Scalable product duplicate detection solutions using LSH

A Duplicate Detection Method that seeks to effectively and efficiently preserve duplicate pairs in a minimal set of candidate pairs using LSH

Richie Lee (505917KL)

Erasmus University Rotterdam
PO Box 1738, NL-3000 DR
Rotterdam, the Netherlands
505917KL@student.eur.nl

1 Introduction

In today's world, we are rapidly moving towards a more digital world. Especially under circumstances such as Covid-19, this digitisation is accelerating more than ever before. E-commerce, or more specifically Web-shops, are a good example of an industry which successfully reaped the benefits of recent developments. Sales have increased, not only due to physical store restrictions, but also because of customers that are starting to get more comfortable online. This latter statistic is significantly dependent on user-experience and one of the features that is often taken for granted in this field is the topic of this report, namely *product duplicate detection*.

This report will investigate two important challenges in duplicate detection:

1. Scalability: In order to maintain computational feasibility for large amounts of data (products), we introduce *Locality-Sensitive Hashing (LSH)*, an algorithm that aims to minimise comparisons between products with low probabilities of being duplicates.
2. Classification of duplicates: Though exact copies are easy to identify, this becomes significantly more difficult once partial duplicates are introduced. Our solution includes *Jaccard-similarity*, a statistic used for gauging similarity between sets, which for our purposes include sets of product characteristics.

2 Data

The data includes 1624 products with data on the *title*, *features*, *url*, *store* and *model ID's*. The model ID is used to identify true duplicates. This yields 691 items which are included in one or more of the 399 duplicate pairs.

3 Methodology

This section goes into detail on how the product information is first compressed, then reduced to candidate pairs and then compared for duplicate classification. Figure 1 provides an overview of how these steps are implemented.

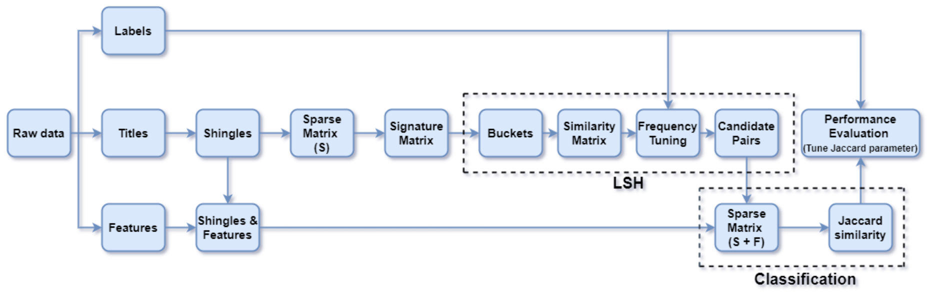


Fig. 1: Overview duplicate detection steps

[1]First, all **Titles**, duplicate pairs (**Labels**) and **Feature** key-value pairs are stored for every product, of which only the titles will be used for candidate pair selection (LSH).

These titles are first cleaned by removing uninformative symbols[2], after which the titles are split into substrings of stand-alone words (excluding single letter strings). These substrings will be used as **Shingles**, where every shingle can be interpreted as a product characteristic. The next step collects all distinct shingles, which will now all be assigned an unique integer index as identifier. Now, we are able to summarise every product's title as a vector of integers. These vectors are then converted to sparse vectors, which only store the identifying indices for present shingles. By collecting all shingle sparse vectors, we now obtain a **Sparse Matrix (S)**.

Subsequently, this sparse matrix can now be used to approximate similarity between products. First, the list of all shingle indices is shuffled. Then, we iterate over shuffled vector of indices, where for every iteration, we check whether the index (corresponding to a shingle) is present in the current products sparse vector. Once we find a match, the number of iterations it required to find the first match is saved (only band-widths of size 1 are considered for simplicity). Once this process is repeated for every product a single permutation is completed. Then, this process is repeated for a prespecified number ($n_{perm} = 100$) of different random permutations. The collection of all permutations is then defined as the **Signature Matrix**, a compact new representation of the all the product titles.

3.1 LSH

If products were to share the same iteration value for the same permutation, this would imply the corresponding shingle to be a common one amongst this group of product. This information is applied by distributing products in **Buckets** with their respective iteration value, where common shingle products are allocated to the same bucket.

This bucket information is then restructured to a **Similarity Matrix** in which entries (i, j) take value 1 if products i and j share the same bucket, and 0 otherwise.

