Data Mining

Homework 3

Due: 11/5/2014, 23:59.

You must hand in the homeworks electronically and before the due date and time. Check the web page for instructions about collaboration, about being late, and about handing in the homework.

Hand in:

- The `.py` files with the source code.
- Instruction to execute them.
- The output files generated, and output at the screen.

Problem 1. We will run locality-sensitive hashing (LSH) on a set of announcements that we will obtain from the kijiji web site. In this exercise we do the first step, which is downloading and parsing the web pages with the announcements.

For downloading the web pages you may use the package `Requests` or the package `urllib2`. To parse the page you can either use regular expressions through the package `re` (it is anyway a good idea become familiar with regular expressions), or, probably better, use an HTML/XML parser. The `Beautiful Soup` package is a good one but it loads the whole file in memory. This is fine for this problem, since the pages to parse are small, but be careful if you want to use it on large XML files; for those ones check the `lxml` library and the tutorial at

`http://www.ibm.com/developerworks/xml/library/x-hiperfparse/`

Write a program that will download from `http://www.kijiji.it` and parse all the job positions in Rome about *Informatica/Grafica/Web* with a *Contratto*. Download regular and top announcements, but not sponsored ads. Save in a tab-separated value (TSV) file, for every job (one line per job), the *title*, *short description* (from the job summary page), *the location*, the *publication date* of the job announcement, the *URL link* to its web page, and the *full description* of the ad; for the full description you need to visit the ad web page. **Because you will make a lot of calls to the kijiji site, make sure that you have a delay (use: `sys.sleep()`) between different downloads of kijiji pages, to avoid being blocked.**

Problem 2. Here we are asking to implement nearest-neighbor search for text documents. You have to implement shingling, minwise hashing, and locality-sensitive hashing. We split it into several parts:

1. Implement a class that, given a document, creates its set of character shingles of some length k. Then represent the document as the set of the hashes of the shingles, for some hash function.

2. Implement a class, that given a collection of sets of objects (e.g., strings, or numbers), creates a minwise hashing based signature for each set.

3. Implement a class that implements the locally sensitive hashing (LSH) technique, so that, given a collection of minwise hash signatures of a set of documents, it finds the all the documents pairs that are near each other.

To test the LSH algorithm, also implement a class that given the shingles of each of the documents, finds the nearest neighbors by comparing all the shingle sets with each other.

We will apply the algorithm to solve the problem that companies such as kijiji face when companies or individuals post many copies of the same announcement—usully they want to block

announcements that are near duplicates. We will work on the announcements for job positions of Problem 1.

We want to find announcements that are near duplicates. We will say that two announcements are near duplicates if the Jaccard coefficient of their shingle sets is at least 80%. We will use shingles of length 10 characters. Find values for r and b (see Section 3.4 in the book) that can give us the desired behavior. To plot the graph that gives the probability as a function of the similarity for different values of r and b you can use, for example, Wolfram Alpha.

To apply the algorithm you have the following tasks:

1. Find the near-duplicates among all the announcements of Problem 1 using LSH.
2. Find the near-duplicates among all the announcements of Problem 1 by comparing them with each other.
3. Report the number of duplicates found in both cases, and the size of the intersection.
4. Report the time required to compute the near duplicates in either case.

You will need a way to create a family of hash functions. One way is to use a hash function and a code similar to the following.

```
# Implement a family of hash functions. It hashes strings and takes an
# integer to define the member of the family.
# Return a hash function parametrized by i
import hashlib
def hashFamily(i):
  resultSize = 8        # how many bytes we want back
  maxLen = 20           # how long can our i be (in decimal)
  salt = str(i).zfill(maxLen)[-maxLen:]
  def hashMember(x):
    return hashlib.sha1(x + salt).digest()[-resultSize:]
  return hashMember
```

Note that this code is an overkill because we use a cryptographic hash function, which can be very slow, even though it is not needed to be as secure. However, for the necessities of the homework we will use it to avoid having to install some external hash library.

Problem 3. When talking in the class about the randomized algorithm for computing frequent itemsets, we mentioned the following problem: How can we sample a uniformly random sample of exactly k lines from a file whose number of lines we do not know in only one pass? Here we will study the special case of $k = 1$, although the algorithm can be generalized for any k.

To abstract the problem, assume that you have a set of n elements that arrive one after the other *but you do not know* n. The goal is to maintain an element, so that when the sequence stops, each element has probability $1/n$ to be selected.

If you want stop before reading to try to come up with the algorithm.

The following algorithm achieves this: We will keep the random element in a variable. In addition we will count the elements that have arrived so far. When the first element arrives we store it in the variable. Now for each i, when the ith element arrives, then with probability $1/i$ we replace the element in the variable with the element that just arrived; otherwise, we do not do anything.

Prove that when the n elements have arrived (and the sequence is stopped) each element that has arrived has probability exactly $1/n$ to be in the variable. (**Hint:** Use induction.)

Problem 4. Very often, when we search for frequent itemsets, we can be tricked: we may find items that are frequent even though the fact that they are frequent may be just because of chance and not because of any underlying reason. In this problem we will see an example of this situation.

Assume that we have n items, m baskets, and for every basket each item appears with probability p, independently of all the other items.

1. Consider an itemset of k items $\{i_1, i_2, \ldots, i_k\}$. Calculate what is the expected number of times that we will find the itemset in the m baskets.

2. Let's fix $n = 2000$, $m = 10^5$, and $p = 0.005$. How many times do we expect to see a particular item? How many times do we expect to see a particular pair of items?

3. Now perform some simulations using the values above and compute the frequency of the most frequent pair. Repeat them 10 times. How many times you did you find an item appearing more than 10% of its expectation? How many times did you find a pair that appears at least 5 times its expectation?

4. How do you explain the large difference between 2 and 3 in the case of pairs? And why do we get different results for single items and pairs?

Even though the data were generated completely at random, some pairs appear to be much more frequent than others. This example shows us that we should be careful not to draw fast conclusions. Generally, often we may think that there is some signal in our data, when in reality there is none.

Optional (extra credit): Propose and implement a way to deal with the problem above.