IT UNIVERSITY OF COPENHAGEN

Master Thesis

Similarity Search with Spectral Hashing

Author: Irina Alina Gabriela Luca
Supervisor: Samuel McCauley, Ph.D. Computer Science

IT University of Copenhagen
Copenhagen, Denmark

A thesis submitted in fulfillment of the requirements for the degree of Master of Science.

January 2018

Abstract

The purpose of this thesis is to explore a binary code encoding technique of Y. Weiss, A. Torralba and R. Fergus called *Spectral Hashing* [46]. We aim to shed light on the underlying theory and provide a deeper understanding with regards to its implementation.

Based on the original Spectral Hashing paper and the related literature, we discuss the constraints the algorithm relies on and shape the theoretical frame around it in order to find the purpose behind each of its underlying mechanisms.

Furthermore, we evaluate the performance of Spectral Hashing over several datasets and investigate parameter dependencies. We propose a series of heuristic approaches and compare them in terms of performance. Among these, one seems to perform better than *vanilla* Spectral Hashing overall. In order to achieve this, we explore some of the characteristics of Spectral Hashing and build our optimizations on top of them.

We analyze the evaluation type proposed by the authors of the original Spectral Hashing paper. We also propose a slightly different evaluation type, where we consider F_1-score a more appropriate quality metric to use in the context of our application. Depending on the number of bits, we provide a method to calculate the average Hamming radius over a given dataset required to search and find the nearest neighbors with the best F_1-score.

Thus, this thesis looks deeper into the Spectral Hashing implementation, links it with the underlying theory and proposes a series of SH heuristics. The supporting Python source code can be visited at: `https://github.itu.dk/irlu/SH` or `https://github.com/irina2014/Spectral_Hashing`.

Acknowledgments

First and foremost, I would like to express the deepest gratitude to my supervisor Samuel McCauley, for his extraordinary commitment to this project, for his genius ideas, for his incredible dynamism and positive energy, for making this experience an exciting adventure, for taking all the time in the world to meet, including off-days, holidays, his own deadlines etc., for explaining me a thousand times until I understood, for giving me space in the hardest moments and helping me find the good solutions all the time, for his exceptional teaching skills and for the fact that he read through the student in me as through an open book and filled the many blank spaces. Without his companionship, this thesis would have not been possible.

I would also like to thank my schoolmate Jan Enghoff for all his support, for all the nice conversations and explanations which led me to improve in key points of the thesis, for all his patience and encouragement.

Special thanks to Jon Elmquist Schmidt and Oliver Hvarre for reading my thesis, giving me great feedback and helping me adjust it to the better.

Last, but not least, I would like to thank my parents and my best friend for supporting and encouraging me to engage in a thesis which seemed very hard to me, but ended up being the best choice.

Contents

List of Figures

1

Introduction

Similarity search represents one of the fundamental topics in computer science, with applications in areas such as machine learning [14], clustering [27], data mining [44], information retrieval [22], computer vision [7], pattern recognition [17], and so forth. Also known as the Nearest-Neighbor Search problem (*NNS*), similarity search becomes relevant in many contexts such as edge prediction (e.g. recommendation systems based on collaborative filtering), pattern mining or substructure discovery (e.g. subgraphs of a graph can reveal valuable information), classification (classifying a data point depending on its neighborhood), scene recognition, image segmentation etc.

1.1 Motivation

Similarity search is performed on large amounts of data represented in high-dimensional space. There are two main aspects which deepen the need of improving NNS: the large dimensionality and the big volume of data.

Data representation. Usually, data is stored and represented as numerical vectors [20]. Mapping from data points to vectors of real numbers is defined as *an embedding*. Generally, an embedding represents a mapping from a metric space to another, by still preserving pairwise distances among data

points to some extent [6].

Content-based retrieval. NNS has been 'extensively [...] applied in many fields, such as content-based multimedia retrieval' [47]. 'Content-based' search focuses on analyzing items by their content (e.g. shape, color, texture etc.) rather than by their metadata (e.g. attached tags, keywords, descriptions etc.) and becomes advantageous because it does not rely on annotation quality. Moreover, metadata is highly subjective and does not always effectively describe objects, which makes metadata-based search even more questionable.

1.1.1 Content-based retrieval challenges

Nowadays, the most dominating data types are complex, feature-rich media objects such as videos or images. Searching through such items by their content instead of metadata poses a much more considerable challenge.

1.1.1.1 High-dimensionality

The more complex, feature-rich and noisy the data, the more challenging content-based similarity search becomes. For instance, two images exposing the same scene can lead to two different bit-level representations, which makes visual similarity retrieval more difficult. Search performance generally degrades as dimensionality increases [44], a phenomenon known as *the curse of dimensionality*.

The curse of dimensionality. Initially discussed by Richard Bellman [11], *the curse of dimensionality* represents a phenomenon which occurs when increasing the dimensions of our data. The bigger the dimensionality, the more data we need to actually represent the space. For instance, in the attempt of

transitioning from a set of 10 1-dimensional data to a 2-dimensional space, the amount of data required in order to fill the new space is actually much bigger (i.e. 10^2). This effect complicates NNS, since both space and running time grow exponentially with the number of dimensions [21].

1.1.1.2 Big data volume

Large amounts of data threaten both the processing capacity and performance of a content-based retrieval system, since they introduce more complexity. However, improving algorithms to the extent where they can handle a big volume of data has proven to optimize the speed in many applications (e.g. computer vision) [29].

Additionally, we can relate this to the high-dimensionality as well. As previously mentioned, the more dimensions we work with, the more data we need. For instance, a large dataset is beneficial for the good performance of a learning algorithm [39], but for this, the algorithm is firstly required to support big volume of data.

1.2 Problem Definition

NNS. Given a dataset D of n d-dimensional data points and a dissimilarity measure M, construct a data structure which, given any query point q, finds the point p in D nearest to q [34], where the notion of nearness refers to similarity and thus, depends on M.

The notion of similarity. The similarity between two data points depends on several aspects and it should be context-particularized [48]. In this thesis, the embedding used as a measure of object similarity is Euclidean.

The embedding in use influences the performance of the application. The geometric relationship between two points x_i and x_j provides the actual distance, while simultaneously expressing the similarity between them. For instance, the nearness between two images is defined by the distance between their representative vectors (e.g. image descriptors) in the chosen embedding space, which for us, is Euclidean.

k-NN. One of the most conventional variations of the Nearest-Neighbor Search problem is k-NN, which refers to the task of identifying the set of k closest data points for a given query point. This thesis focuses particularly on k-NN rather than the traditional NNS.

1.3 k-NN Extensions

k-NN methods can be classified as *exact* and *approximate*.

1.3.1 Exact k-NN

Given a query point q, exact k-NN retrieves the precise set of k nearest neighbors of q. Even if this is easily applicable on small datasets, it becomes cumbersome for large datasets of high dimensionality [34]. Falling prey to the curse of dimensionality, exact k-NN scales poorly even when the number of dimensions goes above $d \in (10, 20)$ [17], therefore becoming expensive in terms of time and space requirements.

1.3.2 Approximate k-NN

Considering the complexity of exact k-NN methods, approximate k-NN relies on the premise that 'in many cases it is not necessary to insist on the exact answer; instead, determining an approximate answer should suffice' [17]. This introduces the idea of trading search accuracy for reduced query time, but at the same time not providing any guarantees to pursue the precise nearest neighbors. In fact, Approximate Nearest-Neighbor Search (*ANNS*) can be almost as effective as exact NNS, supported by the fact that finding sufficiently many nearest neighbors to a given query point can be desirable in many setups where obtaining a much quicker approximation is more important than spending more time waiting for an exact answer.

1.4 Hashing for Similarity Search

Considerable efforts have been devoted to Approximate Nearest Neighbor Search in recent years. One of the well-known solutions for performing ANNS is hashing.

The concept of hashing. Generally, hashing consists of 'transforming the data item to a low dimensional representation, or equivalently a short code consisting of a sequence of bits' [43]. A hash function is effective if it maps inputs to outputs uniformly, since the hashing cost scales up with the number of collisions. Inputs are distributed across a series of *buckets*, where by *bucket*, we refer to a low dimensional representation b_i assigned to the inputs which can be categorized under this label depending on certain criteria (i.e. all the inputs which collide in bucket b_i).

Hashing in the context of similarity search. Unlike common hashing,

where collisions are undesirable, hashing in the context of similarity search is designed to generate collisions for similar data points. The idea is to hash nearest neighbors to the same bucket, while dissimilar items should be placed in further buckets, such that pursuing nearest neighbors for a query point can be easily performed by returning the points in the associated bucket.

The locality-sensitive property. ANNS exploits the property according to which items alike are hashed to the same bucket with a higher probability than dissimilar items, which we refer to as *the locality-sensitive property* [43]. Moreover, if the used encoding technique properly preserves pairwise distances between data points when mapping to the more compact representation of the given high-dimensional space, retrieving nearest neighbors based on the determined hashcodes should provide good approximations in terms of similarity.

Data-independent versus data-dependent hashing methods. Encoding techniques can be divided in two broad categories: data-independent, designing hash functions which do not look at data distribution, such as Locality Sensitive Hashing (LSH) [37], and data-dependent, defining hash functions which take into account structure of the data and follow a learning/training phase on their way towards hashing, such as Spherical Hashing (Sph) [19], Spectral Hashing (SH) [46] etc.

2

The Algorithm

This chapter sheds light on how Spectral Hashing (SH) works at a high level. We refer to the original algorithm as *Vanilla SH* throughout the thesis.

Vanilla SH - a learning algorithm. Among all the hashing approaches addressed so far in the context of similarity search, machine learning solutions seem to be more successful than data-independent methods (e.g. LSH) [46]. Essentially, there has been much interest in training techniques and Vanilla SH also adopts a learning approach.

The ML approach. In a machine learning approach, the performance of an algorithm is evaluated by creating two different datasets from the original data, namely a *training set* and a *testing set*.

Shortly, the algorithm *learns* from the data and makes predictions on it. The training set is a set of n data points used in the learning/training phase in order to fit a *model*. Depending on the context, a model can take different forms, such as a mathematical equation, a neural network, a classifier etc. Its role is to describe the training set (e.g. data distribution). Under the assumption that two datasets sampled from the same probability distribution follow similar patterns, the testing set is finally used to evaluate how well the model approximates properties of the training set.

Vanilla SH content-specific phases, briefly. At a high level, Vanilla SH consists of the following main phases:

1. Identify the k axes of largest variance over the training set. These are referenced as *principal components* or *eigenvectors*;
2. Determine the k best split points over the k principal components and store them according to how well they partition the training set (i.e. a split point dividing between two well-defined clusters is considered a good partition, while one dividing across both clusters is considered a less fortunate partition). The k best split points correspond to what we define later on as the k smallest non-zero *eigenvalues*;
3. Use k functions to divide both the training and the testing set according to the set of k previously learned eigenvalues. These are referred to as *eigenfunctions*;
4. Threshold the k eigenfunctions at 0 to generate k-length hashcodes: positive values are assigned a bit of 1, whereas non-positive values are assigned a bit of 0.

Vanilla SH ML-specific phases. Seen from a machine learning perspective, the algorithm can be divided into the following:

1. Training Phase (corresponding to 1 and 2).
2. Compression Phase (corresponding to 3 and 4).

Data normalization. As a pre-processing step, normalization of both the training and the testing set is required in order to assign equal significance to each of the d dimensions in the original space. In other words, the algorithm is fed d-dimensional data, which is standardized with min-max normalization,

such that values in each dimension range in the interval $[0, 1]$. This provides data consistency and assigns equal importance to each and every dimension in the dataset.

2.1 Training Phase

The model in Vanilla SH. In this algorithm, the model represents a set of k principal component axes defined over training data, together with additional parameters describing these, such as their ranges and the set of k split points where partitioning is optimal, based on theory.

Fitting the model on the training set is equivalent to *learning* properties of the data which we use later on in the compression phase, where we generate hashcodes. As input to the training phase, we pass the training set and the binary code length, defined as k. The k principal components we select as part of the model represent the k largest axes over the training data. We identify these as being the eigenvectors obtained from the covariance matrix of the training set.

In other words, we change the basis to a better representation of the data by restricting it to the k largest axes because they preserve most of the information from the original space. During compression, both the training and the testing data get projected on the chosen principal components. The benefit of selecting the basis of largest variation consists of maintaining a minimal projection error when the original space is being projected onto the new space. For instance, it is best to project the data represented in Figure 2.1 on the large direction rather than on the small one, as the projection becomes dependent on the length of the small axis and thus, is minimal.

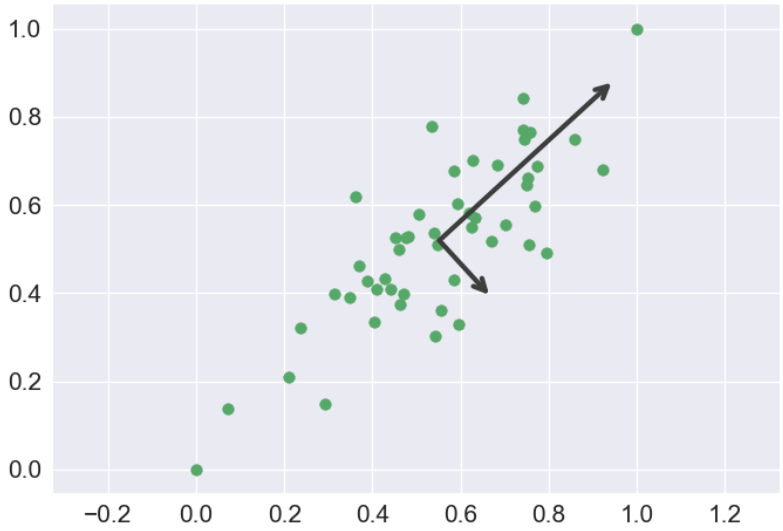


Figure 2.1: Principal component axes over 2-dimensional data. The large axis is optimal for data projection, as it preserves most of the information in the original data.

Fitting a multidimensional rectangle distribution to the data. The next step during the training phase achieves a rectangular approximation along each individual principal component. Strictly speaking, Vanilla SH relies on the assumption that each newly selected axis consists of uniformly distributed data. The algorithm prolongs each principal component range with its fifth percentile value in both ends (i.e. each axis becomes 10% longer). The intention behind extending each axis relies on generalizing the selected basis in order to ensure a good projection of the data, regardless which kind of data is given as input to projection (i.e. training or testing data). In other words, principal component scores (values of the projected data) can 'fall' outside the original unextended axes. This occurs when choosing these axes over a dataset (i.e. the training data), but instead, projecting another dataset over them (i.e. the testing data). As such, we store the extended principal components' ranges in the model and use it constantly later on, in the compression phase, regardless which dataset we pass for encoding.

Equation of an eigenfunction. The original SH paper [46] defines a single-dimension analytical eigenfunction ϕ_i, as follows:

$$\phi_i(x) = \sin\left(\frac{\pi}{2} + \frac{m\pi}{b-a}x\right) \tag{2.1}$$

Equation 2.1 introduces parameters which are referenced as such:

1. x - principal component score: the score/projected value of a given data point on the i^{th} principal component;
2. m - mode: a transposed vector whose non-zero entry (referred to as *mode value* m_i) expresses how many times the single-dimension analytical eigenfunction ϕ_i intersects the principal component axis PC_i;
3. b, a - maximum, minimum value of the i^{th} principal component.

All these variables contribute to the formation of the sine function on the i^{th} principal component axis, with the intent of partitioning scores across it later on, during compression. For instance, sine frequency is described by the multiplication $\frac{m\pi}{b-a}$ in Equation 2.1. If the i^{th} principal component axis spans within the interval $\left[0, \frac{\pi}{5}\right]$, also knowing that the sine function starts from a value of 1 on the y-axis (since the eigenfunction adds the constant $\frac{\pi}{2}$ to the sine input) and having a mode value $m_i = 2$, then the sine function crosses the x-axis ten times within the interval $[0, \pi]$, as shown in Figure 2.2 (see the points marked with red). However, out of these ten intersections with the x-axis, only two are contained in our principal component interval axis, which is exactly the value of our mode m_i. This frequency scales with the mode value m_i, as well as with a decrease in the principal component range. In other words, an increase in mode m_i by an amplitude amp_1, along with a decrease of the range $(b-a)$ by amplitude amp_2, together describe a

sine function which is $(amp_1 * amp_2)$ times more frequent than previously (it intersects x-axis $(amp_1 * amp_2)$ more times than before).

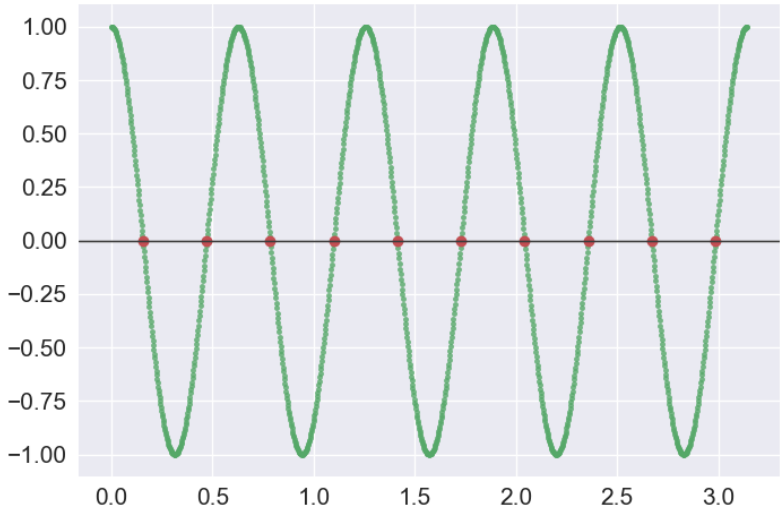


Figure 2.2: Sine curve described by Equation 2.1, where the plot shows the sine frequency over the interval $[0, \pi]$. The principal component axis ranges in the interval $\left[0, \frac{\pi}{5}\right]$. The eigenfunction parameters are: $a = 0, b = \frac{\pi}{5}, m = 2$. Intersection points with the x-axis are marked with red.

Components of the model in Vanilla SH. The training phase aims to store a model which includes the following:

1. Minimum and maximum values (a and b) for each of the k selected principal components;
2. The newly chosen basis (i.e. the principal components);
3. The variable $\omega_0 = \frac{\pi}{b-a}$, for each of the k principal components;
4. k modes - one mode for each of the k principal components.

As soon as the model is stored, the training phase ends, being followed by the compression phase.

2.2 Compression Phase

The compression phase aims to map the original space (i.e. Euclidean) to Hamming space, therefore obtaining binary codes for all the points in a given dataset.

Compression steps, briefly. Compression uses the components of the model stored during training (see section 2.1). Given a dataset X, it goes through the following steps:

1. Normalize X and project it onto the k eigenvectors obtained from training (the k principal components stored by the model). Each eigenvector direction describes a rectangle, with both positive and negative scores;
2. Center each of the k principal component axes, such that values end up only being positive;
3. For each i^{th} bit of the k desired bits, select the corresponding parameters to create a single-dimension analytical eigenfunction ϕ_i, as stated in Equation 2.1. Furthermore, threshold each obtained eigenfunction at 0, in order to obtain bits: assign positive outputs a bit of 1 and non-positive outputs a bit of 0.
4. Concatenate all the bits resulted from item 3 to obtain the final hashcodes.

Binary code length in relationship with the number of principal components. In the setup proposed by Vanilla SH, the number of bits used for encoding (defined as k) equals the number of principal components stored in the model after training, unless the dimensionality of the original dataset (defined as d) is smaller than k, in which case the number of principal components equals d.

Bits derived from principal components. Not all the principal components provide bit contribution to the final binary codes. Depending on the structure of the given dataset, while some of the principal components attach p bits to the final hashcode, where $p \leqslant k$, others might actually not contribute with any bit at all. Knowing that a principal component PC_i can be associated with several modes, this depends on the maximum mode value $\max_{m_i}$ (see item 2 in section 2.1) of each principal component, as it indicates the number of bits derived from PC_i. It makes sense that a principal component PC_j with maximum mode value $\max_{m_j} = 0$ is not taken into account for bit extraction, since the eigenfunction built from it never intersects the x-axis (i.e. hypothetically, any bit b_j extracted from it has the same value, regardless the data point it represents; therefore, PC_j is not an axis which adds meaning to our final hashcodes).

Number of divisions across the same principal component. Suppose we have a principal component PC_i, associated with a sequence of modes of mode values stored in $S_{\text{mode\_values}}$. Each mode value of mode m_j in $S_{\text{mode\_values}}$ corresponds to an individual eigenfunction and indicates the number of intersections between sine and x-axis (i.e. the number of divisions across PC_i described by mode m_j alone). However, the maximum number of divisions across PC_i is given by the sum of all mode values in $S_{\text{mode\_values}}$, as follows:

$$\max_{num\_divisions(PC_i)} = \sum S_{\text{mode\_values}} = \frac{\max_{S_{\text{mode\_values}}}(\max_{S_{\text{mode\_values}}} + 1)}{2} \quad (2.2)$$

The reason why we state it as a *maximum* number of divisions is because some of the divisions might interfere and split in the exact same points, which leads, in fact, to a smaller number of divisions across the principal component.

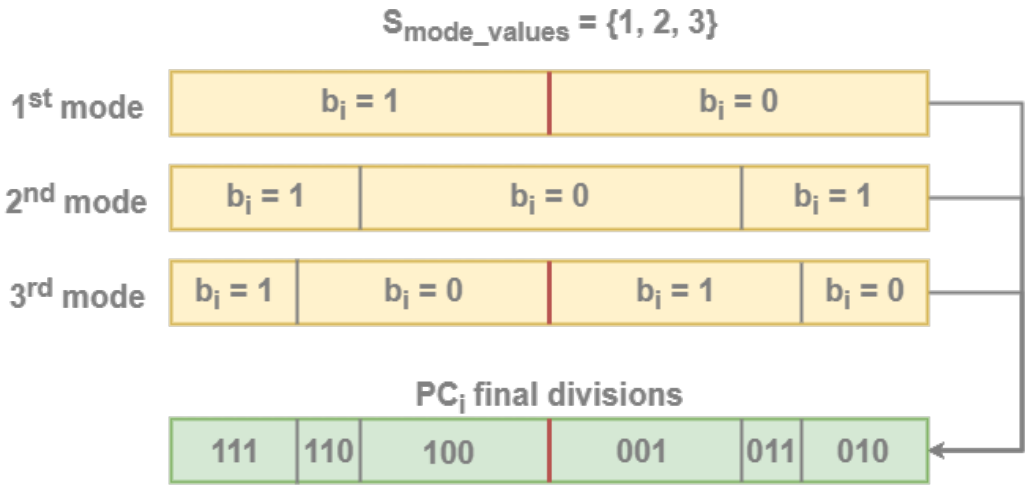


Figure 2.3: All divisions on a given principal component PC_i, where maximum mode value is $\max_{S_{\text{mode\_values}}} = 3$ of a mode values set $S_{\text{mode\_values}} = \{1, 2, 3\}$. This leads to three individual partitions on the same rectangle, each corresponding to one of the mode values in $S_{\text{mode\_values}}$. The vertical divisions are points where bits change. For instance, the line dividing PC_i by the first mode value splits the axis into points of bit $b_i = 1$ on the left side and points of bit $b_i = 0$ on the right side. The red and slightly thicker lines coincide.

Figure 2.3 illustrates the idea of dividing PC_i with a number of sine functions equal to a maximum mode $\max_{S_{\text{mode\_values}}} = 3$, where the mode values are $S_{\text{mode\_values}} = \{1, 2, 3\}$. After combining the divisions made by all the mode values, we obtain a number of divisions smaller (namely 5) than the maximum possible ($\max_{num\_divisions(PC_i)} = \frac{3*4}{2} = 6$), since the one division in the first mode coincides with the middle division in the third mode.

That being said, the compression phase ends with a k-bit hashcode generated for each item in the given dataset.

3

The Underlying Theory

The aim of this chapter is to provide, through selective literature references, a clearer perspective of the theoretical concepts behind Vanilla SH. We review the conditions of what makes an efficient binary code, along with how the algorithm theoretically intends to achieve this. Moreover, the chapter emphasizes the relationship between what it means to efficiently encode a dataset and graph partitioning. Spectral relaxation is outlined as a solution to the problem, proposing a subset of thresholded eigenvectors obtained from the graph *Laplacian*. The remainder of the chapter is devoted to showing how to compute the code representation of novel data points, as well as to how this relies on the distinction between an eigenvector and an eigenfunction.

3.1 What Makes a Good Binary Code

In order for a binary code to be effective, it needs to fulfill the following requirements [46]:

1. It is easily generated for novel data points;
2. It can encode an entire dataset with a compact representation (short number of bits);

3. It preserves the pairwise distances between points relatively well when mapping from a continuous space (Euclidean) to a discrete one (Hamming).

