Practical Applications of Locality Sensitive Hashing for Unstructured Data

Jake Drew
Computer Science and Engineering Department
Southern Methodist University
Dallas, TX, USA
www.jakemdrew.com
jakemdrew@gmail.com

Michael Hahsler
Engineering Management, Information, and, Systems Department
Southern Methodist University
Dallas, TX, USA
www.michael.hahsler.net
mhahsler@smu.edu

Introduction

Working with large amounts of unstructured data (e.g., text documents) has become important for many business, engineering and scientific applications. The purpose of this article is to demonstrate how the practical Data Scientist can implement a Locality Sensitive Hashing system from start to finish in order to drastically reduce the time required to perform a similarity search in high dimensional space (e.g., created by the terms in the vector space model for documents). Locality Sensitive Hashing dramatically reduces the amount of data required for storage and comparison by applying probabilistic dimensionality reduction. In this paper we concentrate on the implementation of min-wise independent permutations (MinHashing) which provides an efficient way to determine an accurate approximation of the Jaccard similarity coefficient between sets (e.g., sets of terms in documents) [2,3].

This article explains the concept of Locality Sensitive Hashing in practical terms by:

1. Providing a very brief academic history
2. Explaining Locality Sensitive Hashing's general purpose and possible use cases
3. Introducing Minhashing
4. Describing the Jaccard similarity coefficient as an LSH measure of similarity
5. Demonstrating how to implement a simple Minhashing LSH system using C#

A Very Brief Academic History

The concept of Locality Sensitive Hashing has been around for some time now with publications dating back as far as 1999 [1] exploring its use for breaking the curse of dimensionality in nearest neighbor query problems. Since this time various applications of Locality Sensitive Hashing have been making appearances in academic publications all over the world. Even very successful technology companies like Google have published improved LSH algorithms [2] using a consistent weighted sampling method "where the probability of drawing identical samples for a pair of inputs is equal

to their Jaccard similarity" [2]. In fact, college textbooks produced from some of America's most prestigious universities like Stanford now include entire chapters dedicated to finding similar items using Locality Sensitive Hashing [3].

However, well over ten years after the first publication, Data Scientists are still hard pressed to find practical implementations of Locality Sensitive Hashing systems which demonstrate why this concept is useful in "big data" applications, or how one would go about creating such a system for finding similar items.

The Locality Sensitive Hashing Use Case

The exponential growth of data over the past twenty years has now created many instances where searching for similar items using all of the available and relevant information is not feasible or not fast. First, consider indexing all the web pages in existence for the purpose of creating a new webpage search engine. One problem is to make sure that the same content is not indexed multiple times. Identifying near identical web pages by comparing a new page to all other pages would not be practical for a number of reasons. The disk space to store all pages alone required for such a task would be gigantic, and searches against such a large amount of data would prove very inefficient when considering how fast the internet changes and grows.

Next, we will consider one additional form of unstructured data which is not quite as obvious... Let us now consider a gene sequence. Imagine breaking apart a gene sequence's text into smaller chunks for the purposes of machine learning and finding other similar sequences. To provide personalized medication, you may be trying to identify a virus using a gene sequence, or you are looking at partitions of an entire human genome to rapidly find segments which may contain high similarity to a known genetic mutation. Even if we created "smaller chunks" of only 100 characters in length, starting at each position within the entire gene sequence's text, a gene sequence containing only 4 unique characters A, C, T, and G could generate more than 10^60 possible unique values. On the surface, this may not sound like a "big data" problem. However, we could all agree that even a reduced version of such a model would not fit on a lap top for use in a remote village in Africa.

Locality Sensitive Hashing can be used to address both of the challenges described above. It is a technique for fitting data with a very large feature spaces into unusually small places. Likewise even smaller feature spaces can also benefit from the use of Locality Sensitive Hashing by drastically reducing required search times and disk space requirements. Instead of storing and searching against all available raw data or even random samples of all raw data, we can use LSH techniques to create very compact signatures which replace storing all of the features typically required for such searches. For example, in the case of displayed webpage text, all displayed text tokens sampled from a document now become a small collection of integers which will be stored and used for all subsequent similar webpage comparisons.

