Introduction

As data continues to grow, it is becoming increasingly critical for organizations to be able to easily manage and sort their datasets, whether they come from databases, files or other sources. For this purpose, Talend already offers Talend Data Inventory, a tool that makes it easy to find and understand your data with powerful and versatile search, provenance and dataset overview. It also helps to identify data silos in all data sources and eliminates them with reusable and shareable data assets.

In order to continuously improve the Talend Data Inventory quality of service, Talend Lab team is interested in integrating a tool that can detect the similarities between datasets, a functionality that can be used in different use cases, the most relevant one is at the moment when the user introduces a new dataset, we want to find the ones that are similar to it, and by providing this information to the user we can propose some actions he can do, he can for example compare and merge the similar datasets or update the existing one with new records. Another use case is to assign a tag to a dataset according to the tags of the ones that are similar to it, for example if we find that a dataset is similar to two dataset with the tag "*finance*", we can suggest this same tag to the user.

In this work, we tackle the problem of detecting similarities between dataset as a Nearest Neighbor Search (NNS) problem, we explore the state of the art with focusing on Locality-Sensitive Hashing (LSH) methods and we study their different versions and the metric they consider.

We also present the solution built during this internship, in which we use a method of the LSH family called Cross-Polytope, our goal is to scale its idea that was initiated to work at row level and get it work at the column level.

Chapter 1

State of the art

1.1 Nearest Neighbor Search problem

Nearest Neighbor Search (NNS) is a topic that attracted much active interest in the machine learning research field as its exhaustive solutions require a lot of time and resources to find the solution. It can find its applications as a fundamental approach in multiple areas as recommendation systems, searching in multimedia data, information retrieval, as well as in pattern recognition to name but a few.

The problem of Nearest Neighbor Search is formulated given a set of n points of dimension d and a query point q, it is concerned with finding the point (or points) closest to this query point. In the figure 1.1, a basic example is presented on the left with 09 points and a query point, the algorithm should return the point s as the nearest neighbor.

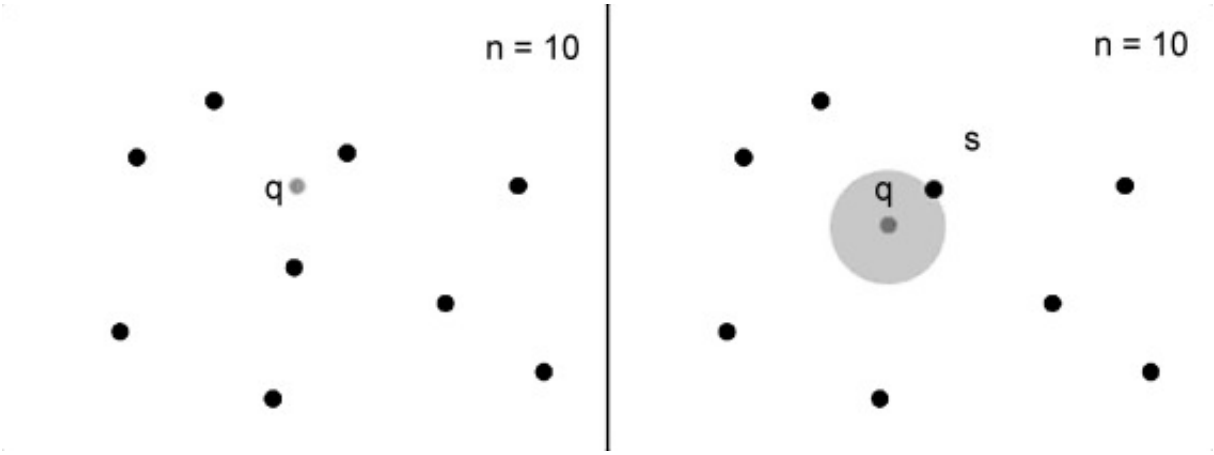

Figure 1.1: Basic example of Nearest Neighbor Search

The simplest solution, or the native solution is to compute the distance between the query point and all the other points with a chosen metric. This solution guarantees to return the exact nearest neighbor, but its computational complexity is at the order of $O(nd)$ which will make it become more and more complex with the explosive growth in the scale of datasets (when the number of points and their dimension get bigger), that's what is called Curse of Dimensionality problem where the accurate algorithms wouldn't meet the requirements to guarantee the efficiency of the solution. Thus, the exact nearest neighbor problem has been reformulated into (Approximate Nearest Neighbor) where the

focus is on improving the efficiency of the solution with a tradeoff on the accuracy (Hajebi et al., 2011).

1.2 Approximate Nearest Neighbor Search

Approximate Nearest Neighbor (ANN) is a reformulated version of the Nearest Neighbor Search where the solution is found by the use of an index that gives a latent representation of the original data points, the intuition is to divide the data points into subsets (bucket) of data points according to this new latent representation, and to select one subset do the search on it. The loss of information between the original data points and their latent representation makes the solution not guaranteed to be exact.

The methods implementing the idea of ANN can be classified under four major families:

- Hashing based algorithms: The idea behind this family of methods is apply a set of hashing functions on the data points and to consider their results as the new representation (in a lower dimensional space) of their original points.
- Quantization based algorithms: The idea is to decompose the space into a Cartesian product of low dimensional subspaces and to quantize each subspace separately. (Jégou et al., 2011)
- Tree based algorithms: The general idea is to form a tree where the root node represents all the dataset, and to start splitting this data to two halves by a hyperplane orthogonal to a chosen dimension at a threshold value. (Silpa-Anan and Hartley, 2008)
- Graph based algorithms: The idea is to represent the data as a graph where the nodes are the data points, and an edge between two nodes defines a neighbor-relationship between their respective points. (Wang et al., 2021)

Hashing based algorithms, Quantization based algorithms, Tree based algorithms, and Graph based algorithms.(Wang et al., 2021).

In this present work we explore the hashing based algorithms, or Locality-Sensitive Hashing (LSH) in order to see how can they be suitable for the problem of detecting similarities between datasets.

1.3 Locality-Sensitive Hashing

Locality-Sensitive Hashing (LSH) is one of the methods that tackles the problem of the approximate nearest neighbor search, its main idea consists of using the hashing [1] techniques to map the near data points to the same buckets [2] with high probability, and map the distant ones to the same buckets with low probability. The intuition behind it is to apply a number of hashing functions on the data points to get their representations in a new latent space, and the goal with these hashing functions is to have nearby points in the original space close from each other in the latent space.

[1] It is the process of transforming an object into a relatively shorter fixed-length value that we call key, and it is used to make the search operation faster.

[2] A bucket is a group of elements that share the same hash value

Locality-Sensitive hashing and Classical hashing

In classical hashing or hash-based search the idea is to load a set of elements in a hashing structure that has multiple entries and to have a hash function that maps every element from the set to one of the entries. The goal behind choice is to speed up the search process, which can be guaranteed by minimizing the collisions between the set elements.

On the other hand, in Locality-Sensitive Hashing the idea is to map data points to different buckets in a way that the nearby elements fall in the same buckets with high probability. Unlike classical hashing where the goal is to minimize collisions, in LSH we look to maximize them.

1.4 LSH formulation

Given a dataset $\mathbb{D}$ of size $n * d$, n elements of d features (or n rows and d columns), and a set $H(.)$ of m hashing functions, each element $x \in D$ is mapped to a binary vector of dimension m defined with $H(x) = [\ h_1(x),\ h_2(x), ..,\ h_m(x)]$ (Chi and Zhu, 2017)

A hash function is said locality-sensitive if for two points x_i and x_j it satisfies the following conditions:

- if $d(x_i, x_j) \leq R$ then $P(h(x_i) = h(x_j)) \geq p_1$.
- if $d(x_i, x_j) > cR$ then $P(h(x_i) = h(x_j)) < p_2$.

Where:

- $c > 1 \in \mathbb{R}$
- p_1 is the probability for nearby points.
- p_2 is the probability of points being far apart.

The gap between p_1 and p_2 is quantified by the parameter c, and it determines how sensitive the hash function is to the changes in distance, the formula of this sensitivity is given by: $\rho = \frac{\log 1/p_1}{\log 1/p_2}$

The figure 1.2 below, shows in the red zone the meaning of the first condition that consists of having all the points in the red circle colliding with the query point with a high probability. While the second condition is shown in the blue zone which means that all the points in the purple circle have a small probability of colliding with the query point.

The distance function d mentioned in the two conditions 1.4 can be different depending on the nature of the treated problem, and so are the hashing functions used in LSH which can estimate different metrics.

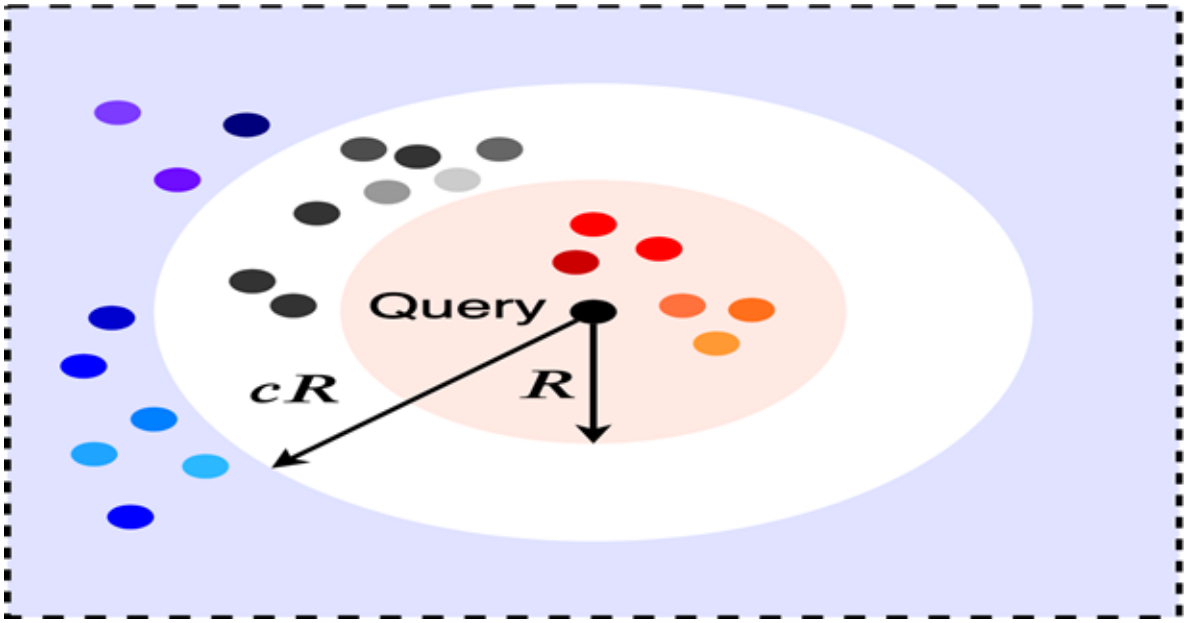

Figure 1.2: Locality-Sensitive Hash Family demonstration. Source: Blog in randorithms

1.5 Metrics

In this section we present some of the relevant metric to measure similarities between data points and explore where they can be used.

