Tackling Toxicity with Deep Learning

Hoon Shin

November 17, 2019

1 Abstract

A couple years ago, Google's Conversation AI team released a massive dataset of internet comments labelled by toxicity. These comments were drawn from a diverse range of conversations to best reflect the use of words and phrases in different contexts. This paper will compare the ability of various neural network architectures to determine the toxicity of these comments. These toxicity networks could be used for important applications in the future like moderating online conversations in real-time by filtering out offensive content.

2 Introduction

The dataset for this project consisted of roughly 2 million comments. Each comment was reviewed by 10 human annotators, who classified it as Not Toxic, Hard to Say, Toxic, or Very Toxic. The Conversation AI team then found the fraction of comment ratings that fell under Toxic or Very toxic to get its toxicity score.

The data was provided in table form with each entry containing a comment and its corresponding target (toxicity score). The goal of the machine learning model would be to predict the target of a comment based on its text.

3 Preprocessing 1.0

Preprocessing is a common step in ML projects and involves manipulating data into a form that can be easily understood by a computer model. For the dataset used in the project, the comment text was initially very messy, and contained lots of unnecessary punctuation, symbols, emojis, and internet slang (Fig. 1).

```
7                                          FFFFUUUUUUUUUUUUUUU
36          Not for long!  \n\n(Troll-In-Training since 2016)
1804808     https://www.cbpp.org/research/federal-tax/repu...
44          YET ANOTHER BARACK OBAMA LIBERAL MEDIA CONSPIR...
1804827     Like this?\n\nhttp://www.crazydaysandnights.ne...
1804833                                                Yawn 😴
1804834                        Now post as Thuggernaut.\n\n😐
1804783                                  Stay classy dems 😅
Name: comment_text, dtype: object
```

Figure 1: raw data example

To remove the excess content, a function format_comments was defined. This function took a comment as an input and returned just the words in the comment with punctuation, digits, and control characters removed.

```
import re
import string
def format_comments(comment):
  text = comment.lower()
  text = re.sub(r"""(?i)\b((?:https?:(?:/{1,3}|[a-z0-9%])
  |[a-z0-9.\-]+[.](?:com|net|org|edu|gov|mil|aero|asia|biz|cat
  |name|post|)\b/?(?!@)))""", '', text)
  text = re.sub(r'[\n\r\t]', '', text)
  text = text.strip()
  text = ''.join(i for i in text if not i.isdigit())
  text = ' '.join(word.strip(string.punctuation) for word in text.split())
  return text
```

format_comments was applied to every comment in the dataset, and its outputs formed the words column (Fig. 2).

	target	comment_text	words
0	0.000000	This is so cool. It's like, 'would you want yo...	[this, is, so, cool, it's, like, would, you, w...
1	0.000000	Thank you!! This would make my life a lot less...	[thank, you, this, would, make, my, life, a, l...
2	0.000000	This is such an urgent design problem; kudos t...	[this, is, such, an, urgent, design, problem, ...
3	0.000000	Is this something I'll be able to install on m...	[is, this, something, i'll, be, able, to, inst...
4	0.893617	haha you guys are a bunch of losers.	[haha, you, guys, are, a, bunch, of, losers]

Figure 2: table example

Even with the comments formatted, there was still too much data to train a model on with the limited processing power that I had access to. So to shrink the size of the dataset, only comments with 8 words were selected. A length of eight was chosen since it would be long enough for a model to determine the toxicity of the comment, but not so long that the model would lose accuracy. Loss of accuracy usually occurs with more words since machine learning models cannot determine which parts of a comment are important, so relevant details may be overwhelmed by useless information.

The comments of length 8 were split into clean comments with a target rating of 0 and toxic comments with a target rating greater than 0.

```
#selects toxic comments 8 words long
train_toxic = train.loc[(train.words.map(len) == 8) & (train.target > 0)]


#selects clean comments 8 words long
train_clean = train.loc[(train.words.map(len) == 8) & (train.target == 0)]
```

Next, the comments were broken into two datasets, a training set and a development set. The training set is what the models were trained on, while the development set was used for comparing the performance of already trained models. The reason why the training set can't be used for both purposes is that high accuracy on the training set doesn't always translate to high accuracy overall on a task. Take toxicity classification for example: A model may be excellent at identifying the toxicity of comments in the training set because it has picked up on very detailed patterns in the training data but could perform

poorly when it encounters other data which doesn't follow the same very specific patterns.

