Locality Sensitive Hashing for Nearest Neighbour Search

Gaurav Joshi[1] **Adit Kaushik**[2] **Arun Reddy**[3] **Aditya Deshmukh**[3] **Rutwik More**[3]

[1] 21110065 [2] 21110010 [2] 21110029 [2] 21110014 [2] 21110133

Abstract

Finding the Nearest neighbour for a query from a large corpus of data is an essential but challenging problem. In this report we will describe how Locality Sensitive Hashing can be a useful method to find the approximate nearest neighbours, in a much faster time than simple linear search.

Contents

The contents of the paper are as follows:

Hashing

Hashing is a technique or process of mapping keys, and values into the hash table by using a hash function. It is done for faster access to elements. The efficiency of mapping depends on the efficiency of the hash function used. [1]

Hashing can also be viewed as taking in objects as inputs and then making a function to different buckets in which objects are kept in different buckets no matter how close the objects keys are. Analogically we can think of it like you are storing students grade information in shelves with their roll nos as their keys, we will keep the grades in different shelves for different roll nos. The hash function is the function that maps the keys to its hashes and the hash table is the corresponding table.

Common Hash functions:-

- **Identity Hash Function:** The identity hash function is used when the keys are integers in a small range, the key value is the index of the value in an array.
- **Mid-Square Hash Function:** A mid-squares hash code is produced by squaring the input and extracting an appropriate number of middle digits or bits. For example, if the input is 123,456,789 and the hash table size 10,000, squaring the key produces 15,241,578,750,190,521, so the hash code is taken as the middle 4 digits of the 17-digit number (ignoring the high digit) 8750. [2]

[1] https://www.geeksforgeeks.org/hashing-data-structure/

[2] https://en.wikipedia.org/wiki/Hash$_function$/

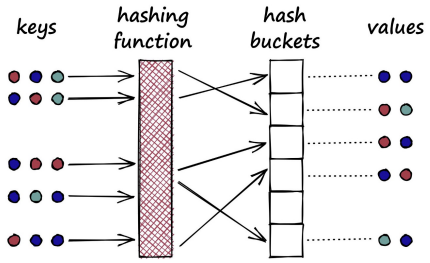

Figure 1: The normal hash function maps keys to hashes and tries to minimize clashes Source:- https://www.pinecone.io/learn/locality-sensitive-hashing/

- **Division Hash Function:** In this, we take modulo of the key value with a large prime number and use them as indexes. A drawback is that division is much slower than multiplication, in computers.
- **Multiplicative Hash Function:** In this, we use the hash formulae $h_a(k) = floor((aKmodW)/(W/M))$, which will return us a value from 1 to M. This is a useful method as when $W = 2^w$, and $M = 2^m$ (for w being the machine word size and m being any number) the modulo 2^w can be done by default, and the division is a simple left shift.
- **Zobrist Hash Function:** It is the most used hash function, and constructs families of hash functions using XOR and look up tables.

Properties of Good Hash functions:

- Uniformity The keys should be uniformly spread, not clustered together
- Efficiency The hashing operation should be done fast
- Universality The hash operations should be usable for many types of key inputs
- Variable Range One should be able to use keys of variable ranges

Locality Sensitive Hashing

Locality Sensitive hashing (LSH) is an algorithmic technique which hashes similar input items into the same buckets

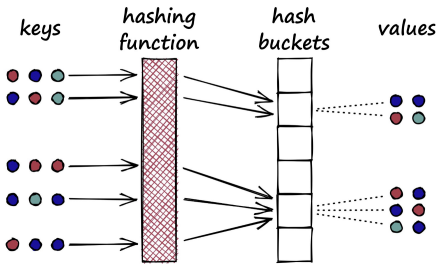

Figure 2: The locality sensitive hash function maps keys to hashes and tries to maximise clashes Source:- https://www.pinecone.io/learn/locality-sensitive-hashing/

with a high probability. (The set of buckets is much smaller than the set of possible input items) [3]

Mathematical basis of LSHs and ANNs We can formally state the problem of Approximate Nearest neighbour search as:-

Given a dataset D, with n points, and d dimensions. The goal of c-ANN is to find the neighbours in the c neighbourhood of query point q.