1.5.1 Jaccard metric

The Jaccard metric is a metric used to calculate the similarity between two sets of values, it is defined as the size of the intersection set divided by the size of the union set. It can be expressed with the following expression:

$$J(x, y) = \frac{x \cap y}{x \cup y}$$

The figure 1.3 illustrates visually how is the Jaccard metric computed.

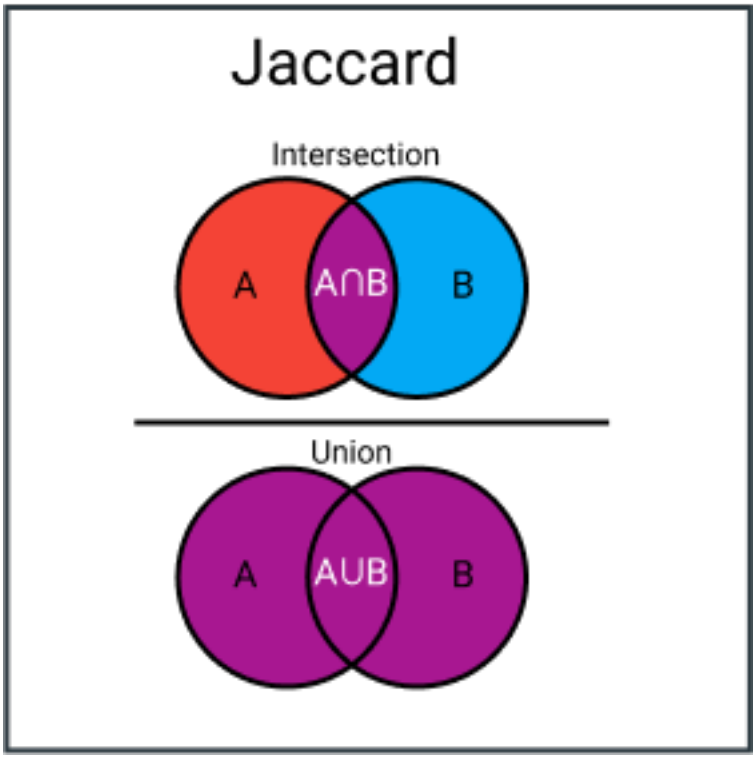

Figure 1.3: Collision probability of LSH for the Jaccard metric

To get an estimation of the Jaccard metric using LSH is defined as: $H(x) = min(\pi(x_i))$. It represents the index of the first row in which contains the minimum value of the data point (or the first occurrence of 1 in case of binary data) after a permutation π.

The collision probability between two data points following with this hashing method is exactly equal to the Jaccard similarity (Yu and Weber, 2022):

$$P(H(x) = H(y)) = J(x, y) = \frac{|x \ \cap \ y|}{|x \ \cup \ y|}$$

Jaccard similarity has found its use in multiple fields, we can mention its use in text mining with the example of duplicate document detection, it can also be used in e-commerce and recommendation systems to find similar customers via their purchase history.

One obvious remark about the Jaccard metric is that it can biased by the size of the data, more precisely the large datasets increase significantly the size of the union and thus reduce the similarity metric, so a dataset can have a lower similarity metric with a dataset of large size than the one with a dataset of smaller size even if they share more values.

1.5.2 Hamming Metric

In case of binary data, this distance is expressed as the sum of the absolute values of the element-to-element difference between two data vectors. More generally the Hamming distance between two vectors is the number of distinct values between them.

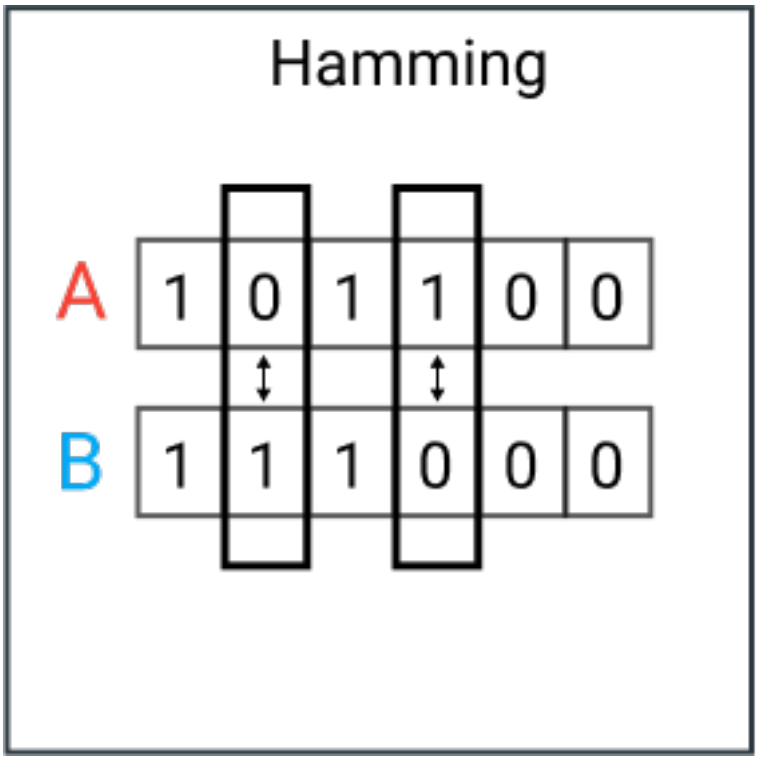

Figure 1.4: Hamming metric illustration

To get an estimation of the Hamming metric using LSH, the hash function used is $h(x) = x_i$ where i is chosen randomly and x_i represents the i^{th} dimension of the data point x.

The probability of collision between two elements x and y is given by:

$$P(H(x) = H(y)) = 1 - \frac{hamming(x, y)}{d}$$

Where $hamming(x, y)$ is the hamming distance, and d is the dimension of the data points. This probability comes from the intuition that the number of indices that verify $x_i = y_i$ is $n - d(x, y)$, and when this quantity is divided by n, the total number of indices, we get the quantity described above.

It is obvious that this measure takes only into account does the fact that the values are different or equal, and doesn't pick attention to their values.

This Hamming metric is more fit to use when the magnitude is not an important measure. Its main use is in the case we want to measure similarities or differences between binary data vectors.

1.5.3 Euclidean Metric

It is the most commonly used metric as it represents the length of the segment connecting the two points.

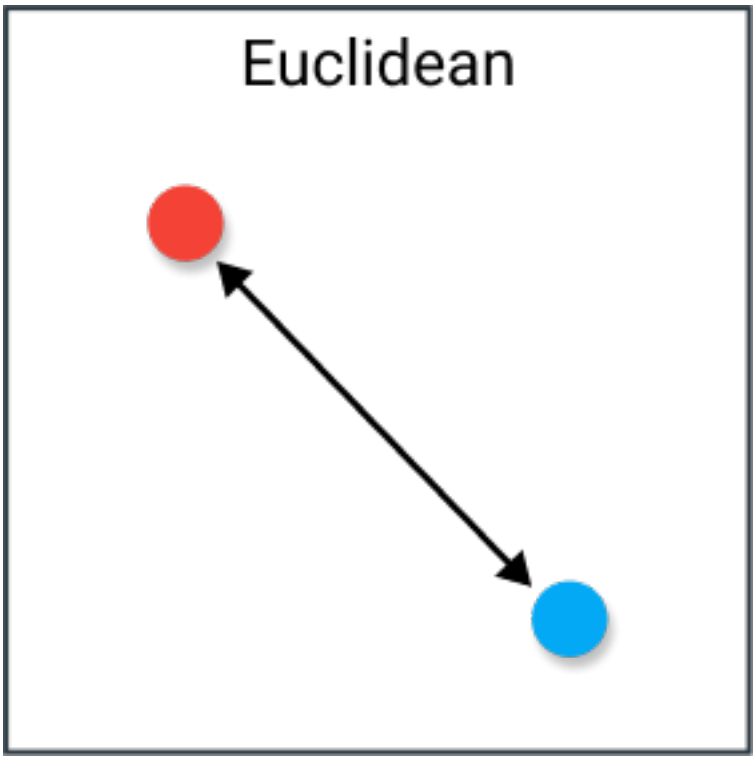

Figure 1.5: Euclidean metric illustration

To get an estimation of the Euclidean metric using LSH, the used hashing function is

$$h(x) = [\frac{a.x + b}{w}]$$

Where a is a d-dimension vector drawn from any Gaussian distribution and b is drawn uniformly from $[0, w[$ where w is the number of the hashing functions.

