Masaryk University
Faculty of Informatics

Implementation of Learned Metric Index

Master's Thesis

Bc. Terézia Slaninákova

Brno, Fall 2020

Masaryk University
Faculty of Informatics

Implementation of Learned Metric Index

Master's Thesis

Bc. Terézia Slaninákova

Brno, Fall 2020

This is where a copy of the official signed thesis assignment and a copy of the Statement of an Author is located in the printed version of the document.

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Bc. Terézia Slaninákova

Advisor: RNDr. Matej Antol

Acknowledgements

First and foremost, I would like to thank my supervisor, RNDr. Matej Antol, for including me in his team to investigate questions from which this thesis arose, for his vision, guidance, patience, and constructive feedback. Furthermore, my gratitude goes to my other team-mates: doc. RNDr. Vlastislav Dohnal, Ph.D. for his plentiful advice and endless patience with a newcomer to his area of expertise, and Mgr. et Mgr. Jaroslav Oľha for asking the right questions that repeatedly put us on the correct path.

Last but not least my thanks goes to my loved ones for supporting me through the last year and a half of working on this problem.

Abstract

High-dimensional data, such as images, form a significant portion of digital data collections. Naturally, as the requirements for increasingly efficient storage and searching are growing, more research is being dedicated to improving solutions for these very problems. Due to the nature of multimedia data, traditional database systems fall short to provide such solutions, however, we have seen promising results from the domains of *Similarity searching* and *Content-based retrieval*. The goal of both is to organize high-dimensional data in a way that allows for as fast and as accurate retrieval as possible. However, such systems are built on rigid structures with little space for optimizing for a specific dataset. With the goal of improving the accuracy and efficiency of answering queries, in this thesis, I introduce a machine learning-enhanced indexing structure for complex data. Specifically, I describe the concept, the implementation, and present the experimental results on two real-world datasets in comparison to the two state-of-the-art indexing approaches – M-tree and M-index.

Keywords

similarity searching, indexing, machine learning, learned index, metric spaces, complex data

Contents

Introduction **1**

1 Indexing structured data **3**
1.1 *The searching problem and basics of indexing* 3
1.2 *B-Trees and other data structures* 4
1.3 *The motivation behind Learned indexes in structured data* . 5

2 Searching in the metric space **9**
2.1 *The similarity searching problem* 9
2.2 *Similarity queries* 10
2.3 *Indexes in the metric spaces* 11
2.3.1 M-tree 11
2.3.2 M-index 12
2.3.3 Searching in M-tree and M-index 12
2.4 *Learned indexes in metric spaces* 14

3 Learned Metric Index **15**
3.1 *Problem statement* 15
3.1.1 Enforcing reasonable bucket size 15
3.1.2 Supervised training 17
3.2 *Learned Metric Index architecture* 19
3.3 *Training phase: Building LMI* 20
3.3.1 Naive learned metric index 20
3.3.2 Adjusted learned metric index 21
3.4 *Searching phase: Approximate k-NN search* 22

4 Implementation of Learned Metric Index **25**
4.1 *Implementation details* 25
4.2 *Choosing a classifier* 28
4.3 *Space and time requirements: A case study* 32
4.4 *Hyperparameter search* 35

5 Experimental results **39**
5.1 *Experiment setup* 39
5.1.1 Visualizing experiments 39
5.2 *Datasets* 41

5.2.1 CoPhIR dataset 41
5.2.2 Profiset dataset 42
5.2.3 Data preprocessing 42
5.3 *Experiments* 42
5.3.1 CoPhIR 43
5.3.2 Profiset 46
5.4 *Experiments with alternative approaches* 49
5.4.1 Queries out of dataset (OOD) 49
5.4.2 Unsupervised learning 50

6 Discussion **55**
6.1 *Hypothesis statement and LMI design* 55
6.2 *Experiments' setup* 56
6.3 *Optimization of achieved results* 56
6.4 *Future research directions* 57
6.4.1 Unsupervised learning 57
6.4.2 Queries out of the dataset 58
6.4.3 Strategies for forming a priority queue 58
6.4.4 Optimizing building cost 58
6.4.5 Resource efficiency 59

7 Conclusion **61**

A Remaining experimental results **63**
A.1 *CoPhIR experiments* 63
A.2 *Profiset experiments* 65

B Hardware specification **67**

C Acknowledgements **69**

Bibliography **71**

Introduction

The field of data management is arguably one of the oldest ones in computer science. For as long as there was data, there was a need to organize it and perform basic operations on it. One may even argue that such systems existed before computers, for example in the minds of local librarians. First commercial *Database management systems* appeared in the late 1960s and since then became a backbone to virtually all businesses and institutions we know.

When we look at databases as an infrastructure for data, they are built to be mostly general-purpose. They work irrespectively of the underlying data they store and this rigidity has advantages – for example, there are specific guarantees of returned results for a range request run on a B-Tree index structure. This is in clear contrast to how machine learning works with data, where each trained instance of any algorithm/model is heavily data-dependent. In recent years, a significant amount of research work has addressed the possibilities of enhancing or even replacing standard database models and approaches using machine learning. One of the most inspirational, out of multiple research ideas, is *The case for Learned Index Structures* [1], which focused on the problem of database indexing. The authors argued, that database indexing can be effectively thought of as a (machine) learning problem. Their model, composed of multiple neural networks, was able to take advantage of the underlying data distribution of a given database and performed faster querying than conventional indexing data structures such as B-Trees or Hash Maps.

In this thesis, I introduce and study a structure built on the same premise – *LMI – Learned Metric Index*, in the context of large-scale similarity searching in multimedia data and benchmark it against traditional community-acknowledged index structures: M-Tree and M-Index.

The outline of this thesis is the following: In Chapter 1, I introduce the problem of indexing, its current state, motivate the need to explore approaches outside of conventional static indexes, and present the research done on this topic.

In Chapter 2, I focus on the field of similarity searching, its native data structures, and unique challenges as compared to database indexing in general.

Chapter 3 is dedicated to the detailed description of LMI, the specifics of the architecture, building, and searching procedures.

In Chapter 4, I present the main contribution of my work - the implementation of LMI with the description of the codebase, the development process, and the machine learning pipeline.

Comments on the experiment design and presentation of achieved results are shown in Chapter 5. Here I demonstrate that machine learning models are indeed able to learn useful data representations from a dataset, that make them very competitive to M-Trees and M-Indexes in terms of efficiency and accuracy of a k-NN search.

Chapter 6 contains an analysis of the results. I summarize my efforts, comment on the benefits and shortcomings, the difficulties, and the lessons learned while working on this thesis. Equally, I outline the future work that I would like to see done in this research area.

This thesis standardly wraps up with a Conclusion contained in Chapter 7.

