Group Testing for Accurate and Efficient Range-Based Near Neighbor Search for Plagiarism Detection

Harsh Shah[1], Kashish Mittal[1], and Ajit Rajwade[1]

Department of Computer Science & Engineering, Indian Institute of Technology Bombay

Abstract. This work presents an adaptive group testing framework for the range-based high dimensional near neighbor search problem. Our method efficiently marks each item in a database as neighbor or non-neighbor of a query point, based on a cosine distance threshold without exhaustive search. Like other methods for large scale retrieval, our approach exploits the assumption that most of the items in the database are unrelated to the query. Unlike other methods, it does not assume a large difference between the cosine similarity of the query vector with the least related neighbor and that with the least unrelated non-neighbor. Following a multi-stage adaptive group testing algorithm based on binary splitting, we divide the set of items to be searched into half at each step, and perform dot product tests on smaller and smaller subsets, many of which we are able to prune away. We experimentally show that, using softmax-based features, our method achieves a more than ten-fold speed-up over exhaustive search with no loss of accuracy, on a variety of large datasets. Based on empirically verified models for the distribution of cosine distances, we present a theoretical analysis of the expected number of distance computations per query and the probability that a pool with a certain number of members will be pruned. Our method has the following features: (*i*) It implicitly exploits useful distributional properties of cosine distances unlike other methods; (*ii*) All required data structures are created purely offline; (*iii*) It does not impose any strong assumptions on the number of true near neighbors; (*iv*) It is adaptable to streaming settings where new vectors are dynamically added to the database; and (*v*) It does not require any parameter tuning. The high recall of our technique makes it particularly suited to plagiarism detection scenarios where it is important to report every database item that is sufficiently similar item to the query.

Keywords: near neighbor search, group testing, image retrieval

1 Introduction

Near neighbor (NN) search is a fundamental problem in machine learning, computer vision, image analysis and recommender systems. Consider a database $\mathcal{D}$ of N vectors given as $\{\boldsymbol{f_1}, \boldsymbol{f_2}, \cdots, \boldsymbol{f_N}\}$ where each $\boldsymbol{f_i} \in \mathbb{R}^d$. Given a query vector

$\boldsymbol{q} \in \mathbb{R}^d$, our aim is to determine all those vectors in $\mathcal{D}$ such that $\text{sim}(\boldsymbol{q}, \boldsymbol{f_i}) \geq \rho$, where $\text{sim}(.,.)$ denotes some similarity measure, and ρ denotes some threshold on sim. This is called a similarity measure based NN query. Another variant is the K nearest neighbor (KNN) search where the aim is to find the K most similar neighbors to $\boldsymbol{q}$.

A significant portion of the literature on NN search focusses on *approximate* methods, often resulting in less than perfect recall rates, i.e. many database images that are similar to the query image are not retrieved. This deficiency renders them unsuitable for applications requiring high recall, such as plagiarism detection. In such scenarios, the query vectors, such as images or music files, are compared to an existing database of similar entities, and *all* vectors exhibiting high similarity with the query need to be retrieved for plagiarism assessment. The failure to retrieve a near neighbor could potentially lead to unfair evaluation of submissions to photography or music competitions, or online forums.

Related Work on NN Search: There exists a large body of literature on approximate NN search in higher dimensions. One of the major contributions in this area is called Locality Sensitive Hashing (LSH) [28]. Here, a hashing function partitions the dataset into buckets, such that only close vectors are likely to be hashed to the same bucket. The LSH family of algorithms consists of many variants such as those involving multiple buckets per query vector, data-driven or machine learning based construction of LSH functions [15, 19], and count-based methods [31]. In general, LSH based methods are difficult to implement for very high dimensions owing to the curse of dimensionality in constructing effective hash tables. Dimensionality reduction algorithms can mitigate this issue to some extent, but they are themselves expensive to implement and incur some inevitable data loss. There is also significant research in graph-based methods for nearest neighbor (NN) search, as explored in [22,25,32]. These methods typically achieve high accuracy and efficient query times. However, they often need relatively high pre-processing times, making them less suitable for dynamic datasets that require frequent addition of new points. In addition, various randomized variants of KD-trees and priority search K-means trees have been investigated for efficient approximate NN search in high-dimensional spaces, as seen in [35] and [33]. Despite their efficiency, these approaches involve numerous parameters, which can complicate tuning, especially for large datasets.

Overview of Group Testing: In this work, we present a group testing (GT) based approach for the NN problem. We start by presenting a brief overview of the field of GT. Consider N items $\{x_i\}_{i=1}^N$ (also called 'samples' in many references), out of which only a small number $s \ll N$ are 'defective'. We consider the ith item to be defective if $x_i = 1$ and non-defective if $x_i = 0$. GT involves testing a certain number of '**pools**' ($m \ll N$) to identify the defective items from $\{x_i\}_{i=1}^N$. Each pool (also called 'group') is created by mixing together or combining subsets of the N items. Thus the jth pool ($j \in [m]$) is represented as $y_j = \bigvee_{i=1}^N A_{ji} x_i$, where $\bigvee$ stands for bit-wise OR, and $\boldsymbol{A} \in \{0,1\}^{m \times N}$ is a binary pooling matrix where $A_{ji} = 1$ if the ith item participates in the jth pool and 0 otherwise. GT has a rich literature dating back to the seminal 'Dorfman's

method' [20] which is a popular two-round testing strategy. In the first round of Dorfman's method, some N/K pools, each consisting of K items, are tested. The items that participated in pools that test *negative* (i.e., pools that are deemed to contain no defective item) are declared non-defective, whereas all items that participated in the *positive* pools (i.e., pools that are deemed to contain at least one defective item) are tested individually in the second round. Theoretical analysis shows that for $s \ll N$, this method requires much fewer than N tests in the worst case. This algorithm was extended to some r rounds in [30], where positive groups are further divided into smaller groups, and individual testing is performed only in the rth round. The methods in [20,30] are said to be **adaptive** in nature, as the results of one round are given as input to the next one. The GT literature also contains a large number of **non-adaptive** algorithms that make predictions in a single round. A popular algorithm is called COMP or combinatorial orthogonal matching pursuit, which declares all items participating in a negative pool to be negative, and all the rest to be positive [14]. Many variants of COMP such as Noisy COMP (NCOMP) or Definite Defectives (DD) are also popular [14]. A popular class of non-adaptive GT algorithms [23] draw heavily from the compressed sensing (CS) literature [17, 38]. Such algorithms can consider quantitative information for each $\{x_i\}_{i=1}^{N}$ so that each $x_i \in \mathbb{R}$, and every pool y_j also gives a real-valued result. The pooling matrix $\boldsymbol{A}$ remains binary, and we have the relation $\forall j \in [m], y_j = \sum_{i=1}^{N} A_{ji} x_i$. Given $\{y_j\}_{j=1}^{m}$ and $\boldsymbol{A}$, the unknown vector $\boldsymbol{x}$ can be recovered using efficient convex optimization methods such as the LASSO given by $\hat{\boldsymbol{x}} = \text{Argmin}_{\boldsymbol{x}} \|\boldsymbol{y} - \boldsymbol{A}\boldsymbol{x}\|_2^2 + \lambda \|\boldsymbol{x}\|_1$, where λ is a regularization parameter [26]. As proved in [18, 26], the LASSO estimator produces provably accurate estimates of $\boldsymbol{x}$ given a sufficient number of measurements (or pools) in $\boldsymbol{y}$ and a carefully designed matrix $\boldsymbol{A}$.

