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

November 12, 2017

```
In [1]: %reload_ext watermark
        %watermark -a 'Javier González Ferrer & Hamed Mohammadpour - KTH Royal Institute of Te

Javier González Ferrer & Hamed Mohammadpour - KTH Royal Institute of Technology 2017-11-12

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

1 Locality Sensitive Hashing Demo

In this notebook we are going to see our implementation of LSH step by step.

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

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

```
In [2]: import locality_sensitive_hashing.utility as utility
        import locality_sensitive_hashing.shingling as shingling
        import locality_sensitive_hashing.minhashing as minhashing
        import locality_sensitive_hashing.lsh as 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 following data is acquired from here

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.
        numDocs = 100
        dataFile = "./data/articles_" + str(numDocs) + ".train"
        truthFile = "./data/articles_" + str(numDocs) + ".truth"
```

1.2.2 1. Parse The Ground Truth Tables:

Build a dictionary mapping the document IDs to their plagiaries, and vice versa.

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

        # Open the truth file.
        f = open(truthFile, "r")

        # For each line of the files...
        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]

        # Close the data file.
        f.close()

        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 we have

While creating of k-shingles, we also use a hash functions to convert them to a 4-bit long integer.

```
In [6]: %%time
        k = 5 # k-shingle hyper-parameter

        doc_list = []
        shingle_set = {}

        f = open(dataFile, "r")

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

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

        # Close the data file.
        f.close()
```

```
A Jupyter Widget



CPU times: user 303 ms, sys: 34.9 ms, total: 338 ms
Wall time: 481 ms
```

1.2.4 3. Compute Jaccard similarity

In this section we will compute Jaccard similarity with the utility function for later comparison with MinHash and LSH accuracy.

Jaccard similarity is defined as:

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

```
In [7]: %%time

        # run test if the number of docs is small
        jaccard_threshhold = 0.60
        similar_docs = {}
        if numDocs <= 2500:

            for i in tqdm_notebook(range(numDocs), desc='1st loop'):
                set1 = shingle_set[doc_list[i]]

                for j in range(i+1, numDocs):
                    set2 = shingle_set[doc_list[j]]
                    jaccard_sim = utility.compare_sets(set1, set2)
                    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)
```

```
A Jupyter Widget


Documents t980 and t2023 have jaccard sim. of 0.9901
Documents t1088 and t5015 have jaccard sim. of 0.9916
Documents t1297 and t4638 have jaccard sim. of 0.9902
Documents t1768 and t5248 have jaccard sim. of 0.9901
Documents t1952 and t3495 have jaccard sim. of 0.9869

{
 "t980": "t2023",
 "t2023": "t980",
 "t1088": "t5015",
 "t5015": "t1088",
 "t1297": "t4638",
 "t4638": "t1297",
 "t1768": "t5248",
 "t5248": "t1768",
 "t1952": "t3495",
 "t3495": "t1952"
}
CPU times: user 1.05 s, sys: 73.2 ms, total: 1.12 s
Wall time: 1.21 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 lenght (from number of hash functions n) so all documents get a signiture 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 following 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 which is maximum value.

```
In [8]: %%time

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

        for i in tqdm_notebook(range(numDocs),
                               desc="Calculating min hash for {} documents".format(numDocs)):
                set1 = shingle_set[doc_list[i]]
                min1 = minhashing.MinHashing(set1, n)
                minhash_doc_list.append(min1.signature)
```

```
A Jupyter Widget



CPU times: user 8.88 s, sys: 280 ms, total: 9.16 s
Wall time: 9.75 s
```

4.2 Compare all document pairs for finding duplicates In this section we will compare the minhash of documnts which we calculated in previous part to find the similar documents. This comparison takes less than 80 ms for 100 documents where as Jaccard similarity took around 9s.

```
In [9]: %%time

        minhash_threshhold = jaccard_threshhold

        for i in tqdm_notebook(range(numDocs), desc='1st loop'):
                min1 = minhash_doc_list[i]

                for j in range(i+1, numDocs):
                    min2 = minhash_doc_list[j]

                    minhash_sim = utility.compare_signatures(min1, min2)
                    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 0.9900
Documents t1297 and t4638 have minhash sim. of 1.0000
Documents t1768 and t5248 have minhash sim. of 0.9900
Documents t1952 and t3495 have minhash sim. of 1.0000
```

```
CPU times: user 70.2 ms, sys: 14.1 ms, total: 84.3 ms
Wall time: 79.4 ms
```

1.4 LSH

So we arrive to the fun part of the assigment, to find similar documents using Locality Sensitive Hashing.

For this, we will pass a dictionary of Document ids and a list of minhash signiture for each document. In the LSH class, we calculate the best number of `r` and `b` by iterating through factors of `n` which is number of signitures and choose the ones which appriximate closer to $ t = (1/b) ^{(1/r)} $

```
In [10]: %%time

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

         lsh_sim = lsh.LSH(doc_signitures_dict, lsh_threshold).similar_pairs
```

```
b is 10 and r is 10, And best approximation for threshold 0.8 is t = 0.7943282347242815
10 10
CPU times: user 4.72 ms, sys: 1.73 ms, total: 6.45 ms
Wall time: 5.47 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 6 ms in comparison to 80 ms of minhash algorithm and ~9s of 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 `numDocs` to larger number to see the performance. In our tests with 10,000 documents, the minhashing process took around 16 min. Then comparing the signituares 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, we got 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 'Javier González Ferrer & Hamed Mohammadpour' -v -d -r
```

```
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
```