1 Indexing structured data

This chapter is dedicated to the analysis of indexes as used in traditional database systems.

1.1 The searching problem and basics of indexing

Searching is a crucial operation branching to all areas of computer science. Understandably, many efforts were put into developing the most effective algorithms and heuristics to search a given collection, particularly when the collection is of a bigger size. The algorithm itself is, however, only one piece of the puzzle. Even if the collection is small, there are noticeable differences in the efficiency of any given searching algorithm depending on the specifics of the data. For example, if the data is not sorted, searching will be slower, and we oftentimes incorporate a sorting procedure beforehand to account for it. Furthermore, if the data is not ordinal, searching is not as straight-forward anymore.

The problem of searching within database systems is motivated by hardware and is built around minimizing the number of disk accesses when processing a query. The data structure addressing this is a *database index*, depicted in Figure 1.1. At the cost of using additional storage space, the role of an index is to map search queries (*keys*) to database records, particularly without having to access more than just a small portion of all the records in the process [2].

The basic division of simple indexes is into *dense* and *sparse*. In a *dense index*, every record is referenced by an entry in the index file. This type of index is advantageous when it can fit in the main memory and the data can not. That way we can find any record given its search key with one disk I/O. *Sparse index*, on the other hand, references only a subset of entries and therefore consumes less space, but usually takes more time to find a record in the data file [2].

Indexes may also span into multiple levels – in that case, we call them *Multilevel indexes*. These hierarchical structures are useful when one index file might be too large and could not fit in the main memory anymore. In that case, many disk I/O's would be required to get to the record. The idea is to put the more efficient index to the front.

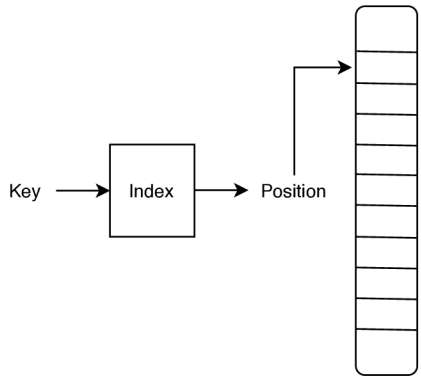

Figure 1.1: Conceptual depiction of a key being mapped into a position within database.

1.2 B-Trees and other data structures

B-trees are the most common form of an index in database systems [2]. A B-Tree, shown in Figure 1.2, represents a family of data structures with a variant used the most: a B+ tree. B-Trees are *balanced* trees with each node (including the root) consisting of several *keys*.

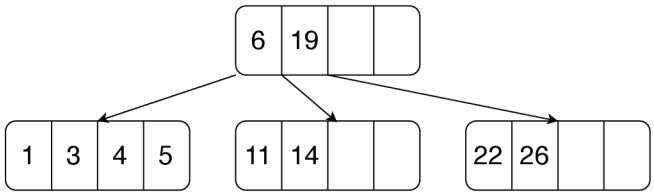

Figure 1.2: Example of a B-Tree of order 4: Every node has at most 4 children, a non-leaf node with k children contains k-1 keys.

A big advantage of a B-Tree is its time complexity of $\mathcal{O}(\log n)$ for search, insert, and delete operations. Its structure also allows it to store a billion key values for order of 64 at a height of 5, so the number of disk accesses is reasonably minimized as well. These properties therefore directly translate to fast retrieval [3]. They are also particularly suitable for range requests. Other structures to be considered in database design are *Hash maps* and *Bloom filters*. In brief, one particular benefit of Bloom filters over other structures is their fast ability to check for

the existence of a record, while Hash maps perform noticeably well in single-key look-ups [1].

1.3 The motivation behind Learned indexes in structured data

Database systems are primarily built around and on structures described in the section above. As mentioned in the Introduction – they provide a generic solution that fits a variety of use cases for data storage, since building a specialized set of algorithms to provide basic operations for every dataset would be cumbersome.

However, this 'one-data structure-fits-all' approach often exhibits shortcomings. Consider an example presented by Prof. Tim Kraska, the lead author of *The case for Learned Index Structures* [1], at the Data science summit [4]:

Problem 1. *Indexing dataset composed of all integers from 900 to 800 million*

This could, by all means, be done with a B-Tree:

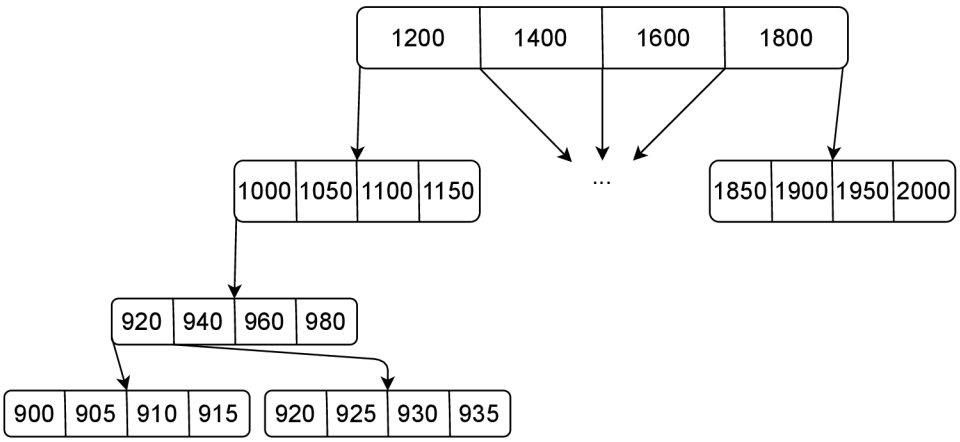

Figure 1.3: Portion of a B-Tree indexing integers from **Problem 1**.

However, using a simple look-up array in the form `data_array[key-900]` is arguably more elegant and better-performing than a B-Tree. One other way to look at this array is that it is a *linear regression* model

with $slope = 1$ and $intercept = -900$. The same would hold true for a similar problem: Indexing all even integers from 900 to 800M, where the optimal indexing structure would be the following array: `data_array[(key-900)/2l]`. So although the robustness and complexity of many state-of-the-art indexing structures, such as B-trees, make them usable with a wide range of datasets and gives them comfortable complexity guarantees, they do not represent the most suitable model for all types of datasets [1].

It is clear that this is an example of a very specific distribution, where we can see the pattern, but what if the same is true for data with a variety of distributions that are not so straight-forward? As we know, this is where machine learning could excel, since its job is to find patterns in the data, that humans may have problems discovering and can not program dedicated solutions to explicitly.

As shown in the previous section, databases are not tied to a single data structure, rather, a lot of them are known and used based on their suitability for particular requests. One might then consider models from the machine learning realm as yet another extension of this existing family of models.

