Fast Similarity Graph Construction via Data Sketching Techniques

Hoorieh Marefat

A thesis submitted to the Faculty of Graduate Studies in partial fulfillment
of the requirements for the degree of Master of Science

Graduate Program in Electrical Engineering and Computer Science
York University
Toronto, Ontario

September 2021

Abstract

Graphs are mathematical structures used to model objects and their pairwise relationships. Due to their simple but expressive abstract representation, they are commonly used to model various types of relations and processes in technological, social or biological systems and have found numerous applications. A special type of graph is the similarity graph in which nodes represent entities and there is an edge connecting two nodes if the two entities are similar based on some similarity measure. In a typical scenario, raw data of entities are provided in the form of a relational dataset, matrix or a tensor and a similarity graph is built to facilitate graph-based analysis like node importance, node classification, link prediction, community detection, outlier detection, and more.

The ability to construct similarity graphs fast is important and with a potential for high impact, thus several approximation techniques have been proposed. In this work, we propose data sketching based methods for fast approximate similarity graph construction. Data sketching techniques are applied on the raw data and are designed to achieve desired error guarantees. They can drastically reduce the size of raw data on which we operate, allowing for faster construction and analysis of similarity graphs, but with approximate results. This is a desirable tradeoff for many applications in diverse domains.

Through a thorough experimental evaluation, we demonstrate that our sketching methods outperform sensible baselines and competitor methods proposed for the problem. First, they are much faster than exact methods while maintaining high accuracy in constructing the similarity graph. Furthermore, our methods demonstrate significantly higher accuracy than competitive methods on generic graph analysis tasks. We demonstrate the effectiveness of our methods on different real-world graph applications.

Acknowledgements

I wish to express my deepest gratitude to my supervisors Professor Aijun An and Professor Manos Papagelis without whose persistent help and guidance, I could not pass this road. I would like to recognize the invaluable assistance that you provided during my study. Thank you for providing me the opportunity to grow my knowledge and learn from you. Your constructive advice as well as criticism always made me to sharpen my mind. I would also like to show my appreciation to my thesis committee members, Professor Natalija Vlajic and Professor Manar Jammal for their valuable comments and constructive feedback. I am very grateful for the time you spent on my project. With your constructive feedback, you helped me to elevate the level of this work significantly.

I would like to extend my sincere thanks to my dear husband, Sina, who had been beside me all way along and in all ups and downs. I want to show my appreciation to you for all the support you gave me. Thank you for being such a great motivator and exemplar for me. Furthermore, I want to thank my parents who nurtured me in my personal, educational and professional life. I cannot show enough how grateful I am for everything you have done for me during my life. Thanks for your endless support and belief in me.

Table of Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

The graph data structure is one of the most prevalent data structures used in Computer Science because of its ability to show relationships between entities in an effective way. The goal of this thesis is to explore how we can build similarity graphs, i.e. graphs in which nodes represent entities in a dataset and edges representing similarities between two entities, in an efficient way.

There are many applications in which the use of a similarity graph can be beneficial, such as nearest neighbor search [32][45][54], collaborative filtering based recommendation [16][13][58], link prediction [28][30][70], clustering [49][65][72], and maximum inner product search (MIPS) [50]. For example, in the recent work of Morozov et. al. [50], it is shown that using similarity graphs, the MIPS problem (which is an essential operation in many machine learning tasks) can be solved efficiently.

Constructing similarity graphs can be very time-consuming as it normally involves the computation of similarities between all pairs of entities in a dataset. Furthermore, this problem suffers from curse of dimensionality. If data is high-dimensional, the speed of processing decreases significantly. There are many applications in which similarity graphs need to be built many times, such as in data streams where insertion, deletion and modification of data

entities occur frequently over time. Improving similarity graph construction can result in significant improvements in the efficiency of the whole application as it is one of the most time-consuming parts of the application.

An example of applications that deal with streaming data is news recommendations. The news articles are time-sensitive and recommendation of news to users is normally based on recent news. In collaborative filtering based news recommendations, computing similarities between users or articles is an essential operation. Using a similarity graph can facilitate such a computation. Due to the dynamic nature of the data, similarity graphs may need to be re-constructed over time as new articles or users are being added and old ones may retire. Thus, in such an application the need for having a fast approach for graph construction rises.

Our methods are suitable for constructing similarity graphs of high-dimensional and sparse datasets. Examples of such datasets include document-word representations in information retrieval or user-item matrices typically found in recommendation systems (e.g., matrices representing ratings of users to movies or restaurants, binary matrices representing transactions of customers buying products, and so on). In these applications, the matrix represents documents or users as rows and the columns represent words in the literature or items (movies, restaurants, products) the users provided a rating for or purchased, respectively. The end result is a matrix with a large number of rows and columns. Such data matrices are typically very sparse, meaning that most values are equal to zero. The semantics of a zero value are that information is not available or is missing. For example, in a popular movie database, users have on average watched 30 movies (out of million movies available). So for each matrix row there are 30 non-zero entries on average and all other values are zero. In these setting, zero entries can be discarded as they don't carry significant information, occupy large amounts of memory, and render complex computations to be slow.

Data sketching makes construction of graphs from the mentioned matrices much more efficient. Using data sketching techniques, instead of working with high-dimensional data matrices, we can work with much lower dimensions and compute pairwise similarities in a much more efficient way. The objective of this research is to develop efficient and effective method for similarity graph construction using data sketches. The method needs to be fast

in terms of running time, as well as accurate compared to constructing a similarity graph based on the original input dataset.

1.2 Contributions

The main objective of this research is to address the problem of similarity graph construction from high-dimensional data. Given raw data represented by a matrix (or sparse vector representation of a matrix) where rows represent entities or instances and columns represent attributes, we aim to propose an efficient and effective method for similarity graph construction using data sketches. The method should be fast in terms of running time, as well as accurate compared to constructing the same similarity graph based on the original input dataset. Below are the contributions of this thesis:

- We introduce novel algorithms for the construction of similarity graphs. In our algorithms, data sketching techniques are used to make the process more efficient. Data sketches are much smaller than the original data and by using them, high-dimensional data is converted to much lower dimensions. In addition, we propose to use an inverted index for pairwise similarity computations, which helps to eliminate many redundant computations and comparisons, saving a huge amount of time.

- We propose three algorithms each of which has a trade-off for speed and space. When there is no restriction on the space or linear space is provided, two of our algorithms having data preprocessing (which we call offline methods) can be used. Another method we propose has an online approach of data processing and uses much less space than the other two. However, it is shown that this method works slightly slower. So based on the needs one might have, any of our algorithms can be selected and worked with.

- We demonstrate the effectiveness and efficiency of our methods via experiments on different datasets and different setting of parameters. We compare our proposed algorithms with established baselines and a commonly used competitor and show that our

methods are significantly faster than the baselines (up to **62**× faster) with a negligible sacrifice on the accuracy. While the competitor method has a similar speed to ours but its accuracy is much lower.

- We analyze the effectiveness of our methods on different graph downstream tasks like node centralities, node rankings and k-nearest neighbours to see how errors of sketching propagate to the graph applications. We show that compared to our competitor, our accuracy is much higher on such tasks suggesting that our methods can be good options for similarity graph construction and further graph analysis.

- We show that our methods show consistent behaviour even when the parameters of the experiments are changed. For the sketching as well as the graph applications, we change parameters specific to the task and show that our methods are much less sensitive to those changes than the competitor which makes them more robust and reliable to use in different settings.

1.3 Outline of The Thesis

The structure of the thesis is as follows: In section 2, we provide the background of our work and talk about different lines of research that are related to our topic. We provide different categories and introduce some works for each of them. In section 3, we introduce all definitions and our problem statement and elaborate on them. Also, we propose our solutions for the mentioned problem and the methodology we have towards it. Next, it would be the evaluation which we bring in section 4. We show thorough evaluations on how our algorithms and other competitors work based on different criteria. Section 5 is the last one in which we offer a summary and conclusion of our work.

Chapter 2

Background & Related Work

In this section, we go through the background of our work and the basic structures that are used throughout the thesis. Beside that, we review different lines of research that are related to our work and elaborate them.

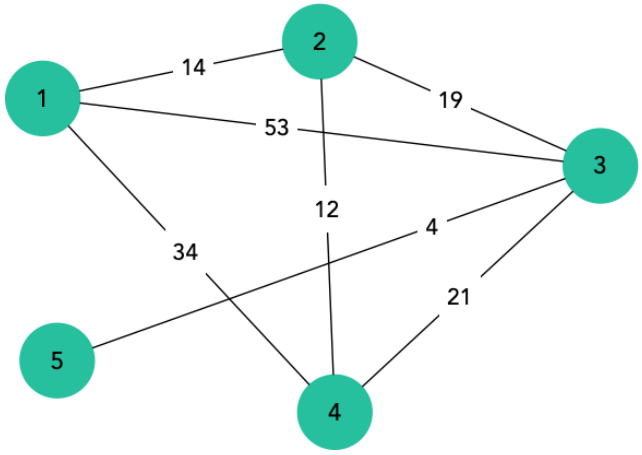
2.1 Different Similarity Related Graphs

2.1.1 Similarity Graph

A similarity graph $G(V, E)$ is a graph where V is the set of vertices or nodes representing entities and E is the set of edges representing similarities between those entities. Figure 2.1 shows an example of such a graph. According to the figure, consider we have a dataset representing users as rows and items they bought as columns. Each cell shows how many numbers of an item a user has bought. If we use dot product as the measure of similarity, users 1 and 2 will have the similarity value $3 \times 4 + 0 \times 5 + 2 \times 1 + 0 \times 0 + 10 \times 0 = 14$, so there is an edge with value 14 between the node 1 and 2. For users 1 and 3 we have similarity value of 53 and the edge between them has value 53. The same applies to all other pairs of users. As it can be seen, the similarity value between user 1 and 5 is 0 so there is no edge between these two nodes in the similarity graph. To conclude, a similarity graph is a graph in which nodes are entities and edges are the similarity between them. Such graphs

UserID	Item1	Item2	Item3	Item4	Item5
1	3	0	2	0	10
2	4	5	1	0	0
3	1	3	0	1	5
4	0	2	2	0	3
5	0	0	0	4	0

(a) Dataset



(b) Similarity graph

Figure 2.1: In subfigure 2.1a, a dataset is shown which represents users as rows and items as columns. Each cell shows how many numbers of an item a user has bought. In subfigure 2.1b, the similarity graph of the presented dataset is built. Nodes of this graph show users and edges are similarity of them based on the *dot product* measure.

have many different applications in recommendation systems, nearest neighbour search, information retrieval and so on.

2.1.2 ϵ-Graph

An ϵ-graph is a usual graph with the difference that all the edges have value above the ϵ threshold. Consider the graph presented in Figure 2.1, if we set $\epsilon = 15$, the edges that are below this value i.e. $(1, 2), (2, 4), (3, 5)$ will be omitted. In Figure 2.2, the idea is presented. Left graph shows the original one and right graph shows its ϵ-graph.

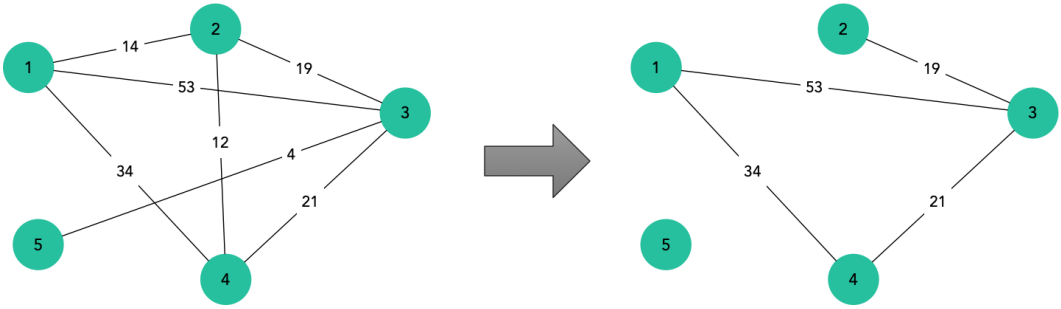


Figure 2.2: Applying $\epsilon = 15$ threshold to the graph presented in Figure 2.1. Edges having weight below the threshold are omitted which results in removal of $(1, 2), (2, 4), (3, 5)$

2.1.3 Nearest Neighbour Graph

A nearest neighbour (NN) graph is a directed graph in which each node has a directed edge to its nearest neighbour node. The notion of nearest can be regarded as the most similar or the least distant. The edges are directed because if node B is nearest for node A, this does not imply its reverse and node B can have another node like C as its nearest neighbour. In Figure 2.3, we show the NN graph built from the similarity graph represented in Figure 2.1.

Beside NN-graphs we have kNN-graphs which are more general. In kNN-graphs, node A has a directed edge to node B if node B is among the k^{th} most similar or least distant nodes of the graph for node A. So for each node, instead of 1 nearest neighbour node, we have k nearest neighbours.

Nearest Neighbour Search (NNS)

In this problem, we are given a dataset and a query point, we try to find the nearest point or the most similar one to the query in the dataset.

2.2 Similarity Graph Construction & Applications

Similarity graph construction can be a daunting task in terms of runtime and memory usage if not operated efficiently. This task includes computing similarities for all pairs of instances in the dataset. Consider the example of users and the items that they have bought as brought in Figure 2.1a, if we have n number of users and d number of items, the time

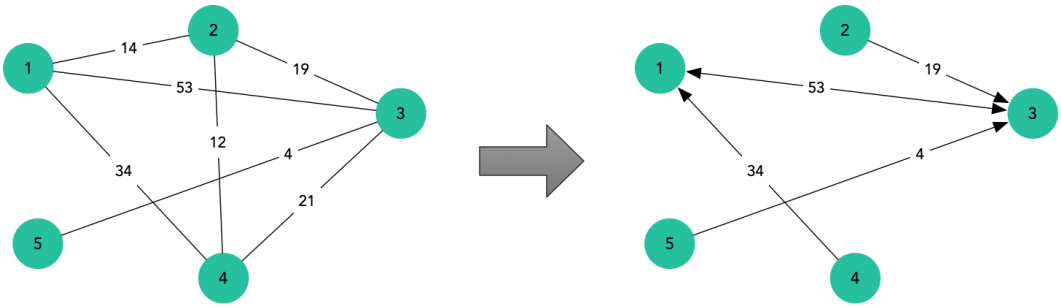


Figure 2.3: Nearest Neighbour Graph constructed from the similarity graph represented in Figure 2.1. As we used *dot product* for the similarity measure, for each node we choose the node that has the highest value of similarity. As it can be seen for node 4 nearest neighbour is node 1; however, nearest neighbour of node 1 is not node 4 and instead it is node 3. So the edge between nodes 1 and 4 is not bi-directed. However, because nearest neighbour of node 3 is node 1 and for node 1 also it is node 3, there is a two-way directed edge between them.

complexity would be $O(n^2d)$. This process would be very time-taking not only in terms of n, but also when dimension d is very large. There are a number of works proposed to speedup the pairwise comparisons by the usage of GPUs or distributing the data to address the issue of having large number of instances in the dataset. Furthermore, working with less number of dimensions when d is too large can boost the process significantly too. However, there are a few works presented to tackle the problem of dimensionality and our work is one of them. In this section, we provide different approaches for pairwise similarity computations.

2.2.1 Pairwise Similarity Search Using Inverted Indices and Distribution

In this category, inverted indices are used for speeding up the process of similarity search. An inverted index is the opposite of a forward index. An illustration is provided in Figure 2.4. If we have documents that in each of them a geo-scopeID is present, a forward index maps each docID to a geo-scopeID; while an inverted index does the reverse and for each geo-scopeID it shows which documents have them. Using inverted indices makes the process of finding similarities much faster.

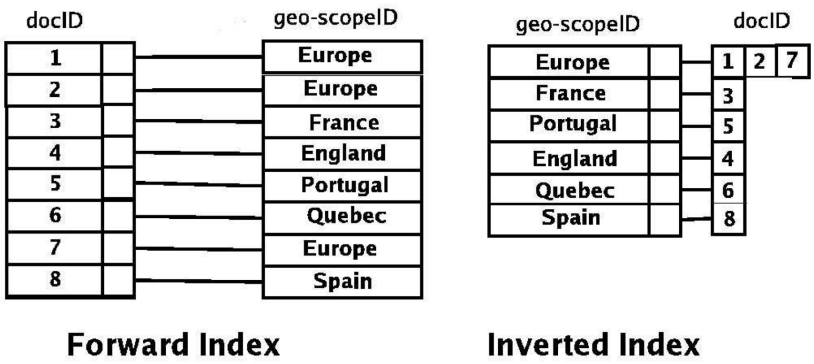


Figure 2.4: From Andrade et.al. in *Indexing Structures for Geographic Web Retrieval.* If we have documents that in each of them a geo-scopeID is present, a forward index maps each docID to a geo-scopeID; while an inverted index does the reverse and for each geo-scopeID it shows which documents have them.

All-Pairs Similarity Search

Bayardo et. al. [11] give a solution on how one can make use of these indices for finding similarities between all pairs of instances in the dataset that are above a specific threshold. They propose exact solutions rather than approximations. In this work a basic algorithm for finding pairwise similarities using inverted indices is brought. Then, they provode two levels of refinement on the basic algorithm to make it even faster. The basic algorithm builds inverted indices gradually. While each instance of the dataset is being put in indices, its similarity to the instances that are already indexed is computed. So the details of the algorithm are as follow: It starts with empty initialization of inverted indices. Then there is a pass over all instances of the dataset. For each instance, similarity score of all indexed instances to the current one is calculated. Scores are maintained in a dictionary and updated during time. When score computation is done, the current instance would be indexed as well and the process goes on till the last instance. Finding similarity score is done very easily. Suppose we have some indexed documents and we want to find similarity of a document to the ones that are indexed. For each word the query document has, we go to its index and for each document having that word, we update the pairwise similarity score. This is the basic algorithm suggested by Bayardo et. al. [11].

They propose two refinements to this method exploiting the similarity threshold. The threshold is used to reduce the amount of information indexed. This has a great impact because scanning indices takes less time and candidates are reduced. However, these two approaches are impractical as for the input they need a special sort and order on dataset, both on columns and rows. This special sort and order is not present in real-world datasets and preparing a dataset for these algorithms is a huge bottleneck. Authors assume that the mentioned dataset already exists and do not consider the preprocessing of data for that. There are other works using inverted indices for similarity search; some of them using distributed frameworks and some using GPUs. We elaborate them in the next sections.

