Using LSTM Text Generation to Procedurally Generate Text

Kyle Morris
Winter 2020
https://github.com/jkylemorris/DSC-Portfolio

Text generation is a machine learning problem whereby we can calculate the probabilities of the next character appearing in a text and, by repeatedly predicting the next character, generate text. The text generated depends on the input text used – if we use a book on piracy in the Caribbean Sea in the 18th Century, we are going to get text that sounds quite different than if we used a collection of patents filed in the early part of the 21st century due to how our model was changed.

For our project, we will be utilizing a model called LSTM or Long Short-Term Memory to generate text based off our chosen corpus. LSTM is useful because it controls the flow of information. We start with a sample of about 60 characters or so and then predict the next character. Then, instead of using the 61 characters we have, we drop the first character and consider the remaining 60 characters as we predict the next character. Each character has a likelihood calculated and we choose the most likely character each time.

It is really fascinating to see this in action. However, that is only a portion of what we will be attempting to do in our project here. In this case, we are going to vary the inputs into our LSTM model in order to build up a database of actual samples from our text and generated samples, then run a categorization model to see if we can fool the model. Ideally, we will find the sweet spot for what we refer to as temperature where we are generating text that is difficult for our algorithm to classify correctly.

Why are we hoping to fool our algorithm? Well, one practical application of this would be in worldbuilding when it comes to crafting artificial worlds and storytelling. If you have a fantasy realm that you want to make it seem realistic, you are going to need some backstory. Maybe some novels for the players to find. However, to do it right, you would need a single person (or a team) spending months only on filling out these texts. One of the applications we have seen as development costs skyrocket is the idea of procedurally generating content and our LSTM generator would be a prime candidate for assisting with this. The team could write a few hundred pages of backstory, in-universe mythology, how-to guides, etc. then use the model to extrapolate. Most players are not going to give the text more than a cursory glance and the creatives have more freedom than to churn out endless pages of books.

With the secondary classification stage of this project, we can be a little more systematic in how we approach our generation. At a certain point we hope to find where our generated text begins fooling our algorithm and the casual observer at least part of the time. I do not believe we will get to the point where we can fool it consistently, but even around just a 95% accuracy would be phenomenal.

In order to begin generating our text, we need a corpus, or a collection of written text. This corpus is what we will use to actually train to train the first part of our model, the LSTM text generator. In this case, we chose a novel in the public domain that much of the public will be familiar with – the classic Sherlock Holmes tale the Hound of the Baskervilles. The full text of the novel, as it is the public domain, is available via the Project Gutenberg web site. We have elected to utilize the book in the txt file format as it is simplest for us to import into our program. We import the text as all lowercase, so as to not differentiate between upper case and lower-case letters.

We use a package called Keras to create our LSTM model. We then train our model based on our sample text – in our case, we have 60 epochs that generate text at 4 different temperatures. These results are stored as it can be interesting to see the model evolve across the different epochs. Here is an example of the text in our penultimate epoch at temperature .5: “se parts, but there was in him a certain wanton be about the convict and had lived man a could not less that you were clenty—the baronet." "really, watson, and a hard of the house docails i stay indated you interesting me the station, and the more well, watson, sir. He we assuatuse the convict that he had been the fact of this street and asked us might you will state which i had to rought the granself of his" and here is it at temperature 1.2: "se parts, but there was in him a certain wanton dug a. toldy door ounsing!" wese it is all letter at a gutanting every blowk, and the old distanterly enony astone no defebid corks it espoder thrill." all wo stanes, thowe gradity to me risefels." your fece. did victedly we are fof levell, at mast by right, good to escands for the fres care of i will mighted, and leave ne. it had suoyw inin" had".

As you can see, there is a tradeoff. The temperature of .5 results in text that at least appears to flow – the higher temperature text certainly uses words, but they are not necessarily English words. Our approach predicts the next *character*, not the next word and this is why we are so interested in narrowing down that sweet spot.

Now that our model is generated, we need to create our dataset. For the next portion, the classification portion, we want a good sample of the example corpus, and a good sample of our generated text at various temperature levels. I decided to go stepwise in this approach and generate text at temperature levels of .1, .2, .25, .3, .4, .5, .6, .7, .75, .8, .9, 1.0, 1.1, 1.2, and 1.25. Roughly 1000 samples from the original corpus were selected, and roughly 1000 rows of generated text approximately 400 characters in length were generated for each temperature level and added to a single pandas data frame. This resulted in 15,985 rows with two columns. The first column is the temperature used – the second is the actual text. The sample text was assigned a temperature of -1 so we could retrieve it easily later without having to pick it out of the middle of the generated text.

