Andromeda :: Poetry Generator
Team Members: Keisha Arnold, Jacob Karcz, Carrie Treeful
CS 467 Capstone
Final Report

Introduction

The Andromeda team set out to create a poetry generator using a design based on deep learning and neural networks. It was primarily research based as none of us had any experience with the subject matter or the associated tools and libraries. This steep learning curve turned out to be one of the main challenges as there was an a lot to learn in a limited amount of time. After several weeks of trial and error, we were able to successfully create a neural network model using the Keras library and train it on Shakespearean sonnets. This model produces text output that is close to a Shakespearean sonnet, but the model training doesn't capture a sonnet's distinct rhyming and meter structure. To do that we also wanted to implement a model in TensorFlow by attempting to replicate Miyamoto and Cho's description of a gated word-char LSTM. [11] Though their paper was quite detailed, there weren't many similar models implemented in TensorFlow. That coupled with learning the TensorFlow API made it a slow and at times frustrating process. Unfortunately, although we were tantalizingly close we could not get this model to output text.

The other main obstacle we faced was computing power. The calculations involved in training the neural networks need a significant amount of computing power. One team member had a NVIDIA GeForce GTX 950 GPU, but the rest of us had older, slower computers. Even on the newer GPU training took several hours and maxed out resources, leaving us unable to work during training. To remedy this, we used low-cost AWS EC2 instances which still usually took several hours to train, but freed up our local resources.

Submission Contents

- Zip file of functional code for Keras and TensorFlow model (see content descriptions below)
    - This includes source code for our Keras and TensorFlow models as well as Windows executable files to run the programs
- Final Report (PDF)

<u>Code contents</u>

*The two directories in bold contain the source code for the different neural network models:

- Keras Bidirectional LSTM model
- Gated Recurrent Language model by Miyamoto and Cho

**See instructions to run the executables in the next section below.

```
demo_char_lstm.exe  (customizable text generating LSTM to train and generate text samples)
generate_char_lstm.exe  (generates a 14 line sonnet from our trained baseline model)
generate_yoon_kim.exe  (generate a poem fragment from the Yoon Kim model)
README.txt
/data
    model_char_lstm.h5
    model_char_lstm.json
    sonnets.txt
    train.txt
    test.txt
    valid.txt
    epoch024_6.1626.model.data-00000-of-00001
    epoch024_6.1626.model.index
    epoch024_6.1626.model.meta
/source_code
    /website
        Dockerfile
        /app
            main.py  (Flask app)
            /static
                fonts.css
                default.css
                /fonts
                /images
            /templates
                index.html
                about.html
                useful_links.html
            /results
                line_number.txt  (saves the spot in the char_lstm_poems.txt file)
                char_lstm_poems.txt  (text file of generated poems)
    /demo_char_lstm  (customizable text generating LSTM to train and generate text samples)
        demo_char_lstm.py
        README.txt
        /data
            sonnets.txt
```

- /generate_char_lstm (generates a 14 line sonnet from our trained baseline model)
    - generate_char_lstm.py
    - convert.py
    - README.txt
- /generate_yoon_kim (generate a poem fragment from the Yoon Kim model)
    - generate_yoon_kim.py
    - README.txt
- **/char_lstm_model** (Keras Bidirectional LSTM Model)
    - utils.py (reads and processes the data from the text files)
    - test.py (tests a trained model from file)
    - train_BI_3_512_ND_adam.py (trains a 3-layer bidirectional model with 512 nodes)
    - sample.py (generates text from a trained model file)
    - README.txt
    - /models
    - /results
    - /checkpoints
    - /logs
    - /data
        - train_sonnets.txt (124 sonnets used for training)
        - test_sonnets.txt (15 sonnets used for validation during training)
        - val_sonnets.txt (15 sonnets used for testing)
        - sonnets.txt (text file containing all 154 sonnets)
- **/gated-rlm** (Gated Recurrent Language Model)
    - README.txt
    - base.py (basic helper functions & Keras' Progbar object to visualize training)
    - data_preprocess.py (from M & C source code, some functions were modified)
    - layers.py
    - gated_model_w_layers.py (an initial, messy implementation of the model)
    - gated_rlm.py   (a cleaner implementation of the model)
    - train.py (training script, though most of the training logic still resides in gated_rlm.py)
    - wordChar_prep.py (various collected helper utility functions)
    - /data
        - /MnC_dicts
            - char_dict.pkl (character vocab for the model from M & C script)
            - word_dict.pkl (word vocab for the model from M & C script)
        - /Yoon_dicts
            - char_vocab.pkl (character vocab for the model from Yoon Kim's script)
            - Word_vocab.pkl (word vocab for the model from Yoon Kim's script)
        - /Basic_dicts
            - char_dict.txt (simple char vocab without indices, one char per line)
            - word_dict.txt (word vocab with occurrences, for GloVe)
        - /shakespeare_corpus
            - /raw

```
                citation-n-legal_stuff.txt
                shakespeare_complete.txt
                Shakespeare_sonnets.txt
        /tokenized
                shakespeare_tokenized.txt  (complete works of shakespeare tokenized)
    /sonnets
        /raw  (our initial text files by Carrie)
                sonnets.txt
                train.txt
                test.txt
                valid.txt
        /tokenized  (same files as /raw, but tokenized)
                sonnets_tokenized.txt
                train_tokenized.txt
                test_tokenized.txt
                valid_tokenized.txt
    GloVe_vectors_trimmed.200d.npz  (trimmed word emb trained on entire corpus)
/tools
    build_dictionary_char.py  (build a char dictionary in cPickle, by Miyamoto et al)
    build_dictionary_word.py  (build a word dictionary in cPickle, by Miyamoto et al)
    tokenize_file.py  (preprocess files fed to the model, dictionary scripts, and GloVe)
/run_dir (empty folder from which to run the model to min. clutter from output)
```

Instructions to run the executables

These specific instructions to run the .exe files are also in the README.txt located in the same directory as the executables. Since our deep learning models require many different libraries and dependencies, we have included three Windows executable files of our source code that can be run on Windows without having to download anything.

The folder contains three Windows executable files: demo_char_lstm.exe and generate_char_lstm.exe, and generate_yoon_kim.exe. To run the programs, open Command Prompt in Windows and navigate to the folder that contains the executable files. The files rely on information stored in the data directory so they shouldn't be moved from this location. Below are instructions for running each program.