This process is then repeated n_{perm} times after which all similarity matrices are summed. In this sum of similarity matrices, higher values, corresponding to Frequency of common buckets for products, can be interpreted as higher expected value in terms of title similarity. This summed similarity matrix is then reduced to candidate pairs in which all pairs above a **Tuned Frequency** threshold are selected. The frequency threshold f^* is selected by minimising the number of necessary candidate pairs (and thus comparisons to be made), while ensuring that an acceptable number of true duplicate pairs are not discarded

[1] This implementation of candidate pair selection aims to follow [1], with the exception for the signature matrix sums and minor simplifications

[2] The omitted interpunction includes: ( ) - ' " / , [ ]

during LSH (discarded, non-candidate pairs are not compared and will therefore not be detected). Mathematically, the tuning can be expressed as follows:

$$\begin{aligned} &\mathit{Minimise} && |CP(f^*)| = \underset{f}{\operatorname{argmin}}\ |CP(f)| && (1)\\ &s.t. && |TP \notin CP(f)| \leq a && (2)\\ & && f \in \{0, \cdots, n_{perm}\} && (3) \end{aligned}$$

with $|\cdot|$ denoting set cardinality (number of pairs in set), CP and TP referring to the set of candidate and true duplicate pairs respectively, and constant a ($a = 1\%$) denoting the (a-priori selected) number of acceptable discarded true duplicate pairs.

The LSH steps are now concluded by selecting the set of **candidate pairs** using the tuned frequency threshold f^*.

3.2 Classification

Rather than using title information only, classification also implements feature data. This is done by extending the list of product characteristics from shingles only, to **Shingles and Features**. More specifically, features are kept as key-value pairs, as a means to preserve interpretation. This extended list of characteristics is used to construct a new **Sparse Matrix (S + F)**.

Subsequently, all selected candidate pairs are compared by means of **Jaccard similarity** of the sparse vectors. Jaccard similarity is obtained through the ratio of matching sparse vector elements to the total number of distinct elements in for both products combined, which can be expressed as follows:

$$JS(p_i, p_j) = \frac{|(p_i \cap p_j)|}{|(p_i \cup p_j)|}, \tag{4}$$

with $|\cdot|$ denoting set cardinality, and p_k denoting the set sparse vector elements for product k.

Lastly, a threshold for the Jaccard similarity T_{js} is tuned for which the classification follows:

$$pair(i,j) = \begin{cases} \mathit{duplicates}, & \text{if } JS(p_i, p_j) \geq T_{js}, \qquad T_{js} \in (0,1) \\ \mathit{distinct}, & \text{otherwise.} \end{cases} \tag{5}$$

Through grid search T_{js} can be optimised for various performance measures. This implementation optimises for F1-measure, a metric that will be further elaborated on in section 4.

3.3 Bootstrap

To evaluate performance, a training-test data split is used. These datasets are obtained by applying bootstrapping, an algorithm that obtains its training data by randomly sampling products with replacement, with the remainder being used for testing. Additionally, when products are sampled multiple times, all copies but one will be removed. The number of draws for training is 1624, the number of products in the original dataset, which on average should yield a train-test split of 63% and 37% respectively.

For each bootstrap iteration, the full algorithm (Figure 1) is ran on the training first, after which the optimal frequency parameter (f^*) and jaccard-similarity threshold parameter (T_{js}) are saved and used for duplicate detection tetsing. This prevents data-leakage through optimisation of parameters.

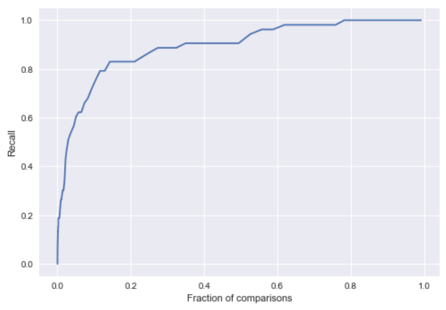
4 Results

4.1 Scalability

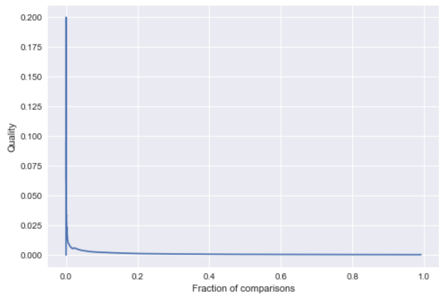
In order to account for scalability, all computations were executed using (nested) lists. Additionally, the usage of sparse matrices over boolean matrices reduced the order of magnitude of matrix dimensions from 10^4-10^5 to less than 10^2. No concerning increase in computing time was observed as the number of products increased, as mostly, the less-demanding calculations of the algorithm were affected more. A single bootstrap iteration of the algorithm (including tuning) generally requires approximately 3 minutes for the full dataset.

4.2 Classification performance

This section will discuss the results for the bootstrap testing sets. The performance of the full algorithm can be evaluated by considering its two main components: candidate pair selection (LSH) and classification (Jaccard-similarity). Figure 2 considers three performance metrics for various levels of fraction of comparisons, which is defined as the fraction of total number of candidate pairs (selected using LSH) among the total number of possible pairs[3]. Moreover, accuracy measures of the Jaccard classification algorithm are displayed as well.