With our text generated, it was now time to put it in a form that could work with our logistic regression model. We opted to clean up all the text initially – we could have cleaned each dataset before running our data, but then we would have had to clean up our sample text twice for each set, so we just went ahead and did it all. Punctuation was stripped out and any stopwords were removed as well. Stopwords are words that are important to understanding text as a human but not so much as a computer – words like a, the, and, a distract our algorithm from the important context words and so we remove them.

Just under 2 hours later and we had a clean dataset! We now create the 15 datasets we will be using for the rest of the project. Each one consists of approximately 1000 rows of sample text directly from the Hound of the Baskervilles, sampled randomly, and approximately 1000 rows of generated text. Each dataset corresponds to a single temperature ranging between .1 and 1.25 as we previously generated. However, we want the second column, temp, to denote whether the text was generated (1) or a real sample (1). So, we replace all instances of -1, the default temperature for real text, with 0, and our temperature level with 1. This allows our model to classify each row as either Real or Generated.

From there, a test and a sample split are generated. Essentially, we sample a small portion of our dataset and set it aside. Then we train our model on the remaining, then see how it performs on the portion of the data we set aside. Instead of gauging the model's accuracy on data is has already seen before, we introduce new data though technically part of the original dataset. The split is roughly 75% (or 1500 rows) in the training set, and 25% (or 500 rows) in the test set. We performed this for all 15 datasets.

Then, for every dataset in our collection, we train a model. We used logistic regression, the more accurate of models from our previous exploration and trained each dataset separately. Then, we tested the accuracy of the model both in terms of actually guessing the test set correctly and also what we refer to as the F-Score. The F-Score is a metric which allows us to quantify the difference between a model that is accurate but not precise, and one that is precise but not accurate. Then the scores started rolling in.

As expected, the first few temperatures were a bloodbath. These were the generated texts with lower temperatures that were fairly obviously generated text. It was almost as if the text was stuck in a loop and tended to repeat itself over and over again. For temperatures .1, .2, .25, and .3, our model identified 100% of the genuine text as genuine text, and 100% of the generated text as generated text. But then! A light at the end of the tunnel. When our temperature was set to .4, the model identified two generated texts as real ones. We have begun to fool our model!

When finished with our model, we graphed the temperature versus the F-Score. Here is our result:

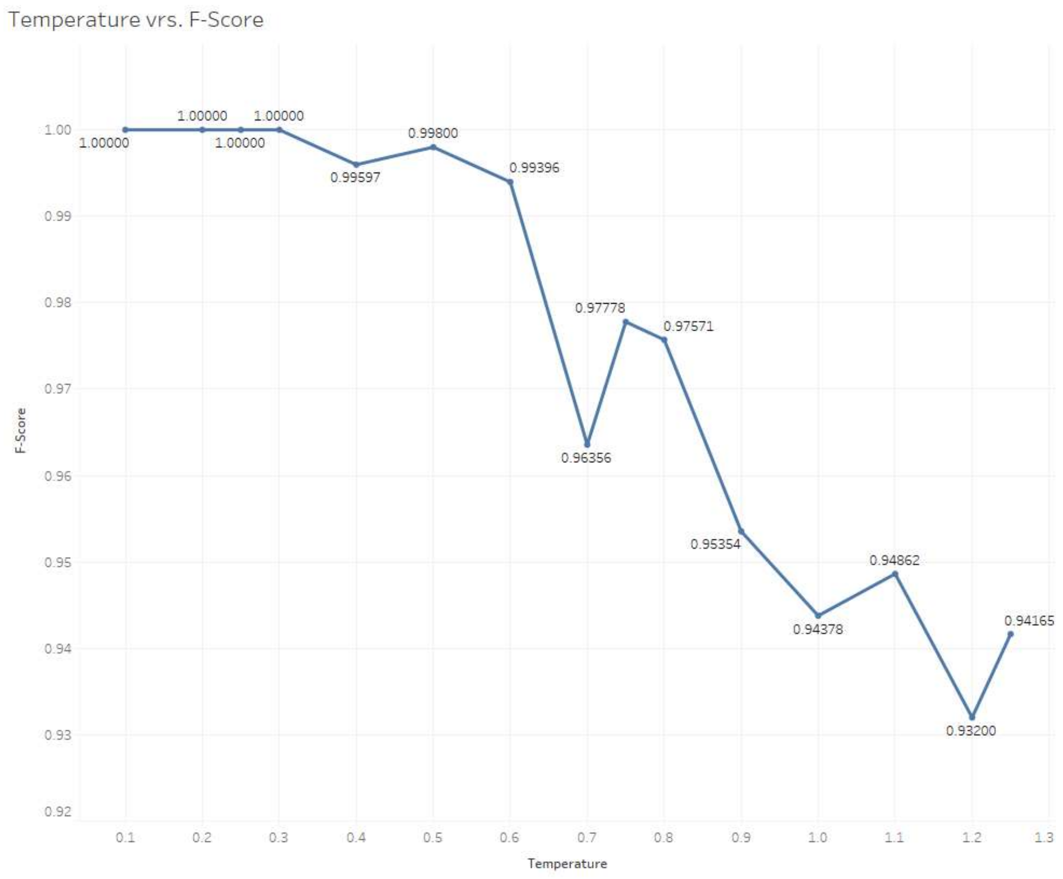

Fig. A: Temperature versus F-Score

As you can see, we start out strong but there is a pronounced dip at the .7 temperature that then rises again for .75 and .8 but then continues the downward trend. It seems my suggestion would be to go with the .7 temperature for now. Yes, the F-Score decreases as our temperature increases but at a certain point it becomes unreadable for a human and obvious that it is machine generated. Take for instance this example of a .7 temperature: “go carefully when there’s a lady in the case. even the best black dome and suspeck find. the words stood and set maked some u“vuk for i cannot the family from me sure as i do now sherlock “and the beart to friends than himerling spotieity and head fance shot from the moor, which from the death of the country. The man through that service at her. the man that this mind, on the countrysir complete intently this dear Watson” versus this example at temperature 1.25: “go carefully when there’s a lady in the case. even the best house to me that it drewing?” “did you sherlinghte?” cract, and ustime?” “on anxumintfull the wholl of mark movem” evening chear to me.’ you is but heply, and i ours, and sir henryfurly pufhingled his haund were upon a hilling still the everyonless crime!” “. then, barrymore anow at wend heaven favolwisital defenghisence over this crock, and he w”

A little bit more obvious to our human reader which version is our generated text, even though it fools our model more! It is for this reason that in the future I would generate the text somewhere in the .7 range. Enough that we start fooling the model, but not completely obvious to the human observer. How did our model handle the highest temperature, 1.25?

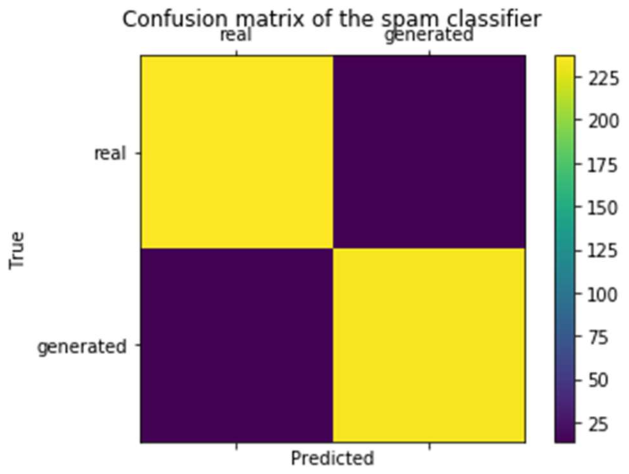

Figure B: Confusion Matrix, Temperature = 1.25

```
Accuracy Score: 0.942
Report:
              precision    recall  f1-score   support

           0       0.94      0.94      0.94       251
           1       0.94      0.94      0.94       249

   micro avg       0.94      0.94      0.94       500
   macro avg       0.94      0.94      0.94       500
weighted avg       0.94      0.94      0.94       500

F-Measure: 0.942
```

In this case, it identified 15 generated rows as real, and 14 real rows as generated. While it still identified the majority of rows correctly, we did have a bit of confusion. Let us look at the temperature we thought worked best for our purposes, .7:

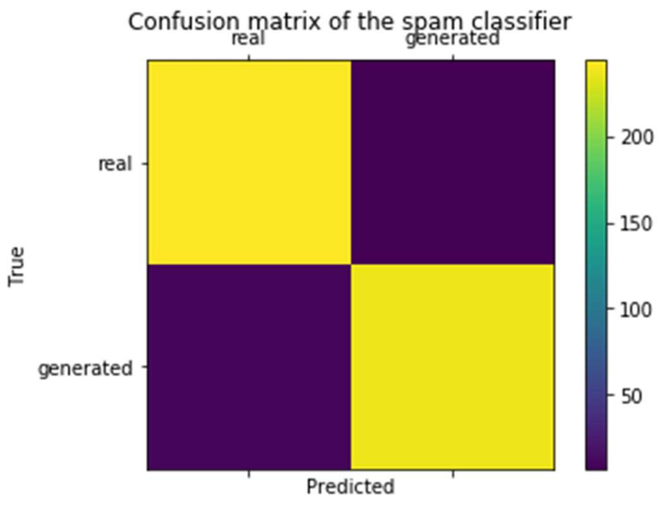

Figure C: Confusion Matrix, Temperature = 0.7

```
Accuracy Score: 0.964
Report:
              precision    recall  f1-score   support

           0       0.96      0.97      0.96       251
           1       0.97      0.96      0.96       249

   micro avg       0.96      0.96      0.96       500
   macro avg       0.96      0.96      0.96       500
weighted avg       0.96      0.96      0.96       500

F-Measure: 0.964
```

In this case, 11 generated rows were identified as real and 7 real rows were identified as generated. The performance, from a machine standpoint, is not quite as good at fooling our model as the 1.25 temperature but it is much better at blending in from a human perspective.

So, did we accomplish what set out to do? Absolutely! While our new chapters of the adventures of Sherlock Holmes are not quite in a publishable form, they pass muster when it comes to a quick perusal and that is what we were hoping to accomplish. As the first results trickled in, we were concerned that our model would be too good at identifying the generated text but as the temperature rose so did our hopes.

It is likely that as the cost to develop video games will simply continue to increase over time. It appears that development costs have increased tenfold for big budget games about every 10 years or so. Anything that can reduce development costs, free up developers for other projects or features, and speed up development times is going to be adopted quickly. This sort of procedurally-generated content is only the beginning – we have seen world building utilizing similar methods for years now. I would not be surprised if this technique is deployed in the near future with a more robust text generation algorithm!

References:

Alur, V. (2016, August 18). *Simple Logistic Regression using Keras*. Medium. https://medium.com/@the1ju/simple-logistic-regression-using-keras-249e0cc9a970.

Bansal, S. (2019, January 5). *Beginners guide to text generation using lstms*. https://www.kaggle.com/shivamb/beginners-guide-to-text-generation-using-lstms.

Brownlee, J. (2020, August 14). *Logistic Regression for Machine Learning*. Machine Learning Mastery. https://machinelearningmastery.com/logistic-regression-for-machine-learning/.

Chollet, F. *Keras documentation: Character-level text generation with LSTM*. Keras. https://keras.io/examples/generative/lstm_character_level_text_generation/.

Doyle, S. A. C. *THE HOUND OF THE BASKERVILLES*. The Hound of the Baskervilles, by Arthur Conan Doyle. https://www.gutenberg.org/files/2852/2852-h/2852-h.htm.

Jeffries, A. (2013, February 21). *You'll tweet when you're dead: LivesOn says digital 'twin' can mimic your online persona*. The Verge. https://www.theverge.com/2013/2/21/4010016/liveson-uses-artificial-intelligence-to-tweet-for-you-after-death.

K, B. (2020, August 22). *Next Word Prediction with NLP and Deep Learning*. Medium. https://towardsdatascience.com/next-word-prediction-with-nlp-and-deep-learning-48b9fe0a17bf.

Natural Language Toolkit¶. Natural Language Toolkit - NLTK 3.5 documentation. https://www.nltk.org/.

Stoop, W., & Van Den Bosch, A. *How Algorithms Know What You'll Type Next*. The Pudding. https://pudding.cool/2019/04/text-prediction/.

Yager, N. (2018, May 16). *Neural text generation*. Medium. https://medium.com/phrasee/neural-text-generation-generating-text-using-conditional-language-models-a37b69c7cd4b.