Data structures
Hashing

1 Introduction

Retrieving similar (text) documents is an important task for many data-mining applications such as plagiarism detection, search engines, web crawlers and recommender systems.

In this session, we will develop a simple system to detect similar news articles. This can for example be used in a news recommender system where we want to make sure that we do not recommend two very similar articles to the user. We will start with a naive implementation and improve this step by step to make it more efficient making clever use of hash functions.

2 Representing text as sets

The start code contains a readData method. This method expects a path to a text file (e.g. articles_100.train) as input and fills two data structures: a List<Set<String>> with for each article a set of the unique words in that article and a List<String> with the article IDs.

The start code contains different sized datasets. Start with the smallest one to make sure your method works correctly and then use the larger datasets to evaluate the performance. The ground truth (which articles match) are provided in the ".truth" files. These files are used in the readGroundTruth method.

3 Calculating the similarity between sets

To find similar articles, we need to find similar sets. One way to calculate the similarity between sets is the Jaccard index. The Jaccard index is defined as the size of the intersection divided by the size of the union of the sample sets A and B.

$$J(A,B) = J(B,A) = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|}$$

- Implement a class that calculates the Jaccard index for every pair of documents. This class implements the DuplicateFinder interface. The findDuplicates method returns all Pairs where the similarity is higher than the provided threshold.

4 Optimization 1: replacing string comparisons by integer comparisons.

To calculate the Jaccard index, we need to compare two sets of strings. This is an expensive operation as string comparisons are expensive and each set can be large. Integers are much easier to compare than strings.

- Implement a class that calculates the Jaccard index for every pair of documents. But now first replace each String with an integer.
- To transform words to integer IDs we have two options, each based on hashing. Either we assign our own unique id to each word (the mapping can be efficiently stored in a hashtable) or we use the built-in hashcode() method of a Java String to determine this value. What is the advantage and disadvantage of each method ?
- How much faster is the jaccard method now ?

5 Optimization 2: Minhashing

Calculating the intersection and union of two very large sets of is an expensive operation, even if the sets contain integers. As it turns out we can use hashing to approximate the Jaccard index more efficiently.

Minhashing is an algorithmic technique that hashes items to short codes. Similar items will result in similar codes. In our case this means that we can create a hash of each document and that similar documents will result in similar hash codes. Note that this is the opposite of what we typically want of a hash function. If we use a hash function for a hash table for example, we try to hash similar keys to completely different hash codes. Also for cryptographic hash functions that are used to validate the authenticity of documents, you typically want that small changes result in very different hash codes.

Minhashing can be used to quickly **estimate** how similar two sets are. It is commonly used in search engines to find duplicate web pages.

Minhash uses k different hash functions. For each hash function and each document, it finds the element in the document that results in the lowest hash code. Each document is then represented by these k elements. We then check how many of these k minhashes match (y) and use y/k as an estimate for the Jaccard index.

Example

word	hash 1	hash 2
a	8	2
b	23	8
c	3	9
d	5	12
e	51	6
f	1	21

Given two hash functions (k=2), we map each word to its hash. Each document is then represented as the two words of that document that resulted in the lowest hash value for one of the hashes. By comparing these minhashes, we estimate the Jaccard similarity.

doc1={a,c,d,e} is mapped to {c, a} because "c" results in the lowest hash code for hash function 1 and "a" results in the lowest hash code for hash function 2.

doc2={a,b,d,f} is mapped to {f, a} because "f" results in the lowest hash code for hash function 1 and "a" results in the lowest hash code for hash function 2.

The Jaccard index is estimated to be 0.5. $(c \neq f, a = a)$.

The more hash functions you use, the more accurate the estimation will be but of course the more expensive it is.

Our hash functions are defined as $h(x) = (ax + b)\%c$. Where the coefficients a and b are randomly chosen integers less than the maximum value of x. c is a prime number slightly bigger than the maximum value of x. Choosing k different hash functions thus involves choosing k random integers for a and b. You can find a list of possible prime numbers for c here: http://compoasso.free.fr/primelistweb/page/prime/liste_online_en.php. For this assignment, you can just pick a large prime number. x is the element of the set and is here assumed to be an integer (the result of section 4 or 5).

- Implement the Minhashing approximation to find the similarities between sets.
- Experiment with different k values.
- How much faster is our Minhashing approach compared to our Jaccard method ?