This same hashing family can be used to get estimation of other metrics, it just needs to change the distribution of the rotation vector a, we draw a from Levy distribution in the case of the Minkowski metric (OpenGenus-Foundation and Aditya, 2019), and from Cauchy distribution in the case of the Manhattan metric (OpenGenus-Foundation and Aditya, 2018).

Contrary the the Hamming metric, the euclidean metric takes into account the the magnitude of the values of the data points. But on the other hand, it has a limitation when dealing with data whose the features are expressed in different units, a limitation that can be resolved with normalizing the data. Moreover, the values of this metric become less useful when the data has a lot of features. And this comes from the fact that the geometric intuitions that we had from the three-dimensional and the two-dimensional spaces are not often applied in high dimensions. (Domingos, 2012)

The euclidean metric is more often used in algorithms like K-Mean where the data dimension is low, and where the similarity between data points is gauged enough with a straight forward distance.

1.5.4 Angular metric

The angular or cosine metric is simply the cosine measure between the vectors of the two data points, it can also represents the inner product between them if they are both normalized.

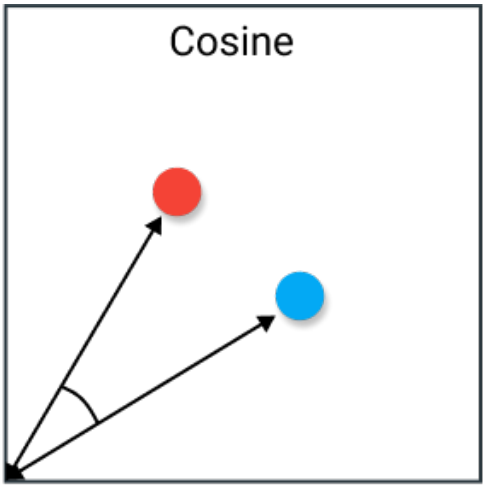

Figure 1.6: Angular metric illustration

To get an estimation of the Angular metric using LSH, the hash function used is defined as: $H(x) = sgn(a.x)$ where sgn is the sign function, and a is a random d-dimensional vector drawn from any Gaussian distribution. The illustration in the figure 1.7 shows that the random drawn line (issued from a vector drawn from any Gaussian distribution) splits the points in two groups, a group in the positive side and another in the negative one.

The probability of having collision between two elements is given by:

$$P(H(x) = H(y)) = 1 - \frac{\theta}{\pi}$$

The intuition behind this formula is that given an angle θ between two vectors and a random line, this line has $\frac{\theta}{\pi}$ chance to cross this angle, which means that it is also the chance to have these two vectors separated by the line. The probability of collision will be: $1 - \frac{\theta}{\pi}$ (Jafari et al., 2021). The figure 1.8 illustrates the meaning behind the expression of the probability collision.

This metric has found its use in multiple cases, one of them is for similarity estimation in text embeddings, as it doesn't present the limit of the euclidean metric with high dimensional data. It can also be used for similarity estimation between picture and fingerprint embeddings.

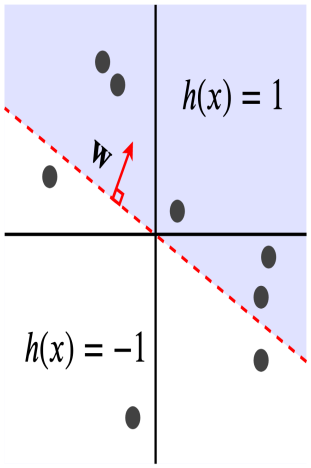

Figure 1.7: Angular metric estimation with LSH

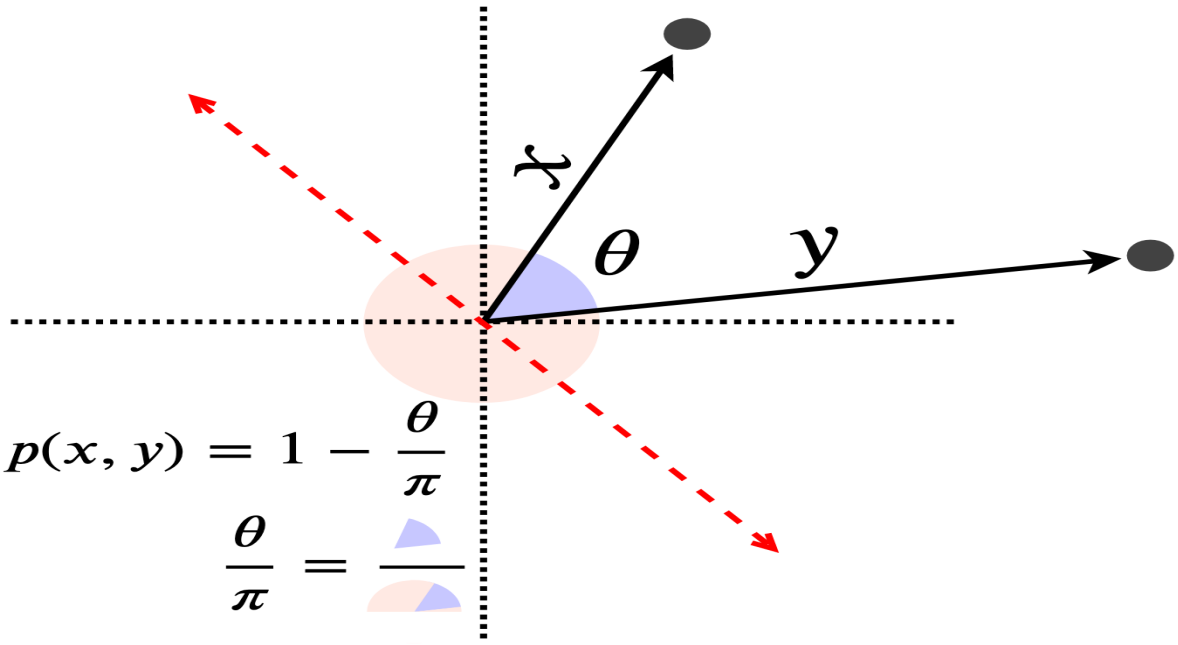

Figure 1.8: Collision probability of LSH for the angular metric

1.6 LSH steps

Before presenting the different LSH methods, let's start by presenting the global approach shared by all of them, it consists of three steps:

- Hashing.
- Amplification.
- Bucketing.

1.6.1 Hashing

This step consists of applying a set of hashing functions $H = (h_1, h_2, .., h_m)$ on the n data points to get their signatures. It's the step that makes the difference between the hashing methods as it is what makes the effect of choose the hash family clearer. At the end of this step, the result is a matrix where each data points is presented by its signature, this matrix is called signature matrix.

$$\begin{bmatrix} h_1(x_1) & h_1(x_2) & \cdots & h_1(x_n) \\ h_2(x_1) & h_2(x_2) & \cdots & h_1(x_n) \\ \cdots & \cdots & \cdots & \cdots \\ h_m(x_1) & h_m(x_2) & \cdots & h_m(x_n) \end{bmatrix}$$

This step of bucketing is contributing on reducing the complexity of the search process by reducing the dimension of the data points and representing them on a latent space. It lets the search process dealing with a reduced size of vector rather than to perform on the entire vectors.

1.6.2 Amplification

The amplification step the signature matrix from the previous section is divided into b bands each one of size r. This means that, for a set of hashing functions (h_1, h_2, $\cdots$, h_p) such that $p = b * r$, we will have this set divided into b sub-sets ([h_1, $\cdots$ h_r], [h_{r+1}, $\cdots$ h_{2r}], $\cdots$, [$h_{(b-1)*r+1}$, $\cdots$ $h_{p=b*r}$]). The figure 1.9 shows how the signature is divided.

The intuition behind doing such dividing, is to minimize the collisions between points that are not similar. In another way, we aim to minimize the false positives that may occur when we consider one hash value at a time while making buckets by not considering one hash value at a time but regrouping them into b set of hashing values.

1.6.3 Bucketing

The bucketing step is the last step in the process in each of the LSH methods. It consists of creating buckets and assigning data points to them. To do this, it gets the output of the amplification step which are hashing values divided into subsets such that each subset of a data point x is used to identify the bucket in which x is assigned. Each data point can be mapped to different buckets as it has different subsets of hashing values.

The figure 1.10 show an example on how the different data points are assigned to buckets.

At the end of this step, the data points sharing the same bucket are considered as candidates to be similar.

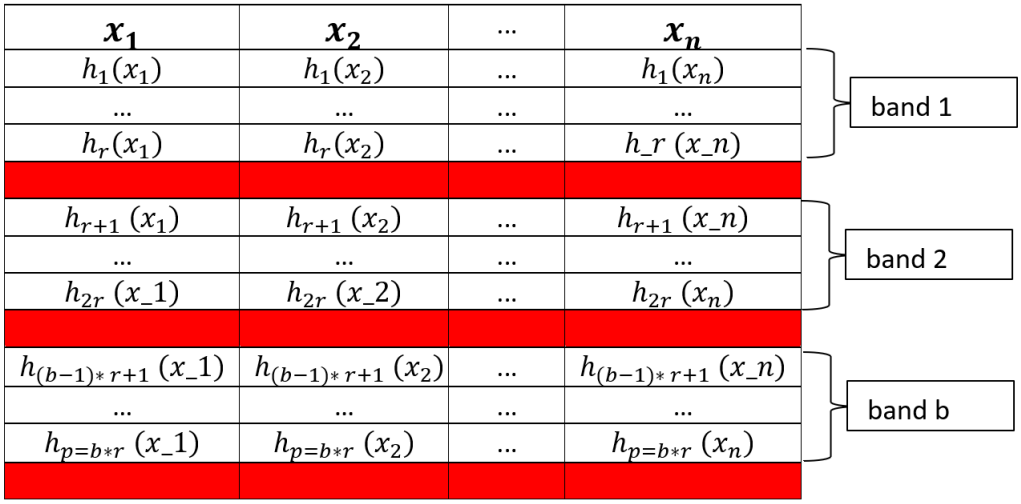

Figure 1.9: Amplification step illustration: Division of the signature matrix

This step of bucketing is contributing on reducing the complexity of the search process by focusing its search on the subset of data points that share at least one bucket with the query point rather than having to do the search over all the data points.