Say in the training set that the word 'pickle' occurs in multiple comments with a high toxicity rating. Though the word 'pickle' is not inherently toxic, the model might learn that it is since it has only seen pickle used in toxic contexts. As a result, it might see a comment such as 'pickles are tasty' and incorrectly give it a high toxicity rating based on inferences from the training data. For this reason, an unbiased second dataset, the development set, is needed to reliably assess the performance of different models.

```
#initialize x_train, y_train
X_train, Y_train = train['words'].values, train['target'].values


#initialize x_dev, y_dev
X_dev, Y_dev = dev['words'].values, dev['target'].values
```

Once the comments were split into train and dev sets, the Tokenizer function was applied. Tokenizer builds a dictionary of all the unique words and characters in a given text (in this case, the comments) and assigns each of them an index ranked by number of occurrences. The more common a word is the lower its index. So words like 'the', 'and', and 'as' were given indices close to one while a term like 'berniebro' which only occurred once had a much higher index. What's useful about the indexing is that it converts words which are easily interpreted by humans into numbers which are easily interpreted by a computer model. After the Tokenizer dictionary was built, all of the words were converted into their corresponding indices with the texts_to_sequences method.

```
#create a token dictionary of all the words used in the training and dev sets
tokenizer = text.Tokenizer(filters='')
tokenizer.fit_on_texts(list(X_train) + list(X_dev))


#convert the words to corresponding numbers in dict
```

```
X_train = tokenizer.texts_to_sequences(X_train)
X_dev = tokenizer.texts_to_sequences(X_dev)
```

At this point, preprocessing was finished and I tried training some toxicity models with the comments. Unfortunately, all the models performed poorly, which was likely caused by too much loss of information in the preprocessing step. I suspect that without emojis, internet slang, and contractions in the comments, the model was unable to make any meaningful connections between text and toxicity.

4 Preprocessing 2.0

For my second attempt at preprocessing, I took a more conventional approach and looked at what machine learning researchers had done in the past.

Before we continue, let's take a moment to review an important concept in natural language processing: word embeddings. Word embeddings are dictionaries which contain millions of words which each correspond to a list of numbers called a word vector. You can think of each number in the word vector as representing a characteristic of the word. For instance, the 1st number in the vector could represent gender, with words like man and king having negative values, words like woman and queen having positive values, and words with no clear gender having zero values. Similarly, the second word could represent the tone of the word, third could represent the part of speech...etc. In practice, a machine learning model decides the values in a word vector, so it is unlikely that each number in the vector falls neatly into a single category like gender. Rather, each number probably represents a hybrid of multiple characteristics of the word. The important takeaway is that word embeddings encapsulate the information of a word into a compact form which a computer can easily analyze.

There are many different high-quality word embeddings that are publicly available. For this project, I used a word embedding developed by Facebook called FastText. Benefits of using FastText include efficiency, speed and accurate

vectors for rare words. An example of a FastText embedding for QED is shown below.

```
array([ 2.380e-02,  2.600e-03, -1.212e-01,  1.954e-01,  1.321e-01,
       -8.820e-02, -2.700e-02, -9.350e-02,  2.074e-01,  5.040e-02,
        8.190e-02,  6.200e-03, -6.470e-02,  1.095e-01,  2.390e-01,
                ..........
        1.238e-01, -1.507e-01, -9.240e-02,  1.037e-01, -2.460e-02,
       -2.968e-01,  8.000e-04, -2.320e-02,  1.682e-01, -7.550e-02,
       -9.360e-02],
      dtype=float32)
```

Figure 3: embedding of 'QED'

A common step I saw in preprocessing code was to construct build_vocab and check_coverage functions. build_vocab takes a list of strings (a string is a datatype that is used for text) as input, and returns a dictionary of all the unique words in the text and their number of occurrences. check_coverage takes a vocab list and word embedding as inputs and determines the percentage of words and total text from the vocab that is in the embedding as well. It also prints out the ten most common words in the vocab which are not in the embedding. This list is called the oov.

What's useful about these functions is that they show the exact percentage of comments which can be interpreted by a word embedding, and in turn, a machine learning model. The goal is to maximize the coverage of the embedding so the model can extract as much valuable information from the comment text as possible.