Let q be the query point, and o' be its nearest neighbour. Then we have to find the points such that:

$$dist(o, q) \leq c * dist(o', q)$$

In k-c-ANN, we try to find the top k points in this space

Hashing based methods try to form nearest neighbours in high dimensional dataset by projecting into low dimensional spaces using hash functions, (while maximising collisions)

For two points x, y in d Dimensional dataset, $D \in R^d$, the Hash Function H is (R, cR, P_1, P_2) sensitive if:

$$\text{if } |x - y| \leq R \text{ then } P[H(x) = H(y)] \geq P_1$$

$$\text{if } |x - y| \geq cR \text{ then } P[H(x) = H(y)] \leq P_2$$

As this technique hashes similar items to similar hashes, we can use it in data clustering and nearest neighbour search. One crucial way it differs from conventional hashing is that is it maximises the hash collisions while the hash functions try to minimise the collision.

To extend our analogy, if the schools wants to keep and compare the performance of students from similar financial background, we would want to use a LSH which will keep students with similar parental income in the same shelf.

Locality Sensitive hash function are built around the method with which we measure the amount of similarity between two objects. We measure the similarity with distances, the lower the distance between two objects (their representation as sets, vectors , bits) the closer they are in substance.

Popular LSH families for common distances:

- Hamming The hamming lsh is based on the hamming distance which gives the number of same bits between two items

  Eg: 110011 and 010100 have hamming distance 2

  The hash function we use is $H(x) = x_i$, which means the item is mapped to its i^{th} bit. This is a map from R^d to R

  If x and y are at a distance r, then:

  $$P[H(x) = H(y)] = 1 - r/d$$

- Minkowski The minkowski distance hash function is:-

  $$H_{\mathbf{a},b}(x) = \lfloor (\mathbf{a}.x + b)/w \rfloor$$

  where a is vector, b is an integer and w is the bin or bucket width. We can calculate the probability using

  $$P[H(x) = H(y)] = \int_0^w f_p(t/r)(1 - t/w)\, dt$$

- Angular The angular distance is measured by the angle θ between x and y We can take the hash function as

  $$H(x) = sign(\mathbf{a}.x) \text{ , where a is a vector}$$

  Then the probability of hash collision is:

  $$P[H(x) = H(y)] = 1 - r/d$$

- Jaccard The jaccard distance is the one which we will use in our LSH based query finding method. The jaccard distance is:

  $$J = (A \cap B)/(A \cup B)$$

Combining mulitple LSH's: We can combine multiple LSH hashing function using AND and OR operations and get more complex hash functions.

Approximate Nearest Neighbours

Finding nearest neighbors in high-dimensional spaces is an important problem in several diverse applications, such as multimedia retrieval, machine learning, biological and geological sciences, etc. [4]

For low dimensions (less no of features), data structures like KD trees and SR are effective and optimised for time, but for higher dimensions, (which we frequently get for representation of complex data) these become time intensive and often take more time than linear search due to curse of dimensionality. Instead of focussing on removing this curse, we try to obtain the approximate nearest neighbours of the objects. Here we are trading off accuracy for a higher performance, but as has been shown in various results, the approximate neighbours from a large corpus are actually quite good searches results so, the exact solutions are not required.

Vector Embeddings

Vector embeddings are arrays of number (int or float), which are representations of complex objects like audios, images, sentences, graphs etc. They carry most of the crucial information of the items they are representing.

For example, a good vector embedding of an sentence will contain its syntactical meaning, its usage of nouns, certain important words in the sentence, the emotion with which it is was said, etc. As we are converting complex sentence into say a vector of 200 floats, it will most likely loose out on

[3] Rajaraman, A.; Ullman, J. (2010). "Mining of Massive Datasets, Ch. 3

[4] https://arxiv.org/abs/2102.08942

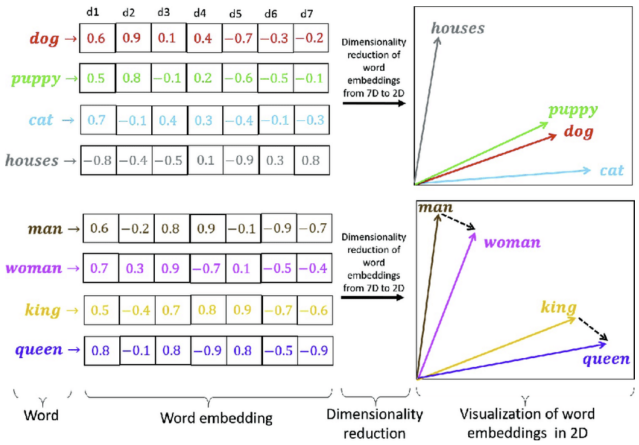

Figure 3: Vector embeddings help us to convert object into float arrays representative of the charachteristics of the objects Source:- Rozado, David,10.1371/journal.pone.0231189

