Case Study -3

Word prediction using Markov Model with High and Low Token Volume

by

Pradeep Sathyamurthy

Under the guidance of: Prof J. Gemmell

DePaul University

16th November 2017

Table of Contents

Abstract:

With the invention of smart phones, we have come across many messenger applications like what's-app, Facebook chat, etc., Whenever I type a message in these applications I get an auto suggestion for the next word. I was fascinated with this technology, thing which fascinated me more is my phone providing a suggestion in local dialects at times correctly. I always want to find the working model behind this functionality of word prediction and with this course (CSC-529) 'Advance Data Mining' I could find that Markov model is more used by these kinds of applications. So, as part of this case study, I will first provide my understanding on Markov's model, hidden Markov's model and then explain how I applied my Markov model to build a word prediction output from a song lyrics I downloaded from a famous music website called https://www.musixmatch.com/ which is a low volume sequential data and with 'Sherlock Holmes story' which is a high volume to sequential data.

Introduction:

In messenger's auto suggestion makes our job easy in doing fast typing. Such application uses the smart phones capability of building a Markov's Model to suggest the text that would occur next when I type a word based on the probability distribution measure in its background. It uses the current state to predict then next state, for e.g. if I have trained my messenger applications with my house address as 901 South Ashland Avenue then whenever I type 901 my messenger would suggest South and when I select South it suggest me Ashland and when I select Ashland it will suggest Avenue. Thus, it directly or indirectly helps me in building my complete address in the end. However, if I try to type the same pattern in my friend's mobile, such suggestions do not occur. This is because my smart phone is trained with my address details '901 South Ashland Avenue' numerous number of times in various occasions. Each time I type this address, it measures the probability of words occurrence and accordingly suggest me accurately in future.

In this project, I would also want to experiment one such this like producing a pair words in which word one will suggest the next word but using a text downloaded from a music lyrics maintenance website called http://musicmatch.com

Process Work Flow in this Case Study:

Dataset Description:

For this case study, I tried to scrape data from https://developer.musixmatch.com/ using the API (application programming interface) exposed by MUSIXMATCH. In this case study, I wished to extract the lyrics of track '***BATTERY***' composed by the band '***METALLICA'***. I used below API methods:

1. ***artist.search*** = Search for artists in MUSIXMATCH database
2. ***track.search*** = Search for track in MUSIXMATCH database
3. ***track.lyrics.get*** = Get the lyrics of a track

These API methods where used to extract the lyrics of 'BATTERY' from METALLICA and the same was stored in prady_lyrics.txt. This text file 'prady_lyrics.txt' is used as input to train the Markov's Model from which I tried to generate the word predictions based on the probability distribution of the word from the current state.

Data Scraping:

Web data scraping[1] is a technique to extract data from a website. It is an activity which needs to be done with responsibility and it is subjected to the terms and condition of each website. Few websites like Facebook or twitter enforce its users to login before a data is scraped from their website while there are few websites like job portal, flight booking websites which do not ask user to login while displaying their information. However, if any action needs to be performed on that information displayed, then website will request user to register & login or some kind of authentication. Web scraping, I do as part of this case study is from musixmatch.com which is a world's largest lyrics platform where users can search and share lyrics. It has 60 million users and 14 million lyrics. Musixmatch.com share an API which can be accessed with an authentication key.

With necessary authentication and above said methods exposed by API, I could extract any song's lyrics in musixmatch.com using a JSON format. However, since I used a free subscription, they allowed me to extract only 30% of any song lyrics and not a complete song. JSON, being a very thin wire format, which resembles dictionary structure of python, I did not used any extensive python package to extract or treat the data. Below is the data which was extracted from album 'Battery' composed by 'Metallica' which can be considered as a sample of sequential data format which act as a best input for our Markov's model to predict text:

• Lashing out the action • Returning a reaction • Weak are ripped and torn away • Hypnotizing power • Crushing all that cower • Battery is here to stay • Smashing through the boundaries	• Lunacy has found me • Cannot stop the battery • Pounding out aggression • Turns into obsession • Cannot kill the battery • Cannot kill the family • Battery is found in me • Battery, Battery

Markov's Model:

Markov model is one of the machine learning techniques which is used to study or experiment with sequential data. As thought by Prof, it is a stochastic model, that is the model is used for randomly changing variables and events. Markov's model very importantly depends on the current state, and not on previous states which occurred before. This model heavily depends on the weighted distribution of sequential data.

