367K
Dev	335	46K
Test	335	46K

Table 1: SONNET dataset statistics.

Given the poems, we use word and character statistics derived from Shakespeare's 154 sonnets to filter out all non-sonnet poems (to form the "BACKGROUND" dataset), leaving the sonnet corpus ("SONNET").[5] Based on a small-scale manual analysis of SONNET, we find that the approach is sufficient for extracting sonnets with high precision. BACKGROUND serves as a large corpus (34M words) for pre-training word embeddings, and SONNET is further partitioned into training, development and testing sets. Statistics of SONNET are given in Table 1.[6]

4 Architecture

We propose modelling both content and forms jointly with a neural architecture, composed of 3 components: (1) a language model; (2) a pentameter model for capturing iambic pentameter; and (3) a rhyme model for learning rhyming words.

Given a sonnet line, the language model uses standard categorical cross-entropy to predict the next word, and the pentameter model is similarly trained to learn the alternating iambic stress patterns.[7] The rhyme model, on the other hand, uses a margin-based loss to separate rhyming word pairs from non-rhyming word pairs in a quatrain. For generation we use the language model to generate one word at a time, while applying the pentame-

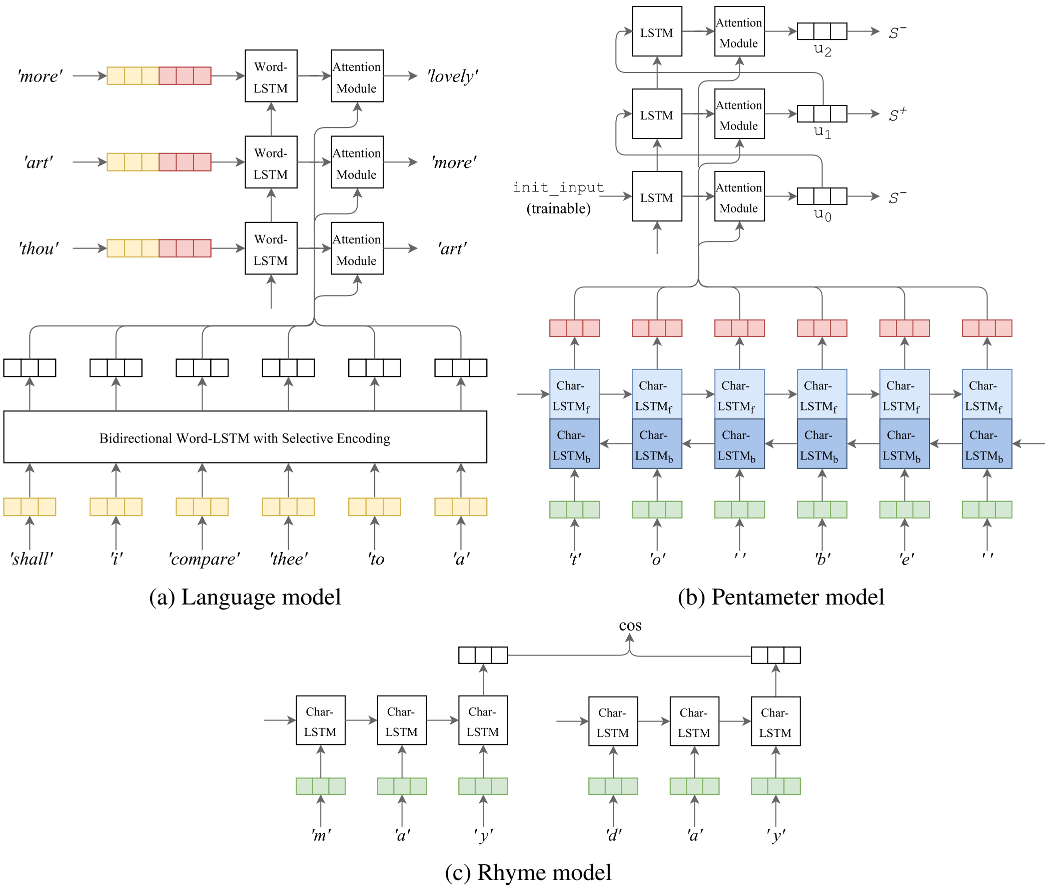
[3] There are other forms of sonnets, but the Shakespearean sonnet is the dominant one. Hereinafter "sonnet" is used to specifically mean Shakespearean sonnets.

[4] `https://www.gutenberg.org/`.

[5] The following constraints were used to select sonnets: $8.0 \leqslant$ mean words per line $\leqslant 11.5$; $40 \leqslant$ mean characters per line $\leqslant 51.0$; min/max number of words per line of 6/15; min/max number of characters per line of 32/60; and min letter ratio per line $\geqslant 0.59$.

[6] The sonnets in our collection are largely in Modern English, with possibly a small number of poetry in Early Modern English. The potentially mixed-language dialect data might add noise to our system, and given more data it would be worthwhile to include time period as a factor in the model.

[7] There are a number of variations in addition to the standard pattern (Greene et al., 2010), but our model uses only the standard pattern as it is the dominant one.



