Building k-nn Graphs From Large Text Data

Thibault Debatty*, Pietro Michiardi†, Olivier Thonnard‡ and Wim Mees§
*Royal Military Academy, Brussels, Belgium, Email: thibault.debatty@rma.ac.be
†EURECOM, Campus SophiaTech, France, Email: pietro.michiardi@eurecom.fr
‡Symantec Research Labs, Sophia Antipolis, France, Email: olivier_thonnard@symantec.com
§Royal Military Academy, Brussels, Belgium, Email: wim.mees@rma.ac.be

Abstract

In this paper we present our new design of NNCTPH, a scalable algorithm to build an approximate k-NN graph from large text datasets. The algorithm uses a modified version of Context Triggered Piecewise Hashing to bin the input data into buckets, and uses NN-Descent, a versatile graph-building algorithm, inside each bucket. We use datasets consisting of the subject of spam emails to experimentally test the influence of the different parameters of the algorithm on the number of computed similarities, on processing time, and on the quality of the final graph. We also compare the algorithm with a sequential and a MapReduce implementation of NN-Descent. For our datasets, the algorithm proved to be up to ten times faster than NN-Descent, for the same quality of produced graph. Moreover, the speedup increased with the size of the dataset, making NNCTPH a sensible choice for very large text datasets.

1. Introduction

A graph is a mathematical structure made up of nodes (or vertices) connected with edges. In some cases, the edges have a weight, resulting in a weighted graph. Instead of physical relationships like "is a friend of", "likes" or "has an hyperlink to", edges may also represent the similarity between nodes. This results in a nearest neighbors graph, of which two flavors exist. The most commonly used is the k-nearest neighbors graph (k-NN graph), where each node is connected to (has an edge to) its k nearest neighbors, according to a given similarity metric.

Another possibility is to create an ϵ-NN graph, a graph where an edge exists between two nodes if their distance is less than a pre-defined threshold ϵ. However, it has been shown in [1] that ϵ-NN graphs easily result in disconnected components. Moreover, it is usually difficult to find a good value of ϵ which yields graphs with an appropriate number of edges [2]. From a practical point of view, it is more efficient to build a k-NN graph and afterward filter the graph with different values of ϵ. Hence most of the research on graph building currently focuses on k-NN graphs.

Until recently, no algorithm was able to quickly build k-NN graphs from a large text datasets. Therefore, in [4] we proposed a preliminary design of NNCTPH, a new scalable algorithm to build k-NN graphs from large text datasets. The algorithm uses a custom Context Triggered Piecewise Hashing (CTPH) function, which tends to produce the same hash for similar input strings, to bin the input text data into buckets. Then, the algorithm computes the subgraph inside each bucket, and uses multiple stages to join these unconnected subgraphs. The algorithm is based on the MapReduce programming model and can be executed on scalable computing frameworks such as Hadoop. Furthermore, it uses a single job (as opposed to iterative algorithms) which facilitates the task of the resource scheduler of systems like Hadoop or Spark.

In this paper, we enhance the original design of the algorithm by using NN-Descent inside the buckets to reduce the computational cost of the algorithm. We also experimentally test the algorithm on different datasets consisting of the subject of spam emails. We test the influence of the different parameters of the algorithm on the number of computed similarities, on processing time, and on the quality of the final graph. We also compare the algorithm with a sequential and a MapReduce implementation of NN-Descent, and show that NNCTPH largely outperforms state of the art approaches in term of run-time.

The rest of this paper is organized as follows. In Section 2 we present existing algorithms to build a k-NN graph, and algorithms that perform nearest neighbor search in general. In Section 3 we present the implementation details of our algorithm. In Section 4 we show experimental results, and compare our algorithm with a sequential and a MapReduce implementation of NN-Descent. Finally, in Section 5 we present our conclusion and directions for future work.

2. Related work

Different approaches exist to build a k-NN graph. Some of them tolerate incorrect edges to speedup the building process and produce an approximate graph, while others produce an exact graph. In both cases, these building algorithms are closely related to nearest neighbor search algorithms. But when it comes to building a k-NN graph from a big unstructured text dataset, where each node consists of a string, none of these offer an efficient solution.

The naive method, also called linear search, consists in computing the distance between the query point and every other point in the set, keeping track of the "best node so far" (or k "best nodes so far"). Similarly, the most naive way to build a complete k-NN graph is is to use brute force to compute all pairwise similarities. Then, for each node, the algorithm keeps only the k edges with the highest similarity. This method has a computational cost of $O(n^2)$ and is thus very slow, even implemented in parallel.

