CSE6339 – 2nd Assignment

Yasser Gonzalez-Fernandez – ygf@yorku.ca, ygonzalezfernandez@gmail.com

March, 2014

***Abstract*—Solutions to the second assignment of the course CSE6339 3.0 Introduction to Computational Linguistics. Instructor: Prof. Nick Cercone, Department of Electrical Engineering & Computer Science, York University, Canada.**

I. Introduction

The motivation for the assignment is illustrated in the following statement. "If enough monkeys were allowed to pound away at typewriters for enough time, all the great works of literature would result" [1]. The typist monkeys in question represent a device – in the assignment, a computer program – capable of generating a random sequence of characters.

Computer simulation programs of different complexity are used, ranging from zero-order monkeys – that generate all characters with the same probability – to programs using statistics of order greater than zero computed from writing samples. The output of the programs is compared to the content of the literary work they are trying to reproduce in order to evaluate their performance. The statistics computed from the writing samples are also used to perform other tasks such as author attribution and genre classification.

The remainder of the report is organized as follows. Firstly, Section II describes the implementation of the programs created for the assignment. Then, Section III continues with the actual solutions to the nine problems. Lastly, Section IV summarizes the results and provides some concluding remarks.

II. Program Documentation

The solutions to the problems in the assignment were implemented as a collection of Python scripts. Each individual script uses a group of common functionalities contained in a main module `monkeys.py`. In the code, external dependencies were kept at minimum by relying only on the Python Standard Library. The rest of this section describes the most important data structures and algorithms used in `monkeys.py`. The complete source code of the module is included in an appendix at the end of the report.

A. Preprocessing

The distinction between upper and lower case characters is ignored in the assignment. A total number of 40 different characters is considered: 26 alphabetic characters folded to lower case (`a`, `b`, `c`, ..., `z`), 13 punctuation marks (space, `,`, `.`, `;`, `:`, `?`, `!`, `(`, `)`, `-`, `'`, `"`, `@`), and any digit (`0`–`9`) is counted as `#`. The `map_input_char` function translates any input character in a writing sample to the characters considered in the assignment. A character that do not appear among the 40 characters is mapped to `@` (in addition to `@` itself).

Additionally, an integer between 0 and 39 is assigned to each one of the 40 characters following the ASCII order – i.e. space gets 0, `!` gets 1, `"` gets 2, ..., and `z` gets 39. The functions `char2index` and `index2char` return the index corresponding to a character and vice versa. These indexes identify the entries of the frequency tables presented next in this section.

B. N-Gram Frequency Tables

Solving the problems in the assignment requires counting the occurrences of all the character n-grams appearing in a writing sample. This section discusses how this information can be represented and the algorithms that work with this representation.

1) Representation: The most important data structure in the assignment is the n-gram frequency table. A frequency table T contains the frequency counts of every combination of n characters over the alphabet of $m = 40$ possible characters. A particular entry $T_{c_1,\ldots,c_n}$ in the table can be identified by the indexes $c_1, \ldots, c_n \in [0, m-1]$ of the n-gram characters. Also, if we follow the analogy of monkeys at typewriters, the entries $T_{c_1,\ldots,c_{n-1},k}, k = 0, \ldots, m-1$ determine the layout of the typewriter corresponding to the n-gram prefix $c_1, \ldots, c_{n-1}$.

A table T is represented as a Python nested list with n levels and m entries per level. This data structure is equivalent to a multidimensional $n \times m$ array, since Python lists are implemented as arrays of pointers in CPython. This array-based representation guarantees O(1) read/write access to any particular entry in the table. Based on this structure, the function `compute_freq_tab` builds a frequency table of the specified order from a writing sample given as a group of plain-text files.

One drawback of the array-based representation is that the table may contain many zero entries corresponding to n-grams that never appeared on the writing sample. A more memory-efficient implementation could have been based on a hash table (i.e. a Python dictionary) that stores only the n-grams with nonzero frequencies. Nevertheless, since only a maximum of third-order n-grams are considered in the assignment, the explicit array-based representation was preferred. The more compact representation is however used to save the frequency tables to files on disk. The function `write_freq_tab` saves a mapping of the n-gram with nonzero frequencies as a JSON dictionary. Consequently, the function `read_freq_tab` reads a frequency table from a file in JSON format.

2) Most Probable Path: Problem 1f requires using the procedure to compute the most probable path in a second-order frequency table described in the handout [1]. The function `most_probable_freq_tab_path` implements a simple generalization of this algorithm for n-grams with $n \geq 2$. Given

a frequency table of order n and a prefix of $n-1$ characters, the path is built incrementally by adding the character that completes the n-gram with the highest frequency at the end of the path. The algorithm stops when all characters are already included in the path or an n-gram prefix with zero occurrences is found.

3) Reducing the Resolution: For the solution of Problem 1d, it is necessary a method to change the resolution of the frequency tables. The function `reduce_freq_tab_resolution` achieves this purpose by reducing the number of keys on the typewriters. If $F > 0$ is the maximum frequency in a table T of order n, the entries of the reduced frequency table T' are obtained as follows,

$$T'_{c_1,\ldots,c_n} = \left\lfloor \frac{T_{c_1,\ldots,c_n}}{\alpha F} \right\rfloor,$$

where $c_1, \ldots, c_n$ identify an entry in the table and $\alpha \in (0, 1]$ is the reduction rate – i.e. all entries are normalized by a fraction of the maximum frequency. This method tries to preserve the relative proportions between frequencies that remain greater than zero after the transformation.

4) Simulation: The function `simulate_freq_tab` samples a number of characters following the distribution encoded in the frequency table. For a frequency table of order n, the first $n-1$ characters are always sampled uniformly. Also, a character is sampled uniformly if all the entries in the frequency table for the current n-gram prefix have zero frequencies in order to always satisfy the required number of characters. The resulting string of characters is written to a plain-text output file.

The auxiliary function `_freq_tab_sample` is called from `simulate_freq_tab` and it is responsible for sampling a single character given an n-gram prefix of $n-1$ characters. The code in `_freq_tab_sample` first computes the list of cumulative frequencies for the entries corresponding to the n-gram prefix. Then, an integer between zero and the cumulative sum of frequencies is sampled and its position in the cumulative list of frequencies determines the character to be returned. Since the position of the can be determined in $\mathrm{O}(\log m)$ using a bisection algorithm, the running-time for sampling a single character is $\mathrm{O}(m)$ – i.e. it is dominated by the computation of the cumulative frequencies.

C. Evaluating the Output

Some of the problems in the assignment require comparing the performance of different simulation programs. Since the purpose is to reproduce a literary work, the content of a file generated by `simulate_freq_tab` is compared to the text of the book to be reproduced. This comparison is performed in terms an evaluation measure calculated by the `relative_word_yield` function. The relative word yield is defined as the number correct words in the simulated file divided by the total number of words in the simulated file – a word is considered correct if it appears in the book to be reproduced. In order to get a reasonable estimation of the performance measure, the simulated files contain ten times the number of characters in the book to be reproduced.

D. Writing Sample Profiles

The solutions to Problems 1g, 1h, and 1i use a technique called Common N-Gram (CNG) profiles [2] to discover similarities between authors, and to perform author attribution and genre classification. The function `cng_profile` builds a CNG profile of the specified length L from a frequency table. A Python dictionary is used to represent the profile, as a mapping of the L most-frequent n-grams to their normalized frequencies. Also, the function `cng_dissimilarity` computes a positive dissimilarity measure between two profiles. The function returns zero for texts with identical L most-frequent n-grams. For more information about the CNG profiles, please refer to the citation above.