(a) Language model

(b) Pentameter model

(c) Rhyme model

Figure 2: Architecture of the language, pentameter and rhyme models. Colours denote shared weights.

ter model to sample meter-conforming sentences and the rhyme model to enforce rhyme. The architecture of the joint model is illustrated in Figure 2. We train all the components together by treating each component as a sub-task in a multitask learning setting.[8]

4.1 Language Model

The language model is a variant of an LSTM encoder–decoder model with attention (Bahdanau et al., 2015), where the encoder encodes the preceding context (i.e. all sonnet lines before the current line) and the decoder decodes one word at a time for the current line, while attending to the preceding context.

In the encoder, we embed context words z_i using embedding matrix $\mathbf{W}_{wrd}$ to yield $\mathbf{w}_i$, and feed them to a biLSTM[9] to produce a sequence of encoder hidden states $\mathbf{h}_i = [\overrightarrow{\mathbf{h}}_i; \overleftarrow{\mathbf{h}}_i]$. Next we apply a selective mechanism (Zhou et al., 2017) to each $\mathbf{h}_i$. By defining the representation of the whole context $\overline{\mathbf{h}} = [\overrightarrow{\mathbf{h}}_C; \overleftarrow{\mathbf{h}}_1]$ (where C is the number of words in the context), the selective mechanism filters the hidden states $\mathbf{h}_i$ using $\overline{\mathbf{h}}$ as follows:

$$\mathbf{h}'_i = \mathbf{h}_i \odot \sigma(\mathbf{W}_a \mathbf{h}_i + \mathbf{U}_a \overline{\mathbf{h}} + \mathbf{b}_a)$$

where $\odot$ denotes element-wise product. Hereinafter $\mathbf{W}$, $\mathbf{U}$ and $\mathbf{b}$ are used to refer to model parameters. The intuition behind this procedure is to selectively filter less useful elements from the context words.

In the decoder, we embed words x_t in the current line using the encoder-shared embedding matrix ($\mathbf{W}_{wrd}$) to produce $\mathbf{w}_t$. In addition to the word embeddings, we also embed the characters of a word using embedding matrix $\mathbf{W}_{chr}$ to produce $\mathbf{c}_{t,i}$, and feed them to a bidirectional (character-level) LSTM:

$$\begin{aligned} \overrightarrow{\mathbf{u}}_{t,i} &= \mathrm{LSTM}_f(\mathbf{c}_{t,i}, \overrightarrow{\mathbf{u}}_{t,i-1}) \\ \overleftarrow{\mathbf{u}}_{t,i} &= \mathrm{LSTM}_b(\mathbf{c}_{t,i}, \overleftarrow{\mathbf{u}}_{t,i+1}) \end{aligned} \qquad (1)$$

We represent the character encoding of a word by concatenating the last forward and first back-

[8] We stress that although the components appear to be disjointed, the shared parameters allow the components to mutually influence each other during joint training. To exemplify this, we found that the pentameter model performs very poorly when we train each component separately.

[9] We use a single layer for all LSTMs.

ward hidden states $\overline{\mathbf{u}}_t = [\overrightarrow{\mathbf{u}}_{t,L}; \overleftarrow{\mathbf{u}}_{t,1}]$, where L is the length of the word. We incorporate character encodings because they provide orthographic information, improve representations of unknown words, and are shared with the pentameter model (Section 4.2).[10] The rationale for sharing the parameters is that we see word stress and language model information as complementary.

Given the word embedding $\mathbf{w}_t$ and character encoding $\overline{\mathbf{u}}_t$, we concatenate them together and feed them to a unidirectional (word-level) LSTM to produce the decoding states:

$$\mathbf{s}_t = \text{LSTM}([\mathbf{w}_t; \overline{\mathbf{u}}_t], \mathbf{s}_{t-1}) \tag{2}$$

We attend $\mathbf{s}_t$ to encoder hidden states $\mathbf{h}'_i$ and compute the weighted sum of $\mathbf{h}'_i$ as follows:

$$\begin{aligned} e^t_i &= \mathbf{v}_b^\intercal \tanh(\mathbf{W}_b \mathbf{h}'_i + \mathbf{U}_b \mathbf{s}_t + \mathbf{b}_b) \\ \mathbf{a}^t &= \text{softmax}(\mathbf{e}^t) \\ \mathbf{h}^*_t &= \sum_i a^t_i \mathbf{h}'_i \end{aligned}$$

To combine $\mathbf{s}_t$ and $\mathbf{h}^*_t$, we use a gating unit similar to a GRU (Cho et al., 2014; Chung et al., 2014): $\mathbf{s}'_t = \text{GRU}(\mathbf{s}_t, \mathbf{h}^*_t)$. We then feed $\mathbf{s}'_t$ to a linear layer with softmax activation to produce the vocabulary distribution (i.e. $\text{softmax}(\mathbf{W}_{out}\mathbf{s}'_t + \mathbf{b}_{out})$, and optimise the model with standard categorical cross-entropy loss. We use dropout as regularisation (Srivastava et al., 2014), and apply it to the encoder/decoder LSTM outputs and word embedding lookup. The same regularisation method is used for the pentameter and rhyme models.

