Parlez-vous Français, or English ?

Language detection fun with LSTMs

Today we'll be building a machine learning model to detect languages, specifically French and English, using long short term memory networks (LSTMs). The goal is to create a black-box of sorts that will take any string of text as an input and will produce the language of the string as the output. Of course, in this scenario we'll know exactly what's happening inside the so-called "black box".

LSTMs are a cool (and powerful) class of recurrent neural networks (RNNs) that allow for the learning of long-term dependencies in sequences of inputs using special structures called "memory cells" and "gates". We won't get too bogged down in the details of LSTMs here so if you're interested see http://colah.github.io/posts/2015-08-Understanding-LSTMs/ for a more in-depth look at their inner workings. They're naturally applicable to text processing tasks if the problem/data is set up correctly: assume the "sequences" refer to strings of text where each "input" is a character. Train an LSTM on sequence-of-characters data and the result is essentially a machine learning model that acts like it *understands language* (spooky huh?). So, if we train one LSTM to "understand" French and another to "understand" English then we can (in theory) create a hybrid of the two to detect languages! At least, that's the motivation for using LSTMs for language detection in the first place.

Our training and testing data is relatively simple yet unstructured. We've got the declaration for human rights in English and again in French.

Framework

Here's a rough outline of the sequential tasks our language detection model will go through:

1. Word embedding: convert raw text to a model-friendly representation
2. LSTMs: train the English and French LSTMs
3. Synthesis: Combine the individual LSTM outputs to produce a single language prediction

For tasks 1 and 2 we'll be following a simple Keras tutorial[1].

Word embedding

Every natural language processing task has the problem of what to do with the unstructured mess that is "text". Since all (or most?) machine learning models require some sort of design matrix (or tensor) with numeric contents, we've got to convert the text into a model-friendly version.

First, let's consider what our rows or observations are going to be. At the end of the day we want our model to take a string as input and output its language of origin so we'll conceptually set each row to represent a sub-string of text in the original document. Notice there will be a lot of overlap in the strings if we just continually "slide" our sub-string window down the document one character at a time. Some implementations skip a certain number of characters each "slide" but for our purposes we just keep it simple and take every possible sub-string (of a specified length).

[1] https://github.com/fchollet/keras/blob/master/examples/lstm_text_generation.py

“An apple a day keeps the doctor away.”	“An ap”
	“n app”
	“ appl”
	“apple”
	“pple “
	“ple a”
	…

Figure 1: A look at how we’re creating the sub-strings. In this example, we create sub-strings of length 5 and don’t have any “skips” in the “slicing” window.

Here’s what that looks like in practice (reading in the documents, preprocessing, and experimental setup steps are omitted for brevity):

```
# split strings into gram_size length sub-strings, skipping by step_size chars each slice
gram_size = 5
step_size = 1

# english
eng_sub_strs = []
eng_next_chars = []
for i in range(0,len(eng_text)-gram_size,step_size):
    sub_str = eng_text[i:i+gram_size]
    eng_sub_strs.append(sub_str)
    eng_next_chars.append(eng_text[i+gram_size])

# french
frn_sub_strs = []
frn_next_chars = []
for i in range(0,len(frn_text)-gram_size,step_size):
    sub_str = frn_text[i:i+gram_size]
    frn_sub_strs.append(sub_str)
    frn_next_chars.append(frn_text[i+gram_size])
```

Notice we’re taking the characters that follow each sub-string as well. We’ll cover this in more detail later, but the gist of it is that we’re training our LSTMs to predict the character that comes *after* each substring and using the confidence of the character-level predictions as a proxy for how “convinced” each LSTM is that the input string comes from the language it’s trained on. In other words, these following characters are the response that the LSTMs are trained on. For now, just go with it, it’ll all make sense in a little bit I promise.

