An Explanation of Locality Sensitive Hashing (LSH) with a Demonstration and Discussion of its Use in Plagiarism Detection

By:

Jay Roberts, Spencer Fradkin, and Joe Maxwell

**Introduction/Background**

Locality Sensitive Hashing (LSH) is a similarity search algorithm that allows us to identify similar pairs of items without having to compare all data points pairwise (which can be computationally difficult for sufficiently large datasets). The general idea of LSH is to split signatures related to our data into pieces and organize them into bins. If two subsets of the original signatures we are comparing are identical and contained in the same bin, we consider the original two data points to be candidate pairs (possibly similar). In order to apply this technique, we need to discuss how the data is actually organized into signatures that preserve the similarity between data points contained in a dataset.

*Shingling/One-hot Encoding:*

Since we are typically working with textual data, we need a way to turn the text into meaningful numbers that we can then turn into signatures. Shingling is the process of breaking text into **k** shingles where each shingle is a **k** length window that is moved across the text character by character. Once the data is shingled, we create a vocabulary which is a list containing every unique shingle from every datapoint in the set. Next, we take each datapoint (sentence, paragraph, etc..) and apply one hot encoding to transform it into a vector. To one-hot encode a single datapoint, first, we create an empty vector of 0s of size equal to the vocabulary. Next, we loop through the shingles for that datapoint and fill the **i**th index of the vector with a 1 corresponding to the **i**th index of the shingle in the vocabulary list. After this is done for n data points in the set, the result is n one-hot encoded vectors of 0s and 1s.

*Minhashing:*

The size of the vocabulary created in the last section can be extremely large. Therefore, our one-hot encoded vectors which have the same size as the vocabulary are usually high dimensional. To reduce the dimension of each one-hot vector, we apply Minhashing. Minhashing transforms each one-hot vector into a signature that preserves much of the similarity between vectors. To minhash a high-dimensional vector into shorter signatures, we first select the length we want our signatures to be. The more characters in our signature, the more similarity we can potentially preserve. For an **n** length signature, we need **n** minhash vectors. To create a single minhash vector, we fill an empty **k** length vector with a random permutation of integers from 1 to **k+1** where **k** is the length of the vocabulary we created previously. The ith character in the signature for a single one-hot vector is created by finding the lowest integer in the minhash vector such that the value at the same index in the one-hot vector is a 1. Doing this process **n** times for each of the **n** minhash vectors gives us a signature for a single one-hot vector.

*Locality Sensitive Hashing (LSH):*

To find candidate pairs from these signatures, we apply LSH. First, we break each signature into **b** bands of length **p**/**b** where **p** is the length of the signature associated with each datapoint. Once we have done this for every signature, we organize each signature’s bands into **b** bins according to its place in the original signature. Then, we check to see if we have any identical bands in the bins. If we do, we consider the original sentences associated with those bands a candidate pair. This entire process allows us to only check the jaccard similarity of candidate pairs which is a lot faster than checking the jaccard similarity of every pair of points.

**How Much Faster is LSH Compared to a Brute Force Approach?**

To illustrate the inefficiency of comparing every pair of points to find the ones that have a sufficient Jaccard similarity, we use the notion of complexity. For an array of size **n**, we require **n + (n-1) + (n-2) + ... + 1 =n\*(n-1)/2** calculations to complete the algorithm. This means that the algorithm scales roughly quadratically with respect to the size of the list of data we are using. To express this, we ran the algorithm with sentence count **n**=2250, 4500, and 9000 and received runtimes of 10 seconds, 49 seconds, and 3.5 minutes respectively. It is easy to see that as we double our array size **n**, the time it takes to complete the algorithm more than doubles. This is consistent with the complexity we calculated earlier, and we can see that as **n** gets larger and larger, the runtime will grow increasingly large faster and faster.