This is one of the main arguments of *The case for Learned Index Structures* [1] – the fact that all existing index structures can be viewed as models, that can be replaced by any mathematical concept of a model (including machine learning). Furthermore, the authors show that machine learning models can be trained for the same purpose of answering database queries while presenting a number of performance benefits.

The authors deployed this idea in the context of B-Trees in a database system with sequential data ordering. The structure of a B-Tree was maintained, but each internal node of the tree was replaced by a learned model. The query look-up worked analogically to a regular B-Tree, by gradually narrowing down the range of possible results as the query descended down the structure. The difference was in how this narrowing was achieved: B-Tree does it by comparing the query to the objects stored in the internal nodes, learned index classifies the query using a learned model, with each possible class representing another learned model on the next level of the hierarchy. Each successive layer of models is trained on a narrower segment of the data, increasing the

specificity of the classification results. The finding of the authors was that their approach provided many benefits over a standard B-Tree implementation, especially in terms of look-up time and index size.

To paint the whole picture though, the authors discussed possible drawbacks of a machine learning approach to this problem: Firstly, machine learning algorithms cannot provide the semantic guarantees that are traditionally associated with static indexes, since they are inherently probabilistic. This introduces uncertainty to the retrieval process, thereby slightly decreasing precision and recall, and removing any guarantees that all of the queried data will be retrieved. The second concern applies to the fact that machine learning models (mainly neural networks) are notoriously slow to train.

As a primer for the next chapter, from the perspective of *Similarity searching*, the first drawback is not as significant. The type of searching performed in the similarity searching domain is almost exclusively approximate, therefore the need for specific semantic guarantees is not as strong as in the precise searching setup that we know from traditional database systems. The second concern is more practical and I address it specifically model to model in Chapter 4.

2 Searching in the metric space

This chapter contains an overview of the problems addressed by similarity searching and how learned indexes can be applied to this domain.

2.1 The similarity searching problem

Searching in indexed structured data, presented in the previous chapter, is straight-forward in the sense that results are exactly equal to the search queries. However, exact-match retrieval, most prominently associated with traditional database systems, is itself not sufficient to address the needs of ever-expanding digital libraries storing mainly multimedia data. A broader definition of a "match" when searching in such collections is therefore needed and this is what the field of similarity searching addresses. By utilizing the mathematical notion of *metric space* as a basis for the definition of nearness, the problem of searching is defined as follows:

Problem. *Given a universe $\mathcal{D}$ of data objects, the distance measure (function) $d(\cdot,\cdot)$ and a set $X \subseteq \mathcal{D}$ of n elements, structure the data such that the proximity queries can be answered efficiently* [5].

Where X is the collection, or dataset at hand and *structuring* directly translates to using an index to reduce the number of distance computations needed when searching.

The distance function d is an operation applied between two objects to evaluate their proximity. To consider it valid in the similarity searching domain, it must satisfy so-called metric postulates for $x, y, z \in \mathcal{D}$:

- $d(x, x) = 0$ - *reflexivity*
- $d(x, y) > 0$ - *strict positiveness*
- $d(x, y) = d(y, x)$ - *symmetry*
- $d(x, y) \leq d(x, z) + d(z, y)$ - *triangle inequality*

Many of such distance measures are chosen based on the type of dataset, for example, the well-known *Euclidean distance* is chosen primarily for numerical vectors, *Levenshtein distance*, typically used in the field of information retrieval, is chosen for string vectors and *Jaccard's coefficient* works well for sets.

2.2 Similarity queries

Queries are typically objects from the collection, that are used as points of interest within a database and their specific function depends on the querying algorithm, also known as a similarity query. The two most popular ones are **Range query** and **k-nearest neighbors query**. Range query $R(q, r)$ retrieves objects that are located within a radius r from the query. To do this, one must find the distance from the query to all objects in the set $\mathbf{X}$ – an operation, that can be relatively costly depending on the cardinality of $\mathbf{X}$ and the chosen distance measure. k-nearest neighbors query $kNN(q)$, as the name suggests, is based on the k-nearest neighbors algorithm, which returns k objects with the lowest proximity to the query. Again, the distance from the query to all objects in the set $\mathbf{X}$ is needed.

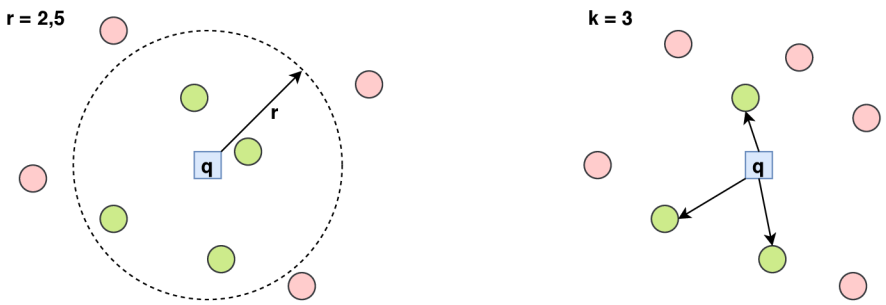

Figure 2.1: Illustration of a range query (left) and a k-NN search query (right).

In this work, the k-nearest neighbors querying algorithm will be used both with learned indexes and similarity searching indexes.

2.3 Indexes in the metric spaces

There are quite a few indexing structures in the similarity searching domain, many of which have been gradually improved upon over the last decade or two: a family of methods from the original M-tree [6], Spatial Approximation Trees [7], or Cover Trees [8]. These structures divide data objects into groups or clusters respecting their distribution in space. They have sub-linear search time $\mathcal{O}(n^{\alpha})$, where $\alpha \leq 1$ depends on data distribution. Another prominent group of indexes is based on permutations of pre-selected anchor objects (pivots). M-index [9] and its later improvement PPP-Codes [10] take k closest pivots for each object in the dataset and group them to form buckets stored on a disk. The latest research effort in this domain is represented by independent filtering techniques that can be plugged in to further eliminate accessing an excessive amount of data objects on a disk, e.g., Binary Sketches for Secondary Filtering [11]. A comprehensive summary of all the indexing approaches can be found in [5].

M-tree and M-index were chosen as the most suitable index structures to base our research on. Both were equally used as bases for Learned Metric Index as well as baselines in Results, so in the following subsections, I introduce them in greater detail.

2.3.1 M-tree

M-tree [6] is a well-established index structure in similarity searching applications. It was conceived with support for dynamical data in mind and is often used as a reasonable baseline when developing new indexes. The process of building an M-tree starts by constructing a tree-like structure bottom-up, where each node is represented by a pivot and its covering radius. This tuple allows for recursive subdivision of the metric space into smaller and smaller regions enclosed by circles, as seen in Figure 2.2. The outcome of the partitioning procedure is a balanced tree, where all the objects are stored in the leaf nodes and inner nodes are used for navigation within the structure.

