CS230

"Araby" Language Model — Modeling a Messy Chat Language

Rami Botros
Google Inc., Mountain View, CA
ramibotros@google.com
Project code on Colab: https://bit.ly/2JWEoUw
Backup: https://bit.ly/2HTtCMy

Abstract

Numerous Arabs use a latinized representation of Arabic (denoted here as `Araby`), especially when writing online text and/or chat messages, by transliterating Arabic words into Latin characters. Most commonly, the rules for such unofficial writing are very loosely defined and can vary between populations and from person to person. We use Tensorflow to build a language model on an `Araby` chat/SMS corpus and use it for text generation.

1 Introduction

Millions of Arabs use a latinized, transliterated Arabic script when writing online text [1]. This form of "chat" Arabic has many names, but we will refer to it as `Araby`. The percentage of usage in different studies varies between 10%-40%, depending on the populations surveyed. Many Arabic speakers who cannot read/write in standard Arabic script, have to rely on `Araby` for written communication. For those still in the process of learning standard Arabic, `Araby` writing can also be quite helpful. Additionally, being completely in ASCII, it avoids text coding issues. An example of an `Araby` sentence is:

```
Ana raye7 el gam3a el sa3a 3 el 3asr.
```

The general principle of `Araby` is to spell words using Latin characters, such that if that spelling were to be read out loud by an English reader it would sound reasonably similar to the Arabic word. Numerals such as 2,3,5 and 7 are used to represent sounds that do not exist in English.

Some software applications already support `Araby`. For instance, Google Translate can convert `Araby` to standard Arabic [2]. In the future, it is reasonable to expect some demand for ASR systems or a software keyboards that output `Araby` text directly. **In this work, we build and test a proof-of-concept statistical neural-network based language model (LM) for `Araby` using a large chat corpus.**

1.1 `Araby`'s Modeling Challenges

From a modeling perspective, `Araby` presents an interesting set of challenges. Generally, `Araby` follows largely fuzzy rules, which have naturally emerged from the preferences of its users. `Araby` remains without formal definitions today. This results in high ambiguity in the text, which when produced by different users will usually differ in some of the following aspects:

CS230: Deep Learning, Winter 2018, Stanford University, CA. (LateX template borrowed from NIPS 2017.)

- Ambiguous Spelling. Transliteration into `Araby` is clearly ambiguous and many users tend to disagree on how to spell many words. It is hard to say what would "sound right" if read out loud in English. A well-known example for this is the variations in the Latinized spelling of Arabic names, such as: `Mohammad`, `Mohamed`, `Mohammed`, `Muhammad`, etc....
- Optional Vowelization. This is perhaps related to the point above, but deserves special consideration. In classical Arabic script vowels are optional. Hence, an Arabic writer/reader is used to relying on human inference when it comes to textual representation of vowels, a fact which carries over to `Araby`. The same user might spell out the vowels of some given word in one sentence, then omit them in the next.
- High Morphology. This is another property that `Araby` inherits from classical Arabic: many grammatically-separate pieces are combined together in one long "word" (i.e. without space separation). To illustrate, the word-token

  `mate3melhloosh`

  translates to Do not [you (subject)] do it for him. It is worth noting that these condensed phrases are especially ambiguous when it comes to online spelling, yet some specific one-letter changes would change the semantic meaning (e.g. gender). High morphology also causes an explosion in vocabulary size, i.e. the number of possible unique word tokens.
- Code switching. `Araby` writers are usually comfortable with Western languages like English or French, and would often mix different languages in the same sentence.
- Geographic Variation. `Araby` is mainly used to denote informal/colloquial speech, which itself is loosely defined and varies largely from one region to another [3].

1.2 Model Definition

We stick to the conventional definition of an LM, where the input a sequence of context tokens and the output is a prediction for the next token, given as a probability distribution over the vocabulary [4]. We train a recurrent neural network (RNN) model on this task using a large text corpus. We use a held-out dev corpus to inform decisions on hyperparameters and early stopping. Finally, we test our model's predictions by evaluating the probability of an unseen test corpus. We also use our trained model to sample a generated text and manually inspect it.

Because of the problems with high morphology, the model works on character tokens, as opposed to word tokens. We thus avoid high vocabulary size and OOV rates, and allow the model to potentially muster together long word-tokens.