3.2 Theoretical Constraints in Vanilla SH

Constraints leading to balanced partitioning of the data. Similarly to the constraints Spherical Hashing relies on, in order to make a good binary code [19], Vanilla SH formulates the following: each bit fires with a probability of 50%, while there exists pairwise independence between hashing functions. In other words, each bit achieves a division of the dataset in relatively equal parts. The independence constraint confirms that, for any two bits i and j, or for that matter, for any two hashing functions i and j, their intersection results into four equal divisions, each containing one fourth of the items in the dataset. Therefore, if these constraints are applied simultaneously and across all pairs of hashing functions, the result is a balanced distribution of data points for each hashing function.

Preliminary definitions. Before elaborating over these constraints, we define n as the size of the dataset subject to encoding. The set of n hashcodes representing the given dataset is referenced as $\{y_i\}_{i=1}^{n}$, while the affinity matrix over the given dataset is referred to as $W_{n\times n}$. The parameter k always refers to the chosen binary code length.

The affinity matrix defined. The affinity matrix is often used in the context of dimensionality reduction techniques. The entry W_{ij} in our $W_{n\times n}$ matrix defines a measure of the nearness/similarity between two items x_i and x_j and can generally be expressed in terms of the Euclidean distance (though this is not necessarily a requirement). The value W_{ij} is defined by the following heat kernel [46]:

$$W(i,j) = exp(-\left|x_i - x_j\right|^2 / \epsilon^2) \tag{3.1}$$

As explained in 'Understanding Complex Datasets: Data Mining with Matrix Decompositions' [36], the affinity matrix captures local nearness of the items in a dataset. Any two points x_j and x_j are considered to be near neighbors if the Euclidean distance between them is smaller than ϵ, a scaling parameter which tells how quickly x_i influences neighbors of type x_j (see chapter 4, [36]).

The objective function. In Vanilla SH, the locality-sensitive property (see chapter 1) relies on minimizing the average Hamming distance between similar points. This idea is expressed by the following objective function and its constraints:

$$\begin{aligned} \textit{minimize} : \quad & \sum_{ij} W_{ij} \left|y_i - y_j\right|^2 \\ \textit{subject to} : \quad & y_i \in \{-1, 1\}^k \\ & \sum_i y_i = 0 \\ & \frac{1}{n} \sum_i y_i y_i^T = I. \end{aligned} \tag{3.2}$$

The minimization function in equation Equation 3.2 defines the average Hamming distance between similar items. This is expressed through the sum of all pairwise distances in Hamming space, where the distance $d_{Hamm_{ij}} = \left|y_i - y_j\right|^2$ between any two k-length binary vectors y_i and y_j is assigned a different weight/importance, depending on the affinity value W_{ij}. However, W_{ij} is a measure of the Euclidean distance between items x_i and x_j, where x_i and x_j are the data points in Euclidean space which correspond to the binary vectors/hashcodes y_i and y_j. In other words, W_{ij} represents the weight of how

important it is that items y_i and y_j are mapped close together in Hamming space. If W_{ij} is small, that means x_i and x_j are most likely not close to each other originally and therefore, the weight of importance must reflect that in order to map the same relationship in Hamming space between y_i and y_j. In the opposite case, W_{ij} is bigger when it becomes important to preserve the distance $d_{Eucl_{ij}} = |x_i - x_j|^2$ more. Figure 3.1 illustrates the concept: the closer the items x_i and x_j, the bigger the value of W_{ij}, meaning the more important it becomes to approximate the two items as being near neighbors. As a consequence, a less accurate mapping of $d_{Eucl_{ij}}$ with $d_{Hamm_{ij}}$ is penalized more by the value of W_{ij}.

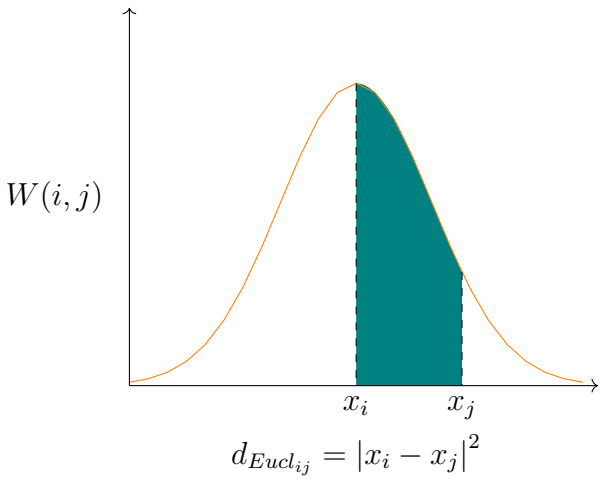


Figure 3.1: The nearer the two points x_i and x_j are in Euclidean space, the bigger the weight for mapping x_i and x_j closer. The y-axis shows the weight value W_{ij} and how it relates to the Euclidean distance between points, indicated on the x-axis.

The balance constraint in the objective function. The first constraint associated with the objective function (i.e. $\sum_i y_i = 0$) states that each bit is turned on with a probability of 50%.

The uncorrelatedness constraint in the objective function. The other constraint (i.e. $\frac{1}{n}\sum_i y_i y_i^T = I$) requires bits in the binary code to be uncorrelated. Previously, the definition of a good binary code stated pairwise bit independence as one of the requirements. However, the initial constraint is being relaxed now, since uncorrelated is a less strict condition than independent. We elaborate more on the distinction between the two and emphasize some of the implications between them, as follows.

The notions of uncorrelatedness, orthogonality and linear independence. In linear algebra, the concepts of uncorrelatedness, orthogonality and linear independence all point to a lack of relationship between variables, but are, in fact, fundamentally different. Note that we use the terms *orthogonal* and *perpendicular* interchangeably, where they indicate vectors that form a 90° angle.

Let us consider two vectors v_i and v_j and define the three concepts from an algebraical point of view:

1. v_i and v_j are linearly independent if there does not exist a constant c such that $c \cdot v_i = v_j$;
2. v_i and v_j are orthogonal if their dot product equals 0: $v_i \cdot v_j = 0$;
3. v_i and v_j are uncorrelated if their covariance is 0: $Cov[v_i v_j] = \sigma_{v_i v_j} = 0$.

From a geometrical perspective, considering v_i and v_j two n-dimensional vectors, the three concepts are defined as follows:

1. v_i and v_j are linearly independent if their axes do not coincide;
2. v_i and v_j are orthogonal if their axes are perpendicular;
3. v_i and v_j are uncorrelated if once centered, they become orthogonal.

Unlike orthogonality and linear independence, which tell something about variables v_i and v_j themselves, uncorrelatedness tells something about the centered variables v_i and v_j (variables v_i and v_j after substracting the corresponding mean from each).

Implications among uncorrelatedness, orthogonality and linear independence. There are several implications among these three concepts:

1. Two variables can be orthogonal, but not uncorrelated, in case they originally form a 90° angle, but not anymore after being centered;
2. Two variables can be uncorrelated, but not orthogonal, in case they are not originally perpendicular, but become so after centering them;
3. Two variables are both uncorrelated and orthogonal if centering does not change the angle they describe;
4. Orthogonality is a particular case of linear independence;
5. If v_i and v_j are independent, it implies they are uncorrelated, but the opposite implication is not necessarily true, because two uncorrelated variables can still depend on each other.

More on the uncorrelatedness constraint in the objective function. Coming back to the constraint $\frac{1}{n}\sum_i y_i y_i^T = I$ from Equation 3.2, the uncorrelatedness requirement can be explained by the following reasoning: the non-diagonal entries of $\frac{1}{n}\sum_i y_i y_i^T$, on any position of indices (r, c), where $r \neq c$, equal 0 on average when dimensions r and c are not correlated. That means $y_i(r)y_i^T(c)$ and $y_j(r)y_j^T(c)$ cannot both be 1-bits (represented as '1' in the objective function) or 0-bits (represented as '-1' in the objective function). Keeping in mind the first constraint (i.e. $\sum_i y_i = 0$), where $y_i \in \{-1, 1\}^k$, the bits from binary vectors y_i and y_i^T which land on non-diagonal positions

in $y_i y_i^T$ must be different, so they can yield zeros on average and lead to the identity matrix. However, even if distinct, there can still exist a dependence between them, which means precisely that they are uncorrelated, and not independent. The diagonal entries of $\frac{1}{n}\sum_i y_i y_i^T$ always add up to ones, as for any diagonal entry (r, c), where $r = c$, we multiply bits corresponding to the same dimension (i.e. the same bits multiplied with themselves occupy the diagonal entries in the identity matrix I).

The relationship with graph partitioning. As referenced in the original *SH* paper [46], the attempt of minimizing the value of the objective function (see Equation 3.2) corresponds to solving a balanced graph-partitioning problem. Drawing the parallel with the graph-partitioning problem is equivalent to mapping the data points to a set of graph vertices $V(G)$, such that the heat kernel value $W(i, j)$ determines the cost/weight between item x_i and item x_j. Considering the previous constraints, a binary code of length 1 partitions the graph in two equal subsets of vertices, where a subset is described by codeword $y_i = 1$, and the other one by codeword $y_j = 0$. However, the same constraints must be fulfilled by each and every bit in the hashcodes. Unfortunately, as the authors of 'A Tutorial on Spectral Clustering' [28] also explain, the balancing requirement makes this minimization(min-cut) problem NP-hard.

3.3 The Theory at the Basis of Vanilla SH

Before introducing the solution proposed in the original SH paper [46] for our problem, it is necessary to give a brief account of the main theoretical concepts Vanilla SH uses. This includes the following: linear transformations, definition of both an eigenvector and an eigenvalue, the covariance matrix, PCA, dimensionality reduction and manifold learning, Laplacian eigenmaps

and spectral graph theory.

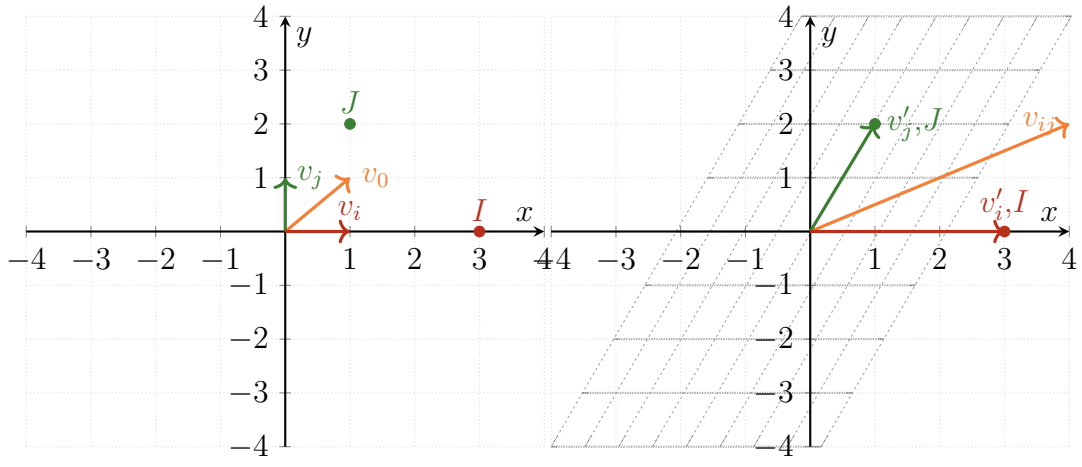
3.3.1 Linear Transformations

Square matrices and their characteristic eigenvectors. In linear algebra, every square matrix has eigenvectors. We aim to discuss linear transformations because they are a convenient tool for defining eigenvectors. Applying a linear transformation M_{tr} defined by equation $M_{tr} \cdot v_0 = v_{ij}$ to vector v_0 is equivalent to transitioning v_0 with a force M_{tr} until it turns into v_{ij}.

Linear transformation example. Figure 3.2 provides an example. Consider point $O(1, 1)$ on the left. O is represented by vector v_0 (i.e. $v_0 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$). However, v_0 can also be written as the dot product between the identity matrix I and itself. That means we apply the identity transformation to it and it remains unchanged:

$$v_0 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \cdot v_0$$

Vectors v_i and v_j in Figure 3.2a are columns in the identity matrix $I_2 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$. For this reason, they are called *basis vectors* and represent the coordinate system shown in Figure 3.2a.



(a) (b)

Figure 3.2: The transformation is done with matrix $M_{tr} = \begin{bmatrix} 3 & 1 \\ 0 & 2 \end{bmatrix}$. After the transformation, the input vector $v_0 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$ turns into the output vector $v_{ij} = \begin{bmatrix} 4 \\ 2 \end{bmatrix}$.

If generalizing the identity transformation I_2 to a transformation M_{tr}, vector v_0 from a coordinate system like the one in Figure 3.2a becomes vector v_{ij} in a coordinate system like the one in Figure 3.2b. After applying a linear transformation to vector v_0 with matrix M_{tr}, the new basis vectors are $v'_i = \begin{bmatrix} 3 \\ 0 \end{bmatrix}$ and $v'_j = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$. Vectors v'_i and v'_j are, in fact, the initial vectors v_i and v_j in a new basis, which linearly transitioned up to points I and J. In linear algebra, such a transformation is defined as *change of basis* [2]. This is relevant for us because Vanilla SH applies a change of basis to the input dataset, throughout PCA, which we introduce later on.

3.3.2 Eigenvectors

Definition of an eigenvector. After applying the linear transformation in Figure 3.2 to vector v_0 and the unit vectors v_i and v_j, they transition from the state in Figure 3.2a to the state in Figure 3.2b. Among these, all change direction except for v_i. For this reason, v_i together with other vectors of the same behavior are called eigenvectors. As such, we define an eigenvector as any vector v_k which responds to a matrix transformation M_{tr} as if the matrix itself is a scalar coefficient, because v_k already points in the direction towards which M_{tr} moves. Consequently, the only effect M_{tr} exerts on vector v_k is either stretching or compressing it, but never moving it from its initial axis.

Properties. We identify some useful eigenvector-related properties:

1. A square matrix $M_{n\times n}$ can have as many unique eigenvectors as it has columns/dimensions;
2. Eigenvectors exist only for square matrices;
3. The eigenvectors of a matrix are orthogonal with respect to each other [38].

Motivation. Eigenvectors become very advantageous in matrix decomposition (i.e. representing a matrix by a product of other matrices). In particular, they come into play when using matrix diagonalization. This means an initial square matrix $M_{n\times n}$ is turned into a diagonal matrix (i.e. a matrix only with zero values, except on the diagonal entries) which maintains the underlying properties of $M_{n\times n}$. In fact, matrix diagonalization is equivalent to *changing the basis* (see subsection 3.3.1) and selecting a better one, composed of a set of eigenvectors. Therefore, we consider the diagonal matrix our transformation matrix with respect to the new basis of eigenvectors, which make for an interesting and simpler coordinate system (i.e. basis).

3.3.3 Eigenvalues

Definition of an eigenvalue. Each eigenvector has an associated eigenvalue. λ_k is eigenvalue for eigenvector v_k if it is a scalar representing the amount of expansion/compression of vector v_k on its axis. Briefly, an eigenvalue is the factor by which its corresponding eigenvector is being stretched or compressed during a linear transformation.

Given this definition, the linear transformation shown in Figure 3.2 can be expressed in terms of eigenvector v_i as such:

$$M_{tr} \cdot v_i = \lambda_i \cdot v_i \Leftrightarrow \begin{bmatrix} 3 & 1 \\ 0 & 2 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \lambda_i \cdot \begin{bmatrix} 1 \\ 0 \end{bmatrix} \Leftrightarrow \begin{bmatrix} 3 \\ 0 \end{bmatrix} = \lambda_i \cdot \begin{bmatrix} 1 \\ 0 \end{bmatrix} \Rightarrow \lambda_i = 3$$

The eigenvalue as a scalar. The sign of the eigenvalue announces how the eigenvector moves along its axis during the linear transformation. For instance, in Figure 3.2, having the eigenvalue $\lambda_i = 3$, eigenvector v_i expands in the same direction which it initially points to and becomes three times as big in magnitude. With a negative λ_i, for example $\lambda_{neg_1} = -3$, v_i flips. If $\lambda_i \in [0, 1]$, for instance $\lambda_{neg_2} = \frac{1}{2}$, v_i gets squashed by a factor of λ_{neg_2}.

3.3.4 The Covariance Matrix

Definition. Eigenvectors derive from a square matrix $Cov_{n\times n}$, which we define as the *covariance matrix*. Its diagonal entries $Cov_{n\times n}(i, i)$ indicate the variation in the dimensions of type i, while the non-diagonal entries $Cov_{n\times n}(i, j)$ indicate the covariance between dimensions i and j (i.e. the linear relationship established between dimensions i and j or how much they vary from the mean with respect to each other). If lower values in dimension i correspond to higher values in dimension j, meaning that i and j behave

contrastingly, that implies negative covariance (i.e. $Cov_{n\times n}(i,j) < 0$). If the bigger values in dimension i generally tend to correlate with the bigger values in dimension j, covariance is positive (i.e. $Cov_{n\times n}(i,j) > 0$).

Properties. We identify the following useful properties:

1. The covariance matrix is always symmetric;

2. Like the *eigenpairs* (the eigenvectors and their associated eigenvalues), the covariance matrix describes how data is shaped. Finding the one equals finding the other, as the two are mathematically connected. Entries in the covariance matrix indicate data orientation (covariance) on off-diagonals and data distribution (variance) on diagonals;

3. The covariance matrix can be seen as a linear operator applied on data [41]. As shown in subsection 3.3.1, any linear transformation (e.g. $Cov_{n\times n}$) can be exclusively expressed in terms of any of its eigenpairs (v_{eig}, λ_{eig}):

$$Cov_{n\times n} \cdot v_{eig} = \lambda_{eig} \cdot v_{eig} \tag{3.3}$$

   Thus, we can represent $Cov_{n\times n}$ in terms of an eigenvector v_{eig} and its magnitude λ_{eig};

4. If the covariance matrix $Cov_{n\times n}$ has zero off-diagonal values, the diagonal contains eigenvalues indicating data spread on each dimension;

5. Writing the covariance matrix in terms of its eigenvectors is called *eigendecomposition of the covariance matrix* and can be driven from Equation 3.3. If we consider V a matrix gathering the set of all eigenvectors and L a matrix gathering the set of all eigenvalues on the diagonal, we have the following:

$$Cov_{n\times n} \cdot V = L \cdot V \tag{3.4}$$

From Equation 3.4, we can directly conclude the formula of the covariance matrix in terms of all its eigenpairs:

$$Cov_{n\times n} = V \cdot L \cdot V^{-1} \tag{3.5}$$

Equation 3.5 explains how eigenvectors mathematically originate from the covariance matrix.

3.3.5 Dimensionality Reduction by Manifold Learning

Definition of dimensionality reduction. Given a set of n d-dimensional data points $X_n = \{x_1, x_2, ..., x_n\}$, the problem of dimensionality reduction is equivalent to finding a set of n k-dimensional data points $Y_n = \{y_1, y_2, ..., y_n\}$, such that $k \ll d$ and Y_n represents X_n by maintaining its underlying properties (i.e. preserves pairwise distances or has locality preserving properties).

Definition of manifold learning. Vanilla SH is a learning algorithm which reduces the dimensionality of a d-dimensional dataset X_n by identifying a *manifold*. We define the manifold as a space over X_n which is locally Euclidean, but globally exhibits a more complex structure (i.e. a sphere, an S-curve, a torus etc.).

Along these lines, we perceive SH as an algorithm that learns a manifold, but one of a much simpler structure, namely linear. This idea relies on the fact that SH performs PCA, thus finding a set of linear axes which we think of as a manifold by itself.

Manifold learning targets a broad spectrum of fields, such as machine learning, pattern recognition, data compression or database navigation [16]. Generally, manifold learning methods rely on the assumption that high-dimensional data lies, in fact, in a lower-dimensional format, which can be learned and represents the data much more concisely, without much noise and

redundancy across dimensions. Such a data shape allows for a geometrical dimensionality reduction, which works well with linear algebra.

Manifold learning in the context of Vanilla SH. Manifold learning plays a significant role in the context of Vanilla SH because by learning the underlying structure of the data and creating a model, the algorithm is able to hash data points unavailable during training and integrate them in the learned embedding, a procedure referenced in the original SH paper as *out-of-sample extension* [46].

Classification of dimensionality reduction methods. Dimensionality reduction methods can be classified as linear (e.g. Principal Component Analysis/PCA, Linear Discriminant Analysis/LDA) or non-linear (e.g. Locally Linear Embedding/LLE, kernel PCA, Isomap). While the former only identifies linear manifolds, the latter can learn more complex manifold morphologies. Vanilla SH reduces dimensionality linearly, by using PCA.

3.3.6 Principal Component Analysis

Definition. As one of the underlying mechanisms of Vanilla SH, *Principal Component Analysis* (*PCA*) is a dimensionality reduction technique used for pattern mining, preprocessing or data compression [38].

Motivation. Given a dataset X_n, the typical motivation behind PCA is to reduce a set of dimensions (a given basis/coordinate system) to a smaller set of new dimensions, which are linear combinations of the initial ones and preserve most of the information in X_n (i.e. maximizing the variance of the original data in a lower-dimensional space). We define the newly obtained dimensions as *principal components* and consider them 'the most meaningful basis' [38] to represent the input data. Indeed, this is equivalent to changing the basis (see subsection 3.3.1) of X_n. Data can be expressed in terms of any

axes, but it is more convenient to represent it in terms of orthogonal ones, such as the eigenvectors [38]. Thus, PCA achieves dimensionality reduction by projecting X_n onto a novel feature space (see Figure 3.3 [42]). This is accomplished by decreasingly sorting the eigenpairs and maintaining only the k largest in terms of eigenvectors.

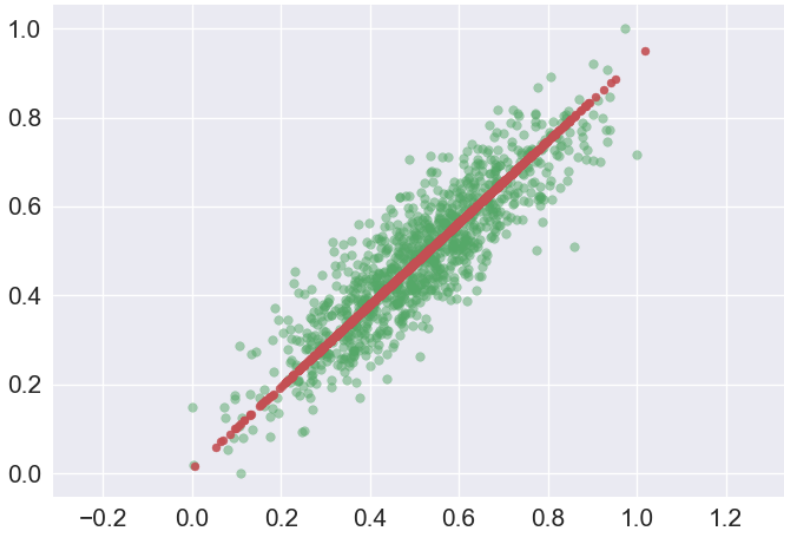


Figure 3.3: Data projection effect from the 2-dimensional data represented by the green points to the 1-dimensional data represented by the red line (i.e. the largest eigenvector).

Being the initial phase of Vanilla SH, PCA becomes useful in our context for several reasons:

1. It provides the best known linear approximation for a high-dimensional space [16];
2. It optimizes space usage and speeds up the process by reducing dimensionality before any other steps in the algorithm;
3. It discards data redundancy and noise, which avoids overfitting during the training phase and feeds data to the algorithm in a convenient form.

3.3.7 Laplacian Eigenmaps

Vanilla SH, an eigenvalue problem with an eigenvector solution, similarly to other algorithms. The eigenvector solution proposed in the original SH paper [46] is precisely the same used in algorithms such as Laplacian Eigenmaps [8] or Spectral Clustering [30]. They all use spectral graph theory, which hints to why the algorithm is called *Spectral* Hashing.

In 'Normalized Cuts and Image Segmentation' [35], Shi and Malik interpret image segmentation as a graph partitioning problem. They underline the idea that dimensionality reduction and clustering are highly correlated. If dimensionality is effectively reduced, such that local neighborhood information is well preserved, the final embedding might outline clusters' formation (in case the dataset has meaningful clusters to begin with). They partition an image optimally with a minimization criterion (i.e. the *normalized cut*), which leads to an eigenvalue problem and therefore, to an eigenvector solution. As such, we can draw a parallel between their approach and what Vanilla SH proposes. Even if attempting to solve two different problems, both approaches rely on spectral graph concepts and use linear algebra and matrix theory to capture properties of the graph a given dataset is mapped to.

3.3.8 Spectral Graph Theory

This subsection sheds light on the spectral graph theory which is closely related to Vanilla SH.