III. Solutions

This section is divided into nine subsections, each one corresponding to one problem in the assignment. Table I shows the data made available for the assignment.

TABLE I
DATA MADE AVAILABLE FOR THE ASSIGNMENT.

No.	Author	Title	Chars.
1	C. Dickens	A Christmas Carol	168,925
2	C. Dickens	A Tale of Two Cities	773,928
3	E. Bronte	Wuthering Heights	662,869
4	A. Bronte	Agnes Grey	384,300
5	C. Bronte	Jane Eyre	1,051,336
6	E. R. Burroughs	Tarzan of the Apes	492,783
7	E. R. Burroughs	Warlord of Mars	319,854
8	E. R. Burroughs	The People that Time Forgot	212,736
9	E. R. Burroughs	The Land that Time Forgot	205,762
10	H. R. Haggard	King Solomon's Mines	454,437
11	J. Cleland	Fanny Hill	474,725
12	L. Carroll	Alice's Adventures in Wonderland	148,580
13	L. Carroll	Through the Looking Glass	167,015
14	W. Irving	Legend of Sleepy Hollow	69,164
15	Sir A. C. Doyle	The Adventures of Sherlock Holmes	575,574
16	Sir A. C. Doyle	The Lost World	432,766
17	Sir A. C. Doyle	The Hound of the Baskervilles	327,348
18	Sir A. C. Doyle	Tales of Terror and Mystery	421,264
19	M. Twain	Adventures of Huckleberry Finn	578,345
20	M. Twain	The Adventures of Tom Sawyer	397,076
21	M. Twain	A Connecticut Yankee in King Arthur's Court	657,430
22	N. Machiavelli	The Prince	286,854
23	H. G. Wells	War of the Worlds	346,458
24	H. G. Wells	The Time Machine	182,881
25	F. Kafka	Metamorphosis	122,139
26	F. Kafka	The Trial	462,372
27	R. Kipling	The Jungle Book	279,771

A. Problem 1a

> "Simulate the straightforward monkey problem. Let the program run long enough to give a meaningful estimate of the yield of words. The result will provide a useful comparison with later forms of the problem."

The simulation of the straightforward monkey problem is performed by calling `simulate_freq_tab` with `None` as the

argument corresponding to the frequency table. A subset of the books listed in Table I was selected to evaluate the performance of the simulation program (the first book by each author). The following Python code runs the simulation and computes the relative word yield for every book.

```
from monkeys import simulate_freq_tab, relative_word_yield

corpus = (
    ('books/christmas_carol.txt', 168925),
    ('books/wuthering_heights.txt', 662869),
    ('books/agnes_grey.txt', 384300),
    ('books/jane_eyre.txt', 1051336),
    ('books/tarzan_of_the_apes.txt', 492783),
    ('books/king_solomons_mines.txt', 454437),
    ('books/fanny_hill.txt', 474725),
    ('books/alices_adventures_in_wonderland.txt', 148580),
    ('books/legend_of_sleepy_hollow.txt', 69164),
    ('books/the_adventures_of_sherlock_holmes.txt', 575574),
    ('books/adventures_of_huckleberry_finn.txt', 578345),
    ('books/the_prince.txt', 286854),
    ('books/war_of_the_worlds.txt', 346458),
    ('books/metamorphosis.txt', 122139),
    ('books/the_jungle_book.txt', 279771)
)

for book_path, book_chars in corpus:
    simulate_freq_tab(None, 10 * book_chars, 'tmp.txt')
    word_yield = relative_word_yield('tmp.txt', book_path)
    print '%s,%s' % (book_path, word_yield)
```

Table II summarizes the results of the (zero-order) straightforward monkey problem. The results are surprisingly high for characters being generated with the same probability – values over 25% were obtained for seven of the books. It seems that common words such as short articles are generated very frequently. These results are considered as the baseline for the simulation algorithms implemented in the following problems.

TABLE II
RELATIVE WORD YIELD OF THE STRAIGHTFORWARD MONKEY PROBLEM.

Book No.	Avg. Rate	Book No.	Avg. Rate
1	14.97%	14	7.88%
3	26.55%	15	31.49%
4	19.51%	19	34.18%
5	31.02%	22	26.17%
6	26.28%	23	14.91%
10	25.13%	25	11.87%
11	18.81%	27	13.91%
12	17.71%		

B. Problem 1b

> "Use the data in Table III to simulate the first-order monkey problem. Again let the program run long enough to give a meaningful estimate of the yield of words to permit comparison with other results on relative word yield. Try running this simulation program against other corpora."

The first-order frequency table given in Table III was typed into a file in JSON format. This file is loaded using the `read_freq_tab` function and then the simulation is performed by calling `simulate_freq_tab`. The same subset of the books from Problem 1a is used in order to make the results comparable. The following Python code runs the simulation and computes the relative word yield for every book.

TABLE III
CHARACTER DISTRIBUTION FROM ACT III OF HAMLET.
NOTE: 35,224 CHARACTERS, A SMALL CORPUS.

Char.	Freq.	Char.	Freq.	Char.	Freq.
space	6,934	r	1,593	p	433
e	3,277	l	1,238	b	410
o	2,578	d	1,099	v	309
t	2,557	u	1,014	k	255
a	2,043	m	889	'	203
s	1,856	y	783	j	34
h	1,773	f	629	q	27
n	1,741	c	584	x	21
i	1,736	g	478	z	14

```
from monkeys import (read_freq_tab, simulate_freq_tab,
                     relative_word_yield)

corpus = (
    ('books/christmas_carol.txt', 168925),
    ('books/wuthering_heights.txt', 662869),
    ('books/agnes_grey.txt', 384300),
    ('books/jane_eyre.txt', 1051336),
    ('books/tarzan_of_the_apes.txt', 492783),
    ('books/king_solomons_mines.txt', 454437),
    ('books/fanny_hill.txt', 474725),
    ('books/alices_adventures_in_wonderland.txt', 148580),
    ('books/legend_of_sleepy_hollow.txt', 69164),
    ('books/the_adventures_of_sherlock_holmes.txt', 575574),
    ('books/adventures_of_huckleberry_finn.txt', 578345),
    ('books/the_prince.txt', 286854),
    ('books/war_of_the_worlds.txt', 346458),
    ('books/metamorphosis.txt', 122139),
    ('books/the_jungle_book.txt', 279771)
)

freq_tab = read_freq_tab('act_iii_hamlet_1st_order.json')
for book_path, book_chars in corpus:
    simulate_freq_tab(freq_tab, 10 * book_chars, 'tmp.txt')
    word_yield = relative_word_yield('tmp.txt', book_path)
    print '%s,%s' % (book_path, word_yield)
```

Table IV summarizes the results of the first-order monkey problem. In general, the performance is similar to the straightforward monkey problem. There are improvements on some of the books, but e.g. the results on books where the straightforward monkey simulation achieved values over 30% are worst in this case.

TABLE IV
RELATIVE WORD YIELD OF THE FIRST-ORDER MONKEY PROBLEM WITH THE CHARACTER FREQUENCIES IN TABLE III.