Distributed Solutions

In order to make pairwise similarity search scalable in terms of number of instances in the dataset, one can use distributed solutions. Consider we have large amount of data residing in different locations. A distributed solution works with very large data not residing on just one machine and on multiples of them in a cluster.

MapReduce is a distributed programming model. The general structure of a MapReduce framework is shown in Figure 2.5. Each node in the cluster maps its data to some intermediate outputs. The output consists key/value pairs. In the shuffle stage, output of map stage is grouped based on the keys they have. The pairs having the same key are grouped together and are put on the same machine. As the last step, in the reduce stage all the groups from previous stage are processed in parallel using reduce functions to give the final output.

Elsayed et. al. [22] give a distributed solution for similarity search using MapReduce. In this method, there are two steps of MapReduce jobs as shown in Figure 2.6. In the first one, documents are indexed and inverted indices are built which is shown in the figure as *Indexing* part. The second step is finding pairwise similarities based on the inverted indices shown as *Pairwise Similarity*. In the indexing step, documents are given to mappers. The mapper produces key/value pair tuples having terms as keys and value as tuple of document ID and the frequency of the term in the document. In the shuffle stage, all the tuples having

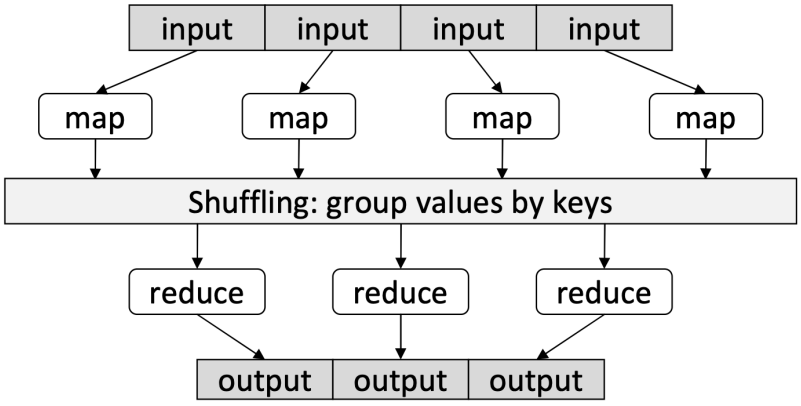


Figure 2.5: This is a representation of MapReduce structure provided in [22].

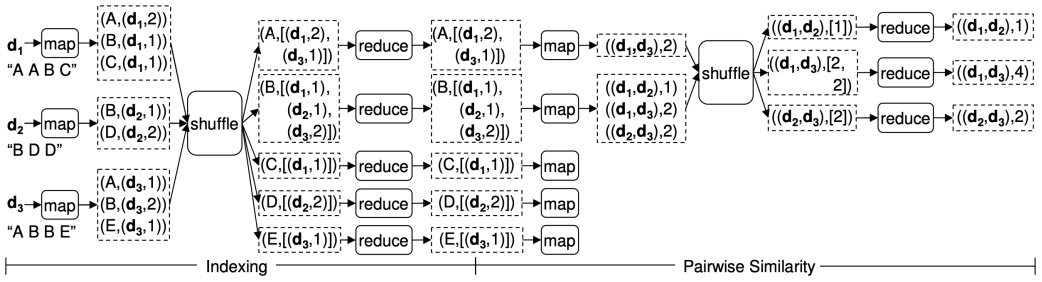


Figure 2.6: MapReduce structure used for similarity search suggested by [22]. There are two steps of MapReduce jobs. In the first one, documents are indexed and inverted indices are built which is shown in the figure as *Indexing* part. The second step is finding pairwise similarities based on the inverted indices shown as *Pairwise Similarity*.

the same key (the same term) are gathered and put together on the same machine. The output of the shuffle stage is the inverted indices we desired so reducers just simply forward the input they are given. In pairwise similarity step, each mapper produces all the pairwise combinations of the document IDs for each term. This would be the key and the product of their weights serves as the value. Then, in the shuffle stage same pairs of documents are gathered and put together. Then, the reducer sums all the similarity values of the pairs to give the final simialrity value.

The work presented by Elsayed et.al. [22] is an algorithm giving the exact answer for all similarities. However, huge amount of data should be shuffled in the network between different machines and that is a problem. Lin [44] provides other solutions for this problem.

He proposes a brute force algorithm as well as two other ones using indexing techniques. They are called Parallel Queries and Postings Cartesian Product. The latter one is a refinement of Elsayed's work described earlier. He suggests that we move some of the works that reducers do to the map stage in order to reduce amount of data being transferred.

Furthermore, Phan et.al. [57] provide a MapReduce framework that any kind of similarity search related tasks for pairwise documents, a pivot document, range queries and k-nearest neighbour queries can be answered. The overview of their framework is provided in Figure 2.7. As the first step, worksets which are sets of documents are given as input to the framework. Then, a MapReduce job is done on these documents. This MapReduce job does a prior filter on the input which includes elimination of duplicate, common and the lonely words. The output of this step is a customized inverted index which is used for the second MapReduce job. In this step inverted indices are used and pairwise similarities are computed based on them. Query parameter filterings can be applied here in order to change the similarity search scenario to the desired task including pre-pruning, kNN query and range query filterings. In addition, Phan et. al. provide an extension to this solution [56] adding normalization steps while building inverted indices. The main structure is very similar except that 2 more MapReduce jobs are added to do the desired normalizations.

Dimension Independent Similarity Computation (DISCO)

This method is proposed by Bosagh Zadeh et.al. [69] and is used to compute all pairwise similarities between high-dimensional vectors. It is implemented on MapReduce framework. In this work, authors optimized the amount of data shuffled in the network as well as the reduce-key complexity which is defined as the maximum number of items that are reduced to a single key. These optimizations have a great impact on efficiency of the algorithm.

The method works independent of number of dimensions. Authors propose a special sampling technique in which points that have many nonzero dimensions are sampled with lower probability than points having few dimensions as nonzero. This is the key to the whole dimension independent computations. Then, the output of the reducers are random variables whose expectations are the similarities and they prove accuracy theoretically.

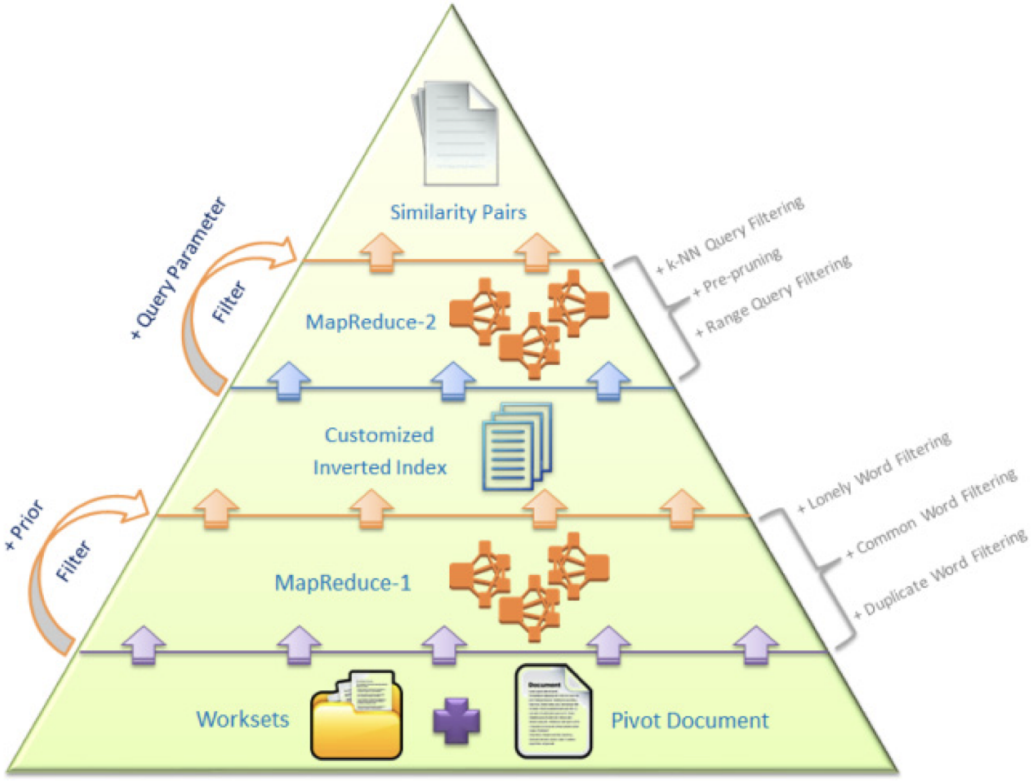


Figure 2.7: The overview structure of similarity search related tasks proposed in [57]

2.2.2 Pairwise Similarity Search on GPUs

In this category, the focus is on maximizing parallelization of processing by using GPUs. In the work presented by Zhou et. al. [71]. an inverted index based framework is proposed to find similarities on GPUs. The overview is shown in Figure 2.8. On the top left, a relational table is provided. Each row of this dataset can be represented as a set of tuples showing attributes and their values. $O_1 = \{(A, 1),\ (B, 2),\ (C, 1)\}$ is an example. $Q1$ and $Q2$ show two example range queries. For $Q1$, we are querying tuples having $1 \leq A \leq 2,\ 1 \leq B \leq 1,\ 2 \leq C \leq 3$. On the right side of the Figure, the inverted index structure is presented. Keys of the inverted index are tuples of attribute headers and values, posting lists show which objects have those tuples. The count table shows how many number of tuples in the query each of the objects of the dataset contain. This count shows similarity of objects to the query so similarities can be obtained easily with this structure on GPUs.

As explained, this work focuses on finding similar objects to the query and is not an all pairs similarity search. For that purpose, a data structure is designed for the count table. This data structure is called Count Priority Queue (c-PQ) which makes the process of finding top k similar objects much more efficient. It is a hash table that keeps a few candidates and for getting the query result, it should be scanned just once. Another novelty of this work is the ability to work with many different kinds of data. The popular data types are supported by Locality Sensitive Hashing (LSH) [24] and the complex ones are supported by Shutgun and Assembly (SA) [8]. By using these two techniques, many data types are supported to build an inverted index based on them and searching queries.

Furthermore, a fast GPU k-selection algorithm is proposed in [34]. In this work the problem of finding k nearest neighbours to a query is investigated and it is not an all pairs similarity search. When working with GPUs, compressed representations of vectors are very convenient to use because memory is limited. Instead of working with a very high-dimensional vector, one can use a small and compact representation of it using different encodings. Compression can be done using binary codes or quantization. In this work, Product Quantization (PQ) is used. Furthermore, implementations on GPU have their own limitations and a CPU design cannot be translated to a GPU one in a straightforward way.

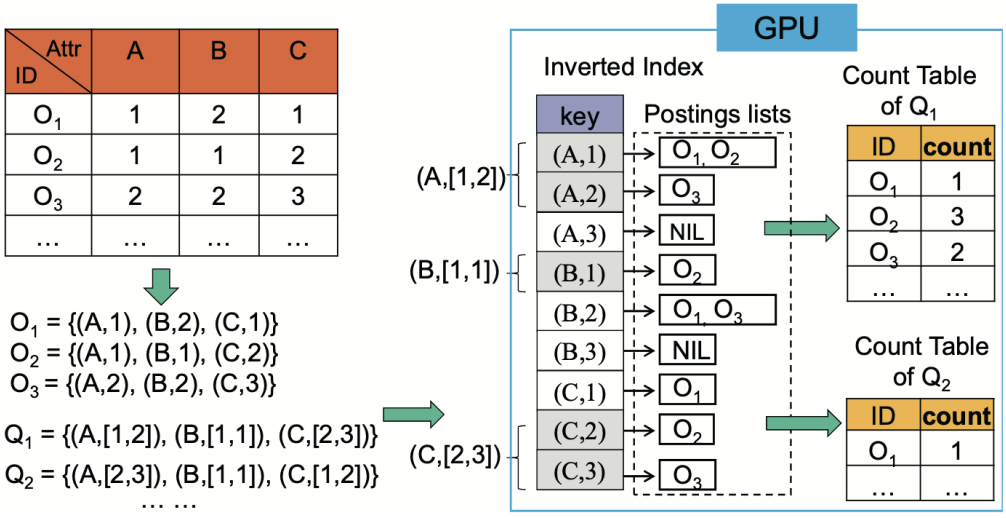


Figure 2.8: This is the overview of the method suggested by Zhou et. al. [71] for finding similarities using GPUs.

This work provides different GPU layout decisions to make use of the power of parallelism they have as much as possible and make the process much faster.

As another example, Jian et. al. [31] propose a method for top-k cosine similarity search via a special binary quantization technique. This method does not use any special indexing construction. The algorithm compresses floating point numbers to very small binary codes and does an XOR on them.

Gowanlock et.al. [27] propose a solution for the problem of similarity self-join on GPUs. In this problem, all entities in the dataset that are within ϵ distance of each other are found. They provide a grid-based specific index structure suitable for GPU in order to do range queries. By using properties of this index, some candidate sets are filtered. This helps to reduce overhead of the search. Also, based on the variance of dimensions, data is reordered. Furthermore, some expensive calculations are pruned to improve performance of the overall search. By using these optimizations, the whole process is performed with much higher efficiency.

2.2.3 Similarity Graph Applications

Similarity graphs are widely used in the literature and there are numerous applications for them. Here we bring some of them and elaborate.

Recommendation Systems & Collaborative Filtering

Recommendation systems aim to predict preference of a user about an item in order to give a recommendation; for example, in music or video applications when the system wants to recommend a music or a video to a user or in online shopping when an item is suggested, etc. One of the main approaches to address this problem is Collaborative Filtering. This approach is based on the fact that if two people have had the same taste in the past, in the future they will like similar items too. One way of solving this problem is using similarity of users for which a similarity graph can be used. Therefore, an item that is enjoyed by similar people to the user can be recommended to him/her [16][13][58].

Link Prediction

The link prediction problem is the problem of predicting future links between different nodes in the graph; for example predicting friendship of users in a social network, finding hidden links for security purposes like detecting gang interactions, etc. There are a number of approaches to solve this problem. One of the most successful ones is the similarity based methods which are widely used. These methods consider the structure of the network and having similarity graphs helps the process greatly. Some of them work based on node neighbourhood features like common neighbours, Jaccard coefficient, etc., Others work based on path features like shortest path distances, Katz metric, ... or even based on vertex and edge attributes [28][30][70][68].

Data Clustering

Another application of similarity graphs is data clustering [49][65][72]. In clustering, we want data points residing in the same cluster to be more similar to each other than data points in other clusters. So we want to maximize similarity of items within a cluster and

minimize it inter-cluster. For this purpose, a similarity graph can be used. We can find minimum cuts recursively in order to partition or cluster dataset.

Maximum Inner Product Search (MIPS)

The MIPS problem has many applications in Machine Learning. The problem is formulated as following. Given a large dataset of vectors $X = \{x_i \in \mathbb{R}^d \mid i = 1, ..., n\}$ and a query $q \in \mathbb{R}^d$, we need to find the vector x_j such that $\langle x_j, q\rangle \geq \langle x_i, q\rangle = x_i^T q\ ,\ i \neq j$. As suggested in [50], previously, people were using two approaches to solve this problem. First, reducing MIPS to an NNS problem using traditional methods like tree partitioning, LSH, etc. The second approach was to filter vectors that are not promising based on inner product upper bounds like Cauchy-Schwarz inequality. In the work presented by Morozov et.al. [50] a new approach is presented. They propose that by using similarity graphs, this problem can be solved in a more efficient way. The authors provide theoretical analysis to show why similarity graphs are very effective in solving this problem.

Nearest Neighbour Search (NNS)

One of the applications that similarity graphs can be used in is NNS. Because a large body of works is offered in this area, we devote a section to this topic and elaborate the main approaches to solve this problem. Although this topic is very related, it is not exactly the same as similarity graph construction. As shown in Figure 2.3, a similarity graph is a weighted undirected graph in which each edge shows similarity between two nodes. However, an NN graph is a directed one in which each node has a directed edge pointing to its nearest neighbour. There are many works proposed to solve the problem of NNS; some of them make use of similarity graphs and some build special data structures optimized for the query search and answer the query with those data structures. In the next section, we provide different categories of work in this area and elaborate them.

2.3 Nearest Neighbour Graph Construction and Search

The problem of nearest neighbour graph construction and search has many applications in Computer Science. In information retrieval systems where we want to find the most similar documents to a query, content based image retrieval, optical character recognition and many other applications. Because datasets can be very large in terms of number of instances as well as dimensionality of dataset, giving an exact solution might be very time-inefficient. Therefore, there are many approximated solutions proposed that are accurate enough but much more efficient than the exact solution.

Similarity graph construction is related to NN-graph construction but it has its own differences. In a similarity graph we have all the pairs of similarities that are nonzero as edges. However, in a k-NN graph we have just the most k similar nodes for each node and we can optimize k-NN graph construction process accordingly. However, similarity graphs can be used for k-NN search too. The problem of k-nearest neighbour search is defined as having a query and a dataset including different data points, we want to return k most similar data points of the dataset to the query. For this purpose, many methods first build an index or a graph and then run a search algorithm on this intermediate data structure. This data structure can be a similarity graph, a tree, etc. Here we provide some of the most prevalent methods for NNS problem.

There are different categories of work in the area of NN-graph construction and search. A couple of surveys exist representing the most important methods in each category [9][43][55]. One category of the methods build a graph and based on that graph nearest neighbours of nodes or queries are obtained. This graph is built such that information of data points based on their neighbours are used. The main idea is that neighbours of neighbours might be a neighbour. For example, if node A has neighbour node B and node B has neighbour node C, we can check if A is a neighbour of C too or not. The other category builds special trees to answer the search query via partitioning the space recursively. The space is partitioned hierarchically via pivoting or compact partitioning techniques like clustering, Voronoi partitions or random division of space. The last important category includes methods that use

Locality Sensitive Hashing (LSH) as their foundation. In these methods, the goal is addressing the problem of dimensionality. Multiple hash functions are applied on data and high dimensional data points are mapped to lower dimensions or short codes consisting sequence of bits. There are two points of view. One is that we hash data points such that similar instances be hashed to the same code. The other one is that we hash high-dimensional points to lower dimension codes and find similarities in this low dimensional space. LSH based NNS methods are the most related to our work and we provide comparisons with them in the experiments section.