2.3.2 M-index

M-index is a more recent addition to the shelf of metric space indexes, which unlike M-tree, divides the metric space in a top-down manner by using *Voronoi partitioning* together with a priori selected pivots. Here, the subtrees are formed recursively by assigning objects to the closest pivot, which results in an unbalanced tree structure.

2.3.3 Searching in M-tree and M-index

For the purposes of putting these two techniques into the context of this thesis, it is important to point out that both of them use the concept of pivot objects and distances of other objects to pivots when searching in the metric space. This means that the traversal of the tree starts in areas defined by the nearest pivots, giving us a higher chance of finding the nearest objects and therefore eliminating the number of distance calculations needed to arrive at desired nodes in the tree. A big bottleneck of this procedure are the distance computations themselves, e.g., Euclidean distance has linear time complexity, and depending on the number of tree nodes to be processed, that are stacked in the so-called *priority queue*. This operation can be called tens to thousands of times before a satisfactory answer is returned. A big advantage of any indexing structure is therefore if it can severely limit the number of distance computations or get rid of them altogether.

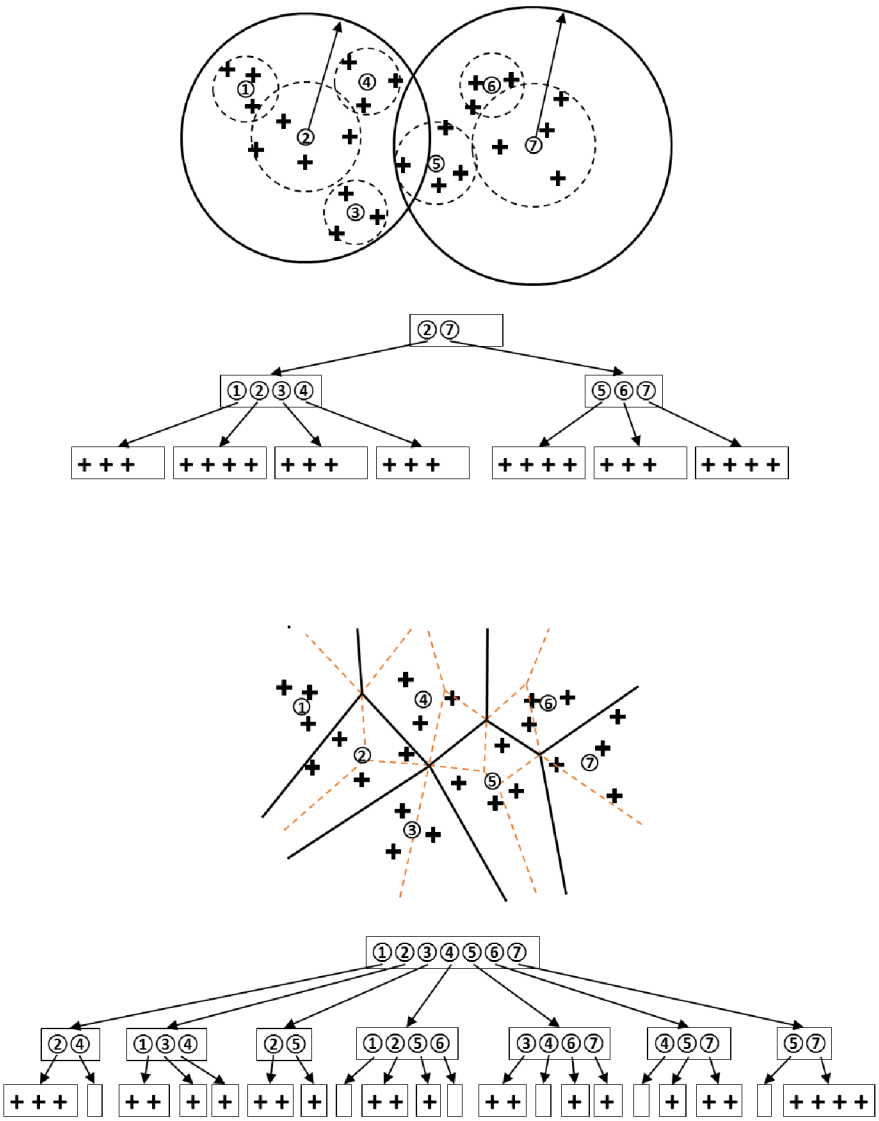

Figure 2.2: Paritioning procedure of M-tree (top) and M-index (bottom) [12].

2.4 Learned indexes in metric spaces

As far as we know, there is no reason why the idea of training hierarchy of learned models, as briefly described in the Section 1.3 could not be applied in the domain of similarity searching in metric data. However, there are several key differences between index models designed for structured data and their counterparts in metric spaces, all of which pose unique challenges when trying to apply a learned approach.

Firstly, index models in metric spaces tend to suffer from considerably worse performance due to the *"curse of dimensionality"*, wherein the increasing number of dimensions decreases the variation of pairwise distances between the data objects. This, as well as other properties of complex data, renders their organization and retrieval quite challenging.

Secondly, when querying metric spaces, it is commonly accepted that the answers often do not (or even can not) attain perfect accuracy, simply because of the type of queries that metric spaces usually address. Consider the difference between a query looking for 'John' in a traditional database, and a query searching for an audio file that is the closest match to the provided recording. There are two main reasons why we may consider loosening our demands on accuracy. For one, the "similarity" between two real-life objects is subjective. Every model that determines the similarity between such objects, even if it were to perform with perfect accuracy, will only be an approximation bounded by the authors of the metric and their views on similarity. Secondly, we only use data descriptors which are themselves limited approximations of the data they represent. As a result, inaccuracy is often inherent in the data, even before an index is taken into account. Therefore, in typical real-life applications of similarity searching in complex data, a faster approximate result is usually preferred to a slower one with 100% accuracy.

3 Learned Metric Index

This chapter documents specifics of the *Learned Metric Index* - the problem statement, architecture, training, and searching procedures.

3.1 Problem statement

As mentioned earlier, the goal of this thesis is to construct *Learned Metric Index* (*LMI*), a probabilistic-driven index built with machine learning models, which should be a viable alternative to other indexes currently used in the metric space domain. Its role is therefore identical to the one of M-tree or M-index in regards to the data, i.e., **Given a dataset, partition it into a tree-like structure in a way that allows for fast and accurate retrieval**. Following the example set by Kraska et al. [1], in *the Learned metric index*, my colleagues and I chose to purposefully omit insert and delete operations and focus solely on search and its improvement. However, I briefly outline the possible performance regarding inserts in Section 5.4.