Book No.	Avg. Rate	Book No.	Avg. Rate
1	15.01%	14	7.89%
3	22.35%	15	23.46%
4	15.31%	19	24.94%
5	23.50%	22	19.50%
6	21.01%	23	14.68%
10	21.89%	25	12.37%
11	18.67%	27	17.40%
12	15.51%		

C. Problem 1c

> "Use the data supplied, data listed in Table I, to simulate the second-order and third-order Bronte

monkey problem. Again let the program run long enough to give a meaningful estimate of the yield of words to permit comparison with other results on relative word yield. Try running this simulation program against other authors listed."

The three books written by the Bronte sisters listed in Table I (books 3, 4, and 5) are used together as the writing sample to build second and third order frequency tables. The following Python code runs the simulation and computes the relative word yield for the books in every case.

```
from monkeys import (compute_freq_tab, simulate_freq_tab,
                     relative_word_yield)

corpus = (
    ('books/christmas_carol.txt', 168925),
    ('books/wuthering_heights.txt', 662869),
    ('books/agnes_grey.txt', 384300),
    ('books/jane_eyre.txt', 1051336),
    ('books/tarzan_of_the_apes.txt', 492783),
    ('books/king_solomons_mines.txt', 454437),
    ('books/fanny_hill.txt', 474725),
    ('books/alices_adventures_in_wonderland.txt', 148580),
    ('books/legend_of_sleepy_hollow.txt', 69164),
    ('books/the_adventures_of_sherlock_holmes.txt', 575574),
    ('books/adventures_of_huckleberry_finn.txt', 578345),
    ('books/the_prince.txt', 286854),
    ('books/war_of_the_worlds.txt', 346458),
    ('books/metamorphosis.txt', 122139),
    ('books/the_jungle_book.txt', 279771)
)

bronte_books = (
    'books/wuthering_heights.txt',
    'books/agnes_grey.txt',
    'books/jane_eyre.txt'
)

for order in (2, 3):
    freq_tab = compute_freq_tab(order, *bronte_books)
    for book_path, book_chars in corpus:
        simulate_freq_tab(freq_tab, 10 * book_chars, 'tmp.txt')
        word_yield = relative_word_yield('tmp.txt', book_path)
        print '%s,%s,%s' % (book_path, order, word_yield)
```

Table V summarizes the results with the second and third order Bronte monkey problem. In this case the results are considerably better than previous results on every book, which illustrates the power of the higher-order statistics. The results with the third-order statistics are also consistently better than with the second-order statistics. The highest relative word yield is obtained for one of the Bronte books with 45.97%.

TABLE V
RELATIVE WORD YIELD OF THE SECOND-ORDER (2ND) AND THIRD-ORDER (3RD) BRONTE MONKEY PROBLEM.

Book No.	Avg. Rate		Book No.	Avg. Rate	
	2nd	3rd		2nd	3rd
1	24.78%	40.03%	14	18.57%	37.56%
3	32.43%	44.57%	15	30.65%	41.72%
4	27.70%	41.87%	19	35.89%	45.42%
5	35.38%	45.97%	22	28.89%	40.63%
6	29.42%	41.20%	23	26.02%	40.65%
10	30.33%	42.09%	25	23.02%	38.12%
11	26.16%	41.41%	27	26.94%	41.13%
12	24.93%	38.88%			

D. Problem 1d

"Investigate the effects of resolution on monkey literacy in the simulation. For example round off the matrix elements to the smallest number of places – or use an equivalent means to reduce the number of keys on the typewriters."

A book of medium size (book 4) was selected among the books listed in Table I to investigate the effect of resolution on the relative word yield. The study is performed for 11 equally spaced values of the reduction rate α in $[0, 1]$ – the argument of `reduce_freq_tab_resolution`. The effect is analyzed on first, second, and third order frequency tables. The following Python code collects the necessary data.

```
from monkeys import (compute_freq_tab, reduce_freq_tab_resolution,
                     simulate_freq_tab, relative_word_yield)

book_path, book_chars = 'books/agnes_grey.txt', 384300

for order in (1, 2, 3):
    orig_freq_tab = compute_freq_tab(order, book_path)
    for rate in (0.1 * k for k in xrange(11)):
        freq_tab = reduce_freq_tab_resolution(orig_freq_tab, rate)
        simulate_freq_tab(freq_tab, 10 * book_chars, 'tmp.txt')
        word_yield = relative_word_yield('tmp.txt', book_path)
        print '%s,%s,%s' % (order, rate, word_yield)
```

The output of the Python script was processed using R in order to generate the plot on Fig. 1. On the first and second order frequency tables, reducing the resolution gradually increases the relative word yield up to around $\alpha = 0.5$ and then it decays. On the other hand, the reduction of the resolution decreases the performance on the third-order frequency table for every value of α. The integer division by a fraction of

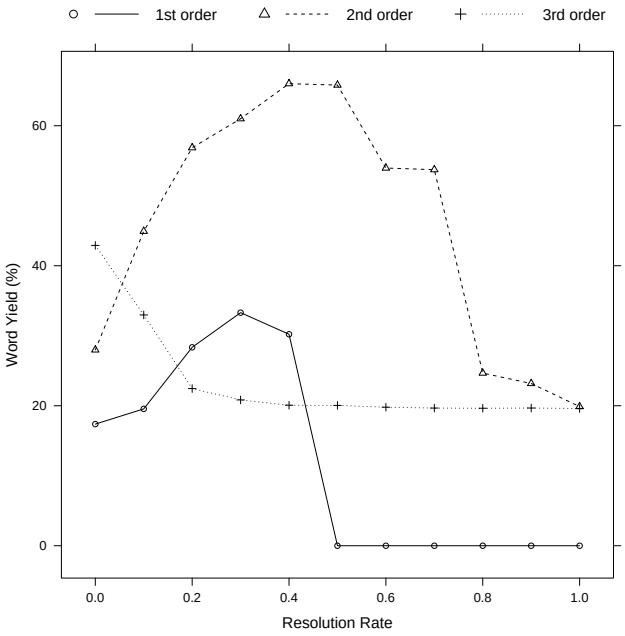

Fig. 1. Effect of resolution on monkey literacy.

the maximum frequency zeroes entries with low frequencies that produce "noise" on the generated output. Therefore, the accuracy of the simulation increases – possibly at the expense of less variability in the output. Because of the greater number of entries on the third-order frequency tables, the frequencies seem to be more evenly distributed and the reduction affects also the generation of correct words.

E. Problem 1e

> "Write a routine to compute correlation matrices of the type shown in the handout [1] from data supplied (the books shown in Table I)."

The same book number 4 used in the previous problem is used here. The following Python code computes second and third order frequency tables from the book and save them to a file in JSON format.

```
from monkeys import compute_freq_tab, write_freq_tab

freq_tab_2nd_order = compute_freq_tab(2, 'books/agnes_grey.txt')
write_freq_tab(freq_tab_2nd_order, 'agnes_grey_2nd_order.json')

freq_tab_3rd_order = compute_freq_tab(3, 'books/agnes_grey.txt')
write_freq_tab(freq_tab_3rd_order, 'agnes_grey_3rd_order.json')
```

The output is processed using R to generate a graphical representation of the normalized frequency tables similar to the ones showed in the handout. Fig. 2 shows the second-order correlation matrix. Note the high frequency of the entires `th`, `he`, and `e` corresponding to the article `the`. Fig. 3 shows three selected entries of the third-order correlation matrix. The high frequency of the n-gram `the` is also evident.