Overview of our work: GT algorithms have been applied to the NN search problem recently in [21, 29, 36]. However in these papers, the nearest neighbor queries are of an *approximate* nature, i.e. the results may contain a number of vectors which are not near neighbors of the query vector $\boldsymbol{q}$, and/or may miss some genuine near neighbors. On the other hand, we focus on *accurate* query results with good computational efficiency in practice. Our approach is based on an adaptive GT approach called binary splitting. Like existing GT based algorithms such as [21,29,36], our approach is also based on the assumption that in high-dimensional NN search, most vectors in the database $\mathcal{D}$ are dissimilar to a query vector $\boldsymbol{q}$, and very few vectors qualify as near neighbors. However, our method does not require the distance between the most unrelated neighbor and the most closely related non-neighbor to be large, unlike [21]. Additionally, many existing approaches [1, 15, 16, 21, 24, 29, 33, 34, 36] require parameter tuning in a manner that is not fully data driven. It is also a tedious process for large databases, particularly when dealing with algorithms with large index building times. Moreover, algorithms with large index building times or those unable to efficiently handle the addition of new points are ill-suited for streaming settings. In this work, we present a range-based near neighbor search algorithm in high dimensions built on a group testing framework that has the following appealing

properties for softmax-based feature descriptors: (*i*) low index building time, (*ii*) perfect precision and recall without exhaustive search, (*iii*) no parameter tuning, (*iv*) low time complexity ($\mathcal{O}(1)$) for addition of new vectors making the algorithm suitable for streaming tasks, and (*v*) a unique method of incorporating distributional properties of the similarity values between query and database vectors into the search process.

Organization of the paper: The core method is presented in Sec. 2. We present detailed analytical comparisons with existing GT-based techniques for NN search in Sec. 3. Experimental results and comparisons to a large number of recent NN search methods are presented in Sec. 4. A theoretical analysis of our approach is presented in Sec. 5. We conclude in Sec. 6.

2 Description of Method

In our technique, a pool of high-dimensional vectors is created by simply adding together the participating vectors, which may for example be (vectorized) images or features extracted from them. Consider a query vector $\boldsymbol{q} \in \mathbb{R}^d$ and the database $\mathcal{D}$ consisting of $\{\boldsymbol{f_1}, \boldsymbol{f_2}, \cdots, \boldsymbol{f_N}\}$ where each $\boldsymbol{f_i} \in \mathbb{R}^d$. Throughout this paper, we consider the similarity measure $\text{sim}(\boldsymbol{q}, \boldsymbol{f_i}) \triangleq \boldsymbol{q}^t \boldsymbol{f_i}$. For simplicity, we consider the query vector $\boldsymbol{q}$ as well as the vectors in $\mathcal{D}$ to be element-wise non-negative. This constraint is satisfied by images or their features based on ReLU or softmax non-linearities (see the end of this section for a method to handle negative-valued descriptors). We also consider that the vectors in question have unit magnitude, without any loss of generality. Due to this, $\boldsymbol{q}^t \boldsymbol{f_i}$ acts as a cosine similarity measure.

Our aim is to retrieve all 'defective' members of $\mathcal{D}$, i.e. each member $\boldsymbol{f_i}$ for which $\boldsymbol{q}^t \boldsymbol{f_i} \geq \rho$, where ρ is some pre-specified threshold. With this aim in mind, we aim to efficiently prune away any vector $\boldsymbol{f_i}$ from $\mathcal{D}$ for which it is impossible to have $\boldsymbol{q}^t \boldsymbol{f_i} \geq \rho$. Vector pooling using summations proves to be beneficial for this purpose. Consider the toy example of a pool $\boldsymbol{y} = \boldsymbol{f_1} + \boldsymbol{f_2} + \boldsymbol{f_3} + \boldsymbol{f_4}$. If $\boldsymbol{q}^t \boldsymbol{y} < \rho$ (i.e. the pooled result is negative), then one can clearly conclude that $\boldsymbol{q}^t \boldsymbol{f_i} < \rho$ for each $i \in \{1, 2, 3, 4\}$. However if $\boldsymbol{q}^t \boldsymbol{y} \geq \rho$, i.e. it yields a positive result, then we need to do further work to identify the defective members (if any). For this, we split the original pool and create two pools $\boldsymbol{y_1}$ and $\boldsymbol{y_2}$, each with approximately half the number of members of the original pool. We recursively compute $\boldsymbol{q}^t \boldsymbol{y_2}$, and then obtain $\boldsymbol{q}^t \boldsymbol{y_1} = \boldsymbol{q}^t \boldsymbol{y} - \boldsymbol{q}^t \boldsymbol{y_2}$ (where both $\boldsymbol{q}^t \boldsymbol{y}$ and $\boldsymbol{q}^t \boldsymbol{y_2}$ have already been computed earlier, and hence $\boldsymbol{q}^t \boldsymbol{y_1}$ is directly obtained from the difference between them). Thereafter, we proceed in a similar fashion for both branches.

We now consider the case of searching within the entire database $\mathcal{D}$ defined earlier. We initiate this recursive process with a pool $\boldsymbol{y_0}$ that is obtained from the summation of all N vectors. If $\boldsymbol{y_0}$ yields a positive result, we test the query vector with two pools: $\boldsymbol{y_{11}}$ obtained from a summation of vectors $\boldsymbol{f_1}$ through to $\boldsymbol{f_{\lfloor N/2 \rfloor}}$, and $\boldsymbol{y_{12}}$ obtained from a summation of vectors $\boldsymbol{f_{\lfloor N/2 \rfloor + 1}}$ through to $\boldsymbol{f_N}$. If $\boldsymbol{y_{11}}$ tests negative, then all its members are declared negative and no further tests are required on its members, which greatly reduces the total number of tests.