To assess the quality of LMI, its performance is measured in terms of the accuracy of its retrieval procedure. More specifically, the accuracy is measured as the average recall, i.e., precision of the k-nearest neighbors (k-NNs).

3.1.1 Enforcing reasonable bucket size

There are two factors that influence the speed of retrieval in any index. The first one is the time it takes for a query to reach the target bucket, which is directly related to the depth and width of the index and specifics of the querying algorithm – i.e., whether it is primarily breadth-first, depth-first, or a combination of the two. The second factor is the number of objects covered by a bucket since we need to linearly search the bucket to find the object itself as the final step.

For an illustration, consider an extreme example of a primitive index in Figure 3.1 on the left. When evaluating the query, the whole dataset would need to be sequentially searched. Such index is practically useless. The other extreme case would be a very wide and deep structure, shown in the same figure on the right. Here, each leaf node

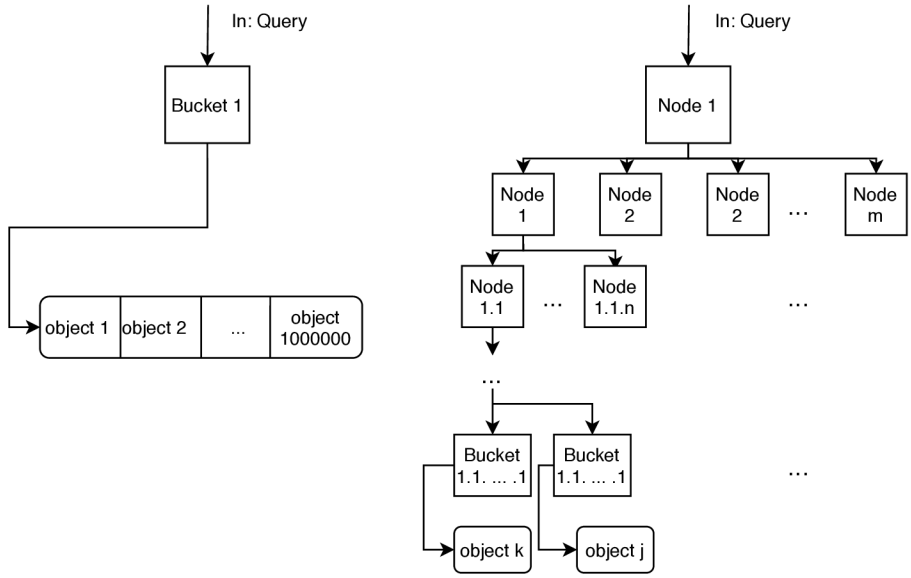

Figure 3.1: Two depictions of suboptimal indexes – primitive (and therefore useless) one on the left consisting of just 1 bucket and unnecessarily robust on the right.

would contain only 1 object. In this case, a sequential scan is fast, but there is a significant cost associated with navigating through the tree.

In theoretical parlance, Figure 3.1 reflects that we should aim for a logarithmic ($\mathcal{O}(\log n)$) or lookup complexity with the structure if possible. The same applies to optimal index construction in the field of database systems, as introduced by B-tree more than 30 years ago [13].

Translated into the speed of retrieval, an index consisting of 1 node would be slow to find the result, because even though the node will be visited fast, the cost of sequential search over the whole dataset is enormous. On the other hand, in a structure with 1 object per bucket, the sequential search would be instant, but the overhead in constructing this index and navigating throughout its structure would equally be a huge detriment.

3.1.2 Supervised training

In order to enforce that the speed-recall trade-off is reasonable in LMI, we chose to use an **existing index as a basis for the training**, making training LMI a **supervised problem**, rather than an unsupervised one (although one unsupervised approach was attempted and is briefly visited in subsection 5.4.2).

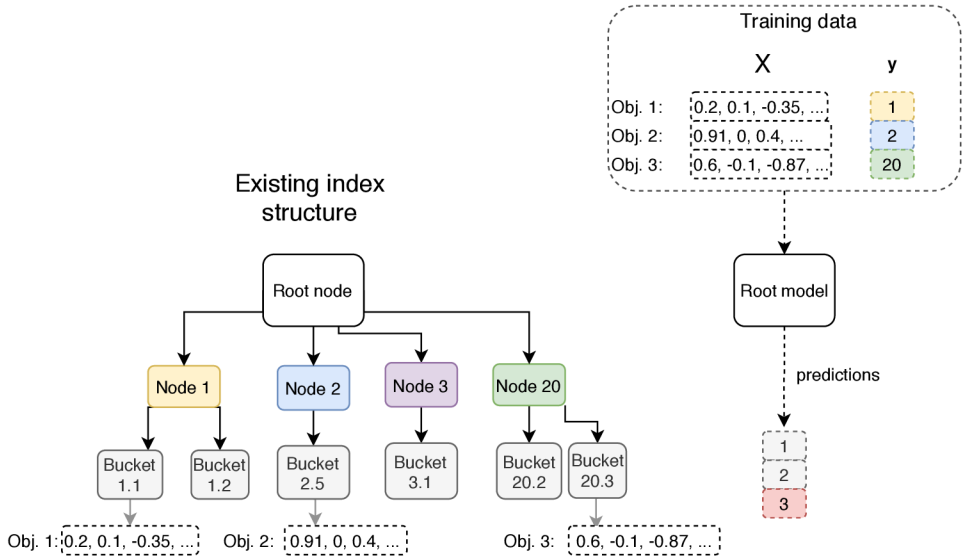

Figure 3.2: Example of the supervised training procedure with 3 objects. The 'y' labels are assigned based on position of the objects in the original index structure (on the left). Root model then outputs predictions that may or may not correspond with the original ground truth labels (e.g. notice that Object 3 was incorrectly classified into the class 3 instead of 20).

These original index structures are M-tree and M-index – commonly used indexing methods in metric spaces – that serve as models of performance comparison with different LMI setups (see Chapter 5). However, since the *objective function* is not to match the target labels, rather regroup the data so that the objects are accessed in a manner that prioritizes visiting similar objects together, by using an existing indexing structure as a ground truth we are at risk of overtraining the models to be good at finding only the query itself. As will be shown

later, I am using a number of approaches to mitigate this problem mostly by limiting the amount of overfitting any one model might be exposed to. At the same time, I try to balance the opposite extreme of underfitting the models, to overcome potential performance loss.