Next, we ran the entire LSH algorithm and checked the runtime of this process for the **n**=9000 case. The entire LSH process including calculating the jaccard similarity of candidate pairs after the LSH process was completed took about 6 seconds. Next, we checked to see which of the pairs had a Jaccard similarity greater than .6 and returned roughly 18.5k pairs for the LSH method and 19.3k pairs for the brute force method. After this, we calculated the intersection between these two sets and found that it was exactly 18.5k. This means that the LSH method returned 18.5k correct pairs out of a total 19.3k pairs, extremely accurate for a process that took 1/32 of the time! Additionally, we did not optimize any parameters through this process, so it’s possible that there is a set of parameters that returns even better results than this. It’s also important to note that the brute force method scales worse with larger datasets, so the LSH method becomes more useful as datasets become larger.

**Plagiarism Detection and LSH:**

In our project, we used the existing Python code from *Tutorial #3* and adapted it to conduct our own experiments with a small, original data set. Our team decided to create our own dataset in order to better understand the underlying algorithm and results. The experiment consisted of two parts: text with stopwords and text without stopwords. In the creation of the dataset, we labeled each row as 0 or 1 where 1 indicates plagiarized and 0 not plagiarized. In total, we expected 5 candidate pairs since there were 5 excerpts plagiarized exactly once. Our goal was to initialize the LSH algorithm with and without stop words to find the plagiarized excerpts.

Our team achieved positive results in the dataset including stopwords. Using the default values **k**=8, 100-length minhash arrays, and 20 buckets, the algorithm could only find 3 or less candidate pairs. Using **k**=3, 60-length signature arrays, and 20 buckets, the LSH algorithm consistently found more than 5 candidate pairs. We used the previous configurations to calibrate our search for the optimal initialization. Essentially, we hypothesized that the optimal **k** (number of characters in a shingle) would be between 3 and 8. Immediately, choosing **k**=6 proved to make 4 candidate pairs consistently. After observing this behavior, we began to adjust the length of the minhash arrays and found that our algorithm consistently finds 4 correct candidate pairs with an initialization of **k**=6, 40-length minhash arrays, and 20 buckets.

Although the selections of **k**=8 and **k**=3 are arbitrary, it makes sense that both of those fail by producing less and more candidate pairs, respectively. By shingling every 8 characters, we create a sample space that is too large for the limited amount of data in this dataset. By creating a sample space that is too large, we also create a variability in our signatures which leads to less “hits” when mapping the signature sub-vectors to each bucket. Conversely, by shingling every 3 characters, we create too small of a sample space which causes not enough variability in our signatures and too many “hits” when mapping the signature sub-vectors to each bucket. Using the logic from above, we found that a k=6 leads to the most consistent correct results. Additionally, we also decreased the length of the signatures to keep the number of sub-vectors small. Again, with a smaller dataset, we had to decrease the sample space of both our signature vectors and shingles in order to get the correct results.

We then decided to test dropping stopwords from the text in our dataset. Stopwords can be defined loosely as “necessary words often used” such as common articles, prepositions, and pronouns. We began testing the text with stopwords with values at **k**=6, 40-length signatures, and 20 buckets(the initialization deemed favorable from our previous test). Immediately, the algorithm began outputting the 5 correct candidate pairs consistently.

We believe that the positive result found by removing stopwords is due to the makeup of our dataset. We employ four main types of plagiarism: 1. Direct copy and paste, 2. Use synonyms with the same sentence structure, 3. Change sentence structure with the same words, and 4. Use filler words between main words. The basic LSH model accounted for method one of plagiarism. By removing stopwords from the text, we were additionally able to identify the fourth type of plagiarism that we were missing previously.

When deciding whether to remove stop words to detect plagiarism, it depends on what you consider plagiarism and how strict you want to be. The more strict you are, the higher the chance somebody could be falsely accused. In light of this, it is best to check for plagiarism with and without stop words. There are cases where a professor would want to include every character in a paper to find documents directly copied and pasted. On the other hand, if a person tried to mask plagiarism by adding filler words or changing sentence structure, removing the stopwords could prove useful in identifying instances where this occurred.

**Optional Flagging and Similarity Comparison:**

Using the optimal parameters above, we ran the model including stop words to find only 4 candidate pairs (intentionally using the randomness to ensure a false negative). The intent here is to allow users to employ an additional layer of ‘padding’ to the model that could trigger a candidate pair based on a threshold of similarity instead of only bucket matching.