Introducing spectral graph theory. Initially using matrix theory and algebraic methods to discover underlying graph properties (e.g. local closeness, isomorphisms, clusters' formation), nowadays, spectral graph theory moves

towards interpreting theory more visually. The study of graph eigenvalues and especially their geometrical meaning has applications in various fields, such as chemistry, physics, quantum mechanics etc. [13].

Terminology. A dataset X_n can be represented as an undirected weighted graph $G(V, E)$. V indicates the set of vertices, where each vertex maps a data point from X_n. E is the set of edges, where each edge $e(x_i, x_j)$ is positive and maps the connection between two data points x_i and x_j (i.e. it is a measure of the Euclidean distance between them).

The eigenvalue problem. Given a square matrix M_{tr}, the eigenvalue problem is defined as identifying an eigenvector v_{eig} and its associated eigenvalue λ_{eig}, such that applying a linear transformation over v_{eig} is equivalent to transitioning it on its own axis by a factor of λ_{eig}. This is expressed by the equation:

$$M_{tr} \cdot v_{eig} = \lambda_{eig} \cdot v_{eig} \tag{3.6}$$

Operators on V. If we want to repeatedly apply an operator over a vertex/vector $x \in V$, eigenpairs provide a convenient basis for expressing this. If we write x in terms of an eigenvector (i.e. $x = \sum_i c_i \cdot v_{eig}$) and consider Equation 3.6, then repeatedly applying the operator M_{tr} over vertex/vector x can easily be expressed in terms of eigenpairs [40]:

$$M_{tr}^k \cdot x = \sum_i c_i \cdot M_{tr}^k \cdot v_{eig} = \sum_i c_i \cdot \lambda_i^k \cdot v_{eig} \tag{3.7}$$

The adjacency matrix W as an operator. Matrices can encode similarity between items in a given dataset X_n, which provides an effective way of decomposing it. A useful operator associated with $G(V, E)$ is the adjacency operator, corresponding to the adjacency matrix W. As stated in [9], there are several ways of constructing it. Given the heat kernel (see Equation 3.1) in

the original SH paper [46], the affinity matrix $W(i,j) = exp(-|x_i - x_j|^2/\epsilon^2)$ is an ϵ-neighborhoods variation of the adjacency matrix. Even if it provides the advantage of a symmetric relationship [9], it does not underline the properties of $G(V,E)$ as effectively as other operators [40].

The *Laplacian* matrix L as a better operator. As literature explains it (e.g. 'Understanding Complex Datasets: Data Mining with Matrix Decompositions' [36]), given a dataset X_n, the *Laplacian* matrix L achieves a better mapping of similarities. Even though the adjacency matrix W can emphasize useful graph properties (e.g. betweenness centrality), it does not embed all the graph properties which allow Vanilla SH to identify the best graph partitioning, because 'it embeds the graph inside-out' [36]. This means properties such as closeness centrality or clusters' formation might very well be omitted from the matrix representation. This is the reason why in Vanilla SH, the *Laplacian* matrix is a more convenient representation than the adjacency matrix. Instead, the *Laplacian* matrix 'captures the inside-out transformations from graphs to geometric space' [36]. Even by definition, it seems more effective than the adjacency matrix W, since it combines both the adjacency matrix W and matrix D, which we define as *the diagonal matrix*. D is a zeros matrix except the diagonal entries $D(i,i)$, which indicate the degree of vertex x_i (the number of edges incident to it). L is defined as follows:

$$L = D - W \tag{3.8}$$

Properties of L. The *Laplacian* matrix has the following useful properties:

1. It is symmetric (this derives from both D and W being so);
2. It is positive semi-definite, which means it can always be eigendecomposed [5]. Consequently, there exists a matrix X, called *Cholesky factor* [15], such that $L = X \cdot X^T$;

3. All eigenvalues of L are non-negative and real scalars;

4. Its eigenvectors are pairwise orthogonal when their eigenvalues are distinct [5];

5. Its smallest eigenvalue is 0, having the all 1's vector as the corresponding eigenvector [36];

6. The number of eigenvalues which equal 0 is the same as the number of connected components in the underlying graph [36]. This might provide information regarding the number of clusters in the dataset.

Eigendecomposition of the *Laplacian* matrix. The original SH paper [46] strongly relies on the *eigendecomposition* of the *Laplacian* matrix (often referred to as *diagonalization*), which becomes possible because L is positive semi-definite (see properties of the *Laplacian* matrix). Briefly, diagonalizing a matrix corresponds to decomposing it in a special form and finding a particular coordinate system to represent it. In fact, this new coordinate system is given by the eigenpairs of the matrix.

Let us follow the exact equations which lead to *Laplacian*'s diagonalization, as this explains the eigenvector minimization constraint outlined in the original SH paper [46] (see *Spectral Relaxation*). Since the eigenvectors of L are pairwise orthogonal (see item 4 of the *Laplacian* matrix), we can merge them all in an orthogonal matrix Y. From this, we can derive the following equality [5]:

$$Y^{-1} = Y^T \tag{3.9}$$

If the eigenvalues of L are also stored diagonally in a diagonal matrix Λ (i.e. where all the non-diagonal entries equal 0), the generic eigenvalue Equation 3.6 becomes:

$$LY = Y\Lambda \implies L = Y\Lambda Y^{-1} \tag{3.10}$$

From Equation 3.9 and Equation 3.10, we obtain the following:

$$L = Y\Lambda Y^{T} \tag{3.11}$$

Since L is diagonalizable and can be represented in terms of an equivalent diagonal matrix, Equation 3.11 can be written as follows:

$$L = Y\Lambda Y^{T} \iff \Lambda = Y^{T}LY \tag{3.12}$$

Equation 3.12 becomes particularly relevant, as it is referenced in the original SH paper [46] (see minimization function in the *Spectral Relaxation* section).

Square matrix properties closely related to eigenvalues. The trace, rank and determinant of a square matrix X are highly connected to X's eigenvalues [5], as such:

1. The trace of X equals the sum of its eigenvalues;
2. The determinant of X equals the product of its eigenvalues. This makes perfect sense geometrically (e.g. for a $2D$ example, the determinant consists of the area between the two coordinate axes and that is given by the product of their magnitudes/eigenvalues);
3. The rank of X indicates how many non-zero eigenvalues X has.

The Fiedler vector. The second smallest eigenvector derived from the *Laplacian* matrix L, also known as *the Fiedler vector*, plays an important role

when it comes to mapping the vertices of its underlying graph $G(V, E)$ to a line. This is because it creates a very good arrangement of the vertices along the line. This corresponds to *walking* across the graph vertices and placing them on a line next to each other in a way that best describes the distances or, for that matter, the similarities among them. Such an arrangement can give a hint about clusters and therefore, how vertices group in terms of similarities [36].

***Laplacian*'s connection to graph minimization problems/*Laplacian*'s Quadratic Form.** One of the *Laplacian* operator's properties which hints to why L is convenient for Vanilla SH and holds for every n-dimensional vector/vertex v is the following

$$v^T L v = \frac{1}{2} \sum_{i,j=1}^{n} w_{ij} \left(v_i - v_j\right)^2 \tag{3.13}$$

where w_{ij} defines similarities and v_i and v_j are dimension values of v [36]. The right side is closely related to the objective function in Equation 3.2 shown in the original SH paper [46]. The left side takes the same form as Equation 3.12, where v is only one of the eigenvectors of matrix Y (Y stores all the eigenvectors of L). Known as *Laplacian's quadratic form*, this relationship suggests the *Laplacian* matrix is suitable for solving minimization problems related to graph cuts. Vanilla SH is also such a problem. Additionally, 'this form measures the smoothness of' v [40], where vector v can be thought of as a function over vertices. It makes sense that the smoother the vector v, the smaller the quadratic form, in relation to the minimization function.

The *Laplace-Beltrami* operator. The *Laplacian* matrix is closely related to the *Laplace-Beltrami operator*. In fact, it represents a 'discrete version of the *Laplace-Beltrami* operator' [36].

The *Laplacian*, intuitively. Conceptually close to derivatives, the *Laplacian* operator takes in a function and returns another one.

Let us consider a function f which accepts two arguments x and y. Figure 3.4 illustrates a possible input space S_i, represented by the lower plane and a possible output space S_o, drawn above S_i, as a curved plane. We reference the *Laplacian* of f as Δf. This also accepts the two arguments x and y, outputting a scalar v_{out}, where $v_{out} \in S_o$. The *Laplace-Beltrami* operator is associated with the concept of second derivative, because Δf represents the 'divergence of the gradient' of function f [3]:

$$\Delta f(x, y) = Laplacian(f(x, y)) = div(grad(f)) \tag{3.14}$$

Knowing that $f(x, y)$ returns a scalar and the gradient $grad(f)$ consists of a vector field (see the arrows of S_i in Figure 3.4), the divergence $div(grad(f))$ of a vector field results into a scalar value. Generally, the gradient represents the slope of the tangent of f. As such, the gradient corresponds to a vector field in S_i, where each vector points to a direction. The set of all such vectors converge in some zones (see the red arrows of S_i in Figure 3.4) and diverge in others (see the green arrows of S_i in Figure 3.4). This is reflected in the output space S_o by the structure of the curved plane (i.e. up-hills and down-hills). An up-hill gets generated in zones where the input space vectors converge, while a down-hill gets generated in zones where the input space vectors diverge. In zones where the vector field of S_i does not describe any particular vector orientation, divergence tends towards zero. The output space S_o consists of minimum points where the divergence of the gradient is high and maximum points where the divergence of the gradient is low. Following this intuition, the *Laplace-Beltrami* operator provides a measure of how much of a maximum or minimum point the input (x, y) is: if (x, y) evaluates to higher values than neighboring points, it is negative, whereas if (x, y)

evaluates to lower values than neighboring points, it is positive. In other words, the *Laplace-Beltrami* operator indicates the degree to which the value of the field deviates from the mean value calculated over the neighbourhood (i.e. the shape of the field). Since this is a similar behavior to the second derivative, the *Laplace-Beltrami* operator works like a second derivative for multivariable functions.

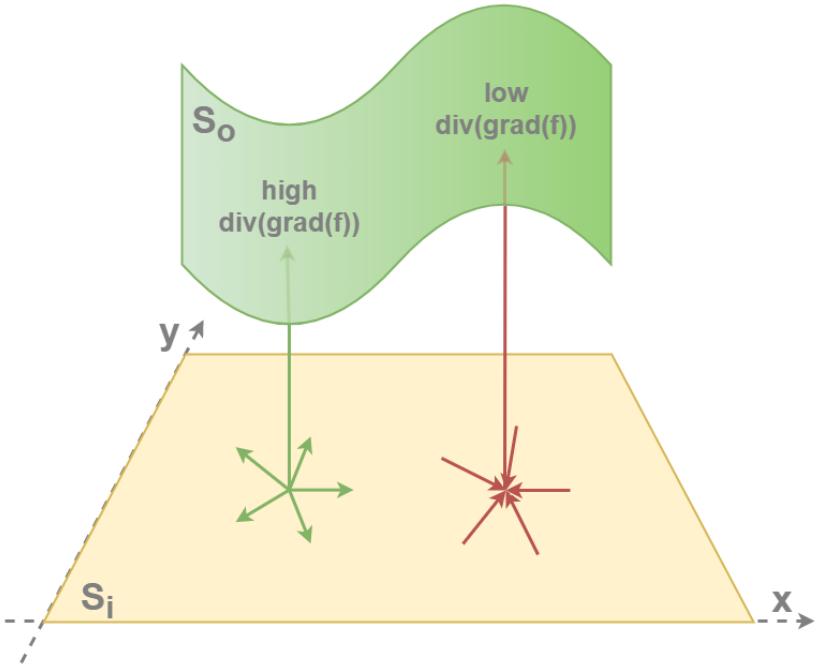


Figure 3.4: Intuitive representation of the *Laplace-Beltrami* operator. The input space S_i is illustrated by the lower (yellow) plane, while the output space S_o is illustrated by the upper curved (green) plane. The arrows in S_i describe a vector field.

3.4 Spectral Relaxation

Connections with Spectral Clustering. The spectral relaxation used in Vanilla SH is identical to the one used in Spectral Clustering [28]. By relaxing the constraint according to which entries of the solution vectors $Y_n = \{y_i\}_{i=1}^n$ are restricted to only two values, we obtain an easier problem whose solutions are the k eigenvectors of L with minimum non-zero eigenvalues. Nevertheless, the purpose is still to partition the graph $G(V, E)$. For this reason, we must transform the real-valued eigenvectors of the new relaxed problem back into discrete vectors. This is where thresholding at 0 comes into play in the algorithm.

Distinction between PCA's basis and *Laplacian*'s eigenvectors. Let us remember the previous discussion about PCA's role in Vanilla SH (see subsection 3.3.6). In fact an initial step in the algorithm, PCA finds the most convenient basis to represent a given dataset X_n. The newly identified coordinate system is given by a set of eigenvectors (i.e. principal components). Nevertheless, these are different from the eigenvector solution proposed in *Spectral Relaxation* [46]. While PCA's principal components provide information regarding the axes of largest variance in the data, the eigenvectors proposed as a solution to our problem offer information about where it is most beneficial to create cuts in the graph $G(V, E)$.

3.5 Out of Sample Extension

***Laplacian*'s role.** The *Laplacian* draws a correspondence between graph as a discrete representation and metric spaces/manifolds as a continuous representation. A distance on the graph between any two vertices v_i and v_j can be effectively represented in the *Laplacian*'s eigenvectors space by a

spectral distance of the embedding [1].

Computing hashcodes for novel data points. The connection between the *Laplacian* matrix and the *Laplace-Beltrami* operator is similar to the correspondence between an eigenvector and an eigenfunction and represents the key to compute the binary representation of an unseen data point. As literature states it, the eigenvectors derived from the *Laplacian* matrix L converge to the eigenfunctions of the *Laplace-Beltrami* operator on manifolds [10]. The former hints to a discrete representation and therefore, it is restricted to only providing hashcodes for data points known of it. However, the latter involves a continuous representation and therefore, it becomes generic, lacks restrictions and can assign hashcodes to any given data point, even if unseen.

4

Deeper into the SH Implementation

This section explains the implementation of Vanilla SH in more detail, as it is proposed in the provided Matlab code [45].

Training phase implementation. Figure 4.1 gives an overview of the training phase in Vanilla SH.

Starting from the top left corner of Figure 4.1 and ending with the bottom right one, the training phase of the algorithm goes through the following:

1. *1T*: We pass the algorithm a normalized d-dimensional training set X_{train} of size n;
2. *2T*: Given k the dimensionality we select for encoding, we apply PCA on X_{train} in order to find a good basis for it (see subsection 3.3.6). This means we determine a set of k new dimensions, namely the principal components (*PCs* in Figure 4.1), which form a novel space for representing X_{train};
3. *3T*: We project the initial dataset X_{train} onto the chosen space (i.e. X_{train} is effectively reduced in dimensionality from n to k);
4. *4T*: From each chosen principal component PC_i (where $i \leq k$), we extract minimum and maximum values and implicitly, range values;

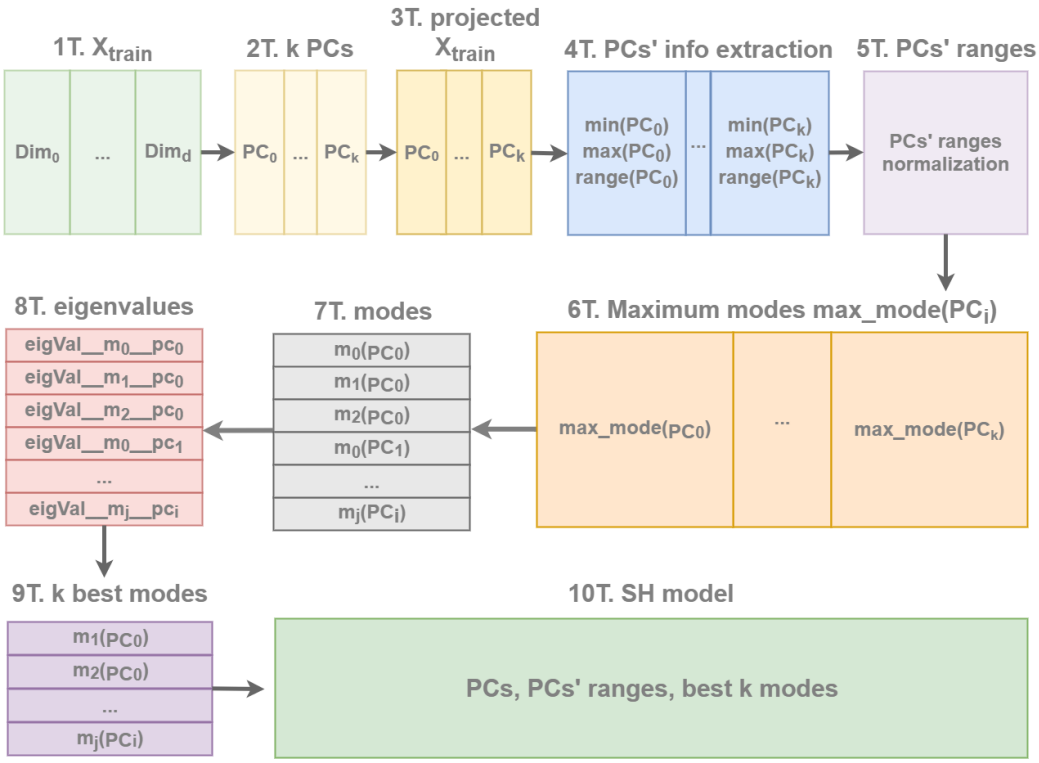


Figure 4.1: Implementation steps for Vanilla SH - the training phase.

5. *5T*: We normalize principal components' ranges, such that the maximum of all ranges always has value 1;

6. *6T*: Given the normalized ranges, we calculate a maximum mode value $max\_mode(PC_i)$ for each principal component PC_i. This depends on how long PC_i is (i.e. on how much data varies in each dimension). The longer the principal component, the bigger the value of its maximum mode;

7. *7T*: Knowing the maximum mode value $max\_mode(PC_i)$, we create a number of modes for each principal component PC_i. The bigger the maximum mode for a principal component, the more modes we generate for it. The mode of a principal component PC_i looks like a transposed

vector m_i =[0 0 0 ... $mode\_value(PC_i)$ ... 0], with zero entries everywhere, except the i^{th} position, where we encounter the actual mode value of PC_i. Furthermore, $mode\_value(PC_i)$ can be at most the maximum mode $max\_mode(PC_i)$ and it indicates the $mode\_value(PC_i)^{\text{th}}$ time principal component PC_i is used for partitioning and bit extraction later on, in the compression phase;

8. *8T*: To each mode created in step 7, we associate an eigenvalue by calculating it with the formula stated in the original SH paper [46];

9. *9T*: One of the resulted eigenvalues in step 8 always equals 0. However, according to the spectral graph theory (see subsection 3.3.8), we select the k smallest non-zero eigenvalues. For this reason, we sort the set of eigenvalues ascendingly and pick the modes corresponding to the k smallest non-zero ones.

10. *10T*: The last step of the training phase involves storing the model, so it can be used later on, in the compression phase. The model consists of: principal component axes, principal component ranges and the best k modes selected in step 9 (see section 2.1).

Compression phase implementation. Figure 4.2 provides an overview of the compression phase in Vanilla SH.

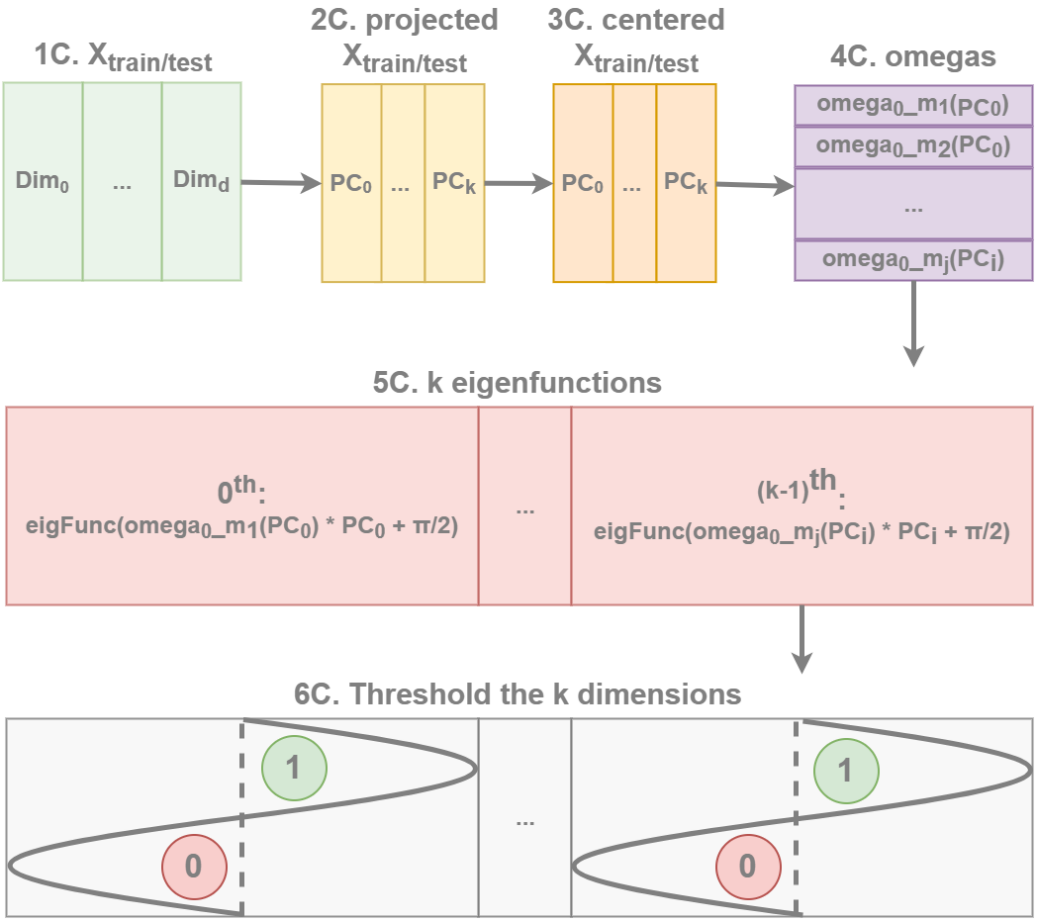


Figure 4.2: Implementation steps for Vanilla SH - the compression phase.

Starting from the top left corner of Figure 4.2 and ending with the bottom one, the algorithm goes through the following steps during compression:

1. *1C*: Accepts a normalized d-dimensional dataset referred to as $X_{\text{train/test}}$, since it is X_{train} when we compress the training set and X_{test} when we compress the testing set;

2. *2C*: We project $X_{\text{train/test}}$ on the k principal components stored in the model after training (see Figure 4.1). Importantly, these are defined

only on the training set, but the exact same axes are used for compressing both X_{train} and X_{test};

3. *3C*: We center the data in $X_{\text{train/test}}$ (i.e. we substract the corresponding minimum value from each of the k dimensions);

4. *4C*: For each of the k modes m_i stored in the model during training, knowing the principal component it derives from (e.g. PC_j), we calculate a variable $omega_0\_m_i(PC_j)$ by

$$omega_0\_m_i(PC_j) = \frac{\pi}{range(PC_j)} \tag{4.1}$$

   For a mode, $omega_0$ expresses how many times the associated principal component's range goes around the quadrant with a unit measure of π. We refer to the set of all k $omega_0$ as to *omegas*;

5. *5C*: Having determined the principal components, their associated *modes* and *omegas* (see 4), we use the eigenfunction definition from the original SH paper [46] (see Equation 2.1) in k independent iterations, where each is applied to one dimension at a time. A dimension consists of its associated principal component scores, outputted from the eigenfunction with the corresponding parameters $omega_0$ and mode value m;

6. *6C*: In each iteration from 5, we threshold the given dimension at 0. Whenever the principal component score outputted from the eigenfunction is positive, the resulted bit is 1, whereas in the opposite case, the bit is 0. In the end, all the bits are concatenated into a hashcode.

5

SH Variations and Improvements over Vanilla SH

This section proposes a series of SH variations which slightly deviate from the original implementation and seem to improve performance in most cases. We discuss one variation at a time, but for comparison purposes, we start with Vanilla SH itself. The variations are referred to as:

1. Vanilla SH (the original implementation, also abbreviated as V-SH);
2. Balanced SH (abbreviated as B-SH);
3. Median SH (abbreviated as M-SH);
4. PCcutrepeated SH (abbreviated as PCCR-SH);
5. PCcutrepeatedmultiplebits SH (abbreviated as PCCRMB-SH);
6. PCdominancebymodesorder SH (abbreviated as PCDMO-SH);

The proposed variations differ from Vanilla SH only in terms of compression, while the training phase remains unchanged.

5.1 Vanilla SH

In order to draw a comparison among the SH variations, we firstly aim to underline some particularities of Vanilla SH.