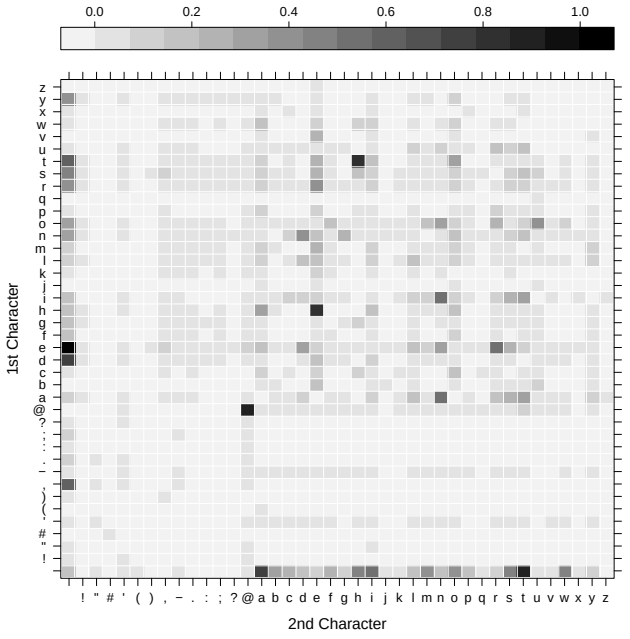

Fig. 2. Second-order correlation matrix built from A. Bronte's "Agnes Grey".

F. Problem 1f

> "Using the algorithm in the handout [1] with the pair-correlation matrix generated from Irving (book shown in Table I), compute the most probable digraph path which starts with the letter `t`. Compare the result with that given in the handout for Poe's 'The Gold Bug"'.

The most probable digraph path starting with the letter `t` is computed not only for W. Irving's "Legend of Sleepy Hollow" but also for the same subset of books used on Problem 1a. In addition, third-order frequency tables are built from the books and the most probable trigraph paths starting with the letters `th` are presented. The following Python code computes the digraph and trigraph paths from the books.

```
from monkeys import compute_freq_tab, most_probable_freq_tab_path

corpus = (
    'books/christmas_carol.txt',
    'books/wuthering_heights.txt',
    'books/agnes_grey.txt',
    'books/jane_eyre.txt',
    'books/tarzan_of_the_apes.txt',
    'books/king_solomons_mines.txt',
    'books/fanny_hill.txt',
    'books/alices_adventures_in_wonderland.txt',
    'books/legend_of_sleepy_hollow.txt',
    'books/the_adventures_of_sherlock_holmes.txt',
    'books/adventures_of_huckleberry_finn.txt',
    'books/the_prince.txt',
    'books/war_of_the_worlds.txt',
    'books/metamorphosis.txt',
    'books/the_jungle_book.txt',
)

for book_path in corpus:
    freq_tab = compute_freq_tab(2, book_path)
    digraph_path = most_probable_freq_tab_path(freq_tab, 't')
    freq_tab = compute_freq_tab(3, book_path)
    trigraph_path = most_probable_freq_tab_path(freq_tab, 'th')
    print book_path, digraph_path, trigraph_path
```

Table VI shows the digraphs and trigraph paths. The most probable digraph in W. Irving's "Legend of Sleepy Hollow" (book 14) is `the andis,@wofry."bulmpk!-cq` and the trigraph path is `the somplack,@`. The most probable digraph path in Poe's "The Gold Bug" given in the handout is `the andisouryplf'bj`. All these paths begin with the word `the` followed by a space, which is also true for the rest of the paths. All the digraph paths continue with `and` after the space, while some trigraph paths include words such as `said` and `was`. Common words such as `is`, `our`, `or`, and `of` are also visible.

G. Problem 1g [1]

> "Design and implement an experiment using data from the books shown in Table I that might be used to perform author attribution. Discuss your solution and provide reasons why it is likely or not likely to solve the problem definitively."

The authors with more than one book listed in Table I – i.e. C. Dickens, E. R. Burroughs, L. Carroll, Sir A. C. Doyle,

[1] The initial implementation of `cng_dissimilarity` was incorrect (common n-grams were counted twice), so the numerical results presented in this section are not valid.

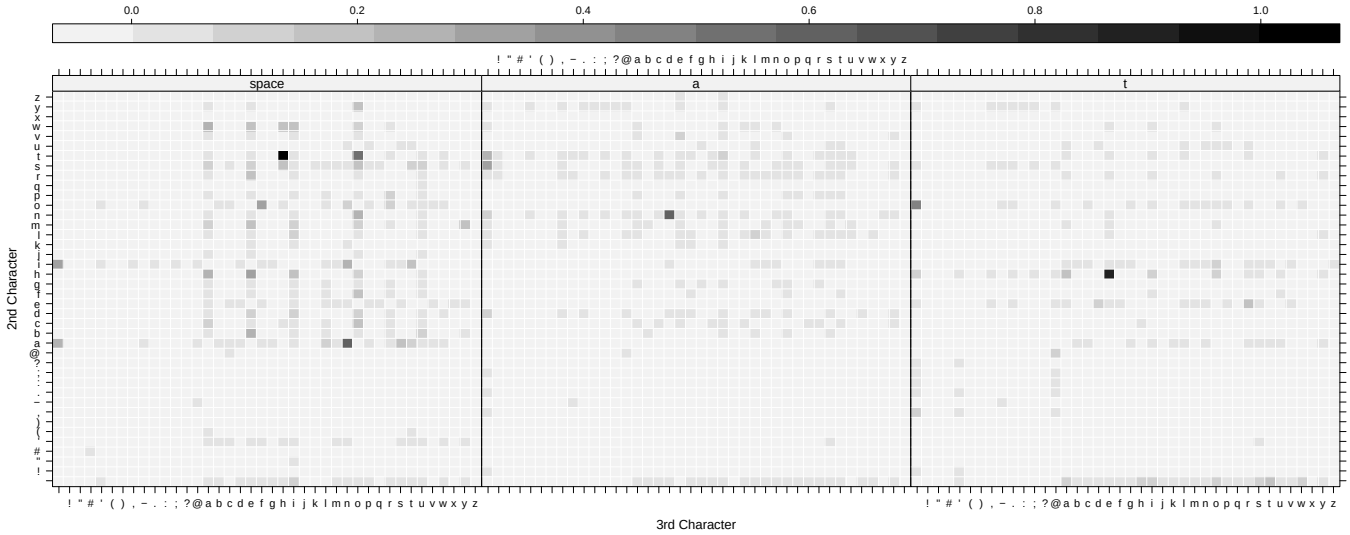

Fig. 3. Selected entries of the third-order correlation matrix built from A. Bronte's "Agnes Grey".

TABLE VI
MOST PROBABLE DIGRAPH (STARTING WITH T) AND TRIGRAPH (STARTING WITH TH) PATHS.