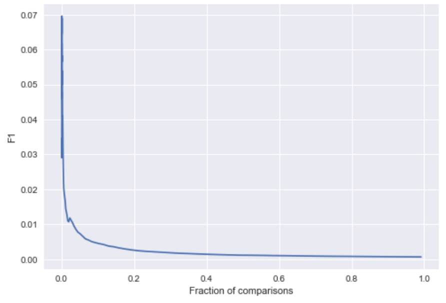


(a) LSH: Pair completeness (recall)

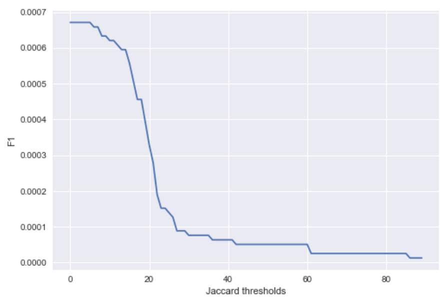


(b) LSH: Pair quality (precision)

[3] The number of possible pairs is computed using: $\frac{n(n-1)}{2}$, where n denotes the total number of products.



(c) LSH: F1-measure



(d) Jaccard: F1-measure

Fig. 2: **Performance measures for LSH and Jaccard-classification**

After investigation of the performance-metrics, the following conclusions are drawn:

1. **Pair completeness (recall)**: the fraction of the number of correctly detected duplicates among the total number of duplicate classifications. Maximising recall will minimise the number of false negatives (undetected product duplicates). In subfigure 2a, it can be seen that duplicates are preserved adequately as the recall manages to remain high, despite substantial numbers of candidate pair reductions.
2. **Pair quality (precision)**: the number of correctly detected duplicates out of the total number of duplicates present. Maximising precision will minimise the number of false positives (distinct products classified as duplicates). Subfigure 2b highlights the challenge well, as the values precision values are low overall. This could be due to a low number of duplicate pairs, relative to the total number of possible pairs (approximately 0.03%). Alternatively, the LSH classification criteria could contain insufficient product information. This could be improved by expanding the product information by including features, rather than title shingles only.
3. **F1-measure (LSH)**: the harmonic mean between pair quality and completeness, which seeks to capture a balance of both properties. The shape of the plot in Subfigure 2c shows that for this dataset, the performance is strongly driven by the poor pair qualities. Hence, it can be concluded that minimising false positives can be considered as the main challenge for this particular LSH implementation. This challenge is likely driven by the same obstacles as pair quality.
4. **F1-measure (Jaccard)**: Subfigure 2d shows notable behaviour as the best classification performance is achieved for low Jaccard-similarity thresholds ($T^*_{fs} = 0$), implying redundancy of the entire classification part in the duplicate detection algorithm.

5 Conclusion

In conclusion, though algorithm appears to scale well. We conclude it to not be compatible with the dataset due to its poor performance in both candidate selection and correct classification.

5.1 Discussion

The unsuccessful candidate selection was a result of poor pair quality (precision), which we hypothesize to be related to the imbalanced dataset (0.03% duplicate) or due to an insufficient amount of product information used during LSH. The classification showed superior performance for notably low Jaccard-similarity thresholds, giving rise to the idea of investigating Jaccard similarities distributions:

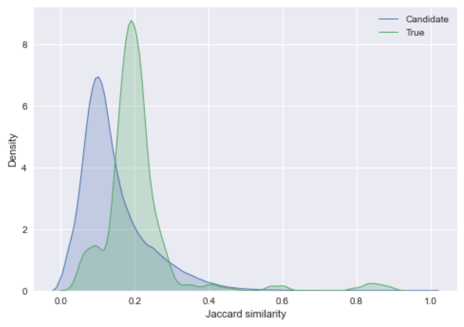


Fig. 3: Jaccard similarity distribution for true duplicate and candidate pairs

Though it may be intuitive that duplicates would share a near-perfect level of (Jaccard) similarity, Figure 3 rejects this assumption convincingly. Therefore, pursuing high product similarity through Jaccard-similarity maximisation is concluded to be ineffective. As a (partial) solution, we could look into alternative criteria to represent product information, or into methods of reducing noise in the product information.

Additionally, we also observe a negative bias in terms of Jaccard similarity distributions for the LSH candidate pairs. This implies that, on average, the candidate pairs are significantly less similar than true duplicates, whereas ideally, these distributions would be the same. Therefore, increasing the strictness on candidate pair selection (shifting the distribution to the right) could be helpful.

References

1. Van Dam, I., van Ginkel, G., Kuipers, W., Nijenhuis, N., Vandic, D., Frasincar, F.: Duplicate detection in web shops using lsh to reduce the number of computations. In: Proceedings of the 31st Annual ACM Symposium on Applied Computing. pp. 772–779 (2016)