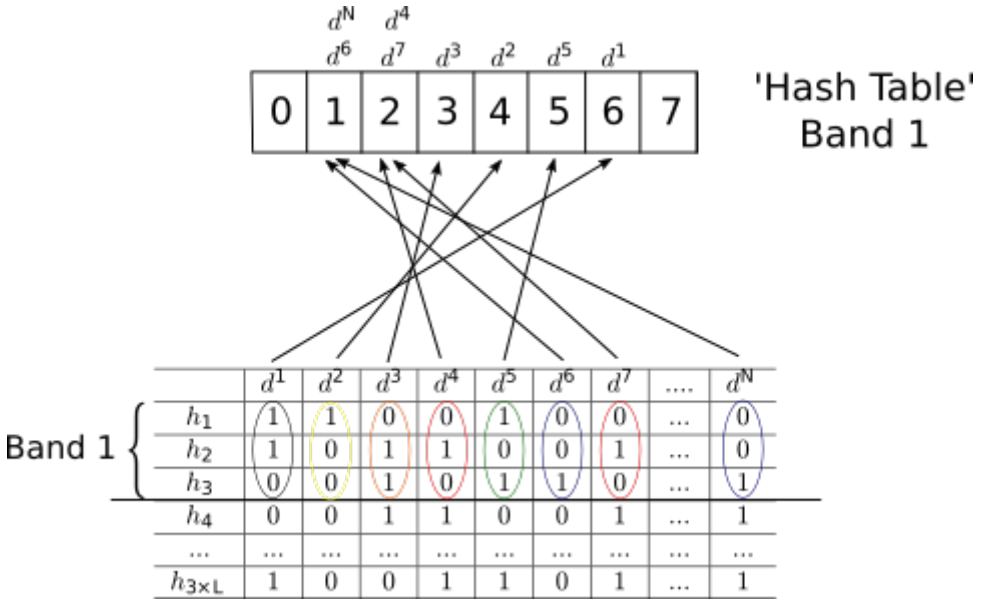

Figure 1.10: Bucketing step example

1.7 The LSH methods

In the previous section we have shown the different metrics that can be used to measure the distances or similarities between data points and that can be used in our case, and we briefly introduced the LSH methods that can be used to estimate each one of the metrics. In this section we will present in details the process of applying each one of them.

1.7.1 Minwise-Hashing LSH

Minwise-Hashing or MinHash LSH is one of the methods of LSH, it was proposed to estimate the Jaccard similarity metric between data points of any sizes. It relies on the fact that applying a random permutation to the set of data points, the chance that the smallest item under this permutation in sets A and B are precisely the same as their similarity of Jaccard. (Yu and Weber, 2022)

Formulation

The classic formulation of MinHash was to use a set of permutations over the data points, and take the argument corresponding to the minimum value of each permuted column (or corresponding to the first occurrence of 1 in case of binary data).

Knowing that the permutations consume lot of memory when the number of data points get bigger, it becomes more complicated to use them. A new formulation has occurred, where each set is converted to a MinHash signature using a set of m minwise-hashing functions $(h_{min_1}, h_{min_2}, \cdot, h_{min_m})$. A minwise-hashing function's value $h_{min}(X)$ of a set X is obtained by taking the minimum from the result of applying a set of independently generated hashing functions $(h_1, h_2, .., h_l)$ on a column: $h_{min}(X) = min(h_1(X), h_2(X), \cdot, h_l(X))$ (Zhu et al., 2016). Each function h_i will then return an integer value. Each column will have a value for each of the m minwise-hashing functions which is its hash value, and the column can be represented by the concatenation of its hash values, the concatenation that is also called the column signature. The signature matrix is the concatenation of the columns signatures, and it will be on size (m, d).

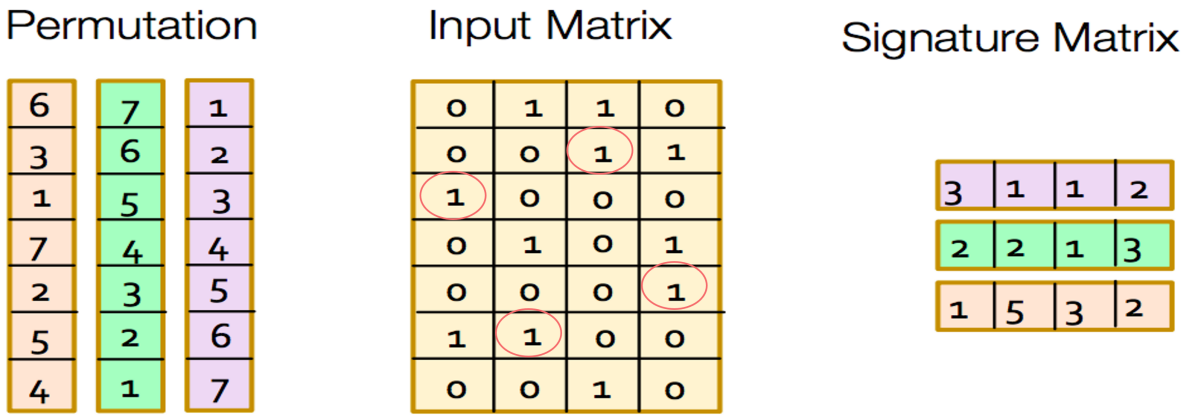

Figure 1.11: Minwise-Hashing LSH example

The figure 1.11 shows an example on hashing with MinHash LSH.

When applied on a data matrix of size (n, d), MinHash LSH has a time complexity of $O(m\ log\ m + n)$; where m is the number of hash functions. (Ertl, 2020)

The main advantage of MinHash LSH is its suitability for discrete sets of data as it relies on the Jaccard similarity which is an accurate measurement for this type of data. On the other hand, MinHash LSH is less suitable when it comes to treating and detecting similarities between continuous datasets (data with real numbers for example), and this is again related to the measure that it tries to estimate (Jaccard similarity), which is not accurate in this case.

1.7.2 Simhash LSH

SimHash is one of the methods of LSH, it was proposed to estimate the Angular metric between two data points. It relies on drawing random lines (from Gaussian distribution) on the data space and determine the value of the hashing by observing if the data point is on the positive or negative side of this randomly drawn line. The positive side indicates a hashing value of 1, and the negative side indicates a hashing value of 0 (sometimes -1 is used instead).

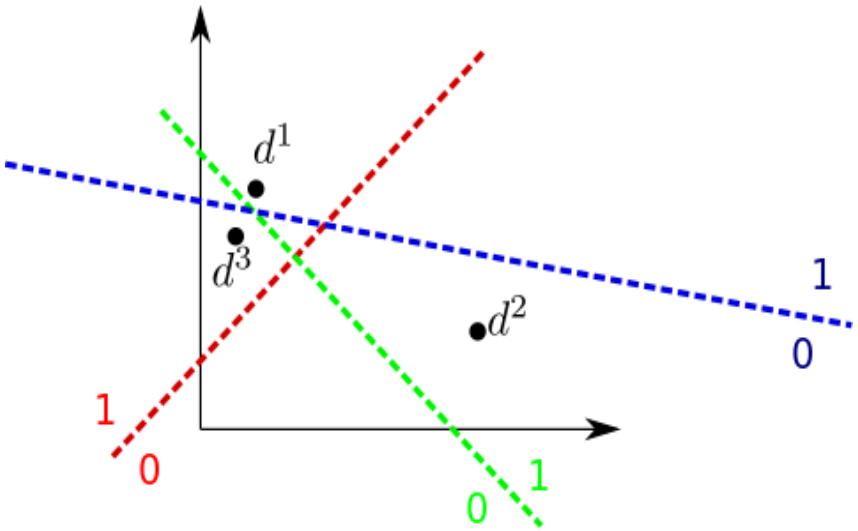

Figure 1.12: Simhash LSH illustration

The figure 1.12 shows an example on how are the hashing values determined according to their positions with the randomly draw lines.

Formulation

SimHash method takes p random normalized vectors, and compute the scalar product of each one of them by the data point's vector x (this will be identical to the cosine similarity since the vectors are normalized):

$$h_p(x_i) = 0 \quad if \quad w_i.x_i \leq 0 \;\; ; \;\; h_p(x_i) = 1 \;\; if \;\; w_i.x_i > 0$$

The probability of collision as we saw in section 1.5.4 (for one hash) is: $1 - \frac{\theta}{\pi}$. Now, having p hashing functions it becomes:

$$P(H(x_i) = H(x_j)) = (1 - \frac{\theta}{\pi})$$

Simhash was one of the first variants of LSH and was widely used in near duplicate document detection (plagiarism detection for example). The other variants extended the

use to document similarity, image classification and so on. SimHash gives advantages over MinHash LSH (1.7.1) when it comes to the cases where the data is continuous, in this way, when a couple of data points is detected similar, it implies that they had a small angle between them as this method estimates the angles between data points.

It also found use in the classification of users' interests as it was introduced in Evaluation of Cohort Algorithms for the FLoC API (Research and Ads, 2020).

SortingLSH

This same paper (Research and Ads, 2020) has introduced a new LSH method that relies on Simhash called SortingLSH.

The intuition behind the SortingLSH algorithm is to sort the hashing vectors given by the Simhash method, and group them to form clusters. It is a method that processes over the simHash clusters, in order to give more control on the size of each cluster which is an important criteria in the case of Ads to assure the user's privacy.