2 Related work

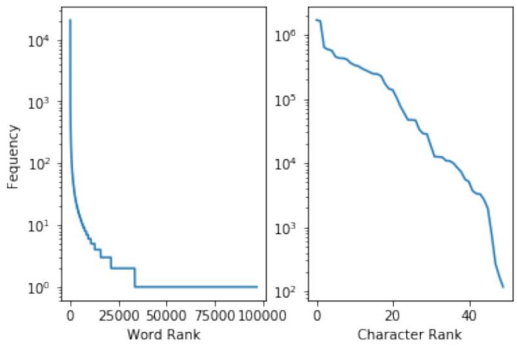
Some research on the usage and patterns of `Araby` can be found in [1]. It is stated that the history of this language began with early online systems like IRC, when classical Arabic script had low technical support.

NNLM: We build on the recent advances in neural-network based language modeling (NNLM). [5] gives an early, successful feed-forward NNLM that relies on word embeddings. [6] uses techniques like backpropagation-through-time (BPTT) and gradient clipping to create an RNN-based language models with better performance and less parameters per context length. It has been shown that LSTMs improve performance [7], esp. for longer sequences, since they tackle the vanishing gradient problem. This particular aspect is especially relevant to our work, since working with character features forces us to deal with longer sequences. Since then, numerous research on LSTM or GRU-based LMs has been done, which can sometimes be combined with attention layers to also yield interesting interpretable attention scores. An overview can be seen in [8].

Many of the current state-of-the-art models use self-attention mechanism, like the recent BERT model [9]. We think that our corpus is not large enough to justify using such large models. Moreover, we can hardly benefit from any pre-trained BERT models, since we are working on a novel language.

Character-level models have been applied successfully in many LM research works, esp. for highly morphological language like standard Arabic [10]. Other works combined characters and words as features for language models [11].

Figure 1: Word vs. Character distribution in terms of token frequency rank



For model regularization, we adopt some of the techniques mentioned in [12]. We test a variant of tied word embeddings and variational dropout. We also sample training sentences with random lengths to address issues with fixed-length BPTT, which is mentioned in the cited work: With fixed BPTT, some token will always end up in the same time step, which can be disadvantageous to training.

Previous work on `Araby` has largely focused on transliterating it back to standard Arabic, see for example [13]. In that work, an `Araby` trigram Kneser-Ney LM is utilized. We could not find any deep-learning based LM of `Araby` in the literature.

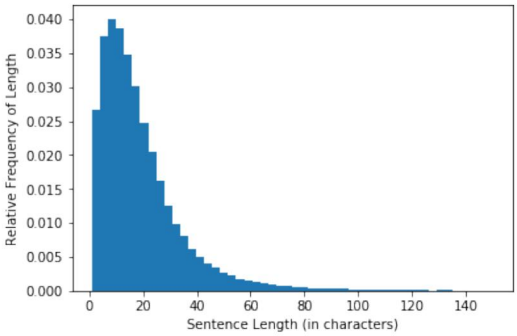
3 Dataset and Features

We use the LDC2017T07 dataset [14], which gives a collection of chat SMS messages, and perform some preprocessing on its text. Importantly, we filter any lines that include classical Arabic letters, so as to focus our model on a pure `Araby` task. We also filter Unicode emojis and normalize representations of ASCII-based emoticons. The whole text is lowercased to limit our model's vocabulary (i.e. number of characters). We insert a sentence-end token at the end of each chat message, i.e. punctuation marks and line breaks are not necessarily followed by the sentence-end token. We normalize the spacing to always be one space character, and perform some punctuation normalization. An example sentence could then be:

```
ana esmy mohamed.  :)
```

We end up with a corpus of around 184K sentences, and total of 3.7 million running characters, with 52 unique character tokens. We hold out 10K sentences as a test set. For the training set, we continuously re-sample different slices of fixed length (100) from any given sentence. If the sentence ends before the slice reaches the length 100, we pad it with special padding tokens.

Figure 2: Sentence length (in number of characters) histogram



As can be seen in Figure 1, the word distribution is very head-heavy and has a long tail. On the other hand, the character distribution follows Zipf's law more closely, which would presumably make modeling sequences of characters easier. From the sentence length histogram shown in Figure 2, it can be seen that the vast majority of sentences have less than 100 characters, which has informed the choice for the input size for our model.

