We implemented minhash and locality sensitive hashing to estimate Jaccard similarity among documents and to identify near-duplicate documents.

1: Procedures in constructing minhash matrices

Given a collection of documents $D = \{D_1, \dots, D_N\}$ consisting of terms $\{T_1, \dots, t_M\}$, the term-document matrix is an $M \times N$ matrix, and this matrix contains all the information to compute Jaccard similarity of any two documents from the collection. We need to construct a $K \times N$ MinHash matrix that can be used to estimate similarity of any two of the documents, where K is the number of random permutations.

The tasks are accomplished by the class `MinHash`, which has the following constructor and methods.

`MinHash(String folder, int numPermutations)` is a constructor where **`folder`** is the name of a folder containing our document collection for which we wish to construct MinHash matrix. **`numPermutations`** is the number of permutations to use in creating MinHash matrix.

`allDocs()` returns an array of *String* consisting of all the names of files in the document collection.

`exactJaccard(String file1, String file2)` gets names of two files in the collection, **`file1`** and **`file2`**, and returns the exact Jaccard similarity.

`minHashSig(String fileName)` returns the MinHash signature (array of *ints*) of the document, **`fileName`**.

`approximateJaccard(String file1, String file2)` estimates and returns the Jaccard similarity of **`file1`** and **`file2`** by comparing their MinHash signatures.

`minHashMatrix()` returns the MinHash matrix of the collection.

`numTerms()` returns the number of terms in the collection.

`numPermutations()` returns the number of permutations used to construct the MinHash matrix.

`buildBinaryTermFrequencyAllDocs()` collect all terms in the documents. This method sets up two *Maps*, **`binaryTermFrequencyAllDocs`** and **`binaryTermFrequencyDoc`** during its routine; the former is implemented by **`HashMap<String, Integer>()`** to store terms (*String*) and corresponding id numbers (*Integer*), and the latter is implemented by **`HashMap<String, Set<Integer>>()`** to store the set of terms appearing in each document denoted by term id numbers (*Integer*) and **`fileName`** (*String*). The procedure to assign an integer to each term is as follows.

```
assign numTerm to 0
for each document in the collection
    create a Set, binaryTermFrequency (implemented by HashSet<Integer>), to store the
    terms that appear in the document by the id numbers (Integer)
    for each line in the document
        preprocess the line and return words as terms[1]
        for each returned term
```

[1]We form words as terms with the following pre-processing: convert all words into lower case, remove only the following punctuation symbols, periods, comma, colon, semicolon, apostrophe. And we then remove all STOP words - any word with length less than three and "the". The method **`wordsProcessdor`** in the class is for this task.

```
                if the term has not appeared so far, not in binaryTermFrequencyAllDocs
                    add the term into binaryTermFrequencyAllDocs with its term number assigned
                    to numTerm[2]
                    increment numTerm
                end
                    get the term id and add it to binaryTermFrequency[3]
            end
        end
        add the pair of fileName and binaryTermFrequency to binaryTermFrequencyDoc[4]
    end
```

minHashTerm(Integer item, int permutationInd) uses the designated permutation function, **permutationInd**, to calculate the hash value of the given term, **item**. To save the memory space on store permutations, we chose to generate permutations by using $ax + b \mod p$. This works because if such a function with an appropriate choice of a prime number p is able to serve as a one-one function that simulates a permutation. Since our goal is to produce an index, not just an integer, before doing modular hashing, we masks off the sign bit to turn the 32-bit integer into a 31-bit nonnegative integer, i.e., $ax + b$ & `0x7fffffff` (the idea comes from `http://algs4.cs.princeton.edu/34hash/`).

The procedure on generating k permutation functions is implemented in the class `PermutationsFunctions`, whose constructor takes **numPermutations** and **numTerms** as parameters. The p value is decided based on **numTerms**; more precisely, p is the closest prime number that is greater or equal to **numTerms**. For each permutation function, two integer numbers, that is a and b, are randomly picked between 0 and p. During the procedure, if there are any duplicate values ($a_i \neq a_j, b_i \neq b_j, \forall 0 \leq i, j < k - 1$), the number is regenerated till no numbers are duplicate.

2: Tests on how accurately MinHash matrices do to estimating similarity

We creates a class `MinHashAccuracy` that tests how accurate the estimated Jaccard similarity is using MinHash matrix. This class has one method.

accuracy(String folder, int numPermutations, double error) gets a name of folder, number of permutations and error parameter ($\epsilon < 1$) as parameters (in that order). The method creates an instance of `MinHash`, and for every pari of files in the document collection, compute the exact and estimated Jaccard similarities (obtained from methods **exactJaccard** and **approximateJaccard**, respectively). It at the end reports the number of pairs for which exact and approximate similarities differ by more than ϵ.