2.3.1 Neighbourhood or Graph Based Methods

These methods build a graph that for each data point its neighbourhood information with other points of dataset or a set of pivot points are kept [43]. So a k-NN graph or other graph types are built and using them, NNS is answered. In these methods, when searching for the nearest neighbour of the query, the graph is traversed in a greedy way and search stops when a stopping condition is met. In the following, we bring some of the main approaches belonging to this category.

Small World

In the real-world, the small world phenomenon explains that each two strangers in the world can be linked by a small number of other people as hops. A graph having small world feature has following properties: each of its nodes are connected to a small proportion of the whole nodes of the graph, if a node has two neighbours it is very probable that these two can be neighbours of each other and as the last one, each two nodes of the graph can be linked to each other via a small number of hops in the middle. A very important feature of such graphs is that the diameter of the graph with N number of nodes is bounded by $\log(N)$ [38].

In the paper presented by Malkov et.al. [46], a small world graph is used for approximate nearest neighbour (ANN) search. This graph is constructed gradually by adding data points to the graph one by one. When a node is inserted, set of its closest neighbours are detected

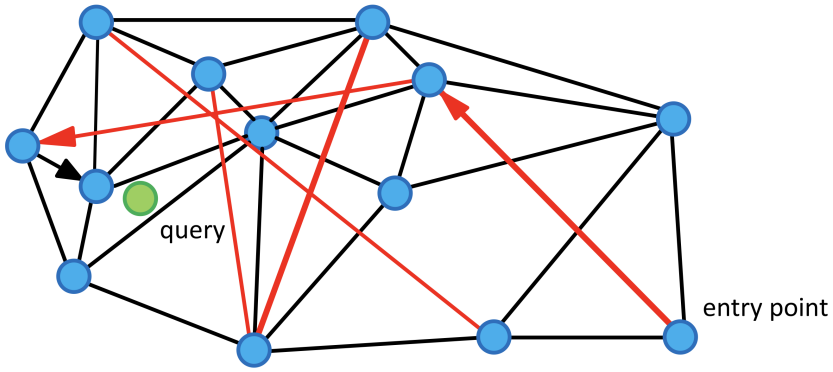


Figure 2.9: The Small World graph structue suggested by Malkov et.al. [46]. Blue nodes represent nodes of the graph. The black edges show the approximation of Delaunay graph. The red edges are long range links. In this figure it is shown that by starting from the entry point, which links are traversed to get to the closest node to the query.

and an edge is added between them which is an approximation of the Delaunay graph. These edges are regarded as short range links. By ongoing insertion of nodes, short range links become a long range link. This makes this graph a Navigable Small World (NSW). For searching the graph to find nearest neighbours of a query, a number of subsearches are done. In each of them, we start with a random node and visit its closest neighbours which are not visited before until the stopping condition is met. Figure 2.9 shows the structue of the mentioned Small World graph. Blue nodes represent nodes of the graph. The black edges show the approximation of Delaunay graph and the red edges are long range links. In this figure it is shown that by starting from the entry point, which links are traversed to get to the closest node to the query.

Hierarchical Navigable Small World

Malkov et.al. present another method based on NSW graphs. In Hierarchical Navigable Small World (HNSW) method [47], a multi-layer hierarchical graph is built. Figure 2.10 shows the mentioned HNSW graph in which links of the graph are separated based on their distance scale or length. The most upper layer has the longest links and the closer to the zero level, length of links become smaller. Also, number of points in the upper most layer is the least and the zero layer contains all data points. Each layer of this graph is a

proximity graph and edges of each layer are short links approximating the Delaunay graph. If we combine all connections of all layers, we end up with an NSW graph (like the one we elaborated in Small World section).

Construction of this graph is done by inserting data points one by one. Each node has a maximum layer like l which is selected randomly and the node is inserted to all layers between $0 - l$. To insert each node, first we start from the top most layer. We explore this layer and find the neareset neighbour (a local minimum) for the point. Then we proceed to one layer down. In this layer, we start our search from the local minimum point we obtained from the previous layer and recursively we find local minimum in this layer too. We do this process for all the layers of top most to l. Then, for layer l, we insert the node in the graph connecting it to its nearest nodes and go down till the 0 level. The search process is similar to the insertion process except that we assume $l = 0$ for the query and give the neighbours in the ground level as the result.

2.3.2 Tree Based Methods

This category includes tree based space partitioning methods. The space is partitioned hierarchically via pivoting or compact partitioning techniques. In the pivoting methods, we have some points as pivots and partitioning is done based on the distance of data points to those pivots. Compact partitioning methods have another point of view. One approach is to use clustering on data points, another one can be based on Voronoi partitions or random division of space [43]. We provide two of the main methods in this category in the following.

FLANN

FLANN is an open source library which stands for Fast Library for Approximate Nearest Neighbours. The work which presents FLANN [52] provides two different approaches for ANN search: *multiple randomized kd-trees*[59] and *priority search k-means tree*.

A kd-tree is a space-partitioning data structure in a k-dimensional space. In the construction of a randomized kd-tree, data points are recursively divided to two halves. For division, a dimension is selected and split is done using a perpendicular hyperplane. This

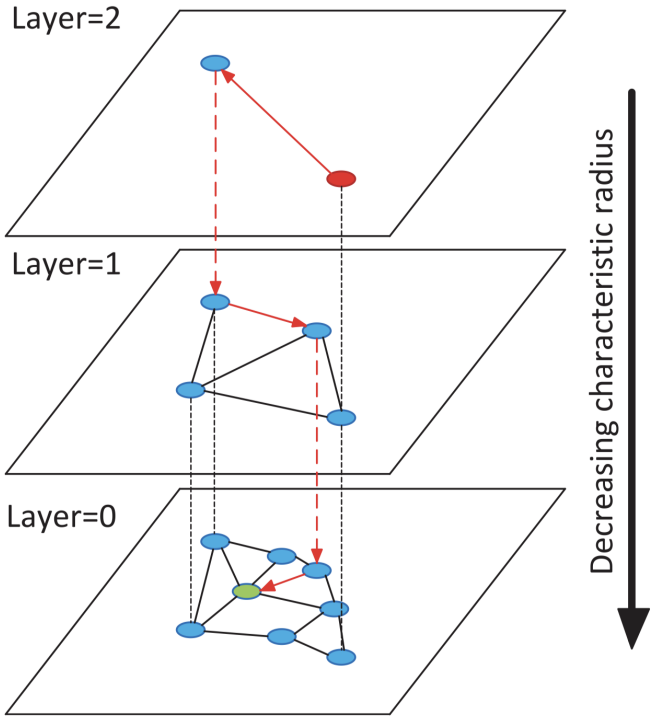


Figure 2.10: This figure shows the structure of HNSW graph suggested in [47]. Level 0 has the shortest links and as we increase levels, links become longer and number of nodes decrease. Each layer is a proximity graph by itself. Figure shows a search process. Search starts from the top most layer (layer 2) and from the red node. A local nearest neighbour node is found in layer 2 and that node is the entry point to the layer 1. In layer 1 also the local minimum is found and served as the entry point for layer 0. The nearest neighbour in the ground level (0) would be the result of this search.

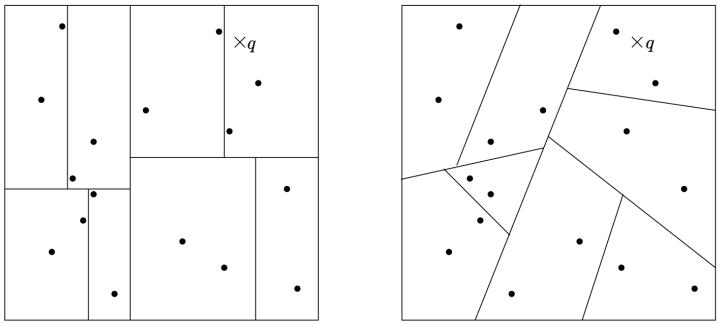


Figure 2.11: Spatial partitioning of $\mathbb{R}^2$ by a k-d tree shown in left vs. an RP tree shown in right [19].

hyperplane passes mean value of data points in the selected dimension. But how is the dimension selected? It is chosen from top 5 ones that have the largest variance for the values of data points in that dimension. In this method, multiple randomized kd-trees are built and for the query they are searched in parallel. The priority search k-means tree works based on clustering. It clusters data points based on all their dimensions contrary to the previous method which uses just one dimension to partition. In this method, data points are clustered to k regions using k-means and then on each region recursively the same method is applied. The stopping condition for each region is met when the region has less number of points than k.

Annoy

This method is one of the most successful ANN algorithms and is being used in Spotify.com. In this method, multiple random projection trees are built [19]. In Figure 2.11, space partitioning done by a k-d tree vs. an RP tree is shown. At each step, space is partitioned to two subspaces by taking two random nodes in the subset and dividing the space by the hyperplane that is equidistant to them. This is done k times and a forest of trees is built. This method is empirically designed and implemented.

2.3.3 LSH Based Methods

Locality Sensitive Hashing (LSH) was first introduced in [24]. Since then, this method has been vastly used in ANN search algorithms. This method provides theoretical guarantees on the quality of the result for the query. The main idea is to address the problem of dimensionality. Multiple hash functions are applied on data and high dimensional data points are mapped to lower dimensions or short codes consisting sequence of bits.

There are two main approaches of using hashing. One is to use locality sensitive property which says that similar items have larger probability to be mapped to the same hash bucket (or code) than the ones that are not similar to each other. Therefore, NN candidates of the query are explored in the bucket that the query is hashed to. The other usage of hashing is to reduce dimensions by using short codes and then finding approximate distances on these short codes instead of the high-dimensional original space. This needs that distance computation using short codes is efficient and also these codes preserve original similarities of data points [67]. Now, we bring some methods in this category.

SRS

This method [66] is used for doing an ANN search in high-dimensional Euclidean space. LSH is used to tackle the problem of curse of dimensionality. In this method theoretical guarantees are provided for the quality of the results. Also, it is said that the index size used by LSH algorithms is too large but authors propose that a tiny index is required in this work. The main idea here is to project high-dimensional data points to low-dimensions (as low as 10 dimensions) via 2-stable random projections. The goal is that the distance between points in the projected space over their distance in the original space follows a known distribution. If we show distance of query q to the point o in the original space as $dist(o)$ and in the projected space as $\Delta_m(o)$, $\frac{\Delta_m^2(o)}{dist^2(o)}$ follows $\chi^2(m)$ distribution with mean m. This means that in this method the algorithm reduces d-dimensional c.ANN query to an m-dimensional exact k-NN query.

Scalable Graph Hashing (SGH)

In the work presented by Jiang et.al. [32] a new graph hashing technique is proposed. Graph hashing methods need to compute pairwise similarities between all pairs of points in the dataset. However, due to the time and space limitations, approximations or subsampling is needed. In this work, authors provide an approach to approximate and use the similarity matrix without explicitely computing it. The approximation used in this work is as follows: Consider we have a dataset having n data points with dimensionality as d. By hashing, each data point is mapped to a binary code with the length c meaning that there are c binary hash functions $\{h_k(.) \mid k = 1, 2, ..., c\}$ based on which the binary code of any point like $\mathbf{x}$ is computed, i.e. $\mathbf{b} = [h_1(\mathbf{x}),\ h_2(\mathbf{x}),\ ...,\ h_c(\mathbf{x})]^T$. Hashing should preserve distance of the points in the original space. So if two points in the original space are similar to each other, their Hamming distance in the new space should be small too.

Two important elements of this work are the objective function and the feature transformation method. The objective function is as follow:

$$\min_{\{\mathbf{b}_l\}_{l=1}^n} \left(\sum_{i,j=1}^{n} (\tilde{S}_{ij} - \frac{1}{c}\mathbf{b}_i^T \mathbf{b}_j)^2 \right) \tag{2.1}$$

where $\mathbf{b}_i$ is the binary code of data point $\mathbf{x}_i$, n is number of points and $\tilde{S}_{ij} = 2S_{ij} - 1$. S_{ij} is the similarity value between points $\mathbf{x}_i$ and $\mathbf{x}_j$. By using feature transformations, the similarity graph matrix $\tilde{S}$ is approximated without being explicitely computed.

2.4 Data Sketching and Sampling Techniques

In this section, we provide some of the sampling and sketching techniques which help to reduce size of the dataset. This size reduction is in terms of dimensionality of the dataset.

2.4.1 Conditional Random Sampling (CRS)

CRS [41] is a combination of sampling and sketching which reduces dimensionality of data. It is used for approximating inner product, l_2-distance and l_1-distance. It has two stages, one

is creating sketches of data in order to make them smaller and the other is using sketches for the estimation of pairwise similarities. Steps of sketching are shown in Figure 2.12. Columns of the dataset are first permuted in order to make order of columns random. Then zero entries of rows are discarded and the nonzero entries remained are called *postings*. Postings contain key/value pairs showing column headers and their values. Sketches are front of the postings which means a few of the nonzero entries of the rows. Sketches for different rows can have different lengths. In the estimation stage, for each pairwise comparison of two rows of sketches, the length of the sketch involved in the computation is conditioned on the length of two sketches we are opertaing on and is called D_s. Consider we have two sketches K_1 and K_2 having key/values as ID/value. $D_s = min(max(ID(K_1)),\ max(ID(K_2)))$ and it shows what is the min of the maximum IDs in both sketches. This D_s value is computed pairwise and is used in estimating pairwise similarities. In the following we bring different similarity measures provided in this work and show how they are approximated using sketches. This method also provides theoretical variance for estimations.

We bring inner-product, l_1-distance and l_2-distance for original data as a, $d^{(1)}$, $d^{(2)}$ and then also provide approximations of them for the conditional random samples as $\hat{a}$, $\hat{d}^{(1)}$ and $\hat{d}^{(2)}$.

$$a = \sum_{i=1}^{D} u_{1,i}u_{2,i} \qquad \hat{a} = \frac{D}{D_s}\sum_{i=1}^{D_s} \tilde{u}_{1,i}\tilde{u}_{2,i}$$

$$d^{(1)} = \sum_{i=1}^{D} |u_{1,i} - u_{2,i}| \qquad \hat{d}^{(1)} = \frac{D}{D_s}\sum_{i=1}^{D_s} |\tilde{u}_{1,i} - \tilde{u}_{2,i}|$$

$$d^{(2)} = \sum_{i=1}^{D} |u_{1,i} - u_{2,i}|^2 \qquad \hat{d}^{(2)} = \frac{D}{D_s}\sum_{i=1}^{D_s} (\tilde{u}_{1,i} - \tilde{u}_{2,i})^2$$

In these equations, D is the dimensionality of the original data and u_1 and u_2 are two rows in the original dataset that we want to measure their similarity. $\tilde{u}_1$ and $\tilde{u}_2$ are the conditional random samples with length D_s obtained by CRS from the mentioned rows. As it can be seen, in formulas used for sketches we have $\frac{D}{D_s}$ which tries to extrapolate the similarity value to all the dimensions in the dataset.

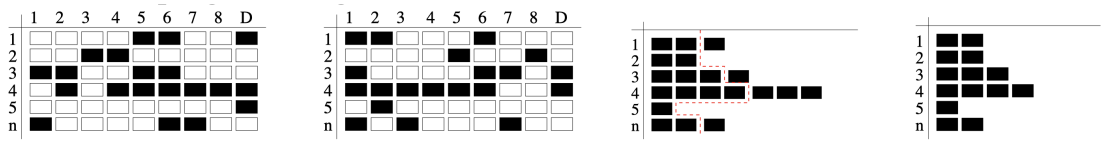


Figure 2.12: The sampling/sketching procedure suggested in [41]. Columns of the dataset are first permuted in order to make order of columns random. Then zero entries of rows are discarded and the nonzero entries remained are called *postings*. Postings contain key/value pairs showing column headers and their values. Sketches are front of the postings which means a few of the nonzero entries of the rows. Sketches for different rows can have different lengths.

CRS is a powerful tool for sampling/sketching because not only it's a simple method, but also makes the process much more efficient. Furthermore, having theoretical bounds makes it more reliable [41][40]. Because of these virtues, we chose this method as the baseline sampling method used in our work.

2.4.2 Normal Random Projection (NRP)

This method is one of the most widely used methods for dimensionality reduction [14][42][1]. Consider we have data matrix $A \in \mathbb{R}^{n \times D}$ with n number of rows and D dimensions. If we multiply this matrix by another matrix $R \in \mathbb{R}^{D \times k}$ whose entries are i.i.d. using Normal distribution of $N(0, 1)$, we end up with a compact representation of $B = AR \in \mathbb{R}^{n \times k}$ in which $k \ll D$. Because the random matrix R consists of i.i.d. entries in $N(0, 1)$, this method is called *normal random projection* [14]. Therefore, dataset is first projected to lower dimensions and pairwise similarities are computed between projected rows.

2.5 Other Related Works

In this section, a few lines of research that are related to our work to an extent are provided.

2.5.1 Graph Sampling

There are numerous works done in this area. Based on a survey proposed by Hu et.al. [29], when the whole graph is known, graph sampling is choosing a subset of vertices and/or edges from a graph such that its size would be smaller but it maintains properties we are interested in. Different techniques are used for this purpose such as vertex sampling, edge sampling, vertex sampling with neighbourhood and traversal based sampling. Choosing each of them, some graph properties are preserved. As a result, we can construct efficient estimators for the original graphs.

Leskovec et. el. [39] explain different approaches for graph sampling as well as different criteria for evaluation. For each graph property, they use its distribution and using D-statistic, agreement of distributions from sampled and original graphs are evaluated. Different sampling algorithms explored in this work are random node selection algorithms, random edge selection and sampling by exploration of the graph. This work does not provide theoretical analysis.

In the work done by Ahmed et.al. [2], a topology based sampling method is proposed. In this method, nodes and edges are not simply sampled, but they use edge-based node selection which makes the probability of selecting high degree nodes higher. The last step of the method is adding some more edges to the sample so that the sampled graph has a similar structure to the original one. Furthermore, different graph crawling approaches are analyzed in [20] like Breadth First Search (BFS), Depth First Search (DFS) and Random First Search (RFS) which all can be categorized as traversal based approaches. Snowball sampling presented by Goodman [26] is another approach that k random nodes are sampled, then like a snowball effect each selected node offers k other nodes with which it interacts the most. Then each of the offered nodes should do the same recursively for s number of stages and the procedure goes on. Random walk can be considered as a special case of snowball sampling with some differences. In random walks each node offers just one another node. There are numerous variants of random walk approaches [48][33][25] but the main idea for all of them is that we start from a node, from that node we visit one of its neighbours and repeat these steps for a specified number of steps and then give all the visited nodes as

sample nodes of the graph.