Another approach is to use some kind of index to speedup nearest neighbors search. These techniques usually rely on the branch and bound algorithm, and the index is used to partition the data space. For example, a k-d tree, that recursively partitions the space into equally sized sub-spaces, can be used to speedup neighbor search. R-trees can also be used for euclidean spaces. In the case of generic metric spaces, vantage-point trees, also known as metric trees, and BK-trees can be used. But these approaches are hard to implement in parallel on a shared nothing architecture like MapReduce (MR). In [3] for example, the authors present a distributed k-NN graph building algorithm, but use a shared memory architecture to store a kd-tree based index.

Some nearest neighbors search algorithms use Locality-Sensitive Hashing (LSH), like [6], to hash the input items so that similar items are mapped to the same buckets with a high probability. As opposed to conventional hash functions, such as those used in cryptography, the goal of LSH is to maximize the probability of collision between similar items. Various authors also propose algorithms relying on LSH the build k-NN graphs. In [8], the authors use LSH to divide the dataset into small groups. Then, inside these small groups, the algorithm builds the k-NN graph. As groups are not overlapping, the constructed graph is a union of multiple isolated small graphs. To build the final graph, and improve the approximation quality, the division is repeated several times to generate multiple approximate graphs, which are combined to produce the final graph. They also show experimentally that their algorithm is much faster than existing algorithms, for similar quality of the built graph. But LSH functions are defined only for some similarity measures (l_p, Mahalanobis distance, kernel similarity, and χ^2 distance). The algorithms relying on LSH can thus not be used to build a k-NN graph from text data using an edit distance (Levenshtein distance) or another similar function (weighted Levenshtein distance, Jaro-Winkler distance, Hamming distance) as a similarity metric.

A different and versatile algorithm to efficiently compute an approximate k-NN graph is described in [5]. The algorithm, called NN-Descent, starts by creating edges between random nodes. Then, for each node, it computes the similarity between all neighbors of the current neighbors, to find better edges. The algorithm iterates until it cannot find better edges. The main advantage of this algorithm is that it works with any similarity measure. The authors also propose a MapReduce version of the algorithm, but it requires multiple iterations to converge, and two MR jobs per iteration. Moreover, the algorithm requires to read and write a lot of data on disk between MR jobs. Although the sequential version of the algorithm proved to be very efficient, these constraints make it inefficient when implemented in parallel. This will be confirmed during the experimental tests presented below.

As no current algorithm was suitable for building a k-NN graph from a big text dataset, in [4] we proposed a new MR algorithm that requires a single iteration and a single MR job, and does not rely on a shared index. Internally, the algorithm uses a specific hashing scheme, called Context Triggered Piecewise Hashing (CTPH), also known as Fuzzy Hashing, to bin the input data into buckets. To build the subgraphs inside the buckets, the original algorithm used the naive method, which has a computational cost $O(n^2)$. Consequently the algorithm was very sensitive to skewed data, as in this case the computational time was largely dominated

by the largest buckets. Moreover, the algorithm only supported a fixed number of buckets.

3. NNCTPH

We present here the design of an enhanced version of NNCTPH. The algorithm, presented in Algorithm 1, requires a single MR job. In the map phase, the algorithm uses a modified CTPH function to produce a hash of each input string. This hash value is then used to bin the string into a bucket. Each reduce task builds a k-NN graph of the strings in the bucket. We experimentally found that, for small datasets, the naive method requires less computations and processing time than sequential NN-Descent. Therefore, if the number of strings in the bucket is smaller than a given threshold θ, the reduce task uses the naive method, otherwise it uses NN-Descent.

Algorithm 1 NNCTPH

Input:
s the number of stages
c the number of characters of emitted hash
l the number of letters used to produce the hash

procedure MAP($string$)
  h = CTPH($string$, s, c, l)
  // $string$ is emitted s times
  for i in $0..s$ **do**
    // As key, we concatenate the stage i and
    // a substring of c characters of the hash
    // starting at i^{th} character
    $key = i$ _ SUBSTRING(h, i, c)
    EMIT($< key, string >$)
  end for
end procedure

procedure REDUCE($key, < strings >$)
  if SIZE($strings$) $< \theta$ **then** BRUTEFORCE
  else NNDESCENT
  end if
  for $string$ in $strings$ **do**
    // Emits the k strings from this bucket
    // that have the highest similarity with $string$
    EMIT($<$ $string$, EDGES($string$, k) $>$)
  end for
end procedure

To control the number of buckets, and hence the number of strings per bucket, we modified the original CTPH function to: *i)* produce a hash of variable size; and *ii)* use only a subset of letters in the hash, instead of the 64 original letters. Moreover, if we only emit each string once, we will end up with a series of unconnected subgraphs, as each string is binned into a single bucket, and no edges are created between the strings of different buckets. To reconnect the graph, in the map phase, the algorithm creates a longer hash (using a coefficient we call $stages$) and emits the input string once for each subpart of the hash.

