Lab 1 - Locality Sensitive Hashing - Javier-G Hamed-M

November 12, 2017

```
In [1]: %reload_ext watermark
        %watermark -a 'Antonio Javier González Ferrer & Hamed Mohammadpour' -v -d -r
```

```
Antonio Javier González Ferrer & Hamed Mohammadpour 2017-11-12

CPython 3.6.3
IPython 6.1.0
Git repo: git@github.com:jgonzalezferrer/locality_sensitive_hashing.git
```

1 Introduction

In this notebook we are going to see the different stages of finding textually similar documents based on Jaccard similarity using the shingling, minhashing, and locality-sensitive hashing (LSH) techniques and corresponding algorithms.

First we are going to load the data and then run and visualize the following classes: - Shingling - MinHash - LSH

Let's get started by importing all the necessary libraries in the first cell:

```
In [2]: from locality_sensitive_hashing.utility import compare_sets, compare_signatures
        from locality_sensitive_hashing.shingling import Shingling
        from locality_sensitive_hashing.minhashing import MinHashing
        from locality_sensitive_hashing.lsh import LSH

        from tqdm import tqdm, trange, tqdm_notebook # Printing progress bar
        import json
```

1.1 Helper functions

```
In [3]: # For printing maps and dictionaries in sorted, beautiful format
        def printify(my_dict):
            print(json.dumps(my_dict, indent=1))
```

1.2 The data

The data we have used to test the algorithms contains piece of news where the goal is to detect plagiarism between them and has been extracted from the School of Informatics of The University of Edimburgh.

1.2.1 Load the data

```
In [4]: # You can run this code for different portions of the dataset.
        # It ships with data set sizes 100, 1000, 2500, and 10000.
        num_docs = 100
        data_file = "./data/articles_" + str(num_docs) + ".train"
        truth_file = "./data/articles_" + str(num_docs) + ".truth"
```

1.2.2 1. Parse The Ground Truth Tables

We first build a dictionary mapping the document IDs to their plagiaries, and vice versa for testing the performance of the algorithms:

```
In [5]: plagiaries = {}

        # Open the truth file.
        with open(truth_file, "r") as f:
            for line in f:

                # Strip the newline character, if present.
                if line[-1] == '\n':
                    line = line[0:-1]

                docs = line.split(" ")

                # Map the two documents to each other.
                plagiaries[docs[0]] = docs[1]
                plagiaries[docs[1]] = docs[0]

        printify(plagiaries)
```

```
{
 "t1088": "t5015",
 "t5015": "t1088",
 "t1297": "t4638",
 "t4638": "t1297",
 "t1768": "t5248",
 "t5248": "t1768",
 "t1952": "t3495",
 "t3495": "t1952",
 "t980": "t2023",
 "t2023": "t980"
}
```

1.2.3 2. Compute k-shingles of all documents

The first approach is to convert the documents into sets. We split the words of each document into k different shingles. In this case, a k-shingle is a sequence of k consecutive characters that appears within the documents. We assume we do not have repeated shingles into one document.

The value of k should be large enough specially for large documents. We have set this value to $k = 9$. Furthermore, while creating of k-shingles, we use a hash function to compress them to a 4-byte integer. Therefore, the range of shingles will be from 0 up to $2^{32} - 1$.

```
In [6]: %%time
        k = 9 # k-shingle hyper-parameter

        doc_list = []
        shingle_set = {}

        with open(data_file, "r") as f:

            for i in tqdm_notebook(range(num_docs)):
                document = f.readline()
                doc_id, doc_body = document.split(" ", 1)

                doc_list.append(doc_id)
                shingle_set[doc_id] = Shingling(doc_body, k).shingles
```

```
A Jupyter Widget
```

```
CPU times: user 256 ms, sys: 28.3 ms, total: 284 ms
Wall time: 287 ms
```

1.2.4 3. Compute Jaccard similarity

In this section we will compute Jaccard similarity for shingles with the utility function for later comparison with MinHash and LSH accuracy. The Jaccard similarity is defined as:

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

Therefore, two documents A and B will be similar is they have common shingles.

```
In [7]: %%time

        # Run test if the number of docs is small
        jaccard_threshhold = 0.60
        similar_docs = {}
        if num_docs <= 2500:
            print("Comparing {} documents -> {} comparisons..."
                  .format(num_docs, num_docs*(num_docs-1)//2))

            for i in tqdm_notebook(range(num_docs), desc='1st loop'):
                shingle_i = shingle_set[doc_list[i]]
```

```
                for j in range(i+1, num_docs):
                    shingle_j = shingle_set[doc_list[j]]
                    jaccard_sim = compare_sets(shingle_i, shingle_j)
                    if jaccard_sim >= jaccard_threshhold:
                        similar_docs[doc_list[i]] = doc_list[j]
                        similar_docs[doc_list[j]] = doc_list[i]
                        print("Documents {} and {} have jaccard sim. of {:.4f}"
                              .format(doc_list[i], doc_list[j], jaccard_sim))

        printify(similar_docs)
```

```
Comparing 100 documents -> 4950 comparisons...


A Jupyter Widget


Documents t980 and t2023 have jaccard sim. of 0.9840
Documents t1088 and t5015 have jaccard sim. of 0.9870
Documents t1297 and t4638 have jaccard sim. of 0.9850
Documents t1768 and t5248 have jaccard sim. of 0.9857
Documents t1952 and t3495 have jaccard sim. of 0.9826

{
 "t980": "t2023",
 "t2023": "t980",
 "t1088": "t5015",
 "t5015": "t1088",
 "t1297": "t4638",
 "t4638": "t1297",
 "t1768": "t5248",
 "t5248": "t1768",
 "t1952": "t3495",
 "t3495": "t1952"
}
CPU times: user 1.06 s, sys: 69.7 ms, total: 1.13 s
Wall time: 1.18 s
```

1.3 MinHash

Comparing all possible combinations of k-shingles of documents for large number of documents can take a very long time or become totally incomputable. For this reason we use MinHash algorithm which creates a unique hash of fixed length (from a number of hash functions n) so all documents get a signature of length n.

