All Pairs Similarity Search Term Paper

Eric Russo Daniel Jeffries

December 2018

1 Abstract

There are many different situations in which one would like to find all objects whose similarity to a given object exceeds a threshold, or all such pairs of objects that satisfy this constraint. This is the fundamental objective of the all-pairs similarity search problem. The problem can also be viewed as a generalization of k-nearest neighbors applied to high dimensional sparse vectors. This type of problem is of particular importance for a search engine such as Google, as a common query may be to find all sets of k-similar search queries. Various heuristic and approximation algorithms exist for this task, an example being the widely used MinHash algorithm. This paper specifically focuses on answering the problem exactly.

2 Introduction and Applications

Several different choices of a similarity metric are used in practice, common examples being Cosine similarity, Jaccard similarity, and Overlap similarity. Cosine similarity, defined as the dot product of two normalized vectors, has been shown to be an effective discriminator in practice, and is used to illustrate the following algorithms. A naive method of answering the query would be forced to enumerate $O(n^2)$ different dot products and test if each satisfies the threshold. The essence of Google's paper is to provide a series of optimizations that use sorting and various data structures to minimize the number of these considered pairs. There are countless applications of All Pairs Similarity Search. For instance, any data with an equivalent set representation, such as a n-gram counts for a word document, has an equivalent sparse vector representation.

2.1 Image Features

Many machine learning algorithms perform feature extraction on images, and their representations can have vary degrees of sparsity. For instance, a large data set of many classes of images will have many more extracted features than images, and images in different classes will have plenty of completely different

features. A fast All Pairs algorithm would allow for a fast search for similar images, even for images that were not present in the data set initially.

2.2 Document Similarity

Given that most text documents draw from a very large set of possible words and word-combinations (n-grams), word vectors tend to be very sparse. This makes document similarity a very natural application of this algorithm. Document similarity is a very useful metric, both in the domains of filtering web pages for search and determining authorship or plagiarism.

3 Notation, Fundamental Algorithm, and Data Structures

The input to the algorithm is a set V of real-valued vectors. A sparse vector is a list of pairs $(i, x[i])$ such that $x[i] > 0$. The magnitude or norm of a vector is denoted $||x||$, while the $size$ of the vector (the number of non-zero features) is denoted $|x|$. The maximum weight value for a feature over all vectors is denoted $maxweight_i V$ and the maximum value for a specific vector is denoted $maxweight_i x$.

Because the algorithm is exact, the worst case complexity will always be $O(n^2)$, as all $\binom{n}{2}$ pairs could possibly satisfy the threshold. However, on sparse vector sets, the run time is often substantially faster in practice. The first step towards preprocessing the data is to build a set of inverted lists $I_1 ... I_m$ for all the features, where I_i stores all pairs $(x, x[i])$ such that $x[i] > 0$. This significantly reduces the number of multiplication operations that would have been performed while computing dot products. Inverted lists are constructed dynamically, vector by vector, and features from the current vector are multiplied by corresponding features in the inverted list. A Hash table is used to provide a mapping from candidate vectors to the running dot product computed over a subset of the current vector's features. Google gives a naive solution which is the first iteration of their All Pairs algorithm.

4 Naive Approach

The simplest solution to the All Pairs problem is to create a set of inverted lists $I_1 ... I_m$ for all vector features before calculating and pairwise similarities. However, not all inverted lists need to be scanned for all pairs of vectors, as most vector pairs share few or zero dimensions. This most basic approach also wastes time in calculating $sim(y, x)$ when $sim(x, y)$ has already been computed. One basic algorithm that avoids these pitfalls is described as follows:

- Initialize inverted lists and results list to be empty
- Loop through every vector x, do:

- Create an empty hash map from vector id to weight
- For every feature i in x with nonzero weight, multiply its weight against all weights stored in the inverted list for that feature, storing the total dot-sum for every individual vector in the hash map
- For every candidate vector with non-zero value in the hash map, if their value meets the threshold, add the pair to the list of results
- End loop, then add $x's$ vector index and features with non-zero weights to the inverted lists for those features

Google's first algorithm takes advantage of the symmetry of candidate pairs and builds the inverted lists dynamically. That is, the features of a vector x are only added to the inverted list after matches are found for x. When vector y is being considered for matches, its features are only compared against those previously indexed vectors.