If $\boldsymbol{y_{11}}$ tests positive, then it is split into two pools $\boldsymbol{y_{21}}, \boldsymbol{y_{22}}$ with roughly equal number of members. Tests are carried out further on these two pools recursively. The same treatment is accorded to $\boldsymbol{y_{12}}$, which would be split into pools $\boldsymbol{y_{23}}, \boldsymbol{y_{24}}$ if it were to test positive. This recursive process stops when the pools obtained after splitting contain just a single vector each, in which case each such single vector is tested against $\boldsymbol{q}$. Clearly, the depth of this recursive process is $\lceil \log_2 N \rceil$ given N vectors in $\mathcal{D}$. This procedure is called **binary splitting**, and it is a multi-round adaptive group testing approach, applied here to NN search. For efficiency of implementation, it is important to be able to create the different pools such as $\boldsymbol{y_0}, \boldsymbol{y_{11}}, \boldsymbol{y_{12}}, \boldsymbol{y_{21}}, \boldsymbol{y_{22}}, \boldsymbol{y_{23}}, \boldsymbol{y_{24}}$, etc., efficiently. For this purpose, we maintain cumulative sums of the following form in memory:

$$\widetilde{\boldsymbol{f}}_{\boldsymbol{1}} = \boldsymbol{f_1}, \widetilde{\boldsymbol{f}}_{\boldsymbol{2}} = \widetilde{\boldsymbol{f}}_{\boldsymbol{1}} + \boldsymbol{f_2}, ..., \widetilde{\boldsymbol{f}}_{\boldsymbol{N}} = \widetilde{\boldsymbol{f}}_{\boldsymbol{N-1}} + \boldsymbol{f_N}. \quad (1)$$

The cumulative sums are useful for efficient creation of various pools. They are created purely *offline*, independent of any query vector. The pools are created using simple vector difference operations, given these cumulative sums. For example, notice that $\boldsymbol{y_0} = \widetilde{\boldsymbol{f}}_{\boldsymbol{N}}, \boldsymbol{y_{11}} = \widetilde{\boldsymbol{f}}_{\boldsymbol{\lfloor N/2 \rfloor}}$ and $\boldsymbol{y_{12}} = \widetilde{\boldsymbol{f}}_{\boldsymbol{N}} - \widetilde{\boldsymbol{f}}_{\boldsymbol{\lfloor N/2 \rfloor}}$. The overall binary splitting procedure is presented in Alg. 1 (a schematic diagram is provided in [13]). For the sake of greater efficiency, it is implemented non-recursively using a simple stack data structure. This binary splitting approach is quite different from Dorfman's algorithm [20], which performs individual testing in the second round. Our approach is more similar to Hwang's generalized binary splitting approach, a GT procedure which was proposed in [27] (see also [8], Alg. 1.1 of [14]). However compared to Hwang's technique, our approach allows for creation of all pools either in a purely offline manner or with simple updates involving a single vector-subtraction operation (for example, $\boldsymbol{y_{12}} = \widetilde{\boldsymbol{f}}_{\boldsymbol{N}} - \widetilde{\boldsymbol{f}}_{\boldsymbol{\lfloor N/2 \rfloor}}$ in the earlier example). On the other hand, the method in [27] detects defectives one at a time, after going through all $\log_2 N$ levels of recursion. Once a defective item is detected, it is discarded and all pools are created again. The entire set of $\log_2 N$ levels of recursion are again executed. This process is repeated until all defectives have been detected and discarded one at a time. The method from [27] is computationally inefficient for high-dimensional NN search, as the pools must be created afresh for each level of recursion. This is because if a vector at index i is discarded, all cumulative sums from index $i+1$ to N need to be updated. Such updates are not required in our method. We also emphasize that the techniques in [14,20,27,30] have all been applied *only* for binary GT and *not* for NN search (see the beginning of Sec. 5 for an important difference between binary GT and NN search). The binary splitting procedure saves on a large number of tests because in high dimensional search, we observe that most vectors in $\mathcal{D}$ are dissimilar to a given query vector $\boldsymbol{q}$, as will be demonstrated in Sec. 4. After the first round where pool $\boldsymbol{y_0}$ is created, all other pools are essentially created randomly (since the initial arrangement of the vectors in $\mathcal{D}$ is random). As a result, across the different rounds, many pools test negative and are dropped out, especially in later rounds. In particular, we note that there is no loss of accuracy in this method, although we save on the number of tests by a factor more than ten on some large datasets (as will be shown in Sec. 4).

Algorithm 1 Iterative Binary Splitting (see [13, Fig. 1] for a diagram)

1: $Fs \leftarrow [\widetilde{\boldsymbol{f_1}}, \widetilde{\boldsymbol{f_2}}, ..., \widetilde{\boldsymbol{f_N}}]$
2: $S \leftarrow$ stack for storing triples $\{si, ei, sim\}$ where si, ei are start and end index for a pool, sim = dot product between query vector $\boldsymbol{q}$ and pool vector.
3: $Flg \leftarrow$ array of size N flags, all 0's initially (0 = non-neighbor, 1 = neighbor of $\boldsymbol{q}$)
4: $S.push\ \{1, N, \boldsymbol{q}^t\widetilde{\boldsymbol{f_N}}\}$, i.e. $si = 1, ei = N$
5: **while** $S.size() \neq 0$ **do**
6: $\{si, ei, sim\} \leftarrow S.top(); S.pop()$
7: **if** $sim \geq \rho$ **then**
8: $n \leftarrow ei - si + 1$
9: **if** $n > 2$ **then**
10: $rsim \leftarrow \boldsymbol{q}^t(\widetilde{\boldsymbol{f_{ei}}} - \widetilde{\boldsymbol{f}}_{\boldsymbol{si+\lfloor n/2 \rfloor -1}})$ ▷ testing right-half
11: $S.push\ \{si + \lfloor n/2 \rfloor, ei, rsim\}$; ▷ push right branch onto stack
12: $S.push\ \{si, si + \lfloor n/2 \rfloor - 1, sim - rsim\}$; ▷ push left branch onto stack
13: **else**
14: **if** $n = 2$ **then**
15: $rsim \leftarrow \boldsymbol{q}^t(\widetilde{\boldsymbol{f_{ei}}} - \widetilde{\boldsymbol{f_{si}}})$
16: **set** $Flg[ei] = 1$, **if** $rsim \geq \rho$
17: **set** $Flg[si] = 1$, **if** $sim - rsim \geq \rho$
18: **else**
19: $Flg[si] = 1$ ▷ Mark si a neighbor
20: **end if**
21: **end if**
22: **end if**
23: **end while**

Pooling using Element-wise Maximum: We have so far described pool creation using vector summation. However, pools can also be created by computing the element-wise maximum of all vectors participating in the pool. Given a subset of vectors $\{\boldsymbol{f_k}\}_{k=1}^K$, such a pool $\boldsymbol{y_m}$ is given by: $\forall j \in \{1, 2, ..., d\}, y_{m,j} = \max_j(f_{k,j})$ where $f_{k,j}$ stands for the jth element of $\boldsymbol{f_k}$. We denote this operation as $\boldsymbol{y_m} = \max(\{\boldsymbol{f_k}\}_{k=1}^K)$. If $\boldsymbol{q}^t\boldsymbol{y_m} < \rho$, it follows that $\boldsymbol{q}^t\boldsymbol{f_k} < \rho$ for every $\boldsymbol{f_k}$ that contributed to $\boldsymbol{y_m}$. Hence pools created using an element-wise maximum are also well suited for iterative binary splitting in a similar fashion as in Alg. 1. Separate element-wise maximum vectors need to be stored in memory for each pool. In case of negative values in the query and/or database vectors, the element-wise maximum can be replaced by an element-wise minimum for every index j for which $q_j < 0$. Unlike the summation technique, this handles negative-valued descriptors, albeit at the cost of more memory in order to store both element-wise maximum and element-wise minimum values for each pool.

