Character level language models and sequence modelling

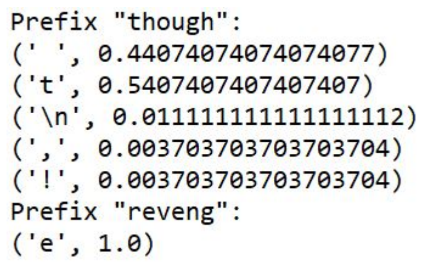
Xiulong Liu
Department of Electrical and Computer Engineering
University of Washington
SEATTLE, WA 98195
`xl1995@uw.edu`

Abstract

Character level language generation is a very basic problem in studies of language model. For unstructured text, n-gram model exhibits amazing performance in language generation tasks in terms of spelling and punctuation accuracy, and it is highly intuitive to analyze. In comparison, Recurrent Neural Network using LSTM and GRU has similar performance but requires complex model. By milestone, I complete training n-grams model and LSTM model on Shakespeare's dataset with various hyperparameters, i.e, orders in n-grams and number of layers in RNN. And the performance of generated texts are evaluated using perplexity and cross-entropy loss under different parameters.

1 n-grams

n-grams model is a probablistic model for predicting the next token based on previous $n-1$ tokens.The basic assumption made is that the next token depends only on previous $n-1$ tokens such that it follows Markov properpty. In character-level model, n is called the order of n-grams. n-gram is essentially a MLE, which maximizes $P(x_m|x_{m-1}, x_{m-2}, \ldots, x_{m-n+1})$, where x_m is the next character to be predicted, $x_{m-1}, x_{m-2}, \ldots, x_{m-n+1}$ is previous $n-1$ character. In experiment, the conditional probability to be maximized is modeled by building a dictionary of all possible prefixes with length $n-1$ in training set, and for each prefix, a histogram of the next possible character is built to estimate the distribution of conditional probability. From training, we could inspect some histograms by polling the dictionary of fixed length of prefix. Here are some examples shown in Figure 1.



```
Prefix "though":
(' ', 0.44074074074074077)
('t', 0.5407407407407407)
('\n', 0.011111111111111112)
(',', 0.003703703703703704)
('!', 0.003703703703703704)
Prefix "reveng":
('e', 1.0)
```

Figure 1: sample histograms for prefix on the order of 6.

Note that with prefix "though", there are 2 most likely next character, one is 't', the other is ' ' because they represent word "thought" and "though" respectively. With prefix "reveng", the only possible next character is 'e', which makes the word "revenge". Then I use the histogram built through training to

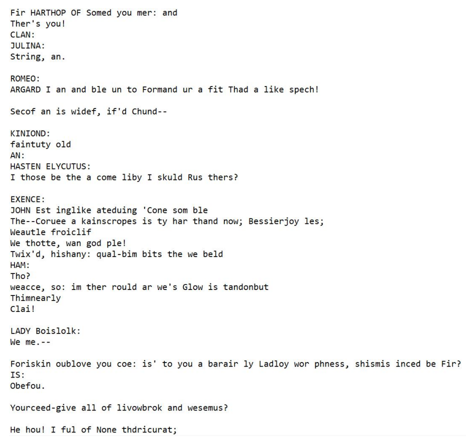
Table 1: Perplexity of n-grams on different orders

n-grams perplexity		
Order	Perplexity(NY Times)	Perplexity(Sonnets)
1	13.97	11.77
2	10.70	7.73
3	10.13	6.35
4	12.95	7.528
5	20.90	11.77
6	32.56	19.47

generate sample paragraphs. At first, I choose the next character with maximum likelihood and adjust the order from 1 to 6, and evaluate their perplexity on test set. The test sets I use are New York Times article and Shakspeare Sonnet. Perplexity is a measurement of how well a probability distribution or probability model predicts a sample. For n-grams, the perplexity is described by Equation (1).

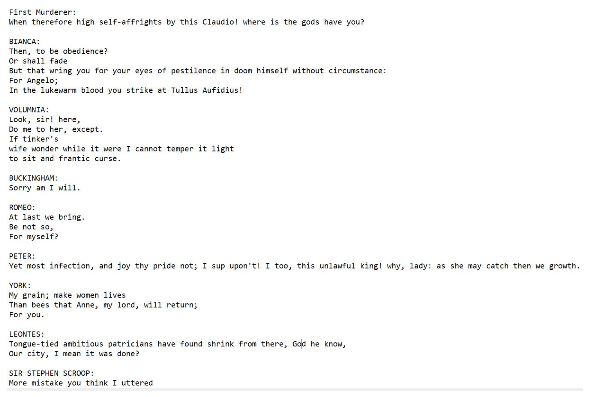
$$perplexity = \sqrt[n]{\prod_{i=1}^{n} \frac{1}{P(x_i|x_{i-m}, x_{i-m+1}, \ldots, x_{i-1})}} \tag{1}$$