Although graph sampling is a related area to our work, it has differences to what we propose. For graph sampling, the procedure starts with a graph and then it is sampled; however, our work is dedicated to building the graph. So our work is one step before any sampling can happen, to construct the graph in an efficient way.

2.5.2 Graph Sketching

Beside of graph sampling, we have graph sketching. Although graph sketching focuses on linear measurements of the graph and is different to sampling, in some works these two topic titles are used interchangeably. Hence, we devote a section to explore this topic. Sketching is very useful when we have large streams of data. When we encounter large size of graph data coming and we have to process information continuously, working with sketches makes the process much faster. Also, they help parallel processing of information. Sketches are much smaller in size and yet good representatives of original data.

Cormode provides a survey on this topic [17]. He defines sketch as a linear transformation of a matrix or a vector to lower size and considers it as summary of the data which contains all the information. There are many approaches for graph sketching. Ahn et. al. [3] were pioneers in this area of research. In their method size of graph is reduced from $O(n^2)$ (where n is number of nodes) to d dimensions. They show that different properties of the original graph can be approximated by the sketch graph with high probability. Some of the properties used in this work are connectivity, k-connectivity, bipartiteness and the weight of the minimum spanning tree.

Definitions of dynamic graph sketches are provided in [4]. It is shown that sketches are constructed from linear projections of data. In this paper, authors propose that sketches can be used efficiently in dynamic graph streams where there is a sequence of additions or deletions on edges of the graph. In certain situations where we do not have enough space to store the graph itself, we can store its sketch and whenever needed, construct the original graph based on sketch. Furthermore, they propose that sketches can be used for distributed applications. Sending sketches over the network is much more communication efficient than

sending the whole graph.

There are many other works in this area. One is frequency based for which an example can be count-min sketch [18]. This method uses sublinear space and gives approximations with high accuracy. Compressed sensing is a very related topic in which a transformed signal is reconstructed via linear measurements similar to sketching [21] [53]. Furthermore, there are other works like building sketches for graphs and finding clusters based on them [51], using all_distance sketch as a paradigm of graph neighbourhood sketching [7], topological graph sketching based on min-wise hashing for local neighbourhood [10], etc. Again, all these works start with a given graph and build sketches based on them while our work is dedicated to the graph construction part.

2.5.3 Graph Sparsification

Graph sparsification is used when we have very large (or dense) graphs that consume a lot of storage and computations on them are not efficient. By sparsification, graph becomes much sparser in terms of edges without producing too much error on the computation results. One of the common properties used in different applications is the graph cut. A cut for the graph is defined as a partition of nodes to two disjoint sets. The cut-set consists all the edges that have one endpoint in one partition and the other endpoint in the other. Also, cut value is the number of edges in the cut-set or the total weight of the cut-set edges. The minimum cut problem is the problem to find a cut of minimum value. In graph sparsification, we want to build a sparse graph from the original one and give the min cut values with high accuracy. It is shown in the literature that random sampling is an effective approach for this purpose and the minimum cut in the sparse graph is $(1+\epsilon)$-times minimum cut in the original graph. Karger et.al. [37][35][36] and Benczur et.al. [12] have provided many works in this area. They show that by having graph $G(V, E)$, on the same set of vertices V, if we sample each edge e with probability p_e, we will get a graph in which minimum cut value is $(1+\epsilon)$-times its value in the original graph.

The works that we mentioned are cut sparsifiers; however, we have spectral sparsifiers which are stronger version of cut sparsifiers. Spielman et.al. [60] propose a local clustering

approach for partitioning of graphs. In this approach, clustering is used for partitioning of the graph in a way that we find approximate sparsest cut. Therefore, this method can be used as an spectral sparsifier. Using this partitioning approach, Spielman et.al. [64] propose a method that uses the Laplacian matrix of the graphs. In [61] authors show that for each Laplacian matrix A there is a weighted graph. If we have a subgraph that is roughly the same as the spanning tree of the original graph with a Laplacian matrix B, we call this subgraph an ultra-sparsifier and we say that B is a good precondition for A. To sum up, for spectral sparsification, spectral similarity of graph Laplacians are used [64].

There are many other works for both cut and spectral sparsifiers [63][62][23]. The edges can be sampled randomly, based on their connectivity, sampling random spanning trees of the graph, etc. Ahn et.al. [6] propose an approach for streaming setting and where we have linear space for storage. They provide a solution that passes over the data once. In [4] Ahn et.al. construct a sketch-based sparsifier, i.e. via linear projections of graph and they provide $(1+\epsilon)$-approximation for all cut values of the graph. In [5], construction of spectral classifiers in a dynamic setting where edges can be added or removed are analyzed.

In summary, although graph sparsification is a related area to our research, it has its own differences. In sparsification we try to make a large graph smaller in terms of edges while approximating specific properties of the graph with high accuracy. While our work tries to build a graph from data sources like matrices, relational tables, etc. in an efficient way. The graph that we output then can be fed to a sparsifier if further size reduction is needed.

2.5.4 Spanners

An $\alpha - spanner$ of a graph G is a spanning tree in which the distance between every pair of vertices is at most α times their distance in G [15]. The goal is to find spanners with small size and low (α) parameter [4] in order to make distance approximations as accurate as it can be. Therefore, spanners have a focus on node distances and are used when distances are the concern.

Chapter 3

Proposed Methodology

In this section, we provide the symbols used throughout the thesis and their definitions as well as the problem statement. Then, we present our proposed methods and elaborate them in details.

3.1 Preliminaries & Definitions

In table 3.1, a list of symbols used frequently in the thesis is provided. The dataset we use is a matrix with rows representing entities and columns attributes. We denote the set of entities in the dataset as R, the set of attributes as A, and the set of attribute values as V. The dimensionality of the dataset is the size of the attribute set A, which is denoted as d. If a data matrix is sparse, it has many zero entries. We can represent such a matrix by its *sparse vector representation*, in which a row r is represented by a set of attribute-value pairs whose value is not zero, that is, $r = \{(a, v) \mid a \in A \ \& \ v \in V \ \& \ v \neq 0)\}$. Figure 3.1 illustrates an example data matrix and its sparse vector representation.

	A1	A2	A3	A4	A5
R1	5	1	0	0	4
R2	0	0	3	0	0
R3	0	0	2	0	1
R4	3	2	0	0	4
R5	0	3	0	0	2

(a) A data matrix

R1 = {(A1, 5), (A2, 1), (A5, 4)}
R2 = {(A3, 3)}
R3 = {(A3, 2), (A5, 1)}
R4 = {(A1, 3), (A2, 2), (A5, 4)}
R5 = {(A2, 3), (A5, 2)}

(b) Sparse vector representations of the rows of the matrix

Figure 3.1: Different representations of a data matrix

Table 3.1: Table of Symbols

Symbol	Definition
D	Data matrix with rows representing entities and columns attributes. Its set of entities is denoted as R, the set of attributes as A, and the set of attribute values as V
n	number of rows of the matrix
d	number of columns of the matrix
r	$\{(a, v) \mid a \in A \ \& \ v \in V \ \& \ v \neq 0\}$
ϵ	similarity threshold
k	sketch size

3.2 Problem Definition

Suppose we are given a data matrix with rows representing entity set R and columns the attribute set A, or the sparse vector representation of the data matrix. Our goal is to build a graph whose nodes represent the entities in R and edges represent the similarities between two nodes based on the attributes and their values in the matrix, in a fast and efficient way. To achieve this goal, we aim to develop efficient algorithms for building an approximate similarity graph that contains only the edges whose similarity value is above a similarity threshold.

Definition 1 (Approximate Similarity Graph Construction) *Given a similarity threshold ϵ and a data matrix whose rows represent entities and columns represent attributes or the sparse vector representation of the data matrix, the problem is to build a similarity graph $G(V, E)$ where V is the set of entities in the data matrix and E is the set of edges where each edge represents the similarity between two nodes and the similarity is above the ϵ threshold.*

3.3 Proposed Work

Building similarity graphs is an expensive operation due to the number of pairwise similarity computations. Furthermore, the dimensionality of data (i.e., the number of attributes) affects the speed of similarity computations significantly. The higher dimensionality is, the slower the process would be. The complexity of such a task is $O(n^2d)$ where n is the number of instances in the dataset and d is the dimensionality. In this work, we propose an efficient algorithm for the construction of a similarity graph given a data matrix or its sparse vector representation.

We use the Conditional Random Sampling (CRS)[41] technique combined with the use of an inverted index for pairwise computations. The first technique is used for tackling the problem of dimensionality while the latter is used for pruning all unnecessary pairwise computations. The combination of these two techniques results in a considerably fast method.

3.4 Overview of the Algorithm

Our method starts with a data matrix (or its equivalent sparse vector representation) and returns a similarity graph. It has three major steps as shown in Figure 3.2. The first step of the algorithm is data sketching in which high-dimensional data matrix is converted to small-sized sketches. The second step is computing pairwise similarities. This step is done by using an inverted index. The last step is the graph construction. The resulted similarity graph can be used in many different graph downstream tasks such as nearest neighbour search (NNS), link prediction, collaborative filtering and maximum inner product Search (MIPS) to name a few.

Algorithm 1 shows the overview of the proposed method. The inputs to the algorithm include the dataset D. This dataset can be a data matrix with rows being instances and columns being their attributes or the sparse vector representation of the matrix. Other inputs are the dataset dimensionality which is the number of attributes, a similarity threshold and the size of the sketch. The output is the similarity graph built based on sketches. In the first step, the dataset is sketched using the CRS technique [41]. For sketching, we

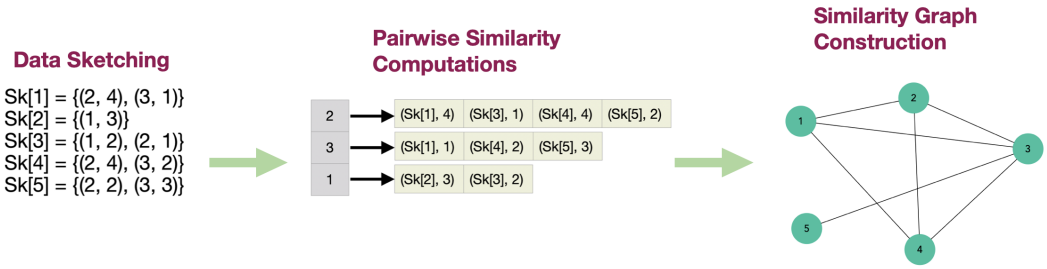


Figure 3.2: Overview of our method. The first step of the algorithm is data sketching in which high-dimensional data matrix is converted to small-sized sketches. The second one is computing pairwise similarities which is done by using an inverted index and the last step is the graph construction.

need the original dataset D as well as size of sketch k. The output of this step includes the sketches of the dataset (represented by Sk in Algorithm 1) and the maximum column IDs of the sketches (denoted as Sk_Max_Id). We will describe the sketches and their maximum column IDs in the next section. The second step is the pairwise similarity computation. This step takes the outputs of sketching, d and the ϵ threshold as inputs. The output is the similarity values between all pairs of sketches which are above the ϵ threshold. In the last stage, based on the computed similarity values, the similarity graph is built and returned as the final output of the algorithm.

Algorithm 1: Approximate Similarity Graph Construction

Input: Dataset: **D**, dataset dimensionality: d, similarity threshold: ϵ, size of sketch: k

Output: similarity-graph: Sk_G

1 $Sk,\ Sk\_Max\_Id \leftarrow$ **Sketch**(D, k)
2 $Sk\_sim \leftarrow$ **PairwiseSimilarities**($Sk,\ Sk\_Max\_Id,\ \epsilon,\ d$)
3 $Sk\_G \leftarrow$ **Graph**(Sk_sim)
4 **return** Sk_G

3.5 Data Sketching

We use conditional random sampling (CRS) [41] to reduce dimensionality of the data. CRS is a data sketching technique (described in Chapter 2.4.1) suitable for high-dimensional

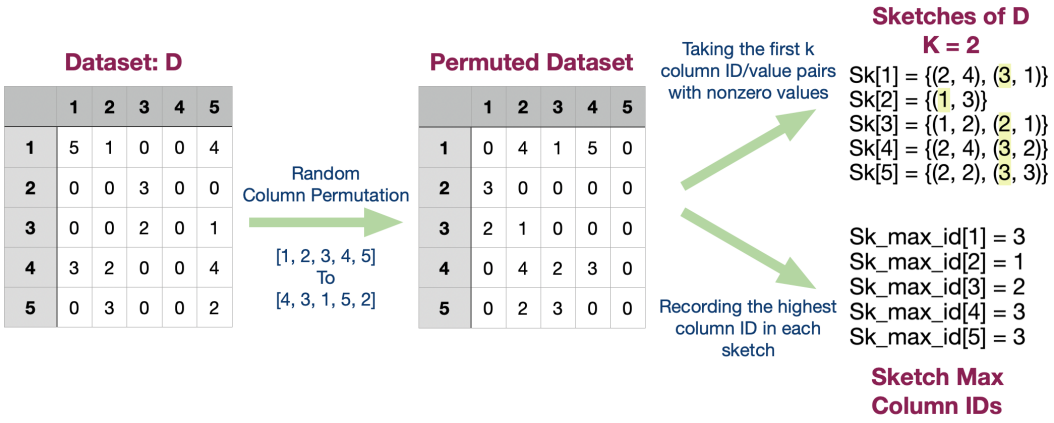


Figure 3.3: Data sketching overview on a data matrix. Given a data matrix, the columns of the matrix are first randomly permuted. A randomly permuted dataset is shown in the middle of the figure, where the random permutation of the list of columns $[1, 2, 3, 4, 5]$ is $[4, 3, 1, 5, 2]$. After the permutation, the sketch of each row is generated by taking the first k column ID-value pairs with non-zero values in the permuted column order. During the computation of the sketches, for each row or sketch we also obtain the highest column ID of the column ID-value pairs in the sketch and store it in a one dimensional array Sk_max_id which will be used for efficient computing of the pairwise similarities between rows/entities. In the figure, the highest column ID in each sketch is highlighted with yellow color.

sparse datasets where zero entries are insignificant. It has two stages: one creating sketches of data in order to make them smaller and the other using sketches for the estimation of pairwise similarities. We adopt its first stage to create data sketches.

3.5.1 Data Sketching from Data Matrix

Figure 3.3 illustrates the steps of the data sketching procedure if the input data is a data matrix. Given a data matrix, the columns of the matrix are first randomly permuted. The purpose of the permutation is to eliminate the dependencies or special orderings on the columns the dataset might have. A randomly permuted dataset is shown in the middle of the figure. The permutation of the columns $[1, 2, 3, 4, 5]$ is $[4, 3, 1, 5, 2]$ meaning that the contents of the first column are put in the 4th column, the contents of the 2nd column are put in the 3rd column, the contents of the 3rd column are put in the first one, etc. Note that we do not need to actually change the data matrix. The permutation can be done on the set of column IDs.

After the permutation, the sketch of each row is generated by taking the first k column ID-value pairs with non-zero values in the permuted column order. If a row has less than k non-zero values, the length of its sketch (that is, the number of column ID-value pairs) is less than k. Thus, the sketches of different rows may have different lengths. The value for the sketch size k is typically set as the average number of attribute-value pairs with non-zero values among the rows as suggested in [41].

During the computation of the sketches, for each row or sketch we also obtain the highest column ID of column ID-value pairs in the sketch and store it in a one dimensional array Sk_max_id. For example, in Figure 3.3, for row 1 we have $Sk[1] = \{(2, 4), (3, 1)\}$ which contains the first two column ID-value pairs with nonzero values after permutation of columns for the first row. Thus, the highest column ID of the column ID-value pairs in the sketch of first row is 3. That is, $Sk\_max\_id[1] = 3$. This Sk_max_id array will be used for efficient computing of the pairwise similarities between rows/entities. Note that the CRS data sketching algorithm does not produce this array. We extend CRS in this regard for the purpose of speeding up similarity computation, to be described in the next section.

3.5.2 Data Sketching from Sparse Vector Representation of Data

There are settings in which we are not given a matrix, instead a sparse vector representation of a large and sparse matrix is given. For example, if the data set is a set of text documents, each document can be represented as a set of words where a word is considered as an attibute and the value for the attribute can be the number of times the word occurs in the document. Using this representation, the many entries of the matrix carrying zero values are omitted and there would be a significant decrease in the amount of data stored. In this case, our input would be *sparse vector representation* of the matrix as described in Figure 3.1.

Data sketching can be done on such a matrix representation as pictured in Figure 3.4. First, the columns of the matrix are permuted. In Figure 3.4, the random permutation of the list of columns $[1, 2, 3, 4, 5]$ is $[4, 3, 1, 5, 2]$ meaning that the contents of the first column are put in the 4th column, the contents of the 2nd column are put in the 3rd column, etc. In the figure, for the first row in the original dataset we have $R_1 = \{(1, 5), (2, 1), (5, 4)\}$.

Because the contents of the 1st column are permuted and put in the 4th one, $(1, 5)$ changes to $(4, 5)$. For $(2, 1)$, based on the permutation of 2nd column to 3rd one, we have $(3, 1)$ and the same happens for $(5, 4)$ and it is mapped to $(2, 4)$. So the first row in the permuted dataset is $R_1 = \{(4, 5), (3, 1), (2, 4)\}$. After permutation, we have to take the first k ID-value pairs of each row. The order of the pairs are based on their new column IDs in an ascending order. So we sort pairs of each row based on their new column IDs in an ascending order and take the first k entries of them. These selected pairs form the sketches of the rows. In addition to the sketches, we have to get the maximum column ID in each sketch which is simply the column ID of the last entry of the sketch shown in the figure with the yellow color. This approach is another implementation of the CRS and can be applied in situations where we are given a sparse vector representation of a matrix.

3.5.3 Choosing The Sketch Size: k

As mentioned in the previous subsections, the value of the sketch size k is typically set to be the average number of attribute-value pairs with non-zero values among the rows (as suggested in [41]). However, getting the average number of nonzero entries among the rows of the dataset requires a full scan of the dataset, which is computationally inefficient. Instead, one approach is to resort to sampling. We sample rows of the dataset and compute the number of non-zero values on the sampled rows. Based on the central limit theorem[1], it is known that the sampling distribution of the sample means approaches a normal distribution as the sample size gets larger — no matter what the shape of the population distribution is. Therefore, for a reasonable number of samples (typically this fact holds for sample sizes over 30), we can set k to be equal to the mean of the sample distribution.

