For this assignment, you'll implement both locality based hashing (LSH) and a brute-force nearest neighbor approach on a particular dataset. You'll measure the difference in running times as well as the differences in similarity between nearest neighbors. You should be able to see how much faster LSH is, with similar accuracy.

In this assignment, we'll be looking for nearest neighbors not for classification purposes, as we did on the previous assignment; instead, we're going to take a dataset of documents, and for each, find all documents similar to it. This could be done via an awful $O(n^2)$ approach of comparing each document to every other document, but we'll do much better than that.

This approach *can* also be used for k-NN classification on unseen test data; we'll talk about that in class.

Programming Environment

I wrote this program in Python, and in the end it worked great. A major problem I had to overcome, however, is that the timing numbers came out crazy when I used the typical Python environment, i.e., the usual one you get when you type `python`. (Incidentally, this environment is called "CPython", because the environment itself is written in C.) That's CPython interprets the code that it runs, which is generally slow, but some specific built-in commands run under-the-hood with compiled C code, which is fast. If you use more built-in compiled commands with the brute-force algorithm, you'll get the incorrect sense that the brute-force approach is dramatically faster.

To resolve this, you need to use a programming environment where everything is compiled to machine language or close before the program starts running. I ran my program with `pypy`, which is a Python compiler that does exactly like this. Two fantastic things happened when I switched to `pypy`:

- Both algorithms ran *much* faster than when running with `python`.
- The timing measurements all started to make sense.

You should similarly be able to program this using the standard Java, C, C++, or other environments that compile code to machine language before running. I believe, but am not entirely sure, that R does *not* do this. Feel free to ask on Piazza if you have questions about your particular programming environment.

Data

The data you'll use for this assignment comes from the so-called "Bag of Words" dataset at the UCI Machine Learning Repository. This dataset is actually four datasets; we'll use the one called `enron`. If you missed all of the news excitement a number of years ago regarding the collapse of the energy company Enron, feel free to dig through some news archives to read about the scandal. This remarkable dataset contains nearly 40,000 internal emails from the company that were released for the public record.

All four datasets are located this public directory. You'll only need to look at the `readme.txt` file, which explains the layout, and the file `docword.enron.txt` which contains the actual data. Note that this file has been compressed via gzip, and so you'll need to use the command `gunzip` to uncompress it. Happily, this file has already converted all words to integers. The words themselves are unnecessary for this exercise, but if you're curious to see a list of them, they're in `vocab.enron.txt`. I've also placed these files within `/Accounts/courses/cs324`. Uncompressed, this file is about 47 MB, so *do not place this in your home directory on our department network*. Make sure that you put the file somewhere local, such as `/tmp`.

This dataset contains counts for the number of times each word appears, but you'll ignore the count. Since we're focusing on Jaccard similarity, you only need keep track of whether or not a word appears.

Part 1: Comparing Jaccard Similarity with Minhashing

Start off by writing a program that reads in the above data in some fashion and stores it in some kind of in-memory data structure. One word of warning: don't store the data in dense-format, i.e. with each column representing a document, each row representing a word, and using a 0 or a 1 to indicate whether or not a word is present. The collection has approximately 28,000 unique words and 40,000 documents. The above will take nearly all or more memory than your computer likely has available – do the math and see. It would also be miserably slow and inefficient. I chose to represent each document as a *set* of words. Both Python and Java have the ability to efficiently manage sets (`set` and `HashSet`, respectively).

Reading in all 39,861 documents takes a while, and it is silly for you to wait for your program to do this every time it runs while you are debugging (and while the graders are grading). Implement the capability for your program to only read in the first n documents, and thus simulate the effect of having a smaller dataset to work with. This is a very common trick practicioners use to debug code that runs on large datasets; it's much faster than running it on the whole dataset every time you re-test your program.

Write code that allows you to compute the Jaccard similarity between two particular documents of interest (as identified by their document id). If you have used built-in set capability as I did above, this should be very, very easy using associated built-in set operation tools.