3 Comparison to other Group Testing Methods for NN Search

GT algorithms of different flavors have been applied to the NN search problem in [21, 29, 36]. A detailed description of these techniques is presented in the

supplemental material at [13, Sec. 2]. Compared to these three afore-mentioned techniques, our method is exact, with the same accuracy as exhaustive search. The methods [21,29,36] require choice of various hyper-parameters (R, the number of backpropagation steps, for [36]; sparse coding parameters in [29]; bloom filter and hash function parameters (B, R, m, L) in [21] – see [13, Sec. 2]) for which there is no clear data-driven selection procedure. Our method, however, requires no parameter tuning. In experiments, we have observed a speed up of more than ten-fold in querying time with our method as compared to exhaustive search on some datasets. Like [36], and unlike [21, 29], our method does have a large memory requirement, as we require all pools to be in memory. Some techniques such as [21] require the nearest neighbors to be significantly more similar than all other members of $\mathcal{D}$ (let us call this condition $\mathcal{C}1$). Our method does not have such a requirement. However, our method will perform more efficiently for queries for which a large number of similarity values turn out to be small, and only a minority are above the threshold ρ (let us call this condition $\mathcal{C}2$). In our experimental study on diverse datasets, we have observed that $\mathcal{C}2$ is true always. On the other hand, $\mathcal{C}1$ does not hold true in a large number of cases, as also reported in [21, Sec. 5.3].
We also experimented with applying the latest developments in the compressed sensing (CS) literature [38] (which is closely related to GT) to the *approximate* NN search problem, albeit unsuccessfully. The theoretical reasons for the failure of the latest CS techniques for NN search are described in [13, Sec. 3].

4 Experimental Results

In this section, we present experimental results conducted in two settings: (i) Non-streaming setting with a static database, and (ii) Streaming setting with incremental additions of new vectors to the database. In both the settings, we tested our algorithms on the following publicly available image datasets, of which the first two are widely used in retrieval and NN search problems: (i) MIRFLICKR, a set of 1 million (1M) images from the MIRFLICKR dataset (subset of images from Flickr) available at [11], with an additional 6K images from the Paris dataset available at [12], as followed in [36]; (ii) ImageNet, a set of 1M images used in the ImageNet Object Localization Challenge, available at [2]; (iii) IMDB-Wiki, a set of all 500K images from the IMDB-Wiki face image dataset from [9]; (iv) InstaCities, a set of 1M images of 10 large cities [10]. Although, we have worked with image datasets, our method is equally applicable to any other modalities such as speech, genomics, etc.

In the following, we denote a query image by $\boldsymbol{I_q}$. For every dataset containing images $\{\boldsymbol{I_i}\}_{i=1}^N$, feature vectors $\{\boldsymbol{f_i}\}_{i=1}^N$ were extracted as follows: First, each image $\boldsymbol{I_i}$ was passed through the popular VGGNet [37] (as its features are known to be useful for building good perceptual metrics [39]) to obtain intermediate feature vector $\boldsymbol{f'_i}$. Second, a softmax-based feature vector $\boldsymbol{f_i}$ of dimension $d = 1000$, corresponding to the 1000 classes in ImageNet, was extracted further from the VGGNet output vector $\boldsymbol{f'_i}$. In our experiments, we observed

that the softmax-based features yielded superior retrieval recall (defined as # points *retrieved* belonging to the same class as the query point / # points of the same class as the query point) as compared to the baseline VGGNet features, with exhaustive search on different datasets (see [13, Table 1] for more details), even on those datasets which are different from ImageNet. Higher recall is particularly beneficial for NN search application in scenarios such as image plagiarism detection. The feature vectors were all non-negative and were subsequently unit-normalized. For every dataset, NN search was carried out using different methods (see below) on the same unit-normalized feature vectors extracted from 10,000 query images. That is, the aim was to efficiently identify those indices $i \in \{1, 2, ..., N\}$ for which the dot products between $\boldsymbol{q}$ (the feature vector of query image $\boldsymbol{I_q}$) and $\boldsymbol{f_i}$ (both feature vectors unit-normalized) exceeded some specified threshold ρ. For all datasets, the intrinsic dimensionality was computed using the number of singular values of the $d \times N$ data matrix that accounted for 99% of its Frobenius norm. The intrinsic dimensionality for all datasets was more than 200 (for $d = 1000$).

Our binary splitting approach, using pool creation with summation (Our-Sum) and element-wise maximum (Our-Max), was compared to the following recent and popular techniques: (*i*) Exhaustive search (Exh) through the entire database, for every query image; (*ii*) The recent GT-based technique from [21], known as Flinng (Filters to Identify Near-Neighbor Groups), using code from [3]; (*iii*) The popular randomized KD-tree and K-means based technique Flann for approximate NN search [33], using code from [7] with default parameters; (*iv*) The Falconn technique from [15] which uses multi-probe LSH, using the code from [4]; (*v*) Falconn++ [34], an enhanced version of Falconn which filters out potentially distant points from any hash bucket before querying using code from [5]; (*vi*) Ivf, the speedy inverted multi-index technique implemented in the highly popular Faiss library from Meta/Facebook [1, 16]; (*vii*) Hnsw, a graph-based technique also used in [1]; (*viii*) The Scann library [6, 24] which implements an anisotropic vector quantization (KMeans) algorithm. Our methods have no hyperparameters, but for all competing methods, results are reported by tuning hyperparameters so as to maximize the recall (defined below) – see [13, Sec. 5] for more details.

We did not compare with the GT-based techniques from [36] and [29] as we could not find full publicly available implementations for these. Moreover, the technique from [21] is claimed to outperform those in [36] and [29]. We did implement the algorithm from [36], but it yielded significantly large query times. This may be due to subtle implementation differences in the way image lists are re-ranked in each of the R iterations of their algorithm (see Sec. 3). In any case, our method produces 100% recall and precision, unlike [29, 36].