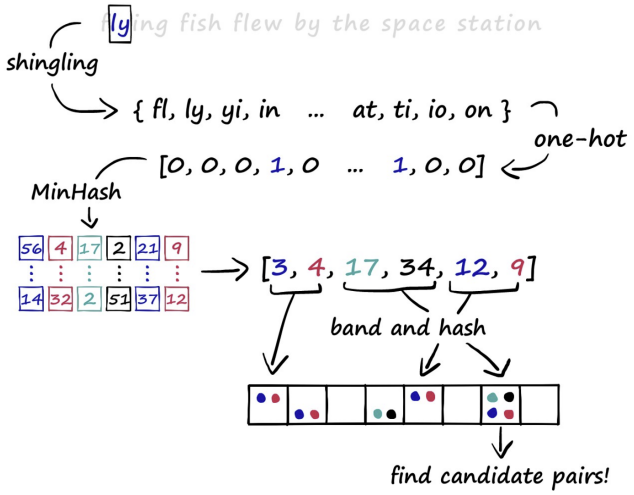

Figure 4: An overview of the process of using LSHs for query searching Source:- https://www.pinecone.io/learn/locality-sensitive-hashing/

some or the other information regarding the sentence, hence the embedding usually has to be made differently for different applications, like some which focus on grammatical correctness, some which focus on the sentences explicit meaning etc.

More Abstractly, the concept of vector embeddings is based on the Cantors theorem [5]. The theorem implies that R^n and R^m are of the same order, then we can make a bijective function which maps a vector in R^n to a vector in R^m. So, we can map a 728X728 float image to a 200 float vector.

One caveat of this idea is that these complex items are more than just the bit wise memory we require to store them. The pixels of the image are linked to each other and a flattened out array is not a very good way to represent it. Also, the images has edges, etc and features which are of more signinficance to the image than mere pixel values. Similary, texts are not just collection of random charachters but these charachters have different meanings and combine to words, which then combine to phrases, with different meanings.

This problem can be mitigated using machine learning methods like CNNs, RNNs which can take these features into account while making the embeddings. For example as the CNN, uses convolution filters on the images, it captures the details which are important to the image.

Implementing LSH for sentence finding

Locality Sensitive Hashing (LSH) is known for two main advantages: its sub-linear query performance (in terms of the data size) and theoretical guarantees on the query accuracy. Additionally, LSH uses random hash functions which are data-independent (i.e. data properties such as data distribution are not needed to generate these random hash functions). Additionally, the data distribution does not affect the generation of these hash functions. Hence, in applications where data is changing or where newer data is coming in, these hash functions do not require any change during runtime. [6]

LSH based query searching can be implemented to find close sentence searches among a large corpus of sentences. A direct use of this can be in search engines, say we input a sentence and the engine return similar sentences with links to their sources attached to them.

We have implemented a simple ANN query searcher which searches for matching sentences from a corpus of 14000 sentences. A sequential order in which we made it.

Order

- Shingling of sentences
- Minhashing and formation of signatures
- LSH and candidate pair formation
- Query making

At the very fundamental level, we will be converting the sentences into vector embeddings(signatures), then comparing the vector embeddings of the query sentence with our corpus of signatures and find the closest sentences to it.

Shingling of Sentences. k-Shingles is a word made by taking k consecutive characters from a sentence. Here we will be making a set of all the possible singles of each sentence, let n be the size of this set. Then we will remove the overlapping ones and create a dictionary of these shingles. We will shuffle the dictionary give each of its word a number. Finally for each sentences we will form an a sparse vector of size n which has a one for the index that matches with a shingle of the dictionary.

[5] https://www.britannica.com/science/Cantors-theorem

[6] https://arxiv.org/abs/2102.08942

```
text = "A boy is happily playing the piano"
# text = ""
ann = ANN(text, candidate_pairs, vocabulary, K,
    |   |   | no_of_splits, signature_array, sentences)
for i in ann:
    print(i)
```

```
A girl is playing the piano
The boy is playing in the mud
A kid is playing the piano
One boy is happily playing a piano
The boy is playing the piano
The boy isn't playing the piano happily
A boy is happily playing the piano
```

Figure 5: The result obtained on querying "A boy is happily playing the piano"

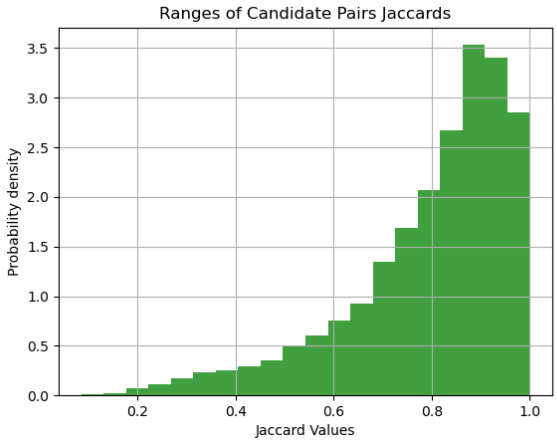

Figure 6: Histogram showing the probability density of the jaccards of all the candidate pairs formed