You should then also write code to produce a signature matrix with a specified number of rows. As we've talked about, each row is a minhash associated with a random permutation of the rows of the original characteristic matrix. You want to do this the efficient and fast way. Don't actually permute the dataset for each minhash; rather, generate a random hash function. I did this as in the reading by generating hash functions that looked like $h(x) = ax + b \ mod \ n$ where n was the number of words in the dataset, and a and b where randomly chosen integers that ranged from 0 to $n - 1$. (You'll want to make sure to choose a so that it's relatively prime with n. That's not hard. but I'll leave that as a little puzzle to think about.) This matrix should be stored as dense in some sort of typical 2-dimensional matrix format.

Once you have produced the signature matrix, write code that allows you to use it to estimate the Jaccard similarity between two particular documents (again identified by id). If you're doing this

correctly, you should be able to see that the estimate gets better and better as you increase the number of rows in the signature matrix.

One warning: since you are using randomization in your program, *make sure that you seed the random number generator* before you start. If you do not do this, your program will run differently every time you run it, and make it nearly impossible to debug. There's no reason that you need to run your program differently every time; in fact, you want it to use the same sequence of "random" permutations every time you run it.

To make grading as straightforward as possible, the program that you actually submit should do the following:

- Do NOT submit the data file itself. We don't need to waste space on Courses with multiple copies of the same 47MB file.
- Specify exceedingly clearly in comments where the string is that contains the location and name of the data file. Since the grader may have the file in a different location than you do, the grader will have to modify this string in your program. Optimally, if you store your data in `/tmp`, that's the same place the grader will use. The grader will be particularly happy if you use `/tmp`.
- Prompt the user to enter in the number of documents for your program to read in from the file.
- Prompt the user to enter in two document id numbers. Note that the document ids are indexed so they start at the number 1. You may wish to subtract 1 from all document ids internally within your program to make hashing easier, but this should be hidden from the user. The user should have the experience that the document id they type in is consistent with the document ids that appear in the file.
- Print out the exact Jaccard similarity of these two documents.
- Prompt the user to enter in a number of rows to use for the signature matrix.
- Print out an estimate of the Jaccard similarity for these same two documents based on the signature matrix.

Part 2: Finding nearest neighbors using both approaches

The brute-force approach

Add to your program the capability for finding the k nearest neighbors for a given document ID, using a simple brute-force approach of computing the exact Jaccard similarity between this document and the rest. Remember that Jaccard similarity *increases* with document similarity; it is not a distance like the metrics that you used on the previous assignment.

As you iterate over all documents, you need to keep track of the k most similar documents you have seen thus far, and which one has the smallest similarity so you can remove it if you find a

better one. This is a perfect application for a priority queue, and I encourage you to use one for this task. Both Python (`heapq`) and Java (`PriorityQueue`) have this capability built-in.

Once you have found the k nearest neighbors for a document ID of interest, average the Jaccard similarities for those k neighbors to get a measure of how close the nearest neighbors for this document are.

Finally, produce a single average score by "averaging the averages" to indicate with a single number the quality of finding k nearest neighbors for all documents in the dataset.

Make sure that you test this on a small subset of your data, using the capability you implemented in Part 1 for only reading in a subset. It will take way too long to run on the full dataset.

The LSH approach

Implement an LSH approach for finding the k nearest neighbors for a given document ID by implementing the banding technique descrived in section 3.4.1 of the textbook. Specify as a parameter in your code the number of rows in each band. Once you have done this, the number of bands can be automatically determined based on the number of signatures that you have. Here's how I chose to implement it:

- I created a Python dictionary for each band. Specifically, I used a `defaultdict` as it made the coding a little cleaner, but that's not critical.
- Within each band, I looked at each document and implemented the technique described in the second paragraph of Section 3.4.1, To hash a vector of r integers, I converted those into a tuple, and used them as the *key* for the dictionary. (Python will let you use a tuple, but not a list, as a dictionary key. Java does not, to the best of my knowledge, have this capability built-in, and so you'll need to implement your own wrapper object and `hashCode` method accordingly. Or perhaps use a MultiValueMap, which I haven't tried. You'd have to get this installed.) The dictionary *value* I represented as a set of document IDs. This way, whenever more than one document produced the same vector of r integers, I would store each of those document IDs in the set.
- Once I had a Python dictionary for each band, for a given document I would find its nearest neighors by first generating a set of candidates. Candidates were all documents IDs that appeared in one or more of the same hash buckets as the document under consideration.
- One major problem that came up, with the textbook doesn't tell you about, is that for any given item you run the risk that there may identify fewer than k candidates. You can manage this to some degree with an appropriate choices for k, r, and the number of rows in your signature matrix; but ultimately, your code and your measurement has to be robust to this. In the event that you cannot find k candidates, you should fill in the remaining neighbors with choices chosen entirely at random from the rest of the dataset.

To make grading as straightforward as possible, the program that you actually submit should do the following:

- Do NOT submit the data file itself. I'm pretty sure that you can't, actually — I think Moodle has a limit on the size of the file you can submit. That's good: we don't need to waste space

on a file submission server with 30 copies of the same 47MB file.

- Specify exceedingly clearly in comments where the string is that contains the location and name of the data file. Since the grader may have the file in a different location than you do, the grader will have to modify this string in your program. Optimally, if you store your data in `/tmp`, that's the same place the grader will use.
- Prompt the user to enter in the number of documents for your program to read in from the file.
- Prompt the user to enter in k, the number of nearest neighbors.
- Prompt the user to enter in a number of rows to use for the signature matrix.
- Prompt the user to enter r, the number of rows in each band.
- Calculate the average value of the average similarity for all documents using the brute force approach. Also measure how long it takes. Make sure that your code prints out intermediate results while it's working so the grader knows it is still working.
- Calculate the average value of the average similarity for all documents using LSH. Also measure how long it takes, and measure how many documents had at least one document chosen at random as its nearest neighbor.

Written report

Submit a written "lab report" describing what you find from an experimental perspective. How does the performance of LSH compare to brute-force, with respect to average similarity and running time? How does this vary with datset size, k, r, and number of rows in signature matrix? Your report should show some plots to show how these variables affect the result, and some commentary by you explaining the results you see, and why they are occurring.

Final Thoughts

Here is a collection of random advice and suggestions, based on scanning through the final Python code that I wrote. Some of this may repeat other comments made above. In some cases I'll give advice for Python or Java; if you're using another language, you'll have to appropriately adapt.

- Seed your random number generator. You want this program to behave the same way every time you run it. Read the documentation on the Python `random` module or the Java `Random` class if you don't know how to do this.
- Organize your program from the very beginning so that you can read in only a subset of the documents of size that you pick. To debug your code, you want it to run quickly, and running it on all the documents every time you test it is exceedingly slow.
- Heavily use built-in capabilities for managing sets, including operations such as intersection, union, and difference. Don't reinvent this yourself.

- Use built-in capabilities for managing priority queues, if your language offers it. Don't reinvent that yourself.
- For any portion of your code that runs slowly, print updates to the screen regularly so that you can tell what it's doing, and so you can guess how long it will take the entire program to run. During slow loops, I typically print out a count every 100 rows or every 1000 rows. You may need to *flush* the output buffer when you do this, to ensure that the output you see is timed consistently with the code that is running. In Python, you do this via `sys.stdout.flush()`. In Java, you use `System.out.flush()`.
- If you are programming in Python, use `pypy`. Do *not* use the regular `python` command.
- If you are programming in Python, it is super useful to learn list comprehensions if you haven't come across them before. They're never critical, but they can really condense your code.
- Don't interactively enter in the values of your parameters every time your run your program. Hard code them in, or use command-line parameters. Your debugging will go MUCH faster if you don't have to type the values in every time you run your program. I asked you to make them keyboard inputs to make it easier for the graders, but you should make that change when you are nearly done.

Have fun with this! This assignment is challenging, but I think you'll learn a whole lot about minhashing, LSH, and possibly some programming along the way.