The compression in Vanilla SH, briefly. As previously described (see Figure 4.1), Vanilla SH compresses a dataset $X_{\text{train/test}}$ by going through k independent iterations. In each iteration, it creates a box of principal component scores with specific parameters, which we reference as *individual data box*. We apply Equation 2.1 over the scores. As a result, we obtain a sine curve over the individual data box and threshold the outputs at 0: positive ones are assigned a bit of 1, while non-positive ones are assigned a bit of 0.

Feeding the right variables to compression. Figure 5.1 provides a more detailed overview of how Vanilla SH hashes data points. Using a dataset $X_{\text{train/test}}$ (step *1V*) which gets projected on the principal components (step *3V*) stored from training (step *2V*), the best k modes also stored in the model (step *4V*) and the associated principal components' *omegas* (see *5V*), the algorithm multiplies all three parameters in k independent iterations, adds $\frac{\pi}{2}$ as a constant, passes them to the eigenfunction for thresholding (see *7V*) and finally, it obtains bits (see *8V*). Importantly, the three parameters multiplied in each individual iteration are combined appropriately: $omega_0$ uses the range of the corresponding principal component of the same iteration, while the mode indicates a cut on the same principal component (see Figure 5.1).

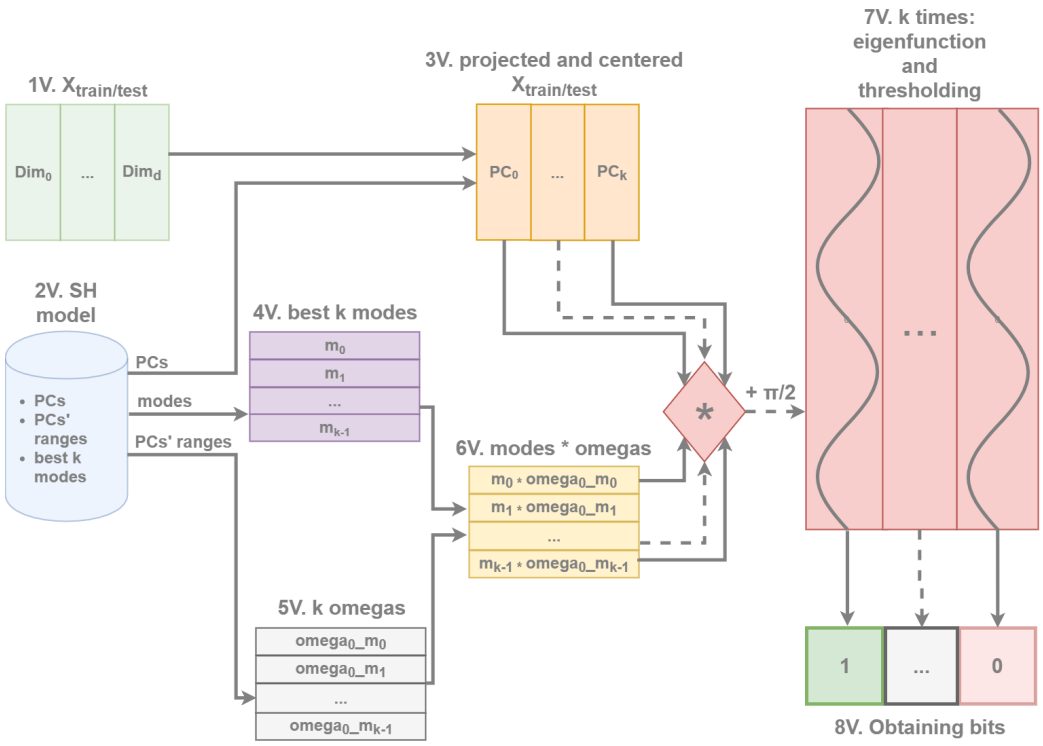


Figure 5.1: Detailed overview of the compression phase in Vanilla SH.

What modes indicate. A mode m_i looks like a transposed vector with all entries zero, except for the mode value m (see Figure 5.2) and provides information through:

1. The column index of the mode value m in mode m_i: It equals the index of the principal component that is to be cut in order to obtain bits;

2. The mode value m: It indicates how many times the associated principal component is partitioned. This is also equivalent to the m^{th} time the same principal component is cut.

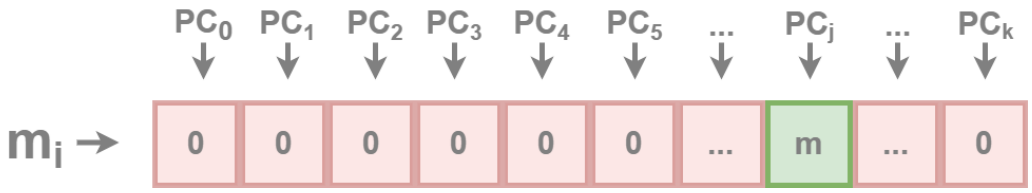


Figure 5.2: Mode format in Vanilla SH.

Modes' order. Each iteration uses one of the k modes stored during training. Vanilla SH associates the k best modes with the k best (smallest) non-zero eigenvalues. As a result, the modes are also ordered from the one corresponding to the smallest eigenvalue to the one corresponding to the k^{th} eigenvalue.

Principal components in relationship with modes. Given the importance of the modes, what they indicate and how data is structured, it is likely for some iterations to work with modes of the exact same principal component PC_j. That only means Vanilla SH obtains more bits from PC_j and decides to ignore other principal components (i.e. not extract any bits based on those). The algorithm most likely relies on the assumption that PC_j might provide more valuable information about the data than others. As a consequence, some of the principal components contribute with more bits than others.

Figure 5.4 shows the correspondence between the modes ($m_{0\text{-}9}$) in Figure 5.3 and the principal components ($PC_{0\text{-}9}$), for each iteration ($i_{0\text{-}9}$). After each iteration, Vanilla SH obtains precisely 1 bit.

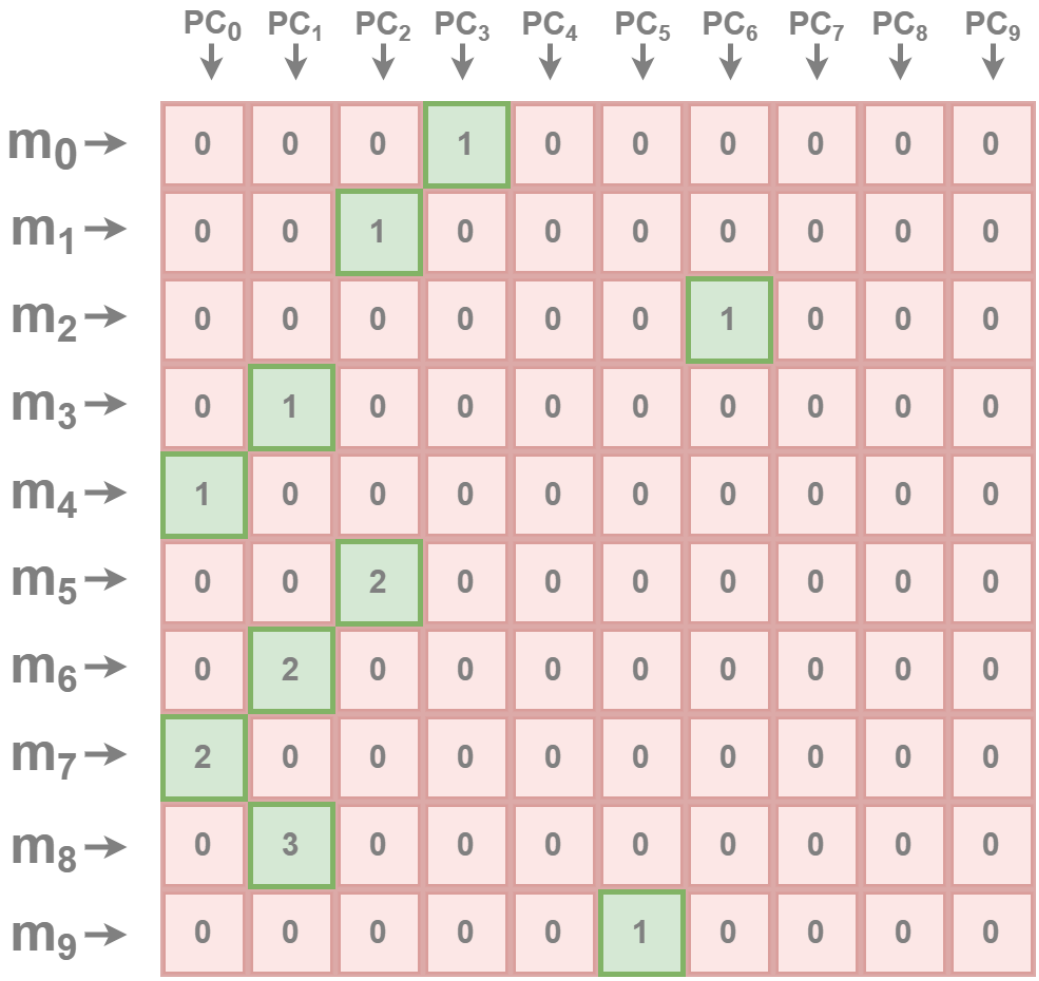


Figure 5.3: Modes matrix example, where we aim for $k = 10$ bits, each resulted from one mode at a time. Even though there are k principal components available for thresholding and bit extraction, in this example, Vanilla SH only cuts some of them (i.e. $PC_{0\text{-}3}$ and $PC_{5\text{-}6}$), while ignoring the rest, which do not contain any mode value across (check the all-zeros columns for PC_4, PC_7, PC_8 and PC_9).

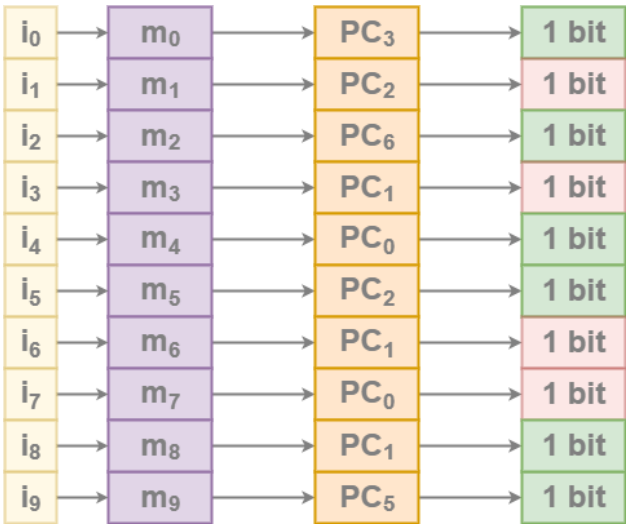


Figure 5.4: The connection between modes and principal components in Vanilla SH, given the modes in Figure 5.3.

The Hamming distance between neighboring buckets - special cases. After generating 1 bit per iteration, Vanilla SH concatenates all k resulted bits into a hashcode associated with a given point x_i. The way Vanilla SH concatenates bits creates interesting patterns regarding the Hamming distance between neighboring buckets in the specific case where bits are extracted from the same principal component.

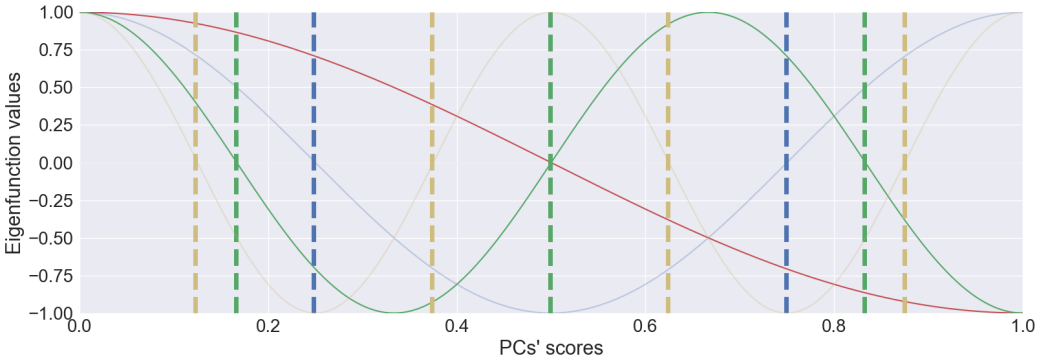


Figure 5.5: Multiple cuts on the same principal component, where each cut corresponds to a different iteration in Vanilla SH. The x-axis shows principal component scores and the y-axis shows eigenfunction values.

Assuming a uniform distribution of the data, Figure 5.5 shows how 4 eigenfunctions (the sine curves) all cut the same principal component with different frequencies (i.e. different mode values). Each sine function is assigned a different color. The same color of the sine marks the vertical cuts at its intersections with the x-axis. Interestingly, the vertical cut in the middle (marked with green) overlaps with a previous cut made in the exact same spot by the red sine curve. This has two consequences:

1. The Hamming distance between the two neighboring buckets in the middle of the individual data box is bigger than the rest of the Hamming distances between other neighboring buckets. The more cuts we have on the same principal component, the bigger the gap between the Hamming distances of the neighboring buckets;

2. Given $k = 4$ bits, the maximum number of buckets which can be used for encoding is $num\_buckets = 16$. Nevertheless, by overlapping cuts, Vanilla SH uses a smaller number of buckets (10 in Figure 5.5).

Buckets' sizes. Once again assuming uniform data distribution, whenever

Vanilla SH cuts the same principal component more than two times, the buckets' sizes start varying. This is also clear in Figure 5.5. The more cuts on the same principal component, the bigger the gap between the buckets' sizes.

5.2 Balanced SH

Further on, we present the first SH variation - Balanced SH.

Motivation. The original SH paper admits as one of the limitations of Vanilla SH the assumption that 'a multidimensional uniform distribution generated the data' [46]. Additionally, it emphasizes the idea that assuming uniform distribution on each of the k rectangular approximations fit on the training set works fine for a diversity of data distributions. Considering this a starting point, there are three main reasons why we propose Balanced SH:

1. In some cases, Vanilla SH does not partition the principal component axes into equally sized buckets;
2. Sometimes, it even throws away some of the buckets and uses fewer than it can actually use;
3. Often, there exists a bigger gap in terms of the Hamming distance between some of the neighboring buckets than between others.

Approach. Compared to Vanilla SH, the compression phase in Balanced SH involves the following:

1. Instead of looping through each mode at a time, as Vanilla SH, it loops through principal components' indices;

2. From the modes matrix, we know both which principal components Vanilla SH obtains bits from and how many bits each principal component contributes with. As such, in iteration i corresponding to principal component PC_i, Balanced SH obtains as many bits as Vanilla SH totally extracts from PC_i. Therefore, the principal components unused in Vanilla SH for bit extraction are also skipped in Balanced SH;

3. Unlike Vanilla SH, Balanced SH obtains a different number of bits from each iteration. For instance, if Vanilla SH uses two individual sine partitions in two different iterations to concatenate 2 bits from PC_i to the final hashcodes, Balanced SH obtains the 2 bits at once. It achieves this by directly cutting one of the two individual data boxes created in two different iterations in Vanilla SH. In fact, it cuts the first individual data box of all the ones ever built in Vanilla SH. This corresponds to principal component index i and its first mode in the modes matrix. As an example, given the modes in Figure 5.3, Balanced SH follows the iterations in Figure 5.6, where we notice the correspondences drawn between principal components' indices and modes' indices.

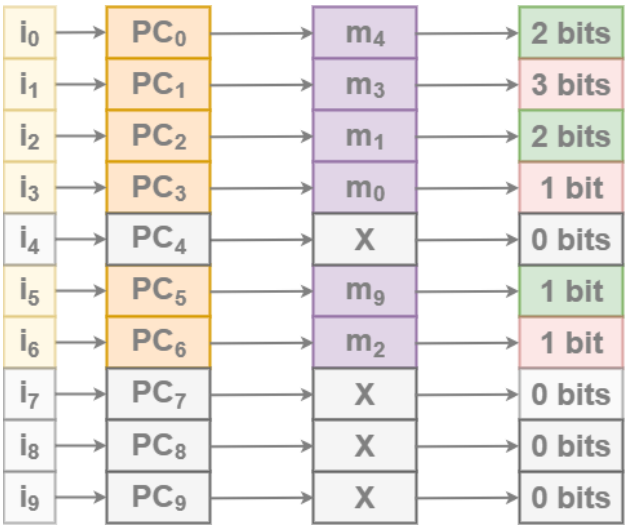


Figure 5.6: Summary of bits' extraction for Balanced SH, along with the correspondences drawn between principal components' indices and modes' indices, based on the modes matrix in Figure 5.3. The iteration rows marked with grey do not occur in Balanced SH.

As a result, the number of iterations lowers, as some of them output more than just 1 bit at a time.

4. As the name suggests, Balanced SH always cuts the principal component axes in equally sized buckets. For uniformly distributed data, all buckets contain approximately the same number of hashed items. For extremely irregular data distributions, or at least if the principal component scores are not uniformly distributed, this method is expected to perform badly (e.g. data with outliers);

5. In order to lower the Hamming distance gap between neighboring buckets often present in Vanilla SH, Balanced SH labels the buckets by generating Gray codes in each iteration and concatenating them. The thought behind this choice is to ensure Hamming distance $d\_ball_{\text{Hamm}} = 1$ between all neighboring buckets, since Gray codes are binary representations where two successive labels differ in one bit. However, this

results in the two buckets situated at opposite ends of a principal component axis differing only a little, as the Hamming distance between them is $d\_ball_{\text{Hamm}} = 1$ (i.e. Gray codes are circular). Yet, this limitation most likely has a less strong effect than the Hamming distance gap in Vanilla SH, since it occurs for one and only one pair of neighboring buckets (the ones at the principal component's extremities), whereas Vanilla SH is likely to cause this Hamming gap for several pairs of neighboring buckets.

6. As a result of often extracting more than only one bit from some of the iterations, the order of concatenating the bits in the final hashcodes differs from the one in Vanilla SH. Nonetheless, this does not affect the evaluation at all, since both the training and the testing set are compressed exactly the same way. In other words, bits in the same position in the hashcodes are comparable, as they originate from the same iteration. For instance, the Hamming distance between hashcodes in pair $(001, 110)$ is identical to the one between hashcodes in pair $(010, 101)$, where the only difference between the two pairs of hashcodes consists of swapping the bits by column;

7. Since compression always projects data on the principal component axes stored during training, when compressing the testing set, outliers might show up. These are principal component scores which take values outside the range of the individual data box range calculated during training. Balanced SH handles the situation by placing all the outliers on the left of the individual data box in the first bucket, and placing all the items on the right of the individual data box in the last of the buckets.

The purpose behind Balanced SH in comparison with Vanilla SH, briefly. Taking into account all the differences between Vanilla SH and Balanced SH, the latter aims to make better use of the number of available

buckets, given a particular number of bits. Additionally, it aims to stress out and use the assumptions regarding uniform data distribution across principal component axes [46]. Therefore, for data distributions where principal components are cut more than two times, Balanced SH is actually expected to perform slightly better than Vanilla SH.

5.3 Median SH

Motivation. Taking into account the fact that Balanced SH is not data-driven, since it splits the space in equal buckets without looking at how data is actually distributed, we propose Median SH. This version adopts the exact same approach as Balanced SH, except for one aspect: it splits scores in each principal component equally with regards to the data distribution. In other words, each bucket contains the same number of items, regardless whether the data is uniformly distributed or not (see the balanced partitioning constraint in section 3.2). Indeed, if the data on each axis is uniform, this version is expected to behave similarly to Balanced SH. Median SH is probably preferred in cases where the data of our individual data box has way too much structure (e.g. too clustered, with very differently spaced out clusters) or where it presents outliers. This is because splitting clusters in some cases might be better than using Balanced SH and obtaining buckets with extremely different sizes (where *sizes* refers to the number of items in the buckets).

Approach. The only difference between Balanced SH and Median SH in terms of implementation consists of the way the individual data boxes are partitioned. Knowing from the start in how many buckets it needs to divide each axis, Median SH cuts recursively by the median point.

5.4 PCcutrepeated SH

This subsection introduces a flavor of Balanced SH which often extracts repeated bits from the same principal component, compared to Vanilla SH or Balanced SH.

Motivation. As previously shown, Vanilla SH and Balanced SH use the correspondence between modes and principal components. This is maintained the same for both versions. As a reminder, Figure 5.7 shows this again, given the modes in Figure 5.3: Vanilla SH uses each one of the purple modes, while Balanced SH only uses the purple modes marked with full opacity (i.e. it cuts only the first individual data box associated with the principal component it extracts bits from, meaning it only uses the first mode).

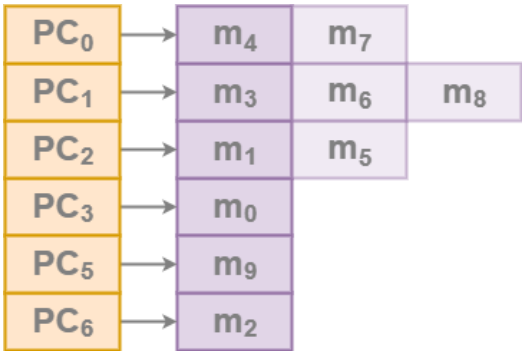


Figure 5.7: The correspondence between modes and principal components, based on the modes matrix in Figure 5.3.

PCcutrepeated SH aims to exploit these connections by slightly swapping them in order to explore whether it makes a difference in terms of performance.

Approach. PCcutrepeated SH does the following:

1. It loops through indices of principal components which normally bring bit contribution in Vanilla SH (or Balanced SH);

2. It intends to extract the same number of bits from each principal component whose index it loops through as Vanilla SH (or Balanced SH) does;

3. It breaks the correspondence between modes and principal components from Balanced SH. Instead, it pairs each principal component with index i (i.e. PC_i) with a mode of the same index i. The intention is to obtain a number of bits from the individual data box created with mode of index i equivalent to how many bits Balanced SH obtains from the principal component of index i. For instance, in Figure 5.7, we notice how Balanced SH extracts 2 bits from an individual data box built up with mode $m = 1$ for principal component with index $i = 2$. Therefore, PCcutrepeated SH aims to still extract 2 bits in iteration with index $i = 2$. However, instead of preserving the mode index $m = 1$ as Balanced SH, it creates an individual data box from a mode with the same index $m = 2$ as the principal component's index (i.e. PC_2). By inspecting Figure 5.7 again, we notice how mode of index $m = 2$ corresponds, in fact, to principal component PC_6. As such, this turns to be a bit extraction from PC_6, and not from PC_2. Figure 5.8 shows how the entire situation from Figure 5.7 changes: starting with the initial principal components which bring bit contribution, we firstly consider how many bits they normally contribute with, and secondly their indices, and use the two in order to actually get the same number of bits from modes of these indices. Notice how we end up obtaining 2 bits from PC_1 (marked with blue in full opacity) in two different iterations, whereas this happens neither in Vanilla SH, nor in Balanced SH.

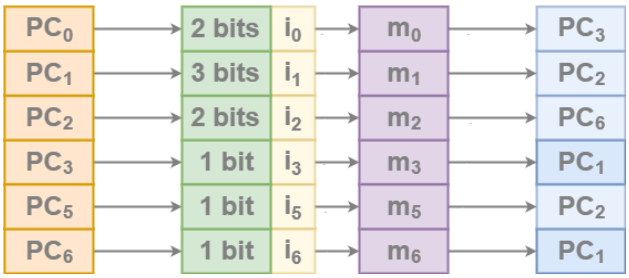


Figure 5.8: The correspondence between modes and principal components in PCcutrepeated SH, based on the modes matrix in Figure 5.3. The effect of repeated bits is emphasized by principal component PC_1 (marked with blue of full opacity), as it is used in two different iterations for bit extraction.

Therefore, one consequence is that some bits obtained from specific principal components are repeated in the final hashcodes. That means they occupy positions which bits from other principal components could, otherwise, have reserved.

5.5 PCcutrepeatedmultiplebits SH

Motivation. Considering PCcutrepeated SH a reference point, we propose PCcutrepeatedmultiplebits SH as a version which obtains repeated bits from specific principal components in a more aggressive fashion.

Approach. Given Figure 5.7, in Vanilla SH we obtain 3 bits for principal component PC_1 from individual data boxes constructed with modes of indices $indices \in \{3, 6, 8\}$, one bit at a time. In this example, PCcutrepeatedmultiplebits SH decides to obtain a number of bits

$$\text{num\_bits} = \frac{|\text{indices}| \cdot (|\text{indices}| + 1)}{2} \tag{5.1}$$

from all the modes indicated by the array $indices = \{3, 6, 8\}$. It does this by taking a number of bits equal to $len(indices)$ from the mode with the first/smallest index in $indices$. Afterwards, it continues taking a number of bits from the next modes decreased by 1 every time (i.e. 2 bits from mode m_6 and 1 bit from mode m_8).

Obviously, by increasing the number of bits we extract from the same principal component, we need to cut off bit extraction as soon as the hashcode length equals k.