More concretely, perplexity is aimed at measuring the average degree of uncertainty for predicting the next token. I first use unsmoothed version of n-gram to train and observe the perplexity of training set. As expected, the perplexity increases as order increases. However, in order to be tested, a smoothed version will be used to avoid non-exsist n-gram in test set. In experiment, I use add one Laplace smooth method. Then I feed my model into two test sets, perplexity of n-grams model on different orders are observed in Table 1. It is clear that for the same order, Shakespeare's Sonnet always achieves smaller perplexity than New York Times. This is intuitive because similarity of two Shakespeare's work will definitely be more "close" than the similarity of Shakespeare's work and NY Times article. On the other hand, as order of level increases, the perplexity of test set first decreases and then increases, and both achieves minimum at order of 3. This is because when order is very low, infernce is hard to make because the number of previous tokens our inference is based on is too small. On the other side, when the order becomes really high, we just overfit the training data in a sense that character combinations in training set will be almost unlikely to appear in other contexts. In my experiment, order of 3 is the best choice to minimize the perplexity of n-grams. are relatively small



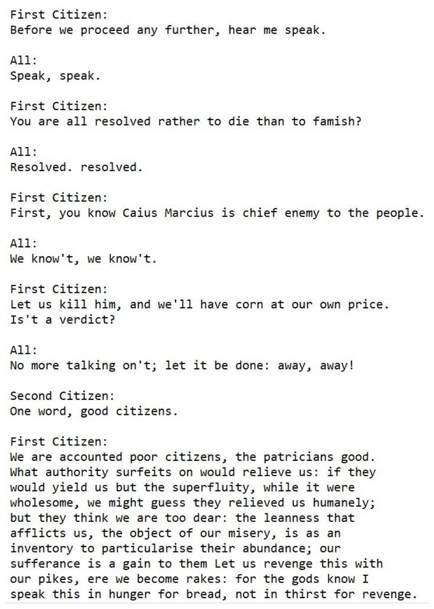
```
Fir HARTHOP OF Somed you mer: and
Ther's you!
CLAN:
JULINA:
String, an.

ROMEO:
ARGARD I an and ble un to Formand ur a fit Thad a like spech!

Secof an is widef, if'd Chund--

KINIOND:
faintuty old
AN:
HASTEN ELYCUTUS:
I those be the a come liby I skuld Rus thers?

EXENCE:
JOHN Est inglike ateduing 'Cone som ble
The--Coruee a kainscropes is ty har thand now; Bessierjoy les;
Weautle froiclif
We thotte, wan god ple!
Twix'd, hishany: qual-bim bits the we beld
HAM:
Tho?
weacce, so: im ther rould ar we's Glow is tandonbut
Thimnearly
Clai!

LADY Boislolk:
We me.--

Foriskin oublove you coe: is' to you a barair ly Ladloy wor phness, shismis inced be Fir?
IS:
Obefou.

Yourceed-give all of livowbrok and wesemus?

He hou! I ful of None thdricurat;
```

Figure 1: order = 2



```
First Murderer:
When therefore high self-affrights by this Claudio! where is the gods have you?

BIANCA:
Then, to be obedience?
Or shall fade
But that wring you for your eyes of pestilence in doom himself without circumstance:
For Angelo;
In the lukewarm blood you strike at Tullus Aufidius!

VOLUMNIA:
Look, sir! here,
Do me to her, except.
If tinker's
wife wonder while it were I cannot temper it light
to sit and frantic curse.

BUCKINGHAM:
Sorry am I will.

ROMEO:
At last we bring.
Be not so,
For myself?

PETER:
Yet most infection, and joy thy pride not; I sup upon't! I too, this unlawful king! why, lady: as she may catch then we growth.

YORK:
My grain; make women lives
Than bees that Anne, my lord, will return;
For you.

LEONTES:
Tongue-tied ambitious patricians have found shrink from there, God he know,
Our city, I mean it was done?

SIR STEPHEN SCROOP:
More mistake you think I uttered
```

Figure 2: order = 6



```
First Citizen:
Before we proceed any further, hear me speak.

All:
Speak, speak.

First Citizen:
You are all resolved rather to die than to famish?

All:
Resolved. resolved.

First Citizen:
First, you know Caius Marcius is chief enemy to the people.

All:
We know't, we know't.

First Citizen:
Let us kill him, and we'll have corn at our own price.
Is't a verdict?

All:
No more talking on't; let it be done: away, away!

Second Citizen:
One word, good citizens.

First Citizen:
We are accounted poor citizens, the patricians good.
What authority surfeits on would relieve us: if they
would yield us but the superfluity, while it were
wholesome, we might guess they relieved us humanely;
but they think we are too dear: the leanness that
afflicts us, the object of our misery, is as an
inventory to particularise their abundance; our
sufferance is a gain to them Let us revenge this with
our pikes, ere we become rakes: for the gods know I
speak this in hunger for bread, not in thirst for revenge.
```