So now we’ve got our “rows” but what about our features (i.e. the “columns” of the design matrix)? We’re going to do something a little fancy here and convert each “row” into a matrix itself! This mini-matrix will have a specified number of rows (we call it `gram_size` in the code snippet above) and a column for each unique character. The values are going to be either a 1 or a 0 indicating characters. We call this the “vectorization” process. A little confused? See the example below for some elucidation.

	a	**b**	**c**	**d**	**e**	**…**	**l**	**…**	**p**	**…**
0	1	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	1	0
2	0	0	0	0	0	0	0	0	1	0
3	0	0	0	0	0	0	1	0	0	0
4	0	0	0	0	1	0	0	0	0	0

Figure 2: An example of how the string “apple” would be represented under our word embedding scheme.

The result is a design *tensor* with dimensions "# of characters in document" *minus* `gram_size` by `gram_size` by "# of unique characters". Luckily LSTMs are designed to take tensor input, so no problems there. In Python, this process looks something like,

```
# create dictionaries mapping characters to their feature (column) index
unique_chars = sorted(list(set(eng_text+frn_text)))
char2idx = {}
idx2char = {}
for i,c in enumerate(unique_chars):
    char2idx[c] = i
    idx2char[i] = c
```

```
# vectorize substrings

# english
X_eng = np.zeros((len(eng_sub_strs), gram_size, len(unique_chars)))
y_eng = np.zeros((len(eng_sub_strs), len(unique_chars)))

for i in range(len(eng_sub_strs)):
    sub_str = eng_sub_strs[i]
    vec = np.zeros((gram_size, len(unique_chars)))
    for j in range(len(sub_str)):
        char = sub_str[j]
        idx = char2idx[char]
        vec[j,idx] = 1

    X_eng[i,:,:] = vec

    next_char = eng_next_chars[i]
    idx = char2idx[next_char]
    y_eng[i, idx] = 1

# french
X_frn = np.zeros((len(frn_sub_strs), gram_size, len(unique_chars)))
y_frn = np.zeros((len(frn_sub_strs), len(unique_chars)))

for i in range(len(frn_sub_strs)):
    sub_str = frn_sub_strs[i]
    vec = np.zeros((gram_size, len(unique_chars)))
    for j in range(len(sub_str)):
        char = sub_str[j]
        idx = char2idx[char]
        vec[j,idx] = 1

    X_frn[i,:,:] = vec

    next_char = frn_next_chars[i]
    idx = char2idx[next_char]
    y_frn[i, idx] = 1
```

LSTMs

Now for the more interesting part. Using Keras, we build a simple LSTM for each language. Each is designed in such a way that it learns to predict the character following the input "mini-matrix" it's given. Specifically, its output is a vector where each component represents a unique character. The values in each component correspond to the probability the next character is that character.

a	**b**	**c**	**d**	**e**	**…**
0.02	0.01	0.01	0.01	0.95	0.00

Figure 3: An example of the vector that might be outputted given the string "puzzl" as input. Notice the component corresponding to "e" has the highest value, so the character predicted to follow "puzzl" would be "e".

The LSTM layer (where it learns the long-term dependencies) has 128 cells and the number of cells the prediction layer has is equivalent to the number of unique characters present in the document(s). Further, it employs the softmax activation function. We go with softmax because its output is essentially

a probability of belonging to a particular class (think of a multinomial distribution). Which corresponds to the previous example in figure 3 where classes are characters. The Python code is relatively simple using Keras,

```
from keras.models import Sequential, Model
from keras.layers import Dense, Activation, Input
from keras.layers import LSTM
from keras.optimizers import RMSprop
```

```
Using TensorFlow backend.
```

```
eng_model = Sequential()

lstm_layer = LSTM(128, input_shape=(gram_size,len(unique_chars)))
eng_model.add(lstm_layer)

pred_layer = Dense(len(unique_chars))
eng_model.add(pred_layer)

pred_layer = Activation('softmax')
eng_model.add(pred_layer)

optimizer = RMSprop(lr=0.01)
eng_model.compile(loss='categorical_crossentropy', optimizer=optimizer)

eng_model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm_2 (LSTM)                (None, 128)               88064
_________________________________________________________________
dense_2 (Dense)              (None, 43)                5547
_________________________________________________________________
activation_2 (Activation)    (None, 43)                0
=================================================================
Total params: 93,611.0
Trainable params: 93,611.0
Non-trainable params: 0.0
_________________________________________________________________
```

```
frn_model = Sequential()

lstm_layer = LSTM(128, input_shape=(gram_size,len(unique_chars)))
frn_model.add(lstm_layer)

pred_layer = Dense(len(unique_chars))
frn_model.add(pred_layer)

pred_layer = Activation('softmax')
frn_model.add(pred_layer)

optimizer = RMSprop(lr=0.01)
frn_model.compile(loss='categorical_crossentropy', optimizer=optimizer)
```