Program 1: demo_char_lstm.exe

A customizable text generating LSTM to train and generate text samples. Because our models are far too large to run on a regular CPU, we created this program to demonstrate how the training and tuning process works. The program can be used with the default input text file of Shakespeare's sonnets or with any text file specified in the input_text argument.

How to run:
Run with defaults in the Windows Command Prompt window type: demo_char_lstm
Run with optional arguments: demo_char_lstm —-epochs=5 —-file=model.h5 —-layers=2

Optional arguments:
--epochs: Number of epochs, type=int, default=1
--layers: Number of hidden layers, type=int, default=1
--nodes: Number of nodes per hidden layer, type=int, default=128
--batch_size: Number of streams processed at once, type=int, default=128
--seq_len: Length of each data stream, type=int, default=50
--optimizer: Optimizer: adam, sgd, rmsprop, type=str, default=adam
--model_type: Type of model: lstm, gru, bidirectional, type=str, default=lstm
--dropout: Dropout value between 0 and 1, type=float, default=0.0
--input_text: Path to input text, type=str, default=data/sonnets.txt
--sample_len: Length of generated text in chars, type=int, default=500
--diversity: Diversity of output between 0 and 1.5, type=float, default=0.5
--load: Path to load model from file, type=str, default=None
--save: Path to save model to file, type=str, default=None
--checkpoints: Interval for saving checkpoint files. They can be used to load the model from a specific epoch. type=int, default=1
--logs: Save log file for use with TensorBoard (0 is false, 1 is true), type=int, default=1)
--reduce_lr: Reduce learning rate on plateau (0 is false, 1 is true), type=int, default=0)

Suggested configurations:
The default arguments create a small model designed to be run on any CPU. As a result, the text output will not be very interesting. To create better results, test out different configurations using the optional arguments. For example:
- Increase the number of epochs
- Increase the number of hidden layers to 2 or 3
- Increase the number of nodes to 256 or 512
- Decrease batch_size to 64 or 32.
- Increase or decrease seq_len
- Save the model and reload it using —-save and —-load. This is helpful for saving and resuming training progress, especially when using slower CPUs.

The source code can be found in source_code/demo_char_lstm/demo_char_lstm.py

Program 2: generate_char_lstm.exe
This program generates a sonnet from our trained baseline model. The baseline model is a character level Bidirectional LSTM. Because of it’s large size, it can take up to a few minutes to generate the poem depending on the speed of the CPU.

How to run:
In the windows Command Prompt window type: generate_char_lstm
The source code can be found in source_code/generate_char_lstm/generate_char_lstm.py
The source code for our baseline model can be found in source_code/char_lstm_model

Program 3: generate_yoon_kim.exe
This program generates a sonnet from the Yoon Kim character aware word-level CNN-LSTM model that we trained but did not write. It is included only as an example of what we hoped to achieve with the output of our gated-rlm TensorFlow model and to compare output with our char lstm model. Because of it's large size, it can take up to a few minutes to generate the poem depending on the speed of the CPU.

How to run:
In the windows Command Prompt window type: generate_yoon_kim

The source code we used for the Yoon Kim character aware word-level CNN-LSTM model can be found here: https://github.com/mkroutikov/tf-lstm-char-cnn
The source code for this executable is in source_code/generate_yoon_kim
The source code for our TensorFlow Gated Recurrent Language Model can be found in source_code/gated-rlm

About the Keras Bidirectional LSTM Model

The source code for the Keras model can be found in source_code/char_lstm_model.
Our first model was based on Andrej Karpathy's suggestion that language generating LSTM models perform their best with 3 layers consisting of 512 memory cells each. [7] It's a char-level model (versus word-level) with no dropout that uses the Adam optimizer. After 29 epochs it reached a loss of 0.003. The following is an excerpt of text that was generated using this first model:

that i and the world see thy sweet seem store,
and therefore hath his good where thou art gene.
but why who hos hasce be it not a canne,
all that the rearest of all thou art,

We experimented with various baseline models (Bidirectional, GRU and LSTM) and adjusted the number of layers and nodes and found that a Bidirectional char-level LSTM with 3 layers and 512 memory cells produced the best output. After only 14 epochs it reached a loss of 0.0019. In addition to having lower loss values, this model had fewer spelling mistakes in the generated text. The following is an excerpt of text that was generated using this model:

for then a primities of the fair me bleat,
which thou art thou thou art thou roses being which is doth spect:
i sing i learned to be the praise thee
and thou canners hand the treasure every where:

The process of creating different models and tuning the hyperparameters is a slow and tedious process because it requires adjusting one parameter at a time, re-training the model, evaluating the results, and repeating. This allowed us to analyze whether adjusting one particular variable was creating better or worse output. We were initially training on our own personal computers, but the amount of resources required forced us to look to AWS for computing power and even then our models would take several hours to train each time.

One of the parameters we researched and experimented with is the shuffle parameter in the fit function of the Keras model. LSTM networks are stateful, but by default Keras resets the network state after each training batch. This suggests that if we had enough memory and a batch size large enough to hold all possible input patterns, the LSTM could use the context of the sequence within the batch to learn the sequence. Keras shuffles the training dataset before each training epoch, so we experimented with disabling shuffling to see if the model learns better by keeping the data sequences sequential. [3]

We also experimented with the sequence length and batch size. By providing more sequence or more previous time steps to the network, we are giving the model more context to work with, letting it learn the interdependencies between characters rather than explicitly defining them and having the network memorize them. But this means it’s slower to train, thus we needed to find the balance between training time and quality of output.

About the Gated Recurrent Language Model

Miyamoto and Cho developed a great model [11] that we hoped to be able to modify with attention. The Idea being that adding an attention module linking the character-level LSTM at the input level and the language modeling LSTM at the output level might allow the model to “pay attention” to the iambics and rhyme scheme of Shakespeare’s sonnets. We feel that adding attention might have been enough for a language modeling neural network to finally master Shakespeare’s style.