We ran the following choices of **numPermutations**: 400, 600 and 800, and the following choices for ϵ: 0.04, 0.07 and 0.09. As a result there were nine possible combinations. The experiments were conducted on the files in the folder ***space***. For each combination, the number of pairs for which estimated and exact

[2] For every preprocessed identical word from the collection, it receives a number as identification. The number starts from 0 and increments; hence the biggest number overall is the number of terms that can be identified among all the documents.

[3] *HashSet* will only put the term ids not previously exists into the set. We do not need to count the frequency of a term.

[4] **fileName** serves as a key and **binaryTermFrequency** is the value in the *HashMap* **binaryTermFrequencyDoc**. The *HashMap* helps the time on retrieving the set of appearing terms for a document.

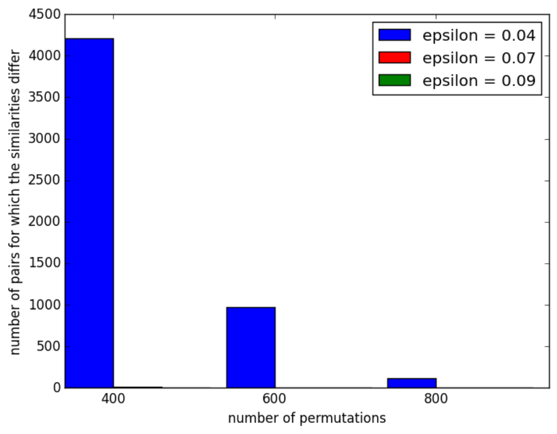

Figure 1: The average number of pairs on 10 runs for which approximate and exact similarities differ by more than ϵ for different **`numPermutations`**

Table 1: The average number of pairs on 15 runs for which approximate and exact similarities differ by more than ϵ for different **`numPermutations`**

	`numPermutations`		
ϵ	400	600	800
0.04	7077.4	1225.07	738.8
0.07	15.0	0.2	0.0
0.09	0.33	0.0	0.0

similarities differ by more than ϵ is recorded. Fig. 1. shows the results from running the program for 10 different times, each of which used different sets of permutation functions. As shown in the figure, the higher the permutation number is, the less the number of pairs for which the similarities differ by more than error thresholds. The same trend presents in an experiments where another 15 runs were conducted. The results are listed in Table 1. There are in total 500,500 pairs (out of the 1,001 files); if very low error threshold and rate results are desired, e.g., $\epsilon < 0.04$, then the number of permutations should be set to a higher number, for example, 800 permutations gives 0.15 error rate, which is tenth the error rate when permutation number is 400.

3: Total run time to calculate similarities

To test whether it is faster to estimate Jaccard similarities using MinHash matrix than to compute the exact ones, we built a class `MinHashTime` including one method.

`timer(String folder, int numPermutations)` gets a name of a folder and number of permutations as parameters (in that order). The method creates an instance of `MinHash`, and for every pair of files in the folder, computes the exact Jaccard Similarity; this task is timed (in seconds). Then it computes the MinHash matrix and use the matrix to estimate the Jaccard similarity of every pair of the documents; the time taken to compute the MinHash matrix and similarities are reported as well.

We used 600 as the permutation number on the files in the folder ***space***. The total run time to calculate the exact and estimated Jaccard similarities between all possible pairs were recorded. Fig. 2. plots the average run time on different operations under three permutations with 50 runs. Calculating the exact Jaccard similarities is almost the same for each permutation number as it made a fixed number of comparisons during the operation and would not be affected by the number of permutations. In contrast, calculating approximate Jaccard similarities involved different number of comparison operations for different permutation number choices but not the number of words appears in the documents. Thus, as the MinHash matrix becomes larger due to **`numPermutations`**, it engaged more comparisons for any pair of documents in the collection, leading to more time on constructing the matrix and calculations. This can be verified in Table. 2 presents the results for a 150-run experiment.