Example with an origin sentence:

There are numerous application that depends on the Markov's model, as part of this case study we will restrict our explanations with respective to word prediction application, where when you type one word an application predicts the next most suitable word to follow. Thus, the word which get suggested next completely depends on the current word and not on the word before it. Let us first discuss this with a simple example below, you must be familiar with many tongue twisters, below is one such, let me call this an Origin sentence:

"Can you can a can as a canner can can a can?"

Weighted distribution:

For the above origin sentence, let us try to understand the weighted distribution. Tokens and keys are two main parameters to find a weighted distribution. 'Keys' are the unique words in a sentence and 'Token' is a word in a sentence. In above origin sentence, we have 5 Keys and 12 tokens:

KEYS: Can, you, a, as, canner

TOKENS: can, you, can, a, can, as, a, canner, can, can, a, can

When we look at the keys distribution/occurrence in tokens we get:

Keys	**Can**	**You**	**a**	**as**	**Canner**
Occurrence	6	1	3	1	1

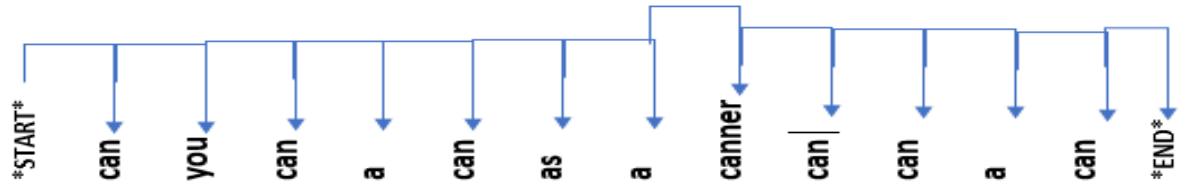
By looking at the above word distribution we can say that word 'CAN' as a probability to occur 6 out of 12 times which is a probability of 0.5. That is key 'CAN' occurs twice when compare to key 'a' and occurs 6 times as much as other keys in the origin sentence. Thus, if I randomly pick of a word in this origin sentence, I can predict the next word will yield me 'CAN' as its probability is 0.5.

Thus, if we compute a weighted distribution for our origin sentence as: number of time a key occurs with respective to total number of tokens in an origin sentence, we get below weighted distribution:

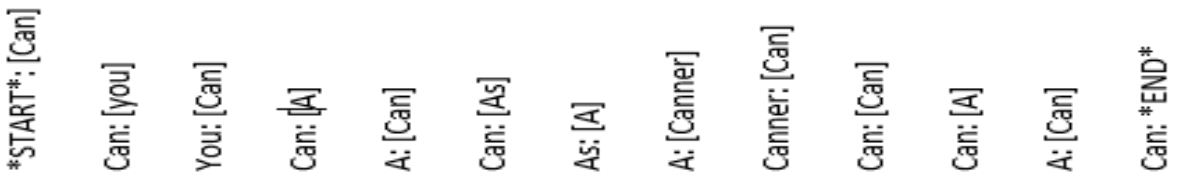
1. Can: 6/12 = 0.5
2. You: 1/12 = 0.083
3. A: 3/12 = 0.25
4. As: 1/12 = 0.083
5. Canner: 1/12 = 0.083

Working of Markov's Model:

As said before Markov's model is a stochastic model which assumes future state depends upon the current state and not on the events which occurred before it. Thus, for our word prediction for the above example we can represent, it shows how one token leads to another:



Let us pair every token that leads to next token:



Now let us organize the above sequence by their first token, we see something interesting here, each token is followed only by a possible key to follow it:

1. ***START*: [Can]**
2. **Can: [You, A, As, *END*]**
3. **You: [Can]**
4. **A: [Can, Canner]**
5. **As: [A]**
6. **Canner: [Can]**
7. ***END*: [none]**

Now, if we give the above structure to someone, and say start with a *Start* key and predict the next word as the key corresponds to that token in order. Look at the below samples as in how we can construct our origin sentence from above structure:

I am starting with *START* ->

1
Can

Choose 1st key for token 'Can' ->

1	2
Can	You

Choose 1st key for token 'You' ->

1	2	3
Can	You	Can

Since 1st key of 'Can' is already used, use 2nd Key ->

1	2	3	4
Can	You	Can	a

1st Key of token 'A' ->

1	2	3	4	5
Can	You	Can	a	Can

3rd Key of 'Can' ->

1	2	3	4	5	6
Can	You	Can	a	Can	As

1st Key of 'As' ->

1	2	3	4	5	6	7
Can	You	Can	a	Can	As	a

2nd Key of 'A' ->

1	2	3	4	5	6	7	8
Can	You	Can	a	Can	As	a	Canner

1st Key of 'Canner' ->

1	2	3	4	5	6	7	8	9
Can	You	Can	a	Can	As	a	Canner	Can

Since, there is no 4th Key of 'Can' -> System do not suggest any word.

	1	2	3	4	5	6	7	8	9	
Start	Can	You	Can	a	Can	As	a	Canner	Can	End

However, if we insist a word has 12 characters. Using the weighted average, we will randomly choose ‘Can’ again as this is the key with highest probability of 0.5 when compare to others, thus we get:

1	2	3	4	5	6	7	8	9	10
Can	You	Can	a	Can	As	a	Canner	Can	Can

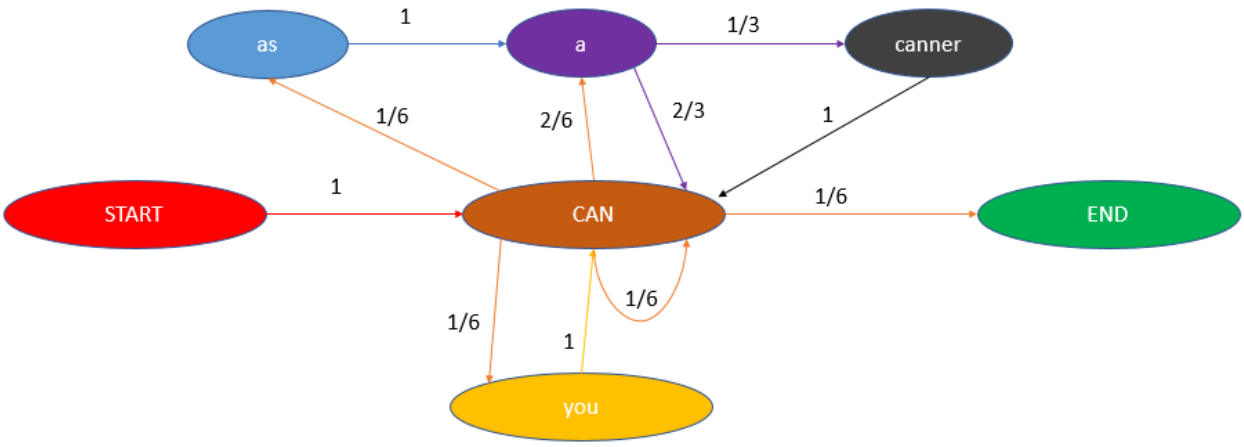
Thus, can will have 3 suggestions and if we go in order [You, A, As, *END*]:

1	2	3	4	5	6	7	8	9	10	11
Can	You	Can	a	Can	As	a	Canner	Can	Can	You
1	2	3	4	5	6	7	8	9	10	11
Can	You	Can	a	Can	As	a	Canner	Can	Can	a
1	2	3	4	5	6	7	8	9	10	11
Can	You	Can	a	Can	As	a	Canner	Can	Can	As

Thus, key to follow token ‘You’ can be -> Can, ‘a’ -> Can, ‘as’ -> a

1	2	3	4	5	6	7	8	9	10	11	12
Can	You	Can	a	Can	As	a	Canner	Can	Can	You	Can
1	2	3	4	5	6	7	8	9	10	11	12
Can	**You**	**Can**	**a**	**Can**	**As**	**a**	**Canner**	**Can**	**Can**	**a**	**Can**
1	2	3	4	5	6	7	8	9	10	11	12
Can	You	Can	a	Can	As	a	Canner	Can	Can	As	a