The model is implemented as follows (image from the publication [11]):

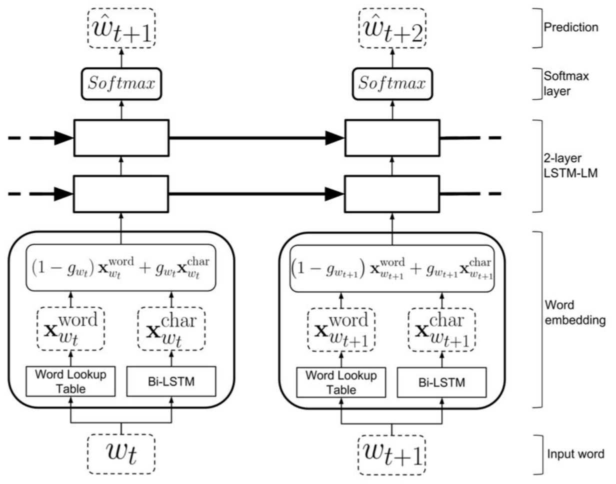

$\hat{w}_{t+1}$
$\hat{w}_{t+2}$
Prediction
$Softmax$
Softmax layer
2-layer LSTM-LM
$(1-g_{w_t})\,\mathbf{x}_{w_t}^{\text{word}} + g_{w_t}\mathbf{x}_{w_t}^{\text{char}}$
$(1-g_{w_{t+1}})\,\mathbf{x}_{w_{t+1}}^{\text{word}} + g_{w_{t+1}}\mathbf{x}_{w_{t+1}}^{\text{char}}$
$\mathbf{x}_{w_t}^{\text{word}}$
$\mathbf{x}_{w_t}^{\text{char}}$
$\mathbf{x}_{w_{t+1}}^{\text{word}}$
$\mathbf{x}_{w_{t+1}}^{\text{char}}$
Word embedding
Word Lookup Table
Bi-LSTM
w_t
w_{t+1}
Input word

Figure 1: The model architecture of the gated word-character recurrent language model. w_t is an input word at t. $\mathbf{x}_{w_t}^{\text{word}}$ is a word vector stored in the word lookup table. $\mathbf{x}_{w_t}^{\text{char}}$ is a word vector derived from the character-level input. g_{w_t} is a gating value of a word w_t. $\hat{w}_{t+1}$ is a prediction made at t.

softmax function is defined as $f(x_i) = \frac{\exp x_i}{\sum_k \exp x_k}$.

Input

The input word (w_t) is embedded at the word and character level by a set of encoding functions that map the input to the corresponding word and character vocabulary indices and then reshape the data into a tensor of the appropriate dimensions.

GloVe Word Embeddings

The word look-up table in the model was implemented with GloVe trained vectors. The GloVe algorithm was developed by Pennington, Manning, and Socher. [13] The model was trained on the entirety of Shakespeare's tokenized corpus to represent the word-level embeddings. After training the GloVe embeddings on Shakespeare's complete corpus, we trimmed any word vector that did not match up to a word in the word vocabulary (word_dict.pkl) created from the sonnets corpus. In this way, the word vectors can capture more semantic meaning while being about 10% of the size. This presumably will yield better results with less computations at the lookup table.

Word Embedding

As described in the section above, word-level input is projected onto the GloVe [13] look-up table to obtain the word-embedding. Word-level embedding is represented by

$$\mathbf{x}_{w_t}^{\text{word}} = \mathbf{E}^{\top}\mathbf{w}_{w_t},$$

where *E* is the lookup table. It's dimensionality is the size of the vocabulary (every distinct word in the sonnet corpus) by the size of the word vector (dimensionality = *Vxd)*. This is transposed on a one-hot-vector representation of the input word.

The character-level input of each word is embedded at the character-level by passing through a Bidirectional LSTM (an LSTM neural network where the updates flow forward and backwards, using the diagram above it would be as if the horizontal arrows point in both directions). The character level embedding is represented by $\mathbf{x}_{w_t}^{\text{char}} = \mathbf{W}^f \mathbf{h}_{w_t}^f + \mathbf{W}^r \mathbf{h}_{w_t}^r + \mathbf{b},$ which is fairly standard notation in neural networks except for the fact that we have forward (*f*) and reverse (*r*) matrices. *W* is a matrix of the weights of the connections between the nodes (neurons) and *h* is the hidden state (here the last one at time-step *t*). *W* is the word and *t* is the timestep. *W* and *b* (bias, essentially it is just another weight) are trainable parameters calculated during training. The forward and reverse outputs of the character-level LSTM are then concatenated. The result of the concatenation is the character-level embedding of the word.

The input word w_t is now represented by two separate embeddings and at this point the two are combined by the beating heart of this model, the gate (g_{wt}):

$$g_{w_t} = \sigma\left(\mathbf{v}_g^\top \mathbf{x}_{w_t}^{\text{word}} + b_g\right)$$
$$\mathbf{x}_{w_t} = (1 - g_{w_t})\, \mathbf{x}_{w_t}^{\text{word}} + g_{w_t} \mathbf{x}_{w_t}^{\text{char}},$$

The gate is implemented with a sigmoid function to maintain nonlinearity within layers. Within the sigmoid function, bias scalar b_g is added to the weight vector V_g (essentially the same thing as *W* earlier) transposed on the word level embedding.

The value of this gate is independent of a time step, the calculated value is applied according to the word regardless of the context in which the word appears in this time step.

The second equation describes how the gate serves to mix the character and word-level vectors. Although this equation is simple enough now to not require an explanation, its power and derivation is amazing.

LSTM Language Model

The output of the gate is then fed into an LSTM language model, which they surprisingly optimized to be just 2 layers and “only” 200 hidden units (compared to Karpathy’s findings of a vanilla LSTM).

They drew their inspiration from Wojciech Zaremba, Ilya Sutskever, and Oriol Vinyals’ non-regularized LSTM in their 2014 paper, Recurrent Neural Network Regularization, they came up with a fairly vanilla LSTM to handle the decoding (text generation).