We tested to see how many additional flags would be generated for each of the mean, max, and min values of the candidate pair for both jaccard and cosine similarities. See Tables 1, 2, 3, and 4 for supplemental data. The results show that using either the max jaccard similarity and the average cosine similarity value both yield only five additional flags. This, compared to the several dozen additional flags from other thresholds, seems like a very manageable number of cases upon which to levy additional scrutiny in the case of plagiarism. Based on these small-scale results, we could recommend that users employ an additional layer of flagging based on the average cosine similarity value.

It should be noted that none of the actual candidate pairs had values that exceeded the max jaccard similarity value. The max jaccard similarity value actually occurred between a pair that we did not generate as a plagiarized pair. For the average cosine value, however, we of course observed some (two out of five) of our true candidate pairs in the generated subset. We believe this artifact of the data gives us reason to believe that in the case of traditional LSH plagiarism detection, cosine similarity may be a better overall detector of plagiarism than jaccard similarity. This may be true because we have ‘vectorized’ our signatures and thus made them inherently more digestible by the cosine similarity than jaccard similarity.

**Conclusion and Future Work:**

Now that we have observed min hashing LSH on a moderate and small sized data set with satisfactory results, we believe we can be confident in its accuracy for small to moderate sized data sets. This may make min hashing LSH appropriate for plagiarism detection within a given school. But when we want to compare plagiarism candidates nationally or even globally, we must ask ourselves how scalable this method is for large and massive data sets.

Recalling the computation time for min hashing, we saw 9000 rows per 6 seconds computation time (or 1500 rows/sec). If we extrapolate this computational time linearly to a data set with say, 10 million equivalently complex entries, we see a computation time of 10,000,000/1,500 = approximately 1.85 hours. While this is still a feasible operation, it becomes prohibitive when we assume that we are likely to have much more complex entries to hash, potentially hundreds of millions of entries (instead of tens), and that we would likely need to do this operation routinely. For all these reasons, when scaling to larger data sets, we would recommend exploring other LSH algorithms such as random projection to see further speed increases.

**Appendix:**

Table 1: Similarity Values for Candidate Pairs

| Candidate Pair Data Only | Jaccard | Cosine | Cosine\_Norm |
| --- | --- | --- | --- |
| Mean | 0.4745 | 0.6814 | 0.5948 |
| Max | 0.6522 | 0.8954 | 1.0 |
| Min | 0.3243 | 0.5241 | 0.2972 |

Table 2: Similarity Values for Non-Pairs

| Non-Pairs Data Only | Jaccard | Cosine | Cosine\_Norm |
| --- | --- | --- | --- |
| Mean | 0.4637 | 0.5419 | 0.3307 |
| Max | 0.7391 | 0.8954 | 1.0 |
| Min | 0.2571 | 0.3673 | 0.0 |

Table 3: Similarity Values for Full Data

| All Data | Jaccard | Cosine | Cosine\_Norm |
| --- | --- | --- | --- |
| Mean | 0.4637 | 0.5452 | 0.3370 |
| Max | 0.7391 | 0.8954 | 1.0 |
| Min | 0.2571 | 0.3673 | 0.0 |

Table 4: False Flags by Similarity and Threshold Value

|  | Additional Flags (Jaccard) | Additional Flags (Cosine) |
| --- | --- | --- |
| Mean | 44 | 5\* |
| Max | 5\*\* | 0 |
| Min | 96 | 58 |

\*Two true pairs included above this threshold (originally 7 pairs, but discounted to 5)

\*\*Zero true pairs included above this threshold

Who did what?

-Spencer:

* Created simplified plagiarism dataset
* Explanation of LSH process
* Comparison of speed of LSH to brute force method

-Jay:

* Removed stop words from dataset
* Calibrated parameters for optimal results
* Wrote application paragraph as well as plagiarism detection and LSH

-Joe:

* Initial model setup
* Optional Flagging and Similarity Comparison
* Conclusion