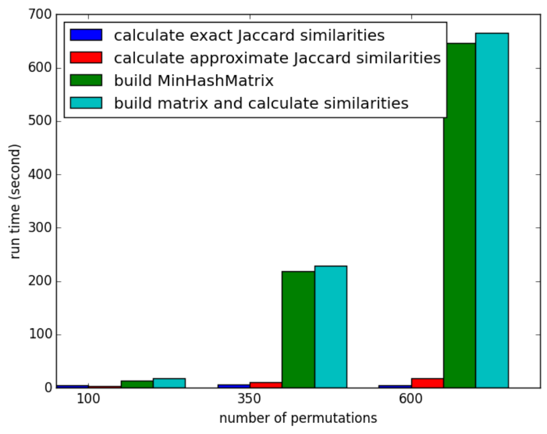

Figure 2: The run time for operations under different **numPermutations**

Table 2: The comparison times and run time under different **numPermutations**

	numPermutations		
	150	350	600
Comparisons for exact	9.3480652×10^7		
Comparisons for estimated	5.005×10^7	1.75175×10^8	3.003×10^8
Time for exact	5.47	5.45	4.68
Time for estimated	2.93	10.40	18.37
Time for mattrix	14.17	208.97	654.31

It is worth to mention that the number of comparisons drawn depends on the implementation. For the exact similarity, we used *HashSet* to store the terms appearing in a document, and check whether it also is in another document by calling its **contains** method. It is more efficient than using *List*. On the other hand, if each document contained a larger number of terms, the comparison for exact similarities would increase; as far as the similarity concern, for a choice with higher number of permutation, it could take fewer time to calculate the approximate ones than the exact ones (like we what we see when the permutation number is 100 in this case).

4: Implementation of locality sensitive hashing

To detect near duplicates of a document, we perform locality sensing hashing. Given a $K \times N$ MinHash matrix M, consider the following: for a given b, divide the rows of M into b bands, each of which consists of $r = \frac{k}{b}$ rows and create b hash tables $T_1, \dots, T_b$. For each document D_i, let **sig** be its MinHash signature, an array of k integers. Divide **sig** into b bands (r entries each), and compute the hash value of each band. If the j-th band of **sig** is hashed to t, then store the document (name) D_i at $T_j[t]$.

To accomplish this, we implemented the following constructor and methods in the class `LSH`.

`LSH(int[][] minHashMatrix, String[] docNames, int bands)` constructs an instance of `LSH`, where **`docNames`** is an array of *String* consisting of names of documents in the collection, and **`minHashMatrix`** is the MinHash matrix of the document collection and **`bands`** is the number of bands (b) for performing locality sensitive hashing.

To implement `LSH`, we need to create b (hash) tables, where b is the number of bands. Though conceptually this is simple, this (may) present(s) a few implementation challenges, such as what elements to store in the tables to represent the document objects, how to add them into the tables, how to decide where to store them and how to retrieve the similar items. There may be more efficient ways, for simplicity, we chose a nested `List` to address such challenges. To be precise, for each band, we create a hash table implemented with *ArrayList* where for each bucket, another `ArrayList<String>` is set up for storing the similar objects, that is file names in our case. The following is a pseudocode for the method **`localitySensitiveHashing`** in the class that accomplish this task.

```
initialize hashTables with ArrayList<List<List<String>>>
for each band of the MinHash matrix
    instantiate an ArrayList<List<String>> $T_j$ and add it to hashTables[5]
    instantiate $N'$ ArrayList<String>'s and add them to $T_j$ in sequence[6]
        for each document in the collection
            hash the band of the document to get the location $t$
            add the name of the document to the $t$-th List of $T_j$
        end
end
```

Again, we use the class `PermutationsFunctions` to implement the hashing algorithm. The hashing functions are randomly generated from the class with two arguments, **`numOfRowsPerBand`** and **`docNames.length`**; equivalently the former is r and the latter is N. As described in the class `MinHash`, two things are necessary to get from this class: the hash table size $p = N'$ and the *Map* that contains a r-tuple of a's and b's (for r rows in each band). Since hashing a fixed-length tuple, we can interpret the input as a vector $\overline{x} = (x_0, \ldots, x_{r-1})$. Then the concept of hashing vectors is considered, and we use the following as the hashing algorithm, $h(\overline{x}) = (\sum_{i=0}^{r-1} h_i(x_i)) \mod p$, where $h_i(x_i) = (a_i x_i + b_i) \mod p$. The idea is based on `https://en.wikipedia.org/wiki/Universal_hashing`. To produce integers between 0 and $p - 1$, in our implementation, the code turns $(a_i x_i + b_i)$ into a nonnegative integer by masking with `0x7fffffff` and then computes the remainder when dividing by p, before modular hashing each time.

nearDuplicatesOf(String docName) takes a name of a document, **`docName`** as parameter and returns an array list of names of the near duplicate documents. The routine is as follows. A *Set* is created to store the file names that are retrieved from the *List*'s.

```
instantiate nearDuplicates with LinkedHashSet<String>
find the minHashSig of docName from the minHashMatrix
for each band of docName
    hash the band of the document to get the location $t$
    retrieve all the objects in the List at location $t$ of the hash table of the band and store them
    into nearDuplicates
end
```

5: Near duplicates in a document collection detection

To detect near duplicates in a document collection, we designed a class `NearDuplicates` that puts together `MinHash` and `LSH`. This class have the following methods.

