LSTMs for Language Detection

Jianan Gao (jgao93)

Qi Zheng (qzheng61)

1. Introduction

Automatic language identification is a natural language processing problem that tries to determine the natural language of a given content. Language detection is a relatively simple class of problems and is one of the first basic text processing techniques in tasks involving Natural Language Process (NPL), such as in Cross-language Information Retrieval. There are a number of free and commercial NLP packages implement a language identification module availed to us: TextCat, LingPipe, Xerox MLTT Language Identifier, Google's AJAX API for Translation and Detection, etc.

In this blog, we want to briefly introduce some common language detection methods, and build our own toy model based on n-grams and LSTMs. You will see how a simple LSTM model could achieve precise classification results within short runtime!

2. Language Detection Methods

Automatic language identification is based on the natural of language. There have been multiple approaches implemented to tackle this problem. The major approaches for language identification include: detection based on stop words usage, detection based on character n-grams frequency, Markov models, machine learning (ML) and hybrid methods.

2.1 Stop Words Detection

Using stop words list is a simple but effective classification method. Stop words refer to the most common words in a language. In practice, such methods have been shown to be effective because these terms are very specific to a language.

2.2 n-Gram Based Text Categorization

Some (Cavnar and Trenkle, 1994) used language profiling built on the most frequent character within the text, with n running from 1 to 5. Then, they use an ad hoc "out-of-place" ranking distance measure to classify the new text into one of the established profiles. They report greater than 90% accuracy using a 300 n-grams profile to identify a text of 300 characters, and 95-100% for longer texts. Such models are relatively easy to build and quick to run, however, they could easily fail in cases where there is a large amount of text to classify. This is a practical limitation of methods based on characteristic sequences and common words. Such methods utilized tokenisation in classification and implemented some empirical measurements, and have been criticized that the statistical properties of the texts are difficult to derive in some cases.

2.3 Character n-Gram Frequency Classification

The character n-grams method was first introduced by Dunning (1994). It is based on Markov chains and Bayesian models, and uses a log-probability sum as distance measure between the text to classify and the language model. The general procedure is that text drawn from a representative sample for each language is first partitioned into overlapping sequences of n characters (n-grams). The frequency counts from the resulting n-grams form a characteristic fingerprint of languages, which can be compared to a corresponding frequency analysis on the test set for which the language is to be determined. Generally, this type of approach work well on text collections extracted from Web Pages, but do not so well for short texts. There are also a few variations and improving techniques of this method depending on specific applications and frameworks.

One commonly used technique is the trigram frequency vectors (Damashek, 1995) which compares a vector of trigram frequencies for the text to classify with the vectors of known language, and chose the closet one. A trigram is formed by three consecutive characters of the text: $T_i = C_{1i}C_{2i}C_{3i}$, then the text can be featurized by an N-d array where N is the number of possible trigrams and v_i is the occurrence frequency of tri-gram t_i. Then we can compute the relative frequency of each trigram in the training set, and further the vector for this language (l^j). For the test text vector(w), we can compute the appearance frequencies of each trigram, and compute the similarity of this vector with the vectors of each language by taking the normalized dot product of them two:

$$w \cdot l^j = \frac{\sum_{i=1}^{N} w_i l_i^j}{|w||l^j|}$$

We then choose the language vector that gives the largest similarity factor to classify the text.

2.4 Markov Models

Hidden Markov Models (HMM) are widely used in the identification of spoken languages, but they are also commonly used in written language detection. As a matter of a fact, the identification of written languages can actually be implemented with explicit Markov Models (MM). We will used Markov methods in our own language detector training model. In its implementation, the training system explicitly knows about a series of different languages. For each language, a model is trained from a corpora to be prepared for comparison with unclassified text. In MMs, each state s_i represents a character n-gram (in our case, 5-gram). The MM predictions are the initial probability and the corresponding transition probabilities based on it:

$$a_{ij} = P(q_{t+1} = s_j | q_t = s_i)$$

a_{ij} indicates the transmitted probability from the state *i* to state *j*. Let $\pi_i = P(q_1 = s_j)$ denotes the probability of starting a sequence in state *i*, then we have:

$$a_{ij} = \frac{\#(s_i \rightarrow s_j)}{\#s_i}$$

$$\pi_i = \frac{\#s_i(t = 0)}{\#sentences}$$

Hence, to identify new text language type, sequence probabilities using each trained language model could be obtained from:

$$P(q_1, \dots, q_T) = \pi_{q_1} \prod_{t=1}^{T-1} a_{q_t q_{t+1}}$$

The Markov chain-based method is a powerful statistic tool for automatic language identification. It has solid theoretical basis, and is widely used as the evaluation tool, within combination of other methods for performance improvement, and model acceleration.