Non-streaming Setting: The different methods are compared in terms of Mean Query Time (`MQT`), Mean Precision (`MP`) and Mean Recall (`MR`) for 10K queries as reported in Table 1. The similarity threshold (ρ) chosen for this setting is 0.8. The recall is defined as as #true positives/(#true positives + #false negatives). The precision is defined as # true positives/(#true positives+

false positives). For each dataset, we have also mentioned the average number of neighbours ($\hat{k}$) having similarity greater than $\rho = 0.8$. The cumulative sums in Eqn. 1 for Our-Sum and the cumulative element-wise maximums for Our-Max were computed offline. The query implementation was done in C++ with the highest level of optimization in the g++ compiler. The codes for Flinng, Falconn++, Scann and Hnsw operate in the KNN search mode, as opposed to a range-based query for Our-Sum, Our-Max, Falconn and Ivf, for which we set $\rho = 0.8$. The Flann method has code support for both KNN and range-based queries, which we refer to as Flann-K and Flann-R respectively. For all KNN-based methods, we gave an appropriate K as input, such that K ground truth neighbors satisfy the similarity threshold of ρ. The time taken to compute K was *not* included in the query times reported for the KNN-based methods. All query times are reported for an Intel(R) Xeon(R) server with CPU speed 2.10GHz and 128 GB RAM. For all methods, the computed query times did not count the time taken to compute the feature vector for the query image.

Streaming Setting: We now consider experiments in a streaming setting where new vectors are added to the database on the fly, i.e. the database is modified incrementally. In our experimental setting, 80% of the full database was initially available to all algorithms for index creation. New points were incrementally added to the database, and after addition of 100 new points, a search query was fired. This process was carried out until the entire database was loaded into memory. The query images were created by introducing controlled perturbations such as dither and blur to images existing in the database (refer to [13] for more details). The similarity threshold for this setting is chosen to be $\rho = 0.9$. Note that this setting is analogous to a typical image plagiarism detection setting, wherein (a) the database is updated with insertion of new points (without deletion) and requires fetching high similarity vectors, and (b) the submitted images could be subtly manipulated to escape plagiarism detection. High recall is desirable in such applications.

The only update required to Our-Sum for insertion of n_s new points will be to produce n_s extra cumulative sums as in Eqn. 1 for a cost of $O(dn_s)$. For comparison, we have selected only those algorithms that include methods for the addition of new points without necessitating the rebuilding of the entire index from scratch. These include Exh, Ivf, Hnsw and Flann-R. Table 2 reports the initial Index Build Time (`IBT`) for 80% of the database, Mean Insertion Time (`MIT`) for new points, besides `MQT`, `MP` and `MR`.

Discussion of Results on Static Datasets (Table 1): We observe that Our-Sum yields lower `MQT` on most datasets as compared to most other techniques, and that too with a guarantee of 100% precision and recall. Falconn produces higher query times than Our-Sum or Our-Max and at the loss of some recall. Falcon++ produces 100% precision and recall but with much higher query times than Our-Sum or Our-Max. Flinng and Flann-K are efficient, but have significantly lower precision and recall than Our-Sum with many of the retrieved dot products being significantly lower than the chosen ρ. Flann-R yields very low recall despite its excellent precision and query time.

HNSW produces higher query times in cases where the number of neighbors is large with some loss of recall. The closest competitors to OUR-SUM are SCANN and IVF, but they do not guarantee 100% precision and recall. In addition, we note that as reported in [24, Fig. 3a], the accuracy of the dot-products retrieved by SCANN will depend on the chosen bit-rate, i.e. the chosen *number of cluster centers* in KMeans, whereas OUR-SUM has no such dependencies or parameter choice. This is a major conceptual advantage of OUR-SUM over SCANN (also see the 'Streaming' setting below). As compared to exhaustive search, OUR-SUM produces a speed gain between 4 to 120 depending on the dataset. OUR-SUM outperforms OUR-MAX in terms of query time, but it is important note that OUR-MAX is capable of handling negative-valued descriptors, although we have not experimented with it in this paper.

	Db.	OUR-S	OUR-M	IVF	FALC	FLANN-R	FALC++	FLINNG	FLANN-K	SCANN	HNSW	EXH
MQT	MIRFL.	54.0	81	94	302	1	509	62	54	23	66	1222
MP	($\hat{k} \approx 1.7K$)	1.0	1.0	1.0	≈1.0	1.0	1.0	0.268	0.743	0.996	0.959	1.0
MR		1.0	1.0	0.997	0.954	0.323	1.0	0.268	0.743	0.996	0.959	1.0
MQT	ImgN.	4.9	7	103	179	2	385	119	40	23	9	1276
MP	($\hat{k} \approx 0.9K$)	1.0	1.0	1.0	1.0	1.0	1.0	0.372	0.310	0.998	0.965	1.0
MR		1.0	1.0	0.997	0.958	0.217	1.0	0.372	0.310	0.998	0.965	1.0
MQT	IMDB.	98.8	249	49	415	1.0	739	46	98	60	1076	800
MP	($\hat{k} \approx 7K$)	1.0	1.0	1.0	≈1.0	1.0	1.0	0.351	0.656	0.997	0.985	1.0
MR		1.0	1.0	0.995	0.995	0.191	1.0	0.351	0.656	0.997	0.985	1.0
MQT	InstaC.	58.0	110	90	330	1	601	57	61	25	298	1174
MP	($\hat{k} \approx 3K$)	1.0	1.0	1.0	≈1.0	≈1.0	1.0	0.293	0.745	0.996	0.952	1.0
MR		1.0	1.0	0.990	0.964	0.324	1.0	0.293	0.745	0.996	0.952	1.0

Table 1: Comparison of mean query time (MQT, ↓) in milliseconds, and mean precision (MP, ↑) and mean recall (MR, ↑) values for our method with summation pooling (OUR-S.); our method with max pooling (OUR-M.); IVF from the FAISS package [1,16]; HNSW from the FAISS package [1,32]; FALCONN [4,15]; FALCONN++ [34]; FLINNG [21]; FLANN [33] with KNN-based (FLANN-K) and range-based queries (FLANN-R); SCANN [6] and exhaustive search (EXH). For all methods $\rho = 0.8$, see text for more details. $\hat{k} \triangleq$ avg. number of database vectors satisfying $\boldsymbol{q}^t \boldsymbol{f_i} \geq \rho$.

Discussion of Results on Datasets with Streaming (Table 2): It is clear that the Index Build Time (IBT) and Mean Insertion Time (MIT) of our method clearly surpasses that of all other methods across all datasets. Furthermore, the space taken for storing the index is only $O(Nd)$ in our method. Since OUR-SUM only has to compute cumulative sums for the newly inserted points, the index update time is very low when compared with other methods. FLANN-R consistently produces lower mean query times, but this comes at the cost of low recall and high insertion time, making it unusable for streaming applications. Similarly, HNSW shows significant gains in MQT for MIRFLICKR, ImageNet and InstaCities datasets, but exhibits a large insertion time across all datasets.