Figure 2: sample generation when order = 40

2 Recurrent Neural Network

After training with n-grams, I start to use RNN with GRU and LSTM to train Shakspeare data. The basic idea of LSTM or GRU is that it maintains both long-term dependencies and short-term dependencies by combination of gating and hidden vectors. In experiment, I rnn RNN with hidden size=100, hidden layers = 2, and using GRU as memory network. I choose cross-entropy loss as objective function and use SGD with Adam optimization method to train over 2000 epoches. The learning rate at initialization is 0.005. I use Pytorch packages to make the settings. The training loss is shown in Figure 3. More analysis on RNN will be shown in final project report.

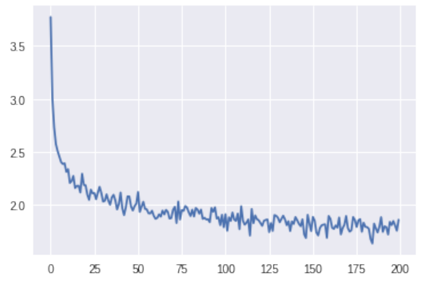


Figure 3: Cross-entropy loss v.s epoches(plot every 10 epoches)

3 Schedule for final project report and presentation

1. Continue to build n-grams model using more advanced smoothing and interpolation technique.
2. Train n-grams model and RNN on structured datasets like Linux Code and Latex. Compare their performance.
3. Discover how hidden layers learn certain features in structured text.

References

[1] Chris Callison-Burch. http://computational-linguistics-class.org/assignment5.html Spring 2018.

[2] Andrej Karphathy. "The Unreasonable Effectiveness of Recurrent Neural Networks" blog. May 21, 2015.

[3] Dan Jurafsky and James H. Martin. Speech and Language Processing (3rd edition draft) .

[4] Yoav Goldberg. "The unreasonable effectiveness of Character-level Language Models (and why RNNs are still cool)" blog

Code

n-grams:

```
# -*- coding: utf-8 -*-
"""ngrams.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1XTK6wOaHW_cuo2ZFYTnIbJJ9YkKyUEbY
"""

from collections import *
def train_char_lm(data, order=5):
    lm = defaultdict(Counter)
    pad = "~" * order
    data = pad + data
    for i in range(len(data)-order):
        history, char = data[i:i+order], data[i+order]
        lm[history][char]+=1
    def normalize(counter):
        s = float(sum(counter.values()))
        return [(c,cnt/s) for c,cnt in counter.items()]
    outlm = {hist:normalize(chars) for hist, chars in lm.items()}
    return outlm

def train_char_lm_smooth(data,order=2,add_k=0.5):
   lm = defaultdict(Counter);
   pad = "~"*order;
   data = pad + data;
   length = len(list(set(data)));
   for i in range(len(data) - order):
      history, char = data[i:i+order], data[i+order]
      lm[history][char] += 1;
   def normalize(counter,length):
      s = float(sum(counter.values()));
      return [(c,(cnt+add_k)/(s + add_k * length)) for c,cnt in counter.items()];
   outlm = {hist:normalize(chars,length) for hist, chars in lm.items()}
   return outlm

!pip install unidecode
import unidecode
import string
file = unidecode.unidecode(open('tiny-shakespeare.txt').read());
lm = train_char_lm_smooth(file, order=1)
print(len(list(set(file))));

from random import random
import math
def generate_letter(lm, history, order):
        history = history[-order:]
        if not history in lm:
           return
        dist = lm[history]
        x = random();
        for c,v in dist:
            x = x - v;
            if x <= 0: return c
        return  dist[math.floor(random()*len(dist))][0];
def  generate_text(lm, order, nletters=1000):
    history = "~" * order;
    out = []
    for i in range(nletters):
        c = generate_letter(lm, history, order)
        history = history[-order:] + c
        out.append(c)
    return "".join(out)

file = unidecode.unidecode(open('shakespeare_sonnets.txt').read());

def perplexity(lm, data, order=1, add_k = 0.5, length = 65):
  N = len(data) - order;
  pad = "~" * order;
  test = pad + data;
  perplex = 1;
  for i in range(N):
    prob = 0;
    history, char = data[i:i+order], data[i+order];
    if not history in lm:
      prob = 1.0 / length;
    else:
     for c, v in lm[history]:
       if c == char:
         prob = v;
         break;
       prob = float(add_k) / length;
    perplex *= (1.0/prob)**(1/N);
  return perplex;

perplex = perplexity(lm, file);
print(perplex);

for i in range(len(lm["fir"])):
        print(lm["fir"][i])

print(generate_text(lm,10));
```