Probability Diagram for above model and word prediction that occurs:



Whatever is explained in above section has been represented in flow chart representation here. If we have trained our Markov model with origin word then below are the suggestion our application will display:

- Start with word ‘Can’, once it is selected next suggested words would be [a, as, you]
- When ‘As’ is selected, suggested words would be [a]
- When ‘A’ is selected, suggested words would be [Can, canner]
- When ‘canner’ is selected, suggested word would be [Can]
- When ‘you’ is selected, suggested word would be [Can]

Hidden Markov's Model:

In our above example states are visible with each transition from one state to another is denoted by a probability measure. However, there are few Markov's model with un-observed state. Such kind of Markov's model built with hidden state into consideration as in simple terms called as Hidden Markov's model. For the application that we are dealing with this study, that is word prediction, hidden states are out of scope.

Building Markov's Model with high volume of token:

More the token we have better we can train the Markov chain model. As my first simple application, I downloaded a story file from web which is '`The Adventures of Sherlock Holmes, by Arthur Conan`'. The file is named as 'sherlock.txt'. This file has a total of 5,94,930 tokens. With such high volume of token, we should be able to generate a complete sentence which are Markov's model generated and are not part of the actual text.

For this I used '**markovify'** which is a python package which is implicitly used to build Markov models and generate text snippets for various application. I used this module and trained my model with 'sherlock.txt' file. Once trained, I used the 'make_short_sentence' method built as part of 'markovify' module to generate short sentences, I restricted myself to generate 10 short sentences with each not exceeding more than 150 tokens in length. Below is the sample text and Markov generated sentences:

Training the Model:

```
In [84]: # Importing the packages
         import markovify as mk
```

```
In [87]: # Get raw text as string
         with open("sherlock.txt") as f:
             text = f.read()
```

```
In [88]: text
```

```
Out[88]: 'ï»¿Project Gutenberg\'s The Adventures of Sherlock Holmes, by Arthur Conan Doyle\n\nThis eBook is for the use of anyone anywh
         ere at no cost and with\nalmost no restrictions whatsoever.  You may copy it, give it away or\nre-use it under the terms of th
         e Project Gutenberg License included\nwith this eBook or online at www.gutenberg.net\n\n\nTitle: The Adventures of Sherlock Ho
         lmes\n\nAuthor: Arthur Conan Doyle\n\nPosting Date: April 18, 2011 [EBook #1661]\nFirst Posted: November 29, 2002\n\nLanguage:
         English\n\n\n*** START OF THIS PROJECT GUTENBERG EBOOK THE ADVENTURES OF SHERLOCK HOLMES ***\n\n\n\n\nProduced by an anonymous
         Project Gutenberg volunteer and Jose Menendez\n\n\n\n\n\n\n\n\nTHE ADVENTURES OF SHERLOCK HOLMES\n\nby\n\nSIR ARTHUR CONAN D
         OYLE\n\n\n\n   I. A Scandal in Bohemia\n  II. The Red-headed League\n III. A Case of Identity\n  IV. The Boscombe Valley Myste
         ry\n   V. The Five Orange Pips\n  VI. The Man with the Twisted Lip\n VII. The Adventure of the Blue Carbuncle\nVIII. The Adven
```

```
In [89]: # Build the model.
         text_model = markovify.Text(text)
```

Testing the Model to generate sentences:

```
In [95]: # Print three randomly-generated sentences of no more than 140 characters
         for i in range(10):
             print(text_model.make_short_sentence(150))
```

```
I assure you that he was writing me a yellow-backed novel.
And there is no wonder that no one to the Project Gutenberg-tm electronic works.
I should prolong a narrative which promises to be still.
It was a delicate part which he had a fear for her.
- You comply with the three figures?
Holmes went to the corner of the friends to hush the matter up, so that I might find it upon the other side.
There was something in her interests.
Well, then, here on this page are the true state of things you can have the great kindness to recommence your narrative.
It was easy to see my husband.
I put my handkerchief round it, woods on three sides, and on the fourth was shuttered up.
```

We see model has generated some sentences which makes some sense and not a random one, for e.g. the sentence '`I assure you that he was writing me a yellow-backed novel.`' This sentence makes a perfect sense. Howe ver, this was model generated based on the weighted probability on each word.