5 Optimizations

Exploiting the Threshold and Sorting Heuristics

The value of the threshold can be exploited directly to reduce the number of candidate pairs considered. We can do this by first reordering the dimensions of all vectors in decreasing order of feature prevalence. In other words, the features that occur most often are processed first. The main optimization behind the second iteration of Google's algorithm is to do this sorting step first, then only index a portion of each vector when dynamically constructing the inverted list. How do we know when to start indexing less common features? Well, we can track an upper bound of the highest possible similarity using the first k features, call it b. One upper bound of the similarity using k features equals $\sum_{i=0}^{k} maxweight_i(V) * x[i]$. Basically if we always take the highest possible value for that feature and multiply it with $x[i]$, that gives us a strict upper bound over those features. Now, when b eventually is greater than or equal to the threshold t, we can finally start indexing features for consideration. The reason this works has to do with the how candidates are considered, which is that any vector with indexed features in common with vector x is tested for similarity. The claim for correctness is: any vector whose similarity (i.e. dot product) with vector x is greater than or equal to t shares at least one indexed feature with x, whose index is $\geq k$. To see why this is true, suppose there is some vector y that $dot(x, y) \geq t$, and that it has no indexed features with index $\geq k$ in common with x. That must mean that all of its common features with x reside in the first $k - 1$ dimensions, and that the dot product of those features with $x's$ corresponding features meets the threshold. However, we computed the strict upper bound on the similarity over the first k features, and if it ever met or exceeded the threshold then we began indexing features starting with k. But, because the dot product between x and y over k features is at most that

upper bound, a feature of index $\geq k$ would have to be included if y met the threshold. Therefore, no such vector can exist, and the claim for correctness is true.

By including this optimization, the size of the inverted list is reduced, and the total number of candidate pairs is kept to a minimum, as we are now indexing only a fraction of the least common features.

5.1 Remscore and Minsize

Another means of exploiting the threshold is by tracking the maximum possible score attainable by any candidate vector that has yet to be indexed, *remscore*. Initially, we set it to the dot product of the vector we're checking and the maximum value of any vector for each of its features:

$$\sum_i x[i] \cdot maxweight_i V$$

Then, after every feature considered in x, we deduct $x[i] * maxweight_i V$ from the upper bound. After this upper bound drops below the target threshold, it is impossible for any vector that has not yet been added to the map to meet the threshold for similarity. So the algorithm switches from adding new candidates to the map to only updated existing ones at this point.

Secondly, we can compute a lower bound on the size of any vector that meets the similarity requirements, *minsize*. Remember, size in this context is the number of non-sparse entries in the vector. It is clear that for all vectors y, $dot(x, y) \leq maxweight(x) \cdot size(y)$. We also want $t \leq dot(x, y)$ for the vectors to be similar, therefore we want $t \leq maxweight(x) \cdot size(y)$, or equivalently $size(y) \geq t/maxweight(x)$. If we first sort all vectors in descending order of $maxweight(x)$, this lower bound for size will increase monotonically during the algorithm. We can, at every feature iteration, remove vectors from our inverted list of that feature that fail to meet the size requirement. Iteratively removing from the front works well in practice, again due to the non-increasing order of the maximum weights. This is because the maximum weight for a vector tends to be inversely correlated with vector size. In our implementation, inverted lists are offset by the number of removed entries, and resized to claim memory when the offset exceeds half of the list length.

5.2 Binary Vector Data

There are many instances where data can easily be transformed to sparse binary vectors, where every entry may represent the presence or absence of a feature. Google proposes further refinements to the algorithm for this particular case. Firstly, it is no longer required to first normalize the input vectors as a preprocessing step. Second, we can now index all features except the first b, where $b/|y| < t$. This way, the product of the unindexed features is less than b,

therefore the cosine similarity is

$$\frac{dot(x, y\_unindexed)}{\sqrt{|x|} \cdot \sqrt{|y|}} \leq \frac{b}{\sqrt{|x|} \cdot \sqrt{|y|}} \leq \frac{b}{\sqrt{|y|} \cdot \sqrt{|y|}} = \frac{b}{y} < t$$

where we used the fact that we're sorting the vectors in non-increasing order of length, so $|x| >= |y|$. Any vector whose cosine similarity meet the threshold must have indexed features such that the sum of the two similarities be greater than t, therefore any such vector y must have an unindexed feature in common with x. This means y will be considered a candidate, unless remscore or minsize optimize it out.