We've got 93,611 parameters to learn! Luckily I've got my GPU set up to do the heavy computation, so it only takes a few seconds to train each LSTM, but your mileage may vary (a lot). We train for 5 epochs which seems to be enough to produce pleasing results while not overfitting the models.

```
# train english model
eng_model.fit(X_eng_train, y_eng_train,
          epochs=5,
          batch_size=128,
          shuffle=True,
          verbose=2)
```

```
Epoch 1/5
1s - loss: 2.7063
Epoch 2/5
0s - loss: 2.0482
Epoch 3/5
0s - loss: 1.7543
Epoch 4/5
0s - loss: 1.5461
Epoch 5/5
0s - loss: 1.3701

<keras.callbacks.History at 0x13bd9c7e0b8>
```

```
# train french model
frn_model.fit(X_frn_train, y_frn_train,
              epochs=5,
              batch_size=128,
              shuffle=True,
              verbose=2)
```

```
Epoch 1/5
1s - loss: 2.5596
Epoch 2/5
0s - loss: 1.9958
Epoch 3/5
0s - loss: 1.7498
Epoch 4/5
0s - loss: 1.5707
Epoch 5/5
0s - loss: 1.4248

<keras.callbacks.History at 0x13bdb5d69b0>
```

Synthesis

Now for the super interesting part. How do we use these LSTMs to detect language? Essentially what we want are scores that describe how likely it is that the input string comes from the English and French languages. Then we can take whichever score is higher and use that as the prediction! So, let's create such a score metric.

Since our LSTMs output probabilities, it's easiest to work with those. With this in mind, probabilistically speaking the score metric looks something like

$$P(word\ |language)$$

Which looks pretty simple but doesn't match what our LSTMs output. Clearly, the language portion of the probability refers to "English" or "French" but we don't have probabilities of *words* we have probabilities of *characters*. Further, we have probabilities of the *next* character. So, let's get a little creative here and define P(word|language) as follows:

$$P(word|language) = \sum_{i=0}^{word\ size} \log P(char_i|char_{i-1},\ char_{i-2}, \dots ,\ char_0,\ START)$$

It looks a little intimidating so let's break it down. Here, START is a "mini-matrix" that just has a bunch of 0's (no 1's) which indicates that no characters have preceded the predicted output. Recall that the LSTMs take a matrix with `gram_size` number of rows and output a probability vector for each potential next character. By feeding in an input of all 0's we're effectively saying "predict the probability of the first character in a sequence". Moving on, $char_i$ is the i^{th} character of the input string. So, we want to

capture the probability of the i[th] character *given all the characters preceding it*. Hopefully this is pretty intuitive. Let's look at an example of how the word "puzzle" as an input would produce a score.

$$
\begin{aligned}
P(puzzle|english) \\
&= \log P("p"|START) + \log P("u"|"p", START) + \log P("z"|\ "u", "p", START) \\
&+ \log P("z"|\ "z", "u", "p", START) + \log P("l"|\ "z", "z", "u", "p", START) \\
&+ \log P("e"|\ "l", "z", "z", "u", "p", START)
\end{aligned}
$$