Before doing any preprocessing steps, I ran the build_vocab and check_coverage functions to establish a baseline for the coverage percentages (Fig. 4)

```
Found embeddings for 13.97% of vocab
Found embeddings for  89.18% of all text
[("don't", 178881),
 ("it's", 100959),
 ("I'm", 82126),
 ("It's", 82038),
 ("doesn't", 60810),
 ("can't", 60616),
 ("didn't", 52169),
 ("isn't", 39964),
 ("that's", 38519),
 ("That's", 37640)]
```

Figure 4: baseline check_coverage

To improve the coverage of the word embedding, I removed all symbols from the dataset that were not in the embedding. This was accomplished in two steps.

(1) Symbols recognized by the FastText embedding were stored in a fast_symbols variable. Symbols in the dataset were stored in a data_symbols variable.

(2) Symbols in fast_symbols and data_symbols were assigned to symbols_to_keep while symbols only in data_symbols were assigned to symbols_to_del. All symbols in symbols_to_del were removed from the dataset.

```
white_list = string.ascii_letters + string.digits + latin_similar + ' '

fast_chars = ''.join([c for c in tqdm(fast_embeddings) if len(c) == 1])
fast_symbols = ''.join([c for c in fast_chars if not c in white_list])

data_chars = build_vocab(list(train["comment_text"]))
data_symbols = ''.join([c for c in data_chars if not c in white_list])

symbols_to_del = ''.join([c for c in tqdm(data_symbols) if not c in fast_symbols])
symbols_to_keep = ''.join([c for c in tqdm(data_symbols) if c in fast_symbols])
```

After removing symbols not in the embedding, I ran the build_vocab and check_coverage functions again. The coverage improved significantly with over 97 percent of comment text included in the embedding, but the oov (out of vector) list containing the most common words missed by the embedding showed that contractions were still unreadable.

```
Found embeddings for 39.02% of vocab
Found embeddings for  97.84% of all text
[("don't", 184979),
 ("it's", 104138),
 ("I'm", 85114),
 ("It's", 83612),
 ("can't", 62625),
 ("doesn't", 62625),
 ("didn't", 54755),
 ("isn't", 41946),
 ("that's", 40346),
 ("That's", 38116)]
```

Figure 5: check_coverage after removing symbols

The issue with contractions was that FastText could only understand them in their expanded form. For example, "isn't" would not be recognized while "is not" would. So the next step was to expand all contractions. This was accomplished using a clean_contractions function which identified when a contraction appeared and mapped it to its most common expanded form. There was some ambiguity in doing this since contractions like "you'll" could have been expanded into "you shall" or "you will", but choosing the most common expansion seemed to work out fine.

```
def clean_contractions(text, mapping):
    specials = ["’", "‘", "", "`"]
    for s in specials:
        text = text.replace(s, "'")
    text = ' '.join([mapping[t] if t in mapping else t for t in text.split(" ")])
    return text
train['comment_text'] = train['comment_text']
.apply(lambda x:clean_contractions(x, CONTRACTION_MAP))
test['comment_text'] = test['comment_text']
.apply(lambda x:clean_contractions(x, CONTRACTION_MAP))
```

```
Found embeddings for 43.55% of vocab
Found embeddings for  99.48% of all text
[('Trudeaus', 4643),
 ("Trump's", 2722),
 ('tRump', 2144),
 ('OLeary', 1987),
 ('SB21', 1805),
 ('Hawaiis', 1645),
 ('theglobeandmail', 1350),
 ('youtu', 1274),
 ('Oregons', 1063),
 ('Comeys', 942)]
```

Figure 6: check_coverage after expanding contractions

Upon expanding the contractions, coverage improved once again but to a lesser extent. By this point, the oov list contained mostly proper nouns and misspellings so the dataset was not modified any further.

The final step in preprocessing was to tokenize the comments, converting them into a numerical form. This time around, I padded the comments to length 100, cutting off comments which were longer than 100 and adding zeroes to those less than 100. The resulting comments were easier to handle since they were all the same length. 100 was chosen as the padding length because it was long enough to preserve most of the information in the comments while keeping them at a size which the model could handle.