Notice that now, for any two similar vectors

$$\frac{dot(x, y)}{\sqrt{|x|} \cdot \sqrt{|y|}} \geq t$$

$$\frac{dot(x, y)^2}{|y|} \geq |x| \cdot t^2$$

And because $dot(x, y) \leq |y|$

$$\frac{|y|^2}{|y|} = |y| \geq |x| \cdot t^2$$

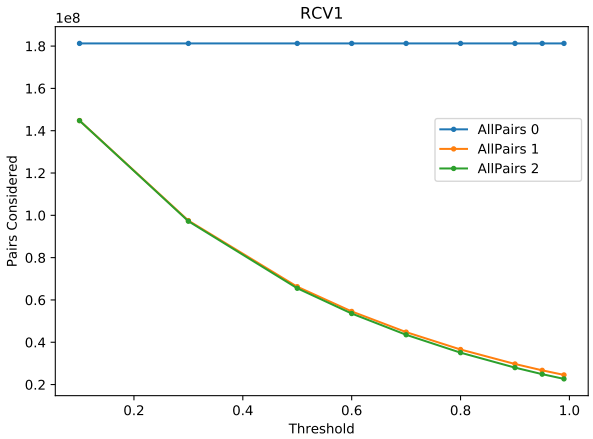
This gives us a new minimum size constraint, which monotonically increases during the algorithm. Additionally, because vectors are sorted in non-increasing order of length, our iterative removal step is guaranteed to remove every vector of insufficient length by scanning from the beginning of each inverted list.

6 Performance Testing on Real World Data Sets

6.1 Data Sets

Reuters CV1

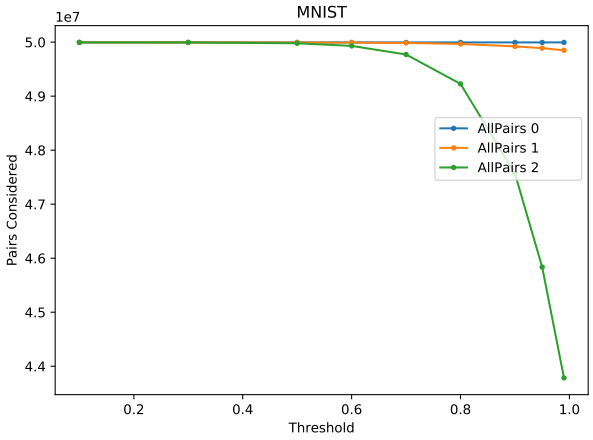
The Reuters Corpus Volume I (RCV1) data set is comprised of sparse word vectors created from thousands of English documents. This data set was originally produced by David D. Lewis of ATT Labs. It contains 47,236 features with an average sparsity of 99.84%.



The first and second optimizations of the All Pairs algorithm performed similarly in reducing the number of pairs considered on this data set. Compared to the naive approach, both optimized versions reduce the number of pairs considered by 63.8% at a threshold of 0.5 and 86.4% at a threshold of 0.99.

MNIST

We constructed a semi-sparse matrix of pixel intensities using the MNIST data set by removing inactive (0 intensity) pixels. It contains 777 features with an average sparsity of 80.55%.



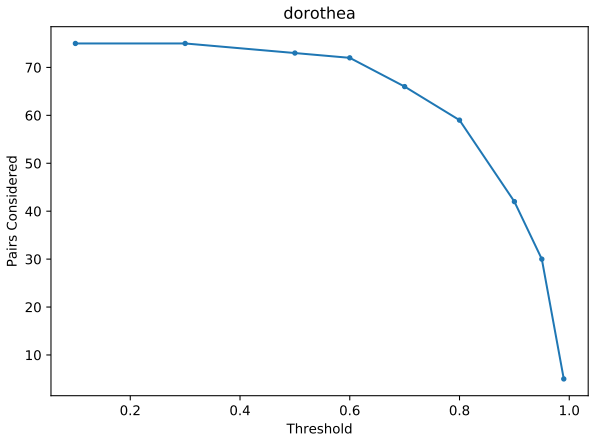
The second optimization of All Pairs performs much better on the MNIST data at high thresholds. At a threshold of 0.99, the second optimization reduces the number of pairs considered by 12.4%, whereas the first optimization reduces the number of pairs considered by 0.3%.