3.6 Pairwise similarity computations

We use an inverted index to help with the similarity computation between each pair of sketches obtained from the previous step of the algorithm. The inverted index is built upon

[1] https://en.wikipedia.org/wiki/Central_limit_theorem

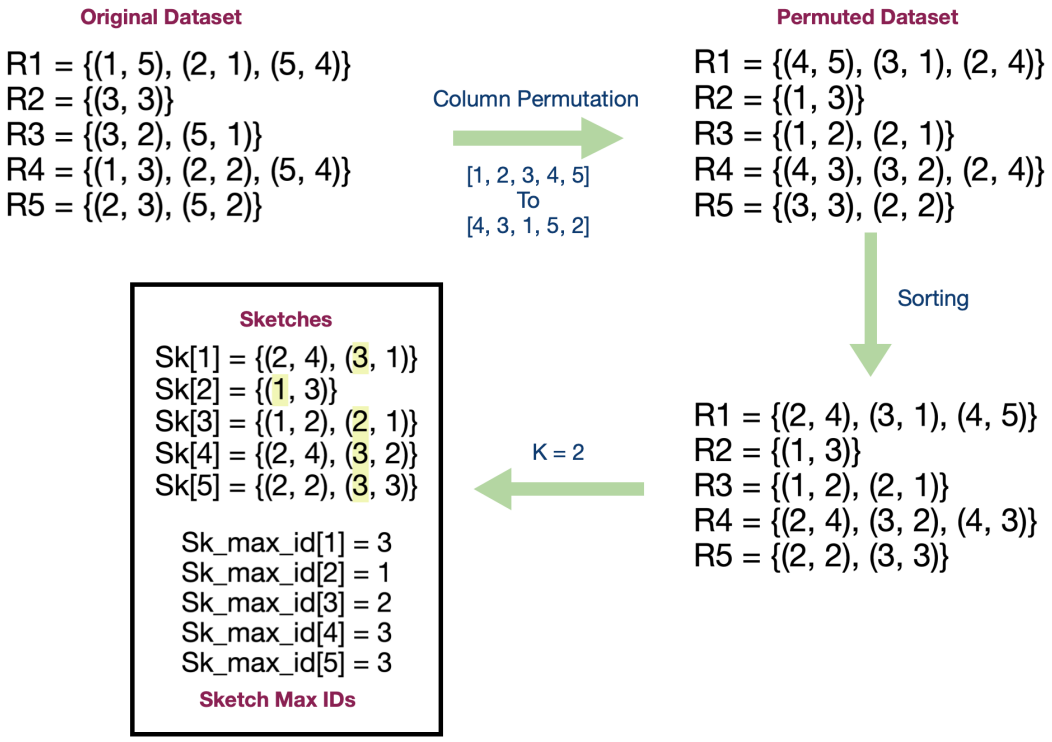


Figure 3.4: Data sketching overview on the sparse vector representation of a matrix. Given a sparse vector representation of a matrix (shown as *Original Dataset*), the steps to form the sketches are shown. First, columns of the matrix are permuted. In this figure the random permutation of the list of columns $[1, 2, 3, 4, 5]$ is $[4, 3, 1, 5, 2]$. Then, each row is sorted based on the column IDs of pairs in ascending order. Sketches are the first k pairs of each row. In this example, $k = 2$. In addition to sketches, we have to take sketch max ids which are the largest column ID in each row of the sketches.

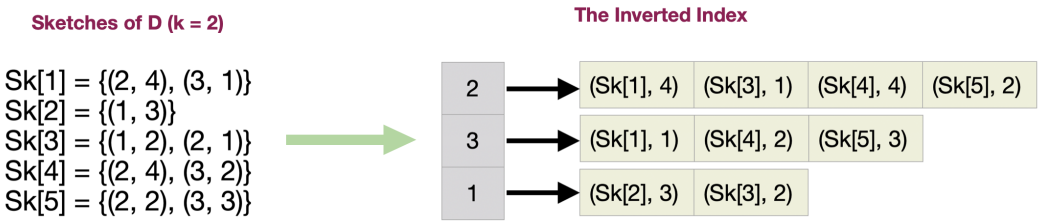


Figure 3.5: In this figure, the inverted index of the sketches of dataset D is presented. For each column ID in the sketches, we have an entry in the index which shows the ID itself and a list of all sketches that have this ID as well as their values.

the sketches of the input data. It uses attributes as keys and maps an attribute to the list of sketches that contain the attribute. Figure 3.5 shows the inverted index built upon the sketches of D in Figure 3.3. For each column ID in the sketches, we have an entry in the index which shows the ID itself and a list of all sketches that have this ID as well as their values. This index is built gradually in our method and sketches are added to it one-by-one as we iterate over them.

For the similarity computations, the similarity measure we work with is the inner-product. However, this work is not limited to just this measure and any other measures using the inner-product (or dot-product) can be used in the following algorithms with a little modification. For example, the cosine similarity measure, Jaccard, overlap and dice are a few examples of such measures that can make use of the dot product value of the two vectors they operate on. With some little modifications, these measures can also be used in our algorithms. For this kind of measures, using the inverted index makes the process much more efficient as it eliminates all redundant and unnecessary comparisons between instances of the dataset. Based on the inner-product, if two vectors do not share any dimensions, their inner product value would be zero. Besides, different instances usually share a small proportion of dimensions so there is no need to have comparisons between them for all dimensions. The inverted index helps to reach this goal.

The main structure of pairwise similarity computations by using an inverted index is shown in Figure 3.6. This structure is the simplified version of our algorithms and some minor parts are not shown here. Now, we elaborate how the method works. From the

previous step of data sketching, we are provided with sketches of data. We iterate over them just once to index each sketch one-by-one and at the same time compute the similarities between the current sketch in the iteration with the previous ones. That is, the inverted index is not pre-built, but it is built at the same time the similarities between sketches are computed.

At first, the inverted index is empty. In Figure 3.6, we assume that we have iterated over the first three sketches and now we are processing the 4^{th} one. The inverted index in this stage contains all the information for the first three of them. It stores attributes and a list of the sketches that have them with their values. So when we want to compute similarity of sketch 4 to the previous ones which are indexed, we just get its attributes, go to their corresponding list in the index and compute its partial similarity to the ones existing in the list. As an example, for sketch 4, we have $Sk[4] = \{(2, 4), (3, 2)\}$. We get the list corresponding to the column ID 2 in the index which would be $[(Sk[1], 4), (Sk[3], 1)]$ as well as the list for column ID 3 which is $[(Sk[1], 1)]$. We compute partial similarity of the sketch 4 to $Sk[1]$ and $Sk[3]$ by multiplying attribute values. We store the similar sketches and their scores in a map having keys as the ID of similar sketches and values as the similarity score.

So at the end as we have iterated over all sketches, all similarities are computed. Just we have to mention that after all the similarity scores are computed and stored, when we want to query pairwise similarity of two sketches, we have to check similarity map of both of the sketches. That's because sketches are indexed one-by-one and the ones that are indexed first usually have smaller similarity maps because a few sketches were indexed at the time they had been processing.

As stated earlier, our method has differences to what is pictured in Figure 3.6. As we are working with sketches instead of original data, the score computations are not the same. In the Related Work section, we explained the estimation stage of CRS. Here we bring the information again to show how the process should be done. We work with inner-product as

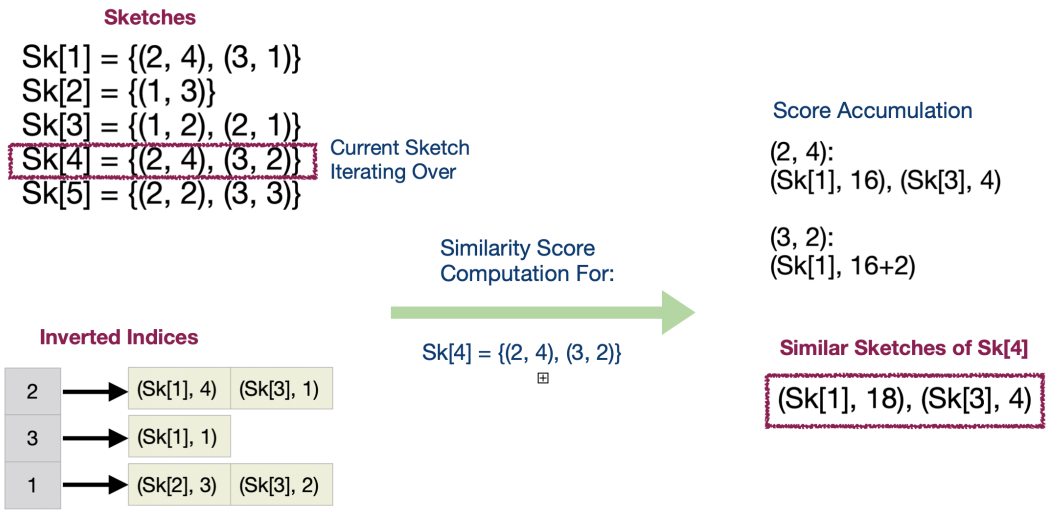


Figure 3.6: Pairwise similarity computations using an inverted index. Sketches are indexed one-by-one. In this figure, we have iterated over the first three sketches and indexed them. Now, we are iterating over the 4^{th} one. In the bottom left corner of the figure the inverted index at this stage is shown. It contains information of the first three sketches. For each sketch we iterate over, first its similarity to the ones that are already indexed is computed and then we index it. For the 4^{th} sketch we have, $Sk[4] = \{(2, 4), (3, 2)\}$ so for each column ID-value pair it has, we get the corresponding list in the index and compute its partial similarity scores. The final similarity scores of each sketch we iterate over would be a map of the sketch IDs which are similar to the current one and their similarity values.

the similarity measure and show it as a. We show its approximation for the sketches as $\hat{a}$.

$$a = \sum_{i=1}^{d} u_{1,i}u_{2,i} \qquad \hat{a} = \frac{d}{d_s}\sum_{i=1}^{d_s} \tilde{u}_{1,i}\tilde{u}_{2,i} \tag{3.1}$$

As it can be seen, for sketches we do not do the computations for all the dimensions. In these equations, d is the dimensionality of the original data (or number of attributes the original dataset has) and u_1 and u_2 are two rows in the original dataset that we want to measure their similarity. $\tilde{u}_1$ and $\tilde{u}_2$ are the sketches obtained by CRS from the mentioned rows. As we know, a sketch is a set of pairs of column (attribute) IDs and their values. So $d_s = min(max(ID(\tilde{u}_1)),\ max(ID(\tilde{u}_2)))$ and it shows what is the min of the maximum column IDs in both sketches. The reason that in the sketching step, we compute Sk_max_id array is for efficient computation of ds values. This d_s value is computed for each two pairs of sketches. The main philosophy is that we do the computations for attributes having ID less than d_s. This would be the same as if we sampled the first d_s columns of the matrix for these two rows. Then, we extrapolate the result for all the dimensions by multiplying $\frac{d}{d_s}$ to similarity values.

Incorporating this computation with an inverted index is not straightforward. Suppose we have a sketch and we want to compute its similarity to the sketches in the inverted index. For every indexed sketch that has a common attribute ID with the one that we are analyzing, we have to compute a d_s value. This computation should be done in an efficient way. We propose Algorithms 2, 3, 4 each show a different approach of addressing this issue.

All our Pairwise Similarities algorithms, start with a given list of sketches, a list of maximum ids for sketches to find d_s, a similarity threshold and also dimensionality of original data d. The goal is to find all pairwise similarities of sketches for which we use a map and show it as S in the following algorithms.

3.6.1 Online Pairwise Similarities

The idea of this algorithm is that we compute d_s values for each pair *on the fly* when we encounter the pair. Algorithm 2 shows the steps. We start with an empty pairwise

similarities map shown as S. This map is gradually filled. Each key in the map would be a sketch and the value also a map of its similar sketches and their score of similarity. Also, an empty inverted index for all the attributes is initiated at first. As explained before, d is the number of attributes so we have d number of lists in the index. In line 3 we have a for loop for iterating over all the sketches. For each sketch, we start with an empty map M which is used for keeping similar sketches and their score of similarity to the current one. Note that for each sketch, we have a map of similar sketches and for all of them we have a map of their maps. In line 5, we iterate over all elements of the sketch and for each attribute it has, we grab its associated list in the index. For example, for (a, v) we have to grab the a^{th} list in the index which we show it as I_a. This list contains all the sketches having that attribute and their values. (y, y_v) shows such a pair. So for each y, we have to find the pairwise d_s value which we do it by getting the max id of x and y from Sk_Max_Id and then get the minimum. Then we have to check if $a \leq d_s$. If so, we are allowed to add y and its score to M. As we are working with sketches instead of original data, the score is computed by equations in 3.1. Next, after we compute similarities for this attribute of the sketch, we add it to the inverted index which is shown in line 10. The next step is to filter scores based on ϵ. Algorithm 5 shows filtering similarities which is done via simply getting the scores map and filtering the ones that are above ϵ. S_x shows similar sketches to x which are above ϵ. In line 12, we add (x, S_x) to the main similarity map S. The output of the algorithm is the S similarity map.

3.6.2 Offline Pairwise Similarities via Sorting

In this algorithm, we want to avoid all unnecessary checks for d_s. In the previous method, we were letting all the sketches to be indexed without any restriction on the order of insertion and max id of the sketch. This led the previous algorithm to have many redundant and repetitive computations which could be easily avoided. If we apply a sorting on the sketches such that the sort gives us some information on what d_s value would be, we can optimize the process significantly.

We know d_s is the minimum of the maximum IDs of the sketches. The main idea of

Algorithm 2: Online Pairwise Similarities

Input: sketches of D: **Sk**, sketch max ids: **Sk_Max_Id**, similarity threshold: ϵ, dimensionality of original data: d
Output: pairwise similarities: $\boldsymbol{S}$

```
1  S ← ∅
2  I_1, I_2, ..., I_d ← ∅ (I_i is the entry for the ith attribute in the inverted index. It will
     contain a list of (x, v)'s, where x is a row/sketch id and v is the value of x for the
     ith attribute.)
3  for x ∈ Sk do
4      M ← ∅ (M holds the similarity values between x and each of the skethes before
         x in Sk)
5      for (a, v) ∈ x do
6          for (y, y_v) ∈ I_a do
7              d_s = min(Sk_Max_Id[x], Sk_Max_Id[y])
8              if a ≤ d_s then
9                  M[y] ← M[y] + (d/d_s) · v · y_v
10         I_a ← I_a ∪ {(x, v)}
11     S_x ← Filter_Similarities(M, ε) (Remove similarities in M whose value is less
         than ε)
12     S ← S ∪ (x, S_x)
13 return S
```

this algorithm is that we sort sketches based on their max ids in such a way that we know whatever sketch which has been indexed, has a smaller max id than any upcoming sketch that is not yet indexed. For this purpose, we should sort sketches based on their max ids in an ASCENDING order. This means that sketches indexed define *min* so that we know d_s would be max id of the indexed sketches. This way we prune many unnecessary further checks on d_s which greatly affects the preformance. This is the beauty of applying this sort on the sketches. However, note that we do not sort sketches themselves and we just sort their indices. In line 3, we are producing indices of the sketches that we should iterate over in order via *arg_sort*. Lines 5 and 6 show how we get the id of sketch we should analyze and the sketch itself. In line 10, for d_s computation we just get the max id of the indexed sketch without any further checks as we know indexed sketches always have a smaller max id than the ones that are not indexed. The rest of the steps of the method are the same as the previous algorithm.

Algorithm 3: Offline Pairwise Similarities via Sorting

Input: sketches of D: **Sk**, sketch max ids: **Sk_Max_Id**, similarity threshold: ϵ, dimensionality of original data: d
Output: pairwise similarities: $\boldsymbol{S}$

```
1  S ← ∅
2  I_1, I_2, ..., I_d ← ∅
3  Sorted_Sk_Indices ← arg_sort(Sk_Max_Id, ascending)
4  for i ∈ range(|Sk|) do
5      x_id = Sorted_Sk_Indices[i]
6      x ← Sk[x_id]
7      M ← ∅
8      for (a, v) ∈ x do
9          for (y, y_v) ∈ I_a do
10             d_s = Sk_Max_Id[y]
11             M[y] ← M[y] + (d/ds) · v · y_v
12         I_a ← I_a ∪ {(x, v)}
13     S_x ← Filter_Similarities(M, ϵ)
14     S ← S ∪ (x, S_x)
15 return S
```

3.6.3 Offline Pairwise Similarities via Matrix Precomputation

The main idea of this algorithm is that we first compute all the pairwise d_s values and whenever we want one of them, we simply get it from d_s pairwise matrix. The main structure of this algorithm is the same as the previous ones. In lines 3-6, we are building the pairwise d_s matrix. In line 11, we just get d_s value from the d_s matrix and do the rest of the steps as prior methods.

3.6.4 Comparison of Pairwise Similarity Algorithms

Time Complexity

Now, we investigate the time complexity of the algorithms. For the sketching stage, we provided two methods. One starts from a data matrix and the other starts from sparse vector representation of the matrix. For the first one which starts from the matrix the time complexity is as follow: a random permutation mapping of columns of the matrix is generated. It has length d which is dimensionality of data and hence, the complexity would be $O(d)$. Then, on the permuted matrix, for each row we have to iterate columns till we

Algorithm 4: Offline Pairwise Similarities via Matrix Precomputations

Input: sketches of D: **Sk**, sketch max ids: **Sk_Max_Id**, similarity threshold: ϵ, dimensionality of original data: d
Output: pairwise similarities: $\boldsymbol{S}$

$S \leftarrow \emptyset$
$I_1, I_2, ..., I_d \leftarrow \emptyset$
$n \leftarrow |Sk|$
for $i \in range(n)$ **do**
  for $j \in range(n)$ **do**
    $d_s\_pairwise[i][j] \leftarrow min(Sk\_Max\_Id[i], Sk\_Max\_Id[j])$
for $x \in Sk$ **do**
  $M \leftarrow \emptyset$
  for $(a, v) \in x$ **do**
    for $(y, y_v) \in I_a$ **do**
      $d_s = d_s\_pairwise[x][y]$
      if $a < d_s$ **then**
        $M[y] \leftarrow M[y] + (d/ds) \cdot v \cdot y_v$
    $I_a \leftarrow I_a \cup \{(x, v)\}$
  $S_x \leftarrow$ **Filter_Similarities**(M, ϵ)
  $S \leftarrow S \cup (x, S_x)$
return $\boldsymbol{S}$

Algorithm 5: Filter Similarities Based on The Threshold

Input: similarity score map: **M**, similarity threshold: ϵ
Output: filtered similarities map: R

$R \leftarrow \emptyset$
for $(y, sim) \in M$ **do**
  if $sim \geq \epsilon$ **then**
    $R \leftarrow R \cup \{(y, sim)\}$
return R

reach the k^{th} nonzero entry in the row and k is the sketch size. If we assume the average position of the k^{th} nonzero entry for all rows is m, the complexity of such a task would be $O(nm)$ with $m < d$. The second sketching method we elaborated starts from a sparse vector representation of a large and sparse matrix. For this dataset representation also we do a permutation of columns which results in $O(d)$ complexity. Then, on each row of the permuted dataset we do the following steps: first we change column IDs based on the permutation, then we sort the pairs based on the column IDs and at last, we take the first k entries. So the complexity would be $O(n \times (f + f.log(f) + k))$ where f is the average number of nonzero entries for all the rows of the dataset. Usually f and k are much smaller than m resulting in $O(n \times (f + f.log(f) + k)) < O(nm)$. This means that working with a sparse vector representation is more efficient than working with a huge and sparse data matrix.