As our sonnet data is relatively small for training a neural language model (367K words; see Table 1), we pre-train word embeddings and reduce parameters further by introducing weight-sharing between output matrix $\mathbf{W}_{out}$ and embedding matrix $\mathbf{W}_{wrd}$ via a projection matrix $\mathbf{W}_{prj}$ (Inan et al., 2016; Paulus et al., 2017; Press and Wolf, 2017):

$$\mathbf{W}_{out} = \tanh(\mathbf{W}_{wrd}\mathbf{W}_{prj})$$

4.2 Pentameter Model

This component is designed to capture the alternating iambic stress pattern. Given a sonnet line, the pentameter model learns to attend to the appropriate characters to predict the 10 binary stress symbols sequentially.[11] As punctuation is not pronounced, we preprocess each sonnet line to remove all punctuation, leaving only spaces and letters. Like the language model, the pentameter model is fashioned as an encoder–decoder network.

In the encoder, we embed the characters using the shared embedding matrix $\mathbf{W}_{chr}$ and feed them to the shared bidirectional character-level LSTM (Equation (1)) to produce the character encodings for the sentence: $\mathbf{u}_j = [\overrightarrow{\mathbf{u}}_j; \overleftarrow{\mathbf{u}}_j]$.

In the decoder, it attends to the characters to predict the stresses sequentially with an LSTM:

$$\mathbf{g}_t = \text{LSTM}(\mathbf{u}^*_{t-1}, \mathbf{g}_{t-1})$$

where $\mathbf{u}^*_{t-1}$ is the weighted sum of character encodings from the previous time step, produced by an attention network which we describe next,[12] and $\mathbf{g}_t$ is fed to a linear layer with softmax activation to compute the stress distribution.

The attention network is designed to focus on stress-producing characters, whose positions are monotonically increasing (as stress is predicted sequentially). We first compute μ_t, the mean position of focus:

$$\begin{aligned} \mu'_t &= \sigma(\mathbf{v}_c^\intercal \tanh(\mathbf{W}_c \mathbf{g}_t + \mathbf{U}_c \mu_{t-1} + \mathbf{b}_c)) \\ \mu_t &= M \times \min(\mu'_t + \mu_{t-1}, 1.0) \end{aligned}$$

where M is the number of characters in the sonnet line. Given μ_t, we can compute the (unnormalised) probability for each character position:

$$p^t_j = \exp\left(\frac{-(j - \mu_t)^2}{2T^2}\right)$$

where standard deviation T is a hyper-parameter. We incorporate this position information when computing $\mathbf{u}^*_t$:[13]

$$\begin{aligned} \mathbf{u}'_j &= p^t_j \mathbf{u}_j \\ d^t_j &= \mathbf{v}_d^\intercal \tanh(\mathbf{W}_d \mathbf{u}'_j + \mathbf{U}_d \mathbf{g}_t + \mathbf{b}_d) \\ \mathbf{f}^t &= \text{softmax}(\mathbf{d}^t + \log \mathbf{p}^t) \\ \mathbf{u}^*_t &= \sum_j b^t_j \mathbf{u}_j \end{aligned}$$

[10] We initially shared the character encodings with the rhyme model as well, but found sub-par performance for the rhyme model. This is perhaps unsurprising, as rhyme and stress are qualitatively very different aspects of forms.

[11] That is, given the input line *Shall I compare thee to a summer's day?* the model is required to output S^- S^+ S^- S^+ S^- S^+ S^- S^+ S^- S^+, based on the syllable boundaries from Section 3.

[12] Initial input ($\mathbf{u}^*_0$) and state ($\mathbf{g}_0$) is a trainable vector and zero vector respectively.

[13] Spaces are masked out, so they always yield zero attention weights.

Intuitively, the attention network incorporates the position information at two points, when computing: (1) d_j^t by weighting the character encodings; and (2) $\mathbf{f}^t$ by adding the position log probabilities. This may appear excessive, but preliminary experiments found that this formulation produces the best performance.

In a typical encoder–decoder model, the attended encoder vector $\mathbf{u}_t^*$ would be combined with the decoder state $\mathbf{g}_t$ to compute the output probability distribution. Doing so, however, would result in a zero-loss model as it will quickly learn that it can simply ignore $\mathbf{u}_t^*$ to predict the alternating stresses based on $\mathbf{g}_t$. For this reason we use only $\mathbf{u}_t^*$ to compute the stress probability:

$$P(S^-) = \sigma(\mathbf{W}_e \mathbf{u}_t^* + b_e)$$

which gives the loss $\mathcal{L}_{ent} = \sum_t - \log P(S_t^\star)$ for the whole sequence, where $S_t^\star$ is the target stress at time step t.

We find the decoder still has the tendency to attend to the same characters, despite the incorporation of position information. To regularise the model further, we introduce two loss penalties: repeat and coverage loss.