5 Building the Models

To build the model I used Keras, an open-source neural network library written in Python which is optimized for quick experimentation. The first layer of the model was the embedding layer, which converted the tokenized comments from the dataset into Fast Text word vectors. For example, the word "apple" with index 4151 would be converted to this 300-dimensional word vector after being passed through the embedding layer: array([-0.066 , -0.027 , -0.0403, 0.0651, -0.0168, -0.0405, 0.1743, 0.0953, -0.0455, 0.0235,...], dtype=float32).

```
from keras.models import Model
from keras.layers import Dense, Input, Dropout, CuDNNLSTM,
CuDNNGRU, Activation, concatenate, Bidirectional, SpatialDropout1D,
GlobalAveragePooling1D, GlobalMaxPooling1D
from keras.layers.embeddings import Embedding
from keras.preprocessing import text, sequence
from keras.initializers import Constant

fast_text_weights = np.reshape(fast_text_weights,
(fast_text_weights.shape[1], fast_text_weights.shape[2]))
```

```
vocab_len = len(word_to_index) + 1
emb_dim = len(fast_word_to_vec['apple'])

fast_embedding_layer = Embedding(vocab_len, emb_dim,
embeddings_initializer=Constant(fast_text_weights), trainable=False)
```

After initializing the embedding layer, I ran SpatialDropout1D with keep_prob of 0.2. This made it so the model would zero each of the 100 words in the comments with probability 0.2. Even though this stage reduced the size of the data by about 1/5th, it improved the performance since it prevented the model from becoming over-reliant on a single word to make predictions about toxicity.

SpatialDropout1D and the Embedding layer were used on all models, but later layers differed between the four models which were tested. The structure of the models is shown below.

5.1 Model 1

Layer (type)	Output Shape	Param #	Connected to
input_1 (InputLayer)	(None, 100)	0	
embedding_2 (Embedding)	(None, 100, 300)	117631200	input_1[0][0]
spatial_dropout1d_1 (SpatialDro	(None, 100, 300)	0	embedding_2[0][0]
cu_dnnlstm_1 (CuDNNLSTM)	(None, 100, 128)	220160	spatial_dropout1d_1[0][0]
cu_dnnlstm_2 (CuDNNLSTM)	(None, 100, 64)	49664	cu_dnnlstm_1[0][0]
global_average_pooling1d_1 (Glo	(None, 64)	0	cu_dnnlstm_2[0][0]
global_max_pooling1d_1 (GlobalM	(None, 64)	0	cu_dnnlstm_2[0][0]
concatenate_1 (Concatenate)	(None, 128)	0	global_average_pooling1d_1[0][0] global_max_pooling1d_1[0][0]
dense_1 (Dense)	(None, 128)	16512	concatenate_1[0][0]
dense_2 (Dense)	(None, 64)	8256	dense_1[0][0]
dense_3 (Dense)	(None, 1)	65	dense_2[0][0]

Total params: 117,925,857
Trainable params: 294,657
Non-trainable params: 117,631,200

The first model had two LSTM layers with 128 and 64 hidden units, following Spatial Dropout. An LSTM, or Long Short-Term Memory layer, is a type of neural network that specializes in learning longer patterns. LSTMs are orga-

nized in a series of cells which each take a single word and a cell state as inputs. The key feature of LSTMs is this cell state which runs from the first cell all the way to the last. Say you had a comment, "Working on my QED project," as an input. Then the LSTM would have 5 cells corresponding to each word in the comment. The cell state would be passed from the 'Working' cell to the 'on' cell and then the 'my' cell carrying useful information about the features of a sentence. These features could include the action, which is 'working', the object, which is 'QED project' or anything else which is useful to the machine learning task.

The number of hidden units in an LSTM measures the number of parameters which need to be trained. In general, LSTM layers with more hidden units are capable of learning more complex tasks but are also more prone to over-fitting to the training data.

Following the LSTM layers, a concatenation of a Global Max Pooling layer and a Global Average Pooling layer was added to the model. Global Max Pooling selected the largest output from the previous LSTM layer and zeroed the values of all the other cells. Global Average Pooling took the average of all the outputs from the previous LSTM layer.

The reasoning for combining Max and Average pooling layers is that the model would be able to preserve information from all the words in a comment with average pooling while placing a special emphasis on important parts with max pooling. The Max Pooling part is particularly important in long comments because it captures the important bits of comments which would otherwise be diluted by nonessential information.

Finally, three dense layers were added to the model. Dense layers are the most simple kind of layer in a neural network and perform linear combinations with weights and biases. For the first two dense layers, a ReLu activation function was used, which zeroes out all negative values. The last dense layer had a sigmoid activation function which outputs a decimal between 0 and 1. This output was the prediction of the model with 0 being not toxic at all and 1 being severely toxic.