Analysis of Pruning (Non-Streaming): For each dataset, we also computed the number of pools that got rejected in every '**round**' of binary splitting (because the dot product with the query image fell below threshold ρ), beginning from round 0 to $\lceil \log_2 N \rceil$. (Referring to Alg. 1, we see that the procedure im-

Index Build Time (in ms) ↓				
Alg.↓, Db. →	ImgN.	IMDB.	MIRFL.	InstaC.
Our-Sum	3789	2000	3774	3765
Exh.	NA	NA	NA	NA
Ivf	15000	9960	15410	18722
Hnsw	7e5	2e5	4e5	4e5
Flann-R	2e6	9e5	2e6	2e6

Average Insertion Time (in ms) ↓				
Alg.↓, Db. →	ImgN.	IMDB.	MIRFL.	InstaC.
Our-Sum	0.28	0.29	0.28	0.29
Exh.	~ 0	~ 0	~ 0	~ 0
Ivf	2.2	2.7	2.2	2.7
Hnsw	236	239	222	219
Flann-R	42	53	27	26

Mean Query Time (in ms) ↓				
Alg.↓, Db. →	ImgN.	IMDB.	MIRFL.	InstaC.
Our-Sum	18	121	86	115
Exh.	609	315	607	605
Ivf	183	67	145	132
Hnsw	10	168	13	70
Flann-R	6	4	5	5

Mean Precision ↑, Mean Recall ↑				
Alg.↓, Db. →	ImgN.	IMDB.	MIRFL.	InstaC.
Our-Sum	1,1	1,1	1,1	1,1
Exh.	1,1	1,1	1,1	1,1
Ivf	1,0.99	1,0.99	1,0.99	1,0.99
Hnsw	0.96,0.96	0.98,0.98	0.97,0.97	0.97,0.97
Flann-R	1,0.31	1,0.59	1,0.49	1,0.54

Table 2: Streaming results: Index Build Time (IBT) for 80% of the database, mean insertion time (MIT) for new points, MQT, MP and MR. For IMDB, the number of insertion operations and search queries is ∼700 and ∼1350 respectively. For all other datasets, the numbers are ∼1400 and ∼2600 respectively.

plements an inorder traversal of the binary tree implicitly created during binary splitting. However the same tree can be traversed in a level-wise manner, and each level of the tree represents one '**round**' of binary splitting.) The corresponding histograms for $\rho \geq 0.7$ are plotted in Fig. 2 of [13], for Our-Sum. As can be seen, a large number of rejections occur from round 14 onward in MIRFLICKR and InstaCities, from round 11 onward in ImageNet, and from round 16 onward in IMDB-Wiki. As a result, the MQT is lowest for ImageNet than for other datasets, and also much lower for MIRFLICKR and InstaCities than for IMDB-Wiki. A similar set of histograms are plotted for Our-Max in Fig. 3 of [13]. Furthermore, we plotted histograms of dot products between every query feature vector and every gallery feature vector, across 10,000 queries for all datasets. These are plotted in Fig. 1, clearly indicating that the vast majority of dot products lie in the bin $(0, 0.1]$ or $(0.1, 0.2]$ (also see Fig. 1 of [13], which shows negative logarithms of the histograms for clearer visualization of values close to 0). This supports the hypothesis that most items in $\mathcal{D}$ are dissimilar to a query vector. However, as seen in Fig. 1, the percentage of dot products in the $(0, 0.1]$ bin is much larger for ImageNet, MIRFLICKR and InstaCities than for IMDB-Wiki. This tallies with the significantly larger speedup obtained by binary splitting (both Our-Sum and Our-Max) for ImageNet, MIRFLICKR and InstaCities as compared to IMDB-Wiki – see Table 1.

5 Theoretical Analysis

For the problem of binary GT, different variants of the binary splitting approach have been shown to require just $s \log_2 N + O(s)$ or $s \log_2(N/s) + O(s)$ tests, depending on some implementation details [14, Theorems 1.2 and 1.3]. Here s

stands for the number of defectives out of N. However in binary GT, a positive result on a pool necessarily implies that at least one of the participants was defective. In our application for NN search, a pool $\boldsymbol{y}$ can produce $\boldsymbol{q}^t\boldsymbol{y} \geq \rho$ for query vector $\boldsymbol{q}$, even if *none* of the vectors that contributed to $\boldsymbol{y}$ produce a dot product with $\boldsymbol{q}$ that exceeds or equals ρ. Due to this important difference, the analysis from [14] does not apply in our situation. Furthermore, it is difficult to pin down a precise *deterministic* number of tests in our technique. Instead, we resort to a distributional assumption on each dot product $\boldsymbol{q}^t\boldsymbol{f_i}$. We assume that these dot products are independently distributed as per a truncated normalized exponential (TNE) distribution:

$$p(x;\lambda) = \begin{cases} \dfrac{\lambda e^{-\lambda x}}{1-e^{-\lambda}}, \text{ for } 0 \leq x \leq 1. \\ 0, \text{ otherwise.} \end{cases} \tag{2}$$

This distribution is obtained by truncating the well-known exponential distribution to the $[0,1]$ range and normalizing it so that it integrates to one. Larger the value of λ, the more rapid is the 'decay' in the probability of dot product values from 0 towards 1. Our empirical studies show that for softmax features, this is a reasonable model for dot product values commonly encountered in queries in image retrieval, as can be seen in Fig. 1 for all datasets we experimented with. Also, the best-fit λ values for MIRFLICKR, ImageNet, IMDB-Wiki and InstaCities are respectively $34, 57, 10, 30$. The TNE model fit may not be perfect, but less than perfect adherence to the TNE model does not change the key message that will be conveyed in this section.

Our aim now is to determine the probability that a pool $\boldsymbol{y}$ with some L participating vectors will produce a dot product value $\boldsymbol{q}^t\boldsymbol{y}$ below a threshold ρ. Note again that such a pool will get pruned away in binary splitting, and therefore we would like this probability to be high. Now, the L individual dot-products (of $\boldsymbol{q}$ with vectors that contributed to $\boldsymbol{y}$) are independent and they are all distributed as per the TNE from Eqn. 2. The mean and variance of a TNE variable with parameter λ are respectively given by $\left(1/\lambda + 1/(1-e^{\lambda})\right)$ and $\left(1/\lambda^2 - e^{\lambda}/(e^{\lambda}-1)^2\right)$. We now apply the central limit theorem (CLT), due to which we can conclude that $\boldsymbol{q}^t\boldsymbol{y}$ is (approximately) Gaussian distributed with a mean $\mu \triangleq L\left(1/\lambda + 1/(1-e^{\lambda})\right)$ and variance $\sigma^2 \triangleq L\left(1/\lambda^2 - e^{\lambda}/(e^{\lambda}-1)^2\right)$. With this, we can easily compute $P(\boldsymbol{q}^t\boldsymbol{y} \leq \rho)$ given a value of L and ρ. Our numerical experiments, illustrated in Fig. 3 of [13], reveal that this probability increases with λ and decreases as L increases. This tallies with intuition, and supports the hypothesis that a larger number of pools will be pruned away in a given round if there is more rapid decrease in the probability of dot product values from 0 to 1.