Since this is fairly common architecture, I won’t go into detail about the equations, but I’ll provide the diagram. The new equations represent “valves” of an LSTM that control what the memory cell should focus on, *f* is the forget gate, *i* is the input gate, and *o* is the output gate; *c* (the memory *c*ell) here is analogous to the hidden state vectors (*h*) in other architectures, *h* in an LSTM has to do more with the output. Note that in this context, sigmoid and hyperbolic tangent functions are applied elementwise (and similarly, $\odot$ is the Hadamard product, or element-wise multiplication of the vectors).

$$
\begin{aligned}
\mathbf{f}_t &= \sigma\left(\mathbf{W}_f \mathbf{x}_{w_t} + \mathbf{U}_f \mathbf{h}_{t-1} + \mathbf{b}_f\right) \\
\mathbf{i}_t &= \sigma\left(\mathbf{W}_i \mathbf{x}_{w_t} + \mathbf{U}_i \mathbf{h}_{t-1} + \mathbf{b}_i\right) \\
\tilde{\mathbf{c}}_t &= \tanh\left(\mathbf{W}_{\tilde{c}} \mathbf{x}_{w_t} + \mathbf{U}_{\tilde{c}} \mathbf{h}_{t-1} + \mathbf{b}_{\tilde{c}}\right) \\
\mathbf{o}_t &= \sigma\left(\mathbf{W}_o \mathbf{x}_{w_t} + \mathbf{U}_o \mathbf{h}_{t-1} + \mathbf{b}_o\right) \\
\mathbf{c}_t &= \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tilde{\mathbf{c}}_t \\
\mathbf{h}_t &= \mathbf{o}_t \odot \tanh\left(\mathbf{c}_t\right),
\end{aligned}
$$

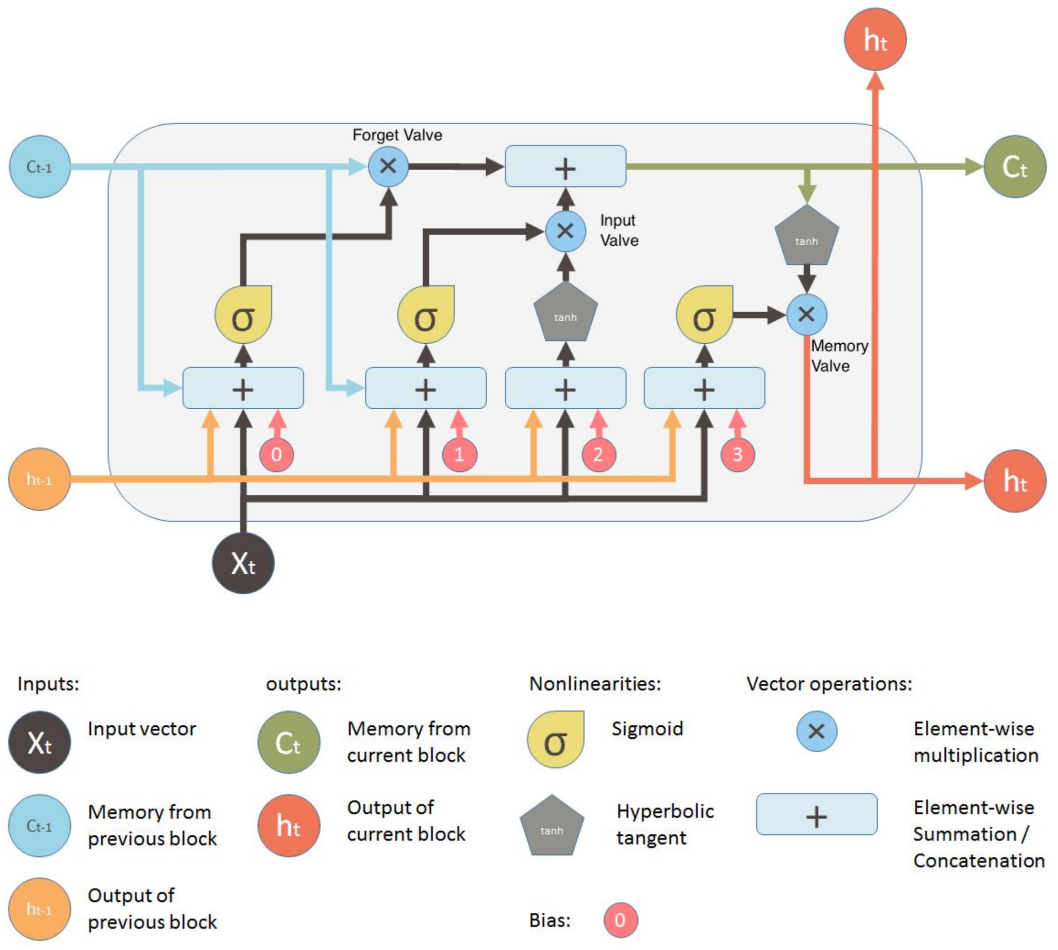


Image source: https://medium.com/@shiyan/understanding-lstm-and-its-diagrams-37e2f46f1714

Softmax Layer

Finally, the softmax layer predicts the output of the model! The output is a probability of word w_{t+1} given the previous word outputs.

$$\Pr\left(w_{t+1} = k | w_{<t+1}\right) = \frac{\exp\left(\mathbf{v}_k^\top \mathbf{h}_t + b_k\right)}{\sum_{k'} \exp\left(\mathbf{v}_{k'}^\top \mathbf{h}_t + b_{k'}\right)},$$

k here is an index, v_k it is the k-th column of parameter matrix *V* (the same vocabulary matrix used for the word level embeddings). Similarly, b_k is the k-th element of the bias vector.

Attention

Unfortunately we did not get far enough along on the project to be able to implement an attention mechanism. While we hope to still complete the project and add attention in the future, this model is still not nearly complete enough to be augmented with an attention wrapper.

Our hope is that attention would allow the model to pick up on the iambics and rhyme scheme that characterize Shakespearean sonnets. Essentially this is another vector (tensor) that the model can reference as it learns to see what might or might not be relevant to a particular memory cell at a particular time. This will create a context vector that the model can reference when it is predicting output. We hoped to implement it by having the model track the hidden states of the encoding LSTM (the character-level LSTM) and the decoding LSTM (the language modeling LSTM) in the model described by Miyamoto and Cho. In this way, the different hidden states would learn what to pay attention to and have that vector as a reference, our hope is that some of these would involve rhyme and meter since that is largely character-level data.

Attention works as follows [10]:
We begin by calculating a score:

$$\text{score}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_s) = \begin{cases} \boldsymbol{h}_t^\top \bar{\boldsymbol{h}}_s \\ \boldsymbol{h}_t^\top \boldsymbol{W_a} \bar{\boldsymbol{h}}_s \\ \boldsymbol{v}_a^\top \tanh\left(\boldsymbol{W_a}[\boldsymbol{h}_t; \bar{\boldsymbol{h}}_s]\right) \end{cases}$$

A few functions have been proposed, but the bilinear function (center) developed by Christopher Manning, one of the instructors of the NLP course at Stanford has shown a lot of success and adoption. H_s is the hidden state of the encoder (the word embedding block) and h_t is the hidden state of the decoder (the LSTM). Whereas the first equation shows the dot product of these two vectors, the middle one essentially mediates the dot-product with a mediating matrix W_a (which is learned during training). This will score each position and decide where to “pay attention”.