4 Methods

The equations goverening LSTM [15] and GRU [16] layers are:

LSTM	GRU
$i_t = \sigma\big(U^i x_t + W^i h_{t-1}\big)$	
$f_t = \sigma\big(U^f x_t + W^f h_{t-1}\big)$	$r_t = \sigma\big(U^r x_t + W^r h_{t-1}\big)$
$o_t = \sigma\big(U^o x_t + W^o h_{t-1}\big)$	$\tilde{h}_t = \tanh\big(U^h x_t + W^h (r_t * h_{t-1})\big)$
$\tilde{C}_t = \tanh\big(U^g x_t + W^g h_{t-1}\big)$	$z_t = \sigma\big(x_t U^z + h_{t-1} W^z\big)$
$C_t = \sigma\big(f_t * C_{t-1} + i_t * \tilde{C}_t\big)$	$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$
$h_t = \tanh(C_t) * o_t$	

x_t and h_t are, respectively, the input token's embedding and the hidden output at timestep t. $U^{(\cdot)}$ and $W^{(\cdot)}$ denote trainable weight matrices that are shared across timesteps, and the output of the sigmoid expressions are usually called gates. (Bias terms have been omitted for simplicity). Stacking multiple layers of LSTM/GRU works by taking the hidden output h_t of the lower layer as the input to the upper one (in place of x_t). The GRU model is simpler and requires less parameters.

The final output of the model is a probability distribution defined over the whole vocabulary:

$$y_t = \text{softmax}(U^l h_t) \tag{1}$$

4.1 Weight Tying

[17] shares the weights between the input word embedding layer and the last output layer, where the dimensions of the weight matrices are transpose of one another. Some theoretical justification for the tying of both weight matrices is given, and the method greatly reduces the number of parameters.

In our case, we found it more effective to define the output layer as an addition layer between both a tied weight matrix and an untied one:

$$y_t = \text{softmax}(U^l h_t + \big(E^l\big)^T h_t), \tag{2}$$

where E^l is the embedding matrix used to obtain the x_t character embeddings, and U^l is an independent trainable matrix. Intuitively, similar to the idea behind residual layers, U^t will only have to learn the difference between the tied output layer and the desirable one. We found that, in our case, this was easier to train than just the free output layer matrix or just the tied one. Since our vocabulary size is small, we can afford to lose the reduction in parameters achieved by the fully tied weights.

5 Experiments/Results/Discussion

Our loss function and performance metric is the bits per character metric, which is the average cross entropy in base 2.

$$\texttt{bpc(test\_string)} = -\frac{1}{T}\sum_{t=1}^{T} log_2(y_t(x_{0:t-1})), \tag{3}$$

where y_t is inferred given all the input characters seen so far. Char-level perplexity would be $2^{\texttt{bpc}}$. T is the length of `test_string`.

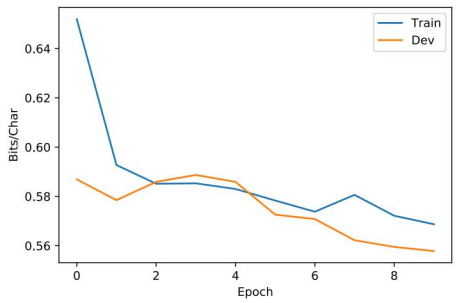


Figure 3: Train/Dev BPC Performance per Epoch

Many model hyperparameters have been chosen by trial and error, such us: embedding dimension size (100), rnn state size (100), rnn type (LSTM vs GRU, LSTM consistently better here). We tried dropout, L2 regularization weights and varying batch size, gradient clipping, all of which did not have a large effect on our baseline. Batch normalization [18] on the input and output of the RNN stack, which resulted in considerable improvements in generalization.

Training was performed using Tensorflow [19] while utilizing the Keras interface on top of it [20].

	Train	Dev	Test
Without Batchnorm Bits/Char (Char-Level PPL)	0.448 (1.36)	0.708 (1.63)	0.689 (1.61)
With Batchnorm Bits/Char (Char-Level PPL)	0.567 (1.48)	0.561 (1.48)	0.553 (1.47)

A model with perplexity P is as surprised with the test data as a model that, on average, has to choose, uniformly and independently, from P characters at each position. Figure 3 shows the train and dev performance history during training.