For example, to run the algorithm with 100 buckets and two stages, the custom CTPH function produces a hash of three characters, using ten letters. If the hash of an input string is `ABC`, the original string will be emitted twice by the mapper: once for `AB`, and once for `BC`. In this way, we can expect that the reduce task for bucket `BC` will produce edges to strings located outside bucket `AB`, hence reconnecting the graph.

The algorithm thus requires three parameters: the number of stages (s), the number of characters of emitted hash (c), and the number of letters used to produce the hash (l). These have an impact on the quality of the graph, on the quantity of data that has to be shuffled, on the parallelism of the algorithm, on the quantity of RAM required by the reducers, and on the number of similarities to compute.

The number of buckets produced by the hashing function is l^c. If we assume the input strings are uniformly distributed over the buckets (if data is not skewed), the number of strings per bucket is $\frac{n}{l^c}$. Dong *et al.* [5] experimentally found that the computational cost of NN-Descent is around $O(n^{1.14})$. As we use NN-Descent inside the buckets to build the subgraphs, the number of similarities to compute is: $O(s \cdot l^c \cdot (\frac{n}{l^c})^{1.14})$. If we choose l and c such that the number of strings per bucket (n/l^c) is constant (with a number of buckets proportional to the size of the dataset), this means that the computational cost of our algorithm is proportional to the number of buckets, and thus proportional to the size of the dataset. This is a lower bound. If the input data is skewed, which is the case of our test dataset, the total number of similarities to compute is higher.

The number of stages s will also have an impact on the quality of the final graph: if more stages are used, the same string will be emitted multiple times. The probability to discover correct edges will thus also rise. The number of stages is also directly proportional to the quantity of data to shuffle and transmit over the network: data to shuffle $= s \cdot n$ where n is the size of the dataset. Using a higher number of stages will thus slow down the algorithm.

In the next section, we perform a sensitivity analysis of the effects and interactions of these parameters.

4. Experimental evaluation

To analyze the performance of our algorithm, we implement it using Hadoop MapReduce and test it on datasets containing the subject of 200.000 spam emails. This dataset is a sample of spams collected by Symantec Research Labs in 2010. It is mainly used to improve spams signature definitions, and to analyze trends in spam campaigns. We also compare it against our Hadoop MapReduce implementations of NN-Descent and of the brute-force method. All algorithms are executed on a cluster of 20 worker nodes, each equipped with a four-core processor, 8GB of RAM, and four 1Gb ethernet cards.

To compute the similarity between spam subjects, we use the Jaro-Winkler distance [7]. This measure of string similarity is normalized such that zero equates to no similarity and one is an exact match.

To measure the accuracy of the output of each algorithm, like in [5], we use *recall*, which is the ratio between the number of correct edges found by the approximate algorithm (where the ground truth are the edges found by the naive algorithm) and the total number of edges created by approximate algorithm.

4.1 Number of stages

We first test the influence of the number of stages used to run the algorithm. The number of stages is the number of times each input string will be emitted by the mapper. We use this coefficient to reconnect the different subgraphs produced by the reducers.

We use NNCTPH to build a 10-nn graph from our dataset. We use hashes of two characters, with 32 possible letters. We thus create 1024 buckets, and we let the number of stages vary between one and ten. The quantity of data that has to be shuffled and transmitted over the network is directly proportional to the number of stages, which is confirmed by our experiments. Using a higher number of stages will thus slightly slow the algorithm down. At the same time, this will distribute the same input string into more buckets, thus increasing the probability to find correct edges.

The resulting running time and recall are shown on Figure 1. As we can see, as little as two stages suffice

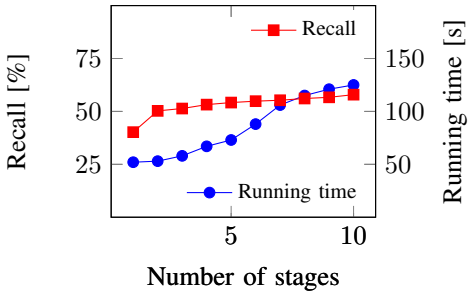


Figure 1: Influence of the number of stages on the running time and recall of NNCTPH.

to correctly discover 50% of edges in the dataset. Increasing the number of stages increases the running time, as expected, but has only a limited effect on recall.