`nearDuplicateDetector(String folder, int numPermutations, int numBands, double s, String docName)` gets the following information as parameters (in that order): a name of the folder

[5]The hash table of band j can later be accessed by obtaining j-th *List* of **`hashTables`**.

[6]N' is the size of the hash table for each T_j, which will be a closet prime number large than or equal to N, the document number.

Queried file	Returned files
`baseball846.txt.copy3`	`baseball846.txt.copy7, baseball846.txt, baseball846.txt.copy4, baseball846.txt.copy5`
`baseball543.txt.copy5`	`baseball543.txt.copy4, baseball543.txt.copy7, baseball543.txt, baseball543.txt.copy2, baseball543.txt.copy6, baseball543.txt.copy1, baseball543.txt.copy3`
`hockey154.txt`	`hockey154.txt.copy1, hockey154.txt.copy4, hockey154.txt.copy6, hockey154.txt.copy3, hockey154.txt.copy2, hockey154.txt.copy5, hockey154.txt.copy7`
`hockey938.txt.copy3`	`hockey938.txt.copy4, hockey938.txt, hockey938.txt.copy7, hockey938.txt.copy1, hockey938.txt.copy2, hockey938.txt.copy6, hockey938.txt.copy5`
`space-480.txt.copy5`	`space-480.txt.copy2, space-480.txt.copy6, space-480.txt.copy7, space-480.txt, space-480.txt.copy3, space-480.txt.copy1, space-480.txt.copy4`
`hockey987.txt`	`hockey987.txt.copy7, hockey987.txt.copy6, hockey987.txt.copy4, hockey987.txt.copy2, hockey987.txt.copy1, hockey987.txt.copy3, hockey987.txt.copy5`
`space-390.txt.copy5`	`space-390.txt.copy2, space-390.txt, space-390.txt.copy6, space-390.txt.copy1, space-390.txt.copy7, space-390.txt.copy4, space-390.txt.copy3`
`baseball886.txt.copy6`	`baseball886.txt.copy1, baseball886.txt.copy4, baseball886.txt, baseball886.txt.copy5, baseball886.txt.copy2, baseball886.txt.copy7, baseball886.txt.copy3`
`space-638.txt.copy4`	`space-638.txt.copy2, space-638.txt, space-638.txt.copy5, space-638.txt.copy1, space-638.txt.copy3, space-638.txt.copy7, space-638.txt.copy6`
`hockey272.txt.copy5`	`hockey272.txt.copy1, hockey272.txt.copy3, hockey272.txt.copy2, hockey272.txt, hockey272.txt.copy4, hockey272.txt.copy6, hockey272.txt.copy7`

Table 3: Near-duplicate list of 10 different queried files

containing the documents, number of permutations for `MinHash`, number of bands in `LSH`, similarity threshold, and name of a document from the collection. This method first computes the list documents that are at least **`s`**-similar to **`docName`** by calling the method **`nearDuplicatesOf`**. As this list may contain some false positives - documents that are less than **`s`**-similar to **`docName`**, the MinHash matrix is used to eliminate the false positives. The method **`getLessThanThresholdDocs`** in the class is designed for this task. It calculates the estimated Jaccard similarities between the designated file **`docName`** and all the near duplicate files obtained from the method **`nearDuplicateOf`**; if the estimated similarity is lower than the threshold **`s`**, the file is a false positive. The false positives from **`nearDuplicatesOf`** are removed, and the remaining files are then returned by **`nearDuplicateDetector`**.

We ran **`nearDuplicateDetector`** on the files from ***F16PA2*** with different inputs and for each input, recorded the returned near duplicates. For each original file in the folder, 7 near duplicate documents were created for the experiments (similarity around 0.96). If taking a look at the names of the files, for example, we can see for file `space-0.txt`, `space-0.txt.copy1`, `space-0.txt.copy2`,. . ., `space-0.txt.copy7` are near duplicates. So if we run **`NearDuplicates`** with input `space-0.txt` and 0.9 as threshold, for appropriate choice number of permutations and bands, the program should ideally minimally output all of `space-0.txt`, `space-0.txt.copy1`, `space-0.txt.copy2`,. . ., `space-0.txt.copy7` as near duplicates. In the following we show that for different choices of parameters, the output near duplicate number varies.