Simhash algorithm is formulated given a dataset D of dimension (n, d), we compute for each data point its hashing vector as in the simHash method: $(h_1 = H_p(x_1), h_2 = H_p(x_2), .., h_n = H_p(x_n))$. Those vectors are sorted in a lexicographical order and then assigned to clusters by generating contiguous groups of at least k hashes.

1.7.3 Cross-Polytope LSH

Cross-Polytope LSH is one of the LSH family methods, it was proposed to estimate Euclidean metric (1.5.3) on a unit sphere, which also corresponds to the Angular metric (1.5.4). It relies on the fact that elements that are close to each other (in the two previously mentioned metrics) are more likely to be close to the same axis after applying a random rotation. (Andoni et al., 2015)

The figure 1.13 show below shows how the cross-polytope hashing works. We transform the data points to bring them to a unit sphere, and then apply a random rotation to each one of them. The hash value of each point will be the index of its nearest axis.

Formulation

Let $\mathcal{H}$ be a hash family for points on a unit sphere $S^{d-1} \subset R^d$, and the random rotation matrix $A \in \mathbb{R}^{d*d}$ with i.i.d gaussian entries. To get the hashing of a data point $x \in S^{d-1}$ with cross-polytope LSH, we compute its normalized random rotation $y = \frac{Ax}{\|Ax\|} \in S^{d-1}$ and then from basis vectors $\{\pm e_i \,|1 \leq i \leq d\}$ the closest one.

This basis vector is the hash of x with cross-polytope LSH. More formally we write the hashing process as follow:

$$\mathcal{H}(x) = \arg \min_{u=\{\pm e_i\}} \| \frac{Ax}{\|Ax\|} - u \|$$

This method has a time complexity of $O(d.n^\rho/p_1)$ and space complexity of $O(n^{1+\rho}/p_1 + dn)$, where $\rho = \frac{\log 1/p_1}{\log 1/p_2}$ is the gap defined in 1.4. (Indyk and Motwani, 2000)

Pseudo-random-rotation

As described in (Andoni et al., 2015), the time complexity to apply a random rotation (multiply $d * d$ matrix with a data point) is expensive, it is at the order of $O(d^2)$ which

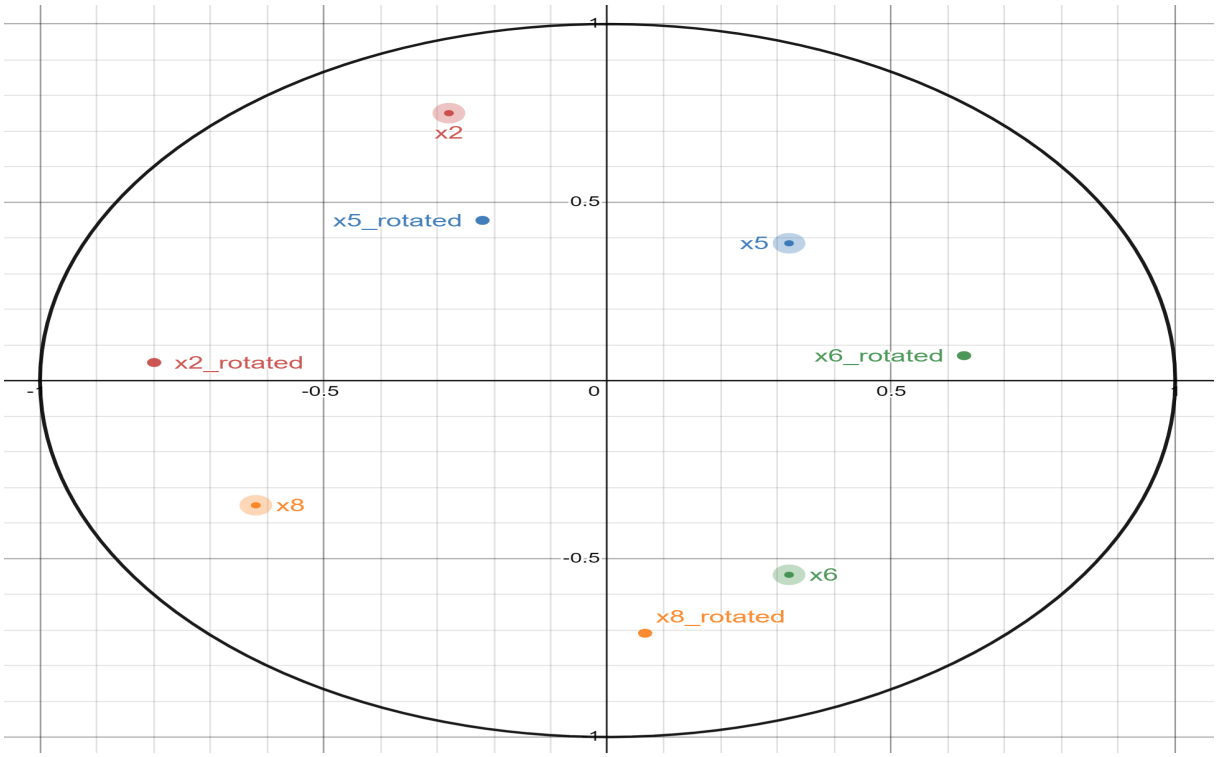

Figure 1.13: Cross-Polytope LSH example

will make the approach obsolete for large dimensions. Empirically, it turns out that the pseudo-random rotation gives similar results to the fully-random one. This pseudo-random rotation is given by the following linear transformation:

$$x \mapsto HD_1HD_2HD_3x$$

Where H is the Hadamard matrix (1.7.3), $D_1, D2$, and D_3 are random diagonal matrices, whose the diagonal vector are randomly picked from $\{1; -1\}$. To make the cross-polytope practical, we use this linear transformation instead of using the random rotation.

$$\mathcal{H}(x) = \arg \min_{u=\{\pm e_i\}} \| \frac{HD_1HD_2HD_3x}{\|HD_1HD_2HD_3x\|} - u \|$$

Hadamard transform

The Hadamard transform is an example of a generalized class of Fourier transforms. It was named after the French mathematician Jacques Hadamard. This transformation relies on a matrix (also named Hadamard matrix) to perform orthogonal, symmetric, involutive, linear operation on 2^m real numbers. (Quantiki).

The Hadamard transforms are mainly used in signal decomposition applications, in image processing, speech processing and power spectrum analysis.

a Hadamard matrix is a square matrix whose entries are either $+1$ or -1 and whose rows are mutually orthogonal. $H_1 = 1$

$$H_2 = \begin{pmatrix} H_1 & H_1 \\ H_1 & -H_1 \end{pmatrix}$$

$$H_4 = \begin{pmatrix} H_2 & H_2 \\ H_2 & -H_2 \end{pmatrix}$$

This method can find its use in different applications like the comparison of image feature vectors, speaker representations, and so on. Its main advantage is that it come with theoretical guarantees to give a good approximation of the angular and euclidean (on a unit sphere) metrics , and also with experimental improvements on its complexity for the large datasets. (Andoni et al., 2015)

Chapter 2

Conception

In this section we present the design of the solution built during the internship with its different components. The main focus was to find a relevant metric (from **??**) to measure the similarity between two columns, and it is the angular metric that has been used as it takes into account the magnitude of each point and it responds well to the problem of big dimensions presented by the euclidean metric, also, it is not limited to measure similarities over discrete data as the Jaccard similarity does.

The choice of the angular metric implies that the LSH method for the indexation process has to be chosen from the ones that estimate this metric. We go for cross-polytope as LSH method to estimate the similarities of the euclidean metric on a unit sphere (which has the same order of magnitude as the angular metric).

Our solution is divided into two main parts, the first part is related to the indexation process where the dataset indexes are computed and stored, and the second part is meant to take the process of detecting similarities between datasets through their previously computed indexes.

2.1 Indexation

As described previously, this phase consists of computing the index of each dataset's column and to store it in a database. This process is completed through the following three steps:

2.1.1 Preprocessing

The indexes are based on the column types, 2 types are considered: textual columns and non textual columns (numerical columns). The preprocessing will be different depending on the column separation process:

Non-Textual (Numerical) columns

One of cross-polytope requirements is that the data should have a size equal to a power of 2, for this we'll set this size on a value between 128 to 2048, and we'll call it DEFAULT_DATA_SIZE.

A random sample of size DEFAULT_DATA_SIZE is picked for each column and then normalized to have the data ready to be indexed with cross-polytope.

Textual columns

The textual columns have a different type of preprocessing, is it based on embedding them with a pre-trained embedding model according to a certain approach that indicates how the embedding model is used over all the cells to have as a result one vector representing the column content. Four approaches will be tested to pick the most accurate one.

Approach 1 : The idea of this approach is to concatenate the texts of each cell to have one text that will be embedded with the chosen embedding model. The output will depend on this embedding model, and the last step will be to pad this output with zeros to make it of a size equal to a power of 2.

Approach 2 : This approach consists of embedding each cell on its own with the chosen embedding model, and then to concatenate this embeddings to form one vector that will be the output of this approach.

Approach 3 : This approach has the same logic as the previous one, it adds an extra step which is to perform a PCA on each embedding just before concatenating in order to have more control over the size of the output and to make it satisfy the cross-polytope conditions easily.

Approach 4 : This approach also has the same logic of the second one, meaning that it starts with embedding each cell independently, and then get the mean of each component of the embedding vector.

$$a_{i/i\in[1,m]} = \frac{1}{n}\sum_{j=0}^{n} e_{j,i} \tag{2.1}$$

Where:

- m is the size of the embedding model's output.
- n is the size of the column.
- e_j is the embedding of the j^{th} cell.
- $e_{j,i}$ is the i^{th} component of the j^{th} cell embedding.

The last step in this approach will be to pad this output with zeros to make it of a size equal to a power of 2.

This step takes as input the concerned dataset, and outputs a list of columns with their preprocessed data for both textual and non-textual data.

2.1.2 Computing index

At the level of computing indexes, we get the preprocessed column groups and we pass both of them to the indexation phase by applying Multi-Probe Cross-Polytope LSH.