We then convert the score to alignment weights:

$$\boldsymbol{a}_t(s) = \frac{\mathrm{e}^{\text{score}(s)}}{\sum_{s'} \mathrm{e}^{\text{score}(s')}}$$

We accomplish this by piping the scores into a softmax function that will yield a probability distribution of how much attention to pay to the “different places” in the source.

Finally, we can build a representation that combines all of the memories weighted by the score:

$$\boldsymbol{c}_t = \sum_s \boldsymbol{a}_t(s)\bar{\boldsymbol{h}}_s$$

This is implemented as a context vector (c_t), essentially a weighted average that combines together all of the hidden states of the encoder weighted by how much attention we’re paying to “it”. c_t is weighted sum of of how much attention we pay to score *s,* at time *t,* multiplied by each hidden state vector.

<u>Where we are and where we hope to go:</u>
At the moment we seem to have been able to recreate the architecture of Miyamoto and Cho’s gated recurrent model. We have also been able to create and train the word embeddings for the lookup table so that they can be properly mapped to words in the word dictionary ({ word : index }). Once we understood exactly how the data had to be preformatted, we were able to write some basic scripts to handle that. We are currently in the process of writing the training algorithm for the model. Fortunately, it only took some minor tweaks of Miyamoto and Cho’s input processing functions to make their code for turning ASCII string input into numpy numeric arrays (by mapping the input to the word or char dictionary indices). These will still need to be reshaped before being fed into the input vectors in the training loop, but at the moment we’re focusing on having a complete training loop which we can then refine and fix.

Eventually, we would like to add some fully connected layers to the LSTM units of the model and the cost and optimisation functions still need work. Also, the sampling script is still blank so more research and coding will have to go into that so we can see how well the model performs. Once we have fully recreated their model and trained it to write coherent Shakespeare sounding lines, we hope to implement the attention wrapper outlined above to capture or refine the iambic pentameter and rhyme scheme of the sonnets. This attention wrapper had a key role in our hypothesis when we embarked on this journey. We would like to eventually show whether or not the attention wrapper will allow the model to mimic Shakespearean sonnets or whether that is still not enough to achieve that goal. It would also be interesting to see how well it performs at other language processing tasks, maybe it's not enough to allow it to pick up on complex language modeling tasks but perhaps the output will be more coherent than other models or something unexpected might come of it. Another thing to try will be to feed the sonnets in as individual files, as this might allow the model to further understand the rhyming structure of the sonnets.

Difficulties in Development:
Unfortunately, from the beginning this was a challenging undertaking. While we were graced with the fortune of having access to Miyamoto and Cho's source code (linked in their publication), understanding their Theano implementation was difficult. These two different libraries have very different specifications and behaviors so porting wasn't as straightforward as one might hope. Theano is popular in academia, but not elsewhere due to the extremely limited documentation and support base. In contrast, TensorFlow, developed and maintained by Google has amazing documentation and phenomenal peer support. The predominant challenge here was in developing the architecture of the model. Other challenges were more specific to TensorFlow, such as the sharing of variables across networks that shouldn't share these (this is why each layer has its own function and scope). Tensor shapes, dimensions, and the highly specific needs of TensorFlow's Bidirectional LSTM were other similar obstacles. An additional setback in the middle of completing the project is that TensorFlow changed some of its specifications about their conditional tensor statements and Bidirectional LSTM requirements. Currently, while attempting the coding loop, the main challenges have again been with the quality of the input data, where certain character orderings caused the training loop (still in development) to crash.

Data pre-processing and input was unexpectedly challenging and time-consuming, plaguing our group down to the last week. Everyone seems to have a different definition of tokenization and having everything match up was more difficult than anticipated until we realized the easiest thing to do was to create a function that does it for us (tokenize_file.py in the "tools" folder). It was important that the input texts, word and character vocabularies, and word embeddings properly recognize words and punctuation. Numbers had to be trimmed and spaces added so that a line like "Feed'st thy light'st flame with self substantial fuel," would become "Feed 'st thy light 'st flame with self substantial fuel ," otherwise the model would treat conjugations as unique words, even "fuel," would be considered a word.

Fortunately a lot of Miyamoto and Chos' functions seem to work with the data we need since they use numpy arrays to process the input. The only difference is that while Theano is time-major, Tensorflow is batch-major so we have to reshape them from a dimensionality of [number_of_time-steps, batch_size, vector_dimensions] to [batch_size, timesteps, vector_dimensions].

Overall, the issues can be attributed to a lack of familiarity with the concepts and libraries involved. The learning curve has been fairly steep, so while we're confident that we can eventually code this model and add the attention mechanism we had envisioned, we will need more time to complete this. Working on this has made it glaringly obvious why the students working on projects of this scope in Stanford's CS-224n course were required to have a PhD student mentor them through the process. This was a complicated architecture to build, formatting the data for the GloVe embeddings was more complicated than anticipated, and finally turning input data into batches and tensor arrays is also non-trivial. I suspect that converting the output from the softmax layer into legible output should be easier, as it should only involve reverse-engineering the input data processing, but if there is one thing we've learned throughout this project is that nothing is as easy or trivial as we would like to believe.

I think a lot of these challenges would have been mitigated with more time and/or familiarity with the subject. We spent about half the time rushing through material that would give us an understanding of the logic and implementation of neural network architectures. We also dove straight into the deep end in TensorFlow, going from no experience to attempting a rather challenging architecture. Luckily there was Miyamoto and Cho's original source code for guidance as well as the material from the Stanford course on Natural Language Processing with Deep Learning. These along with TensorFlow's excellent documentation and the many forum posts on similar issues proved to be invaluable resources as we trudged along. The development process has been slow and tedious as we iteratively code, test, revise, and add to the program. Eventually, we hope to prove whether or not adding an attention mechanism to this gated recurrent language model will allow it to pick up on the rhyme and meter structure.