4.2 Number of buckets

We now study the influence of the number of buckets on processing time and on the quality of produced graph. We have two ways to modify the number of buckets: varying the length of the hash, and varying the number of letters used to produce the hash. Therefore we run three series of tests. For each series, we use NNCTPH to build a 10-nn graph from our spam dataset. Given the results of our previous section, we use two stages as this offers the best trade-off between running speed and recall. In the first and second series, we use respectively one and two characters, and we let the number of possible letters used to compute the hash vary. In the third series, we use a fixed number of possible letters (two), and we let the number of characters of the hash vary. These values are summarized below.

	S1	**S2**	**S3**
Characters	1	2	7 to 12
Letters	20 to 44	4 to 40	2
Buckets	20 to 44	16 to 1600	128 to 4096

The resulting running time and recall of each series of experiments are displayed on Figure 2. As we can observe, the running time and recall both tend to decrease when the number of buckets increases, but the impact on recall is quite limited. We can also see that, for the same number of buckets, shorter hashes (with less characters, but created using more possible letters, like **S1**) produce slightly better graphs but run slower.

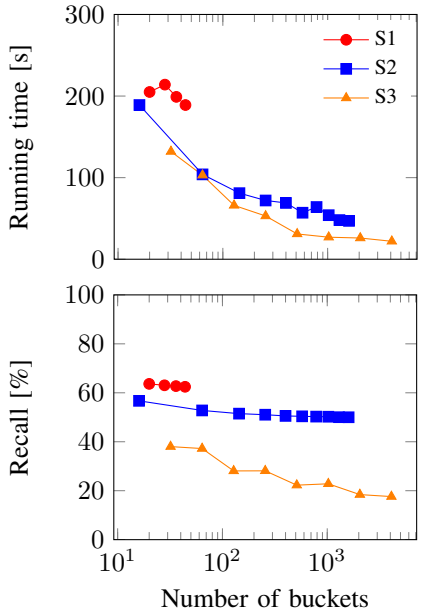


Figure 2: Influence of the number of buckets on the running time and recall of NNCTPH

4.3.Comparison with NN-Descent

We also implement a complete MapReduce (MR) version of NN-Descent, that we compare to our algorithm. We run NN-Descent on our dataset, and for each iteration we measure the total running time and recall. The results are shown on Figure 3. On the same Figure, we also present the results of previous experiments. As we can see, in some cases NNCTPH runs 6 times faster than NN-Descent for the same quality of produced graph, but the attainable recall is limited.

This is mainly due to the principle of binning itself. At some point, the hashing function has to produce different hashes for different input strings. This means that two similar strings, that differ by only one letter, may receive different hashes. Therefore they will be binned into different buckets, which makes the creation of an edge between them impossible. We mitigate this effect using multiple stages, but the resulting attainable recall is still limited to roughly 50%, as shown on Figure 1. We can also reduce this effect by using less buckets, and more strings per bucket, but this increases the computational cost of the algorithm, as shown on Figure 2, and reduces the parallelism of the algorithm. A possible solution to increase recall would be to use different hashing functions in parallel. Furthermore, even if we have an idea of how many edges were

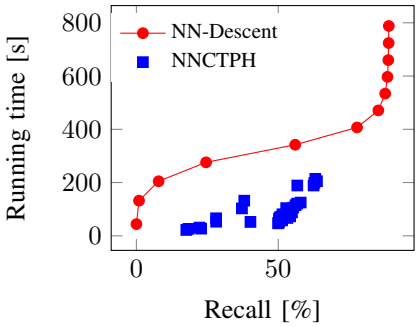


Figure 3: Comparison of NN-Descent and NNCTPH algorithms

correctly discovered by our algorithm, we would like to know which edges are correctly detected, and which ones are not. These are left as a future work.

4.4.Scalability

We now test how the algorithm behaves when the size of the dataset increases. Therefore we use other datasets with up to 800.000 spams. Based on previous experiments, we use two stages, hashes of two characters, and we tune the number of letters used so that each buckets receives an average of 200 spams, as summarized below.