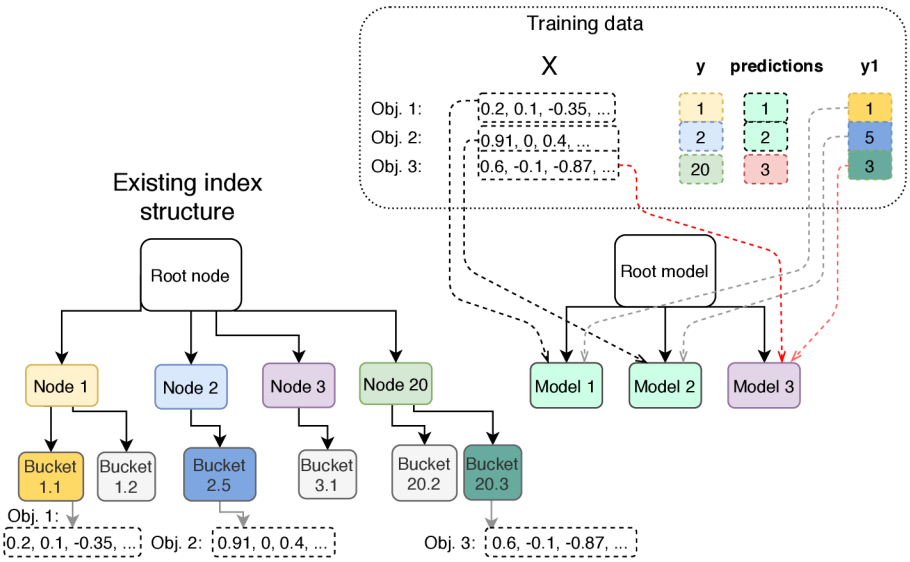

Figure 3.3: Example of the supervised training procedure at the second level. New nodes (children of Root model) are created based on predictions outputted by the Root model. Each new model is then trained, the training data being objects that are assigned to it based on previous classification. The training labels are the categories of the original index at appropriate level (here, the buckets denoted by darker colors). Red dotted line marks a misclassification, categorizing Object 3 into Model 3 instead of Model 20, as would be the case in the original structure

Figure 3.2 and Figure 3.3 show what having an "existing index as a basis for training" looks like in practice. All objects have assigned from 2 to 8 labels (depending on the dataset configuration), one for each level. Based on classification by the *Root model*, the objects descend to the appropriate child models. Figure 3.3 also shows one case of a misclassification – the third object "should" have been classified as 20, but was instead classified as 3.

As will be shown later, handling of the misclassifications is the main difference between the two training algorithms that will be introduced in Section 3.3. Figure 3.3 represents Algorithm 1 – so-called naive approach, where the misclassification continues to the predicted model and its training labels are used further in the misclassified model. This is a way for LMI to create new buckets, since e.g., in Figure 3.3, an object was misassigned to Model 3, which does not have a Bucket 3.3. It is being introduced in the training data originally from Node 20.

In Algorithm 2, misclassifications are being ignored in the training – they do not form training data of misclassified models and their respective buckets are determined after the training phase as the most likely bucket in the LMI. No new buckets are created with this approach.

3.2 Learned Metric Index architecture

Based on the example set by Kraska et al., *LMI* is an index structure whose goal is the same as the one of any other indexing scheme in metric space – keeping track of the data belonging to any of the data partitions of the index hierarchy and being able to perform search operations. More specifically, we focused on evaluating performance both in terms of efficiency and effectiveness on a *30-nearest neighbors* (*30-NN*) query. In principle, `k=30` for the k-NN algorithm is an arbitrary number and *LMI* can be trained on any `k`. Practically speaking though, 30 is sufficiently large to simulate real-world scenarios, but not enormously large to be a bottleneck in terms of time taken for the experiments.

Architecturally, *LMI* is a tree-like structure of models that continually divides the vector space, where the objects reside, into subspaces of decreasing size in each level. The core idea here is that an increasingly narrower focus of a model in terms of the data will produce better classification and therefore better search results. A similar concept is seen in structures like M-tree and M-index. The *Root node* is trained on the whole dataset and produces predictions assigning objects into categories. These categories correspond 1:1 with models in *Level 1* because no classification algorithm produces labels outside of the range of its target variable. Each of these models is trained on a subset of

the dataset, its output is predictions of the categories. These predictions split the subset further into smaller portions with each descend down the tree. This iterative operation continues until the last level is reached, where models are not present since no predictive action is needed anymore. We refer to these numbered groups of objects as *"buckets"* or *"leaf nodes"*. Buckets are containers for *objects*, the smallest unit in the hierarchy. Finally, these need to be linearly searched to find the specific object we are looking for, although this last step is skipped in our implementation and experiments for simplicity purposes. This whole process is formally described in Algorithm 1.

3.3 Training phase: Building LMI

3.3.1 Naive learned metric index

The training procedure (building of the LMI) was heavily inspired by the original Learned Index – **given a tree index structure, replace each internal node with a learned model**. Each model is trained on the sub-section of data that the given node represents (i.e., the model at the top is trained on all of the data, while its descendants are trained on smaller and smaller portions of the data as we descend down the structure). More specifically, a model is presented with a multi-class classification problem, where each of the classes represents a child node. Individual predictions then denote paths between nodes, thereby creating a tree.

Below is the naive algorithm for constructing an LMI, notice that it is fairly simple. There are two outer *"for" loops* because we have to iterate over "rows" and "columns" of the tree, where each new model is trained on a subset of the dataset assigned to a specific node only. This assignment takes place in the inner-most loop, which goes over datapoints in the current dataset, evaluates the current model's predictions for them, and assigns them to specific nodes one level below.

This training approach works reasonably well. This main feature is, that it is prone to creating many reorderings of the data and new buckets. This may slightly bloat the index, slowing down the searching procedure.

To try to overcome this obstacle, we modified the training procedure so that the models still drive the re-orderings of the data, thus

Algorithm 1: Naive Learned Metric Index training

```
Input: pre-processed dataset, index_struct
Output: trained tree of models
model_stack[][];
tmp_dataset[][];
tmp_dataset[1][1] = dataset.data;
M = index_struct.size;
for i <- 1 to M do
    for j <- 1 to max(index_struct[i]) do
        model_stack[i][j] = new model trained on
         tmp_dataset[i][j];
        if i < M then
            for dp ∈ tmp_dataset[i][j] do
                p = model_stack[i][j].predict(dp);
                 tmp_dataset[i+1][p].add(dp);
            end
        end
    end
end
```

respecting their natural distribution, but no new buckets are created, that there would not have been in the original index.

3.3.2 Adjusted learned metric index

I highlighted new parts of the pseudo-code in Algorithm 2 as compared to Algorithm 1. The only difference is that instead of including all the data-points, no matter what their assigned model is based on prediction, in training on all the levels, we allow only the data-points that were correctly classified to continue with training in the level below.