About the Yoon Kim Character-Aware Word Level CNN-LSTM Model

Yoon Kim's model [8] is similar to the gated model from Miyamoto and Cho in that it receives text input at the word and character level simultaneously, but it does so in a very different way. Essentially what Kim did is link a Convolutional Neural Network (CNN) to an LSTM. So the words enter the CNN at the character-level and the word-level output is then fed to the LSTM. In this way it's as though Kim is combining the word lookup table and the character-level LSTM (and the gate) in Miyamoto and Cho's model into a CNN. The characters enter the CNN and a "word embedding" leaves. This "word embedding" is analogous to the output of the gate layer in the Miyamoto and Cho model. When it was tested alongside Miyamoto and Cho's model, the two models performed roughly on par. [12]

Since the two models perform similarly, we trained a TensorFlow implementation of the Yoon Kim model on Shakespeare's sonnets to provide a general idea of what the output of the gated recurrent language model should look like (without attention). The following is an excerpt of text that was generated using this model:

As flowers posterity? of worth, again, I perceived;
Then his my steep-up is my look I compare
My you would rude, from my see mind's be
Die to seen that Time: old hugely tribes:
The oft so believe of you violet lines
To slander's of all this seemly violet me,
You subject it than and taken.
But thou desire actor thy review travel's thy With powers
When acceptance to ten not I breast,
As to thee, disgrace may thou abide;

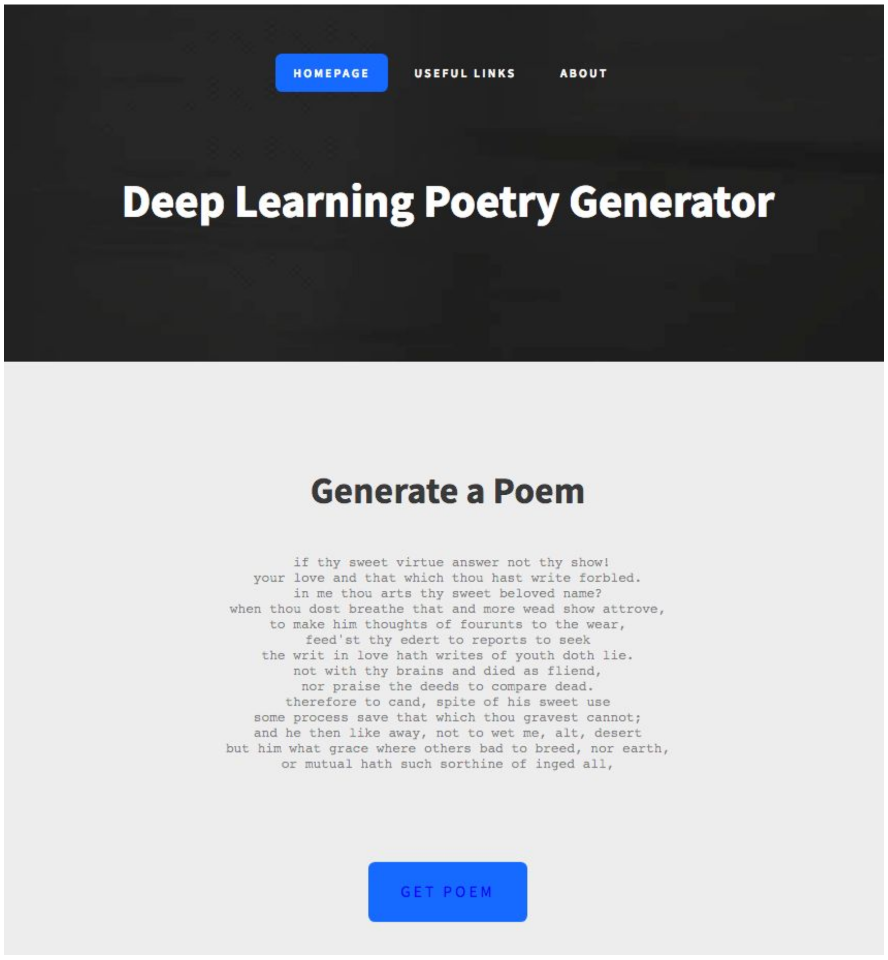
The see or my so; this journey to hideous gone,
So Weary the loves even they by before of bootless
And by view myself thy thing together
What heavy stand to oblation, and thy fragrant pen,
Yet let for thoughts, that the clearer their seemly
That by sad time your is thinly in the alone.

So vulgar 'tis that thou Weary another thou longer
To precious that the pencil, in with gone doth deserts?
To Fairing yet times thou best place.

O, I say fill'd come myself dear the come,
To your my head, couldst like this thee
Against me. nor for let with face wind;
Even not alone shadow he days.
My seeing father image kingdoms eyelids
And barren on fair that bounty disgrace.
end? thou with praise face:
Such make lies power that time wilfulness youth,
That love, all hooks, her fell manners doth housewife
Even that, that out do, miser some and all prevailed?
Thyself affections and place, now verse griefs
Yourself on were my death, though by heart's weep
lamb mortal no unfair are thy lie.
usury, known

Website

URL: http://ec2-18-217-70-169.us-east-2.compute.amazonaws.com/



```
if thy sweet virtue answer not thy show!
your love and that which thou hast write forbled.
in me thou arts thy sweet beloved name?
when thou dost breathe that and more wead show attrove,
to make him thoughts of fourunts to the wear,
feed'st thy edert to reports to seek
the writ in love hath writes of youth doth lie.
not with thy brains and died as fliend,
nor praise the deeds to compare dead.
therefore to cand, spite of his sweet use
some process save that which thou gravest cannot;
and he then like away, not to wet me, alt, desert
but him what grace where others bad to breed, nor earth,
or mutual hath such sorthine of inged all,
```

We wanted to create a user interface if the user was interested in getting generated poems from our trained model. We made a simple web application using Flask framework and Jinja2 templating and deployed it on an EC2 instance with Docker and nginx.
Python has a built in server which is useful for debugging purposes, but it can only handle one request at a time and the app closes when the connection terminates and then must be manually restarted. There were many ways of deploying our app but a lot of them seemed overwhelming, relying on many different components that seemed excessive for our needs. We found a simple tutorial for deploying a Flask app with Docker and nginx that seemed less complicated and used it as an outline to get our app deployed. [9]
We deployed an EC2 instance which ran a Docker container that pulls an image from the Docker Hub repository and installs the packages we specify in the Dockerfile. The image is the base of our container and the one we're using was created by tiangolo [14] which has Python, Flask, Nginx and uWSGI installed on a Debian OS. It sets up a Debian Jesse virtual operating system in our docker container which will run on our Linux Machine in AWS. The Dockerfile specifies packages and libraries needed to run our programs like TensorFlow and Keras so when we build the docker container it will read the Dockerfile line by line to pull the base image, copy our code into the container, and install all the specified packages so we have our own custom container. We can then run the container within an EC2 instance and anyone can view

our running flask app from the public server IP address in their browser. Moreover, we can run it persistently by running the container as a daemon in the background and automatically restart it if the container crashes or the the system is rebooted.
We initially set it up so that each the the user clicked the button, it would run sample.py to generate a new poem from our trained model. Unfortunately, running sample.py usually took a few minutes and the request would time out, so we brainstormed options and our solution was to create a text file containing several poems generated from our model. This solution meant that the user would still get a new poem each time the button was clicked and also didn't have to wait over a minute to get it.