Using the signatures produced by Locality Sensitive Hashing exponentially reduces both storage space and processing time requirements for similar item searches.

Introduction to Minhashing

A form of Locality Sensitive Hashing called Minhashing reduces feature space size using a family of random hashing functions to hash each individual piece of raw input data retaining only the minimum values produced by each unique hashing function.

Wait a minute...? What did he just say...? What does this actually mean...?

Let us start with a very practical example. I am using this next example for simplicity only. I do not recommend that it is the best way to go about comparing text documents for similarity... I have written one article with text tokenization examples using MapReduce [4] style processing and a second article using N-grams [5] which provide additional details on this topic. For this example however, imagine that we are tokenizing documents into individual words or other token forms by whatever process we use. If we wanted to compare our documents for similarity using this strategy, we would have to maintain a collection of all words produced during tokenization by document, and the frequency that each word occurred as well (for frequency weighted calculations). This "word" collection would quickly grow in size as the number of documents increased. In addition, the average length of documents would also impact the size of this collection. In this example our "feature space" size is dictated by the number of unique words we encounter, the number of unique documents we process, and the average length of each document. Using the Minhashing process, the "feature space" described above could be drastically reduced both shrinking the size of our unique "word" collection and the time required to perform document similarity searches against it.

Here are the basic steps for implementing a Minhashing system:

1. The first step in implementing a Minhashing system is to create a family of unique hashing functions.
2. Each word or text token identified during tokenization will be hashed by each unique hashing function.
3. The minimum hash value produced by each unique hashing function for all words within each document processed will be retained within a minimum hash signature representing the unique characteristics of each document processed.
4. The minimum hash signatures for each document can be intersected to produce an accurate approximation of the Jaccard similarity coefficient [2,3].
5. Longer minimum hash signatures (i.e. additional unique hashing functions) will produce more accurate approximations of the Jaccard similarity coefficient [3].

What this means is that given a collection of 300 unique hashing functions, a 2500 word document now becomes a minimum hash signature containing only 300 integer values. Integer values are not only typically smaller than words, but the total number of

items representing the document is now magnitudes of order smaller than its original form.

Measuring the Similarity between Two Minhash Signatures

The Jaccard similarity coefficient between two sets S and T is calculated by dividing the intersection of the sets S and T by the union of sets S and T:

$$J(S,T) = \frac{|S \cap T|}{|S \cup T|}$$

However, the Jaccard similarity between two sets can simply be approximated by intersecting the values between two minhash signatures [3]. For additional explanation and reading on this topic see section 3.3.3 pg. 80-81 in [3]. To state this in layman's terms, two minhash signatures including 300 minimum hash values from 300 unique hashing functions with 300 matching values between both signatures would be 100% similar while two sets with 150 matching out of 300 total values would be only 50% similar.

Implementing a Simple LSH System using Minhashing and C#

Creating a family of *n* unique hashing functions may sound complicated. However, I was able to create the following function in C# using only a couple of lines of code. John Skeet's post [6] explains the logic behind this approach in more detail, although my version has been slightly modified to seed each hash function call with two random numbers. Using two random numbers in this way decreases the overall chance of creating hash functions with duplicated seeds. In fact, potential for this event can be eliminated altogether by using a hashset to ensure that only unique random number seeds are selected. It is also important to note that any data type could be used for hashing purposes since a C# generic input data type is used for the inputData parameter in the Figure 1 example shown below.

```
public static int LSHHash<T>(T inputData, int seedOne, int seedTwo)
{
    unchecked // Overflow is fine, just wrap
    {
        int hash = (int)2166136261;
        hash = hash * 16777619 ^ seedOne.GetHashCode();
        hash = hash * 16777619 ^ seedTwo.GetHashCode();
        hash = hash * 16777619 ^ inputData.GetHashCode();
        return hash;
    }
}
```

Figure 1 - Function seeded with random numbers to produce any number of unique hashing functions.