The repeat loss penalises the model when it attends to previously attended characters (See et al., 2017), and is computed as follows:

$$\mathcal{L}_{rep} = \sum_t \sum_j \min(f_j^t, \sum_{t=1}^{t-1} f_j^t)$$

By keeping a sum of attention weights over all previous time steps, we penalise the model when it focuses on characters that have non-zero history weights.

The repeat loss discourages the model from focussing on the same characters, but does not assure that the appropriate characters receive attention. Observing that stresses are aligned with the vowels of a syllable, we therefore penalise the model when vowels are ignored:

$$\mathcal{L}_{cov} = \sum_{j \in V} \text{ReLU}(C - \sum_{t=1}^{10} f_j^t)$$

where V is a set of positions containing vowel characters, and C is a hyper-parameter that defines the minimum attention threshold that avoids penalty.

To summarise, the pentameter model is optimised with the following loss:

$$\mathcal{L}_{pm} = \mathcal{L}_{ent} + \alpha \mathcal{L}_{rep} + \beta \mathcal{L}_{cov} \qquad (3)$$

where α and β are hyper-parameters for weighting the additional loss terms.

4.3 Rhyme Model

Two reasons motivate us to learn rhyme in an unsupervised manner: (1) we intend to extend the current model to poetry in other languages (which may not have pronunciation dictionaries); and (2) the language in our SONNET data is not Modern English, and so contemporary dictionaries may not accurately reflect the rhyme of the data.

Exploiting the fact that rhyme exists in a quatrain, we feed sentence-ending word pairs of a quatrain as input to the rhyme model and train it to learn how to separate rhyming word pairs from non-rhyming ones. Note that the model does not assume any particular rhyming scheme — it works as long as quatrains have rhyme.

A training example consists of a number of word pairs, generated by pairing one target word with 3 other reference words in the quatrain, i.e. $\{(x_t, x_r), (x_t, x_{r+1}), (x_t, x_{r+2})\}$, where x_t is the target word and x_{r+i} are the reference words.[14] We assume that in these 3 pairs there should be one rhyming and 2 non-rhyming pairs. From preliminary experiments we found that we can improve the model by introducing additional non-rhyming or negative reference words. Negative reference words are sampled uniform randomly from the vocabulary, and the number of additional negative words is a hyper-parameter.

For each word x in the word pairs we embed the characters using the shared embedding matrix $\mathbf{W}_{chr}$ and feed them to an LSTM to produce the character states $\mathbf{u}_j$.[15] Unlike the language and pentameter models, we use a unidirectional forward LSTM here (as rhyme is largely determined by the final characters), and the LSTM parameters are not shared. We represent the encoding of the whole word by taking the last state $\overline{\mathbf{u}} = \mathbf{u}_L$, where L is the character length of the word.

Given the character encodings, we use a

[14] E.g. for the quatrain in Figure 1, a training example is $\{$(*day*, *temperate*), (*day*, *may*), (*day*, *date*)$\}$.

[15] The character embeddings are the only shared parameters in this model.

margin-based loss to optimise the model:

$$Q = \{\cos(\overline{\mathbf{u}}_t, \overline{\mathbf{u}}_r), \cos(\overline{\mathbf{u}}_t, \overline{\mathbf{u}}_{r+1}), ...\}$$
$$\mathcal{L}_{rm} = \max(0, \delta - \text{top}(Q, 1) + \text{top}(Q, 2))$$

where $\text{top}(Q, k)$ returns the k-th largest element in Q, and δ is a margin hyper-parameter.

Intuitively, the model is trained to learn a sufficient margin (defined by δ) that separates the best pair with *all others*, with the second-best being used to quantify *all others*. This is the justification used in the multi-class SVM literature for a similar objective (Wang and Xue, 2014).

With this network we can estimate whether two words rhyme by computing the cosine similarity score during generation, and resample words as necessary to enforce rhyme.

4.4 Generation Procedure

We focus on quatrain generation in this work, and so the aim is to generate 4 lines of poetry. During generation we feed the hidden state from the previous time step to the language model's decoder to compute the vocabulary distribution for the current time step. Words are sampled using a temperature between 0.6 and 0.8, and they are resampled if the following set of words is generated: (1) UNK token; (2) non-stopwords that were generated before;[16] (3) any generated words with a frequency $\geqslant$ 2; (4) the preceding 3 words; and (5) a number of symbols including parentheses, single and double quotes.[17] The first sonnet line is generated without using any preceding context.

We next describe how to incorporate the pentameter model for generation. Given a sonnet line, the pentameter model computes a loss $\mathcal{L}_{pm}$ (Equation (3)) that indicates how well the line conforms to the iambic pentameter. We first generate 10 candidate lines (all initialised with the same hidden state), and then sample one line from the candidate lines based on the pentameter loss values ($\mathcal{L}_{pm}$). We convert the losses into probabilities by taking the softmax, and a sentence is sampled with temperature $= 0.1$.