Classes/Tutorials

Given that the team had no experience with deep learning or neural networks, our project required a lot of learning and research. The following were some resources we utilized:

Udemy Course- Artificial Intelligence A-Z: Learn How to Build an AI [6]
Each of us purchased this course and it provided a good overview of all the different aspects of deep learning. We learned about the different types of neural networks, how they worked and how they were trained. This was one of the first resources that gave us the idea that LSTM (or Long Short-Term Memory) neural networks was a good option for our Poem Generator.

O'Reilly Fundamentals of Deep Learning [4]
In addition to the Udemy course, we referenced a few textbooks. This was one of the more helpful ones. While the Udemy course was good at teaching us high-level concepts, this book talks more in-depth about the mathematical formulas and variables behind neural networks.

Stanford Course- CS224n: Natural Language Processing [10]
This course was a gold mine of information especially because it focused on natural language processing which is what we were trying to achieve with our poetry generator. It went into the nitty gritty details of the core of neural networks, how words and sentences are represented as vectors and the computations behind the hidden layers and back propagation.

The Unreasonable Effectiveness of Recurrent Neural Networks [7]
http://karpathy.github.io/2015/05/21/rnn-effectiveness/
Andrej Karpathy is the Director of AI at Tesla and we found his blog and journals immensely helpful. In this popular article from his blog he trains a character-level language model on text ranging from Paul Graham essays to Wikipedia and books on algebraic geometry, even Linux source code! He also has neat graphics showing the predictions of characters as the neuron fires.

Text Generation with LSTM Recurrent Neural Networks in Python with Keras [2]

https://machinelearningmastery.com/text-generation-lstm-recurrent-neural-networks-python-keras/
https://github.com/fchollet/keras/blob/master/examples/lstm_text_generation.py
This article and corresponding code was a very helpful guide in how to develop an LSTM recurrent neural network for text generation in Python using the Keras library. We adapted our Keras model to this code. We were able to experiment with different model configurations and alter the parameters for each model to see which model and corresponding configurations produced the best output.

Gated Word Character Recurrent Language Model [11]
https://arxiv.org/pdf/1606.01700.pdf
This paper by Miyamoto and Cho describes a gated word-char LSTM which has shown success with natural language processing. Our TensorFlow model was based on their description and low-level Theano code.

Character-Aware Neural Language Models [8]
https://arxiv.org/pdf/1508.06615.pdf
This paper by Kim, Jernite, Sontag and Rush describes a neural language model that relies only on character-level inputs. Their model uses a convolutional neural network (CNN) and a highway network over characters whose output is passed to a long short-term memory (LSTM) recurrent neural network language model (RNN-LM). Predictions from the model are still made at the word-level. Their analysis of this model shows that it is able to encode, from characters only, both semantic and orthographic information despite having 60% fewer parameters.

Software Libraries, Development Tools, etc.

For creating neural network models:
Keras
Keras is a high-level neural networks Python API which runs on top of TensorFlow. We chose this API to work with as our introduction to neural networks because it was a relatively simple API which was user friendly and allowed for fast and easy experimentation.
TensorFlow
TensorFlow is an open-source software library for Machine Intelligence. Originally developed by the Google Brain Team within the Machine Intelligence research organization it utilizes data flow graphs to implement numerical computations. The nodes in the graph represent mathematical operations, while the edges represent multidimensional data arrays (or tensors) communicated between them.

For Creating and Training Word Embeddings
GloVe
Glove (Global Vectors for Word Representation) is an unsupervised learning algorithm for obtaining vector representations of words. It was developed by Jeffrey Pennington, Richard

Socher, and Christopher Manning at Stanford University. [13] GloVe is similar to Word2Vec, but it performs at a higher, obtaining better word relationships from co-occurrences in the training corpus. That is to say, words with similar meanings fall closer to each other in the vector space, related words will fall closer together, whereas words that are completely unrelated (like eyedrops and computer are likely to be farther apart).

For running and training neural network models:
AWS
Amazon Web Services is a secure cloud services platform offering solutions like computing power, database storage, and content delivery. We utilized a free EC2 t2.micro instance to host our web application where there is a convenient user interface for the user to get generated poems.
To train our neural network models, we purchased a Deep Learning Linux AMI configured with NVIDIA CUDA 9, cuDNN 7, NCCL 2.0 and NVIDIA Driver 384.81. It also comes with MXNet, Caffe2 and TensorFlow already installed allowing for a high performance execution environment for deep learning. We launched the AMI on a p2.xlarge instance type which comes with 12 EC2 Compute Units (4 virtual cores) and 1 NVIDIA K80 (GK210) GPU. This product has an hourly fee structure with additional fees for SSD storage so we had to terminate and repurchase the AMI as needed. [1]

Web application:
Docker
Docker provides the ability to package and run an application in an isolated environment called a container. Docker uses a powerful client-server architecture. The Docker client talks to the Docker daemon, which does the building, running, and distributing of your Docker containers. The communication between client and daemon is done using a REST API, over UNIX sockets or a network interface. The Docker daemon (dockerd) listens for Docker API requests and manages Docker objects such as images, containers, networks, and volumes. The Docker client (docker) is how many Docker users interact with Docker. The docker command uses the Docker API, so when you use commands like docker run, the client sends these commands to dockerd, which carries them out. When you use Docker, you are creating and using images, containers, networks, volumes, plugins and other objects. An image is a read-only template with instructions for creating a Docker container. We built an image based on the Docker image created by tiangolo with uWSGI and Nginx for Flask application in Python [14], but also install libraries like TensorFlow and Keras and our applications, as well as the configuration details needed to make our application run. A container is a runnable instance of our image and we can create, start, stop, move, or delete a container using the Docker API or CLI. In our case we use the Docker CLI and run it within and AWS EC2 instance. Having our own customized container was especially convenient given the numerous packages and dependencies that were not only complicated to set up, but also required a lot of memory. With Docker we could have potentially created our container, pushed it to the docker registry, and have our own environment neatly packaged in a container and easily retrieved from anywhere with the command docker pull (as

long as Docker is installed). The underlying technology behind Docker is fascinating and we wish we had found this open platform sooner! [5]