Computing similarities using an inverted index has the complexity of $O(n \times k \times l)$. That's because for each sketch, we have to compute its similarity to other ones. Each sketch has the maximum length of k and for each of its entries, we have to grab the associated list in the inverted index. If we assume the average size of lists in the inverted index is l, time complexity would follow as mentioned. This is much less than $O(n^2 d)$ which is the time complexity of naively computing pairwise similarities.

For d_s computations, we have an online method which computes d_s values on the fly. This method has some redundant computations in the inner-most loop of the algorithm making $O(n \times k \times l)$ being multiplied by a constant factor like c. Because $n \times k \times l$ is very large, as much as we decrease the number of computations in the inner-most loop to decrease the value of c, the runtimes improve significantly. The other two methods we proposed have a preprocessing step to decrease c. One uses sorting which has the time complexity of $O(n\ log(n))$ and the other one, computes all the pairwise d_s values which results in complexity of $O(n^2)$. However, with the decrease these two methods provide on c as well as proper implementation on their preprocessing steps, we will show in the next chapter that the ones having the preprocessing step are faster than the first one which computes d_s values online.

Furthermore, the additional space needed for d_s values is different in the proposed algo-

rithms. The online method has the $O(1)$ space complexity to process d_s values. The sorting method uses linear space i.e. $O(n)$ and the pairwise matrix precomputation method has the $O(n^2)$ space complexity for analyzing d_s values. So in conclusion, based on the restrictions on the space and the speed needed, one can choose to work with any of these algorithms.

3.7 Similarity Graph Construction

This is the last step of the process. So far, we have computed all pairwise similarities and based on them, we build the similarity graph. The graph is built such that entities of the dataset are nodes and each edge shows the similarity value of two entities of the dataset. Since we only keep the edges whose similarity values are greater than ϵ, the graph is an ϵ similarity graph, This similarity graph can be used in many different applications and graph analysis tasks that use similarities between entities. In the Evaluation section, we show some of these applications.

Chapter 4

Evaluation

In this chapter, we provide evaluations on the proposed methods and their competitors. These evaluations are designed to answer the following questions:

- **Q1. Speed:** How do the proposed approximation methods help the speedup of pairwise similarity computations and thus graph construction?

- **Q2. Accuracy:** What is the effect on the accuracy of similarities? How much error is produced? Is it negligible so that we can safely use approximations?

- **Q3. Effectiveness:** How do the approximations on similarities propagate on different downstream graph analysis tasks? How accurate are the results of graph applications?

4.1 Experimental Settings

4.1.1 Machines

Machine used in these experiments is an Intel Xeon E5-2620v3 2.4 GHz having 6 Cores and 64 GB memory.

Table 4.1: Dataset Descriptions

Name	Distribution	Density	Size
Normal2	Normal(μ=200, σ=50)	2%	10k × 10k
Normal4	Normal(μ=400, σ=100)	4%	
Normal6	Normal(μ=600, σ=100)	6%	
Normal8	Normal(μ=800, σ=100)	8%	
Binomial2	Binomial(n=10k, p=0.02)	2%	10k × 10k
Binomial4	Binomial(n=10k, p=0.04)	4%	
Binomial6	Binomial(n=10k, p=0.06)	6%	
Binomial8	Binomial(n=10k, p=0.08)	8%	

4.1.2 Datasets

We provide results for a number of different datasets. The parameters that change in datasets are the distribution and density. We use two different distributions to analyze behaviour of sketching and graph analysis tasks: Normal and Binomial. These two distributions are among the most common patterns of behaviour in different interaction settings, from network connections to rating datasets, etc. Another parameter is the density which shows the average number of nonzero entries in rows of the dataset. We change density to measure accuracy and runtimes and see if the results are dependent on it.

A summary of dataset descriptions is shown in Table 4.1. In the table, size is defined as number of rows × number of columns. As an example, for synthesizing Normal2 dataset, for each row we sample from the Normal distribution having mean at 200 and standard deviation at 50 and it gives us the number of nonzero entries (density) of each row. This dataset has 10k number of rows and 10k number of columns. Also, each row in Binomial2 dataset is generated by a Binomial distribution having $n = 10k$ (number of columns) and $p = 0.02$ based on the density which is 2%.

4.1.3 Methods

We give summary names to each of our methods and use them in the figures to compare results. Online Pairwise Similarities is named *sk_online*. Offline Pairwise Similarities via Sorting has the name *sk_offline_sorted* and the last one which is Offline Pairwise Similarities via Matrix Precomputations is called *sk_offline_matrix*. The methods with which we compare

our methods are *original*, *nrp* and *sk_naive*. *Original* is the conventional vectorized method of computing similarities; for each two rows of the dataset, a vectorized similarity operation is performed. *Nrp* is Normal Random Projection in which we multiply the data matrix $A \in \mathbb{R}^{n \times D}$ with a random matrix $R \in \mathbb{R}^{D \times k}$ to generate a compact representation $B = AR \in \mathbb{R}^{n \times k}$. The random matrix R consists of i.i.d. entries in $N(0, 1)$, hence, it is called *normal random projection* [14]. Therefore, dataset is first projected to lower dimensions and pairwise similarities are computed between projected rows. The *sk_naive* algorithm uses sketches of data for similarity computations. This method is called *naive* becuase opposed to the methods we proposed, this method does not use any indexing techniques and it naively computes similarities for each pair of sketches by comparing all their key, value pairs.

4.2 Q1. Speed

Figure 4.1 shows runtimes for the mentioned methods. The datasets with the same distribution have shown to have the same trends for the runtime and accuracy so here we just represent results for normal4 and binomial4 as representatives for each distribution. Figure 4.1a represents results for normal4 dataset while 4.1b is used for binomial4 dataset. In these figures the x-axis shows the sample sizes reported in percentage and y-axis the runtimes in seconds.

The experiment shown in Figure 4.1 tries to measure how runtime is affected by the size of sketch/sample. Sample size provided in the x-axis shows the sketch size for methods using sketching as well as the sample size for NRP. We want to measure for a fixed distribution and density, how the runtimes change when we increase the size of the sample.

As it can be seen, original has the same runtime for all the sample sizes because it does not use any sampling on the data and simply works with all the attributes. So we just run it once and for all the sample sizes its runtime is the same. For all other methods, the runtime increases when we increase the sample size because there is more information to process. The original method is the slowest one with a huge difference. The reason why the

original method does not perform very well is that it uses all the attributes of the dataset to compute similarities. In sketching techniques, we don't use all the attributes. Instead, we use a proportion of them that contains information (we just work with a part of nonzero entries in each row).

Sk_naive has larger runtime values than other sketching methods due to not using indexing techniques and naively computing pairwise similarities. As it can be observed, on both datasets when we reach 5% sample size, the original method and sk_naive have roughly the same runtime values. We should mention that the original and nrp methods use vectorized operations which are optimized in hardware levels. On the other hand, sketches are sets of key-value pairs and operations on them are not optimized in the hardware level and this makes it an unfair comparison. So they have an unfair advantage over the sketching methods. If we want to be fair and compare methods with the same way of implementation i.e. not using harware optimized or library codes, etc. the runtime of the original method would be **168000** seconds which is roughly **27**$\times$ slower than sk_naive[1]. However, we show the results for library optimized codes for the original method as these codes are publicly available and people usually use these optimized codes. Despite all of these, you can see in the figures that our proposed methods are performing much better than those hardware optimized methods and this is the beauty of this work! Our proposed methods are up to **62**$\times$ faster than the original method.

In sketching methods, when we reach a sample size that includes almost all the nonzero entries, we don't see a significant runtime change by increasing the sample size. For example, in the Normal datasets when the sample size passes $\mu + 2\sigma$, we are carrying almost all the nonzero entries so after that point as we increase the sample size the runtime does not change considerably.

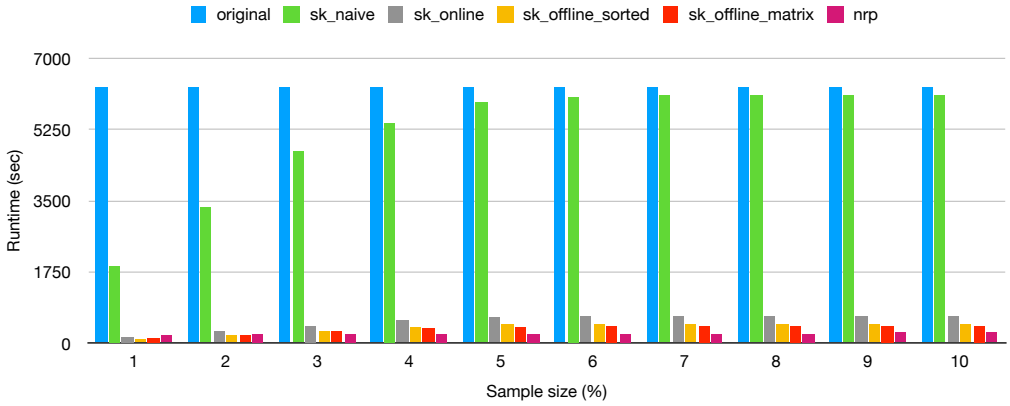
Now, let's analyze our 3 proposed methods. All our methods are among the fastest algorithms as it is shown in the figure. Both of our sketching methods with offline strategies (preprocessing) are faster than the online method with a small gap. In sk_offline_matrix

[1]This runtime is extrapolated. We ran the original method on 1500 number of rows and extrapolated its runtime for 10k rows.

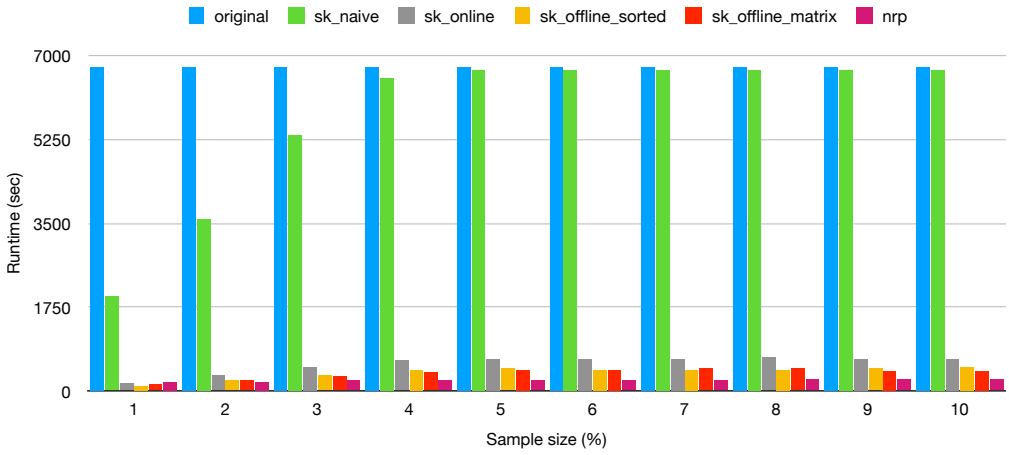
we preprocess data and compute all pairwise d_s values before the similarity computations begin. Then, during the algorithm whenever a d_s value is needed, we get the value from the pre-computed matrix. Sk_offline_sorted has a different preprocessing strategy in a way that we sort sketches based on their max ids so that whenever we need a d_s value for a pair, we already know what it is and there won't be any computations. On the other hand, the sk_online method does not have any preprocessing and computes d_s values whenever we encounter a pair. This leads to redundant computations because we may encounter a pair several times and for each time we have to compute d_s.

In time complexity section, we explained how reducing number of operations in the inner-most loop of our algorithms affects performance and runtimes. We elaborated that the sk_offline_matrix and sk_offline_sorted reduce number of these operations which has a great impact on their performance. Another important factor affecting performance of these methods is their proper implementation in Python programming language. In this language, using *list comprehensions* is much more efficient than using the conventional nested for loops. List comprehensions are represented as: [expression for element in iterable if condition] and they are another syntax for the for loops. However, their execution is much more efficient than a normal loop. As it is shown, the expression is inside the list. This means that for each element in the list, we do not need to load the append attribute of the list and call it as a function at each iteration. This makes this implementation very fast. By using these optimized implementations as well as the reduce in the number of operations in the inner-most loop of the algorithms, as it can be seen in the Figure 4.1, these two offline methods are faster than sk_online. However, sk_online has the advantage of not using any extra memory and in cases where there is a limitation on memory, we can use this method.

Among all the methods compared, NRP is the fastest, followed by our proposed methods with a very small difference. However, in the next sections, we show that there is a huge difference between NRP and our proposed methods in accuracy.

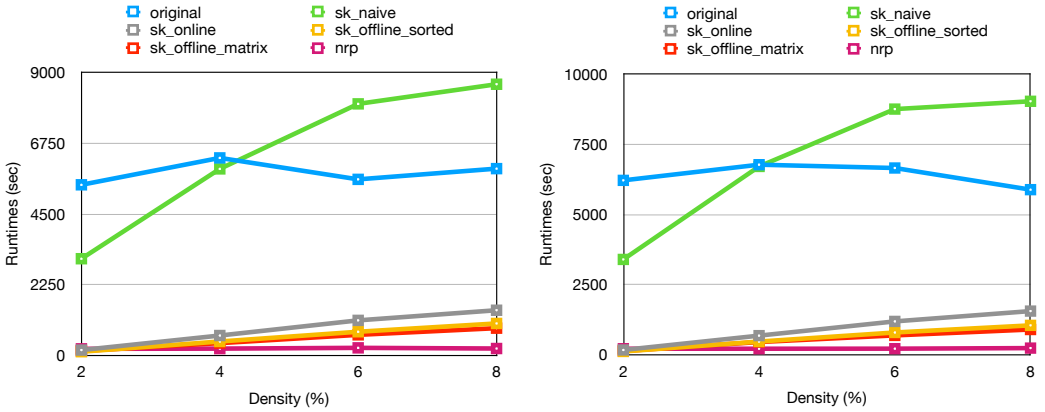


(a) normal4 dataset



(b) binomial4 dataset

Figure 4.1: Runtime results for Normal4 and Binomial4 datasets. X-axis shows the size of samples in percentage and y-axis shows the runtimes. As it can be seen, runtime trends for both datasets are the same.



(a) normal distribution - sample size 5% (b) binomial distribution - sample size 5%

Figure 4.2: Runtime results for Normal and Binomial distributions with different density percentages. Sample size is fixed for 5%.

4.2.1 Effect of density on runtimes

Now, we analyze what effect density has on runtimes. We want to investigate for fixed distributions and a sample/sketch size, how the runtimes change for different densities. This experiment is done to see how sketching methods are affected if the average number of nonzero entries in the rows changes.

Figure 4.2 shows this analysis. In this figure, we provide runtime results for Normal and Binomial distributions with different density percentages. Sample size is fixed for 5%. As much as we increase density, runtimes for sketching methods increase. For sketching techniques if a row has density (number of nonzero entries) less than size of sample, length of its sketch would be number of its nonzero entries ($\leq$ sample size). So although sample size is equal for all these datasets, by increasing density runtimes increase. However, original does not show a trend of change because its runtime just depends on the number of attributes which is always 10k. NRP also does not show changes in runtimes because for all densities it has the same length for the projected rows and it is the sample size.

4.3 Q2. Accuracy

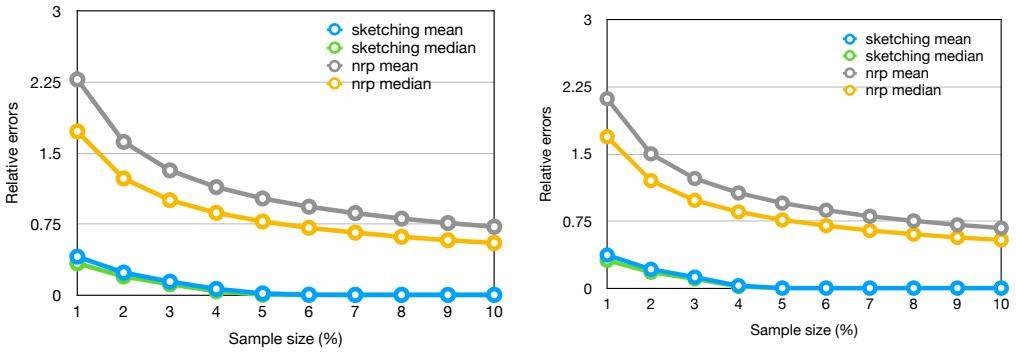
Now, we analyze the accuracy of the mentioned methods on the results of pairwise similarities. The original method gives the exact results. So for all other methods we show the relative errors w.r.t. the result of the original method. In Figure 4.3, result of this experiment is illustrated. As all the sketching methods have the same approximation, we just provide results for one of them (because sk_offline_matrix is the fastest, we use this method). Furthermore, we show the accuracy of NRP as the competitor of our methods and compare the results.

For each two instances of the dataset we have a similarity value, meaning that we have $\frac{n(n-1)}{2}$ number of similarities, where n is the number of rows or instances of the dataset. For measuring the accuracy for each of them, we use the formula: $e_{ij} = \frac{|s_{ij}^{sample} - s_{ij}^{original}|}{s_{ij}^{original}}$ where e_{ij} shows relative error for the pair of instances i and j. s_{ij} is the similarity of those instances and it can be computed from the sampling/sketching techniques or the original method. This formula shows the difference of the similarity values between the original and sample/sketch divided by the original similarity value. We report mean and median of such errors for all pairs of similairites.