We are now equipped to determine the expected number of tests (dot products) to be computed per query, given such a TNE distribution for the dot products. Given a value of ρ, let us denote the probability that a pool with L participants will get pruned away, by $p_L(\rho)$. For Our-Sum, for any $L \geq 6$, the probability $p_L(\rho)$ is computed using the TNE and CLT based method described

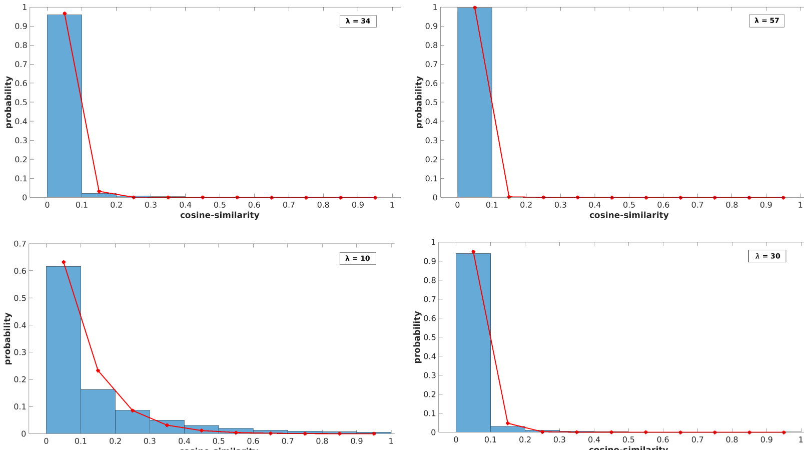

Fig. 1: Normalized histograms (bar graphs in blue), and best-fit exponential distribution approximations (red curves), of dot products between feature vectors of query images (10,000 in number) and all images of MIRFLICKR, ImageNet, IMDB-Wiki and InstaCities databases (left to right, top to bottom). Also see Fig. 1 of [13] for a plot of negative log of the histograms for clearer visualization of the smaller probability values.

earlier. For $L < 6$, the Gaussian distribution can be replaced by a truncated Erlang distribution, as the sum of independent exponential random variables follows an Erlang distribution. Even in this case, $P(\boldsymbol{q}^t\boldsymbol{y} \leq \rho)$ increases with λ and decreases with L.

In round 1, there is only one pool and it contains all N vectors. It will get pruned away with probability $p_N(\rho)$, for a given fixed threshold ρ and fixed λ. The expected number of pools in round 2 will be $Q_2 \triangleq 2(1-p_N(\rho))$, each of which will be pruned away with probability $p_{N/2}(\rho)$. The factor 2 is because each pool gets split into two before sending it to the next round of binary splitting. The expected number of pools in round 3 will be $Q_3 \triangleq 2(1 - p_{N/2}(\rho))Q_2$. Likewise, in the kth round, the expected number of pools is $Q_k = 2(1 - p_{N/2^{k-2}}(\rho))Q_{k-1}$. Hence the expected number of tests per query in our approach will be given by:

$$E(\rho) \triangleq 1 + \frac{1}{2} \sum_{i=2}^{\lceil \log_2 N+1 \rceil} Q_i, \tag{3}$$

as there are at most $\lceil \log_2 N + 1 \rceil$ rounds. The factor $\frac{1}{2}$ is due to the fact that the number of tests (i.e. dot product computations) is always half the number of pools (see second para. of Sec. 2). The value of $E(\rho)$ decreases with λ as $p_N(\rho), p_{N/2}(\rho), ..., p_{N/2^{k-2}}(\rho)$ all increase with λ. A similar analysis for OurMax is presented in [13, Sec. 4].

Our theoretical analysis depends on λ, but note that our algorithm is completely agnostic to λ and does not use it in any manner. For values of $\rho \in \{0.7, 0.8, 0.9\}$, we compared the theoretically predicted average number of dot products ($E(\rho)$) to the experimentally observed number for all datasets. These are presented in Tab. 3, where we observe that for MIRFLICKR, ImageNet and InstaCities, the theoretical number for OUR-SUM is generally *greater* than the empirical one. This is because the dot products for these datasets decay slightly faster than what is predicted by a TNE distribution, as can be seen in Fig. 1. Conversely, for IMDB-Wiki, the theoretical number for OUR-SUM is *less* than the empirical one because the dot products decay slower for this dataset than what is predicted by a TNE distribution. For OUR-MAX, the difference between empirical and theoretical values is larger, as explain in [13].

Database			
MIRFLICKR, $N = 1M$	$\rho = 0.7$	$\rho = 0.8$	$\rho = 0.9$
Empirical (Sum)	51690	44194	37788
Empirical (Max)	75782	60752	47531
Theoretical (Sum)	63838	57553	50094
Theoretical U.B. (Max)	371672	270408	195140
ImageNet, $N = 1M$	$\rho = 0.7$	$\rho = 0.8$	$\rho = 0.9$
Empirical (Sum)	13852	12339	10801
Empirical (Max)	6946	5713	4551
Theoretical (Sum)	34524	32603	31345
Theoretical U.B. (Max)	14160	6267	2775
IMDB-Wiki, $N = 500K$	$\rho = 0.7$	$\rho = 0.8$	$\rho = 0.9$
Empirical (Sum)	160020	141290	125050
Empirical (Max)	240947	197370	158284
Theoretical (Sum)	113600	99552	87835
Theoretical U.B. (Max)	1426259	1336543	1249448
InstaCities, $N = 1M$	$\rho = 0.7$	$\rho = 0.8$	$\rho = 0.9$
Empirical (Sum)	72729	62741	54125
Empirical (Max)	105122	85188	67441
Theoretical (Sum)	71021	63453	57944
Theoretical U.B. (Max)	566498	445181	346927

Table 3: Empirically observed average and theoretically predicted expected number of dot products (Eqn. 3) for queries with $\rho \in \{0.7, 0.8, 0.9\}$ for different datasets. See explanation at the end of Sec. 5. Note that 'Sum' and 'Max' denote algorithms OUR-SUM and OUR-MAX respectively.

6 Conclusion

We have presented a simple and intuitive multi-round group testing approach for range-based near neighbor search. Our method outperforms exhaustive search by a large factor in terms of query time without losing accuracy, on datasets where query vectors are dissimilar to the vast majority of vectors in the gallery. The larger the percentage of dissimilar vectors, the greater is the speedup our algorithm will produce. Our technique has no tunable parameters. It is also easily applicable to situations where new vectors are dynamically added to an existing gallery, without requiring any expensive updates. Apart from this, our technique also has advantages in terms of accuracy of query results (especially recall) over many recent, state of the art methods. A significant limitation of our method is that it currently only supports cosine distances and is applicable only to databases with a sharp decay in feature similarity distribution, as in the case with softmax features. At present, a pool may produce a large dot product value with a query vector, even though many or even all the members of the pool are dissimilar to the query vector. Developing techniques to reduce the frequency of such situations, possibly using various LSH techniques, as well as testing on billion-scale datasets, are important avenues for future work. The implementation of our algorithms can be found at https://github.com/Harsh-Sensei/GroupTestingNN.