Table. 3. lists all the files that are returned as near duplicates for 10 different inputs, running the program with parameters **`numPermutations`** = 1200, **`numBands`** = 100 and threshold **`s`** = 0.9. Except for `baseball846.txt.copy3`, all the true near duplicates were obtained in the end. Looking for the cause, we realize it came from the approximate Jaccard similarity. The three other near duplicates (i.e., `baseball846.txt.copy7, baseball846.txt, baseball846.txt.copy4, baseball846.txt.copy5`) were all presented as well by method **`nearDuplicatesOf`**. However, when estimating the Jaccard similarities with `baseball846.txt.copy3`, the approximated similarities were below the set threshold 0.9; therefore, these three files were removed from the list of near duplicates as false positives. Overall for this test case, the average returned number of near duplicated is 6.7, fairly close to the ideal number 7. Intuitively, to improve this, one may try a higher number of permutations to lower the error rate.

We then conducted a set of experiments where the number of permutations was set to 1000 along with three different numbers of bands considered; for each band number, 700 files were randomly picked from the folder and queried for their near duplicates, and we recorded the following: number of near duplicate files (dup.), false positives (fps.), false positive rate (fpr.), false negatives (fns.), and false negative rate (fnr.). The false positives are the dissimilar pairs of items that hash to the same buckets; the respective rate is the ratio of false positives over the total pairs that hash to the same buckets. We hope that the false positives will be only a small fraction of all pairs. In contrast, the truly similar pairs do not hash to the same bucket under at least one of the hash functions are false negatives; its rate is the ratio of false negatives over the total pairs that do not hash to the same bucket for any hash function. We hope the false negatives will

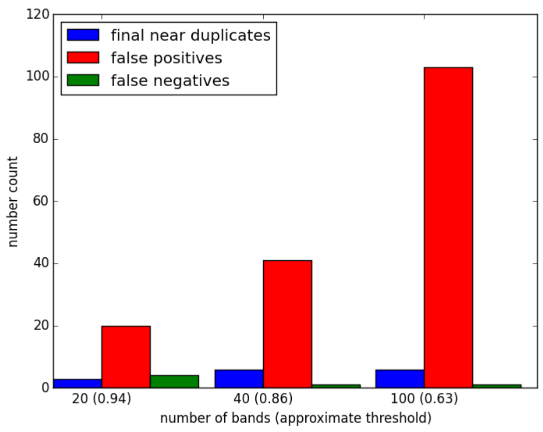

Figure 3: Performance of near duplicate detection with **numPermutations** equal to 1000

Table 4: Performance of near duplicate detection with **numPermutations** equal to 1200

# band	15 (0.97)	20 (0.95)	30 (0.92)	60 (0.81)	100 (0.68)	120 (0.62)
dup.	1.04	2.35	4.13	5.43	5.72	5.85
fps.	15.05	19.86	30.79	60.73	101.94	123.5
fpr.	0.94	0.90	0.88	0.92	0.95	0.95
fns.	5.96	4.65	2.87	1.57	1.28	1.15
fnr. (10^{-5})	24.9	19.4	12.0	6.57	5.35	4.84

be only a small fraction of the truly similar pairs. The threshold **s** is 0.9.

Fig. 3. plots the numbers of final near duplicates, false positive and negatives. The approximate threshold is listed next to the number of bands picked. For example, if $b = 20$, then $r = 50$ and the threshold is approximately at 0.94, which is $\frac{1}{20}^{\frac{1}{50}}$. In this case, $1 - (0.94)^50$ is about 0.95. If we raise this number to the 20th power, we get about 0.358. Subtracting this fraction from 1 yields 0.642. That is, if we consider two documents with 94% similarity, then in any one band, they have only about a 5% chance of agreeing in all 50 rows and this becoming a candidate pair. However, there are 20 bands and thus 20 chances to become a candidate. Roughly one in 1.6 pairs that are as high as 94% will fail to become a candidate pair and thus be a false negative. As the band number increases and the row number per band decreases, the false negatives become less as can be observed in the figure. In contrast, the false positives also go up along with the near duplicates.

This manifest that the approach can produce false negatives - pairs of similar documents that are not identified as such because they never become a candidate pair. There will also be false positives - candidate pairs that are evaluated, but are found not to be sufficiently similar. We further performed another set of experiments where the number of permutation was raised to 1200, and more band numbers were considered. We randomly picked 110 documents from the folder and find their near duplicated. The average of the numbers are listed in Table 4. We observe that if false positives is important, we may wish to select b and r to produce a threshold lower that **s**; if we wish to limit false positives, select b and r to produce a higher threshold. It is north to notice that when the threshold is higher, it results in more false positives, spending more time on examining each candidate pair's signatures.