The hashing method used above is an FVN hash [7] which uses XOR bit shifting to create the seeded hash values. This hashing method has a very low chance of collision. While it works exceptionally well for demonstration purposes, it may be

possible to achieve more accurate LSH search results by choosing a hashing method which has a higher probability for producing collisions between tokens with similar values. In practice, this is sometimes accomplished by creating a thin layer process directly before hash execution which intentionally converts similar tokens within a given threshold to the same value prior to hashing [8]. This approach is left as an independent exercise for the reader.

Using the LSHHash() function above, any number of unique hashing functions can be created by simply selecting and saving random number pairs to be used as seed values. For example, to create a family of 300 unique random hashing functions, the code below can be used to select 300 random number seed pairs. The 300 seed pairs are always used in the exact same order to produce minhash signature values during the minhashing process. For high performance applications, the hash code for each random number seed selected could be saved in order to avoid calculating the seed's hash code using the GetHashCode() method each time the hashing function is called. In this scenario, the actual hash code value for the seedOne and seedTwo input parameters would be passed into the function instead of the random seed's original value.

```
private void createMinhashSeeds()
{
    HashSet<int> skipDups = new HashSet<int>();
    Random r = new Random();
    for (int i = 0; i < minhashes.Length; i++)
    {
        Tuple<int, int> seed = new Tuple<int, int>(r.Next(), r.Next());

        if (skipDups.Add(seed.GetHashCode()))
            minhashes[i] = seed;
        else
            i--;  //duplicate seed, try again
    }
}
```

Figure 2 - Function used to generate any number of random seed pairs used to create a family of random hashing functions.

Each pair of hash seeds above is saved in a minhash array. The length of this array is always equal to the minhash signature's length. For instance, when hashing function number one is needed, the seed pair from position zero within the minhashes array is provided as input to the LSHHash() function described above.

Creating a Minhash Signature

Once a family of random hashing functions has been created, minhash signatures can quickly be generated using any collection of input data. The function in Figure 3 illustrates a minhash signature being created using a collection of integers. However, since a hash code can be generated using any data type, the input data could just as easily be a collection of strings, objects, or even binary data containing videos or

images. This particular characteristic makes minhashing useful for application in a number of different feature reduction scenarios.

```
public int[] getMinHashSignature(int[] tokens)
{
    //Create a new signature initialized to all int max values
    int[] minHashValues = Enumerable.Repeat(int.MaxValue, signatureSize).ToArray();
    HashSet<int> skipDups = new HashSet<int>();
    //Go through every single token skipping duplicates
    foreach (var token in tokens)
    {   //We do not want to hash the same token value more than once...
        if (skipDups.Add(token))
        {   //Hash each unique token with each unique hashing function
            for (int i = 0; i < signatureSize; i++)
            {   //Use the same seeds everytime for each hashing function!!!
                Tuple<int,int> seeds =  minhashes[i];
                int currentHashValue = LSHHash(token, seeds.Item1, seeds.Item2);
                //Only retain the minimum value produced by each unique hashing function.
                if (currentHashValue < minHashValues[i])
                    minHashValues[i] = currentHashValue;
                }
            }
        }
        return minHashValues;
    }
}
```

Figure 3 - This function generates a minhash signature using a collection of integers as input.

The getMinHashSignature() function takes a collection of any number of integers as input, and then each integer within the collection is hashed by each unique hashing function used to create the minhash signature. Duplicate integer values are also skipped as we can be certain that they will not result in the production of a new minimum hash value. Once again, this collection of integers could just as easily be words pulled from a webpage, gene sequence tokens, objects, or any other type of input data. It is also important to notice that this is where the feature space reduction takes place.

A Simple Minhashing Demo Application

In the example application for this article, we generate 10 collections containing up to 100,000 integers within each collection. The example application creates a single integer collection called *the query* which is then compared to each of the 10 integer collections previously mentioned which we will now call *the documents*. The lengths of the individual collections have been randomly determined to better mimic real world similar item searches. First, the Jaccard similarity is calculated between the query and the documents collections as a similarity benchmark. Next, the query and the document collections are all minhashed to create one minhash signature for each collection. For this example we use a family of 400 random hashing functions. After minhashing occurs, only 400 minimum hash values remain within each collection of integers resulting in a dramatic feature space reduction. Finally, the Jaccard

similarity is estimated using only the 400 minhash values for each document. The results are shown in Figure 4 below.