We show results for representatives of each of distributions as trends are similar within each distribution. For both Normal4 and Binomial4 datasets, trends are very similar. As we increase the size of sample, the error decreases. For sketching, by increasing the sample size because we can maintain more information (nonzero entries) for each instance, the errors becomes lower and lower and they reach to zero. Zero error occurs when we roughly have all the nonzero entries of the rows. For NRP, we always have errors in the projected dimensions and error never reaches zero. In summary, the error of sketching is considerably lower than the error of NRP for all sample sizes and sketching is a more reliable sampling technique.

4.3.1 Effect of density on accuracy

Another evaluation we provide is the effect of density on the accuracy. In Figure 4.4, effect of density on the relative errors is represented. We want to see for fixed distributions and

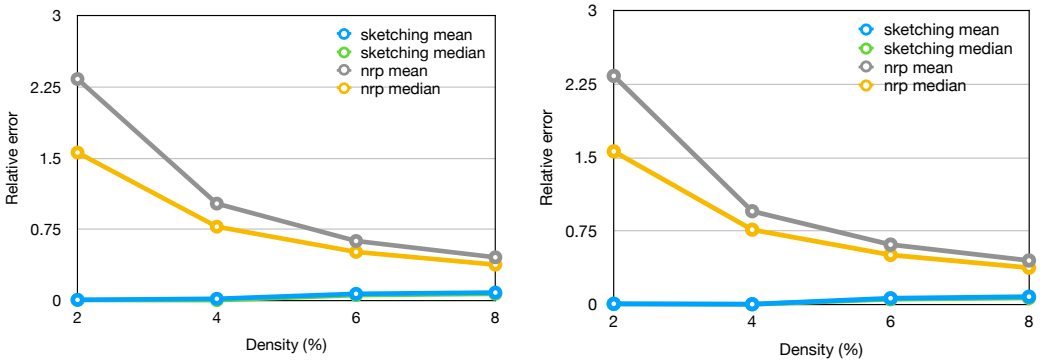


(a) normal4 dataset

(b) binomial4 dataset

Figure 4.3: Relative errors for Normal4 and Binomial4 datasets. X-axis shows the size of samples in percentage and y-axis shows the relative errors. Trends for both datasets are the same and sketching is significantly more accurate than NRP.

a fixed sample/sketch size, how the errors are affected if we increase density of the dataset. Sample size is fixed on 5%. We can see that the error of sketching slightly increases when density is increased. This is due to the fact that the sample size is fixed and when we increase density, there will be more rows with high density for which we are discarding some of the nonzero entries. So the error grows. Although the error increases for sketching, it is still much more accurate than NRP. NRP shows more accuracy for larger density values. This shows that NRP works better on dense datasets.



(a) normal distribution

(b) binomial distribution

Figure 4.4: Relative errors for Normal and Binomial distributions w.r.t different densities for a fixed sample size (5%)

4.4 Q3. Effectiveness

So far we evaluated sampling/sketching techniques and measured how fast and accurate they compute similarities. In this section, we go one step further and evaluate how the errors of sampling approximations propagate in different graph tasks. We build similarity graphs based on the similarities. This step is the same for all the methods so the runtimes would be the same and we don't provide their comparisons. However, accuracy is not the same and we provide its results here.

There are 3 different graph tasks that we consider; however, applications of similarity graphs are not limited just to the following tasks:

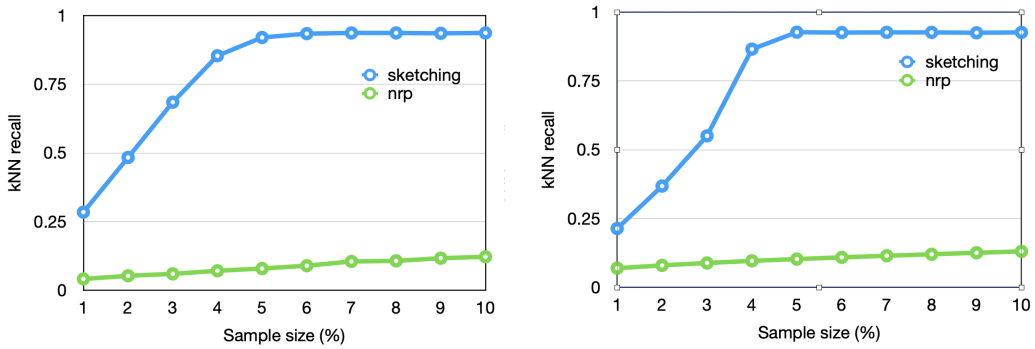
1. K-nearest neighbours
2. Node centrality values
3. Node rankings

These applications show some of the main features of the graphs. K-nearest neighbours show the k neighbours that have the highest similarity to each node. This is a representative of Nearest Neighbour Search (NNS) or k-Nearest Neighbour Search (k-NNS) problems in which using similarity graphs is one of the main approaches to solve the problem. Node centralities show the importance of nodes in the graph. They are widely used in social network analysis for example to find the most influential person in the network, super-spreaders of a disease, etc. Furthermore, node ranking indicates ranking of the nodes based on their importance in the network. We measure the impact of similarity approximations on each of these tasks.

4.4.1 K-nearest neighbours

In Figure 4.5, results for k-nearest neighbours are shown. We have ground truth values from original data and for NRP and sketching we check how many of the k real nearest neighbours of each node we are returning. In other words, we measure kNN recall. For each node we have this value and for all the nodes of the graph, we simply get an average. As it can be seen in the figures, NRP has a very low recall. Sketching shows much higher recall

than NRP. For sketching, as we increase sample size the recall increases. Note that density for these two datasets are 4% and when sample size passes the density, the recall reaches its maximum. In conclusion, for kNN application using sketching is the best as its accuracy is much higher than NRP.



(a) normal4 dataset (b) binomial4 dataset

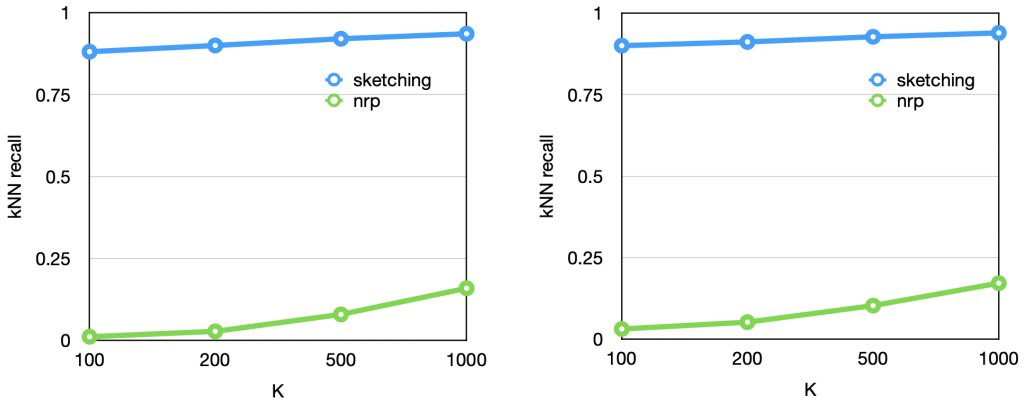
Figure 4.5: kNN recall w.r.t sample size for $k = 500$. The recall for sketching is much higher than NRP for both distributions.

Effect of k in kNN recall

In this experiment, we try to see how the recall changes if we change the k value. Both graphs of Normal4 and Binomial4 datasets have 10k number of nodes (the same as number of rows of the dataset). We change k and analyze the knn recall for each method. Sample percentage is fixed on 5%. Results are shown in Firgure 4.6. This figure shows that as we increase k, recall for NRP increases as it gets more chance to give some of the k nearest neighbours of each node. Despite of the increase NRP shows, its recall value is still very low. For sketching, recall is always around 1 which means independent of number of neighbours (k), accuracy for this method is high.

4.4.2 Node centralities

Node centralities show importance of the nodes in the graph. There are several centrality measures and the difference they have is how they define *importance* of the nodes. Degree,



(a) normal4 dataset (b) binomial4 dataset

Figure 4.6: kNN recall w.r.t k for a fixed sample size (5%). The recall for sketching is much higher than NRP for both distributions.

closeness, betweenness and the eigenvector centralities are a few examples of different kinds of centrality measures. Here we work with eigenvector centrality. This measure assigns relative scores to the nodes of the graph with this philosophy that being connected to high-scoring nodes affects the node's centrality more compared to being connected to the nodes with low scores.

If we have a graph $G(V, E)$, for node $v \in V$ the relative centrality score x_v can be defined as:

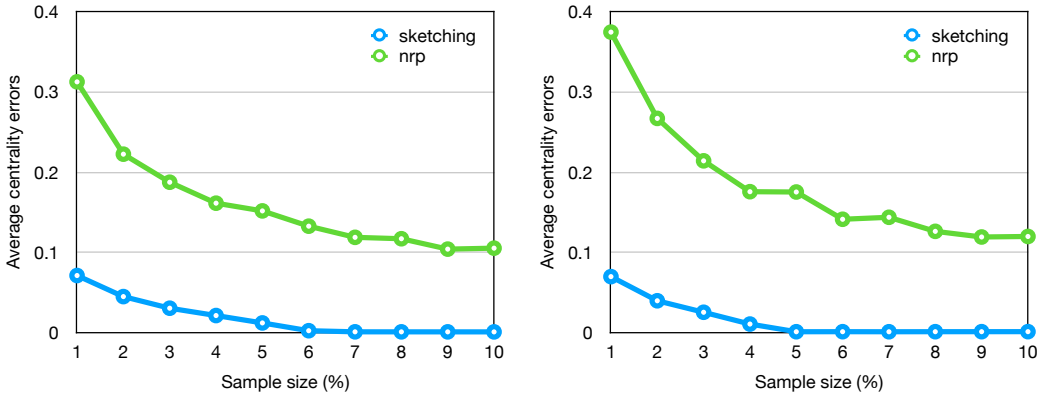
$$x_v = \frac{1}{\lambda} \sum_{t \in M(v)} x_t \tag{4.1}$$

where $M(v)$ is the set of neighbours of node v and λ is a constant. If we denote the adjacency matrix of the graph G by $\mathbf{A}$, this formula can be represented as $\mathbf{Ax} = \lambda\mathbf{x}$ with some small rearrangements.

Here we analyze the average centrality errors. Results are shown in Figure 4.7. In the original graph, each node has a centrality value and in the approximated graphs we have approximations of them. For each node we measure the error as $e_i = \frac{|c_i^{sample} - c_i^{original}|}{c_i^{original}}$ where e_i shows error for node i, c_i is centrality of the node i which can be either in the sample or the original graph. The difference of centralities divided by the original value would be

the relative centrality error for each node. For all the nodes of the graph, we average this error. We report this error for the top 10% most central nodes because there can be many nodes in the graph having very small centrality values (roughly zero) in which the error is not meaningful.

As it is shown in Figure 4.7, the error for sketching is much lower than NRP. Similar to the results of similarity values, as we increase the sample size, the error decreases because more information for each row is carried in the sample. Besides, when sample size passes density of the datasets (4%), the errors for sketching reach to zero. So sketching technique can be a trusted method when we want to build similarity graphs and have centrality analysis on them.

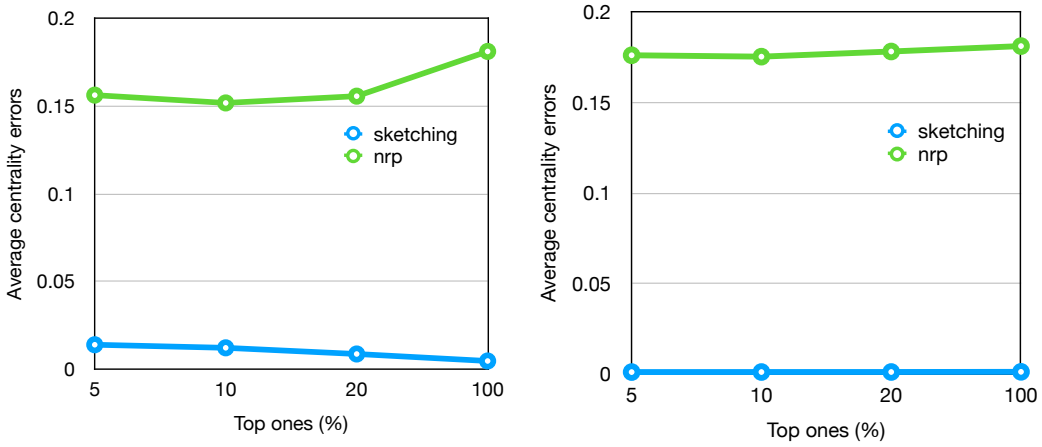


(a) normal4 dataset (b) binomial4 dataset

Figure 4.7: Average centrality errors for normal4 and binomial4 datasets for top 10% most central nodes.

Analysis on node centrality errors for different top percentages

In Figure 4.8 we analyze how the centrality errors change for different top percentages of nodes. For sketching, error decreases when we increase top percentage because there will be more nodes having centrality value the same as original. For NRP it is the opposite and the error increases. As it can be seen, for Binomial4 dataset all the errors of sketching are roughly zero which shows how reliable this method is for the analysis on centralities.



(a) normal4 dataset

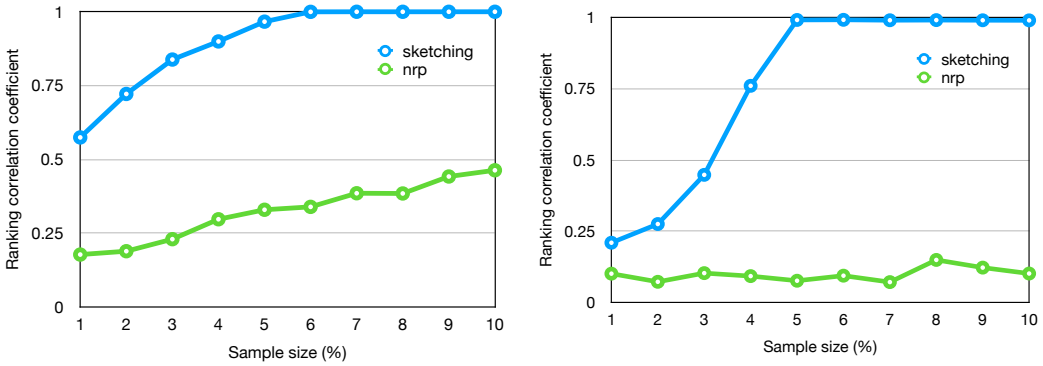
(b) binomial4 dataset

Figure 4.8: Average centrality errors for normal4 and binomial4 datasets for different top percentages of most central nodes. Sample size is fixed on 5%

4.4.3 Node ranking correlations

Based on the centrality of the nodes in the graph, we obtain a ranking for them. We have this node ranking for the original and approximated graphs. We want to measure how well the rankings of the approximated graphs are compared to the original. For this purpose, we use Spearman's ranking correlation coefficient (ρ). This measure evaluates how well the relationship between two variables can be described using a monotonic function. The higher and closer to 1 the ranking correlation is, the better ranking of nodes in original and approximated graphs align.

In Figure 4.9 ranking correlations of sketching and NRP compared to original are reported. We present results for the top 10% most central nodes of the graph and in the next figure we analyze how changing the percentage of top most central nodes has an effect on the ranking correlations. NRP works poorly on this metric, opposed to sketching which gives rankings very close to the original. For this metric as well, when sample size passes density which is 4%, ranking correlations reach to (roughly) 1 for the sketching method. Therefore, for this kind of analysis sketching works much more promising than NRP.



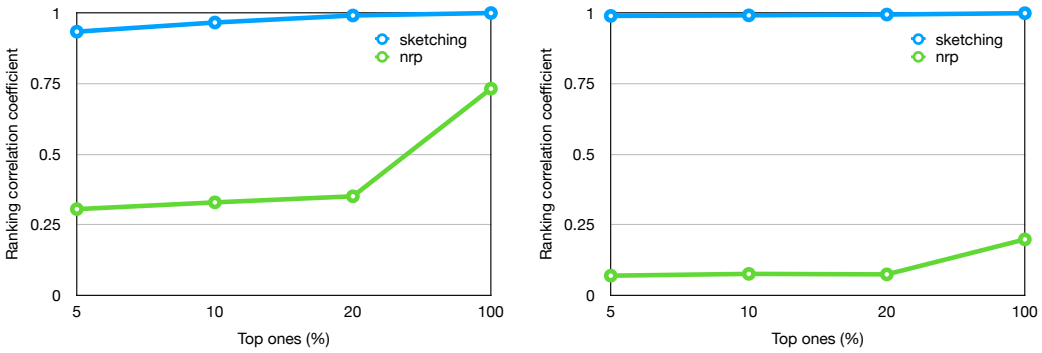
(a) normal4 dataset

(b) binomial4 dataset

Figure 4.9: Centrality ranking correlation coefficients for normal4 and binomial4 datasets for top 10% most central nodes.

Analysis on node ranking correlations for different top percentages

Now, we analyze ranking correlations for different top percentages of most central nodes in the graph. As shown in Figure 4.10, sketching is not affected considerably by the percnetage of the top nodes. This means that it gives constantly promising ranking of nodes based on their centralities. However, for NRP when percentage is increased significantly (from 20% to 100%), ranking correlation becomes higher as there will be more nodes and the chance that they have similar ranking in both graphs increases. In summary, sketching constantly gives very similar ranking of nodes to the original graph while NRP does not perform very well on this metric.



(a) normal4 dataset (b) binomial4 dataset

Figure 4.10: Centrality ranking correlations for normal4 and binomial4 datasets for different top percentages of most central nodes. Sample size is fixed on 5%

Chapter 5

Conclusion & Future Work

5.1 Conclusion

In this work, our goal was to propose an efficient and effective way of construction of similarity graphs. Given a data matrix or its sparse vector representation, similarity graph formulation is an expensive operation because similarities between all pairs of dataset entities need to be computed. Furthermore, dimensionality of data has an impact on speed of the process. Thus, the cost of the naive operation would be $O(n^2d)$ where n is the number of instances and d is the dimensionality of data. We proposed different methods that improve these computations vastly. By using data sketching techniques we reduce the dimensionality of data and the use of an indexing technique helps to eliminate all unnecessary pairwise computations for the instances that do not share a dimension together.