Algorithm 2: Adjusted Learned Metric Index training

```
Input: pre-processed dataset, index_struct
Output: trained tree of models
model_stack[][];
tmp_dataset[][];
tmp_dataset[1][1] = dataset.data;
tmp_dataset_labels[1][1] = dataset.labels;
M = index_struct.size;
for i <- 1 to M do
    for j <- 1 to max(index_struct[i]) do
        model_stack[i][j] = new model trained on
         tmp_dataset[i][j];
        if i < M then
            counter = 0;
            for dp ∈ tmp_dataset[i][j] do
                p = model_stack[i][j].predict(dp);
                if tmp_dataset_labels[i][j][counter] == p then
                    tmp_dataset[i+1][p].add(dp);
                end
                counter = counter + 1;
            end
        end
    end
end
```

3.4 Searching phase: Approximate k-NN search

Once the whole set of models in *LMI* is trained, we can use it to perform search operations (also known as querying). Querying is achieved using the *approximate similarity search*. This searching algorithm is an approximation of an exact similarity search (hence the name), which is much slower to be practical in most applications [5]. Approximate search employs a so-called *Early termination strategy*, that stops the searching before exhausting the whole search space. To do this, we use a fixed set of *stop conditions* to decide when the early termination should occur. The basic premise of this algorithm is to navigate across

the tree structure using the probabilities of all the classes (i.e., models). The higher the probability of a model, the sooner it is visited. A data structure storing the index nodes and designating the order in which to approach them is called a *priority queue*. An example few steps of the algorithm run are displayed in Figure 3.4.

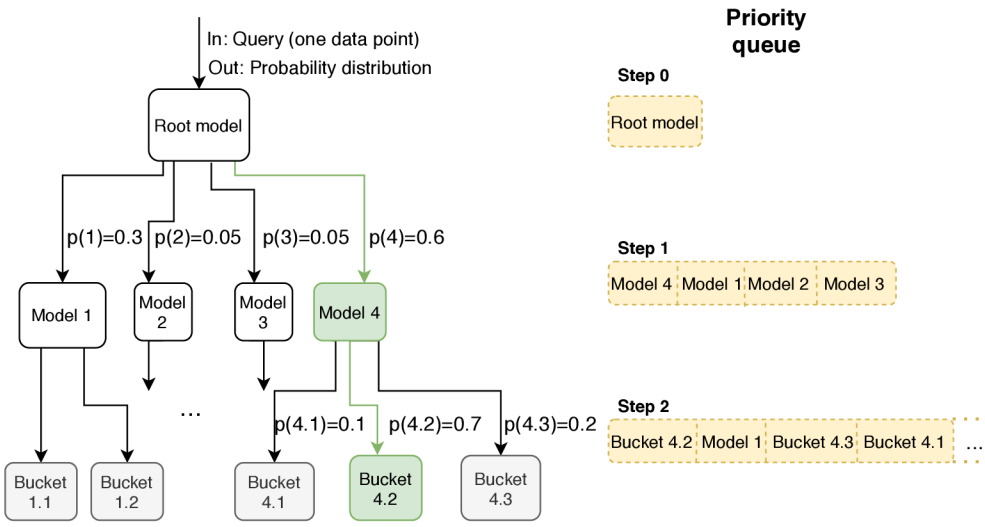

Figure 3.4: Example of an approximate search run that visited 2 nodes (1 model and 1 bucket). The first visited is `Model 4`, since `p(4)` is the highest of all probabilities. The next one is `Bucket 4.2` for the same reason. `Model 1` would follow right before `Bucket 4.3`. This process relies on a priority queue organizing these nodes in a decreasing order based on their probabilities and processing the top-most node at each step. The right side of the figure shows the state of the priority queue in the first 3 steps of the search.

As mentioned earlier, calculating the nearness of any 2 objects in the metric space is a straight-forward task. Both of the metric space datasets used have a defined (yet different, see Section 5.2) set of functions that are used for distance calculation. In order to evaluate the performance of the searching procedure, we have to choose a suitable metric. Information retrieval systems in general usually operate with *Recall* or *Precision*, alternatively with *F1 score* as their combination. *Recall* is defined as the fraction of relevant documents that are retrieved in proportion to all relevant documents in the collection. *Precision*

represents the relevant documents retrieved in proportion to all of the retrieved documents [14].

For this problem, we consider Recall to be an apt metric – we care about finding relevant objects and it is not deeply problematic if non-relevant objects are found along the way as well.

Recall is found by simply collecting the objects from the buckets found by *approximate search* and counting how many of them occur there, in proportion to the chosen `k`. An object in LMI "belongs" to a *Bucket X* if its parent nodes responded with the highest probability to it among all the other nodes.

This approach is in contrast to what was done in *The Case for Learned Index Structures*[1], where the performance evaluation was done on an object level, not a bucket level. However, it is in correspondence with the workings of *M-Index* and *M-tree* algorithms, so to allow for a fair comparison, we opted for this technique.

Naturally, if we search the whole LMI, we get 100% recall, so it is of interest to limit the portion of the index searched and look at the recall at various 'checkpoints'. These checkpoints are defined in the so-called *stop condition*. Since all the datasets we used, have a fixed size of 1 million objects, in all the experiments run we picked the following thresholds: 500, 1 000, 3 000, 5 000, 10 000, 50 000, 100 000, 200 000, 300 000, 500 000 (and optionally 750 000) objects found out of 1 000 000 objects in total. The relationship, which is shown therefore denotes the percentage of nearest neighbors found in relation to the overall percentage of objects searched.

4 Implementation of Learned Metric Index

In this Chapter, I give an overview of the main contribution of my work – the implementation of the *Learned Metric Index*. Here I present the codebase, show what the interaction with LMI looks like (building, searching operations) and how were the specific classifiers chosen for LMI. Subsequently, I will describe the process of hyperparameter tuning. The scope implementation of LMI, however, extends this Chapter, where some aspects were suited to be included in the subsequent ones – e.g., *Data preprocessing* in Section 5.4, *Experiments with alternative approaches*, and many ideas regarding Implementation are expressed in *Discussion* (Chapter 6) as well.

4.1 Implementation details

I implemented *Learned Metric Index* in Python 3.6, this programming language was chosen partly because of my familiarity with it, but mainly because of the rich ecosystem of machine learning libraries Python provides. For using neural networks, I opted for the Keras framework (built on top of TensorFlow) [15], for other machine learning algorithms, Scikit-learn [16] was chosen. Furthermore, mathematical computations were done with Numpy [17] and dataset manipulation with the Pandas library [18]. The prototyping and presentation of the results was done in Jupyter notebook [19] environment. During the development process, I used git for version control with the Institute of Computer Science's GitLab [1].

The architectural requirements were clear and from the beginning, I thought of the problem in an object-oriented fashion. Figure 4.1 shows the full code base, consisting of the classes, dependencies among them, and the script files. These include various supporting functionality, such as an interface to run and evaluate k-NN search, dataset loading functionality, or re-implementation of distance functions for the CoPhIR dataset.