2.5 Machine Learning and Hybrid Methods

Studies (Baldwin et. al, 2010) have been performed on classification methods based on machine learning, including the nearest neighbor model with three distances, Naive Bayes and support vector machine (SVM) model. Results indicate that KNN with n-grams can achieve accuracies between 87% and 90%. Graph representation machine learning techniques also have been implemented based on n-gram features and graph similarities. Precisions achieved could exceed 95% and up to 98%. Centroid-based classification can obtain an accuracy between 97- 99.7%.

On methods such as Pattern Matching and Logistic Regression methods are commonly used for short texts, based on different features to classify languages under the same family of languages. Such approach can reach accuracy up to 97-98% for selected languages and topics. These methods are powerful in dealing sparse and short messages and texts.

Hybrid approaches use one or more of these methods. They combine features from different methods to best utilize their power in specific applications. Different combinations could be quite interesting, and some present very good classification results. You are encouraged to try out other methods yourself.

3. Model Buildup

In this section, we implemented LSTM Recurrent Neural Networks to build a language detection model in Python using the Keras deep learning library. We use eng.txt and frn.txt (the translation of the Universal Declaration of Human Rights in English and French, respectively) dataset to train and test our language model for classifying if a character/string/text is in English or French. The intuition behind our implementation of this language detection is that we want to build a model for each language we would like to identify. Then, we can compare a test text with each one of the language models and classify the text as the language it is most similar to. Take our case for an example, if a given sentence is much closer to the model of English than the model of French (or any other languages), the system will say that this sentence was very likely written in English. LSTM RNN has shown great performance on natural language process, and to a broader sense, recognizing patterns in sequences of data. They are arguably the most powerful type of neural network for its unique feature of “memorization”. They are arguably the most powerful type of neural network, applicable to tasks which can be decomposed into a series of patches and treated as a sequence. Just like we human being understand each word based on your understanding of previous words, the network mimic this powerful feature: it does not throw everything away and start thinking from scratch again. Its 'memory' have persistence. Detailed introduction of the mechanism and guides to RNN and LSTM could be find at these two blogs(1, 2), readers are encouraged to read through these two articles before building diving into building your first LSTM model. We are not going into details of LSTM since it's out of the scope of this blog.

Our annotated codes are provided in the attached Jupyter Notebook, you can refer to the codes for detailed implementation. Generally, our procedure for building up the model is as follows:

1. For eng.txt and frn.txt , we read in the two dataset, clean the data by removing the irrelevant characters and symbols, lowercase all letters for simplicity. We then combine the two datasets, pick out all distinct characters, get the total distinct character counts, then create a mapping of each character to a unique integer, and a reverse mapping from integer to character.
2. For two text files, split each file into 80/20 learning/holdout subsets without shuffling.
3. In this section, we want to generate training and test datasets by apply padding to generate fixed size (5-char in this case) substring pairs. For example, for a test string of 'trump', we want to generate five input pairs (SSSSS, t) (SSSSt, r) (SSStr, u) (SStru, m) (Strum, p), where 'S' is the padding. Training and test sets conform the same pattern. Hence, we divide the whole text into individual character, for a text of length k, we would obtain k characters. Then we assemble them to a k-length dataset of (SSSSS, $text_{[i]}$) pairs, where i~[0, k-1].
4. Now, we prepare 1-to-1 (next) prediction pairs by assembling two consecutive characters to (k-1) pairs of (SSSS$text_{[i]}$, $text_{[i+1]}$) with padding S, where i~[0, k-2].
5. Following this procedure, we prepare 2-to-1(next), .3-to-1, 4-to-1 pairs. For example, the final 4-to-1 would be like (S$text_{[i]}text_{[i+1]}text_{[i+2]}text_{[i+3]}$, $text_{[i+4]}$)
6. Build simple LSTM RNN model by including one single LSTM.
7. Feed the training data to train the model for prediction

8. For each test string, we compute the log likelihood for each language model. We generate the predicted probability for each string as to predict transmitted probabilities. Take the same 'trump' example, we compute P(t|SSSSS), P(r|SSSSt), P(u|SSStr), …, P(p|Strum). In this process, we take log on each probability P, and add them one after another, which corresponds to multiplying them together and then take the log, i.e. $\log(P(t) * P(r)* P(u) * P(m) * P(p)) = \log P(t) + \log P(r)+ \log P(u) + \log P(m) + \log P(p)$.
9. Use standard sklearn to compte ROC where y_hat is the ratio of the log likelihood of two languages.
10. Generate ROC plot.