Examples for generated strings are shown in the following table (provided prefix colored):

3amel eh</S>	3amel eh ?</S>	3amel el 7aga :D</S>
mesh 3ala el 7aga :D</S>	mesh 3ala el 7aga el 7aga	mesh 3ala el 7aga ?</S>
mahma el 7aga :D</S>	mahma el 7aga el 7aga :D</S>	mahma el 7aga el 7aga el
ana msh 3arfa el 7aga :D</S>	ana msh 3arfa kda</S>	ana msh 3arfa el 7aga el
emtan el 7aga :D</S>	emtan el 7aga el 7aga :D</S>	emtan el 7aga ?</S>
matesh 3ala el 7aga 3ala	matesh 3ala el 7aga :D</S>	matesh 3ala el 7aga wala

Text generation was performed using non-random beam search with a character limit of 25 and beam of size 25. Outputs are ordered by higher total probability from left to right. All produced words seem meaningful except `emtan` and `matesh`. There is strong bias to generate the sequence `el 7aga`, which is indeed common in the training corpus. We note that text generation changes drastically for different models, and suffers from repetitions and bias towards specific head expressions.

6 Conclusion/Future Work

We think that overall a good strategy for high performance was to keep the number of parameters small. The model was reasonably successful in modeling `Araby`, but some complicated word expressions were still not captured. We have successfully solved the overfitting problem for average BPC, mostly using batch normalization. When looking at the generated sequences, however, the model is still clearly overfitting on a few patterns. We think the model could benefit from (1) vastly more data, esp. from different apps/platforms, and (2) the model might benefit from careful fine-grained hyperparameter optimization.

References

[1] Jan Arild Bjørnsson. "Egyptian Romanized Arabic: a study of selected features from communication among Egyptian youth on Facebook". MA thesis. 2010.

[2] *Google Translate Help*. https://support.google.com/translatortoolkit/answer/6306392?hl=en. Accessed: 2019-05-30.

[3] *Varieties of Arabic*. https://en.wikipedia.org/wiki/Varieties_of_Arabic. Accessed: 2019-05-30.

[4] Lalit R Bahl, Frederick Jelinek, and Robert L Mercer. "A maximum likelihood approach to continuous speech recognition". In: *IEEE Transactions on Pattern Analysis & Machine Intelligence* 2 (1983), pp. 179–190.

[5] Yoshua Bengio et al. "A neural probabilistic language model". In: *Journal of machine learning research* 3.Feb (2003), pp. 1137–1155.

[6] Tomáš Mikolov et al. "Recurrent neural network based language model". In: *Eleventh annual conference of the international speech communication association*. 2010.

[7] Martin Sundermeyer, Ralf Schlüter, and Hermann Ney. "LSTM neural networks for language modeling". In: *Thirteenth annual conference of the international speech communication association*. 2012.

[8] Kazuki Irie et al. "LSTM, GRU, Highway and a Bit of Attention: An Empirical Overview for Language Modeling in Speech Recognition." In: *Interspeech*. 2016, pp. 3519–3523.

[9] Jacob Devlin et al. "Bert: Pre-training of deep bidirectional transformers for language understanding". In: *arXiv preprint arXiv:1810.04805* (2018).

[10] Yoon Kim et al. "Character-aware neural language models". In: *Thirtieth AAAI Conference on Artificial Intelligence*. 2016.

[11] Rafal Jozefowicz et al. "Exploring the limits of language modeling". In: *arXiv preprint arXiv:1602.02410* (2016).

[12] Stephen Merity, Nitish Shirish Keskar, and Richard Socher. "Regularizing and optimizing LSTM language models". In: *arXiv preprint arXiv:1708.02182* (2017).

[13] Achraf Chalabi and Hany Gerges. "Romanized arabic transliteration". In: *Proceedings of the Second Workshop on Advances in Text Input Methods*. 2012, pp. 89–96.

[14] Zhiyi Song. *BOLT Egyptian Arabic SMS/Chat and Transliteration*. https://catalog.ldc.upenn.edu/LDC2017T07. DOI: LDC2017T07.

[15] Sepp Hochreiter and Jürgen Schmidhuber. "Long short-term memory". In: *Neural computation* 9.8 (1997), pp. 1735–1780.

[16] Kyunghyun Cho et al. "On the properties of neural machine translation: Encoder-decoder approaches". In: *arXiv preprint arXiv:1409.1259* (2014).

[17] Hakan Inan, Khashayar Khosravi, and Richard Socher. "Tying word vectors and word classifiers: A loss framework for language modeling". In: *arXiv preprint arXiv:1611.01462* (2016).

[18] Sergey Ioffe and Christian Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift". In: *arXiv preprint arXiv:1502.03167* (2015).

[19] Martın Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: http://tensorflow.org/.

[20] François Chollet. *keras*. https://github.com/fchollet/keras. 2015.