Book No.	Most Probable	
	Digraph Path	Trigraph Path
1	the andouscrimy,"@w.'lf-bj	the said,"@
3	the andisour,'@cly."w-bj	the and@,
4	the andisoury,@'w."blf-ck;	the and@
5	the andisoury,@"w.'lf-bjp;	the and@."-somplik
6	the andisorzly,"@w.'mpug-f?)	the saingly,@
10	the and,"@siloury.'ck-bj	the sompard,@#
11	the andis,@wofry."bluck;	the of@
12	the andoury,'@sicklf.)-bj	the wasking,'@
14	the andis,@wofry."bulmpk!-cq	the somplack,@
15	the andouris,"@w.'ckly-bj	the and@
19	the andoulis,@"w.'mybrkf-g;	the was@
22	the andis,@wory."culf-#;(?)	the and@
23	the andisofry,@wlup."g-mbj	the somplack,@
25	the andouly,@simpr'v	the was@
27	the andoulis,"@bry.)-ck!'mp?	the was@,

M. Twain, H. G. Wells, and F. Kafka – were selected for the experiment and divided into two training/testing groups. Third-order frequency tables were built for each author using all the books except the first one – e.g. M. Twain's frequency table was built using "The Adventures of Tom Sawyer" and "A Connecticut Yankee in King Arthur's Court". Next, CNG profiles of three different lengths $L \in \{10, 100, 500\}$ were constructed for each author. Finally, the first book by each author – e.g. M. Twain's "Adventures of Huckleberry Finn" – was considered for author prediction. The following Python code implements this experimental procedure.

```
from monkeys import compute_freq_tab, cng_profile, cng_dissimilarity

training_corpus = {
    'C. Dickens': ('books/tale_of_2_cities.txt', ),
    'E. R. Burroughs': (
        'books/warlord_of_mars.txt',
        'books/the_people_that_time_forgot.txt',
        'books/the_land_that_time_forgot.txt'),
    'L. Carroll': ('books/through_the_looking_glass.txt', ),
    'Sir A. C. Doyle': (
        'books/the_lost_world.txt',
        'books/the_hound_of_the_baskervilles.txt',
        'books/tales_of_terror_and_mystery.txt'),
    'M. Twain': (
         'books/the_adventures_of_tom_sawyer.txt',
         'books/a_connecticut_yankee_in_king_arthur_s_court.txt'),
    'H. G. Wells': ('books/the_time_machine.txt', ),
    'F. Kafka': ('books/the_trial.txt', )
}

testing_corpus = {
    'C. Dickens': 'books/christmas_carol.txt',
    'E. R. Burroughs': 'books/tarzan_of_the_apes.txt',
    'L. Carroll': 'books/alices_adventures_in_wonderland.txt',
    'Sir A. C. Doyle': 'books/the_adventures_of_sherlock_holmes.txt',
    'M. Twain': 'books/adventures_of_huckleberry_finn.txt',
    'H. G. Wells': 'books/war_of_the_worlds.txt',
    'F. Kafka': 'books/metamorphosis.txt'
}

for profile_len in (10, 100, 500):
    # Build the author's profile database.
    profiles = {}
    for author, book_paths in training_corpus.iteritems():
        freq_tab = compute_freq_tab(3, *book_paths)
        profiles[author] = cng_profile(freq_tab, profile_len)
    # Predict the authors of the books in the testing corpus.
    for author, book_path in testing_corpus.iteritems():
        freq_tab = compute_freq_tab(3, book_path)
        profile = cng_profile(freq_tab, profile_len)
        # Assign the closest profile from the database.
        pred_author, min_dissim = None, None
        for cand_author, cand_profile in profiles.iteritems():
            dissim = cng_dissimilarity(profile, cand_profile)
            if pred_author is None or dissim < min_dissim:
                pred_author, min_dissim = cand_author, dissim
        print '%s,%s,%s' % (profile_len, author, pred_author)
```

The results of the experiment are summarized in Table VII. It is interesting to see that using only the 10 most frequent n-grams is enough to predict more than 50% of the authors correctly. The success rate increases with larger values of L, and for $L = 500$ a perfect success rate is achieved.

Although outstanding prediction results were obtained with $L = 500$, I believe this approach is not likely to solve the problem definitively. Some situations can be problematic for a prediction technique based solely on the most frequent

n-grams. For example, authors could change their writing style during periods of their lives and this could affect the prediction, particularly if limited writing samples are available.

TABLE VII
RESULTS OF THE AUTHOR ATTRIBUTION EXPERIMENT USING VALUES OF THE GNG PROFILE LENGTH $L \in \{10, 100, 500\}$.

Correct Author	Predicted Author		
	$L = 10$	$L = 100$	$L = 500$
C. Dickens	C. Dickens	C. Dickens	C. Dickens
E. R. Burroughs	E. R. Burroughs	E. R. Burroughs	E. R. Burroughs
L. Carroll	M. Twain	L. Carroll	L. Carroll
Sir A. C. Doyle	M. Twain	Sir A. C. Doyle	Sir A. C. Doyle
M. Twain	M. Twain	M. Twain	M. Twain
H. G. Wells	C. Dickens	E. R. Burroughs	H. G. Wells
F. Kafka	F. Kafka	F. Kafka	F. Kafka
Success Rate	57.14%	85.71%	100.00%

H. Problem 1h [2]

> "Can you develop a metric based on what you have done so far to classify the stories, e.g., as mystery, romance, action/adventure, etc.? Implement your techniques to demonstrate classification. Can the classification scheme you designed help with author attribution? Can you say something about correlations among books written by the same author? Is there any relationship to the styles of the three Bronte sisters' works?"

In addition to author identification, the CNG profiles could be used to classify the stories according to their genre. If a book collection with genre labels were available, a straightforward solution would be to use a supervised learning approach by summarizing the features of each genre using a CNG profile. However, since the data provided for the assignment does not contain genre labels, an unsupervised approach based on clustering is presented. Specifically, the books are grouped using a hierarchical clustering method according to the CNG dissimilarity.

A CNG profile of the $L = 500$ most frequent n-grams based on a third-order frequency table was built for every book on Table I. Then, a CNG dissimilarity matrix was computed for every pair of books. The following Python code implements these two steps and prints the matrix.

```
from itertools import combinations

from monkeys import compute_freq_tab, cng_profile, cng_dissimilarity

corpus = (
    'books/christmas_carol.txt',
    'books/tale_of_2_cities.txt',
    'books/wuthering_heights.txt',
    'books/agnes_grey.txt',
    'books/jane_eyre.txt',
    'books/tarzan_of_the_apes.txt',
    'books/warlord_of_mars.txt',
    'books/the_people_that_time_forgot.txt',
    'books/the_land_that_time_forgot.txt',
    'books/king_solomons_mines.txt',
    'books/fanny_hill.txt',
    'books/alices_adventures_in_wonderland.txt',
    'books/through_the_looking_glass.txt',
    'books/legend_of_sleepy_hollow.txt',
    'books/the_adventures_of_sherlock_holmes.txt',
    'books/the_lost_world.txt',
    'books/the_hound_of_the_baskervilles.txt',
    'books/tales_of_terror_and_mystery.txt',
    'books/adventures_of_huckleberry_finn.txt',
    'books/the_adventures_of_tom_sawyer.txt',
    'books/a_connecticut_yankee_in_king_arthur_s_court.txt',
    'books/the_prince.txt',
    'books/war_of_the_worlds.txt',
    'books/the_time_machine.txt',
    'books/metamorphosis.txt',
    'books/the_trial.txt',
    'books/the_jungle_book.txt'
)

# Build the book's profile database.
profiles = []
for book_path in corpus:
    freq_tab = compute_freq_tab(3, book_path)
    profiles.append(cng_profile(freq_tab, 500))

# Compute the pairwise dissimilarities.
for book_data_i, book_data_j in combinations(enumerate(corpus), 2):
    i, book_path_i = book_data_i
    j, book_path_j = book_data_j
    profile_i, profile_j = profiles[i], profiles[j]
    dissim = cng_dissimilarity(profile_i, profile_j)
    print '%s,%s,%s' % (i + 1, j + 1, dissim)
```

[2] The initial implementation of `cng_dissimilarity` was incorrect (common n-grams were counted twice), so the numerical results presented in this section are not valid.