4. Questions and Discussions

4.1 Evaluation of Our Model

The AUC score of our simple base model achieves 0.895. It runs fast (a few seconds per epoch with a total of 5 epochs). Overall, with a very simple structure, within very short time, our model achieves good predict precision. Hence, we consider it a good model.

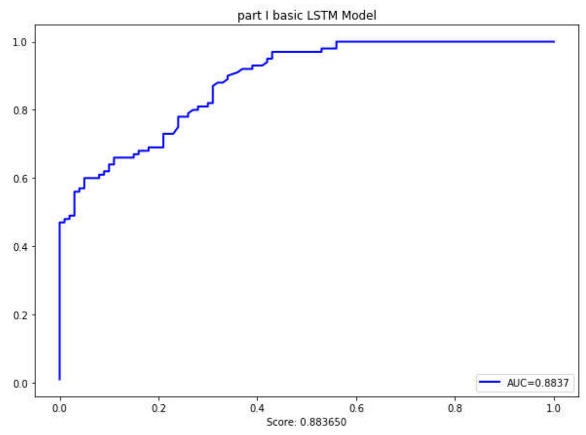

Figure 1. ROC plot for base case LSTM Model

4.2 Alternative Methods

Please refer to Section 2 for detailed discussions of alternative methods in terms of language detection.

4.3 Improving the Models

There are a variety of ways to improve your LSTM model, readers of this blog are encouraged to fine-tune your model both for better performance and further exploring features of the LSTM RNN.

Examples of what you can try:

1. Large LSTM models could be computational intensive. For relatively large models, you may want to use GPU for acceleration, which will significantly speed up your model runtime. However, it won't affect the efficiency, neither does it help with accuracy of model results.
2. Another thing you may want to check out is Google's new Tensor Processing Units (TPU). They are application-specific integrated circuits (ASICs) developed specifically for machine learning. Compared to GPUs, they are designed explicitly for a higher volume of reduced precision computation with higher IOPS per watt.
3. Increasing batch size: in training our model in this case on my PC, the GPU usage merely approaches 20%, hence the batch size could be improved. Increasing the size of a mini-batch could decrease the time to convergence. For large models, pick appropriate sizes for your mini-batches to achieve optimal performance.
4. Carefully tune your model to pick the best combination of hyperparameters to achieve optimal performance. Watch out for overfitting, which happens when a neural network essentially "memorizes" the training data. Using large epochs or batch size or layer numbers dose not correspond to better performance.
5. Regularization helps: regularization methods include L1, L2, and dropout. Using an L1 or L2 penalty on the recurrent weights can help with exploding gradients. Dropout helps preventing neural networks from overfitting.
6. Try different cells: LSTM, GRN cells, and the newest NASCells. NASCell designs RNN by the Neural architecture search neural net and has better performance than LSTMs, GRUs, etc.
7. Introduce residual network to your network. Recent papers interpreted residual networks as exponential ensembling of relatively shallow networks. Including residual networks avoids the vanishing gradient problem by

introducing short paths which can carry gradient throughout the extent of very deep networks.

5. Part II

For Part II, we tested varying cell types and LSTM layer dimension on the base model to check the resulting performance.

5.1 Cell Type

GRU cell

GRU is related to LSTM as both are utilizing different way if gating information to prevent vanishing gradient problem which vanilla RNNs suffer from. To solve this problem, we would like to have gradient values that persist as they go flow backward.

LSTMs have one cell that stores the previous values and hold onto it unless a "forget gate" tells the cell to forget those values. They also have a "input gate" which adds new stuff to the cell and an "output gate" which decides when to pass along the vectors from the cell to the next hidden state. Recall that with all RNNs, the values coming in from X_train and previous hidden state are used to determine what happens in the current hidden state. Results of the current hidden state are used to determine next hidden state. In short, we want to remember stuff from previous iterations for as long as needed, LSTMs add a cell layer to allow this to happen.

At a high level, GRUs work the same way except they don't need the cell layer to pass values along. The calculations within each iteration insure that the current hidden state values being passed along either retain a high amount of old information or are jump-started with a high amount of new information.

Here are some pin-points about GRU vs LSTM:

- The GRU unit controls the flow of information like the LSTM unit, but without having to use a ***memory unit***. It just exposes the full hidden content without any control.