1.3.1 4.1 Compute MinHash of documents

4.1.1 Finding hash functions For generating n random hash functions, we used the universal hashing method inspired from here:

$$h_{a,b}(x) = ((ax + b) \bmod p) \bmod m$$

where a and b are random numbers between 1 and $ m = 2 ^{32} - 1$, c is a prime number larger than m, and m is the maximum possible value.

In [8]:
```
%%time

        n = 100 # Number of minhash hash functions
        minhash_doc_list = []

        for i in tqdm_notebook(range(num_docs),
                               desc="Calculating min hash for {} documents"
                               .format(num_docs)):
                shingles_i = shingle_set[doc_list[i]]
                minhashing_i = MinHashing(shingles_i, n)
                minhash_doc_list.append(minhashing_i.signature)
```

```
A Jupyter Widget
```

```
CPU times: user 8.95 s, sys: 209 ms, total: 9.16 s
Wall time: 9.53 s
```

Building the signatures takes some time but after you have calculated once you don't need to calculate it again. Now we have reduced large sets to short signatures, while preserving similarity. The idea behind MinHashing is that given two documents S_1, S_2, then $Pr[h_{min}(S_1) = h_{min}(S_2)] = J(S_1, S_2)$. Therefore, the expeced similarity of two signatures is equal to the Jaccard similarity of the two documents. The more hash functions (the longer the signatures) we use, the smaller will be the expected error.

4.2 Compare all document pairs for finding duplicates In this section we will compare the minhash of documents which we calculated in previous part to find the similar documents. This comparison takes less than 60ms for 100 documents where as Jaccard similarity took around 8000ms.

In [9]:
```
%%time

        minhash_threshhold = jaccard_threshhold

        for i in tqdm_notebook(range(num_docs), desc='1st loop'):
                minhash_i = minhash_doc_list[i]

                for j in range(i+1, num_docs):
                    minhash_j = minhash_doc_list[j]
```

```
                    minhash_sim = compare_signatures(minhash_i, minhash_j)
                    if minhash_sim >= minhash_threshhold:
                        print("Documents {} and {} have minhash sim. of {:.4f}"
                              .format(doc_list[i], doc_list[j], minhash_sim))
```

```
A Jupyter Widget
```

```
Documents t980 and t2023 have minhash sim. of 1.0000
Documents t1088 and t5015 have minhash sim. of 0.9800
Documents t1297 and t4638 have minhash sim. of 0.9700
Documents t1768 and t5248 have minhash sim. of 0.9800
Documents t1952 and t3495 have minhash sim. of 0.9800

CPU times: user 93 ms, sys: 23.5 ms, total: 116 ms
Wall time: 102 ms
```

1.4 LSH

However, we still have the problem of scalability and the large number of comparisons. The Locality Sensitive Hashing (LSH) solves this problem by generating pairs of candidate documents which a large likelihood of being similar. To do this, we will pass a dictionary of document ids and a list of minhash signature for each document. Given a similarity threshold t, the algorithm will divide this signature matrix M into b bands and r rows and will hash columns of M with the idea of arranging similar columns to same buckets. The key thing here is to choose the best r and b values in order to catch most similar pairs, but few non-similar ones.

In the LSH class, we calculate the best number of r and b by iterating through the factors of n which is the number of signatures and choose the ones which are approximate closer to $ t = (1/b) ^{(1/r)} $:

```
In [10]: %%time

         lsh_threshold = 0.8
         doc_signatures_dict = dict(zip(doc_list, minhash_doc_list))

         lsh_sim = LSH(doc_signatures_dict,
                       lsh_threshold,
                       minhash_threshhold).similar_pairs
```

```
The values for b is 10 and r is 10.
 The best approximation for threshold 0.8 is t = 0.7943282347242815
CPU times: user 4.14 ms, sys: 1.29 ms, total: 5.43 ms
Wall time: 4.32 ms
```

```
In [11]: lsh_sim
```

```
Out[11]: {'t1088': 't5015',
          't1297': 't4638',
          't1768': 't5248',
          't1952': 't3495',
          't2023': 't980',
          't3495': 't1952',
          't4638': 't1297',
          't5015': 't1088',
          't5248': 't1768',
          't980': 't2023'}
```

As we see it can find similar documents among 100 document in less than 4.2 ms in comparison to 80 ms of minhash algorithm and ~8000ms using raw Jaccard similary.

1.5 Going with larger documents

Now we are ready to test our implementation on a bigger sets. You can set `num_docs` to larger number to see the performance. In our tests with 10,000 documents, the minhashing process took around 16 min. Then comparing the signatuares one by one would take around 20 minutes while using LSH this time reduces to 1.14 seconds.

2 Final thoughts

In this notebook we demonstrated the performance and API of our implementation of Local Sensitive Hashing for finding similar documents.

1. Surprising enough, the LSH method reports 100% accuracy even testing with 10,000 documents.
2. For calculating number of bands and rows for `LSH`, we iterate over factors of `n` to find best `b` and `r` to approximate as much as we can.

Our code among with report and data files are published in the following repository on github.

```
In [12]: %reload_ext watermark
         %watermark -a 'Antonio Javier González Ferrer & Hamed Mohammadpour' -v -d -r

Antonio Javier González Ferrer & Hamed Mohammadpour 2017-11-12

CPython 3.6.3
IPython 6.1.0
Git repo: git@github.com:jgonzalezferrer/locality_sensitive_hashing.git
```