	Actual Jaccard	Time in Ticks	Minhash Jaccard	Time in Ticks
Document 1	0.426	17,498	0.410	559
Document 2	0.422	4,270	0.440	107
Document 3	0.310	2,951	0.323	103
Document 4	0.334	3,240	0.328	107
Document 5	0.420	4,118	0.395	89
Document 6	0.294	4,438	0.305	87
Document 7	0.327	3,175	0.313	86
Document 8	0.326	4,180	0.333	87
Document 9	0.246	2,482	0.248	85
Document 10	0.384	3,695	0.385	87

* 1 millisecond = 10,000 Ticks

Figure 4 - Similar Jaccard similarity values can be seen for each document before and after minhashing occurs.

When using a minimum hash signature including 400 unique hashing functions, the Locality Sensitive Hashing process retains both the highest and lowest scoring document rankings. Nine similarity values are less than 3 percentage points in difference after minhashing occurs with a single document's score (#8) showing a 5 percentage points difference. Increasing the number of hashing functions used will also create more accurate minhash similarity approximations, if required. The total number of collection items prior to minhashing includes 72,476 integers. After minhashing, this count is reduced to only 4,400 items which includes 400 integer values for each of the 11 total collections which were minhashed. Since the example application generates new collections using random numbers, similar results can consistently be seen when executing the program repeatedly against the different random number collections created.

Conclusion

Locality Sensitive Hashing offers the opportunity for substantial feature space reduction when creating a similar item search system. Both large and small systems can benefit from LSH processing since reduced feature spaces offer dramatic reductions in overall search times and disk space requirements. When reviewing the empirical example presented above, the reduced feature space produces very good Jaccard similarity estimates while minhashing reduces the needed data by more than an order of magnitude.

Resources

- Please feel free to learn more about me at: http://www.jakemdrew.com/
- The C# Minhasher class code can be viewed here: http://jakemdrew.com/blog/minhasher.htm
- All programming code used for this article including the entire Visual Studio solution can be downloaded here: http://jakemdrew.com/blog/lshexamples.zip

References

1. Gionis, Aristides et al, "Similarity Search in High Dimensions via Hashing", http://www.cs.princeton.edu/courses/archive/spring13/cos598C/Gionis... , accessed on 05/04/2014.
2. Ioffe, Sergey, "Improved Consistent Sampling, Weighted Minhash and L1 Sketching", http://static.googleusercontent.com/media/research.google.com/en/us/pubs/archive/36928.pdf, accessed on 05/04/2014.
3. Leskovec, Jure et all, Mining of Massive Datasets, "Finding Similar Items",http://infolab.stanford.edu/~ullman/mmds/ch3.pdf, accessed on 05/04/2014.
4. Drew, Jake, MapReduce / Map Reduction Strategies Using C#, http://jakemdrew.wordpress.com/2013/01/08/mapreduce-map-reduction-strategies-using-c/ on 05/04/2014.
5. Drew, Jake, Creating N-grams Using C#, http://jakemdrew.wordpress.com/2013/04/23/mapreduce-for-n-grams-using-c/, accessed on 05/04/2014.
6. Skeet, John, What is the best algorithm for an overridden System.Object.GetHashCode, http://stackoverflow.com/questions/263400/what-is-the-best-algorithm-for-an-overridden-system-object-gethashcode, accessed on 05/04/2014.
7. Walker, Julienne, FVN Hash, http://eternallyconfuzzled.com/tuts/algorithms/jsw_tut_hashing.aspx, accessed on 05/04/2014.
8. J. Buhler., Efficient large-scale sequence comparison by locality-sensitive hashing. http://bioinformatics.oxfordjournals.org/content/17/5/419.full.pdf, accessed on 05/04/2014.