605.744 Information Retrieval
Paul McNamee

Programming Assignment #5 (due on day 7 of Module 11)

Near Duplicate Detection

Section 19.6 in IIR discusses the widespread duplication (or plagiarism) of content on the Internet. For this assignment you will implement near duplicate detection to automatically discover duplicates in English news collections of varying sizes. In news 20% to 35% duplication is not uncommon. Using shingling with minhashing and super (aka meta) shingles, estimation of Jaccard similarity, and single link clustering (using a Union Find or Disjoint Set data structure) is a recommended approach for large datasets. A detailed description of such methods is presented in Chapter 3 of Leskovec et al., *Mining of Massive Datasets*, Cambridge University Press, which is available online, along with other materials from their book, including slides. But you are also welcome to explore and use other methods. Simpler approaches may be less effective, but they are often substantially easier to implement and can be efficient. For example, you could decide that any two documents with an entire sentence in common are duplicates. More complex algorithms tend to require effort to implement, and may present scalability challenges on larger datasets (i.e., memory limits, excessive runtimes). If you hit performance limits a profiler may prove useful (e.g., cProfile in Python).

Conditions for this assignment:

- You should not use any software packages or libraries that solve the duplicate detection problem. The goal is for you to implement a solution rather than using an out-of-the-box tool.
- It is perfectly fine to use standard libraries or packages that perform general functions such as bit operations, string comparison, hashing, sorting, edit distance, merging, etc... In your submitted writeup you should cite any packages used that are not a standard part of a programming language.
- The work is to be done on a single machine. While in the real world large-scale deduplication algorithms are often designed to run on parallel hardware, for this lab you will use a single machine.

The input datasets are provided in tab separated value (TSV) files. For each test size there is a single file with lines containing an integer docid followed by a single tab and then a possibly long line of text (i.e., up to 2,500 words). You must create equivalence clusters of near-duplicate documents with the following formatting requirements:

- Each line will contain space separated integer docids from the input file
- Each docid should occur exactly once in the submitted file. No docid may occur on more than one line.
- Docids present on the same line belong to the same equivalence cluster and are deemed to be near duplicates.
- No text should appear in the output file. Only docids should be present. There should be no blank lines.
- You may present clusters in any order in the file. You may choose any ordering of docids on a line.
- A submitted cluster file should be named *jhed-test.txt* (e.g., jlee14-thirtyk.txt). Attach submitted cluster files individually, not in a zip file or tarball.

You do not have to submit results for all test sizes. But only one submission may be provided for a given size. A figure showing the input format and cluster files is shown below:

Input File

1	two cherry pumpkin tarts
2	cherry garcia ice cream
3	two cherry pumpkin pies
4	cheeseburgers in paradise

System A

3 1 2
4

System B

3
4
2
1

Correct

4
1 3
2

Figure 1. This test has four documents. Documents 1 & 3 are near duplicates that should be clustered together. System A returns two clusters. System B did not cluster any docids together; none are deemed duplicates. A perfectly correct clustering is shown at the right.

There are eight input files of sizes: 30, 100, 300, 1000, 3000, 10000, 30000, and 100000. I expect all students will be able to submit results for the smaller test sets. Your grade on the assignment will be based substantially on how large of a collection you are able to process and accurately cluster. Submitted equivalence clusters will be scored against a reference clustering using the B-Cubed metric (Bagga and Baldwin, Entity Based Cross Document Coreferencing Using the Vector Space Model, ACL '98).

The assignment will be graded using the following rubric:

- 25 points. A clear and complete description of methods employed. The focus should be on the algorithmic approach(es), although there should also be details regarding software libraries used and implementation design. I will want to know how documents are processed (e.g., tokenized), how similarity is determined, what hashing libraries are used, etc...
- 15 points. Present some analysis of runtime, memory, and accuracy/errors. I would be interested to see comparisons of alternative approaches or tradeoffs from different implementation design. Use of charts or tables is encouraged. I am not expecting a comprehensive study, but I would like to see some insight (e.g., you did X, and the system was faster; or you did Y, and memory consumption dropped; or you did Z, and accuracy improved).
- 10 points. Submitted cluster files comply with the constraints described previously.
- 50 points. Achieving various targets in B-Cubed F_1 scores. You will earn the points from the highest scoring target which your program achieves. For example, if you submit results for two tests (10,000 and 30,000) and your submissions scored 97.2 on the 10k test, and 96.8 on the 30k test, you would be awarded 42 points for meeting the 10k target value.

Test	B^3 F_1 Target	Points		Test	B^3 F_1 Target	Points
30	80	20		3,000	94	38
100	85	25		10,000	96	42
300	90	30		30,000	97	46
1,000	92	34		100,000	97.5	50

In the table below I report runtimes and approximate memory consumption on the assigned test sets using a single ~300 line Python implementation of shingling/minhashing running on my 3 year old Apple laptop. You do not need to match or improve upon my reported times, I only provide these data to let you know that this is feasible. The assignment will not be directly assessed for efficiency.

Test	Seconds	RAM (MB)		Test	Seconds	RAM (MB)
30	5	17		3,000	495	56
100	17	18		10,000	1654	135
300	51	20		30,000	4993	377
1,000	169	29		100,000	16424	1257

The third-party packages that I used in my implementation were: (a) Avinash Kak's BitVector.py (pip install bitvector); and (b) the Union Find implementation from networkx (pip install networkx).

Submit a writeup, containing the requested description and analysis, as well as your source code in a single PDF document. Also attach between zero and eight cluster files for any test cases that you wish to submit results for.