The cluster analysis was performed in R using the `hclust` function. The data was clustered using an agglomerative method with complete-linkage, which defines the distance between clusters as the dissimilarity between the elements that are farthest away from each other. A dendrogram of the result of the analysis is presented in Fig. 4.

A cut at a given height of the tree corresponds to a clustering of the books. Once the books are grouped, the genre corresponding to each cluster could be obtained by classifying a representative member or through a voting scheme. For example, one of the groups obtained after a cut of the tree at a CNG dissimilarity threshold of 900 contains the following books: "War of the Worlds", "The Time Machine", "Tarzan of the Apes", "The Warlord of Mars", "The People that Time Forgot", and "The Land that Time Forgot". These books could be classified as fantasy novels.

The dendrogram also illustrates that works written by the same author are similar. For example, the books "Metamorphosis" and "The Trial" by F. Kafka clearly form a cluster by themselves. Moreover, it is evident that the works "Wuthering Heights", "Agnes Grey", and "Jane Eyre" written by the Bronte sisters are similar in style.

The proposed classification scheme can help with author attribution. When the algorithm is confronted with a book whose author must be identified, the set of candidate authors can be reduced by considering only authors who have written books of the same genre. This idea could help to reduce the number of false positives.

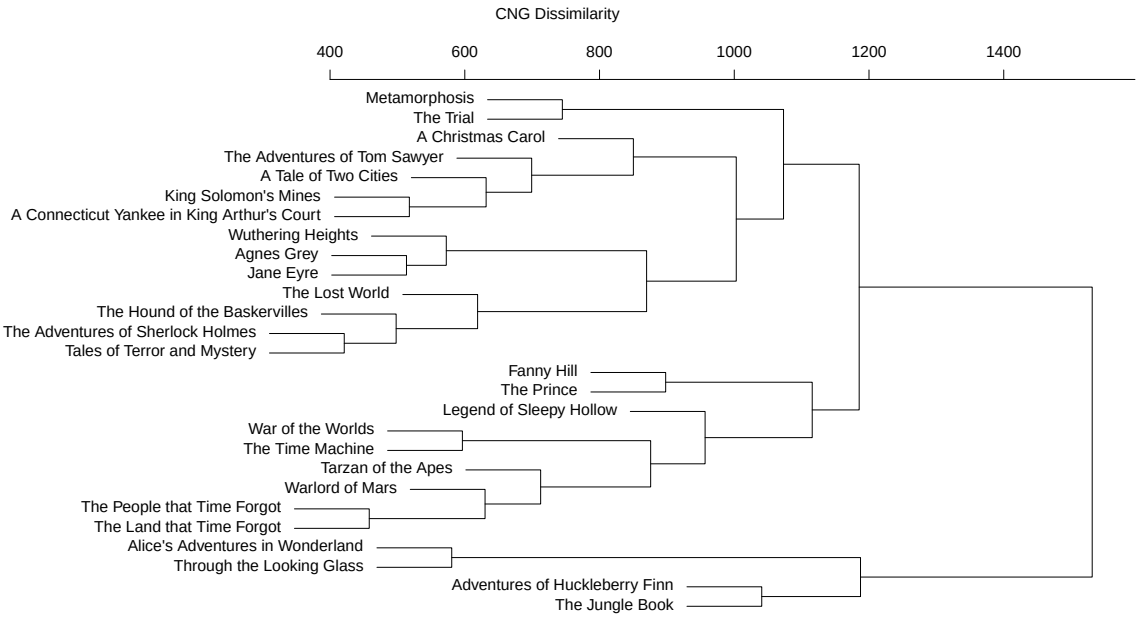

Fig. 4. Hierarchical cluster analysis of the data made available for the assignment according to the CNG dissimilarity.

I. Problem 1i [3]

> "Develop a profile for each of the different authors in Table I and provide a metric and argument that compares and contrasts authors in order to speculate which two authors are the most 'similar' in style."

A CNG profile was built for every author using the writing samples in Table I. Based on the results obtained in Problems 1g and 1h, a third-order frequency table and a CNG profile length $L = 500$ were used. The following Python code computes the CNG dissimilarity between every pair of authors.

```
from itertools import combinations

from monkeys import compute_freq_tab, cng_profile, cng_dissimilarity

corpus = {
    'C. Dickens': (
        'books/christmas_carol.txt',
        'books/tale_of_2_cities.txt'),
    'E. Bronte': ('books/wuthering_heights.txt', ),
    'A. Bronte': ('books/agnes_grey.txt', ),
    'C. Bronte': ('books/jane_eyre.txt', ),
    'E. R. Burroughs': (
        'books/tarzan_of_the_apes.txt',
        'books/warlord_of_mars.txt',
        'books/the_people_that_time_forgot.txt',
        'books/the_land_that_time_forgot.txt', ),
    'H. R. Haggard': ('books/king_solomons_mines.txt', ),
    'J. Cleland': ('books/fanny_hill.txt', ),
    'L. Carroll': (
        'books/alices_adventures_in_wonderland.txt',
        'books/through_the_looking_glass.txt'),
    'W. Irving': ('books/legend_of_sleepy_hollow.txt', ),
    'Sir A. C. Doyle': (
        'books/the_adventures_of_sherlock_holmes.txt',
        'books/the_lost_world.txt',
        'books/the_hound_of_the_baskervilles.txt',
        'books/tales_of_terror_and_mystery.txt'),
    'M. Twain': (
        'books/adventures_of_huckleberry_finn.txt',
        'books/the_adventures_of_tom_sawyer.txt',
        'books/a_connecticut_yankee_in_king_arthur_s_court.txt'),
    'N. Machiavelli': ('books/the_prince.txt', ),
    'H. G. Wells': (
        'books/war_of_the_worlds.txt',
        'books/the_time_machine.txt'),
    'F. Kafka': (
        'books/metamorphosis.txt',
        'books/the_trial.txt'),
    'R. Kipling': ('books/the_jungle_book.txt', )
}

# Build the author's profile database.
profiles = {}
for author, book_paths in corpus.iteritems():
    freq_tab = compute_freq_tab(3, *book_paths)
    profiles[author] = cng_profile(freq_tab, 500)

# Compute the pairwise dissimilarities.
for author_i, author_j in combinations(profiles.iterkeys(), 2):
    profile_i, profile_j = profiles[author_i], profiles[author_j]
    dissim = cng_dissimilarity(profile_i, profile_j)
    print '%s,%s,%s' % (author_i, author_j, dissim)
```

[3] The initial implementation of `cng_dissimilarity` was incorrect (common n-grams were counted twice), so the numerical results presented in this section are not valid.

The normalized CNG dissimilarity matrix of the authors is presented in Table VIII. The minimum entries on each row are highlighted in bold numbers – i.e. the most similar counterparts for each author. Overall, the most similar authors are Sir A. C. Doyle and C. Dickens with a normalized CNG dissimilarity of 0.34, and this relationship is symmetric. Consequently with the results obtained in Problem 1h, it is also noticeable that the Bronte sisters are very similar to each other. For each Bronte sister, the most similar author is always another sister.

