SSIE 500 Section 1, Professor Hiroki Sayama

Validating Shannon’s “Prediction and Entropy of Printed English” in Python’s Natural Language Toolkit (NLTK)

Grady Kurpasi

[gkurpasi@hotmail.com](mailto:gkurpasi@hotmail.com)  
(213) 304-8541

|  |  |  |
| --- | --- | --- |
| A R T I C L E I N F O  7 May 2019  Keywords:  Shannon Entropy  Python  Natural Language Toolki |  | A B S T R A C T  Shannon’s 1950 paper (cited here from publication in 1951) provided a means for describing the Entropy of the character content of printed English and the effectiveness of prediction models based on analysis of N-Grams of preceding text. Because of the computational and linguistic tools at his disposal, some of Shannon’s calculations were partially based on statistical models of English Language. This paper employs The Natural Language Toolkit and its full text corpora to validate Shannon’s models and predictions. |

1. Introduction

Shannon’s “Prediction and Entropy of Printed English (Shannon 1950) provided statistical / computational means of predicting the Entropy of printed English. It further presented a model for predicting characters based on a N-gram of the N preceding characters.

* 1. Shannon Prediction and Entropy

In the paper, Shannon describes both Entropy, the information produced on average for each character and Redundancy the amount of constraint imposed on a text due to the statistical distribution of its characters (Shannon 1950).

His predictions follow his description of the Entropy of a sequence as the Expected value of the information content of component sequences:

He extends this to consider N-grams: sequences of characters employed as statistical indicators of a following character. He cites that written languages natural have statistical tendencies (stemming from underlying phonemic / graphemic properties of the language) for certain characters to predict / constrain the occurrence of following characters. In full written language this becomes increasingly predicted as character sequences combine to form word and sentence level sequences.

He describes N-gram entropy as:

is a block of N-1 letters [(N-1) gram] that is used to predict a following letter.

is an arbitraty letter following (and predicted by)

is the probability of the N-gram

is the conditional probability of following block and equals (Shannon 1950).

Shannon describes this equation for N-gram entropy as “measuring the average uncertainty (conditional entropy) of the next letter *j* when the preceding N-1 letters are known” (Shannon 1950).

Therefore *F*1 represents the expected entropy of characters in a language. *F*2 is the ability of any one character to predict the next. *F*3 is the ability of two characters to predict the next, etc.

As N increases to ∞, entropy *H* equals:

(Shannon 1950) Shannon provides estimates for small values of N, but his calculations in 1950 were based on existing bigram and trigram tables of English character frequencies. He points out that these do not consider bigrams and trigrams that span word boundaries in actual text, and therefore approximate independence of terminal characters of one word and initial characters of the following, which at the word and sentence level is not necessarily the case.

He gave the following values for *F*1-*F*3:

*F*1 = 4.14 bits per letter

*F*2 = 3.56 bits per letter

*F*3 = 3.30 bits per letter

*F*1 was calculated as:

(spaces and punctuation were not considered).

For *F*2, *F*3 calculations were made using digram/trigram approximations:

No values for *F*N > 3 were given.

* 1. Reproducing Shannon

In this paper, we attempt to reproduce Shannon’s results using modern tools including Python 3.7.3 and the Natural Language Toolkit (NLTK) 3.4.1.

Python was selected for its extensive and mature array handling and statistical processing libraries, as well as for the Natural Language Toolkit.

The NLTK library provides access to a range of computational linguistic capabilities in a fairly mature, well documented toolkit. It was selected in particular for exposing a number of full-text corpora which are used here to conduct Shannon’s calculations with cross-word-boundary digrams and trigrams.

1. Methodology

The first step to approaching Shannon’s calculations were to obtain the raw text from one (or more of) the NLTK Corpora.

The Gutenberg Project Corpora and the Brown Corpora were chosen. The Gutenberg Project is a digital archiving initiative that collects electronic versions of text. NLTK exposes a subset of its texts via a corpus module. The Brown Corpora of American English is a ‘tagged’ corpus, meaning each of its roughly 1 million words has been tagged with syntactic meta-information. The Brown Corpora was collected from wide ranging sources and is one of the most frequently employed resources in linguistics research. Unfortunately, this study was not able to compute *F*N values for the Brown Corpus due to computational limitations.

Computational processing time was found to be an issue in processing corpora. Unfortunately, time did not allow a computational complexity analysis of the algorithms, but this is inevitably necessary to continue to employ and expand the code developed for this project.

Due to computational shortfalls, only Herman Melville’s Moby Dick (1851) from the Gutenberg corpus was selected for research. The full text of Moby Dick is about 1.2 million characters long. With punctuation stripped out it is about 950K characters long. It was originally envisioned that native python lists and dictionaries would suffice to store and process these strings. Although they worked well in testing, they proved prohibitively resource intensive when processing the full text.

Code was re-worked to utilize Numpy ndarrays which proved orders of magnitude faster in stripping non-alphabetic characters from the text.

To do this the full text was converted to a numpy array of strings with length (1). (Numpy does not have an explicit char array any longer). A boolean mask array was created by testing inclusion of all characters to an alphabet-only-array, then Numpy.delete was called on the raw character array to create the alphbet-only-text-array. This adjusted array was stored to disk to shorten following processing times.