Minhashing and formation of signatures In Minhashing, we will take all the sparse vectors and make 100 length signatures out of them. For this we take 100 vectors of size n filled with numbers from 1 to n randomly. Then for each iteration, we will check which smallest number of the vector corresponds to a one in the same index in the sparse matrix. For example if the first vector has 1 at index 190, 2 at 487, the sentence vector has 0 at index 190 and 1 at 487, then the first element of ones signature will be 2, and we will keep on going for all 100 vectors. This will give us our signature.

LSH and candidate pair formation In Lsh, we would want to put all the vectors which are close enough to each other in the same bucket, and call them neighbours. A simple way of doing this is, we will split the signatures into m (20) subvectors, so each subvector will be of size 5. Then we will make a m (20) dictionaries and for each m, put the elements which has same subvector into the same bucket(candidate pairs). In this way we are maximising collisions when forming the hashing. The buckets are made using dictionaries so that we can find the required subvector fast, when we are querying the input.

Query Making We will take the query sentence and form its 100 length signature in the same way as for others, we will break it into m subvectors and for each subvector, we will check the respective dictionary to see if there is any matching subsignature. If yes, then we they all are close to the sentence and we can return them.

Results and Analysis

As we can see in the sample query in Fig 5, the search gives us many matching sentences (7 in the example), this is a good proof of concept that our simple implementation is working well, but there are many observations we can make:

1. The selected sentences have atleast a word or a set of consecutive words in common with our query
2. All sentence have the word piano, in them apart from the secound one, but it has the first half matching well with our input.
3. The hashing does not take into the semantics of English language, but if sentences are grammatically correct, then a few matching words usually means the sentences are quite similar.
4. The Figure 6, clearly shows that the formed candidate pairs mostly have above 0.5 jaccard values, which is what a good model should produce. The outliers most probably occur as collision can occur on matching of just 1 subvector out of 20 subvectors of the signatures.

Scope of Improvements in the Model

There are various direction in whihch we can improve this basic implementation of LSH based query search, some of them are:

1. The vector embedding used in our model is based only on the position of characters and does not take into account the complete words, their semantics and their positions. This model can be far improved by using a semantic sensitive embedding (eg. BERT), which make the searches grammatically consistent.
2. The sentences we take only have a registered 40,000 shingles which does not cover all the sentences that may come, we can increase the amount of data used to improve our searcher.
3. As we can see out of 14500 sentences, 13200 buckets were formed for each subvector, which is source of concern as it means that a high percent of sentences did not collide with any other, so this is a potential source of improvement.

Use of LSH in ML and industry

1. Lsh filters can be used for anomaly detection. The training data points are hashed into a set of buckets and the test data is marked as an anomaly if it is hashed to a different one.
2. Recommendation Systems: LSH can be used to build recommendation systems that suggest products or services to users based on their previous behavior. For example,

LSH can be used to find users who have similar preferences and recommend products that these users have purchased or viewed. Almost all big IT companies, online stores, social media platforms use it their platforms. (eg: FAISS by facebook)

3. LSH can be used to optimize the performance of smart contracts on a blockchain. By using LSH to store and retrieve data, smart contracts can process large amounts of information more quickly and with fewer computational resources.

Reformers

Reformer models are essentially transformer models that uses locality sensitive hashing instead of dot product attention. This is also called the Sparse Attention model where in instead of calculating attention between all the tokens, it is only calculated between the nearby tokens. We use LSH's to approximate the nearby tokens as those which fall in the same hash bucket.

Acknowledgments

We are grateful to Profesor Anirban Dasgupta for giving us his valuable time and guiding us in finding and learning more about these cutting edge technology that is currently transoforming many fields in computer science and other places.

References

Nagarkar Et.Al. 2021, A Survey on Locality Sensitive Hashing Algorithms and their Applications, arXiv:2102.08942v1

Indyk Et.Al. 2018, Approximate Nearest Neighbor Search in High Dimensions, arXiv:1806.09823v1

https://www.pinecone.io/learn/wild/

https://www.pinecone.io/learn/locality-sensitive-hashing/

https://huggingface.co/docs/transformers/modeldoc/reformer