Locality Sensitive Hashing (LSH)

Benjamin Nguyen

***Abstract*—Locality Sensitive Hashing (LSH) provides an efficient approach to similarity search. It has countless applications in any field that works with large dataset comparisons. This paper will depict the research and code implementation of a traditional approach of LSH. Each step of the traditional approach will be explored in regard to input, output, functionality, and code implementation.**

Keywords—Locality Sensitive Hashing, similarity search, big data

I. Introduction

LSH is an algorithmic technique that can be used for data clustering and nearest neighbor search. The basis of LSH is to hash similar input items into subsets ("buckets"). The hash functions are designed to maximize collisions, given similar input items. The grouping of the input items into the smaller subsets reduces the time complexity of the similarity search to sub-linear. Additionally, the hash outputs significantly reduce the space complexity of the input items. A traditional approach to LSH involves a series of steps: Shingling, One-Hot Encoding, MinHashing, and Banding.

II. Motivation

Data science is a rapidly growing field with applications in nearly every industry. LSH (or in general, efficient similarity search) is important for countless companies. Social media and entertainment companies rely on similarity search to recommend related content to users. Music and video companies use similarity search when comparing audio or video fingerprints. Machines are able to "learn" through the use of nearest neighbors.

III. Code Implementation

This section will further explore the steps of the traditional approach of LSH, and the resulting code implementation.

A. Shingling

Input is provided as the original input items. Each item is separated into sets of characters of length *k*. *k* is determined by user and varies across application, being affected by size of input items and desired similarity threshold. Larger *k* values produce smaller sets of characters, which produce a greater chance for similarity between input items, which reduces the similarity threshold. *k* value of 8-10 is generally used in practice.

Input for the code implementation is statically provided by the user before the program is run. The chosen example input items are as follows: {*a*: "1170 is a good class." | *b*: "1170 is a great class." | *c*: "I wish to take a nap after the project."} By first glance, it is clear that *a* and *b* are similar, while *c* is not similar with either. For $k = 2$, the shingling of *a* would produce {"11", "17", "70", "0 "…"ss", "s."} The same process is applied for *b* and *c*. Similar shingles exist in both *a* and *b*, implying their similarity.

```
def shingle(text, k):
    shingle_set = []
    for i in range(len(text) - k + 1):
        shingle_set.append(text[i:i + k])
    return set(shingle_set)
```

To further quantify similarity, the Jaccard Index will be used. The Jaccard Index calculates the intersection of two sets over the union of the two sets. A value of 1 signifies perfect similarity (matching sets), while a value of 0 signifies no similarity (no overlap). The Jaccard indices between the input items are as follows {*a/b*: 0.64 | *a/c*: 0.0588 | *b/c*: 0.0577}. The similarity between *a* and *b* has now been effectively quantified.

```
def jaccard(a: set, b: set):
    return len(a.intersection(b)) / len(a.union(b))
```

B. One-Hot Encoding

As input items grow in size, the issue of space complexity arises. LSH addresses this issue by hashing the input items and creating signatures. Before hashing, the shingle sets are converted into sparse vectors using one-hot encoding.

The individual shingle sets are combined into a large shingle set that contains all shingles. A sparse vector is created for each input item. The size of each sparse vector is equal to the size of the combined shingle set. If a shingle in the combined shingle set exists in an individual shingle set, the sparse vector value is set to 1. Otherwise, the sparse vector value is set to 0 (i.e., the shingle existed exclusively in a different input item).

```
for shingle in combine_shingles:
    if shingle in a_shingle:
        a_1hot.append(1)
    else:
        a_1hot.append(0)
    if shingle in b_shingle:
        b_1hot.append(1)
    else:
        b_1hot.append(0)
    if shingle in c_shingle:
        c_1hot.append(1)
    else:
        c_1hot.append(0)
```