Additionally, this version loops in the order of the modes' indices, like Vanilla SH. Figure 5.9 shows the correspondence between modes and principal components in PCcutrepeatedmultiplebits SH, given the modes in Figure 5.3.

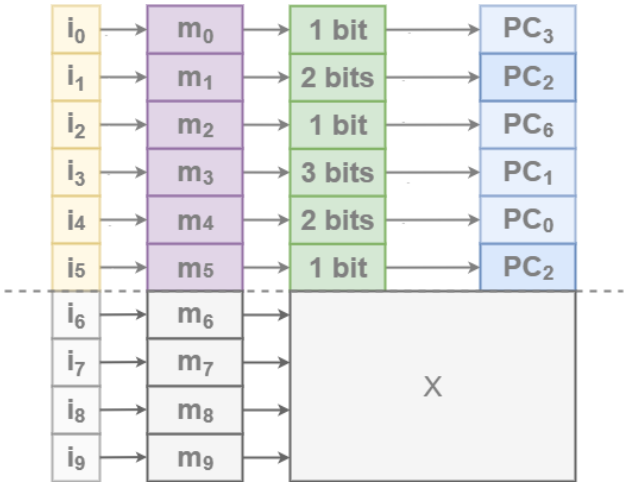


Figure 5.9: The correspondence between modes and principal components in PCcutrepeatedmultiplebits SH, based on the modes matrix in Figure 5.3.

5.6 PCdominancebymodesorder SH

Motivation. This version exploits the following:

1. Given the fact that modes are ordered increasingly in terms of the eigenvalues and knowing that the smallest eigenvalue, as well as the first mode, theoretically provide the best graph partitioning, we aim to obtain a number of bits from each principal component proportional to its importance in terms of the modes' order;

2. The number of bits obtained from each principal component still follows a similar scheme as Vanilla SH. However, we show that empirically, approaching a slightly different order yields better performance.

Approach. In each iteration, PCdominancebymodesorder SH pairs up two numbers, such as: (pc_index, n_bits). The idea is to extract a number of bits $(n\_bits + 1)$ from an individual data box built up with the mode of index pc_index. For instance, starting from the modes matrix in Figure 5.3, we create two arrays, as follows:

1. $pc\_indices\_array$: Contains principal components' indices whose order is derived from the modes' order. For the modes example in Figure 5.3, $pc\_indices\_array = \{3, 2, 6, 1, 0, 2, 1, 0, 1, 5\}$. Note that we preserve repeated entries in $pc\_indices\_array$;

2. n_bits_array: Looking at the modes matrix, it stores the number of bits each principal component contributes to in Vanilla SH decreasingly and it increments each by 1. For the modes example in Figure 5.3, $n\_bits\_array = \{3 + 1, 2 + 1, 2 + 1, 1 + 1, 1 + 1, 1 + 1\}$. The thought behind incrementing each number of bits by 1 relies on weighing the bits representing the most significant cuts more.

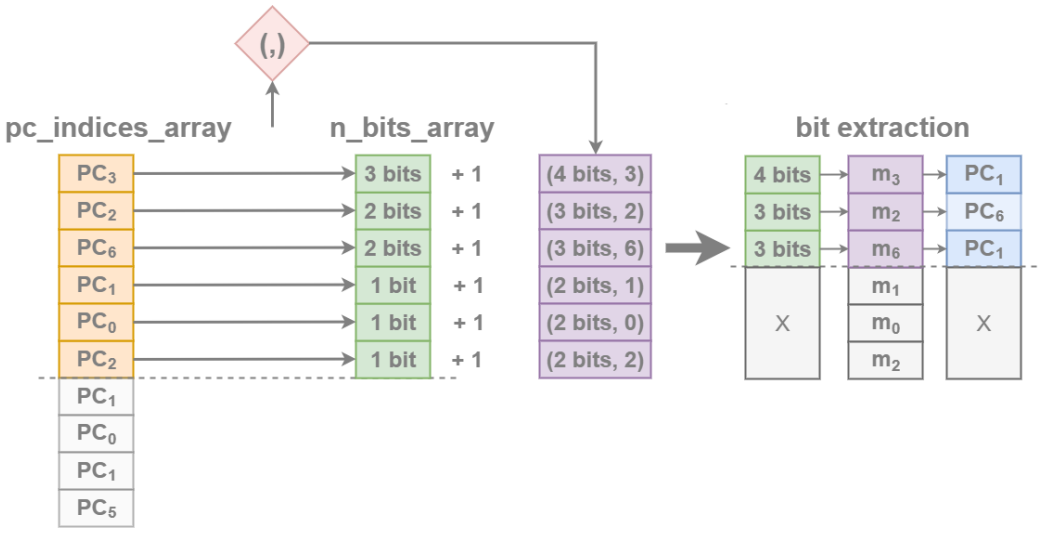


Figure 5.10: The correspondence between modes and principal components in PCdominancebymodesorder SH, based on the modes matrix in Figure 5.3.

By pairing the items in the two arrays up to the size of the biggest one, we obtain an array of tuples like $(n\_bits, index)$. For the modes in Figure 5.3, we obtain the following tuples array: $[(4,3),(3,2),(3,6),(2,1),(2,0),(2,2)]$. The first number of each tuple represents the number of bits we aim to extract from an individual data box built up with a mode of index equal to the second number in the tuple. Even though initially, PCdominancebymodesorder SH aims to use the second item in each tuple as index of which principal component to cut, it experimentally turns out that using it as mode index, instead, increases performance. In other words, we actually do not cut the principal components depending exactly on how dominant they are, but we use this information in our process and end up cutting principal components in a slightly different order than by their importance. Nonetheless, in practice, the modes matrix most likely never associates its first mode with a less significant principal component. Therefore, using the second item of each tuple as mode index instead of principal component index does not make a big difference in practice. Figure 5.10 shows the correspondence between mode

indices and principal component indices, as well as bit extraction, given the modes in Figure 5.3.

Similarly to PCcutrepeatedmultiplebits SH, PCdominancebymodesorder SH uses a cut-off as soon as the hashcode length equals k.

6

Experiments

This chapter presents the experiments conducted for Vanilla SH, as well as for the proposed SH variations (see chapter 5) and aims to interpret the results.

6.1 Experimental Setup

The original SH paper [46] proposes a Matlab code for Vanilla SH. For the purpose of this thesis, the given code has been translated into Python code. Experiments were conducted on the same server in order to obtain comparable results. The use of parallel Python instances sped up the experimental process. Additionally, the evaluation is optimized, such that the ground-truth is calculated once per dataset by storing and reusing it for different models.

Training/testing considerations. All data was sampled with different sample sizes, split into training and testing sets and normalized before being fed to the algorithm. Any testing set is 10% the size of its corresponding training set. Compared to other papers, where training and testing sets coincide [31], in our setup, they are not intended to be the same. This choice relies on the assumption that fitting a model on a dataset and testing

it with the exact same data does not prove how good the model is, since such a testing environment does not stress the model under new conditions. Additionally, in our setup, the testing set consists of data points used as query points for the training set and therefore, it is even more important for these to be unseen items.

6.1.1 Datasets

Experiments were mostly performed on real datasets, but occasionally on artificial datasets, in order to explore particular properties for the proposed methods. The real datasets are referred to as follows:

1. Profi-set [12]: Consists of 20M image descriptors of dimensionality $d = 4,096$, suitable for content-based search;
2. Profi-set-JL [12]: Represents a lower-dimensional version of the Profi-set, reduced with the *Johnson-Lindenstrauss* transform [24] to a dimensionality $d = 256$. This transformation effectively preserves the pairwise distances between items in the dataset;
3. MNIST [26]: Being suitable for ML approaches, it contains 70,000 entries of dimensionality $d = 784$, representing pixel intensities of 28×28 sized images of handwritten digits;
4. SIFT [23]: Includes 1M entries of SIFT image descriptors of dimensionality $d = 128$ which are appropriate to use for ANNS problems.

6.2 Evaluation

The performance of an algorithm is measured with quality metrics.

6.2.1 Quality Metrics

Precision and recall are two of the most popular quality metrics to use. Formally, precision (p) equals the number of true positives (T_p) over the sum of true positives and false positives (F_p), whereas recall (r) equals the number of true positives over the sum of true positives and false negatives (F_n) [4]:

$$p = \frac{T_p}{T_p + F_p} \tag{6.1}$$

and

$$r = \frac{T_p}{T_p + F_n} \tag{6.2}$$

If we consider S_{approx} the set of approximated data points retrieved by the algorithm and S the set of precise data points that we want the algorithm to return (i.e. the set of k nearest neighbors for a given query point q), precision and recall can be defined as such:

$$p = \frac{|S \cap S_{approx}|}{|S_{approx}|} \tag{6.3}$$

and

$$r = \frac{|S \cap S_{approx}|}{|S|} \tag{6.4}$$

Precision defined. Precision measures how accurate the results are, as it evaluates the number of correct predictions over the total number of predictions made.

Recall defined. Recall measures how many of the total relevant results we find, as it evaluates the number of correct predictions over the total number of correct predictions.

Imbalance between precision and recall. Obtaining high precision and

low recall is equivalent to few truly relevant results, out of which most are predicted right. On the other hand, having high recall and low precision is equivalent to returning many results, out of which few are guessed correctly from all the aimed ones [4].

Balance between recall and precision. Normally, recall is obtained at the cost of precision. As literature states it, a good approximate similarity search algorithm must achieve a balance between precision and recall. That means both proving early retrieval of the k good neighbors for a given query (good recall) and guaranteeing that a fair number of the retrieved neighbors are actually among the k seeked ones (good precision) [48]. For instance, only by using the notion of precision, we cannot capture how well an algorithm performs. This is because an approximation result set can contain only right guesses, which is equivalent to $p = 1$, but these can just as well be very few of the actual nearest neighbors. In this case, even if high, the precision becomes meaningless. Figure 6.1 shows an example of this. Given query q, the approximation set $S_{approx} = 9$ and the actual nearest neighbors set S represented by the green nodes, precision has a high value (i.e. $p = 1$). Nevertheless, recall has a much lower value (i.e. $r = \frac{1}{6}$).

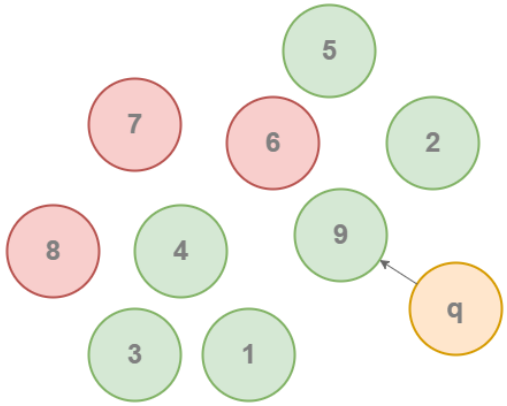


Figure 6.1: Green nodes represent the actual 6 nearest neighbors of query point q, represented with orange. The arrow from q indicates that data point 9 is the only one returned by the approximation algorithm.

F_1-score. In order to achieve the balance between precision and recall, we use F_1-score [18]. F_1-score is defined as the harmonic mean between the two:

$$F_1 = \frac{2 \cdot p \cdot r}{p + r} \tag{6.5}$$

The bigger the gap between precision and recall, the worse the F_1-score becomes. The closer precision and recall are, the better the F_1-score is as well.

6.2.2 Evaluation Types

This subsection analyzes different evaluation approaches and draws a parallel between them in terms of quality metrics.

6.2.2.1 Evaluation type I: Constant Hamming Ball

The original SH paper [46] proposes an evaluation type which uses a constant Hamming ball. We refer to this as *evaluation type I*.

Steps. Given a query q, the task is to approximate its k nearest neighbors in Hamming space, considering the k actual nearest neighbors in Euclidean space. This is equivalent to achieving a good mapping between the two spaces by preserving pairwise distances among data points. Each of these is represented by a different coordinate system/basis (see subsection 3.3.1). Thus, a Euclidean distance $d_{\text{Eucl}} = 1$ is not necessarily the same as a Hamming distance of the same value $d_{\text{Hamm}} = 1$. With this in mind, *evaluation type I* involves the following steps:

1. Calculate d_ball_{Eucl}, where d_ball_{Eucl} is the average Euclidean distance within which the k nearest neighbors of any data point x_i are found, such that $x_i \in X_{\text{train}}$ (we refer to the training set as X_{train});

2. For any data point $x_q \in X_{\text{test}}$ (we refer to the testing set as X_{test}), we fix d_ball_{Eucl} as the radius of a ball around x_q and determine, for all points $x_i \in X_{\text{train}}$, whether x_i is within the ball area of x_q (see Figure 6.2);

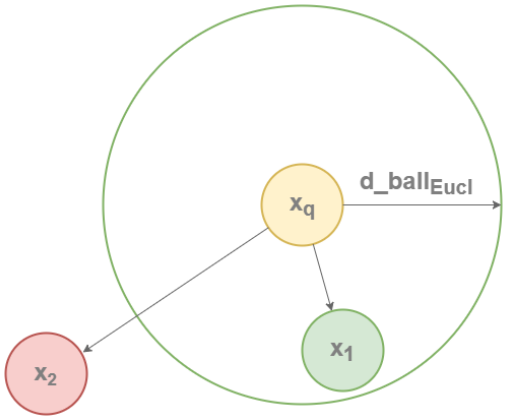


Figure 6.2: d_ball_{Eucl} around query point x_q, such that data points like x_1 are considered near neighbors for x_q, while data points like x_2, placed outside the ball radius, are not.

3. Calculate a matrix $w\_true\_testing\_training$ where an entry $(q_{\text{test}}, i_{\text{train}})$ stores $true$ if i_{train} is considered one of the k nearest neighbors of q_{test} according to the d_ball_{Eucl} definition above (see 2) and $false$ otherwise;

4. After compressing both the training and testing set and obtaining their hashcodes, calculate a matrix d_hamm whose entry $(q_{\text{test}}, i_{\text{train}})$ stores the Hamming distance between q_{test} and i_{train}.

Calculating quality metrics. The original SH paper [46] calculates precision and recall for a static Hamming ball d_ball_{Hamm}, which is gradually increased up to a maximum chosen value. In order to achieve this, they use the following variables:

1. $total\_good\_pairs$: Equals the sum of $true$ values in $w\_true\_testing\_training$ and defines the ground-truth (i.e. all the relevant nearest neighbors calculated in Euclidean space, for all the query points $x_q \in X_{\text{test}}$);

2. *retrieved_pairs*: Defines all the items considered nearest neighbors in Hamming space (i.e. within a Hamming distance of d_ball_{Hamm}), for all the query points $x_q \in X_{\text{test}}$;

3. *retrieved_good_pairs*: Defines all the items considered nearest neighbors in both Euclidean and Hamming space, for all the query points $x_q \in X_{\text{test}}$.

With these variables, precision and recall become:

$$p = \frac{\text{retrieved\_good\_pairs}}{\text{retrieved\_pairs}} \tag{6.6}$$

and

$$r = \frac{\text{retrieved\_good\_pairs}}{\text{total\_good\_pairs}} \tag{6.7}$$

Figure 6.3 shows an example of evaluation where $X_{\text{train}} = \{x_1, x_2, x_3, x_4\}$ and $X_{\text{test}} = \{x_q\}$. We consider two possible approximation outcomes: calling $SH_1(X_{\text{train}}, X_{\text{test}})$ in one case, and calling $SH_2(X_{\text{train}}, X_{\text{test}})$ in the other case. The top square represents the ground-truth, where x_1 and x_2 are the $k = 2$ nearest neighbors of query point x_q. If we run the algorithm $SH_1(X_{\text{train}}, X_{\text{test}})$ with a Hamming ball fixed to $d\_ball_{\text{Hamm}} = 1$, by looking at the outcome, the mapping from Euclidean to Hamming space seems perfect, since $p = r = 1$. However, with the same d_ball_{Hamm}, running the algorithm $SH_2(X_{\text{train}}, X_{\text{test}})$ performs worse, as $p = \frac{1}{3}$ and $r = \frac{1}{2}$. By increasing the Hamming ball to $d\_ball_{\text{Hamm}} = 2$, we manage to capture one more near neighbor, which increases both quality metrics to $p = \frac{1}{2}$ and $r = 1$.

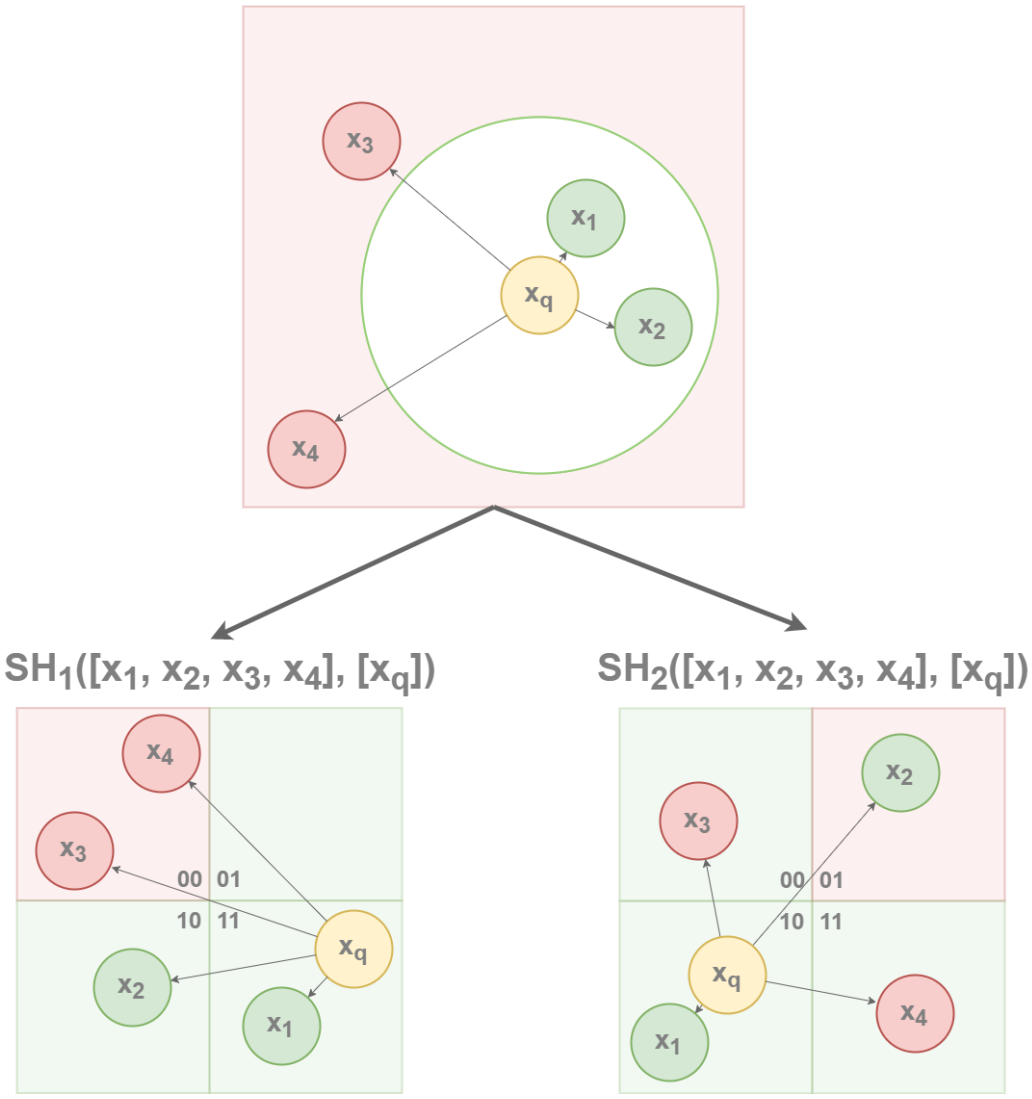


Figure 6.3: The top square represents the ground-truth, while the resulting squares show two possible outcomes of an approximation algorithm. The arrow between any two nodes indicates the distance between them (i.e. Euclidean for the top square and Hamming for the two possible outcomes). Each of the two bottom squares is represented by four buckets, all labeled with binary codes.

For a given query point x_q, if the Euclidean ball approaches the Hamming

ball and pairwise distances among data points are well preserved, the k actual nearest neighbors (the ground-truth) are likely to be found in Hamming space as well.

This type of evaluation becomes sensitive to the proposed distance ball. In fact, it is very different from a precise kind of evaluation, where both the ground-truth and the approximation set are clearly defined, and afterwards compared. Such an evaluation is proposed in other papers [31], where recall is defined as

$$r = \frac{\text{retrieved\_good\_pairs}}{\text{k}} \tag{6.8}$$

such that k indicates the number of nearest neighbors we search for. Compared to this approach, in our case, the $total\_good\_pairs$ is not necessarily equivalent to k. Instead, it very much depends on the Euclidean ball. What we know is only that it equals k, on average.

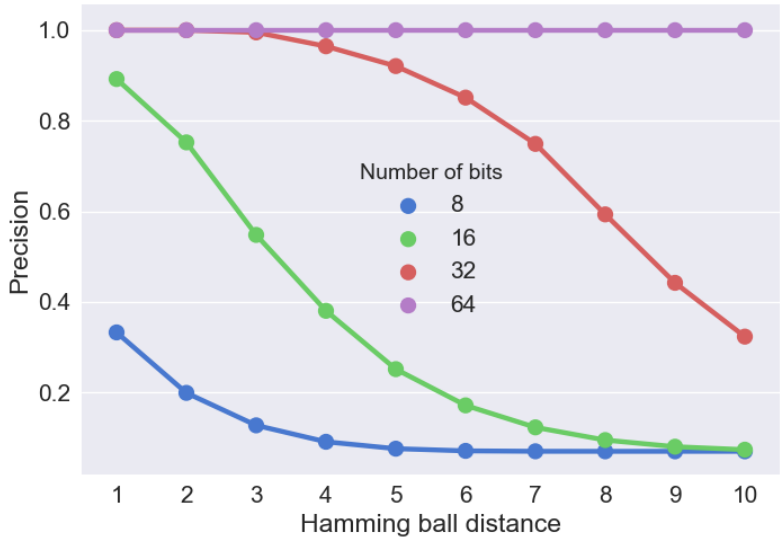


Figure 6.4: Precision scores over MNIST dataset, where we search for $k = 50$ nearest neighbors. The training set size is $n = 1000$, whereas the testing set size is $m = 100$. The x-axis indicates different values of the Hamming ball d_ball_{Hamm}, while the y-axis shows precision values. We plot results for several models (i.e. $num\_bits \in \{8, 16, 32, 64\}$).

Figure 6.4 and Figure 6.5 show how precision and recall scale with the Hamming ball d_ball_{Hamm} for different numbers of bits. As the Hamming ball increases, precision decreases, because by extending the searching area, there is a bigger chance of capturing more undesired items among all the retrieved samples. On the other hand, recall scales up with the Hamming ball d_ball_{Hamm}, since by enlarging the searching space, it is more likely to find more actual near neighbors.

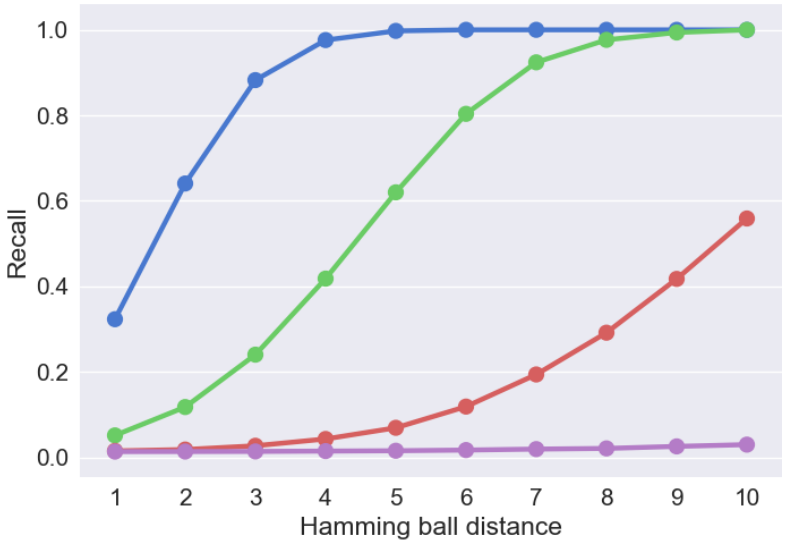


Figure 6.5: Recall scores over MNIST dataset, where we search for $k = 50$ nearest neighbors. The training set size is $n = 1000$, whereas the testing set size is $m = 100$. The x-axis indicates different values of the Hamming ball d_ball_{Hamm}, while the y-axis shows recall values. We plot results for several models (i.e. $num\_bits \in \{8, 16, 32, 64\}$).

Moreover, recall increases much faster for a smaller number of bits than for a bigger number of bits, because an area of Hamming ball $d\_ball_{\text{Hamm}} = 1$ has a different meaning for each of the two models (i.e. it captures more data points in a space where the number of bits is smaller). In other words, the Hamming ball as an absolute value represents a different unit measure for models, depending on the number of representative bits. Precision and recall scores *meet* at an earlier stage of the Hamming ball for a model of a smaller number of bits.