6.2 Binary Data Sets

All features in binary data sets have a value of 1 or 0 and can be conveniently represented by a vector of non-zero feature indices.

Dorothea

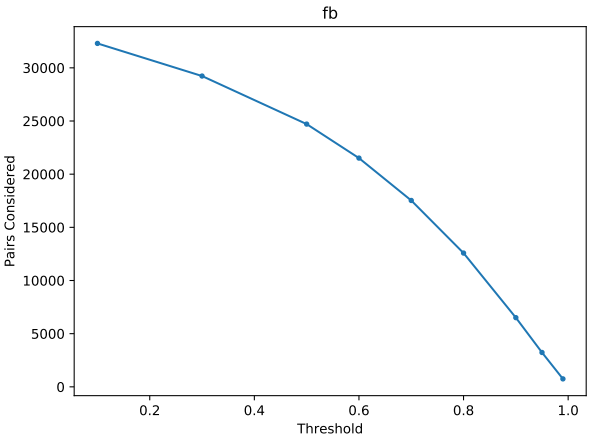
Dorothea is sparse binary data set with features to represent a drug's activation with certain compounds and was originally produced by DuPont Pharmaceuticals Research Laboratories and KDD Cup 2001. It contains 100,000 features with an average sparsity of 99.09%.



Considering the number of vectors in Dorothea is 800, the binary optimization of All Pairs performs very well in reducing the number of pairs considered, from $\binom{800}{2} = 319{,}600$ down to only 5 at a threshold of 0.99.

Facebook Adjacency Matrix

We constructed a sparse binary (adjacency) matrix from the descriptions of ego networks on Facebook. The source data was provided by the Stanford Large Network Dataset Collection and originally produced by J. McAuley and J. Leskovec. The adjacency matrix contains 4,038 features (nodes) with an average sparsity of 97.87%.



The Facebook data exhibits similar performance to Dorothea. For an input size of 3,959 vectors, the binary optimization reduces the number of pairs considered to 748 at a threshold of 0.99.

6.3 Conclusions and Other Results

In general, the All Pairs algorithm performs better on sparse data, and given a specific data set, it considers many less pairs for higher thresholds. This makes it particularly suited for document similarity, where many vectors are approximately similar due to common words and phrases (or keywords and common phrases in programming languages), but only highly similar results are desired in a search.

In our time researching this algorithm, we also used it to compare a plagiarized codebase to the original using sparse vectors generated from 1, 2, and 3-grams, and the algorithm correctly identified the plagiarized files. The similarity for the matches was above 85%, while all other considered pairs of files had similarities less than 50%.

7 Possible Areas for Further Improvement

While we didn't implement it ourselves, we believe that parallelism could be used to achieve great speedup in a number of points during the algorithm. One key way in which multiple threads could be leveraged would be to process features independently during the Find Matches phase of the algorithm. However, this would require that one thread be responsible for a single vector's features, or that an alternative concurrent thread-safe version of a Hash Map be used. Neither of these solutions was able to be effectively implemented by us, so we are unable to analyze the potential speedups at this time. The other avenue for parallelism would be to process vectors with disjoint features in parallel, or vectors with less features in common than a minimum amount. The challenge would lie in

determining these pairs of parallelizable vectors in an efficient manner, another problem that eluded us.

8 References

Original Google paper
Bayardo, Roberto J., et al. "Scaling up All Pairs Similarity Search." Proceedings of the 16th International Conference on World Wide Web - WWW '07, 2007, doi:10.1145/1242572.1242591.
MNIST handwritten digits
Yann LeCun (Courant Institute, NYU) and Corinna Cortes (Google Labs, New York)
`http://yann.lecun.com/exdb/mnist/`
Reuters CV1
David D. Lewis, Yiming Yang, Tony G. Rose, and Fan Li.
`https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/binary.html#rcv1.binary`
Dorothea
DuPont Pharmaceuticals Research Laboratories and KDD Cup 2001
`https://archive.ics.uci.edu/ml/datasets/dorothea`
Facebook Ego Networks
J. McAuley and J. Leskovec
`https://snap.stanford.edu/data/ego-Facebook.html`