C. *MinHashing*

The sparse vectors are then converted into dense vectors (signatures). The size of the signature directly affects the space complexity of the algorithm. For each position in the signature, a MinHash function is randomly generated. Each MinHash function is a randomized order of numbers. The size of each MinHash function is equal to the size of the combined shingle set.

```
# create one MinHash function, randomized order of numbers
def create_hash_func(size):
    hash_ex = list(range(1, len(combine_shingles) + 1))
    shuffle(hash_ex)
    return hash_ex


# create MinHash function for each position in signature
def build_minhash_func(combine_size, signature_length):
    hashes = []
    for _ in range(signature_length):
        hashes.append(create_hash_func(combine_size))
    return hashes
```

Each MinHash function is used to produce the respective positional value within the signature for each input item. The order of numbers within the MinHash function is traversed incrementally, starting at 1 and ending at the length of the combined shingle set. If the given position in the one-hot encoded set is 1, the traversal ends, and the given position becomes the positional value within the item's signature.

```
# create signature for each sparse vector
def create_hash(vector: list):
    global count
    signature = []
    for func in minhash_func:
        if count == 1:
            print("MinHash function for first spot of a_sig:", func)
            print()
            count = 0

        for i in range(1, len(combine_shingles) + 1):
            idx = func.index(i)
            signature_val = vector[idx]
            if signature_val == 1:
                signature.append(idx)
                break
    return signature
```

It is important to note that, on average, the Jaccard index is not greatly changed by MinHashing. As the MinHash functions are randomly generated, however, the introduced error is not deterministic. The introduced error can be somewhat controlled using the length of the signature. Longer signatures introduce less error, but require additional space.

D. *Banding*

The signatures created from the MinHash functions are not likely to match across the entire signature. To lower the similarity threshold, banding is used. Each signature is split into a specified number of bands. Increasing the number of bands makes it easier for the smaller bands to match, decreasing the similarity threshold. Any matches signify the input items as candidate pairs, and the larger signatures can then be further evaluated.

```
def split_vector(signature, b):
    assert len(signature) % b == 0
    r = int(len(signature) / b)
    subvecs = []
    for i in range(0, len(signature), r):
        subvecs.append(signature[i: i + r])
    return subvecs
```

IV. Results and Discussion

The code implementation can be successfully run with any given input items. The output of each LSH function is displayed to the user for easier visualization. Candidate pairs will be produced at the end, which indicate similar input items that may be directly accepted or further examined.

The selection of appropriate parameters is very important to proper LSH functionality. Similarity threshold is inversely related with size of shingles, and directly related with number of bands. Similarity threshold that is too high may introduce false positives, but similarity threshold that is too low may introduce false negatives. Increasing the length of the signature reduces the introduced random error from MinHashing, at the cost of additional space.

V. Conclusion and Future Work

The code is functional in producing candidate pairs, but leaves a large space for additional testing and exploration. As previously discussed, it would be interesting to measure the effects of changing different LSH parameters. Additionally, the LSH function currently operates on only three input items. It should eventually be tested on a larger dataset. Lastly, there exist several implementations of LSH outside of the traditional approach (e.g., random projection). These approaches may be eventually implemented, and the time and space usage may be compared between different approaches.

Acknowledgment

Benjamin Nguyen acknowledges the University of Pittsburgh's Electrical and Computer Engineering Department for guiding him throughout his college career. He also acknowledges Dr. Amr Mahmoud for teaching a great ECE 1170 class (and ENGR 0012 and ECE 0301 and ECE 1195).

References

1. "A simple introduction to locality sensitive hashing (LSH)," *iunera*, 01-Mar-2022. [Online]. Available: https://www.iunera.com/kraken/fabric/locality-sensitive-hashing-lsh/. [Accessed: 02-Aug-2022].
2. "Locality sensitive hashing (LSH): The Illustrated Guide," *Pinecone*. [Online]. Available: https://www.pinecone.io/learn/locality-sensitive-hashing/. [Accessed: 02-Aug-2022].
3. S. Gupta, "Locality sensitive hashing," *Medium*, 01-Apr-2019. [Online]. Available: https://towardsdatascience.com/understanding-locality-sensitive-hashing-49f6d1f6134. [Accessed: 02-Aug-2022].