Two parameters need to be tuned at this level, the number of hashing functions to apply by Cross-Polytope LSH, and the number of probes to consider by Multi-Probe. The choice of these two parameters is explained in section 4

This step outputs two group of indexes, the group of textual column indexes, and the group of non-textual column indexes. Each column index is represented by an array of 2 dimensions, where each row represents the result of the corresponding hashing function,

and each column represents an element of the probing sequence for this hashing function result.

The figure 2.1 illustrates the process of preprocessing and computing the indexes of a dataset.

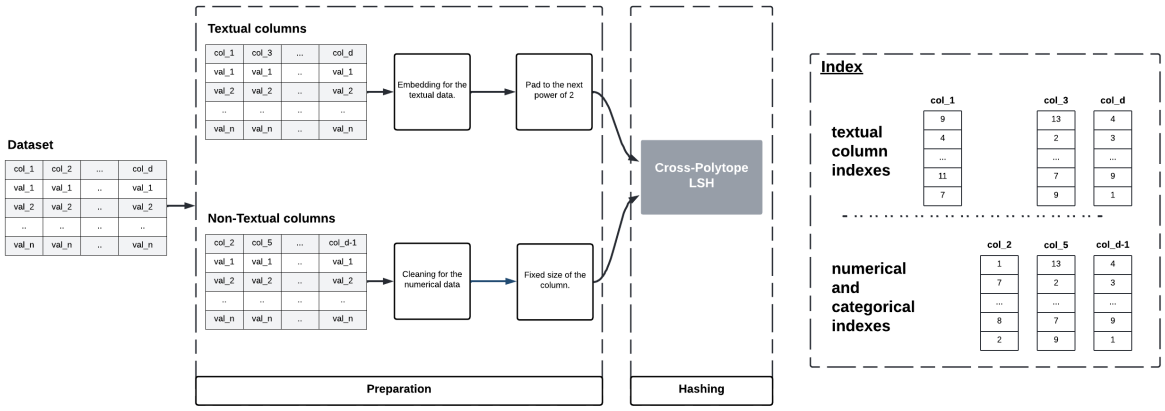

Figure 2.1: Indexation process illustration

2.1.3 Storing index

The indexes computed in the previous step need to be stored in order to be used lately in similarity detection. A key is assigned to every dataset in Talend Data Inventory, this key is used to store the indexes corresponding to each dataset. The indexes are stored in a dedicated database which contains the following attributes:

- **dataset_id**: Integer. The ID of the corresponding dataset.
- **column_name**: String. The name of the index column.
- **column_type**: String. The type of the index column.
- **signature**: Bytes. The encoding of the index.

2.2 Similarity Detection

This second part of our solution is meant to proceed for the similarity detection between two dataset indexes. As described previously, we scale the similarity detection between dataset to the similarity detection between their columns, and from the column similarities we can give a score representing the dataset similarities.

The process of detecting similarities between columns is done by the following steps:

- Use two groups of bucket, the first one for the textual columns and the second for the non-textual column.
- Assign each column to a number of buckets depending on its calculated index and its type.

- Measure the collision rate between the columns that share at least one bucket to give their similarity measure (a score between 0 and 1).

The similarity measure given by this method is inversely related to the angular distance, meaning that a score close to 1 represents a high similarity, and a score close to 0 represents a low similarity.

2.3 Implementation

To implement the solution explained in the two previous sections, five different components have been used:

- **Preprocessing**: It consists of the classes responsible for the preprocessing phase, its main class is *DataPreprocessing* that assures the different types of preprocessing (normalization, sampling, embedding..) with the use of two other classes, *Embedder* and *EmbeddingModel* used to embed the textual columns of a dataset.
- **Hashing**: It is the component that assures the indexation process, two classes have been used to compute the indexes: *CrossPolytope* and *MultiProbeCrossPolytope*, while the class *Index* is used to represent the index computed for a dataset,
- **Storing**: It consists of one class that manages the storing of the computed indexes
- **Query**: This component gives the features used to detect the similarities between indexes through the class *LSHQuery*, and to manage their results through the class *QueryResult*
- **Plotting**: With this component, the results given by *QueryResult* are represented graphically with two types of graphs.

The figure 2.2 represents the class diagram with its different components as explained in the previous paragraph.

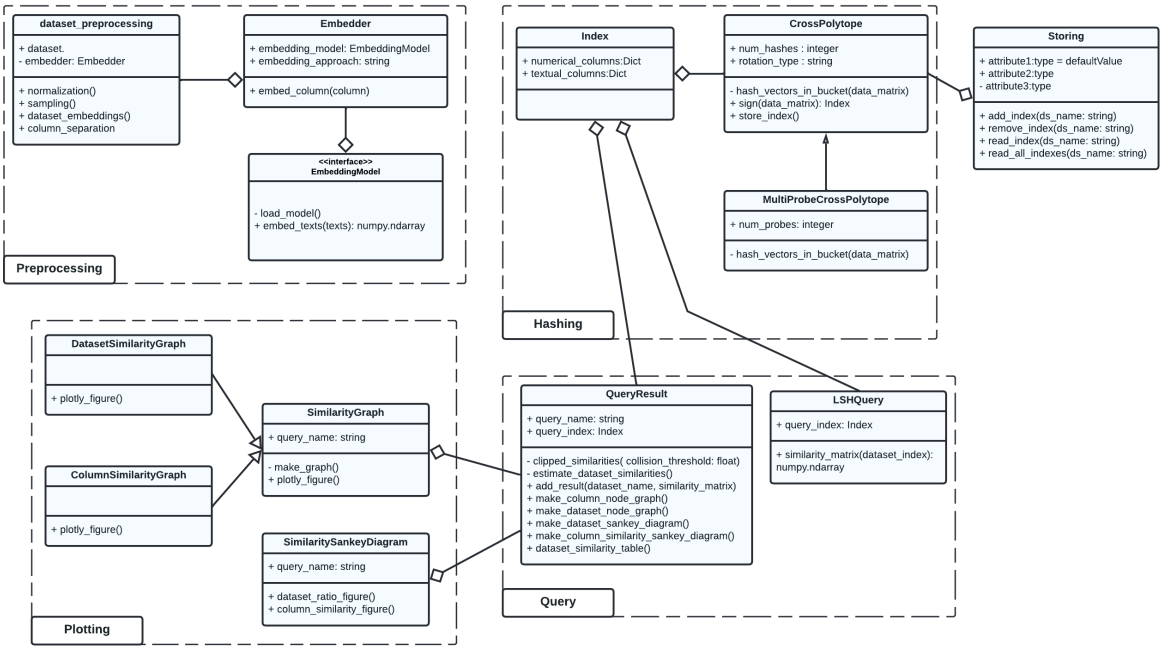

Figure 2.2: Class Diagram

Chapter 3

Realization

In order to implement the proposed solution, we have proposed a proof of concept which is a web application built with streamlit that demonstrates the different features that can be offered.

The use case that we consider in this proof of concept has the following specifications:

- The user can select a dataset to be added and indexed.
- The user can have the previously indexed datasets.
- The user can select a dataset, that we call query dataset, and measure its similarities with the other ones.
- The computed similarities are presented in two sections:
    1. Dataset similarity representation: It orders the dataset by their similarity scores with the query datasets.
    2. Column similarity representation: It shows the details of the column similarities between the query dataset and a dataset that the user selects.

3.1 Technologies and tools used

In order to implement the proposed solution, we have opted for the following technologies, tools and frameworks:

3.1.1 Python

Python is a structured, open source, multi-paradigm, multi-platform programming language that runs on all major operating systems and computing platforms. By offering high-level tools and an easy-to-use syntax, it greatly optimizes the productivity of programmers and has become a leading language in exploratory data analysis and software development.

Python have been chosen for implementing this project for the following reasons:

- It provides a set of libraries, in the machine learning domain, which simplifies the development of our project.
- Python manages its resources (memory, file descriptors) without the intervention of the programmer, by a reference counting mechanism.

- The Lab team of Talend has already used Python in most of their projects, this makes it easier to understand, collaborate, scale and integrate our solution in the future.

We follow up by presenting some of the used libraries during this project:

1. **Numpy**: A library used to manipulate matrices, multidimensional arrays, vectors and polynomials, it has a large number of mathematical functions that can be applied directly to the structures mentioned above. In this project we have used numpy to process matrices and arrays.

2. **Jax**: Following its definition in its official repository (Google, 2022), Google JAX is a machine learning framework for transforming numerical functions, the goal of using this framework is to take advantage from its version of Numpy that provides us with the possibility of doing the different calculations on GPU. It shows up that this choice is accurate if we take in consideration the number of matrix multiplication that we have to do in Cross-Polytope.

3. **Pandas**: Pandas is a python library that allows to easily manipulate data in form of data tables that we call DataFrame with column and row labels. It provides us with some interesting easily used features to read, analyze and complete some preprocessing operations.

4. **IGraph**: It is a library allowing to define and manipulate graphs, it is written in C programming language and can be used with Python or R. We use it in our solution to define the graphs representing the results of similarities between columns and between datasets

5. **Plotly**: Plotly is a complete library for creating static, animated and interactive visualizations in Python. It is used in our solution to represent the graphs defined with IGraph with an interactive figures in the streamlit application.

6. **Transformers**: We use the Transformers library which is a state of the art pre-trained model library for natural language processing (NLP). The library currently contains pre-trained model weights, user scripts and conversion utilities: Bert, Roberta....

3.1.2 Streamlit

It is an open source framework in Python language dedicated for data scientist and machine learning engineers to create complete web applications in a short time as it provides a set of components that we need to have an interactive layout. It also provides a compatibility with different Python libraries which make it easier for building demonstrations that are not exclusive to data scientists.

Streamlit has also a cloud platform that gives its users the possibility to deploy, manage and share their apps.

