Hans Kamin Follow
CS @calpoly, ex-@infostretch, incoming @WalmartTech.
Jul 19, 2017 · 9 min read

Predicting Logic's Lyrics With Machine Learning

. . .

Logic has been a remarkable influence on my life since middle school, when I heard his song "All I Do" for the first time. The mixtape this song belongs to, *Young Sinatra*, singlehandedly made me a fan of hip-hop in all its forms, introducing me to styles old and new that I had otherwise never even considered listening to. I can associate every single one of the songs on that project with a particular feeling or moment from middle and high school, whether it was a time of magnificent joy or terrible sorrow. As a result, I've never encountered a verse or song by Logic that I didn't enjoy, so when I was assigned this project in a data

science class I was taking back in May, I immediately knew which musician I'd focus on.

More than any other coding lab I've been given, this one was by far the most intriguing and exhilarating to me, so I thought it'd be fun to write about it! I'll walk through the Python code I wrote and then discuss some of the strengths and weaknesses of this implementation, as well as how it can be improved in the future. And, of course, a *humongous* shout-out to Professor Dennis Sun at Cal Poly SLO for providing excellent solutions and help, and for assigning such an awesome lab through which to explore data science!

Before we begin, though, it would be wise to ***visit this webpage*** for a quick explanation of Markov Chains and how they work—this is crucial to understanding how to approach the problem. To put it simply, *if we were to model the weather as a Markov Chain, predicting tomorrow's weather would depend solely on today's conditions*.

. . .

Building The Algorithm

The crux of this implementation involves creating a Bigram Markov Chain to represent the English language. More specifically, our chain will be a dictionary object in which each key is a unique tuple consisting of a word and the word that follows it. Using bigrams rather than single words (unigrams) allows us to increase the accuracy and readability of our generated lines because it defines our model such that **the next word in a sentence is predicted based on the previous two words rather than just the immediately preceding one** (more details on this later).

Obtaining The Data

The true first step is to gather all of the lyrics we'll be analyzing. To do this, I web-scraped links to the lyrics for each of Logic's songs, then went through each link to gather all of the relevant text from them. The end result is a list in which each element is a string containing all of the lyrics to one song.

```
# coding: utf-8

# Hans Kamin
# Spring 2017

# Scraping Song Lyrics

import requests
import time
from bs4 import BeautifulSoup

links = []
for pagenum in range(1,3):
    url = "http://www.metrolyrics.com/logic-alpage-%d.
    soup = BeautifulSoup(requests.get(url).text, "html
    # First table on this page contains links to songs
    table = soup.find("table")
    for song in table.find_all('a'):
        links.append(song.get("href"))

# Enter each link and scrape all of the lyrics.
# Each element in our lyrics list will pertain to one
# Parsing through each link takes a while, expect long
lyrics = []
for link in links:
    time.sleep(0.1)
```

Web scraping Logic's lyrics.

The URL of each webpage differed only by the page number at its tail, so I was easily able to make that first `for` loop to iterate through both pages. The `soup` variable I made is an object from the `BeautifulSoup` library; it makes parsing and pulling data from websites incredibly simple. With help from the `requests` library, I was able to pass `soup` the entire HTML code of each webpage. The first table on the page contained all of the links that we needed in order to get to the lyrics of each Logic song, so I used another `for` loop to go through each row in the table; because hyperlinks in HTML are denoted by `a` and `href` tags, I was able to search for the tags to find and append each link to a list of `links`. The next set of `for` loops was used to iterate over each of the `links` I'd just obtained in order to grab each paragraph of text within each song, ultimately letting me append each set of `song_text` to a comprehensive list of `lyrics`. I had to use the `time.sleep`

function to make sure that I wouldn't get blocked or banned for making a ton of requests too quickly.