	T2	**T4**	**T6**	**T8**
Spams (x1000)	200	400	600	800
Characters	2	2	2	2
Letters	32	45	55	64
Buckets	1024	2025	3025	4096

We also compare NNCTPH with our MR implementation of NN-Descent and with our sequential implementation of NN-Descent. For both algorithms, we chose parameters that deliver approximately the same recall. The resulting running times and recalls are displayed on Figure 4. As we can see, the recall achieved by NNCTPH with these parameters is very stable, and the running time rises very slowly. As a result, the bigger the dataset is, the higher the speedup with respect to MR NN-Descent. With our dataset of 800,000 spams, we reach a speedup of nearly an order of magnitude for the same quality of the final graph. Clearly here MR NN-Descent suffers from its iterative structure, which is not well suited for the MR framework, and requires a lot of slow disk I/O operations.

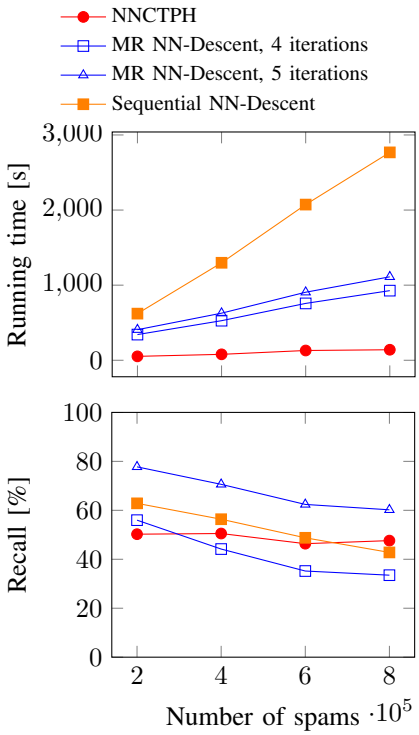


Figure 4: Comparison of NNCTPH and NN-Descent for larger datasets

5. Conclusion and future work

In this paper we presented our implementation of NNCTPH, a MapReduce algorithm that builds an approximate k-NN graph from large text datasets. We used datasets containing the subject of spam emails to experimentally test the influence of the different parameters of the algorithm on the quality on processing time and on the quality of the final graph. We also compared the algorithm with a sequential and a MapReduce implementation of NN-Descent. For our datasets, the algorithm proved to be up to ten times faster than the MapReduce implementation of NN-Descent, for the same quality of produced graph. Moreover, the speedup increased with the size of the dataset, making NNCTPH a perfect choice for very large text datasets.

In the future we plan to further study the quality of the produced graphs: until now we have an idea of how many edges were correctly discovered by our algorithm, but we would like to know which edges are correctly detected, and which ones are not. We also want to study the influence of graph quality on the post-processing algorithms that use an approximate k-nn graph (e.g. connected components). We will have to tackle the problem of skewed data, and study the possibility of using multiple hashing functions in parallel to improve recall. Finally, we plan to compare our algorithm with algorithms that build a k-NN graph using the bag-of-words (BOW) model.

Acknowledgment

This work has been partially supported by the EU project BigFoot (FP7-ICT-317858).

References

[1] Mikhail Belkin and Partha Niyogi. Laplacian eigenmaps for dimensionality reduction and data representation. *Neural Comput.*, 15(6):1373–1396, June 2003.

[2] Jie Chen, H Fang, and Y Saad. Fast approximate k NN graph construction for high dimensional data via recursive Lanczos bisection. *The Journal of Machine Learning Research*, 10(2009):1989–2012, 2009.

[3] Michael Connor and Piyush Kumar. Fast construction of k-nearest neighbor graphs for point clouds. *IEEE transactions on visualization and computer graphics*, 16(4):599–608, 2009.

[4] Thibault Debatty, Pietro Michiardi, Olivier Thonnard, and Mees Wim. Scalable graph building from text data. In *Proceedings of the 3rd International Workshop on Big Data, Streams and Heterogeneous Source Mining: Algorithms, Systems, Programming Models and Applications*, BigMine '14, 2014.

[5] Wei Dong, Charikar Moses, and Kai Li. Efficient k-nearest neighbor graph construction for generic similarity measures. *Proceedings of the 20th international conference on World wide web - WWW '11*, page 577, 2011.

[6] A. Rajaraman and J. Ullman. *Mining of Massive Datasets*, chapter 3. Cambridge University Press, 2010.

[7] William E. Winkler. String comparator metrics and enhanced decision rules in the fellegi-sunter model of record linkage. In *Proceedings of the Section on Survey Research*, pages 354–359, 1990.

[8] Yan-ming Zhang, Kaizhu Huang, Guanggang Geng, and Cheng-lin Liu. Fast k NN Graph Construction with Locality Sensitive Hashing. In *Machine Learning and Knowledge Discovery in Databases*, pages 660–674. 2013.