6.2.2.2 Evaluation type II: Averaged Hamming Ball

Motivation. Given the sensitivity to the Hamming ball d_ball_{Hamm} of the evaluation type proposed in the original SH paper [46], comparing results with a fixed Hamming ball might not always indicate fairness of the scores. For instance, reporting a precision of $p = 1$ in Figure 6.4 for the model of 64 bits within a Hamming ball $d\_ball_{\text{Hamm}} = 2$ only tells us that the query points are simply far away from any other points, but nothing else about how the algorithm actually performs. This is also supported by a recall value $r = 0$.

For the above stated reason, we propose an evaluation type where we consider F_1-score to be a much stronger performance indicator (see subsection 6.2.1). For any dataset, we expect F_1 to increase with the Hamming ball up to a specific point where it reaches a maximum $\max_{F_1}$, after which it is expected to decrease again. As such, $\max_{F_1}$ represents the point where the algorithm achieves the best balance between precision and recall, which happens to be when they are both weighed equally. Figure 6.6 exemplifies this idea for four real datasets (see subsection 6.1.1).

Approach. The evaluation type we propose follows the exact same steps as Evaluation type I (see subsubsection 6.2.2.1), but differs from it in one perspective. Instead of calculating precision and recall up to a maximum chosen Hamming ball distance, it determines the approximate Hamming ball for which F_1-score reaches its maximum, based on a number of assumptions.

Let us assume the algorithm achieves the locality-sensitive property (see chapter 1) in its mapping and produces a good approximation, such that pairwise distances among points from the Euclidean space are well preserved in Hamming space as well. That means the best F_1-score must be obtained for a Hamming ball which contains all the actual nearest neighbors included by the Euclidean ball to start with. Since the Euclidean ball represents the average

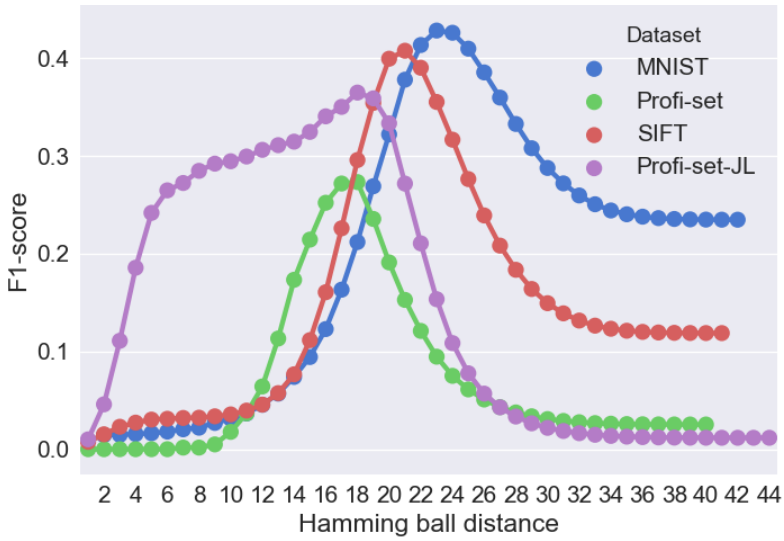


Figure 6.6: F_1-score curves as a function of the Hamming ball distance d_ball_{Hamm} on MNIST, Profi-set, SIFT and Profi-set-JL, where training set size is $n = 1000$, testing set size is $m = 100$, the number of bits for encoding is $num\_bits = 64$ and the number of seeked nearest neighbors is $k = 100$.

Euclidean distance within which we find, on average, the k nearest neighbors for any query point x_q in the *testing set* X_{test}, the Hamming ball we aim to determine must have the exact same meaning as the Euclidean ball, but in Hamming space instead. As a result, we calculate d_ball_{Hamm} as an average distance over the training set (more precisely, over the compressed version of the training set, which consists of hashcodes). As assumed, it turns out the best F_1-score is obtained at a Hamming ball equivalent to the average Hamming ball d_ball_{Hamm}.

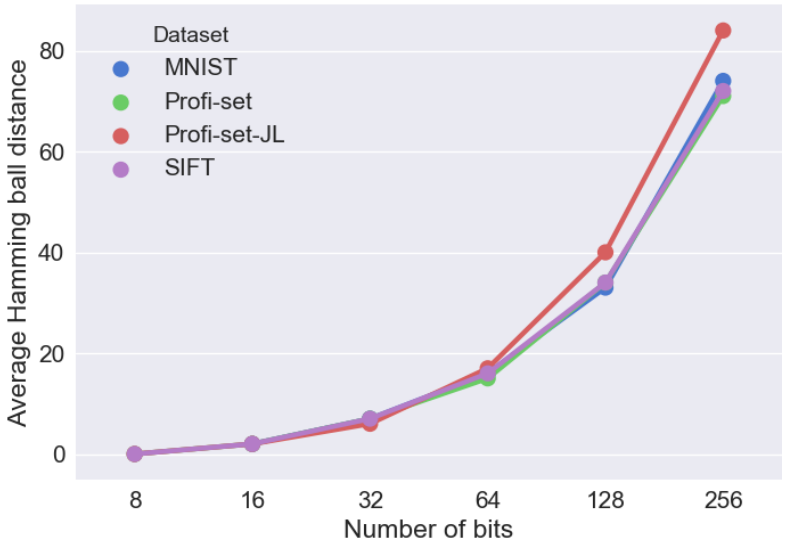


Figure 6.7: The average Hamming ball for which optimal F_1-score is achieved for different number of bits and across different datasets (MNIST, Profi-set, SIFT and Profi-set-JL). We used the following parameters: training set size $n = 20,000$, testing set size $m = 2000$, number of seeked nearest neighbors $k = 100$.

Importance. Even if it implies an extra computation, this type of evaluation provides the user with the possibility to select what Hamming ball to fetch results for, depending on the number of bits chosen for encoding. This dependency is shown in Figure 6.7 for different real datasets, across which Vanilla SH seems to be consistent.

6.3 Parameter Dependencies

This section explores dependencies between different parameters with regards to performance.

Performance in terms of the number of bits. When mapping from

a high-dimensional and continuous space to a space of dimensionality k, it is impossible to fully preserve pairwise distances. However, we expect performance to scale up with the number of bits, since more bits allow us to preserve more of the original information of the data. Figure 6.8 reflects this consistently across all the real datasets.

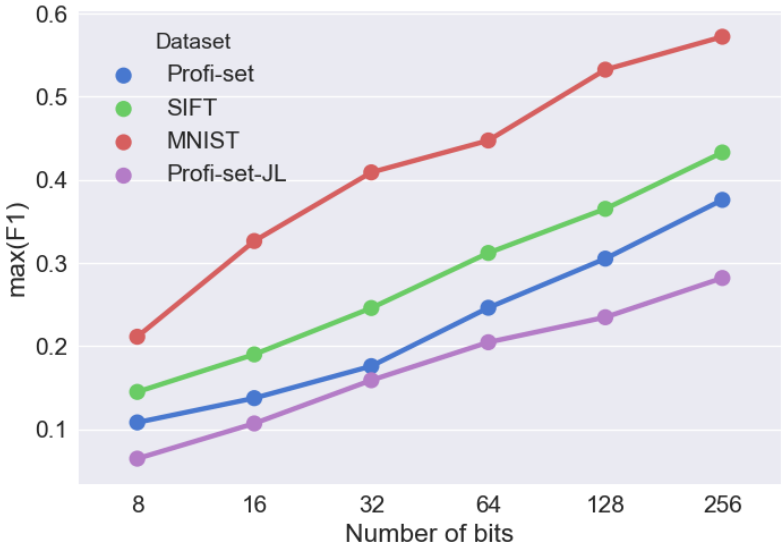


Figure 6.8: Performance scales up with the number of bits. The x-axis indicates the number of bits, while the y-axis indicates performance in terms of the maximum obtained $F1$-score of all the Hamming ball distances. We used the following parameters: testing size $m = 2000$, training size $n = 20,000$ and $k = 100$.

Performance as a function of k. The algorithm performs differently for different values of k (i.e the number of nearest neighbors we search for). This is shown in Figure 6.9. When encoding to a bigger number of bits, we use more buckets than when encoding to a lower number of bits. However, with a bigger number of bits, the same number of data points are distributed to more buckets. That means we end up having fewer items in each bucket on average and therefore, it is easier to find a smaller number of nearest

neighbors.

For example, seeking for $k = 100$ rather than $k = 10$ nearest neighbors when the number of bits is $num\_bits = 256$ is harder, since we search through more buckets and further away from our query point. This slowly transitions towards the opposite case, where having a model of smaller number of bits is equivalent to each bucket containing more points. In this case, searching for $k = 100$ rather than $k = 10$ becomes easier, because there is no need to extend the search across many more buckets in order to find the aimed neighbors. These two opposite cases have a common point, though, which occurs when the number of bits is $num\_bits = 64$. There, it becomes just as difficult to find $k = 10$ neighbors as it is to find $k = 100$ neighbors.

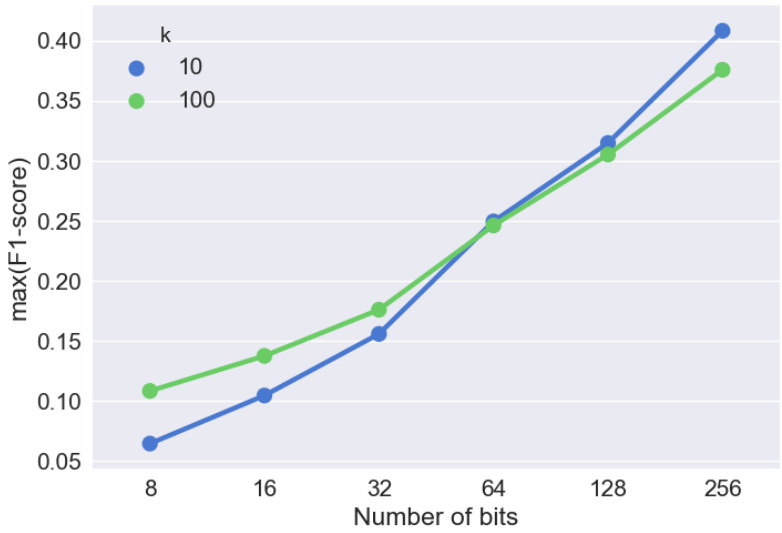


Figure 6.9: The F_1-score depends on the number of nearest neighbors we search for across different models. The x-axis shows the number of bits, while the y-axis shows the best obtained F_1-score. Results originate from the Profi-set, where we used the following parameters: testing size $m = 2000$ and training size $n = 20,000$.

Performance in terms of the dataset dimensionality. The dimension-

ality of the dataset affects the quality of a model. Given a constant training and testing size, performance decreases as dimensionality increases. This is exemplified in Figure 6.10. The experiment is conducted with different models ($num\_bits \in \{16, 32, 64\}$) on a highly clustered dataset, where we vary dimensionality $d \in \{2, 4, 6, 8, 10\}$.

We construct the clustered dataset artificially, by using a Python library (i.e. *scikit-learn* [32]), with the following parameters: dimensionality ($d \in \{2, 4, 6, 8, 10\}$), number of samples (training size $n = 1000$ and testing size $m = 100$), number of clusters ($num\_cl = 5$), standard deviation of each cluster ($std\_dev = 1$), dimensions' range ($[-100, 100]$). The clusters describe a Gaussian distribution and have centers which are chosen uniformly at random.

The smaller the dimensionality of the datatset, the easier it becomes to fit a model and obtain better F_1-score. This is because increased dimensionality adds complexity and involves more sparseness of the data. As a result, it is harder to fit a model when dealing with a bigger number of dimensions.

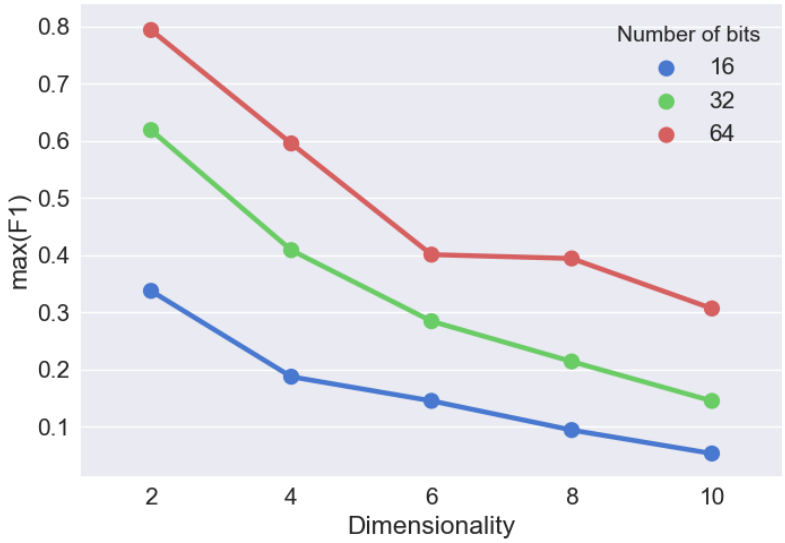


Figure 6.10: Performance decreases as dimensionality increases. We conducted the experiment with Vanilla SH on a highly clustered dataset, which consists of 5 blobs randomly spaced out and of the same standard deviation. We used the following parameters: training size $n = 1000$, testing size $m = 100$ and $k = 10$.

Performance affected by the training size. The training size can influence the performance to some extent. As we sample more points from the original data, F_1-score has a tendency to decrease for all datasets, except for MNIST. This is shown in Figure 6.11, for a model of number of bits $num\_bits = 32$. Profi-set and Profi-set JL have very similar behavior, as they both slightly decrease and then start increasing again when approaching a bigger training size. On the other hand, we expect SIFT to also increase in performance for a training size bigger than $50,000$. MNIST seems to scale up from the beginning. This might be related to the original dataset size, which, for MNIST, is much smaller than for all the other datasets. Therefore, when sampling $50,000$ points, it is very likely that the training and the testing set highly intersect. This could also explain why MNIST generally performs best, compared to all the other datasets.

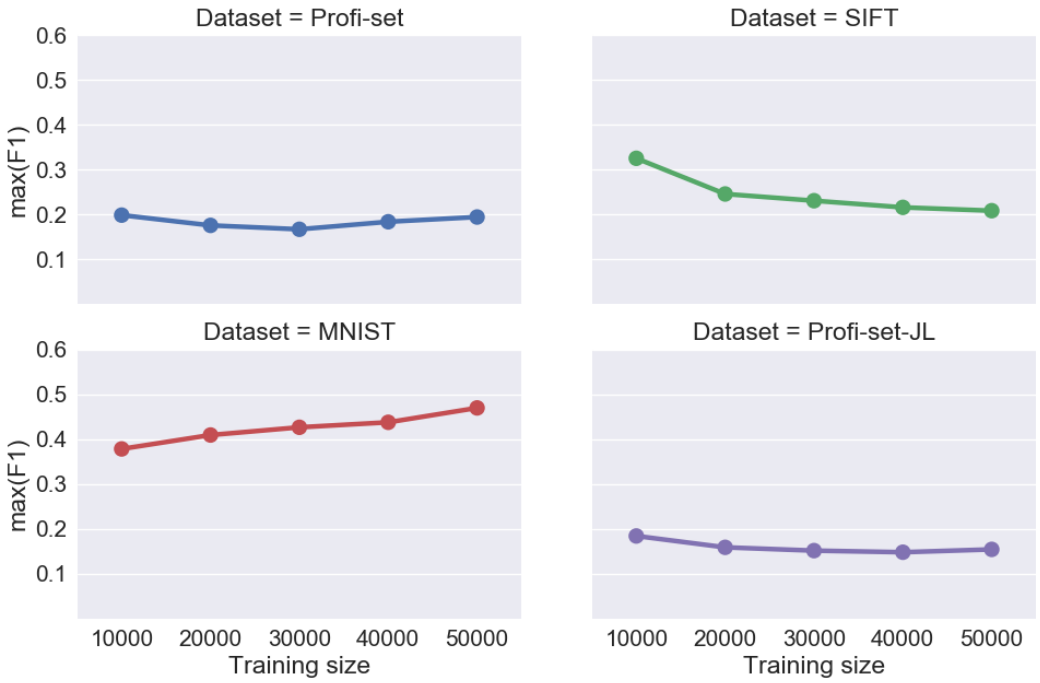


Figure 6.11: Performance is affected by the training size across different datasets. The x-axis shows the training size, while the y-axis shows the best obtained F_1-score. Results are conducted on each of the four real datasets, with a number of nearest neighbors $k = 100$ and a number of bits $num\_bits = 32$.

6.4 Performance of Vanilla SH

Performance results' reconstruction. We aim to have as a starting point a reconstruction of scores on the MNIST dataset, as presented in 'Learning to Hash with Binary Reconstructive Embeddings' [25]. Figure 6.12 demonstrates how precision within a Hamming ball $d\_ball_{Hamm} <= 3$ scales up with the number of bits. It turns out our scores are very much comparable to the ones in the paper. Based on this, we proceed with testing Vanilla SH later on, with our own parameters.

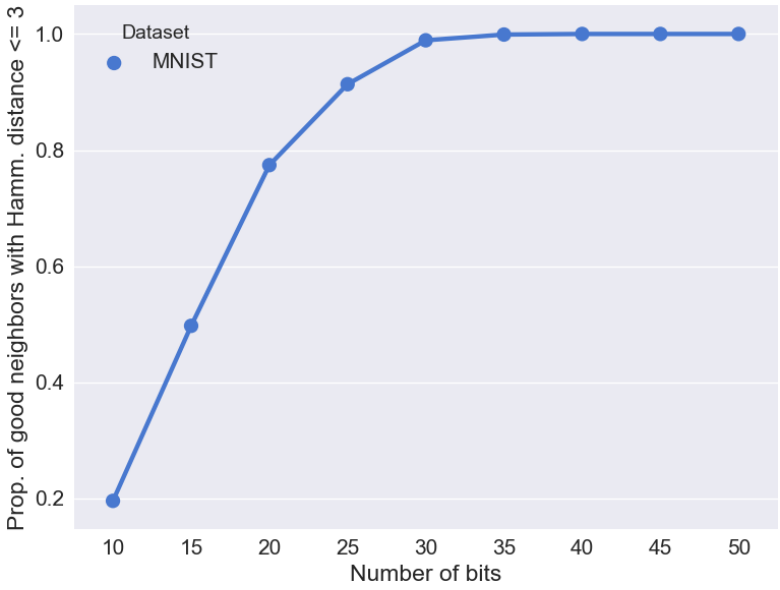


Figure 6.12: Precision of Vanilla SH run on MNIST, with the following parameters: training size $n = 1000$, $k = 50$, $d\_ball_{\text{Ham}} \leq 3$.

Performance on real datasets. As expected, performance grows with the number of bits across all the datasets, as shown in Figure 6.13. However, it seems that Vanilla SH performs much better on MNIST than on all the other datasets. The initial embedding might be a factor contributing to this difference: SIFT, Profi-set and Profi-set-JL are all represented as image descriptors, whereas MNIST consists of normalized images in pixel representation. On the other hand, it makes sense that Profi-set performs worse than SIFT and MNIST, since it is of much bigger dimensionality ($d = 4096$, compared to MNIST of $d = 784$ and SIFT of $d = 128$). Additionally, it is not surprising that Profi-set-JL performs worse than the Profi-set, since it is a version of Profi-set reduced in dimensionality with Johnson–Lindenstrauss transform [24]. In fact, this means the dataset is applied dimensionality reduction twice: once before being fed to Vanilla SH and afterwards during the first phase of the algorithm, when performing PCA. As such, it seems

that more of the dataset information gets lost, which leads to the fact that Profi-set-JL performs worst.

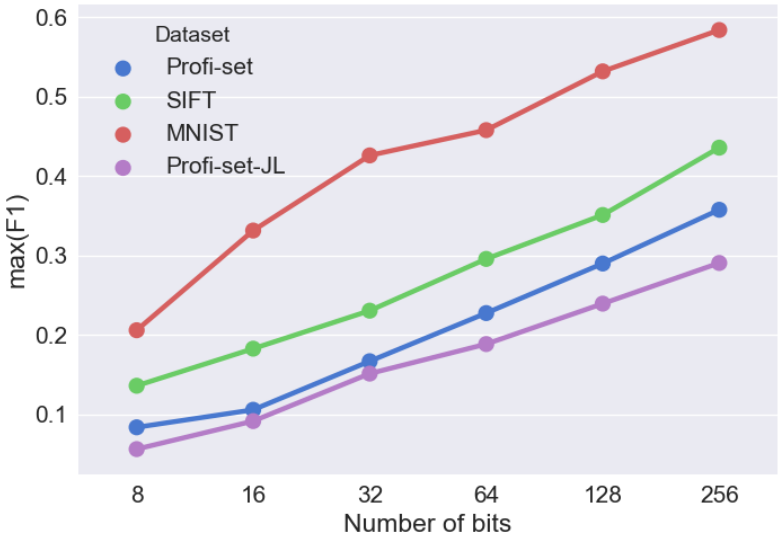


Figure 6.13: Performance of Vanilla SH on different real datasets. The x-axis shows the number of bits, while the y-axis shows the best obtained F_1-score. We used the following parameters: testing size $m = 3000$, training size $n = 30,000$, $k = 100$.

Furthermore, the larger the variance across a dimension, the more cuts Vanilla SH makes on it. Along these lines, the principal components of most cuts are the most valuable ones (i.e. the largest ones, preserving most of the information in the original data). Since Profi-set-JL has the smallest number of maximum cuts on the principal components of all datasets, as shown in Figure 6.14, this could be another explanation why it yields the worst performance, compared to all the other datasets.

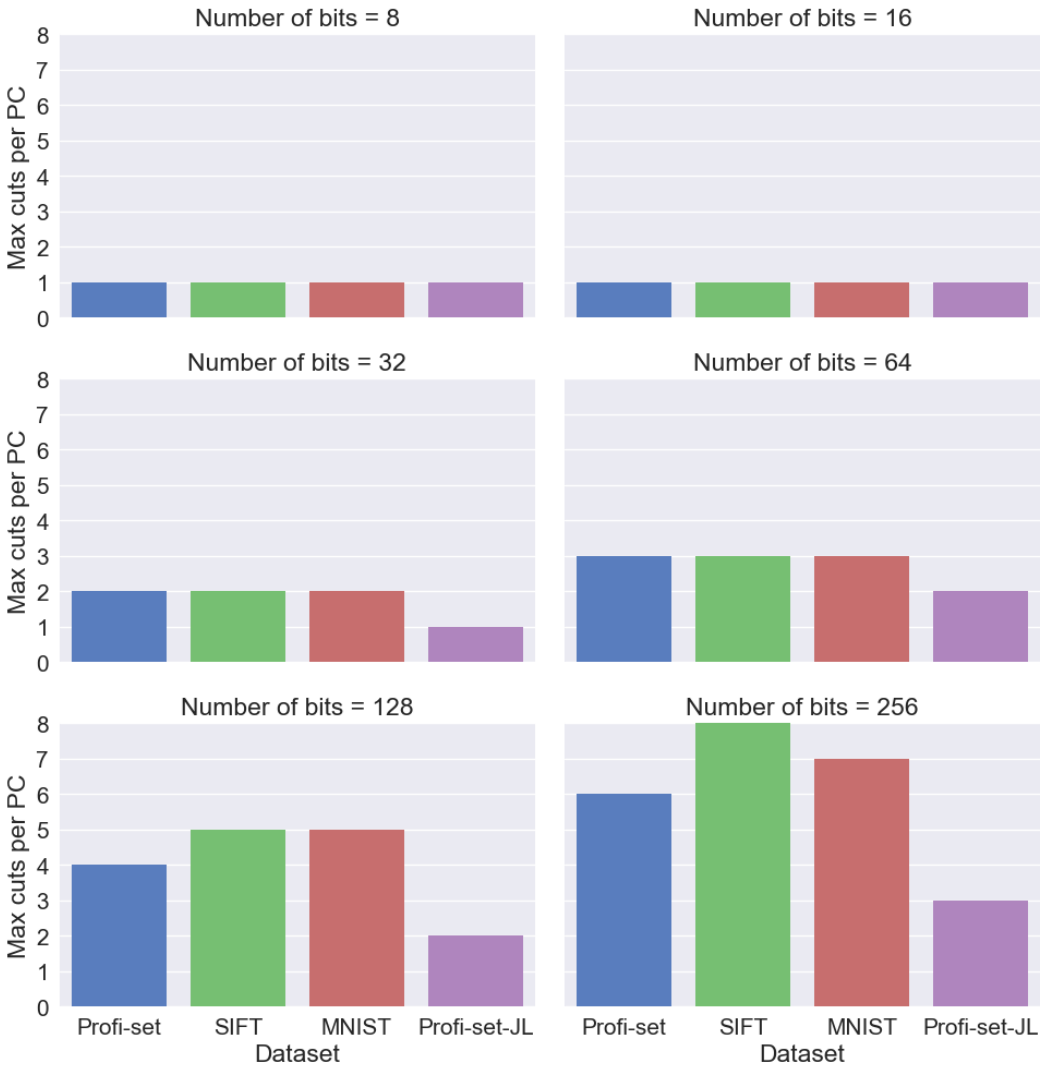


Figure 6.14: Maximum number of cuts across principal components for different models and on different datasets. The x-axis indicates dataset names, while the y-axis shows the maximum number of cuts made on principal components.