Flask/Jinja2

Flask is a microframework for Python. We chose this over other frameworks because it is very well documented and has a low learning curve with Jinja2 templating. All we needed was a simple UI for our poem generator and this fit the bill. We also found their built-in development server and debugger very useful. [15]

Team Member Contributions

Keisha Arnold

- Progress Report Videos and Scripts
- Website, Model hosting, and UI logic for online demonstration
- Assisted Jacob with data preprocessing functions
- Helped debug functions in gated recurrent language model
- Trained and tuned Yoon Kim's character aware word-level CNN-LSTM

Jacob Karcz

- Helped Train and Tune Keras Bidirectional LSTM model
- Helped with Progress Reports
- Pre-processed input data (text tokenization, char/word vocabularies)
- Trained GloVE embeddings for gated rlm on entire Shakespeare corpus and trimmed unused word vectors
- Worked on TensorFlow implementation of Gated Recurrent Language Model (architecture, data preprocessing, training, sampling)

Carrie Treeful

- Created, trained, and tuned the Keras Bidirectional Char LSTM model
- Compiled sonnets for training, testing, and validation
- Created demo_char_lstm and generate_char_lstm programs
- Created the Windows executable files

Conclusion

The underlying motivation for this project was because the team had a mutual interest in wanting to learn more about machine learning and AI. We knew that this project would be very research heavy since none of us had experience with neural networks, but coming from zero experience I think we were a bit unprepared for the sheer amount of information and the steep learning curve involved not only with the subject matter, but also with the required hardware, libraries, and associated tools. Deep learning in itself was new to us, but our poem generator also required Natural Language Processing, which, while fascinating, added another complicated layer given the nuances behind language.

I believe we can say we were successful in that we crammed as much information as we could in our brains in 10 weeks and definitely learned more about machine learning. We were able to successfully create, train, test, and retrieve generated text from a model, albeit not with perfect syntax and rhyme. But really when you think about where we started it is quite fascinating. We did not hard-code any poem specifics into our program. All we did was build a neural network model with a certain number of layers, nodes, and parameters then fed it a bunch of Shakespearean poems. In the beginning, we got random characters strung together as gibberish, but after countless hours of training and fine tuning, the random characters started forming somewhat recognizable words, then finally actual words! The model learned all on it's own what order it needed to place the characters to make words. Our attempt to take that a step further and have it learn what order to place those words to capture correct syntax and rhyme fell just short, however, but we feel it was an ambitious goal to begin with and with a bit more time we think we can get achieve that end goal.
Overall this experience has been a great learning experience for us and though at times it was frustrating and overwhelming, we feel so much more enriched by having gained all this new information on our own and with additional tools in our toolbox.

References

[1] AWS Marketplace (2017). Launch on EC2: Deep Learning AMI with Source Code. [online]. Available at: https://aws.amazon.com/marketplace/fulfillment?productId=01f9cbdd-0d95-4833-af87-68a8294c187e&ref_=dtl_psb_continue&region=us-west-2 [Accessed 25 Oct. 2017].

[2] Brownlee, J. (2016). Text Generation With LSTM Recurrent Neural Networks in Python with Keras. [Blog] Machine Learning Mastery. Available at: https://machinelearningmastery.com/text-generation-lstm-recurrent-neural-networks-python-keras/ [Accessed 2 Oct. 2017].

[3] Brownlee, J. (2016). Understanding Stateful LSTM Recurrent Neural Networks in Python with Keras. [Blog] Machine Learning Mastery. Available at: https://machinelearningmastery.com/understanding-stateful-lstm-recurrent-neural-networks-python-keras/ [Accessed 2 Oct. 2017].

[4] Buduma, N. (2017). Fundamentals of Deep Learning. 1st ed. Sebastopol (CA): O'Reilly.

[5] Docker docs (2017). Docker Overview. [online]. Available at: https://docs.docker.com/engine/docker-overview/ [Accessed 10 Nov. 2017].

[6] Eremenko, K. and de Ponteves, H. (2017). Deep Learning A-Z™: Hands-On Artificial Neural Networks. [online] Udemy. Available at: https://www.udemy.com/deeplearning [Accessed 25 Oct. 2017].

[7] Karpathy, A. (2015). The Unreasonable Effectiveness of Recurrent Neural Networks. [Blog] Andrej Karpathy blog. Available at: http://karpathy.github.io/2015/05/21/rnn-effectiveness/ [Accessed 2 Oct. 2017].

[8] Kim, Y., Jernite, Y., Sontag, D. and Rush, A. (2015). Character-Aware Neural Language Models. [online] Available at: https://arxiv.org/abs/1508.06615 [Accessed 28 Oct. 2017].

[9] London, Ian. (2017). Deploying Flask Apps Easily with Docker and Nginx. [online] Available at: http://ianlondon.github.io/blog/deploy-flask-docker-nginx/ [Accessed 10 Nov. 2017].

[10] Manning, Chris and Socher, Richard. (2017). CS224n: Natural Language Processing with Deep Learning. [online] Stanford University. Available at: http://web.stanford.edu/class/cs224n/ [Accessed 25 Oct. 2017].

[11] Miyamoto, Y. and Cho, K. (2016). Gated Word-Character Recurrent Language Model. [online] Available at: https://arxiv.org/abs/1606.01700 [Accessed 27 Oct. 2017].

[12] Xie, S., Rastogi, R. and Chang, M. (2017). Deep Poetry: Word-Level and Character-Level Language Models for Shakespearean Sonnet Generation. [online] Available at: http://web.stanford.edu/class/cs224n/reports/2762063.pdf [Accessed 28 Oct. 2017].

[13] Pennington, J., Socher, E. and Manning, C. (2014). GloVe: Global Vectors for Word Representation. [online] Available at: https://nlp.stanford.edu/pubs/glove.pdf [Accessed 28 Oct. 2017].

[14] tiangolo. (2017). Docker image with uWSGI and Nginx for Flask applications in Python running in a single container. [online] Available at: https://github.com/tiangolo/uwsgi-nginx-flask-docker [Accessed 10 Nov. 2017].

[15] Ronacher, Armin. (2017). Flask web development, one drop at a time. [online] Available at: http://flask.pocoo.org [Accessed 10 Nov. 2017].