- GRU is relatively new, and in many cases (as is in this case, Figure 2), the performance is on par with LSTM, but computationally ***more efficient*** (*less complex structure as pointed out*).

Here is a good paper that elaborates on gated RNN very well. Check it out if interested.

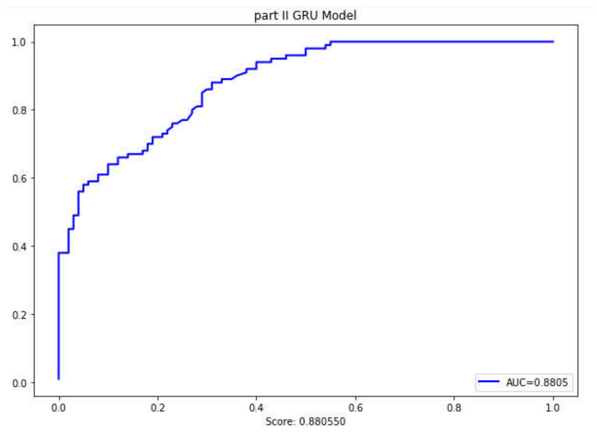

Figure 2. ROC plot for GRU Model

Again, as illustrated, the GRU model performs no worse than LSTM models in most cases (as is in our case), and is generally more computational efficient. Hence, using GRUs for large models could be an alternative to LSTMs.

Basic RNN Cell

To see the improvements of LSTMs (and GRUs) from basic RNN method, we also built a basic RNN model with the same structure with a SimpleRNN layer instead of the LSTM layer. Comparing Figures 1, 2, and 3, we can find that AUCs has been improved by 4.5% and 4% by LSTMs and GRUs, respectively.

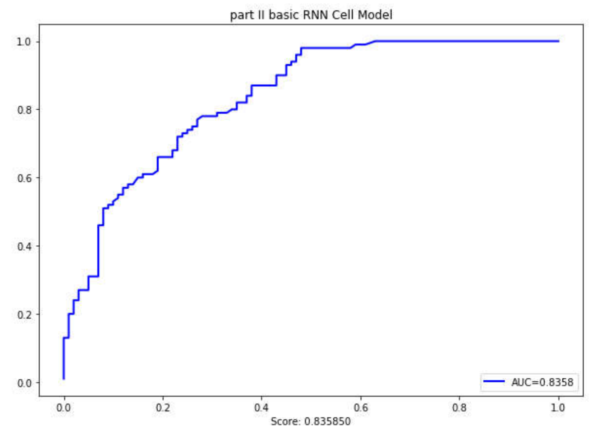

Figure 3. ROC plot for simple RNN Model

5.2 LSTM Layer Dimension (Hidden Units)

The original LSTM with 128 hidden units (i.e., will return a vector of dimension 128). We varied the dimension by a factor of 2 at a time, resulting in three new cases with dimension of 32, 64, 256. Comparing Figures 4, 5, 6 with Figure 1, we find LSTM with 128 hidden units yields the best AUC score. It again confirms our previous claim that increasing the number of certain parameters and making the model more complex does not necessary improve the performance (in many cases actually the opposite).