6.5 Performance Comparison among SH Variations

This section aims to draw a parallel between the different proposed SH variations in terms of performance and provide an interpretation of the results.

6.5.1 Best Performance across SH Variations

Stating the best heuristic. Figure 6.15 and Figure 6.16 show performance in terms of F_1-score of all the SH variants and across different models (i.e. $num\_bits = \{8, 16, 32, 64, 128, 256\}$). Among these, PCdominancebymodesorder SH performs best overall.

Heuristics' bias. The distinction in performance is much clearer for the Profi-set than for other datasets, considering that these heuristics have been developed on it. As such, their good performance on the Profi-set is not surprising. Additionally, even if they are probably more biased towards the Profi-set, their relatively consistent performance across the other real datasets indicates that they provide an advantage in general.

Clear improvement over Vanilla SH. We notice how PCdominancebymodesorder SH achieves better F_1-score than any other heuristic on the Profi-set, while significantly improving over Vanilla SH across all models. For a number of bits equal to $num\_bits = 64$, PCdominancebymodesorder SH reaches an F_1-score bigger than Vanilla SH's with approximately 15% absolute value.

General tendency of performance decrease above a certain number of bits. When it comes to both SIFT and MNIST, all the SH variants' F_1-scores increase with the number of bits, which is also expected. However,

this happens up to a certain point (i.e. $num\_bits = \{64, 128, 256\}$), where all of them seem to decrease in performance, including our best heuristic (i.e. PCdominancebymodesorder SH). This general behavior across heuristics might be caused by the structure of the datasets, since it occurs for SIFT and MNIST, but not for Profi-set and Profi-set-JL. The different structure of the datasets is equivalent to different principal component axes for each. It might be that MNIST and SIFT are more similar in terms of their axes of variance. Let us remember the fact that the longer an axis, the more cuts it involves. Furthermore, certain principal components might be better to cut than others. Thus, we conclude that for both SIFT and MNIST, the decrease in performance for models of size $num\_bits = \{64, 128, 256\}$ occurs because the algorithm does not find a good balance between which principal components are cut (i.e. how important the bits appended to the hashcodes are) and how many times they are cut (i.e. the number of bits extracted from each principal component). This balance is most likely better achieved for the Profi-set and Profi-set-JL, since F_1-score grows more linearly instead of suddenly decreasing above a specific number of bits.

PCdominancebymodesorder SH performs worse than Vanilla SH above a certain hashcode length. Interestingly, PCdominancebymodesorder SH performs worse than Vanilla SH on SIFT, starting with a model of $num\_bits = 256$. We must recall that SIFT has initial dimensionality $d = 128$. However, we aim to encode to 256 bits (i.e. a hashcode two times longer than the dimensionality of the original dataset). It makes sense that Vanilla SH seems to scale up after $num\_bits = 256$, whereas PCdominancebymodesorder SH decreases, since Vanilla SH is more used for encoding a dataset to a bigger code length than its initial dimensionality, as most of the examples in the original SH paper suggest [46]. On the contrary, the purpose behind PCdominancebymodesorder SH, as well as behind any of the heuristics we propose is to achieve a good mapping with the smallest possible binary code length.

PCdominancebymodesorder SH performs worse than other heuristics below a certain hashcode length. Another corner case where PCdominancebymodesorder SH does worse than Vanilla SH and also worse than the other variants is for very small binary code lengths (e.g. $num\_bits = 8$ on MNIST and $num\_bits \leq 64$ on Profi-set-JL). This effect persists more on Profi-set-JL. After having lost a lot more information in comparison with all the other datasets (i.e. being already reduced in dimensionality before feeding it to the algorithm), Profi-set-JL most likely ends up having shorter variance axes than any other real dataset we use (see the maximum number of cuts on Profi-set-JL across all models in Figure 6.14). This implies that Profi-set-JL is the dataset closest to a uniform distribution of all datasets, as the standard deviation across variance axes is lowest here. As such, the more aggressive bit extraction approach adopted by PCdominancebymodesorder SH becomes less effective as well.

All in all, we still consider PCdominancebymodesorder SH the best heuristic overall, considering it fits to our problem definition more, regardless the mentioned corner cases where it performs worse than other variants. As noticed, besides the binary code length, the dataset used also has an impact on the performance.

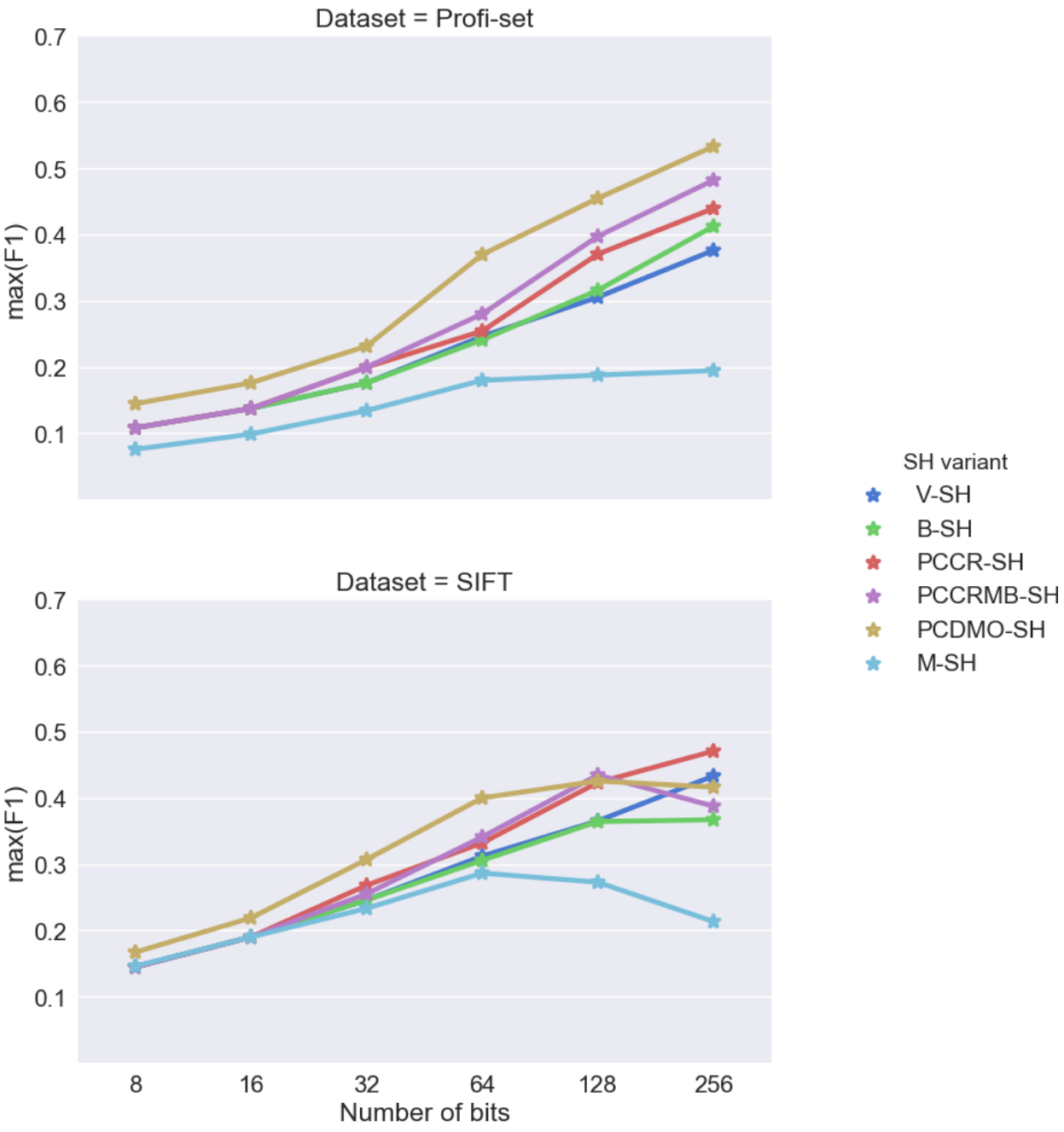


Figure 6.15: Performance in terms of F_1-score (indicated on the y-axis) of all the SH variants across different models (indicated on the x-axis), for the Profi-set and SIFT.

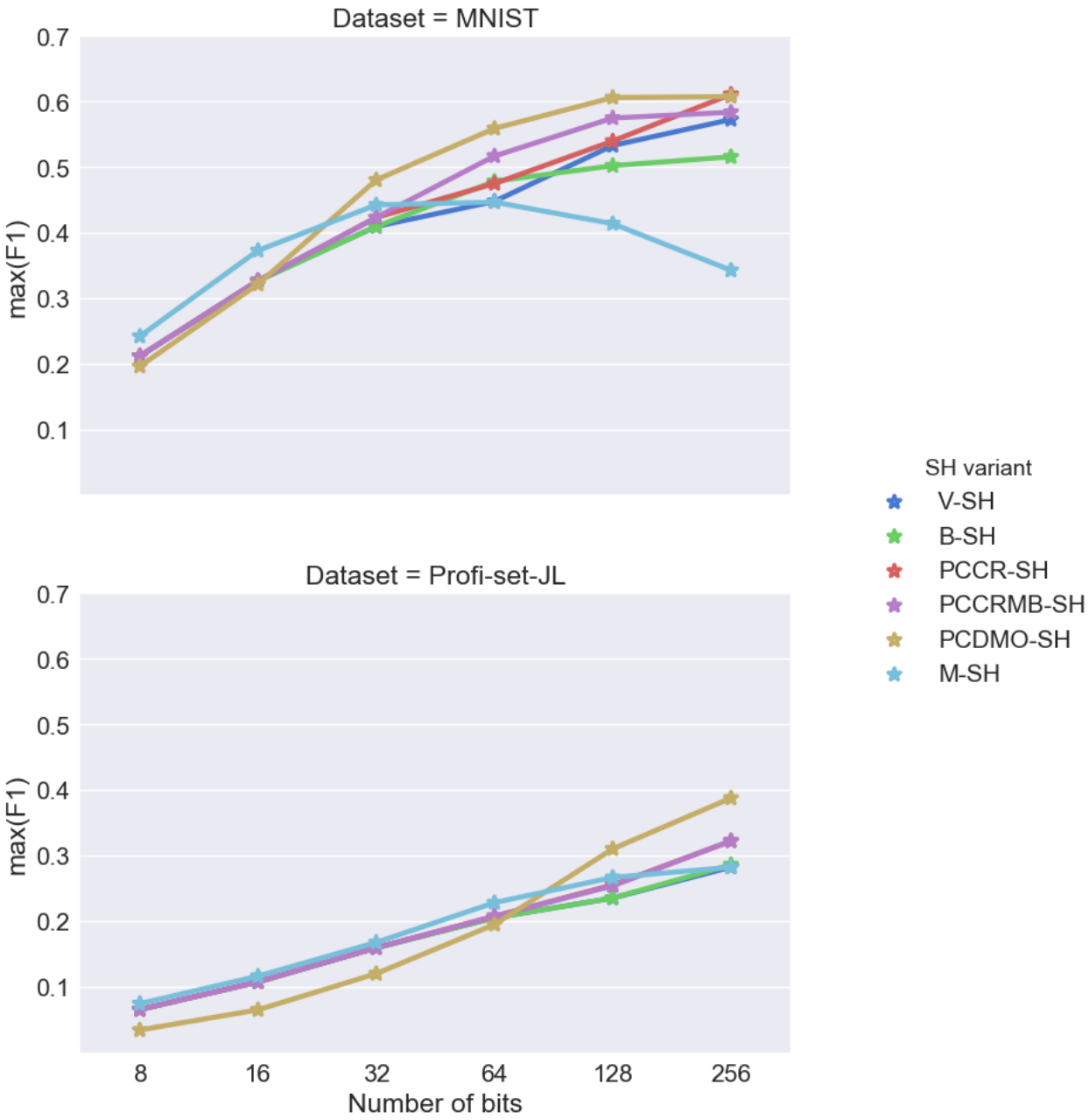


Figure 6.16: Performance in terms of F_1-score (indicated on the y-axis) of all the SH variants across different models (indicated on the x-axis), for MNIST and Profi-set-JL. Note that for Profi-set-JL, Vanilla SH and Balanced SH overlap, while the same occurs for PCcutrepeated SH and PCcutrepeated-multiplebits SH.

6.5.2 Behavior of SH Variations across Real Datasets

Based on Figure 6.15 and Figure 6.16, this subsection analyzes the performance of each heuristic and provides an interpretation of the results.

Balanced SH versus Vanilla SH. We consider Balanced SH the central heuristic among all the ones we propose, since it applies the main methods exploited by the other variants in a more aggressive fashion (see section 5.2).

Balanced SH behaves just as Vanilla SH up to 32 bits for the Profi-set, SIFT and MNIST and up to 128 bits for Profi-set-JL. Above 32 bits, it increases performance over Vanilla SH with approximately 4% in absolute value (i.e. from $F_1 = 0.37$ to $F_1 = 0.41$). However, while it still behaves as Vanilla SH above 128 on Profi-set-JL (see overlapping lines in Figure 6.16), it follows a significant decrease for SIFT and MNIST (i.e. 7% in absolute value for SIFT, from $F_1 = 0.43$ to $F_1 = 0.36$ and 6% in absolute value for MNIST, from $F_1 = 0.57$ to $F_1 = 0.51$).

The cases where the two perform identically are expected. This is because whenever our principal components are cut at most twice, they actually end up partitioning the space identically. If we look back to Figure 6.14, the Profi-set, SIFT and MNIST cut principal components maximum two times below $num\_bits = 32$, whereas Profi-set-JL does the same below $num\_bits = 128$.

Among the cases where the two differ, our expectations only fit with the increase of Balanced SH over Vanilla SH on the Profi-set. This is because in Balanced SH, we exploit some particularities of Vanilla SH which we expect to slightly increase performance (e.g. buckets' labeling, buckets' sizes). However, the results do not fully comply with our ideas. We think Balanced SH's decrease over Vanilla SH for MNIST and SIFT has to do with a possibly far from uniform distribution of the principal components scores. Since Balanced SH cuts the axes in equal buckets regardless the distribution of the

scores, a very clustered structure of the principal components can lead to many empty buckets (i.e. with no data points), as well as to some buckets containing way too many data points. This definitely hurts performance for Balanced SH, whereas Vanilla SH's unequal buckets and larger Hamming distance gap between some of the neighboring buckets (i.e. $d\_Hamm > 1$) seem to win in this case.

PCcutrepeated SH and PCcutrepeatedmltiplebitsSH versus Vanilla SH. While behaving almost identically for Profi-set-JL, PCcutrepeated SH and PCcutrepeatedmltiplebitsSH also perform similarly on the other datasets. Usually, they outperform Vanilla SH, but sometimes, they actually behave very closely to it (see PCcutrepeated SH for $n\_bits = 128$ on MNIST). Nevertheless, PCcutrepeatedmltiplebitsSH strongly follows the same tendency of decreasing in performance as the other proposed heuristics for SIFT, when the number of bits equals 256. This is the only case where PCcutrepeatedmltiplebitsSH behaves worse than Vanilla SH, following a decrease of 5% in absolute value, from $F_1 = 0.43$ to $F_1 = 0.38$.

For the particular case where one of these two variants does worse than Vanilla SH, we find the same explanation as previously discussed for all the proposed heuristics, as this applies to all and mostly becomes visible on SIFT (see subsection 6.5.1).

For most of the other cases, where the two outperform both Vanilla SH and Balanced SH being closely to each other, we believe this occurs due to their strategy of repeating specific bits by sometimes extracting them from the same principal components, instead of getting them from new principal components. Thus, the resulted effect consists of weighing some bits more than others. However, this is equivalent to valuing some cuts in the underlying graph of the given data more than others. In other words, if a bit is important enough such that it divides two well defined clusters in the dataset,

weighing it more rather than using the same weight of importance for every bit in the hashcode can lead to better performance. As such, it seems that both PCcutrepeated SH and PCcutrepeatedmltiplebitsSH manage to assign more importance to the right bits, since they perform better than Vanilla SH and Balanced SH overall. While the former is more subtle, the latter applies this effect more aggressively and seems to generally yield better performance.

Median SH versus Vanilla SH. Median SH is the heuristic which performs worst overall. While this is very consistent for the Profi-set and SIFT (see Figure 6.15), it is partially applied for MNIST and holds even less for Profi-set-JL (see Figure 6.16). For instance, on the Profi-set and SIFT, for a model of $n\_bits = 128$, Median SH decreases approximately 10% in absolute value when compared to Vanilla SH. This implies Median SH is even worse than the other heuristics. The same significant drop holds for MNIST when using a bigger number of bits (e.g. $n\_bits \in \{128, 256\}$).

Nonetheless, Median SH does not always perform worse. It actually performs best for a short bit representation on MNIST (i.e. $n\_bits \in \{8, 16\}$), achieving an approximate increase in F_1-score of 4% in absolute value. Additionally, on Profi-set-JL it either does the same as Vanilla SH or even better (i.e. $n\_bits \in \{64, 128\}$), while never doing worst.

We believe the reason why Median SH performs worst overall is because of a less fortunate space partitioning. It ends up splitting clusters too much, instead of cutting the space between them. As a result, instead of hashing most of the points situated in a well-defined cluster in Euclidean space to the same bucket, a portion of the points in such a cluster get undesirably hashed to a bucket together with points from a different cluster. This is equivalent to a bad partitioning and mapping of points, which can explain the poor performance of Median SH in Figure 6.15.

For the cases when Median SH does not behave as Balanced SH, this can

be an indication that the principal component scores are far from uniformly distributed. Notice how for SIFT, MNIST and Profi-set-JL, Median SH starts differentiating most from Balanced SH above $num\_bits \in \{32, 64\}$ bits. This makes sense, as such: since principal component axes are divided more than 2-3 times (see Figure 6.14) and the longer the axes, the more cuts they get, then these axes might indicate more structure in the data.

PCdominancebymodesorder SH versus Vanilla SH. As stated in subsection 6.5.1, PCdominancebymodesorder SH achieves a significant F_1-score increase compared to Vanilla SH, for some parameters. This is specifically visible on the Profi-set, SIFT and MNIST for a model of number of bits $num\_bits \in \{64, 128\}$.

We claim this big increase in performance is due to the strategy of prioritizing bits by their importance in terms of eigenvalues, as well as applying a similar bit repetition effect, as PCcutrepeated SH and PCcutrepeatedmultiplebits SH. Additionally, incrementing the number of bits extracted from each principal component unlike all the other heuristics contributes to this improvement. This, combined with the much more desirable selection of bits from the right principal components, leads to the best performace overall.

However, for models of small number of bits (i.e. $num\_bits \leq 64$), this method performs worst on Profi-set-JL, while transitioning to performing best for models of big number of bits (i.e. $num\_bits \geq 64$). The bad case might be caused by the fact that bits (most likely too many bits) end up being extracted from the same principal components and filling the whole hashcode. Later on, PCdominancebymodesorder SH's strategy starts improving remarkably, since the hashcode length increases and allows for more bits defining good cuts over the principal components. After all, the heuristics rely on achieving a good balance between which principal components to extract bits from and how many bits to actually extract from them.

6.5.3 Hamming Ball Size across SH Variations

In this subsection, we investigate how the SH variants differ in terms of finding the k nearest neighbors within the Hamming ball.

As shown in Figure 6.17 and Figure 6.18, Median SH finds the k nearest neighbors within a very big Hamming distance, whereas PCdominancebymodesorder SH finds them within a much smaller radius. The other variants seem to behave similarly.

We claim that it is much better to find the k nearest neighbors at a smaller distance, as PCdominancebymodesorder SH does, since this is equivalent to mapping them close together in Hamming space. In contrast to this, mapping them further away (i.e. as Median SH does) means using a different scale of placing them in Hamming space. However, this creates a bigger room between the queries and their near neighbors in Hamming space. Therefore, the chance of hashing data points wrong within that room is also bigger. In other words, the more items we hash, the bigger the chance for mapping them wrong within the Hamming ball which also contains the actual nearest neighbors. This can hurt precision a lot and is actually also supported by the results: not only does PCdominancebymodesorder SH find the k nearest neighbors within a small Hamming ball, but it also performs best overall (and the opposite holds for Median SH).

All in all, a large Hamming ball which contains all k nearest neighbors might not matter much for our application, but it might make a difference in others, as larger Hamming searches can be expensive.

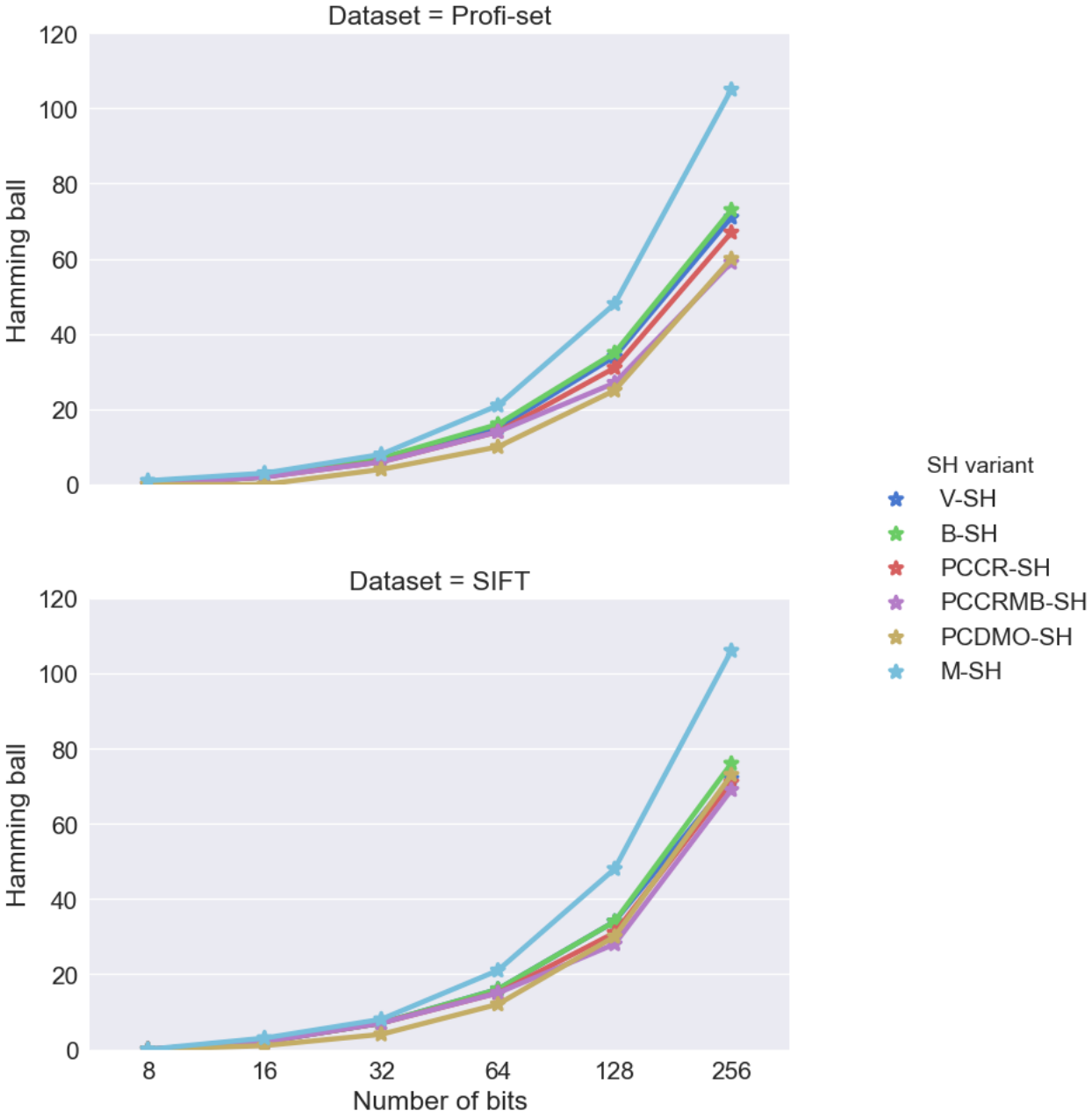


Figure 6.17: The Hamming ball values (y-axis) for which the maximum F_1-score is obtained across several models (x-axis), for Profi-set and SIFT.