5.2 Model 2

Layer (type)	Output Shape	Param #	Connected to
input_2 (InputLayer)	(None, 100)	0	
embedding_2 (Embedding)	(None, 100, 300)	117631200	input_2[0][0]
spatial_dropout1d_2 (SpatialDro	(None, 100, 300)	0	embedding_2[1][0]
cu_dnngru_1 (CuDNNGRU)	(None, 100, 128)	165120	spatial_dropout1d_2[0][0]
cu_dnngru_2 (CuDNNGRU)	(None, 100, 64)	37248	cu_dnngru_1[0][0]
global_average_pooling1d_2 (Glo	(None, 64)	0	cu_dnngru_2[0][0]
global_max_pooling1d_2 (GlobalM	(None, 64)	0	cu_dnngru_2[0][0]
concatenate_2 (Concatenate)	(None, 128)	0	global_average_pooling1d_2[0][0] global_max_pooling1d_2[0][0]
dense_4 (Dense)	(None, 128)	16512	concatenate_2[0][0]
dense_5 (Dense)	(None, 64)	8256	dense_4[0][0]
dense_6 (Dense)	(None, 1)	65	dense_5[0][0]

Total params: 117,858,401
Trainable params: 227,201
Non-trainable params: 117,631,200

Model 2 was the same as model 1 except the LSTM layers were swapped for GRU layers. GRU, or gated recurrent units, function similarly to LSTMs except they have less parameters. They have comparable performance to LSTMs on many tasks and can even exceed LSTM performance on some smaller datasets.

5.3 Model 3

Layer (type)	Output Shape	Param #	Connected to
input_3 (InputLayer)	(None, 100)	0	
embedding_2 (Embedding)	(None, 100, 300)	117631200	input_3[0][0]
spatial_dropout1d_3 (SpatialDro	(None, 100, 300)	0	embedding_2[2][0]
cu_dnnlstm_3 (CuDNNLSTM)	(None, 100, 128)	220160	spatial_dropout1d_3[0][0]
dropout_1 (Dropout)	(None, 100, 128)	0	cu_dnnlstm_3[0][0]
cu_dnnlstm_4 (CuDNNLSTM)	(None, 100, 128)	132096	dropout_1[0][0]
global_average_pooling1d_3 (Glo	(None, 128)	0	cu_dnnlstm_4[0][0]
global_max_pooling1d_3 (GlobalM	(None, 128)	0	cu_dnnlstm_4[0][0]
concatenate_3 (Concatenate)	(None, 256)	0	global_average_pooling1d_3[0][0] global_max_pooling1d_3[0][0]
dense_7 (Dense)	(None, 256)	65792	concatenate_3[0][0]
dropout_2 (Dropout)	(None, 256)	0	dense_7[0][0]
dense_8 (Dense)	(None, 128)	32896	dropout_2[0][0]
dropout_3 (Dropout)	(None, 128)	0	dense_8[0][0]
dense_9 (Dense)	(None, 1)	129	dropout_3[0][0]

Total params: 118,082,273
Trainable params: 451,073
Non-trainable params: 117,631,200

Model 3 used the same layer types as model 1, but added parameters to the LSTM and Dense layers. Additional dropout layers were placed in between the dense and LSTM layers to reduce over-fitting.

5.4 Model 4

Layer (type)	Output Shape	Param #	Connected to
input_1 (InputLayer)	(None, 100)	0	
embedding_1 (Embedding)	(None, 100, 300)	117631200	input_1[0][0]
spatial_dropout1d_1 (SpatialDro	(None, 100, 300)	0	embedding_1[0][0]
bidirectional_1 (Bidirectional)	(None, 100, 256)	440320	spatial_dropout1d_1[0][0]
bidirectional_2 (Bidirectional)	(None, 100, 128)	164864	bidirectional_1[0][0]
global_average_pooling1d_1 (Glo	(None, 128)	0	bidirectional_2[0][0]
global_max_pooling1d_1 (GlobalM	(None, 128)	0	bidirectional_2[0][0]
concatenate_1 (Concatenate)	(None, 256)	0	global_average_pooling1d_1[0][0] global_max_pooling1d_1[0][0]
dense_1 (Dense)	(None, 128)	32896	concatenate_1[0][0]
dense_2 (Dense)	(None, 64)	8256	dense_1[0][0]
dense_3 (Dense)	(None, 1)	65	dense_2[0][0]

Total params: 118,277,601
Trainable params: 646,401
Non-trainable params: 117,631,200

Model 4 used Bidirectional LSTMs instead of regular LSTMs. The advantage of Bidirectional LSTMs is that they store information about the context of a word from both directions. This helps the model because to properly understand the context of the word, it is often necessary to read the words before and after it.