Creating The Chain

It's finally time to dig in and start building our Markov Chain. We write a function that iterates through each word in all of Logic's lyrics in order to generate the model by examining each sequence of two words and creating a list of all of the words that follow each sequence. For more efficient, practical traversal, we use `"<START>"`, `"<END>"`, and `"<N>"` tags to represent a song's beginning, its end, and its newline characters, respectively.

```
# coding: utf-8

# Hans Kamin
# Spring 2017

# Bigram Markov Chain Model

def train_markov_chain(lyrics):
    """
    Args:
      - lyrics: a list of strings, where each string r
                the lyrics of one song by an artist.
    Returns:
      A dict that maps a tuple of 2 words ("bigram") t
      words that follow that bigram, representing the
      chain trained on the lyrics.
    """

    # Initialize the beginning of our chain.
    chain = {
        (None, "<START>"): []
    }

    for lyric in lyrics:
        # Replace newline characters with our tag.
        lyric_newlines = lyric.replace('\n', ' <N> ')
        # Create a tuple representing the most recent
        last_2 = (None, "<START>")
        for word in lyric_newlines.split():
            # Add the word as one that follows the cur
```

Training our Markov Chain.

This `train_markov_chain` function takes in the `lyrics` list we created earlier. We initialize our `chain` with the key `(None, "<START>")` to mark the beginning of a song. As you've probably guessed, we begin with the `None` keyword because there aren't any words that come before the first word of a song. Iterating over each song, we replace all of the newline characters in a song's text with our custom tag, then create a `last_2` variable to track the current/most recent key encountered during iteration. Then, `for` each `word` in a song's lyrics, we insert the `word` into our chain by connecting it to the current key, then updating the current key to reflect the fact that we're moving to the next `word` . If this new key doesn't already exist in the chain, we

can simply create it with an empty list to reflect the fact that it hasn't been seen before. Once the final `word` in the song has been processed, we tack on an `"<END>"` tag and move on to the next song in our collection.

Predicting Lyrics

Once we've built and returned the dictionary representing our Markov Chain, we can move onto the final portion of the algorithm: generating predicted lyrics. Beginning from the `(None, "<START>")` key (the first key in our chain), we randomly sample one of the words in the list connected to that key, then shift the key we're currently examining to account for the word we just sampled. We continue this process all the way through until the `"<END>"` tag is finally encountered.

```
1
2  # coding: utf-8
3
4  # Hans Kamin
5  # Spring 2017
6
7  # Predicting Lyrics From Our Chain
8
9  import random
10
11 def generate_new_lyrics(chain):
12     """
13     Args:
14       - chain: a dict representing the Markov chain,
15                such as one generated by generate_new_l
16     Returns:
17       A string representing the randomly generated son
18     """
19
20     # a list for storing the generated words
21     words = []
22     # generate the first word
23     word = random.choice(chain[(None, "<START>")])
24     words.append(word)
25
26     # Begin with the first bigram in our chain.
27     last 2 = (None, "<START>")
```

Predicting new Logic lyrics.

Thus, after putting all this code together, we can `print(generate_new_lyrics(chain))` to display our predicted lyrics in the console. If you'd like to run all of this code yourself, you can check out my GitHub repository for access to Python files and a Jupyter Notebook.

It's imperative to note, however, that because I use simple random sampling to create new lyrics, I'm also randomizing how much output I actually receive. There were a select few instances in which I received less than one line or even just one word of output, but most of the time the algorithm printed out a giant amount of predicted lyrics. Nonetheless, after searching through the outputs I received from many runs of the algorithm, I got a handful of pretty good lyrics overall, ranging from raw punchlines to downright hilarious quips. Below you'll find my favorite ones, all of which I believe rather closely match Logic's style (with the exception of a few funny and/or weird ones I felt obligated to include).

> *"In the day she love to smoke, yes she fade away"*

> *"So I'm puffing on this vision, the night is my division"*

> *"I'ma show 'em how to act, I'ma get up and then on the back"*

> *"Praise Black Jesus now they call the cops, do it for the life that I'm puttin' on for the props"*

> *"I pretty much knew he was born with the heat, rock more solid than concrete"*

> *"Baby girl can I find humanity?"*

> *"Put my everything into the street, let alone the heat"*

> *"This m**********r better know the Feds is buggin'"*

> *"My life ain't mine, I need you to save me"*

> *"Everybody looking for the street, let alone the heat"*

> *"I'ma keep rapping about all of you guys? F**k all that s**t I was gone for a reason"*