3.1.3 TensorFlow Hub

It is a repository of trained machine learning models ready for downloading, fine-tuning or reusing in different projects, with the possibility to be deployed everywhere.

We used this library to test different embedding models and choose the most suitable one in order to get a representation for the textual columns.

3.2 Architecture

The proof of concept has been implemented using two layers, the logic layer which is responsible of providing the features of indexation, storing, querying and similarity representation, while the second layer, the presentation layer, is responsible of building the user interface that offer the user the possibility to add and index a new dataset, display the stored indexes, look up for similarities and getting their representations.

The figure 3.1 shows the organization of the project folder where *data_fingerprinting* represents the logic layer, and *visual_demo* represents the presentation layer.

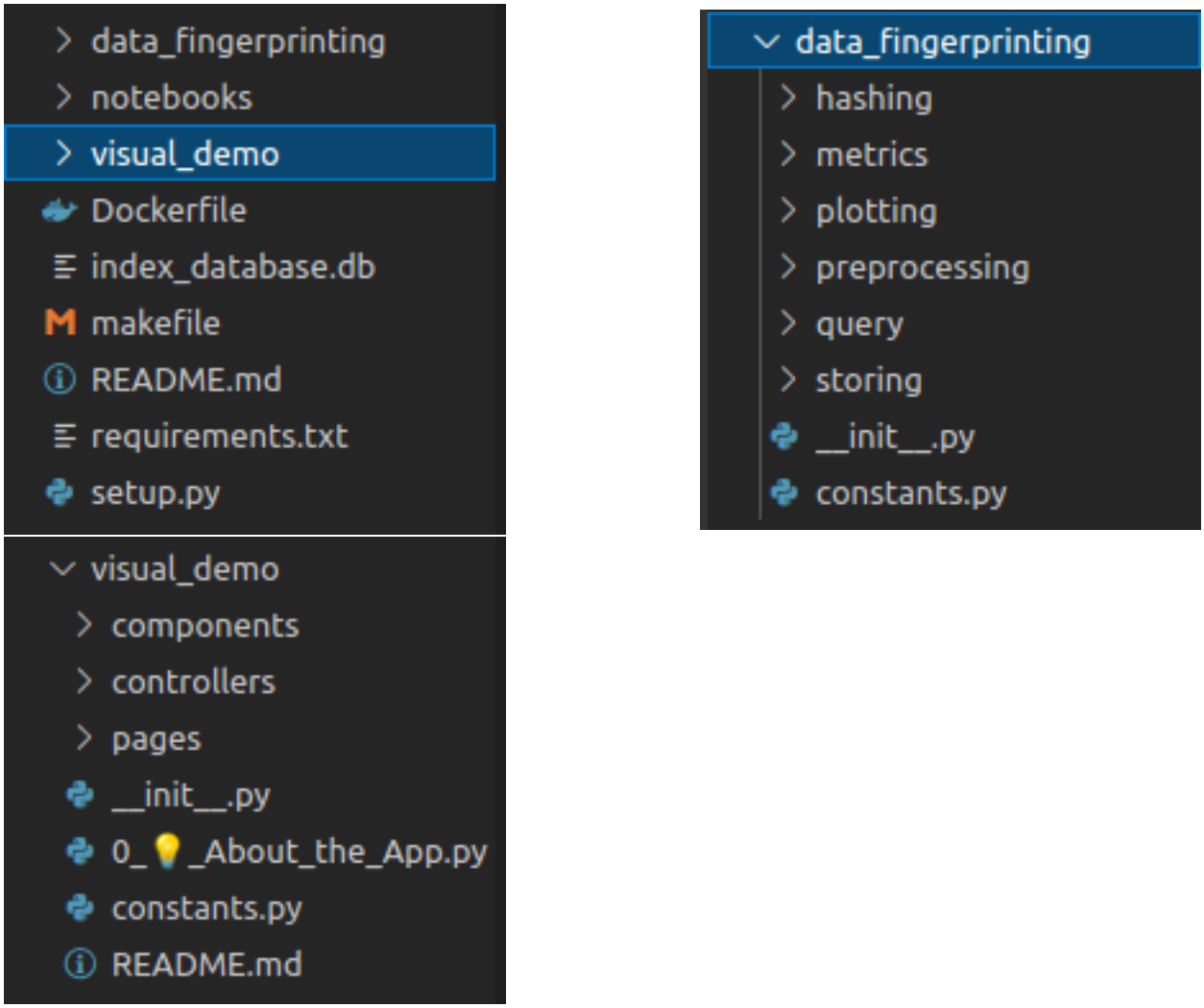

Figure 3.1: The organization of the project folders.

The logic layer contains the five components introduction in the class diagram in section 2.3 which are: Preprocessing, Hashing, Query, Plotting, and Storing. We use SQLite database for storing and reading the indexes, it mainly interacts with the Storing component. The presentation layer is divided into two components, the view component to build the user interface which is a multi-page layout, and the controller component which provides all the interactions between the view component and the logic layer.

The figure 3.2 shows architecture of the proof of concept built to demonstrate the functioning of the similarity detection process with LSH.

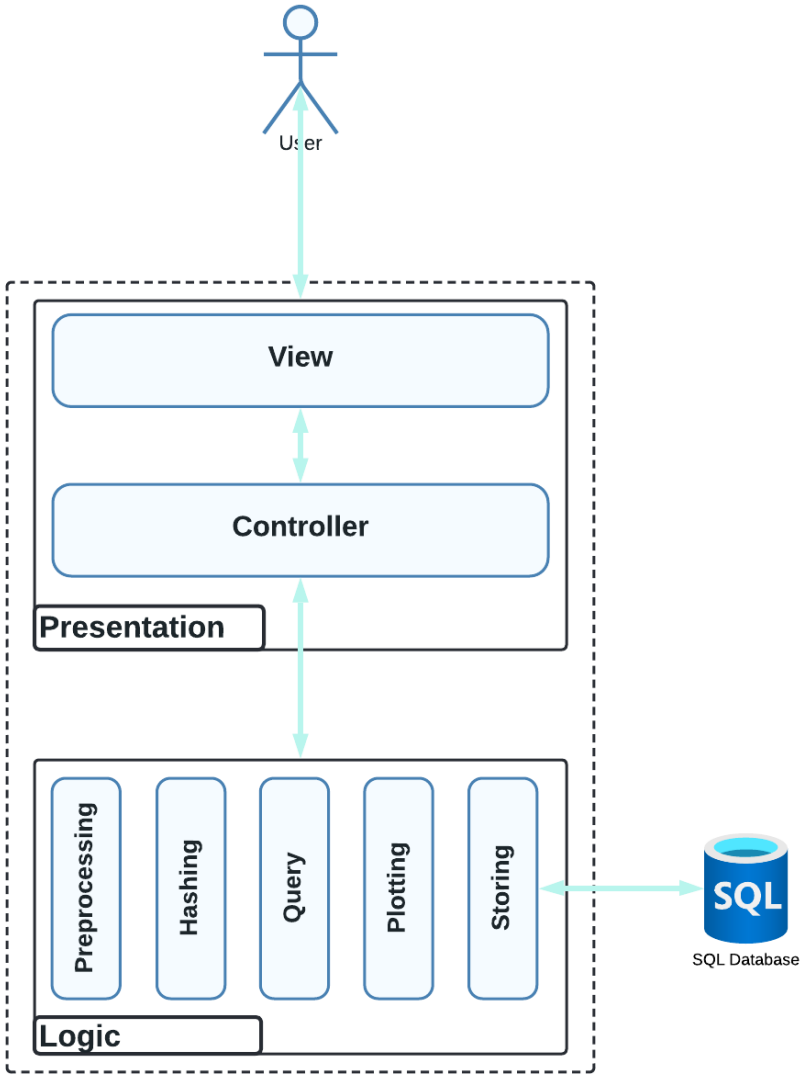

Figure 3.2: The architecture of the proof of concept

Chapter 4

Experiments and results

4.1 Test and Experiments

In this section, the focus is on finding the combination of hyper-parameters and embedding approach leading to the best similarity estimation. To do this we do our experiments under two metrics.

- **Embedding approach metric**: It consists of applying the different embedding approaches proposed in (mention the section when it's done) to get the textual column representations of each dataset couple, and then measuring the similarities between these representations to compare them with the ground truth labels 4.1. The similarity are measured with the theoretical collision probability that we estimate by cross-polytope LSH:

$$P(h(x) = h(y)) = \exp -\frac{\pi^2}{4 - \pi^2}. \ln d \tag{4.1}$$

  .

- **Cross-Polytope hyper-parameters:** It consists of estimating the similarities with cross-polytope using different values for the two parameters: number of hashing functions to use, and the number of probes, and computing the error of this estimation to the theoretical collision probability (4.1).

The set of labelled similarities : It is a dataset containing different sets of datasets associated with lists of columns that can be matched. It is a human labelling done in a previous project at Talend. This labelled datasets are organized as the following example:

dataset 1	**dataset 2**	**column dataset 1**	**column dataset 2**
students	clients	Student Name	client name
students	clients	inscription data	join date
students	clients	Gender	gender
books	publications	release data	publication date
.	.	.	.
books	publications	title	title

To each couple of labelled dataset with their respective similar columns, a label matrix is associated to represent their similarities with the following process:

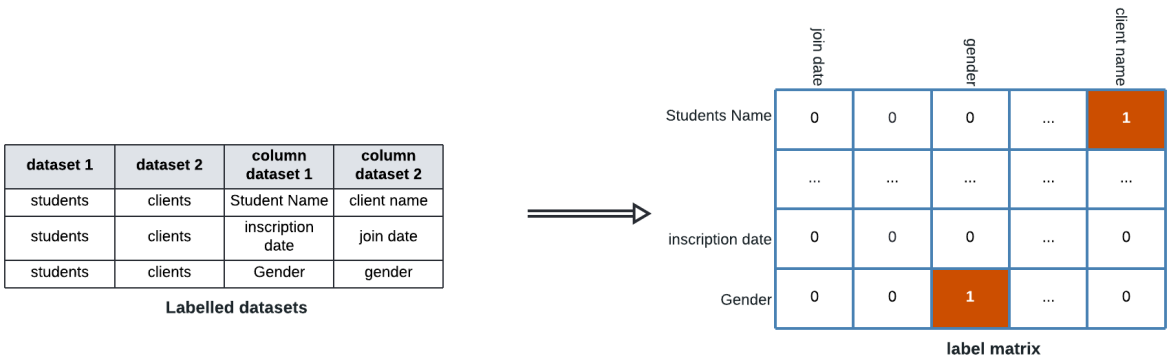

Figure 4.1: Label matrix generation from the labelled column similarities

4.1.1 Embedding approach metric

In this section, the goal is to measure at each time the accuracy of detected similarities compared to the validated ones (in the labelled similarities dataset), with changing the embedding model and the embedding approach.