We proposed three different algorithms for speeding up the pairwise similarity comparison. All the algorithms use an inverted index structure, but have different ways to compute the information needed for calculating a pairwise similarity, each of which is suitable in a specific situation. One of our proposed methods uses an online approach for computing different information needed. In practice, this method is slower than the other two methods we propose; however, no extra memory is needed to store intermediate information produced during the algorithm. The other two methods we proposed have an offline approach

and precompute the information that can be precomputed. These two use $O(n)$ or $O(n^2)$ space for storing the intermediate data information but are faster than the previous one. Therefore, there is a tradeoff for space and speed and one can choose any of them based on their constraints in space or the speed needed.

We compared our methods with the conventional way of similarity calculations as well as the commonly used sampling technique of random projection. We showed to be up to **62**$\times$ faster than the original conventional method with a little sacrifice on the accuracy. Also, compared to random projection, we showed that although runtimes have the same range, our results are more accurate with a huge gap.

Beside similarity computations, we formed approximated graphs and analyzed different graph applications on them. We investigated node importance, node rankings and k-nearest neighbours. It's showed that our methods compared to the competitor are considerably more accurate for all the mentioned graph analysis tasks. In all our experiments, we change different parameters and analyze sensitivity of the methods to that parameter. Our methods have shown much less sensitivity than the competitor which makes them more reliable.

To sum up, our methods improve construction of similarity graphs in terms of speed significantly and with a negligible sacrifice on the accuracy. We evaluated them on different graph applications and based on the results, we showed that they are reliable and can be used safely for such analysis. As a result, our work helps improving many different applications like maximum inner product search, node importance, community detection, nearest neighbour search and more.

5.2 Future Work

There are different ideas on how we can extend this work. One idea is that we expand usability of our algorithm by taking a more general input data structure than the matrix. Tensors are multi-dimensional arrays. A matrix is a two-way tensor. The idea is that instead of working with a data matrix, we work with a tensor having multiple attribute dimensions instead of just one. So based on our examples, we had a matrix having entity set R as rows

and the attribute set A as columns. If we work with tensors, we can have multiple attribute sets along A. This helps to generalize this method to many other data representations and adding many other applications to the scope of usability of these algorithms. Another idea we have is to design the distributed version of our algorithms to increase scalability. Using Spark or Hadoop frameworks, we can make the algorithms distributed and capable of working with very large scale datasets. Furthermore, because our baseline sketching method provides theoretical guarantees on the quality of the results, we suggest investigating theoretical bounds for some of the main applications of the similarity graphs.

Bibliography

[1] Dimitris Achlioptas. Database-friendly random projections: Johnson-lindenstrauss with binary coins. *J. Comput. Syst. Sci.*, 66:671–687, 06 2003.

[2] Nesreen Ahmed, Jennifer Neville, and Ramana Kompella. Network sampling via edge-based node selection with graph induction. 01 2011.

[3] Kook Jin Ahn, Sudipto Guha, and Andrew McGregor. Analyzing graph structure via linear measurements. In *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '12, page 459–467, USA, 2012. Society for Industrial and Applied Mathematics.

[4] Kook Jin Ahn, Sudipto Guha, and Andrew McGregor. Graph sketches: Sparsification, spanners, and subgraphs. In *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, PODS '12, page 5–14, New York, NY, USA, 2012. Association for Computing Machinery.

[5] Kook Jin Ahn, Sudipto Guha, and Andrew McGregor. Spectral sparsification in dynamic graph streams. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*, pages 1–10, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[6] KookJin Ahn and Sudipto Guha. Graph sparsification in the semi-streaming model. In *International Colloquium on Automata, Languages and Programming*, pages 328–338, 2009.

[7] Takuya Akiba and Yosuke Yano. Compact and scalable graph neighborhood sketching. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, page 685–694, New York, NY, USA, 2016. Association for Computing Machinery.

[8] Samuel Aparicio, Jarrod Chapman, Elia Stupka, Nik Putnam, Jer-ming Chia, Paramvir Dehal, Alan Christoffels, Sam Rash, Shawn Hoon, Arian Smit, Maarten D. Sollewijn Gelpke, Jared Roach, Tania Oh, Isaac Y. Ho, Marie Wong, Chris Detter, Frans Verhoef, Paul Predki, Alice Tay, Susan Lucas, Paul Richardson, Sarah F. Smith, Melody S. Clark, Yvonne J. K. Edwards, Norman Doggett, Andrey Zharkikh, Sean V. Tavtigian, Dmitry Pruss, Mary Barnstead, Cheryl Evans, Holly Baden, Justin Powell, Gustavo Glusman, Lee Rowen, Leroy Hood, Y. H. Tan, Greg Elgar, Trevor Hawkins, Byrappa Venkatesh, Daniel Rokhsar, and Sydney Brenner. Whole-genome shotgun assembly and analysis of the genome of fugu rubripes. *Science*, 297(5585):1301–1310, 2002.

[9] Martin Aumüller, Erik Bernhardsson, and Alexander Faithfull. Ann-benchmarks: A benchmarking tool for approximate nearest neighbor algorithms. In Christian Beecks, Felix Borutta, Peer Kröger, and Thomas Seidl, editors, *Similarity Search and Applications*, pages 34–49, Cham, 2017. Springer International Publishing.

[10] Bortik Bandyopadhyay, David Fuhry, Aniket Chakrabarti, and Srinivasan Parthasarathy. Topological graph sketching for incremental and scalable analytics. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*, CIKM '16, page 1231–1240, New York, NY, USA, 2016. Association for Computing Machinery.

[11] Roberto J. Bayardo, Yiming Ma, and Ramakrishnan Srikant. Scaling up all pairs similarity search. WWW '07, page 131–140, New York, NY, USA, 2007. Association for Computing Machinery.

[12] András A. Benczúr and David R. Karger. Randomized approximation schemes for cuts and flows in capacitated graphs. *CoRR*, cs.DS/0207078, 2002.

[13] Upasna Bhandari, Kazunari Sugiyama, Anindya Datta, and Rajni Jindal. Serendipitous recommendation for mobile apps using item-item similarity graph. In Rafael E. Banchs, Fabrizio Silvestri, Tie-Yan Liu, Min Zhang, Sheng Gao, and Jun Lang, editors, *Information Retrieval Technology*, pages 440–451, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[14] Ella Bingham and Heikki Mannila. Random projection in dimensionality reduction: Applications to image and text data. KDD '01, page 245–250. Association for Computing Machinery, 2001.

[15] Leizhen Cai and Derek G. Corneil. Tree spanners. *SIAM Journal on Discrete Mathematics*, 8(3):359–387, 1995.

[16] Xiongcai Cai, Michael Bain, Alfred Krzywicki, Wayne Wobcke, Yang Sok Kim, Paul Compton, and Ashesh Mahidadia. Learning collaborative filtering and its application to people to people recommendation in social networks. In *2010 IEEE International Conference on Data Mining*, pages 743–748, 2010.

[17] Graham Cormode. Sketch techniques for approximate query processing. In *Synposes for Approximate Query Processing: Samples, Histograms, Wavelets and Sketches, Foundations and Trends in Databases. NOW publishers*, 2011.

[18] Graham Cormode and S. Muthukrishnan. An improved data stream summary: The count-min sketch and its applications. 55:58–75, 2005.

[19] Sanjoy Dasgupta and Yoav Freund. Random projection trees and low dimensional manifolds. In *Proceedings of the Fortieth Annual ACM Symposium on Theory of Computing*, STOC '08, page 537–546, New York, NY, USA, 2008. Association for Computing Machinery.

[20] Christian Doerr and Norbert Blenn. Metric convergence in social network sampling. page 45–50, New York, NY, USA, 2013. Association for Computing Machinery.

[21] D.L. Donoho. Compressed sensing. *IEEE Transactions on Information Theory*, 52(4):1289–1306, 2006.

[22] Tamer Elsayed, Jimmy Lin, and Douglas W. Oard. Pairwise document similarity in large collections with mapreduce. In *Proceedings of the 46th Annual Meeting of the Association for Computational Linguistics on Human Language Technologies: Short Papers*, HLT-Short '08, page 265–268, USA, 2008. Association for Computational Linguistics.

[23] Wai Shing Fung, Ramesh Hariharan, Nicholas J.A. Harvey, and Debmalya Panigrahi. A general framework for graph sparsification. In *Proceedings of the Forty-Third Annual ACM Symposium on Theory of Computing*, STOC '11, page 71–80, New York, NY, USA, 2011. Association for Computing Machinery.

[24] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Similarity search in high dimensions via hashing. In *Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB '99, page 518–529, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[25] Minas Gjoka, Maciej Kurant, Carter T. Butts, and Athina Markopoulou. Walking in facebook: A case study of unbiased sampling of osns. In *2010 Proceedings IEEE INFOCOM*, pages 1–9, 2010.

[26] Leo A. Goodman. Snowball sampling. *The Annals of Mathematical Statistics*, 32(1):148–170, 1961.

[27] Michael Gowanlock and Ben Karsin. Gpu-accelerated similarity self-join for multi-dimensional data. In *Proceedings of the 15th International Workshop on Data Management on New Hardware*, DaMoN'19, New York, NY, USA, 2019. Association for Computing Machinery.

[28] Mohammad Al Hasan and Mohammed J. Zaki. *A Survey of Link Prediction in Social Networks*, pages 243–275. Springer US, Boston, MA, 2011.

[29] Pili Hu and Wing Cheong Lau. A survey and taxonomy of graph sampling, 2013.

[30] Md Kamrul Islam, Sabeur Aridhi, and Malika Smail-Tabbone. A comparative study of similarity-based and gnn-based link prediction approaches, 2020.

[31] Xiaozheng Jian, Jianqiu Lu, Zexi Yuan, and Ao Li. Fast top-k cosine similarity search through xor-friendly binary quantization on gpus, 2020.

[32] Qing-Yuan Jiang and Wu-Jun Li. Scalable graph hashing with feature transformation. In *Proceedings of the 24th International Conference on Artificial Intelligence*, IJCAI'15, page 2248–2254. AAAI Press, 2015.

[33] Long Jin, Yang Chen, Pan Hui, Cong Ding, Tianyi Wang, Athanasios V. Vasilakos, Beixing Deng, and Xing Li. Albatross sampling: Robust and effective hybrid vertex sampling for social graphs. In *Proceedings of the 3rd ACM International Workshop on MobiArch*, page 11–16. Association for Computing Machinery, 2011.

[34] Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-scale similarity search with gpus. *IEEE Transactions on Big Data*, 7(3):535–547, 2021.

[35] David R. Karger. Random sampling in cut, flow, and network design problems. In *Proceedings of the Twenty-Sixth Annual ACM Symposium on Theory of Computing*, STOC '94, page 648–657, New York, NY, USA, 1994. Association for Computing Machinery.

[36] David R. Karger. Minimum cuts in near-linear time. *J. ACM*, 47(1):46–76, 2000.

[37] David Ron Karger. *Random Sampling in Graph Optimization Problems*. PhD thesis, Stanford, CA, USA, 1995.

[38] Jon Kleinberg. Kleinberg, j. navigation in a small world. nature 406, 845. *Nature*, 406:845, 09 2000.

[39] Jure Leskovec and Christos Faloutsos. Sampling from large graphs. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '06, page 631–636, New York, NY, USA, 2006. Association for Computing Machinery.

[40] Ping Li, Kenneth Church, and Trevor Hastie. One sketch for all: Theory and application of conditional random sampling. In D. Koller, D. Schuurmans, Y. Bengio, and

L. Bottou, editors, *Advances in Neural Information Processing Systems*, volume 21. Curran Associates, Inc., 2009.

[41] Ping Li, Kenneth W. Church, and Trevor J. Hastie. Conditional random sampling: A sketch-based sampling technique for sparse data. In *Proceedings of the 19th International Conference on Neural Information Processing Systems*, NIPS'06, page 873–880, Cambridge, MA, USA, 2006. MIT Press.

[42] Ping Li, Trevor J. Hastie, and Kenneth W. Church. Very sparse random projections. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '06, page 287–296, New York, NY, USA, 2006. Association for Computing Machinery.

[43] Wen Li, Ying Zhang, Yifang Sun, Wei Wang, Mingjie Li, Wenjie Zhang, and Xuemin Lin. Approximate nearest neighbor search on high dimensional data — experiments, analyses, and improvement. *IEEE Transactions on Knowledge and Data Engineering*, 32(8):1475–1488, 2020.

[44] Jimmy Lin. Brute force and indexed approaches to pairwise document similarity comparisons with mapreduce. In *Proceedings of the 32nd International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '09, page 155–162, New York, NY, USA, 2009. Association for Computing Machinery.

[45] Wei Liu, Jun Wang, Sanjiv Kumar, and Shih-Fu Chang. Hashing with graphs. In *Proceedings of the 28th International Conference on International Conference on Machine Learning*, ICML'11, page 1–8, Madison, WI, USA, 2011. Omnipress.

[46] Yu Malkov, Alexander Ponomarenko, Andrey Logvinov, and Vladimir Krylov. Approximate nearest neighbor algorithm based on navigable small world graphs. *Information Systems*, 45:61–68, 01 2013.

[47] Yu A. Malkov and D. A. Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE Trans. Pattern Anal. Mach. Intell.*, 42(4):824–836, April 2020.

[48] Nicholas Metropolis, Arianna W. Rosenbluth, Marshall N. Rosenbluth, Augusta H. Teller, and Edward Teller. Equation of state calculations by fast computing machines. *Journal of Chemical Physics*, pages 1087–1092, 1953.

[49] Selim Mimaroglu and Ertunc Erdil. Combining multiple clusterings using similarity graph. *Pattern Recognition*, 44(3):694–703, 2011.

[50] Stanislav Morozov and Artem Babenko. Non-metric similarity graphs for maximum inner product search. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018.

[51] Anne Morvan, Krzysztof Choromanski, Cédric Gouy-Pailler, and Jamal Atif. Graph sketching-based space-efficient data clustering. *Proceedings of the 2018 SIAM International Conference on Data Mining*, page 10–18, May 2018.

[52] Marius Muja and David G. Lowe. Scalable nearest neighbor algorithms for high dimensional data. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 36(11):2227–2240, 2014.

[53] S. Muthukrishnan. Some algorithmic problems and results in compressed sensing. Allerton Conference, 2006.

[54] Bilegsaikhan Naidan, Leonid Boytsov, and Eric Nyberg. Permutation search methods are efficient, yet faster search is possible. *Proc. VLDB Endow.*, 8(12):1618–1629, August 2015.

[55] Bilegsaikhan Naidan, Leonid Boytsov, and Eric Nyberg. Permutation search methods are efficient, yet faster search is possible. *The Proceedings of the VLDB Endowment (PVLDB)*, 8, 06 2015.

[56] Trong Nhan Phan, Josef Küng, and Tran Khanh Dang. An efficient similarity search in large data collections with mapreduce. In Tran Khanh Dang, Roland Wagner, Erich

Neuhold, Makoto Takizawa, Josef Küng, and Nam Thoai, editors, *Future Data and Security Engineering*, pages 44–57, Cham, 2014. Springer International Publishing.

[57] Trong Nhan Phan, Josef Küng, and Tran Khanh Dang. An elastic approximate similarity search in very large datasets with mapreduce. In Abdelkader Hameurlain, Tran Khanh Dang, and Franck Morvan, editors, *Data Management in Cloud, Grid and P2P Systems*, pages 49–60, Cham, 2014. Springer International Publishing.

[58] Parivash Pirasteh, Dosam Hwang, and Jai E. Jung. Weighted similarity schemes for high scalability in user-based collaborative filtering. *Mob. Netw. Appl.*, 20(4):497–507, August 2015.

[59] Chanop Silpa-Anan and Richard Hartley. Optimised kd-trees for fast image descriptor matching. 06 2008.

[60] Daniel A. Spielman and Shang hua Teng. A local clustering algorithm for massive graphs and its application to nearly-linear time graph partitioning, 2013.

[61] Daniel A. Spielman and Shang hua Teng. Nearly linear time algorithms for preconditioning and solving symmetric, diagonally dominant linear systems, 2014.

[62] Daniel A. Spielman and Nikhil Srivastava. Graph sparsification by effective resistances. *SIAM J. Comput.*, 40(6):1913–1926, 2011.

[63] Daniel A. Spielman and Shang-Hua Teng. Nearly-linear time algorithms for graph partitioning, graph sparsification, and solving linear systems. In *Proceedings of the Thirty-Sixth Annual ACM Symposium on Theory of Computing*, STOC '04, page 81–90, New York, NY, USA, 2004. Association for Computing Machinery.

[64] Daniel A. Spielman and Shang-Hua Teng. Spectral sparsification of graphs. *SIAM J. Comput.*, 40(4):981–1025, July 2011.

[65] Lubomir Stanchev. Semantic document clustering using a similarity graph. In *2016 IEEE Tenth International Conference on Semantic Computing (ICSC)*, pages 1–8, 2016.

[66] Yifang Sun, Wei Wang, Jianbin Qin, Ying Zhang, and Xuemin Lin. Srs: Solving ¡i¿c¡/i¿-approximate nearest neighbor queries in high dimensional euclidean space with a tiny index. *Proc. VLDB Endow.*, 8(1):1–12, September 2014.

[67] Jingdong Wang, Heng Tao Shen, Jingkuan Song, and Jianqiu Ji. Hashing for similarity search: A survey, 2014.

[68] Chuanming Yu, Xiaoli Zhao, Lu An, and Xia Lin. Similarity-based link prediction in social networks: A path and node combined approach. *Journal of Information Science*, 43, 08 2016.

[69] Reza Bosagh Zadeh and Ashish Goel. Dimension independent similarity computation. *J. Mach. Learn. Res.*, 14(1):1605–1626, January 2013.

[70] Ahmad Zareie and Rizos Sakellariou. Similarity-based link prediction in social networks using latent relationships between the users. *Scientific Reports*, 10, 11 2020.

[71] Jingbo Zhou, Qi Guo, H. V. Jagadish, Luboš Krčál, Siyuan Liu, Wenhao Luan, Anthony K. H. Tung, Yueji Yang, and Yuxin Zheng. A generic inverted index framework for similarity search on the gpu - technical report, 2018.

[72] Tülin İnkaya. A parameter-free similarity graph for spectral clustering. *Expert Systems with Applications*, 42(24):9489–9498, 2015.