6 Comparing Models

Training was performed with minibatches of size 512 for 3 epochs. After the third epoch, training was performed 1 epoch at a time, checking for loss on the development set. As soon as development loss increased, training was stopped and the lowest loss for a model was recorded. The results from training the 4 models are shown below.

Model	Best Loss	# of epochs
1	0.2342	5
2	0.2341	6
3	0.2347	4
4	0.2324	8

Model 4 performed the best with a dev loss of 0.2324 after 8 epochs. This is likely due to the ability of Bidirectional LSTM to record information about context going forward and backward.

Next best was model 2 using the GRU layers. Model 2 had basically the same loss as model 1 which is to be expected because of the similarities between GRU and LSTM units. The GRU units performed slightly better which could be due to the simpler structure of GRUs which makes them less likely to over-fit.

In last place was model 4 which was an expanded version of model 1. Though dropout was introduced, this model still over-fit to the training data as a result of the increased number of parameters. Over-fitting caused the development loss to start increasing after just 4 epochs of training. If additional steps were taken to reduce over-fitting, the performance of this model could have been improved significantly.

7 Conclusion

7.1 Summary

(1) A 2 million comment dataset was sourced from Google's Conversation AI team

(2) Comments were preprocessed to remove everything except words in a dictionary. This approach led to poor performance of the model since symbols, contractions, and slang containing useful information were eliminated.

(3) Comments were preprocessed again but more effectively. Slang, symbols, and contractions were modified so they could be understood by the models.

(4) Four models were trained on the preprocessed comments: Two-layer LSTM, Two-layer GRU, Two-layer LSTM with more hidden units, Two-layer Bidirectional LSTM

(5) The Two-layer Bidirectional LSTM (model 4) had the best performance with a final loss of 0.2324.

7.2 Further Research

In the future, an attention layer could be added to the model to improve performance on comments longer than 20 words. Attention layers prioritize certain information, allowing a model to focus solely on content which is relevant to determining toxicity.

In addition, more variations of model architectures could be tested on the toxicity classification task such as Convolution Neural Nets (CNNs), a type of model commonly used in image recognition.

Through this paper, I learned how to structure a machine learning project and troubleshoot common errors. I would like to continue improving on these skills in the future.

References

[1] Ane Berasategi. Quora Preprocessing Model. https://www.kaggle.com/anebzt/quora-preprocessing-model. [Accessed 12 November 2019].

[2] Francois Challet. Keras. keras.io. [Accessed 10 November 2019].

[3] Colah. Understanding LSTMs. https://colah.github.io/posts/2015-08-Understanding-LSTMs/. [Accessed 17 November 2019].

[4] Deeplearning.ai. Sequence Models. https://www.coursera.org/learn/nlp-sequence-models/home/welcome. [Accessed 2 November 2019].

[5] Facebook. FastText. https://fasttext.cc/. [Accessed 10 November 2019].

[6] Gruber. Liberal Regex Pattern for Web URLs. https://gist.github.com/gruber/8891611. [Accessed 11 November 2019].

[7] Christof Henkel. How To: Preprocessing for GloVe Part1: EDA. https://www.kaggle.com/christofhenkel/how-to-preprocessing-for-glove-part1-eda. [Accessed 8 November 2019].

[8] Jigsaw. Jigsaw Unintended Bias in Toxicity Classification. https://www.kaggle.com/c/jigsaw-unintended-bias-in-toxicity-classification. [Accessed 1 November 2019].

[9] KDNuggets. Text Wrangling Pre-processing: A Practitioner's Guide to NLP. https://www.kdnuggets.com/2018/08/practitioners-guide-processing-understanding-text-2.html. [Accessed 10 November 2019].

[10] Thousandvoices. Simple LSTM.
https://www.kaggle.com/thousandvoices/simple-lstm. [Accessed 9 November 2019].