RNN:

```
# -*- coding: utf-8 -*-
"""char-rnn.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1RiRbOYkcgw7wuOYjIvRfHQAeti4k1hAK
"""

import unidecode
import string
import random
import re

all_characters = string.printable
n_characters = len(all_characters)

file = unidecode.unidecode(open('tiny-shakespeare.txt').read())
file_len = len(file)
print('file_len␣=', file_len)

chunk_len = 200

def random_chunk():
    start_index = random.randint(0, file_len - chunk_len)
    end_index = start_index + chunk_len + 1
    return file[start_index:end_index]

print(random_chunk())

!pip install torch
import torch
import torch.nn as nn
from torch.autograd import Variable

class RNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size, n_layers=1):
        super(RNN, self).__init__()
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.n_layers = n_layers

        self.encoder = nn.Embedding(input_size, hidden_size)
        self.gru = nn.GRU(hidden_size, hidden_size, n_layers)
        self.decoder = nn.Linear(hidden_size, output_size)

    def forward(self, input, hidden):
        input = self.encoder(input.view(1, -1))
        output, hidden = self.gru(input.view(1, 1, -1), hidden)
        output = self.decoder(output.view(1, -1))
        return output, hidden

    def init_hidden(self):
        return Variable(torch.zeros(self.n_layers, 1, self.hidden_size))

# Turn string into list of longs
```

```
def char_tensor(string):
    tensor = torch.zeros(len(string)).long()
    for c in range(len(string)):
        tensor[c] = all_characters.index(string[c])
    return Variable(tensor)

print(char_tensor('abcDEF'))

def random_training_set():
    chunk = random_chunk()
    inp = char_tensor(chunk[:-1])
    target = char_tensor(chunk[1:])
    return inp, target

def evaluate(prime_str='A', predict_len=100, temperature=0.8):
    hidden = decoder.init_hidden()
    prime_input = char_tensor(prime_str)
    predicted = prime_str

    # Use priming string to "build up" hidden state
    for p in range(len(prime_str) - 1):
        _, hidden = decoder(prime_input[p], hidden)
    inp = prime_input[-1]

    for p in range(predict_len):
        output, hidden = decoder(inp, hidden)

        # Sample from the network as a multinomial distribution
        output_dist = output.data.view(-1).div(temperature).exp()
        top_i = torch.multinomial(output_dist, 1)[0]

        # Add predicted character to string and use as next input
        predicted_char = all_characters[top_i]
        predicted += predicted_char
        inp = char_tensor(predicted_char)

    return predicted

import time, math

def time_since(since):
    s = time.time() - since
    m = math.floor(s / 60)
    s -= m * 60
    return '%dm %ds' % (m, s)

def train(inp, target):
    hidden = decoder.init_hidden()
    decoder.zero_grad()
    loss = 0

    for c in range(chunk_len):
        output, hidden = decoder(inp[c], hidden)
        loss += criterion(output, target[c].unsqueeze(0))

    loss.backward()
    decoder_optimizer.step()

    return loss.data.item() / chunk_len
```

```python
n_epochs = 2000
print_every = 100
plot_every = 10
hidden_size = 100
n_layers = 2
lr = 0.005

decoder = RNN(n_characters, hidden_size, n_characters, n_layers)
decoder_optimizer = torch.optim.Adam(decoder.parameters(), lr=lr)
criterion = nn.CrossEntropyLoss()

start = time.time()
all_losses = []
loss_avg = 0

for epoch in range(1, n_epochs + 1):
    [inp, target] = random_training_set();
    loss = train(inp, target);
    loss_avg += loss

    if epoch % print_every == 0:
        print('[%s␣(%d␣%d%%)␣%.4f]' % (time_since(start), epoch, epoch / n_epochs *
        print(evaluate('Wh', 100), '\n')

    if epoch % plot_every == 0:
        all_losses.append(loss_avg / plot_every)
        loss_avg = 0

import matplotlib.pyplot as plt
import matplotlib.ticker as ticker
# %matplotlib inline

plt.figure()
plt.plot(all_losses)

print(evaluate('Th', 200, temperature=0.2));

print(evaluate('Th', 1000, temperature=0.8))
```

Dataset

Shakespeare's work: https://cs.stanford.edu/people/karpathy/char-rnn/shakespear.txt Shakespeare's Sonnet and NY Times test set: http://computational-linguistics-class.org/downloads/hw5/test_data.zip