To enforce rhyme, we randomly select one of the rhyming schemes (*AABB*, *ABAB* or *ABBA*) and resample sentence-ending words as necessary. Given a pair of words, the rhyme model produces a cosine similarity score that estimates how well the two words rhyme. We resample the second word of a rhyming pair (e.g. when generating the second *A* in *AABB*) until it produces a cosine similarity $\geqslant$ 0.9. We also resample the second word of a non-rhyming pair (e.g. when generating the first *B* in *AABB*) by requiring a cosine similarity $\leqslant$ 0.7.[18]

When generating in the forward direction we can never be sure that any particular word is the last word of a line, which creates a problem for resampling to produce good rhymes. This problem is resolved in our model by reversing the direction of the language model, i.e. generating the last word of each line first. We apply this inversion trick at the word level (character order of a word is not modified) and only to the language model; the pentameter model receives the original word order as input.

5 Experiments

We assess our sonnet model in two ways: (1) component evaluation of the language, pentameter and rhyme models; and (2) poetry generation evaluation, by crowd workers and an English literature expert. A sample of machine-generated sonnets are included in the supplementary material.

We tune the hyper-parameters of the model over the development data (optimal configuration in the supplementary material). Word embeddings are initialised with pre-trained skip-gram embeddings (Mikolov et al., 2013a,b) on the BACKGROUND dataset, and are updated during training. For optimisers, we use Adagrad (Duchi et al., 2011) for the language model, and Adam (Kingma and Ba, 2014) for the pentameter and rhyme models. We truncate backpropagation through time after 2 sonnet lines, and train using 30 epochs, resetting the network weights to the weights from the previous epoch whenever development loss worsens.

5.1 Component Evaluation

5.1.1 Language Model

We use standard perplexity for evaluating the language model. In terms of model variants, we have:[19]

- **LM:** Vanilla LSTM language model;
- **LM*:** LSTM language model that incorporates character encodings (Equation (2));

[16] We use the NLTK stopword list (Bird et al., 2009).

[17] We add these constraints to prevent the model from being too repetitive, in generating the same words.

[18] Maximum number of resampling steps is capped at 1000. If the threshold is exceeded the model is reset to generate from scratch again.

[19] All models use the same (applicable) hyper-parameter configurations.

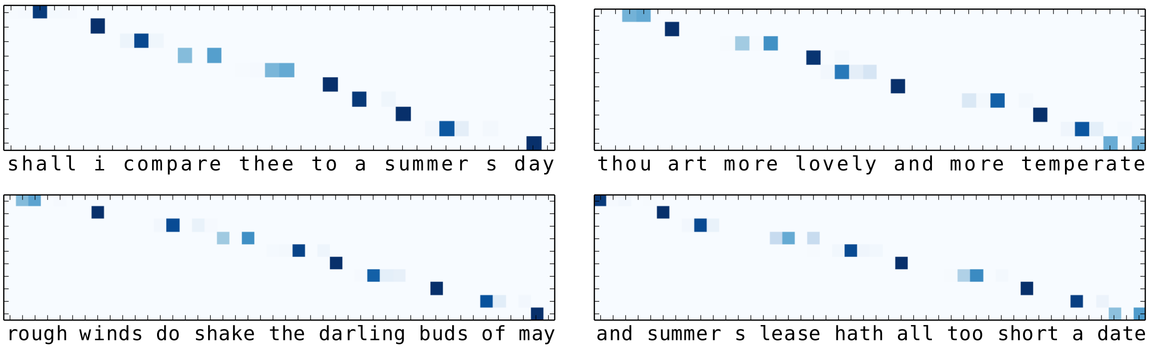


Figure 3: Character attention weights for the first quatrain of Shakespeare's *Sonnet 18*.

Model	Ppl	Stress Acc	Rhyme F1
LM	90.13	–	–
LM*	84.23	–	–
LM**	80.41	–	–
LM**-C	83.68	–	–
LM**+PM+RM	80.22	0.74	0.91
Stress-BL	–	0.80	–
Rhyme-BL	–	–	0.74
Rhyme-EM	–	–	0.71

Table 2: Component evaluation for the language model ("Ppl" = perplexity), pentameter model ("Stress Acc"), and rhyme model ("Rhyme F1"). Each number is an average across 10 runs.

- **LM**:** LSTM language model that incorporates both character encodings and preceding context;
- **LM**-C:** Similar to LM**, but preceding context is encoded using convolutional networks, inspired by the poetry model of Zhang and Lapata (2014);[20]
- **LM**+PM+RM:** the full model, with joint training of the language, pentameter and rhyme models.

Perplexity on the test partition is detailed in Table 2. Encouragingly, we see that the incorporation of character encodings and preceding context improves performance substantially, reducing perplexity by almost 10 points from LM to LM**. The inferior performance of LM**-C compared to LM** demonstrates that our approach of processing context with recurrent networks with selective encoding is more effective than convolutional networks. The full model LM**+PM+RM, which learns stress and rhyme patterns simultaneously, also appears to improve the language model slightly.

5.1.2 Pentameter Model

To assess the pentameter model, we use the attention weights to predict stress patterns for words in the test data, and compare them against stress patterns in the CMU pronunciation dictionary.[21] Words that have no coverage or have non-alternating patterns given by the dictionary are discarded. We use accuracy as the metric, and a predicted stress pattern is judged to be correct if it matches any of the dictionary stress patterns.