,

Building Markov's Model with low volume of token:

As explained in introductory section, I planned to download the lyrics of 'Butterfly' composed by 'Metallica'. Since the Musixmatch API allowed to download only 30% of the lyrics. This seems to be a data with low volume of tokens and hence might produce a poor result. However, as part of this case study my intension was to build a simple Markov's model instead of using the markovify in-built package. Below are the observations of the same:

Original Lyrics:

Below is the 30% of the lyrics downloaded through Musixmatch API:

• Lashing out the action • Returning a reaction • Weak are ripped and torn away • Hypnotizing power • Crushing all that cower • Battery is here to stay • Smashing through the boundaries	• Lunacy has found me • Cannot stop the battery • Pounding out aggression • Turns into obsession • Cannot kill the battery • Cannot kill the family • Battery is found in me • Battery, Battery

Lyrics Generated by Markov's Model:

Below is the song generated by the Markov's model. Since the volume of token is low, we see that sentence generated makes no real sense.

3. Generating words from Markovs Model Trained

In [50]:
```
if __name__ == '__main__':
    mc = MarkovChain()
    mc.add_file('prady_lyrics.txt')
    lyrics = mc.generate_text()
    str1 = ' '
    for lyric in lyrics:
        lyric = str1.join(lyric)
        print(lyric)
```

```
through the boundaries
me battery battery
found me cannot
to stay smashing
cower battery is
battery is here
the action returning
lunacy has found
weak are ripped
the battery cannot
a reaction weak
battery is found
lashing out the
that cower battery
kill the battery
me cannot stop
```

Conclusion

Thus, as part of this case study we saw how to build a Markov's model, train the same with large volume of token and with small volumes of tokens and tested the model by generating the text which is otherwise the prediction of word based on the current word selected and there by framing the whole sentence. I started to generate data with a **starting state**, then I kept looking at the **possible keys** that could **follow the current state** and **make a decision** based on *probability* and *randomness* (**weighted probability**). I kept repeating this until I generate a length of sentence.

Jupyter Notebook

I have coded these in Jupyter notebook which I have shared in the below path:

1. View notebook for word generator from Markov trained with high volume token: click here
2. View notebook for word generator from Markov trained with low volume token: click here

Reference

[1] Research paper on Social Media Analytics https://link.springer.com/article/10.1007/s00146-014-0549-4

[2] https://hackernoon.com/from-what-is-a-markov-model-to-here-is-how-markov-models-work-1ac5f4629b71

[3] https://pypi.python.org/pypi/markovify

Python Code:

Code to build Markov model for Large volume of tokens using inbuilt module:

```
# Importing the packages
import markovify as mk
```

```
# Get raw text as string
with open("sherlock.txt") as f:
    text = f.read()
```

```
text
```

```
# Build the model.
text_model = markovify.Text(text)
```

```
# Print three randomly-generated sentences of no more than 140 characters
for i in range(10):
    print(text_model.make_short_sentence(150))
```

Code to build Markov model for Small volume of tokens using python implementation:

Building the model:

```
import re # used to handle regular expression in extracted file for data clean up

import random # used in building markov model
```

```
from collections import defaultdict, deque # used to build collections in markovs model
import requests # for extracting data through API
# Markov Chain generator
# This is a text generator that uses Markov Chains to generate text using a uniform distribution.
# num_key_words is the number of words that compose a key (suggested: 2 or 3)


class MarkovChain:


  def __init__(self, num_key_words=3):
    self.num_key_words = num_key_words
    self.lookup_dict = defaultdict(list)
    self._punctuation_regex = re.compile('[,.!;\?\:\-\[\]\n]+')
    self._seeded = False
    self.__seed_me()


  def __seed_me(self, rand_seed=None):
    if self._seeded is not True:
      try:
        if rand_seed is not None:
          random.seed(rand_seed)
        else:
          random.seed()
        self._seeded = True
      except NotImplementedError:
        self._seeded = False


# Build Markov Chain from data source.
# Use add_file() or add_string() to add the appropriate format source


  def add_file(self, file_path):
```

```
    content = ''
    with open(file_path, 'r') as fh:
      self.__add_source_data(fh.read())

  def add_string(self, str):
    self.__add_source_data(str)

  def __add_source_data(self, str):
    clean_str = self._punctuation_regex.sub(' ', str).lower()
    tuples = self.__generate_tuple_keys(clean_str.split())
    for t in tuples:
      self.lookup_dict[t[0]].append(t[1])

  def __generate_tuple_keys(self, data):
    if len(data) < self.num_key_words:
      return

    for i in range(len(data) - self.num_key_words):
      yield [ tuple(data[i:i+self.num_key_words]), data[i+self.num_key_words] ]

# Generates text based on the data the Markov Chain contains
# max_length is the maximum number of words to generate

  def generate_text(self, max_length=20):
    context = deque()
    output = []
    if len(self.lookup_dict) > 0:
      self.__seed_me(rand_seed=len(self.lookup_dict))
      idx = random.randint(0, len(self.lookup_dict)-1)
      chain_head = list(self.lookup_dict.keys())
```

```
    context.extend(chain_head)


    while len(output) < (max_length - self.num_key_words):
      next_choices = self.lookup_dict[tuple(context)]
      if len(next_choices) > 0:
        next_word = random.choice(next_choices)
        context.append(next_word)
        output.append(context.popleft())
      else:
        break
    output.extend(list(context))
  return output
```

Data Extraction from Musixmatch.com:

```
apikey = <I purposely masked it>
# Getting the artist_id from band name search
url_artist = "http://api.musixmatch.com/ws/1.1/artist.search"
payload_artist = {'q_artist': 'Metallica', 'apikey': apikey, 'format':'json'}
response_artist = requests.get(url_artist, params=payload_artist)
print(response_artist)
response_artist = response_artist.json()
artist_id = response_artist['message']['body']['artist_list'][0]['artist']['artist_id']
print(artist_id)
# Getting all track_ids available for the artist
tracks = []
url_tracks = "http://api.musixmatch.com/ws/1.1/track.search?"
payload_tracks = {'q_track':'Battery', 'f_artist_id': artist_id,'page':1,'page_size': 10, 'page': 3, 'apikey': apikey}
response_tracks = requests.get(url_tracks, params=payload_tracks)
print(response_tracks)
response_tracks = response_tracks.json()
```

```
for item in response_tracks['message']['body']['track_list']:
    for i in item:
        track_id = item[i]["track_id"]
        print(track_id)
        tracks.append(track_id)
print(len(tracks))
# Getting lyrics from track_ids
data = []
url_lyrics = "http://api.musixmatch.com/ws/1.1/track.lyrics.get?"
track_count=1
for track_id in tracks:
    print(track_id)
    payload_lyrics = {'track_id' : int(track_id), 'apikey': apikey}
    response_lyric = requests.get(url_lyrics, params=payload_lyrics)
    response_lyric = response_lyric.json()
    print(response_lyric['message']['header']['status_code'])
    if (response_lyric['message']['header']['status_code'] == 200):
        response_lyric = response_lyric['message']['body']['lyrics']
        lyric_text = response_lyric['lyrics_body']
        # Cleaning the Data
        bad_string = '''******* This Lyrics is NOT for Commercial use *******'''
        lyric_text = lyric_text.replace(bad_string, '')
        another_bad_string = '''...'''
        lyric_text = lyric_text.replace(another_bad_string, '')
        data.append(lyric_text)
        track_count+1
        print(track_count)

print('Saved a total of %s track' %(track_count))
```

```
# Save data in a file


# Open a file
f = open("prady_lyrics.txt", "wb")
for lyric in data:
    lyric = lyric.encode('ascii', 'ignore')
    f.write(lyric)


# Close opend file
f.close()
```

Generating words from Markov's Model build above:

```
if __name__ == '__main__':
    mc = MarkovChain()
    mc.add_file('prady_lyrics.txt')
    lyrics = mc.generate_text()
    str1 = ' '
    for lyric in lyrics:
        lyric = str1.join(lyric)
        print(lyric)
```