Makes sense once we expand it out, right? We follow this procedure for both the English and the French LSTMs and then take the ratio of the English score over the French score as the final language detection metric. If this metric is large then input string is more likely to be English than French, and if this metric is less than 1 then the opposite is true (the smaller the number the more convincing it's French). All we need a simple function to act as our final language detection model:

```
# build language prediction model on top of english and french LSTMs

def model_predict(test_gram):

    # insert zero vector at 0th position to indicate START
    start = np.zeros((1,len(unique_chars)))
    test_gram = np.insert(test_gram, 0, start, axis=0)

    # initialize log probability metrics
    gram_log_prob_eng = 0.
    gram_log_prob_frn = 0.

    # calculate the probability the next character will be the true character,
    # given the previous characters, for all characters in test_gram
    # essentially the predicted values correspond to
    # P('t'|START), P('r'|START,'t'), P('u'|START,'t','r'), etc.
    for i in range(1,test_gram.shape[0]):

        # pad gram with 0 vectors to indicate START
        # an analogous example:
        # the string 'trump' is split as: '00000' -> 't', '0000t' -> 'r', '000tr' -> 'u', etc.
        pad = np.repeat(start, gram_size-i, axis=0)

        # reformat test_gram for prediction
        temp_gram = test_gram[:i,:]
        temp_gram = np.append(pad, temp_gram, axis=0)
        temp_gram = temp_gram.reshape((1, gram_size, len(unique_chars)))

        # predict probability the next char will be the true char for english and french
        y_hat_eng = eng_model.predict(temp_gram)[0]
        y_hat_frn = frn_model.predict(temp_gram)[0]

        next_char = test_gram[i,:]
        idx = list(next_char).index(1)

        log_prob_eng = np.log(y_hat_eng[idx])
        log_prob_frn = np.log(y_hat_frn[idx])

        # add log(P(temp_gram|chars before)) to total log prob
        gram_log_prob_eng += log_prob_eng
        gram_log_prob_frn += log_prob_frn

        #next_char = idx2char[idx]
        #print(i,next_char,log_prob_eng, gram_log_prob_eng, log_prob_frn, gram_log_prob_frn)

    y_hat = gram_log_prob_eng/gram_log_prob_frn

    return gram_log_prob_eng,gram_log_prob_frn,y_hat
```

Encoding English as "1" and French as "0" we can finally use this score to produce an ROC curve for the language detection model.

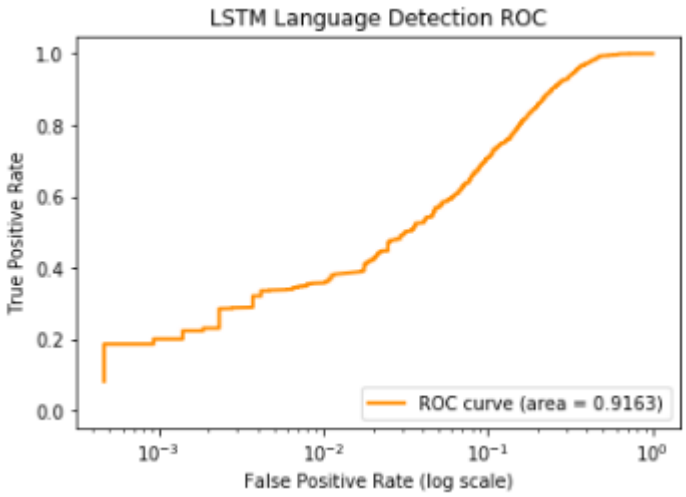

Figure 4: Language detection model's ROC curve (log scale false positive rate).

It's got a respectable AUC of 0.9163 proving that our hybrid language detection model isn't half bad at discerning English from French! It's safe to say that this hybrid language detection model is "good".

Future Work

This model isn't bad but of course it can be improved! Let's look at a few that come to mind:

1. Increase the depth of the LSTM models.

   We could add additional LSTM layers to learn the long-term dependencies of the long-term dependencies! This might produce more accurate results (higher AUC) but would probably take forever to train.

2. Redesign the LSTM for outright language detection.

   Instead of creating a bastard of LSTM models, trained to do one thing and then used to do another, we could design the LSTM to produce either a binary output, either 0 or 1 for French or English, respectively. This would probably make the problem a lot more difficult though, in terms of training. The LSTM would probably need a lot more training data to get a good handle on exactly how to map sequences of characters to 0 or 1 and provide good predictive results. I imagine that the model would probably overfit quickly.

3. Change the training set.

   While the document we use to train works well, we should keep in mind that the test set is created from the original document itself. Since it's a declaration of human rights, the style is inherently different than, say, this blog post. I imagine the model might have a more difficult time discerning between an English blog and a French blog, especially one about technical material like machine learning models. Therefore, using different training set with a wider set of text "styles" would likely improve its performance (in terms of accuracy) in a general sense.