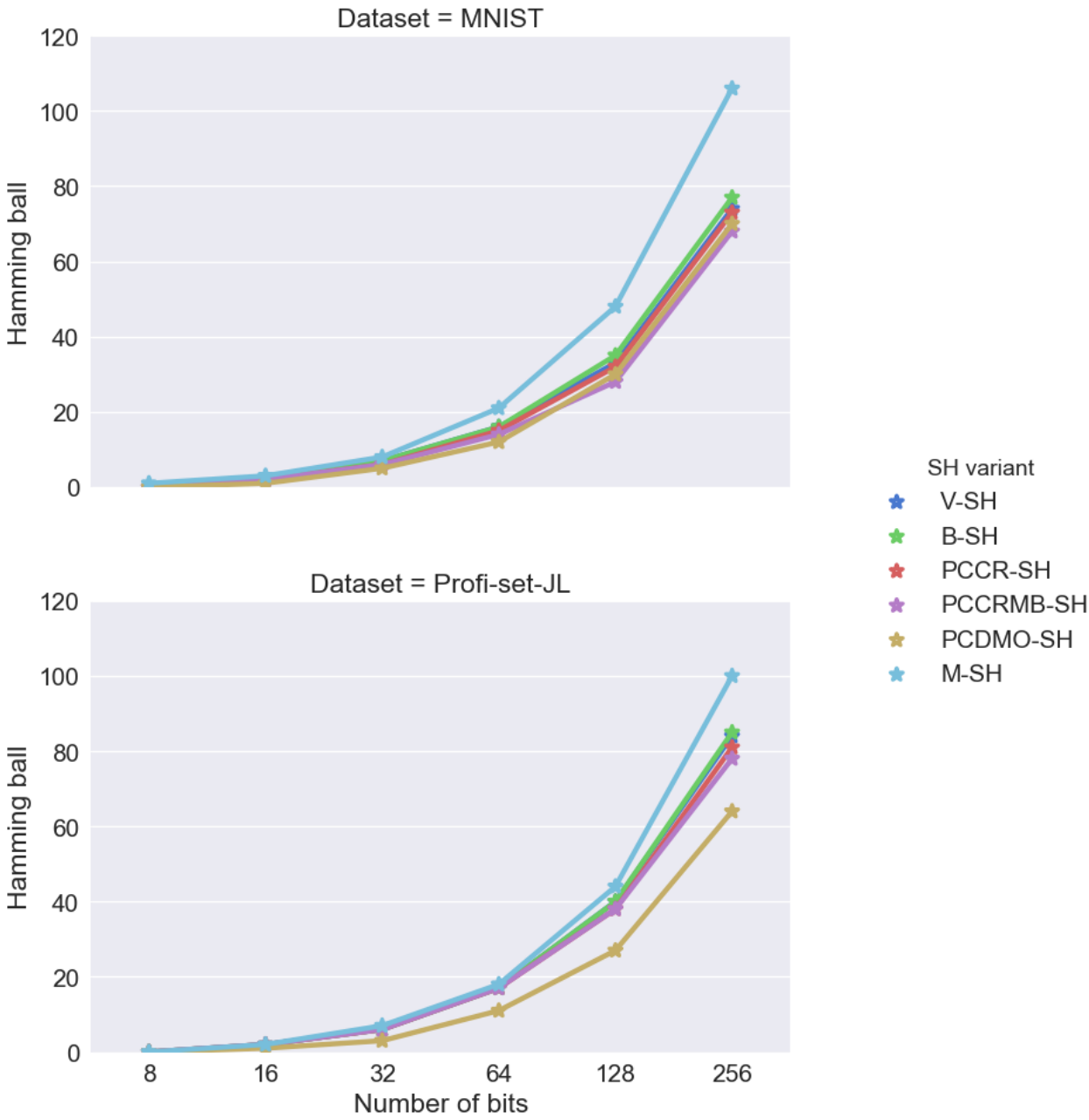


Figure 6.18: The Hamming ball values (y-axis) for which the maximum F_1-score is obtained across several models (x-axis), for MNIST and Profi-set-JL.

6.5.4 Precision and Recall for Fixed Hamming Ball across SH Variations

In this subsection, we look at precision and recall across the SH variations by setting the Hamming ball for each tested model equal to the Hamming ball Vanilla SH needs in order to obtain the best F_1-score for the specific model and on a given dataset. The correspondence between the Hamming balls we set for each model can be reviewed in Figure 6.7 for all the real datasets.

If we compare the precision in Figure 6.19 with the recall in Figure 6.20, it makes sense that the two quality metrics are not far from each other in value for Vanilla SH, since we fetch results for its best F_1-score and therefore, they are weighted equally.

If we were to order the proposed heuristics in terms of their overall performance from the weakest to the strongest, we would think of it as such: Median SH, Balanced SH, PCcutrepeated SH, PCcutrepeatedmultiplebits SH and PCdominancebymodesorder SH. Among these, some show a big difference between precision and recall for the chosen Hamming ball (Median SH and PCdominancebymodesorder SH), while others are more tempered with regards to this (Balanced SH, PCcutrepeated SH, PCcutrepeatedmultiplebits SH).

In general, the figures support all our results. Firstly, they emphasize the trade-off between precision and recall across all variants and for different datasets and prove why Median SH performs worst overall (i.e. it yields high precision and low recall). Secondly, they provide a stronger intuition of why PCdominancebymodesorder SH obtains the best F_1-score overall: while not improving precision as much, it yields a very good recall compared to all the other heuristics.

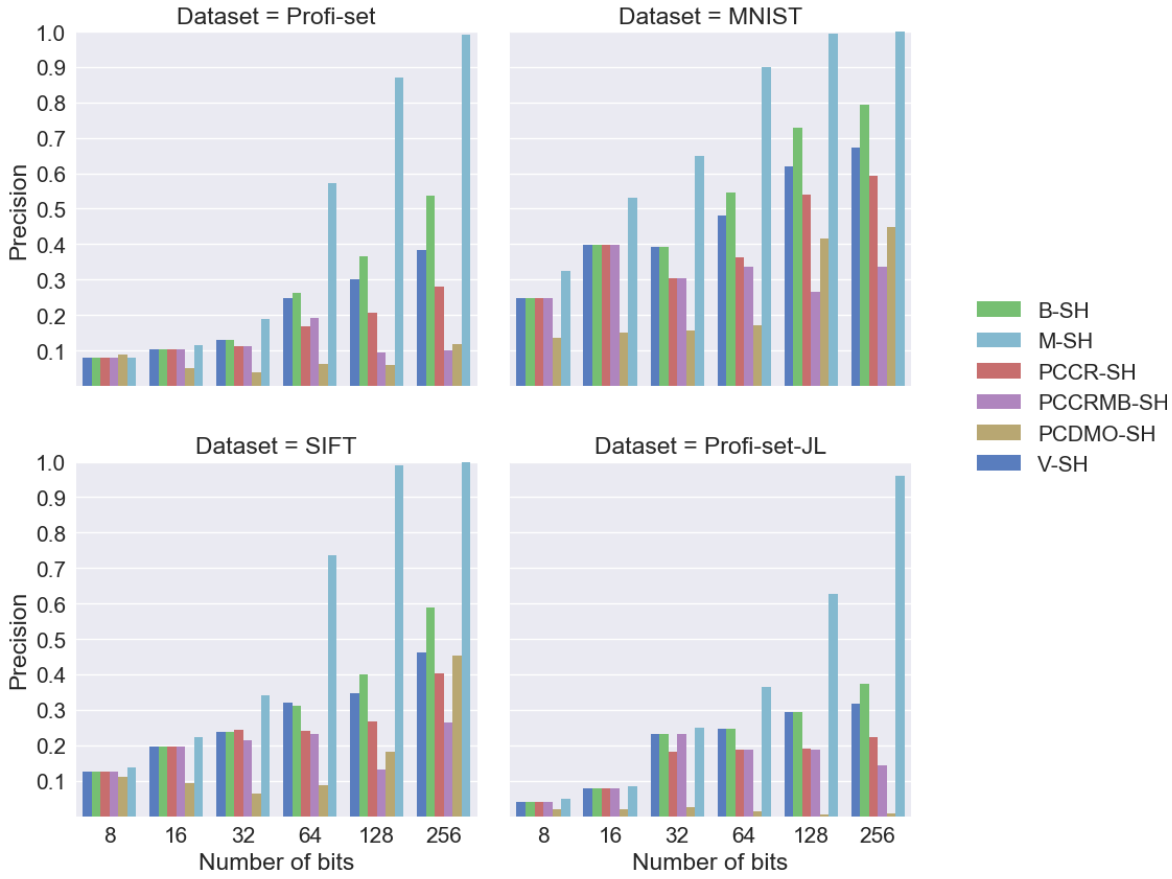


Figure 6.19: Precision values (y-axis) across several models (x-axis) for all real datasets. For each model, results are fetched for a Hamming ball fixed to the Hamming ball value where Vanilla SH reaches the maximum F_1-score. We pair up the models in $num\_bits \in \{8, 16, 32, 64, 128, 256\}$ with the following fixed Hamming ball sets: for the Profi-set $d\_balls_{Hamm} = \{0, 2, 7, 15, 34, 71\}$, for SIFT $d\_balls_{Hamm} = \{0, 2, 7, 16, 34, 72\}$, for MNIST $d\_balls_{Hamm} = \{0, 2, 7, 16, 33, 74\}$ and for Profi-set-JL $d\_balls_{Hamm} = \{0, 2, 6, 17, 40, 84\}$. We used the following parameters: training set size $n = 20,000$, testing set size $m = 2000$, number of seeked nearest neighbors $k = 100$.

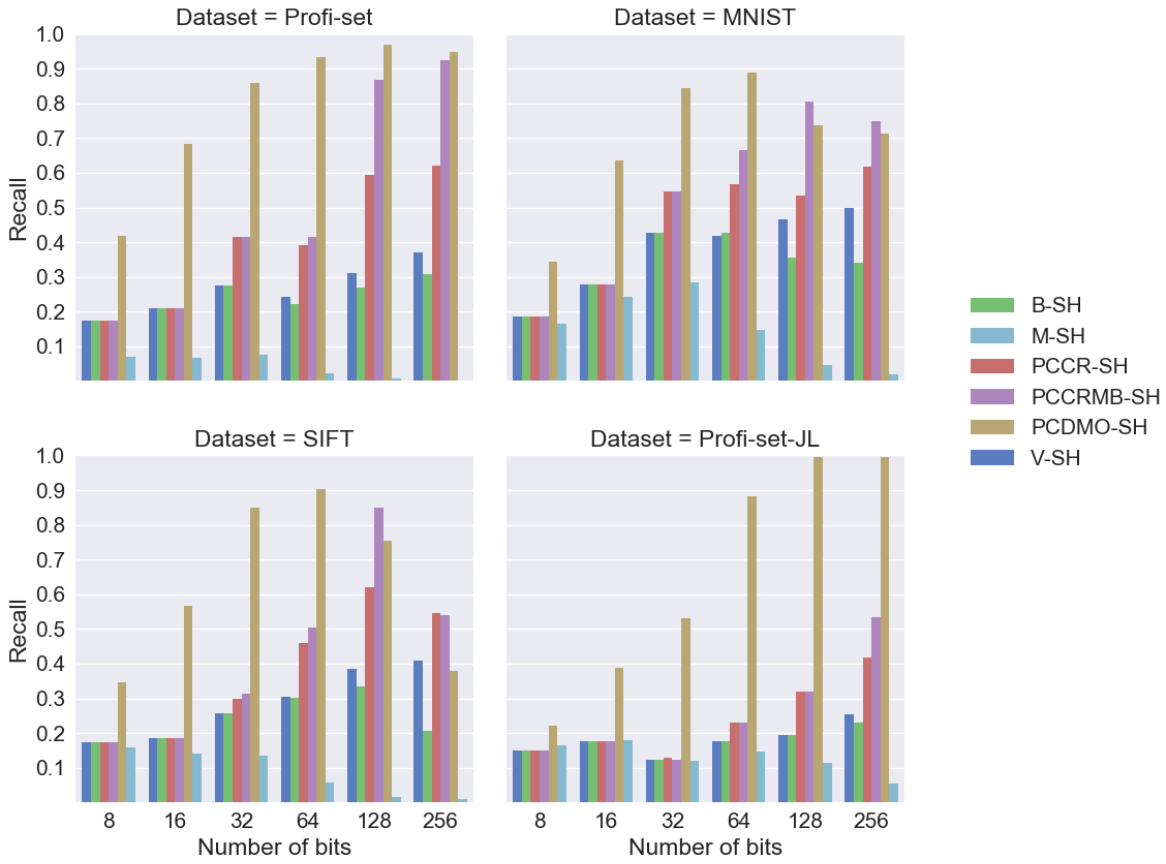


Figure 6.20: Recall values (y-axis) across several models (x-axis) for all real datasets. For each model, results are fetched for a Hamming ball fixed to the Hamming ball value where Vanilla SH reaches the maximum F_1-score. We pair up the models in $num\_bits \in \{8, 16, 32, 64, 128, 256\}$ with the following fixed Hamming ball sets: for the Profi-set $d\_balls_{Hamm} = \{0, 2, 7, 15, 34, 71\}$, for SIFT $d\_balls_{Hamm} = \{0, 2, 7, 16, 34, 72\}$, for MNIST $d\_balls_{Hamm} = \{0, 2, 7, 16, 33, 74\}$ and for Profi-set-JL $d\_balls_{Hamm} = \{0, 2, 6, 17, 40, 84\}$. We used the following parameters: training set size $n = 20,000$, testing set size $m = 2000$, number of seeked nearest neighbors $k = 100$.

7

Conclusion

Contribution. Relying on the original SH paper [46] and literature reviews, we researched the theoretical notions behind Spectral Hashing and aimed to relate them to the implementation [45]. As such, we looked deeply into how Spectral Hashing works, explained each step in the algorithm and emphasized some of its characteristics. For example, we found out that Spectral Hashing does not quite partition the space into equal buckets or that the Hamming distance between neighboring buckets is much bigger for some pairs of buckets than for others. Such characteristics certainly influence performance.

Notably, we exploited some of these characteristics and we came up with a series of heuristics which seem to perform either equally or better. Our thesis draws a parallel among all the proposed SH variants. We also explain how we implement them and compare with the original Spectral Hashing implementation. Notably, one of our heuristics performs best overall (i.e. PCdominancebymodesorder SH), being tested on the four real datasets we evaluate on. In most cases, it yields a significant increase in performance over Vanilla SH.

Furthermore, we analyzed how the algorithm behaves across several real datasets, interpreted the results and identified parameter dependencies.

In the context of testing it, we discussed the evaluation type proposed

in the original Spectral Hashing code [45] and argued why it is not always enough to look at precision or recall alone in order to assess the performance of the algorithm. Thus, we propose a slightly different evaluation type, in which we consider F_1-score a more appropriate quality metric to use. Moreover, our evaluation proposal provides a mapping between the model chosen for encoding (i.e. number of bits) and the Hamming ball required to obtain the best F_1-score.

Future work. Spectral Hashing represents a rich subject, which can be explored much more than we did along this thesis. Besides the already presented topics, we slightly experimented with different aspects of the algorithm, but not in enough detail to cover them here. Therefore, we acknowledge there is room for future work and a diversity of interesting experiments to conduct.

For instance, since Principal Component Analysis is an initial step of Spectral Hashing and to the best of our knowledge, it is sensitive to outliers, it would be interesting to explore how the algorithm behaves in such cases.

Besides outliers, experimenting with several data distributions would be curious as well. Uniformly distributed data, very clustered data (e.g. with clusters differently spaced out from one another and of different standard deviation) or multivariate Gaussian distribution are only some of such examples.

Additionally, the heuristics we propose only exploit the compression phase of the algorithm. As such, we believe an idea of future work is to also explore making improvements over its training phase.

Yet, considering our heuristics have a tendency of decreasing performance above a particular number of bits for some of the datasets, it would be interesting to actually continue improving them for the corner cases where

performance is not as aimed for.

Furthermore, Spectral Hashing uses PCA as an initial step, but as far as we know, this constraints it to finding linear manifolds in the data. Thus, another idea of future investigation is to also explore non-linear dimensionality reduction methods (manifold learning techniques [33]) and integrate them with the algorithm, with the intention of making it find non-linear manifolds in the data, such as S-shapes, tori, spirals, circular distributions etc.

Last, but not least, our thesis explores the performance of Spectral Hashing on image datasets. Extending the investigation to others, such as document datasets, for instance, would be interesting as well.

All in all, we believe our thesis contributes to the understanding of Spectral Hashing and leaves doors open for future perspective. This only demonstrates how rich and interesting the subject is, highlighting, at the same time, the considerable role Spectral Hashing plays in the context of Approximate Nearest Neighbors Search applications.

Bibliography

[1] Data Analysis and Manifold Learning. http://perception.inrialpes.fr/~Horaud/Courses/pdf/Horaud-DAML12.pdf. Accessed: 2017-10-17.

[2] Linear Transformations and Matrices | Essence of Linear Algebra, Chapter 3. https://www.3blue1brown.com. Accessed: 2017-12-11.

[3] Multivariable Calculus. https://da.khanacademy.org/math/multivariable-calculus. Accessed: 2017-12-20.

[4] Precision-Recall. http://scikit-learn.org. Accessed: 2017-12-21.

[5] H. Abdi. The Eigen-Decomposition: Eigenvalues and Eigenvectors. *Encyclopedia of Measurement and Statistics*, pages 304–308, 2007.

[6] A. Andoni. *Nearest Neighbor Search: The Old, the New, and the Impossible*. PhD thesis, MIT, 2009.

[7] J.S. Beis and D.G. Lowe. Shape Indexing Using Approximate Nearest-Neighbour Search in High-Dimensional Spaces. In *Computer Society Conference on Computer Vision and Pattern Recognition*, pages 1000–1006. IEEE, 1997.

[8] M. Belkin and P. Niyogi. Laplacian Eigenmaps and Spectral Techniques for Embedding and Clustering. In *Advances in Neural Information Processing Systems*, pages 585–591, 2002.

[9] M. Belkin and P. Niyogi. Laplacian Eigenmaps for Dimensionality Reduction and Data Representation. *Neural Computation*, 15(6):1373–1396, 2003.

[10] M. Belkin and P. Niyogi. Convergence of Laplacian Eigenmaps. In *Advances in Neural Information Processing Systems*, pages 129–136, 2007.

[11] R.E. Bellman. *Adaptive Control Processes: A Guided Tour*. Princeton University Press, 2015.

[12] P. Budikova, M. Batko, and P. Zezula. Evaluation Platform for Content-based Image Retrieval Systems. In *International Conference on Theory and Practice of Digital Libraries*, pages 130–142. Springer, 2011.

[13] F. R. K. Chung. *Spectral Graph Theory*. Number 92. American Mathematical Society, 1997.

[14] S. Cost and S. Salzberg. A Weighted Nearest Neighbor Algorithm for Learning with Symbolic Features. *Machine Learning*, 10(1):57–78, 1993.

[15] D. Dereniowski and M. Kubale. Cholesky Factorization of Matrices in Parallel and Ranking of Graphs. In *International Conference on Parallel Processing and Applied Mathematics*, pages 985–992. Springer, 2003.

[16] A. Ghodsi. Dimensionality Reduction, A Short Tutorial. *Department of Statistics and Actuarial Science, University of Waterloo, Ontario, Canada*, 37:38, 2006.

[17] A. Gionis, P. Indyk, and R. Motwani. Similarity Search in High Dimensions via Hashing. In *Very Large Data Bases (VLDB) Conference*, volume 99, pages 518–529, 1999.

[18] O. Hassanzadeh, M. Sadoghi, and R.J. Miller. Accuracy of Approximate String Joins Using Grams. In *QDB*, pages 11–18, 2007.

[19] J.-P. Heo, Y. Lee, J. He, S.-F. Chang, and S.-E. Yoon. Spherical Hashing. *Conference on Computer Vision and Pattern Recognition*, pages 2957–2964, 2012.

[20] Y. Huang. Content-based Information Retrieval via Nearest Neighbor Search. *Electronic Theses and Dissertations*, 2016.

[21] P. Indyk. Dimensionality Reduction Techniques for Proximity Problems. *SODA, The Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 371–378, 2000.

[22] P. Indyk and R. Motwani. Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality. In *The Thirtieth Annual ACM Symposium on Theory of Computing*, pages 604–613. ACM, 1998.

[23] H. Jegou, M. Douze, and C. Schmid. Product Quantization for Nearest Neighbor Search. *Transactions on Pattern Analysis and Machine Intelligence*, 33(1):117–128, 2011.

[24] W.B. Johnson and J. Lindenstrauss. Extensions of Lipschitz Mappings into a Hilbert Space. *Contemporary Mathematics*, 26(189-206):1, 1984.

[25] B. Kulis and T. Darrell. Learning to Hash with Binary Reconstructive Embeddings. In *Advances in Neural Information Processing Systems*, pages 1042–1050, 2009.

[26] Y. LeCun. The MNIST Database of Handwritten Digits. *http://yann. lecun. com/exdb/mnist/*, 1998.

[27] C. Li, E. Chang, H. Garcia-Molina, and G. Wiederhold. Clustering for Approximate Similarity Search in High-Dimensional Spaces. *IEEE Transactions on Knowledge and Data Engineering*, 14(4):792–808, 2002.

[28] U. Von Luxburg. A Tutorial on Spectral Clustering. *Statistics and Computing*, 17(4):395–416, 2007.

[29] M. Muja and D.G. Lowe. Scalable Nearest Neighbor Algorithms for High Dimensional Data. *Transactions on Pattern Analysis and Machine Intelligence*, 36(11):2227–2240, 2014.

[30] A.Y. Ng, M.I. Jordan, and Y. Weiss. On Spectral Clustering: Analysis and an Algorithm. In *Advances in Neural Information Processing Systems*, pages 849–856, 2002.

[31] Y. Park, M. Cafarella, and B. Mozafari. Neighbor-Sensitive Hashing. *Very Large Data Bases (VLDB) Conference*, 9(3):144–155, 2015.

[32] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[33] S.T. Roweis and L.K. Saul. Nonlinear Dimensionality Reduction by Locally Linear Embedding. *Science*, 290(5500):2323–2326, 2000.

[34] S. Saha and S.P. Ghrera. Nearest Neighbor Search in Complex Network for Community Detection. *CoRR*, abs/1511.07210, 2015.

[35] J. Shi and J. Malik. Normalized Cuts and Image Segmentation. *Transactions on Pattern Analysis and Machine Intelligence*, 22(8):888–905, 2000.

[36] D. Skillicorn. *Understanding Complex Datasets: Data Mining with Matrix Decompositions*. Chapman & Hall/CRC, 2007.

[37] M. Slaney and M. Casey. Locality-Sensitive Hashing for Finding Nearest Neighbors. *Signal Processing Magazine*, 25(2):128–131, 2008.

[38] L. I. Smith. A Tutorial on Principal Components Analysis. *Cornell University, USA*, 51(52):65, 2002.

[39] D. Sonawane and P.M. Yawalkar. A Review on Nearest Neighbour Techniques for Large Data. *International Journal of Advanced Research in Computer and Communication Engineering*, 4, 2015.

[40] D. A. Spielman. Spectral Graph Theory and its Applications. In *The 48th Annual Symposium on Foundations of Computer Science*, pages 29–38. IEEE, 2007.

[41] V. Spruyt. A Geometric Interpretation of the Covariance Matrix. `http://www.visiondummy.com/2014/04/geometric-interpretation-covariance-matrix/`. Accessed: 2017-12-13.

[42] J. VanderPlas. Python Data Science Handbook: Essential Tools for Working with Data. *O'Reilly Media*, 2016.

[43] J. Wang, H.T. Shen, J. Song, and J. Ji. Hashing for Similarity Search: A Survey. *CoRR*, abs/1408.2927, 2014.

[44] R. Weber, H.-J. Schek, and S. Blott. A Quantitative Analysis and Performance Study for Similarity-Search in High-Dimensional Space. In *Very Large Data Bases (VLDB) Conference*, volume 98, pages 194–205, 1998.

[45] Y. Weiss, A. Torralba, and R. Fergus. Advances in Neural Information Processing Systems. `http://www.cs.huji.ac.il/~yweiss/SpectralHashing`, 2008. Accessed: 2017-10-10.

[46] Y. Weiss, A. Torralba, and R. Fergus. Spectral Hashing. *Advances in Neural Information Processing Systems*, pages 1753–1760, 2009.

[47] P. Yuan, C. Sha, X. Wang, B. Yang, and A. Zhou. Efficient Approximate Similarity Search Using Random Projection Learning. In *Web-Age Information Management*, pages 517–529. Springer, 2011.

[48] P. Zezula, G. Amato, V. Dohnal, and M. Batko. *Similarity Search: The Metric Space Approach*, volume 32. Springer Science & Business Media, 2006.