Acknowledgement: The authors thank the Amazon IIT Bombay AI-ML Initiative for support for the work in this paper.

References

1. https://github.com/facebookresearch/faiss 3, 8, 10
2. https://www.kaggle.com/competitions/imagenet-object-localization-challenge/data 7
3. https://github.com/JoshEngels/FLINNG 8
4. https://github.com/FALCONN-LIB/FALCONN 8, 10
5. https://github.com/NinhPham/FalconnLSF 8
6. https://github.com/google-research/google-research/tree/master/scann 8, 10
7. FLANN - fast library for approximate nearest neighbors: wbia-pyflann 4.0.4. https://pypi.org/project/wbia-pyflann/ 8
8. Generalized binary splitting algorithm. https://en.wikipedia.org/wiki/Group_testing#Generalised_binary-splitting_algorithm 5
9. Imdb-wiki – 500k+ face images with age and gender labels. https://data.vision.ee.ethz.ch/cvl/rrothe/imdb-wiki/ 7
10. The InstaCities1M Dataset. https://gombru.github.io/2018/08/01/InstaCities1M/ 7
11. MIRFLICKR download. https://press.liacs.nl/mirflickr/mirdownload.html 7
12. Oxford and paris buildings dataset. https://www.kaggle.com/datasets/skylord/oxbuildings 7
13. Supplemental material. Uploaded on conference portal 5, 6, 7, 8, 9, 11, 12, 13, 14
14. Aldridge, M., Johnson, ., Scarlett, J.: Group testing: an information theory perspective. Foundations and Trends in Communications and Information Theory **15**(3-4), 196–392 (2019) 3, 5, 11, 12
15. Andoni, A., Razenshteyn, I.: Optimal data-dependent hashing for approximate near neighbors. In: Proceedings of the forty-seventh annual ACM symposium on Theory of computing. pp. 793–801 (2015) 2, 3, 8, 10
16. Babenko, A., Lempitsky, V.: The inverted multi-index. IEEE Transactions on Pattern Analysis and Machine Intelligence **37**(6), 1247–1260 (2014) 3, 8, 10
17. Candès, E.J., Wakin, M.B.: An introduction to compressive sampling. IEEE signal processing magazine **25**(2), 21–30 (2008) 3
18. Davenport, M., Duarte, M., Eldar, Y., Kutyniok, G.: Introduction to compressed sensing. (2012) 3
19. Dong, Y., Indyk, P., Razenshteyn, I., Wagner, T.: Learning space partitions for nearest neighbor search. In: ICLR (2019) 2
20. Dorfman, R.: The detection of defective members of large populations. The Annals of mathematical statistics **14**(4), 436–440 (1943) 3, 5
21. Engels, J., Coleman, B., Shrivastava, A.: Practical near neighbor search via group testing. In: Advances in Neural Information Processing Systems. vol. 34, pp. 9950–9962 (2021) 3, 6, 7, 8, 10
22. Fu, C., Xiang, C., Wang, C., Cai, D.: Fast approximate nearest neighbor search with the navigating spreading-out graph **12**(5), 461–474 (2019) 2

23. Ghosh, S., Agarwal, R., Rehan, M.A., Pathak, S., Agarwal, P., Gupta, Y., Consul, S., Gupta, N., Goenka, R., Rajwade, A., Gopalkrishnan, M.: A compressed sensing approach to pooled RT-PCR testing for COVID-19 detection. IEEE Open Journal of Signal Processing **2**, 248–264 (2021) 3
24. Guo, R., Sun, P., Lindgren, E., Geng, Q., Simcha, D., Chern, F., Kumar, S.: Accelerating large-scale inference with anisotropic vector quantization. In: International Conference on Machine Learning. pp. 3887–3896 (2020) 3, 8, 10
25. Hajebi, K., Abbasi-Yadkori, Y., Shahbazi, H., Zhang, H.: Fast approximate nearest-neighbor search with k-nearest neighbor graph. In: IJCAI. pp. 1312–1317 (2011) 2
26. Hastie, T., Tibshirani, R., Wainwright, M.: Statistical learning with sparsity: the lasso and generalizations. CRC press (2015) 3
27. Hwang, F.: A method for detecting all defective members in a population by group testing. Journal of the American Statistical Association **67**(339), 605–608 (1972) 5
28. Indyk, P., Motwani, R.: Approximate nearest authoneighbors: towards removing the curse of dimensionality. In: STOC. p. 604–613 (1998) 2
29. Iscen, A., Chum, O.: Local orthogonal-group testing. In: Proceedings of the European Conference on Computer Vision (ECCV). pp. 449–465 (2018) 3, 6, 7, 8
30. Li, C.H.: A sequential method for screening experimental variables. Journal of the American Statistical Association **57**(298), 455–477 3, 5
31. Lv, Q., Josephson, W., Wang, Z., Charikar, M., Li, K.: Multi-probe LSH: efficient indexing for high-dimensional similarity search. In: Proceedings of the 33rd international conference on Very large data bases. pp. 950–961 (2007) 2
32. Malkov, Y., Yashunin, D.: Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. IEEE Transactions on Pattern Analysis and Machine Intelligence **42**(4), 824–836 (2018) 2, 10
33. Muja, M., Lowe, D.: Scalable nearest neighbor algorithms for high dimensional data. IEEE Transactions on Pattern Analysis and Machine Intelligence **36**(11), 2227–2240 (2014) 2, 3, 8, 10
34. Pham, N., Liu, T.: Falconn++: A locality-sensitive filtering approach for approximate nearest neighbor search. In: Advances in Neural Information Processing Systems. vol. 35, pp. 31186–31198 (2022) 3, 8, 10
35. Ram, P., Sinha, K.: Revisiting kd-tree for nearest neighbor search. In: SIGKDD. pp. 1378–1388 (2019) 2
36. Shi, M., Furon, T., Jégou, H.: A group testing framework for similarity search in high-dimensional spaces. In: ACMMM. pp. 407–416 (2014) 3, 6, 7, 8
37. Simonyan, K., Zisserman, A.: Very deep convolutional networks for large-scale image recognition. In: International Conference on Learning Representations (2015) 7
38. Vidyasagar, M.: An introduction to compressed sensing. SIAM (2019) 3, 7
39. Zhang, R., Isola, P., Efros, A.A., Shechtman, E., Wang, O.: The unreasonable effectiveness of deep features as a perceptual metric. In: CVPR. pp. 586–595 (2018) 7