6 Optimization 3: Locality Sensitive Hashing (LSH)

Minhashing allowed us to compare small hashes instead of large sets. Two sets of m=1000 elements each can for example be mapped to two minhashes with k=10 elements each. Instead of calculating the intersection and union ($O(m^2)$ if done naively), we only need to check which of the k minhash elements match ($O(k)$).

Minhashing makes comparing two articles much easier but to find all duplicates in our dataset we still have to compare the minhash of each article with the minhash of all other articles. This is $O(n^2)$ with n the number of articles in the system. Can we do better than this ? As it turns out, we can again use hashing to improve our performance.

The trick is to realize that we don't need to compare each article with each other article. We can use hashing to divide the articles into buckets where two articles that are very similar should have a high chance of ending up in the same bucket. We then only have to compare the articles that were hashed into the same bucket since these are potential matches. Our hash function might result in collisions where two articles that are not related end up in the same bucket, this will result in a lower performance but it will still be better that the naive approach from the previous section (which can be seen as bucketsize 1). This idea is known as Locality Sensitive

Hashing (LSH). The "locality sensitive" means that similar items should be mapped to similar buckets. Again, this is the complete opposite of what you typically want for a HashTable where similar (but not equal) items should be mapped to completely different indices.

We can visualize LSH as a matrix M with k (=number of minhashes) rows and n (=number of documents) columns. We divide the rows in b bands of r rows each ($r * b = k$). For each band we hash the portion of each minhash in that band to a fixed number of buckets. If the part of two minhashes in this band is the same, they will end up in the same bucket and they are a candidate match.

We repeat this procedure for every band (using a different set of buckets). Instead of comparing the minhash of every document with that of every other document we now only have to compare those documents that were hashed into the same bucket for at least one band.

- Implement Locality Sensitive Hashing. Use Horners method to calculate the hash.
- Experiment with different b values and number of buckets. What is the trade-off between these values?
- How much faster is our Locality Sensitive Hashing approach compared to the previous methods ?

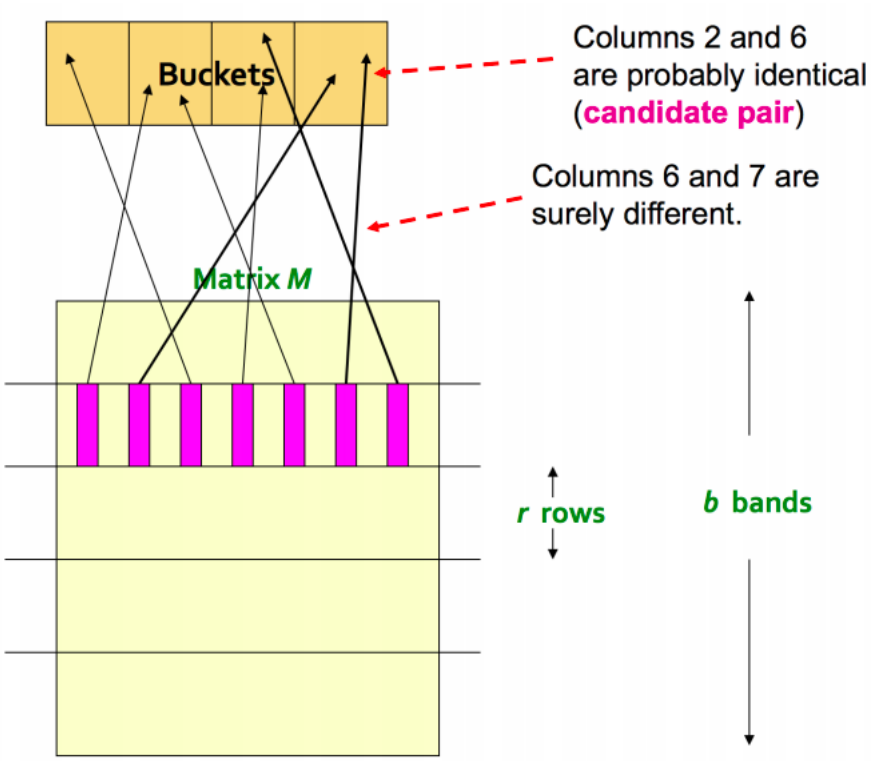

Figure 1: Source: https://towardsdatascience.com/understanding-locality-sensitive-hashing-49f6d1f6134