IV. Conclusions

The solutions to the problems in the assignment illustrate the potential of character-based n-gram analysis techniques.

TABLE VIII
NORMALIZED CNG DISSIMILARITY FOR EVERY PAIR OF AUTHORS.

	C. Dickens	E. Bronte	A. Bronte	C. Bronte	E. R. Burroughs	H. R. Haggard	J. Cleland	L. Carroll	W. Irving	Sir A. C. Doyle	M. Twain	N. Machiavelli	H. G. Wells	F. Kafka	R. Kipling
C. Dickens	–	0.52	0.48	0.41	0.46	0.47	0.60	0.79	0.60	**0.34**	0.49	0.67	0.55	0.56	0.64
E. Bronte	0.52	–	0.41	**0.37**	0.56	0.63	0.62	0.75	0.73	0.52	0.60	0.76	0.64	0.63	0.79
A. Bronte	0.48	0.41	–	**0.37**	0.56	0.54	0.51	0.72	0.70	0.49	0.53	0.66	0.60	0.53	0.77
C. Bronte	0.41	**0.37**	**0.37**	–	0.48	0.49	0.53	0.81	0.67	**0.37**	0.51	0.70	0.54	0.60	0.73
E. R. Burroughs	0.46	0.56	0.56	0.48	–	0.43	0.59	0.85	0.58	**0.35**	0.54	0.69	0.44	0.65	0.67
H. R. Haggard	0.47	0.63	0.54	0.49	0.43	–	0.65	0.78	0.59	**0.39**	0.42	0.70	0.47	0.66	0.60
J. Cleland	0.60	0.62	**0.51**	0.53	0.59	0.65	–	0.96	0.65	0.58	0.73	0.64	0.64	0.74	0.87
L. Carroll	0.79	0.75	0.72	0.81	0.85	0.78	0.96	–	1.00	0.81	**0.66**	1.00	0.87	0.73	0.83
W. Irving	0.60	0.73	0.70	0.67	**0.58**	0.59	0.65	1.00	–	0.60	0.73	0.72	**0.58**	0.79	0.84
Sir A. C. Doyle	**0.34**	0.52	0.49	0.37	0.35	0.39	0.58	0.81	0.60	–	0.48	0.67	0.46	0.53	0.65
M. Twain	0.49	0.60	0.53	0.51	0.54	**0.42**	0.73	0.66	0.73	0.48	–	0.81	0.57	0.56	0.56
N. Machiavelli	0.67	0.76	0.66	0.70	0.69	0.70	**0.64**	1.00	0.72	0.67	0.81	–	0.76	0.78	0.93
H. G. Wells	0.55	0.64	0.60	0.54	**0.44**	0.47	0.64	0.87	0.58	0.46	0.57	0.76	–	0.71	0.70
F. Kafka	0.56	0.63	**0.53**	0.60	0.65	0.66	0.74	0.73	0.79	**0.53**	0.56	0.78	0.71	–	0.75
R. Kipling	0.64	0.79	0.77	0.73	0.67	0.60	0.87	0.83	0.84	0.65	**0.56**	0.93	0.70	0.75	–

In many cases the solutions offered are only an initial approximation that could be extended in order to obtain better results. The Python source code of the programs implemented for the assignment is publicly available at the git repository `https://github.com/ygf/monkeys-typing`. The repository also contains the LaTeX code for this document, and the R scripts and data files to produce the accompanying figures.

REFERENCES

[1] W. R. Bennett, *Scientific and engineering problem-solving with the computer*. Prentice-Hall, 1976, ch. Language.

[2] V. Kešelj, F. Peng, N. Cercone, and C. Thomas, "N-gram-based author profiles for authorship attribution," in *Proceedings of the Pacific Association for Computational Linguistics*, 2003, pp. 255–264.