4. Encode the text differently.

   I'm always hesitant to take a purely statistical approach to a problem that is clearly not founded in statistics. While defining language to be sequences of characters is probably a good approximation, language clearly has more lurking beneath the surface. For instance, semantic information, contextual information, cultural information, etc. I assume it's difficult to capture al that information with such a simple word embedding procedure. So why not use a more robust method such as word2vec, or perhaps some sort of semantic identification model on top of the word embedding model that the LSTMs then learn from. Imagine being able to capture sequences of *ideas* rather sequences of *characters*. At a glance this would probably produce favorable results. Just look at how English and Spanish use adjectives: "blue car" versus "carro azul". One has the adjective first, the other the noun. I'd imagine LSTMs trained on this sort of data would perform at least as well as character-level data.

5. Use a bi-directional RNN with GRU cell instead of an LSTM.

   While both models attempt to learn the long-term dependencies in sequences of inputs, the former does so without using additional "memory cells". This should theoretically speed up training time (not that it's too much of an issue) and may or may not increase accuracy (i.e. AUC).

Alternative Language Detection Models

Language detection isn't new or anything. There are countless other models that attempt the same procedure. For thoroughness let's go through a few along with their pros and cons.

1. N-Gram Based Text Categorization

   Probably one of the most famous text categorization papers, this approach builds profiles of categories (i.e. languages) using the frequencies of n-grams. N-grams are just sequences of characters of length n, equivalent to the sub-strings we created above. The profiles are essentially the n-grams sorted by frequency from highest to lowest. Note profiles can be built for any collection of text (i.e. languages or individual documents or whatever). The classification procedure is relatively straight forward, if we use the context of English versus French language classification, a string of text is assigned to the language that corresponds to the smaller out-of-place measure. Here the out-of-place measure calculates the difference in positions for each n-gram of the string of text in question, and the language profile.

Pros	**Cons**
• Easy to implement • Easy to understand • No unjust assumptions made • Applicable to any text category, not just languages	• Doesn't work well with small strings of text (i.e. might work well with document language classification but not word language classification) • A little too naïve – doesn't take any domain-specific knowledge into account, it's purely simple statistics

	○ Results in an over-reliance on the training data • Was created in ’94

2. Hidden Markov Models

This approach utilizes widely applicable markov chains to model n-gram chains from a probabilistic standpoint. Without getting bogged down in the details, transition probabilities are computed for each n-gram in a set of training corpora, one corpus for each language. An example transition probability would be the probability of going from “sou” to “oup”. These transition probabilities are computed using maximum likelihood estimation. The classification of a string of text would correspond to the language that gives the highest overall transition probability.

Pros	**Cons**
• Relatively easy to understand and implement • Robust theoretical framework surrounding Markov models • Works well with any size of n-gram	• Independence assumption between transitions isn’t justified, i.e. “the future is **NOT** independent of the past, if you know the present” in the context of written language • Using MLEs to estimate transition probabilities guarantees asymptotic convergence to the “truth” but current computing limitations and everything makes me doubt that this is good for language detection. I mean at what point does one have a representative sample of all written language of an entire language? There’s too many stylistic nuances that would require more and more data, in my unexperienced opinion.

3. N-gram Proportion Vector Cosine Similarity

This approach is somewhat similar to the first alternative, but different enough to justify its own place in our list. It creates a “prototypical” vector using the proportions of n-gram occurrences for each language in a set of training corpora (one for each language). Then, by creating a similar proportion vector of n-grams for a string of text, the cosine of the angle between the vector and all the prototypical vectors are calculated (this is considered to be a measure of how similar the string of text is to each of the languages). The language that the string of text is most similar corresponds to the language that results in the largest cosine similarity measure.

Pros	Cons
• Easy to implement • Cosine similarity is widely used	• Using proportions loses a lot of information – it's a little too naïve of an approach • Need a lottt of data to create a "good" prototypical vector

References

1. http://odur.let.rug.nl/~vannoord/TextCat/textcat.pdf
2. http://www.lsi.upc.edu/~nlp/papers/padro04a.pdf