This array was then fed to a modified Hseq function based on the Hseq function developed in SSIE 500 Spring 19 class. The modified Hseq takes an array of 1 length strings and an N value. It iterates through the array of strings building an array of information events of length 1. Once the event array is built np.unique() is used to identify unique information events (characters, or character sequences) as well as their frequency and finally distribution.

Once the frequency distribution is obtained, array scalar calculations are used to generate the H(X) value for the text with N= to (1, 2 or 3).

1. Results

As discussed, a few iterations of code were required to get this thing to run in a timely manner. The N=1 run took a little over 10 hours on a Surface Book 2 laptop. The N=2 run took about 6 hours on a gaming desktop, the N=3 run was left to run overnight on the desktop and not timed.

Memory did not appear to be a performance constraint; CPU utilization appeared to be the bottleneck. There is at least one function that concatenates strings in an array that is estimated to be particularly inefficient.

It was discovered during analysis that a line of code was missed, and that strings were not forced to a single case, meaning that essentially, a 52 letter alphabet was employed. Unfortunately, time did not allow fixing this error and re-running the experiment.

Despite this, results were still very close to Shannon’s

*F*1 = 4.35 bits per letter

*F*2 = 3.72 bits per letter

*F*3 = 3.23 bits per letter

The similarity of this accidental result to Shannon’s original results is perhaps due to the relative infrequency of capital letters in the text compared to lowercase overshadowing the differences in distribution of letters in word initial positions vice word internal.

1. Conclusions

Result are inconclusive until re-run with a 26 letter alphabet. However further investigation is warranted to see if H(X) is more sensitive to alphabet size or length of N-grams (N).

1. References

Shannon, Claude E (1951) Prediction and Entropy of Printed English. *The* *Bell System Technical Journal* (January 1951) 50-64

Mellville, Herman (1851) Moby Dick

1. Python Source Code

import nltk

import numpy as np

# NOTE: had to use ndarrays for this project. Organic Python lists were orders of magnitude slower when processing full text of corpora

import pickle

# Creates char arrays for Herman Melville's Moby Dick

# Creates (2) char arrays.

# alph27arr includes alpha characters and space - all numeric and punctuation characters stripped

# alph26arr includes alpha characters only

ALPH26 = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ"

ALPH27 = " abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ"

ALPH26ARR = np.array(list(ALPH26))

ALPH27ARR = np.array(list(ALPH27))

ALPH26FILE = 'alph26.txt'

ALPH27FILE = 'alph27.txt'

print("")

print("")

print("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*START\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

def StrArrayToStr(strarr=np.empty(0)):

#This function is not efficient needs to be rewritten to use Numpy efficiencies

newstring = ""

for i in range(len(strarr)):

newstring = newstring + strarr[i]

return newstring

def HStrArray(hstrarr, N=1):

#calculates expected information for a sequence

#Expects array of strings length 1 (i.e. char array)

#init information events

events= np.empty(0)

#get all events of length N (starting at 0, 1, 2, ... Total Length - N)

print(len(hstrarr))

for i in range(len(hstrarr)-N):

if i % 1000 == 0 : print(i)

events = np.append(events, StrArrayToStr(hstrarr[i:i+N]))

#get unique events and counts of occurrence

unique, counts = np.unique(events, return\_counts=True)

print("DEBUG HStrArray Unique Counts ", dict(zip(unique, counts)))

#calculate frequency distribution of all events

dist = counts / len(hstrarr )

#print (dist)

#return Execpected Information

return -sum(dist \* np.log2(dist))

def MakeCharArrays(corpus):

# Get raw text of corpora

if corpus == 'brown':

raw = nltk.corpus.brown.raw()

elif corpus == 'melville': # Get raw text of Moby Dick

raw = nltk.corpus.gutenberg.raw("melville-moby\_dick.txt")

elif corpus == 'gutenberg':

raw = nltk.corpus.gutenberg.raw()

else:

return False

# Convert raw text string to Numpy ndarray

chararr = np.array(list((raw)))

# Create boolean mask for alphabet + space characters in character array

mask = np.isin(chararr, ALPH27ARR)

# Remove non-alphabet / space characters

nospecial = chararr[mask]

# Write adjusted text to disk as char array

with open(corpus + ALPH27FILE, 'wb') as filehandle:

pickle.dump(nospecial, filehandle)

# Create boolean mask for alphabet characters in character array

mask = np.isin(nospecial, ALPH26ARR)

#Remove non-alphabet charcters

nospace = nospecial[mask]

# Write adjusted text to disk as char array

with open(corpus + ALPH26FILE, 'wb') as filehandle:

pickle.dump(nospace, filehandle)

# Code to read alph2Xfile

# with open(alph27file, 'rb') as filehandle:

# newlist = pickle.load(filehandle)

return True

#MakeCharArrays('brown')

# raw = nltk.corpus.brown

# Code to read alph2Xfile

with open('melville' + ALPH26FILE, 'rb') as filehandle:

newlist = pickle.load(filehandle)

#print(newlist.tostring())

mylist = newlist[:20]

#The following line was run on three times on different systems w parameter

# N = 1, 2, 3

print (HStrArray(newlist, 3))

print("Done")