> *"Oh my, my, my, feeling this villainous vibe"*

> *"But I take the bus from my problems, Lord help me solve them"*

> *"Now I'm praying for somebody to save me, no matter what you believe is right"*

> *"You got everything to lose, like a goddamn king"*

> *"Pawns tend to carry on with no dial tone"*

> *"Yeah, know what? I'll make dead f****n' presidents to represent me"*

> *"The life that I'm puttin' on, this is a facade"*

> *"She don't wanna cry anymore, destitute and less informed"*

> *"I see myself at the Louvre, and I know my mind playing tricks on me"*

> *"I feel like I'm killin' my dreams, life fading away"*

> *"Why nobody wanna say I can rap"*

> *"Like abracadabra when that magician pull up the road"*

> *"Homies in my studio, and I was strollin' down the highway"*

> *"This rap s**t another day, another book"*

> *"I see good people who make it rain like no other man"*

> *"I know where to begin to make a killin'"*

> *"Yo, I'ma keep all of this new left over residue"*

> *"Trust me girl I won't be mad, if you heard different someone lied"*

> *"People thinking they on his level, they ain't ready for more bottles"*

> *"Anybody that's riding with me trynna get it like that now"*

. . .

Analyzing Our Results: Strengths & Weaknesses

Observing many, many outputs from our bigram implementation and those from a unigram implementation allows us to reach some important conclusions:

1. **Our model's predictions are accurate, but often recycled.** It's important to note that many of our predicted lines turned out to be nearly identical to lines Logic has actually written, i.e. half of a line from one verse/song combined with half of a line from another verse/song. This is to be expected, as using bigrams yields less variability in predicted words due to basing predictions off the previous two words instead of the one most recent, resulting in sequences of three or more words coming from the same Logic lyric. To put it simply, *using bigrams instead of single words increases readability and similarity to Logic's style, but decreases creativity.*
2. **Our model is slower and generates less output.** The unigram model runs faster because the dictionary object representing its

> Markov Chain has far fewer keys. Our model has so many more keys because it has to process tuples of two words. Furthermore, as I mentioned before, there were times when I received very little to no output, and generally I received less than I did from the unigram implementation. This can be attributed to the smaller number of possibilities for the next word when we're basing it off the previous two words.

So where do we go from here? We've highlighted the strengths and weaknesses of our implementation; how do we actually mitigate those weaknesses and make our model even better? Discerning the central Markov Assumption that limits the model we built is the key to discovering a superior design.

. . .

Finding A Better Way

Modeling a situation with a Markov Chain necessitates assuming that the situation itself satisfies one key statement: a prediction for the next state only depends on the status of the current state, not the rest of the situation's history. For example, using Markov Chains to predict tomorrow's weather requires the conclusion that weather from the past two weeks or more has no effect on tomorrow's conditions—something I think we can all agree sounds pretty far-fetched. Thus, even though using bigrams helped us decrease the magnitude of this assumption in our model, its impact was still prevalent and weakened our results. We need to find an alternative to our model that can at the very least make fewer assumptions.

A **recurrent neural network** is one example of a replacement we can use. While I won't go into much detail here about RNNs, mostly because I'm still only cracking the surface with them myself, I will provide some brief notes. Two of the key characteristics of RNNs are that they don't assume that all inputs are independent of each other and that they're capable of keeping a history of what they've processed, both of which are necessary to improving our model. For more information on how RNNs work and how to implement them, check out the Wikipedia page as well as this tutorial; I'll be learning from both to eventually update my code for better predictions.

. . .

If you've made it this far, thanks for reading about and taking a glimpse into my growing interest in machine learning! Data science as a whole already has so many fascinating and creative applications. I look forward to exploring the many nuances and intricacies in further detail as I work on more projects and continue to improve as a developer. After all, as Logic once wrote (and Paul Brandt before him), how can the sky be the limit when there are footprints on the moon?

Philadelphia, June 2013

Special thanks to my sister Kelsi Kamin for the motivation and constructive feedback she provided as I wrote this!