The embedding models:

- Universal Sentence Encoder Multilingual (USEM)
- Paraphrase Multilingual
- Neural-Net Language Models (NNLM)
- Sentence transformer: distiluse-base-multilingual

The embedding approaches:

- **Approach 1**: (or Concatenate and embed) Concatenate the texts and do the embeddings
- **Approach 2**: (or Concatenate embeddings) Perform the embedding per cell and concatenate the results.
- **Approach 3**: (or PCA on embeddings) Perform the embedding per cell, reduce their dimension with PCA and concatenate.
- **Approach 4**: (or Mean on embeddings) Perform the embedding per cell, and get the mean for each component in the result embedding vectors.

To compute this accuracy, we represent the estimated similarities between couples of datasets in a form of a matrix where each axis represent dataset columns. The next step is to compute for each dataset couple the accuracy of the estimated similarities between the columns that has been labelled similar.

To define this metric mathematically, let E be the estimation matrix, and L be the labelled similarity matrix, and 1_L the number of occurrences of 1 in L

$$metric = \frac{1}{1_L} \sum_i \sum_j L_{i,j} * E_{i,j} \tag{4.2}$$

The figure 4.2 shows an example of two datasets with 4 and 5 columns respectively.

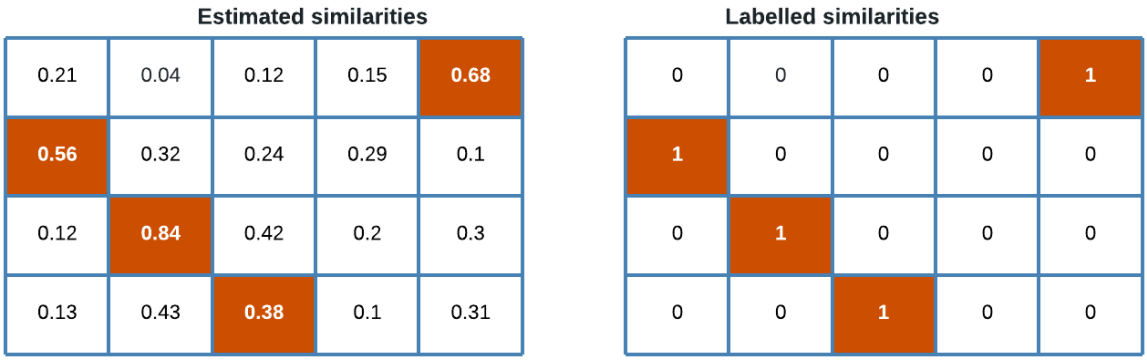

Figure 4.2: Embedding metric computing example

The metric in this case is calculated by picking up the similarities corresponding to labelled similarities and computing their means:

$$metric = \frac{1}{4}(0.68 + 0.56 + 0.84 + 0.38) = 0.615 \tag{4.3}$$

The table 4.1 presents the results of this test with every combination of embedding model with an embedding approach.

	USEM	**NNLM**	**Paraphrase Multilingual**	**Distiluse**
Concatenate and embed	0.301	0.580	0.409	0.365
Concatenate embeddings	0.044	0.06	0.085	0.107
PCA on embeddings	0.275	0.319	0.403	0.350
Mean on embeddings	0.494	0.583	0.572	0.554

Table 4.1: Results for the embedding choice metric

Some remarks can be done over this results:

- The best accuracies were around $0.56, 0.58$.
- The fourth approach gave the best results for each embedding model.
- The notion of angular distance loses its sense when the dimensions are too big case of approach 2 (Concatenate embeddings). As each cell with this approach will be in a dimension of 512 or 368 (depending on the embedding model used), the vector of concatenated word representations will become too large, which will make the angular metric loses some relevance, and also the concatenation of the embeddings won't give a good comprehension of the column content.
- The accuracies depend on the embedding model understanding of different forms of texts, as there may be some forms that are not understandable, for example: links, email addresses, acronyms, ect..

The configuration selected after this experiment is:

- **Embedding model:** Paraphrase Multilingual
- **Embedding approach:** Approach 4; perform the embedding per cell, and get the mean for each component in the result embedding vectors.

4.1.2 Indexation hyper-parameters

In this second test the goal is to find the best combination of number of the hashing functions to use in indexation and the number of probes to use in Multi-Probe LSH. The best combination is the one that gives the index with the minimum error of estimation of the angular distance. To do this, a list of values for each of the two variables and a set of 1000 dataset couples have been set, and the error for a combination of a number of probes and a number of hashing functions is computed by computing the mean absolute error for each couple of datasets and computing the mean of error over all the list of couples.

The pseudo-algorithm 1 describes the steps to complete this test routine.

Algorithm 1: Pseudo-algorithm for the similarity estimation test routine.

```
Input: num_probes: List of integer;
num_hashes: List of integer;
dataset_list: List of dataset couples.
error_matrix: matrix
for (num_probe, num_hash) in (num_probes, num_hashes) do
    errors = []
    for (dataset_1, dataset_2) in dataset_list do
        error = similarity_estimation_error(dataset_1, dataset_2) add_element(errors,
         error)
    end
    error_matrix[num_probe, num_hash] = mean(errors)
end
return error_matrix
```

The similarity_estimation_error function is computed by indexing the datasets in input with the corresponding configuration and compute the mean absolute error with the exact similarities.

The results of this test are represented in the following table where the columns correspond to the number of probe values and the rows correspond to the number of hashing functions.

The table 4.2 shows the results of this experiment, from which we can take the following points:

- The best configurations gave errors in the interval of $[0.45, 0.55]$.
- The results confirms the theoretical result in (Andoni et al., 2015) saying that by using a multi-probe technic we can improve the results of estimation without having to increase the number of hashing functions used, for example, the difference of estimation error between the case of 256 hashing functions and 1024 hashing functions is at the range of 0.003.

Number of Probes / Number of hashing func	**3**	**4**	**5**	**6**	**7**	**8**
16	0.084150	0.091189	0.102268	0.114099	0.126253	0.138592
32	0.069658	0.073694	0.084380	0.097009	0.110436	0.123476
64	0.060197	0.063477	0.075861	0.090637	0.105167	0.119083
128	0.054985	0.056040	0.069854	0.086120	0.101522	0.115687
256	0.049240	0.055169	0.072686	0.089780	0.105259	0.119369
512	0.047183	0.053220	0.072063	0.089319	0.104874	0.119079
1024	0.046043	0.052195	0.071337	0.088481	0.103913	0.117931

Table 4.2: Results for the multi-probe hyper-parameters testing

Bibliography

Andoni, A., Indyk, P., Laarhoven, T., Razenshteyn, I., and Schmidt, L. (2015). Practical and optimal lsh for angular distance.

Chi, L. and Zhu, X. (2017). Hashing techniques: A survey and taxonomy. *ACM Comput. Surv.*

Domingos, P. (2012). A few useful things to know about machine learning. *Commun. ACM*, 55(10):78–87.

Ertl, O. (2020). ProbMinHash – a class of locality-sensitive hash algorithms for the (probability) jaccard similarity. *IEEE Transactions on Knowledge and Data Engineering*, pages 1–1.

Google (2022). Jax. `https://github.com/google/jax`.

Hajebi, K., Abbasi-Yadkori, Y., Shahbazi, H., and Zhang, H. (2011). Fast approximate nearest-neighbor search with k-nearest neighbor graph. pages 1312–1317.

Indyk, P. and Motwani, R. (2000). Approximate nearest neighbors: Towards removing the curse of dimensionality. *Conference Proceedings of the Annual ACM Symposium on Theory of Computing*, 604-613.

Jafari, O., Maurya, P., Nagarkar, P., Islam, K., and Crushev, C. (2021). A survey on locality sensitive hashing algorithms and their applications.

Jégou, H., Douze, M., and Schmid, C. (2011). Product quantization for nearest neighbor search. *IEEE transactions on pattern analysis and machine intelligence*, 33:117–28.

OpenGenus-Foundation and Aditya, C. (2018). Manhattan distance.

OpenGenus-Foundation and Aditya, C. (2019). Minkowski distance [explained].

Research, G. and Ads (2020). Evaluation of cohort algorithms for the floc api.

Silpa-Anan, C. and Hartley, R. (2008). Optimised kd-trees for fast image descriptor matching. In *2008 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–8.

Wang, M., Xu, X., Yue, Q., and Wang, Y. (2021). A comprehensive survey and experimental comparison of graph-based approximate nearest neighbor search. *CoRR*, abs/2101.12631.

Yu, Y. W. and Weber, G. M. (2022). Hyperminhash: Minhash in loglog space. *IEEE Transactions on Knowledge and Data Engineering*, 34(1):328–339.

Zhu, E., Nargesian, F., Pu, K. Q., and Miller, R. J. (2016). Lsh ensemble: Internet-scale domain search.