To extract a stress pattern for a word from the model, we iterate through the pentameter (10 time steps), and append the appropriate stress (e.g. 1st time step = S^-) to the word if any of its characters receives an attention $\geqslant 0.20$.

For the baseline (Stress-BL) we use the pretrained weighted finite state transducer (WFST) provided by Hopkins and Kiela (2017).[22] The WFST maps a sequence word to a sequence of stresses by assuming each word has 1–5 stresses and the full word sequence produces iambic pentameter. It is trained using the EM algorithm on a sonnet corpus developed by the authors.

We present stress accuracy in Table 2. LM**+PM+RM performs competitively, and informal inspection reveals that a number of mistakes are due to dictionary errors. To understand the predicted stresses qualitatively, we display attention heatmaps for the the first quatrain of Shakespeare's *Sonnet 18* in Figure 3. The y-axis represents the ten stresses of the iambic pentameter, and

[20] In Zhang and Lapata (2014), the authors use a series of convolutional networks with a width of 2 words to convert 5/7 poetry lines into a fixed size vector; here we use a standard convolutional network with max-pooling operation (Kim, 2014) to process the context.

[21] http://www.speech.cs.cmu.edu/cgi-bin/cmudict. Note that the dictionary provides 3 levels of stresses: 0, 1 and 2; we collapse 1 and 2 to S^+.

[22] https://github.com/JackHopkins/ACLPoetry

CMU Rhyming Pairs		**CMU Non-Rhyming Pairs**	
Word Pair	Cos	Word Pair	Cos
(*endeavour*, *never*)	0.028	(*blood*, *stood*)	1.000
(*nowhere*, *compare*)	0.098	(*mood*, *stood*)	1.000
(*supply*, *sigh*)	0.164	(*overgrown*, *frown*)	1.000
(*sky*, *high*)	0.164	(*understood*, *food*)	1.000
(*me*, *maybe*)	0.165	(*brood*, *wood*)	1.000
(*cursed*, *burst*)	0.172	(*rove*, *love*)	0.999
(*weigh*, *way*)	0.200	(*sire*, *ire*)	0.999
(*royally*, *we*)	0.217	(*moves*, *shoves*)	0.998
(*use*, *juice*)	0.402	(*afraid*, *said*)	0.998
(*dim*, *limb*)	0.497	(*queen*, *been*)	0.996

Table 3: Rhyming errors produced by the model. Examples on the left (right) side are rhyming (non-rhyming) word pairs — determined using the CMU dictionary — that have low (high) cosine similarity. “Cos” denote the system predicted cosine similarity for the word pair.

x-axis the characters of the sonnet line (punctuation removed). The attention network appears to perform very well, without any noticeable errors. The only minor exception is *lovely* in the second line, where it predicts 2 stresses but the second stress focuses incorrectly on the character *e* rather than *y*. Additional heatmaps for the full sonnet are provided in the supplementary material.

5.1.3 Rhyme Model

We follow a similar approach to evaluate the rhyme model against the CMU dictionary, but score based on F1 score. Word pairs that are not included in the dictionary are discarded. Rhyme is determined by extracting the final stressed phoneme for the paired words, and testing if their phoneme patterns match.

We predict rhyme for a word pair by feeding them to the rhyme model and computing cosine similarity; if a word pair is assigned a score $\geqslant$ 0.8,[23] it is considered to rhyme. As a baseline (`Rhyme-BL`), we first extract for each word the last vowel and all following consonants, and predict a word pair as rhyming if their extracted sequences match. The extracted sequence can be interpreted as a proxy for the last syllable of a word.

Reddy and Knight (2011) propose an unsupervised model for learning rhyme schemes in poems via EM. There are two latent variables: ϕ specifies the distribution of rhyme schemes, and θ defines the pairwise rhyme strength between two words. The model's objective is to maximise poem likelihood over all possible rhyme scheme assignments under the latent variables ϕ and θ. We train this model (`Rhyme-EM`) on our data[24] and use the learnt θ to decide whether two words rhyme.[25]

Table 2 details the rhyming results. The rhyme model performs very strongly at F1 $>$ 0.90, well above both baselines. `Rhyme-EM` performs poorly because it operates at the word level (i.e. it ignores character/orthographic information) and hence does not generalise well to unseen words and word pairs.[26]

To better understand the errors qualitatively, we present a list of word pairs with their predicted cosine similarity in Table 3. Examples on the left side are rhyming word pairs as determined by the CMU dictionary; right are non-rhyming pairs. Looking at the rhyming word pairs (left), it appears that these words tend not to share any word-ending characters. For the non-rhyming pairs, we spot several CMU errors: (*sire*, *ire*) and (*queen*, *been*) clearly rhyme.

5.2 Generation Evaluation

5.2.1 Crowdworker Evaluation

Following Hopkins and Kiela (2017), we present a pair of quatrains (one machine-generated and one human-written, in random order) to crowd workers on CrowdFlower, and ask them to guess which is the human-written poem. Generation quality is estimated by computing the accuracy of workers at correctly identifying the human-written poem (with lower values indicate better results for the model).