APPENDIX
SOURCE CODE OF THE MONKEYS.PY MODULE

```
from bisect import bisect_left
from copy import deepcopy
from itertools import chain, islice, izip, product
from json import dump, load
from random import randint
from re import split


# Build a map of input chars to typewriter chars.
LETTERS = {l:l for l in 'abcdefghijklmnopqrstuvwxyz'}
DIGITS = {d:'#' for d in '0123456789'}
PUNCTUATION = {p:p for p in ' ,.;:?!()-\'"'}
INPUT_CHAR_MAP = dict(LETTERS.items() + DIGITS.items() +
                      PUNCTUATION.items())

# Build a map of typewriter chars to integers and vice versa.
INDEX_MAP = ''.join(sorted(set(INPUT_CHAR_MAP.values() + ['@'])))
CHAR_MAP = {c:i for i, c in enumerate(INDEX_MAP)}

# Number of typewriter chars (40 in the assignment).
NUM_CHARS = len(INDEX_MAP)

# Map any input char into the NUM_CHARS typewriter chars (if a char
# is not found in INPUT_CHAR_MAP it is mapped to @).
map_input_char = lambda c: INPUT_CHAR_MAP.get(c.lower(), '@')

# Assign consecutive integers to every typewriter char (ASCII order).
char2index = lambda c: CHAR_MAP[c]

# The inverse of char2index.
index2char = lambda i: INDEX_MAP[i]


# Compute a freq table of the given order from plain text files. The
# freq table is represented as nested lists with NUM_CHARS entries.
def compute_freq_tab(order, *corpus_files):
    freq_tab = _freq_tab_init(order)
    for corpus_file in corpus_files:
        fd = open(corpus_file, 'r')
        text = fd.read()
        shifted_seqs = [islice(text, i, None) for i in xrange(order)]
        for input_ngram in izip(*shifted_seqs):
            typewriter_ngram = map(map_input_char, input_ngram)
            _freq_tab_inc(freq_tab, map(char2index, typewriter_ngram))
        fd.close()
    return freq_tab


# Reduce the resolution of the typewriter. All the freqs. are
# divided by a percentage of the maximum freq. determined by the
# rate value in (0,1]. A modified copy of freq_tab is returned.
def reduce_freq_tab_resolution(freq_tab, rate):
    order = _freq_tab_order(freq_tab)
    # Find the n-gram with the highest freq.
    max_ngram_freq = 0
    for ngram_index in product(xrange(NUM_CHARS), repeat=order):
        ngram_freq = _freq_tab_get(freq_tab, ngram_index)
        if ngram_freq > max_ngram_freq:
            max_ngram_freq = ngram_freq
    # Calculate the rate that will divide all the freqs.
    rate = float(rate * max_ngram_freq) if rate > 0 else 1
    # Compute the transformed freq. table.
    dup_freq_tab = deepcopy(freq_tab)
    for ngram_index in product(xrange(NUM_CHARS), repeat=order):
        ngram_freq = _freq_tab_get(dup_freq_tab, ngram_index)
        if ngram_freq > 0:
            ngram_freq = int(ngram_freq / rate)
            _freq_tab_set(dup_freq_tab, ngram_index, ngram_freq)
    return dup_freq_tab
```

```
# Export all the n-grams with non-zero freqs to a JSON file.
def write_freq_tab(freq_tab, output_file):
    order = _freq_tab_order(freq_tab)
    nonzero_ngrams = {}
    for ngram_index in product(xrange(NUM_CHARS), repeat=order):
        ngram_freq = _freq_tab_get(freq_tab, ngram_index)
        if ngram_freq > 0:
            ngram = ''.join(map(index2char, ngram_index))
            nonzero_ngrams[ngram] = ngram_freq
    fd = open(output_file, 'w')
    dump(nonzero_ngrams, fd)
    fd.close()


# Build the freq table from the n-gram freqs in a JSON file.
def read_freq_tab(input_file):
    fd = open(input_file, 'r')
    nonzero_ngrams = load(fd)
    fd.close()
    order = len(nonzero_ngrams.iterkeys().next())
    freq_tab = _freq_tab_init(order)
    for ngram, ngram_freq in nonzero_ngrams.iteritems():
        _freq_tab_set(freq_tab, map(char2index, ngram), ngram_freq)
    return freq_tab


# Given an n-gram prefix with (n-1) chars, compute the most probable
# char seq without repeated chars according to the given freq table.
def most_probable_freq_tab_path(freq_tab, ngram_prefix):
    path = ngram_prefix
    ngram_prefix = []
    for char in path:
        ngram_prefix.append(char2index(char))
    while len(path) < NUM_CHARS:
        tab_ref = freq_tab
        for i in ngram_prefix:
            tab_ref = tab_ref[i]
        # Find the char with the highest freq (not already included).
        char_freq, char_index = 0, None
        for i in xrange(NUM_CHARS):
            if tab_ref[i] > char_freq and index2char(i) not in path:
                char_freq = tab_ref[i]
                char_index = i
        if char_freq > 0:
            # Append the char and update the prefix.
            path += index2char(char_index)
            ngram_prefix.append(char_index)
            ngram_prefix.pop(0)
        else:
            # Stop. All remaining chars have zero freqs.
            break
    return path


# Sample num_chars chars from the freq table and save them to
# output_file. freq_tab must be set to None for the straightforward
# monkey problem (order 0).
def simulate_freq_tab(freq_tab, num_chars, output_file):
    order = _freq_tab_order(freq_tab)
    ngram_prefix = []
    fd = open(output_file, 'w')
    for i in xrange(num_chars):
        if order == 0:
            # The straightforward monkey problem (sampled uniformly).
            char_index = randint(0, NUM_CHARS - 1)
        else:
            if len(ngram_prefix) < order - 1:
                # The first (order-1) chars are sampled uniformly.
                char_index = randint(0, NUM_CHARS - 1)
                ngram_prefix.append(char_index)
            else:
                # Sample according to the freq table.
                char_index = _freq_tab_sample(freq_tab, ngram_prefix)
                ngram_prefix.append(char_index)
                ngram_prefix.pop(0)
        fd.write(index2char(char_index))
    fd.close()


# Number correct words in the simulated file divided by the total
# number of words in the simulated file. A word is considered
# correct if it appears in the corpus file.
def relative_word_yield(simulated_file, corpus_file):
    correct_words = 0.0
    simulated_words = _get_words(simulated_file)
    corpus_words = set(_get_words(corpus_file))
    for word in simulated_words:
        if word in corpus_words:
            correct_words += 1
    word_yield = correct_words / len(simulated_words)
    return word_yield


# Build a Common N-Grams (CNG) profile of length profile_len from the
# freq table. The profile is represented as a dict of the ngrams and
# their normalized freqs.
def cng_profile(freq_tab, profile_len):
    order = _freq_tab_order(freq_tab)
    total_freq = 0.0
    ngrams, ngram_freqs = [], []
    for ngram_index in product(xrange(NUM_CHARS), repeat=order):
        ngram_freq = _freq_tab_get(freq_tab, ngram_index)
        if ngram_freq > 0:
            total_freq += ngram_freq
            i = bisect_left(ngram_freqs, ngram_freq)
            if len(ngram_freqs) < profile_len or i > 0:
                ngram_freqs.insert(i, ngram_freq)
                ngram = ''.join(map(index2char, ngram_index))
                ngrams.insert(i, ngram)
            if len(ngram_freqs) > profile_len:
                ngram_freqs.pop(0)
                ngrams.pop(0)
    normalize = lambda freq: freq / total_freq
    profile = dict(zip(ngrams, map(normalize, ngram_freqs)))
    return profile


# Common N-Grams (CNG) profile dissimilarity.
def cng_dissimilarity(profile1, profile2):
    dissimilarity = 0
    for ngram in set(profile1) | set(profile2):
        ngram_freq1 = profile1.get(ngram, 0)
        ngram_freq2 = profile2.get(ngram, 0)
        dissimilarity += (2 * (ngram_freq1 - ngram_freq2) /
                          (ngram_freq1 + ngram_freq2)) ** 2
    return dissimilarity


# Initialize a freq table with all counts set to a given value.
def _freq_tab_init(order, value=0):
    freq_tab = [value] * NUM_CHARS if order > 0 else None
    for n in xrange(1, order):
        tmp = [freq_tab]
        tmp += [deepcopy(freq_tab) for i in xrange(NUM_CHARS - 1)]
        freq_tab = tmp
    return freq_tab


# Return the order of the freq table.
def _freq_tab_order(freq_tab):
    order = 0
    tab_ref = freq_tab
    while isinstance(tab_ref, list):
        tab_ref = tab_ref[0]
        order += 1
    return order


# Get the freq of an n-gram given the indexes of its chars.
def _freq_tab_get(freq_tab, ngram_index):
    tab_ref = freq_tab
    for n, i in enumerate(ngram_index):
        if n == len(ngram_index) - 1:
            return tab_ref[i]
        else:
            tab_ref = tab_ref[i]


# Update the freq of an n-gram given the indexes of its chars.
def _freq_tab_set(freq_tab, ngram_index, ngram_freq):
    tab_ref = freq_tab
    for n, i in enumerate(ngram_index):
        if n == len(ngram_index) - 1:
            tab_ref[i] = ngram_freq
        else:
            tab_ref = tab_ref[i]


# Add 1 to the freq of an n-gram given the indexes of its chars,
# more efficient that using _freq_tab_get and then _freq_tab_set.
def _freq_tab_inc(freq_tab, ngram_index):
    tab_ref = freq_tab
    for n, i in enumerate(ngram_index):
        if n == len(ngram_index) - 1:
            tab_ref[i] += 1
        else:
            tab_ref = tab_ref[i]


# Sample a char from the freq table given the (order-1) chars of
# the n-gram prefix.
def _freq_tab_sample(freq_tab, ngram_prefix):
    # Compute the cumulative values of the freq dist.
    total_freq, cumul_freqs = 0, []
    tab_ref = freq_tab
    for i in ngram_prefix:
        tab_ref = tab_ref[i]
    for i in xrange(NUM_CHARS):
        total_freq += tab_ref[i]
```

```
        cumul_freqs.append(total_freq)
    # Sample an integer in [0,total_freq] and find its position in
    # the ordered list cumul_freqs. The position is the char index.
    char_index = (randint(0, NUM_CHARS - 1) if total_freq == 0 else
                  bisect_left(cumul_freqs, randint(0, total_freq)))
    return char_index


# Return a list with the words in the file.
def _get_words(text_file):
    split_chars = '@'
    for char in PUNCTUATION.iterkeys():
        # The dash must be escaped in the regex.
        split_chars += r'\-' if char == '-' else char
    fd = open(text_file, 'r')
    typewriter_text = ''.join((map_input_char(c) for c in fd.read()))
    fd.close()
    split_re = '[' + split_chars + ']+'
    words = split(split_re, typewriter_text.strip(split_chars))
    return words
```