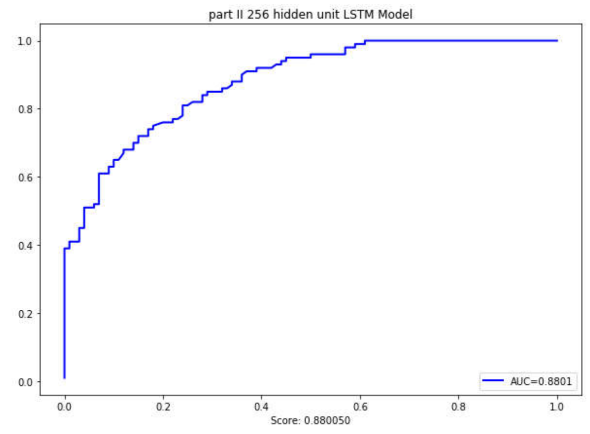

Figure 4. ROC plot for 256 hidden units Model

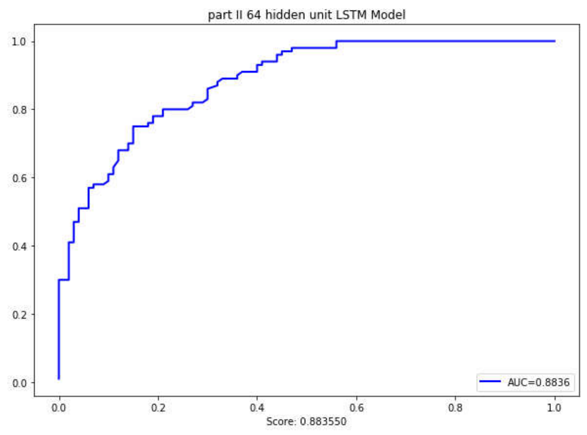

Figure 5. ROC plot for 64 hidden units Model

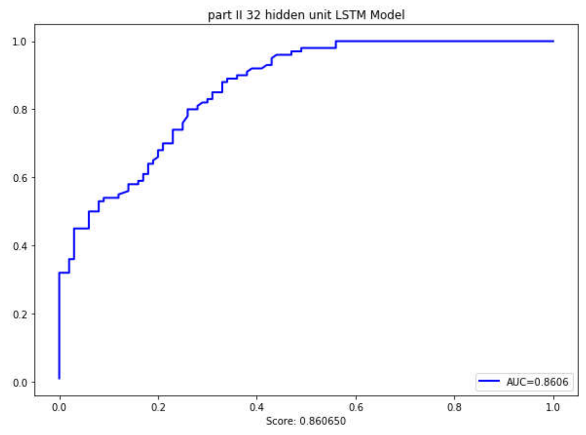

Figure 6. ROC plot for 32 hidden units Model

5.3 Early Stopping

As models might overfit the training dataset whereas yeilding poor generalization ability, we ask if early stopping can improve the performances of the models. Early stopping moniters the validation loss during training, and stops the model if the validation loss does not decrease or even increases. Here we tested two cases: a) train models using fixed 20 epochs b) train models with early stopping. In the first scenario, we did observe overfitting issue as validation loss increased after 8 epochs. However, the overall performances between the two scenarios are quite similar. One possible reason is that though the two eng and frn models in the first scenario both overfitted, the overfitting issue was naturally 'cancelled' during division when calculating y_hat.

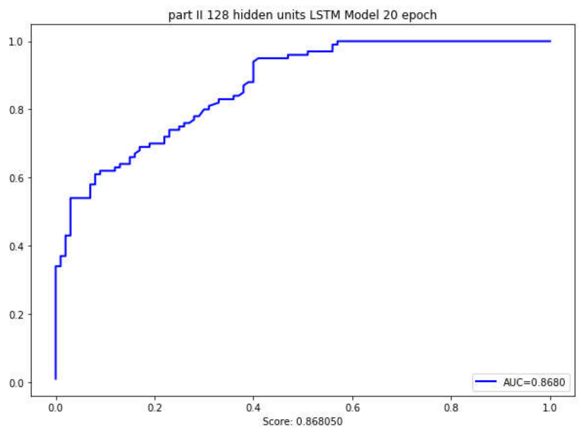

Figure 7. ROC plot for fixed 20 epoch Model

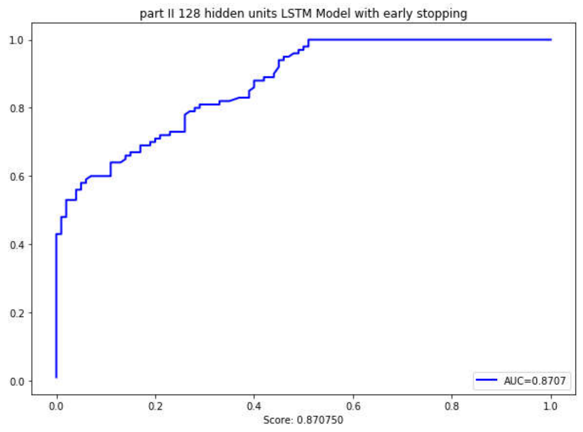

Figure 8. ROC plot for early stopping Model

Reference

Baldwin, Timothy, and Marco Lui. "Language identification: The long and the short of the matter." *Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics*. Association for Computational Linguistics, 2010.

Cavnar, William B., and John M. Trenkle. "N-gram-based text categorization." *Ann Arbor MI* 48113.2 (1994): 161-175.

Damashek, Marc. "Gauging similarity with n-grams: Language-independent categorization of text." *Science* 267.5199 (1995): 843.

Dunning, Ted. *Statistical identification of language*. Computing Research Laboratory, New Mexico State University, 1994.

Understanding LSTM Networks: http://colah.github.io/posts/2015-08-Understanding-LSTMs/

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