We generate 50 quatrains each for `LM`, `LM`** and `LM`**`+PM+RM` (150 in total), and as a control, generate 30 quatrains with `LM` trained for one epoch. An equal number of human-written quatrains was sampled from the training partition. A HIT contained 5 pairs of poems (of which one is a control), and workers were paid \$0.05 for each HIT. Workers who failed to identify the human-written poem in the control pair reliably (minimum accuracy = 70%) were removed by CrowdFlower automati-

[23] 0.8 is empirically found to be the best threshold based on development data.

[24] We use the original authors' implementation: `https://github.com/jvamvas/rhymediscovery`.

[25] A word pair is judged to rhyme if $\theta_{w_1,w_2} \geqslant 0.02$; the threshold (0.02) is selected based on development performance.

[26] Word pairs that did not co-occur in a poem in the training data have rhyme strength of zero.

Model	Accuracy
LM	0.742
LM**	0.672
LM**+PM+RM	0.532
LM**+RM	0.532

Table 4: Crowdworker accuracy performance.

Model	Meter	Rhyme	Read.	Emotion
LM	4.00±0.73	1.57±0.67	2.77±0.67	2.73±0.51
LM**	4.07±1.03	1.53±0.88	3.10±1.04	2.93±0.93
LM**+PM+RM	4.10±0.91	4.43±0.56	2.70±0.69	2.90±0.79
Human	3.87±1.12	4.10±1.35	4.80±0.48	4.37±0.71

Table 5: Expert mean and standard deviation ratings on several aspects of the generated quatrains.

cally, and they were restricted to do a maximum of 3 HITs. To dissuade workers from using search engines to identify real poems, we presented the quatrains as images.

Accuracy is presented in Table 4. We see a steady decrease in accuracy (= improvement in model quality) from LM to LM** to LM**+PM+RM, indicating that each model generates quatrains that are less distinguishable from human-written ones. Based on the suspicion that workers were using rhyme to judge the poems, we tested a second model, LM**+RM, which is the full model without the pentameter component. We found identical accuracy (0.532), confirming our suspicion that crowd workers depend on only rhyme in their judgements. These observations demonstrate that meter is largely ignored by lay persons in poetry evaluation.

5.2.2 Expert Judgement

To better understand the qualitative aspects of our generated quatrains, we asked an English literature expert (a Professor of English literature at a major English-speaking university; the last author of this paper) to directly rate 4 aspects: meter, rhyme, readability and emotion (i.e. amount of emotion the poem evokes). All are rated on an ordinal scale between 1 to 5 (1 = worst; 5 = best). In total, 120 quatrains were annotated, 30 each for LM, LM**, LM**+PM+RM, and human-written poems (Human). The expert was blind to the source of each poem. The mean and standard deviation of the ratings are presented in Table 5.

We found that our full model has the highest ratings for both rhyme and meter, even higher than human poets. This might seem surprising, but in fact it is well established that real poets regularly break rules of form to create other effects (Adams, 1997). Despite excellent form, the output of our model can easily be distinguished from human-written poetry due to its lower emotional impact and readability. In particular, there is evidence here that our focus on form actually hurts the readability of the resulting poems, relative even to the simpler language models. Another surprise is how well simple language models do in terms of their grasp of meter: in this expert evaluation, we see only marginal benefit as we increase the sophistication of the model. Taken as a whole, this evaluation suggests that future research should look beyond forms, towards the substance of good poetry.

6 Conclusion

We propose a joint model of language, meter and rhyme that captures language and form for modelling sonnets. We provide quantitative analyses for each component, and assess the quality of generated poems using judgements from crowdworkers and a literature expert. Our research reveals that vanilla LSTM language model captures meter implicitly, and our proposed rhyme model performs exceptionally well. Machine-generated generated poems, however, still underperform in terms of readability and emotion.

References

Stephen Adams. 1997. *Poetic designs: An introduction to meters, verse forms, and figures of speech*. Broadview Press.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2015. Neural machine translation by jointly learning to align and translate. In *International Conference on Learning Representations*, San Diego, USA.

Steven Bird, Ewan Klein, and Edward Loper. 2009. *Natural Language Processing with Python — Analyzing Text with the Natural Language Toolkit*. O'Reilly Media, Sebastopol, USA.

Julian Brooke, Adam Hammond, and Graeme Hirst. 2015. GutenTag: An NLP-driven tool for digital humanities research in the Project Gutenberg corpus. In *Proceedings of the 4nd Workshop on Computational Literature for Literature (CLFL '15)*.

Kyunghyun Cho, Bart van Merrienboer, Dzmitry Bahdanau, and Yoshua Bengio. 2014. On the properties of neural machine translation: Encoder–decoder approaches. In *Proceedings of SSST-8, Eighth Workshop on Syntax, Semantics and Structure in Statistical Translation*, pages 103–111, Doha, Qatar.

Keunwoo Choi, George Fazekas, and Mark Sandler. 2016. Text-based LSTM networks for automatic music composition. In *Proceedings of the 1st Conference on Computer Simulation of Musical Creativity*, Huddersfield, UK.

Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. 2014. Empirical evaluation of gated recurrent neural networks on sequence modeling. In *NIPS Deep Learning and Representation Learning Workshop*, pages 103–111, Montreal, Canada.

Simon Colton, Jacob Goodwin, and Tony Veale. 2012. Full face poetry generation. In *Proceedings of the Third International Conference on Computational Creativity*, pages 95–102.

Luka Crnkovic-Friis and Louise Crnkovic-Friis. 2016. Generative choreography using deep learning. In *Proceedings of the 7th International Conference on Computational Creativity*, Paris, France.

John Duchi, Elad Hazan, and Yoram Singer. 2011. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12:2121–2159.

Pablo Gervás. 2000. Wasp: Evaluation of different strategies for the automatic generation of spanish verse. In *Proceedings of the AISB-00 Symposium on Creative & Cultural Aspects of AI*, pages 93–100.

Marjan Ghazvininejad, Xing Shi, Yejin Choi, and Kevin Knight. 2016. Generating topical poetry. pages 1183–1191, Austin, Texas.

Erica Greene, Tugba Bodrumlu, and Kevin Knight. 2010. Automatic analysis of rhythmic poetry with applications to generation and translation. In *Proceedings of the 2010 Conference on Empirical Methods in Natural Language Processing (EMNLP 2010)*, pages 524–533, Massachusetts, USA.

Jack Hopkins and Douwe Kiela. 2017. Automatically generating rhythmic verse with neural networks. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (ACL 2017)*, pages 168–178, Vancouver, Canada.

Eric J. Humphrey, Juan P. Bello, and Yann LeCun. 2013. Feature learning and deep architectures: new directions for music informatics. *Journal of Intelligent Information Systems*, 41(3):461–481.

Hakan Inan, Khashayar Khosravi, and Richard Socher. 2016. Tying word vectors and word classifiers: A loss framework for language modeling. *CoRR*, abs/1611.01462.

Y. Kim. 2014. Convolutional neural networks for sentence classification. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP 2014)*, pages 1746–1751, Doha, Qatar.

Diederik P. Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980.

Joel Lehman, Sebastian Risi, and Jeff Clune. 2016. Creative generation of 3D objects with deep learning and innovation engines. In *Proceedings of the 7th International Conference on Computational Creativity*, Paris, France.

Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013a. Efficient estimation of word representations in vector space. In *Proceedings of Workshop at the International Conference on Learning Representations, 2013*, Scottsdale, USA.

Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013b. Distributed representations of words and phrases and their compositionality. In *Advances in Neural Information Processing Systems*, pages 3111–3119.

Yael Netzer, David Gabay, Yoav Goldberg, and Michael Elhadad. 2009. Gaiku: Generating haiku with word associations norms. In *Proceedings of the Workshop on Computational Approaches to Linguistic Creativity*, pages 32–39.

Romain Paulus, Caiming Xiong, and Richard Socher. 2017. A deep reinforced model for abstractive summarization. *CoRR*, abs/1705.04304.

Ofir Press and Lior Wolf. 2017. Using the output embedding to improve language models. In *Proceedings of the 15th Conference of the EACL (EACL 2017)*, pages 157–163, Valencia, Spain.

Sravana Reddy and Kevin Knight. 2011. Unsupervised discovery of rhyme schemes. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies (ACL HLT 2011)*, pages 77–82, Portland, Oregon, USA.

Abigail See, Peter J. Liu, and Christopher D. Manning. 2017. Get to the point: Summarization with pointer-generator networks. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (ACL 2017)*, pages 1073–1083, Vancouver, Canada.

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15:1929–1958.

Bob L. Sturm, Jo ao Felipe Santos, Oded Ben-Tal, and Iryna Korshunova. 2016. Music transcription modelling and composition using deep learning. In *Proceedings of the 1st Conference on Computer Simulation of Musical Creativity*, Huddersfield, UK.

Jukka M. Toivanen, Matti Järvisalo, and Hannu Toivonen. 2013. Harnessing constraint programming for poetry composition. In *Proceedings of the Fourth International Conference on Computational Creativity*, pages 160–160.

Qixin Wang, Tianyi Luo, Dong Wang, and Chao Xing. 2016. Chinese song iambics generation with neural attention-based model. In *Proceedings of the 25nd International Joint Conference on Artificial Intelligence (IJCAI-2016)*, pages 2943–2949, New York, USA.

Zhe Wang and Xiangyang Xue. 2014. In *Support Vector Machines Applications*, pages 23–48. Springer.

Xiaofeng Wu, Naoko Tosa, and Ryohei Nakatsu. 2009. Newhitch haiku: An interactive renku poem composition supporting tool applied for sightseeing navigation system. *Entertainment Computing-ICEC 2009*, pages 191–196.

Xingxing Zhang and Mirella Lapata. 2014. Chinese poetry generation with recurrent neural networks. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP 2014)*, pages 670–680, Doha, Qatar.

Qingyu Zhou, Nan Yang, Furu Wei, and Ming Zhou. 2017. Selective encoding for abstractive sentence summarization. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (ACL 2017)*, pages 1095–1104, Vancouver, Canada.