The main object of interest is *LMI*. It contains all the attributes and methods needed for pre-processing the data, training an LMI of

1. `https://gitlab.ics.muni.cz/445526/learned-indexes`

arbitrary size, and subsequently searching in it with the desired stop condition. Since machine learning algorithms have different APIs for calling the train and predict method, all of them are implemented as separate objects with their own *train* and *predict* methods, inheriting from *BaseClassifier* that unifies some parts of code all of them have in common. *LMI* knows which machine learning algorithm to use based on the specification it receives when the `.train()` method is invoked.

Listing 4.1 shows an example of how a training method is called, where `lmi` is an instance of *LMI* class, `df` is the loaded dataset and we want to train a Random Forest classifier (RF) of 2 levels with 100 and 50 estimators, respectively.

Listing 4.1: Calling the training method

```
train_specs = {"RF": [\
                 {"n_est": 100, "depth": 30},\
                 {"n_est": 50, "depth": 15}\
               ]}
df_result = lmi.train(df, train_specs)
```

Once the LMI is trained, we can call the searching one, as shown in Listing 4.2:

Listing 4.2: Calling the searching method

```
li.search(df_result, query_id=549406,
   stop_cond_objects=[100, 10000], debug=True)
```

In my implementation, I tried to hide as much detail from the final API as possible while allowing for all the control over what classifiers should be used in what specific setting. At the same time, I keep the user informed of what is happening behind the scenes at higher granularity by default or lower one (by explicitly setting `debug=True`)[2].

2. The full demo example is available as a Jupyter notebook at `https://gitlab.ics.muni.cz/445526/learned-indexes/-/blob/master/Example.ipynb`

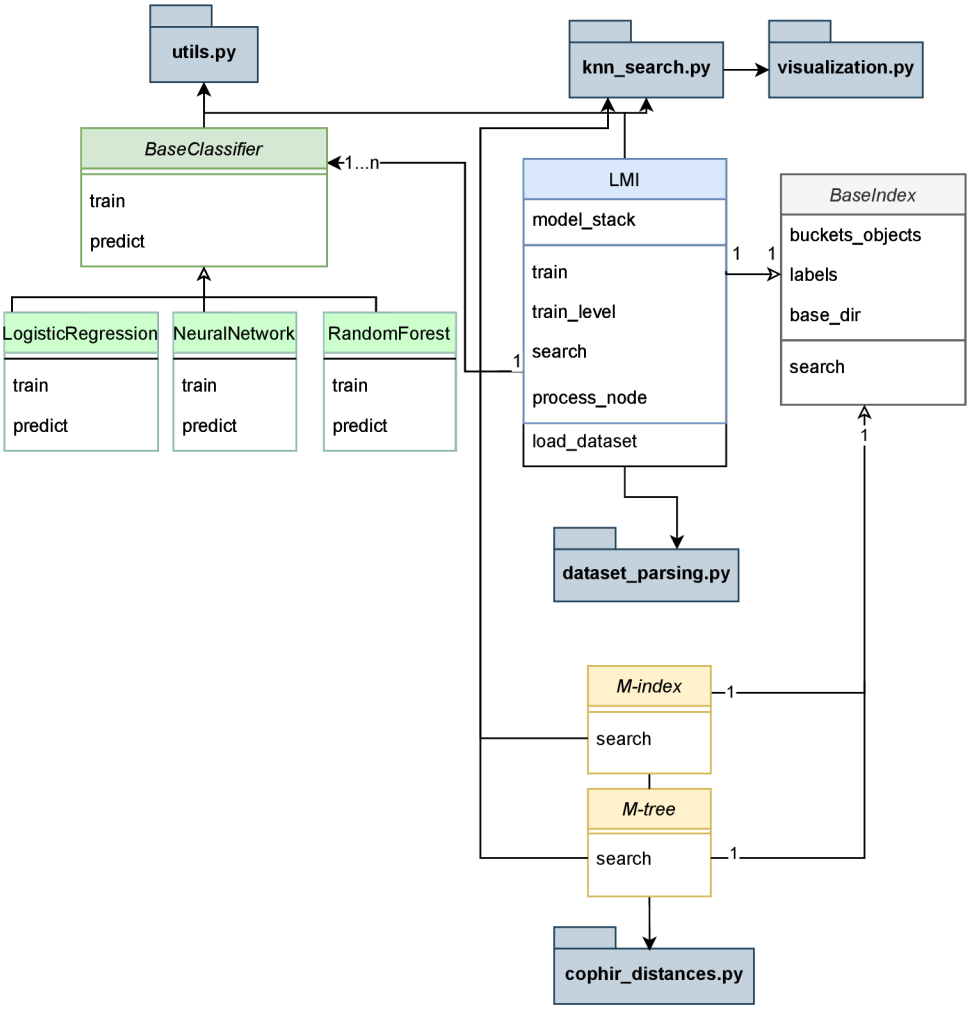

Figure 4.1: Diagram of the classes and source files.

4.2 Choosing a classifier

Choosing the right estimator for a problem is an important step in every machine learning pipeline. Kraska et al. chose to use *Linear regression* and Neural networks, with main focus on the latter, where an argument was made for their great future potential using modern hardware [1]. I decided to follow this example by considering Neural Networks and a counterpart of linear regression – logistic regression in the categorization domain. However, to explore the performance of classifiers more broadly, I set up an experiment involving building LMI with a wider variety of classifiers: Logistic regression, Random forests, k-nearest neighbors classifier, Linear support vector classifier[3], Neural network (NN)[4] and Convolutional neural network (CNN)[5]

In this initial check of suitability, I set out to examine 3 properties: The speed of training (building the tree), the speed of query prediction, and the accuracy of the 30-NN search.

Since LMI is comprised of thousands of nodes (classifiers) in bigger datasets (1-million objects and more) and in searching for k-NNs we are rarely satisfied with tens of found buckets, the time it takes to predict a single query (data-point) in a single model (tree node) is of great interest. Similarly, the time it takes to build an LMI might be of concern, because traditional models are relatively well adapted for quick building and machine learning models are known for slower training times. Figure 4.2 shows pitfalls of some estimators in this domain – namely k-nearest neighbors classifier and Linear support vector

3. All the classifiers mentioned above were run in their default scikit-learn implementations. For more detail see:
`https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html`
`https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html`
`https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html`
`https://scikit-learn.org/stable/modules/generated/sklearn.svm.LinearSVC.html`
4. Consisting of 3 layers of 32-32-16 fully-connected neurons with relu activation function, trained for 10 epochs optimized with Adam with learning rate of 0.001
5. Consisting of 1 layer of convolution with 64 filters, filter size of 3, ReLU activation function, followed by max pooling layer with pool size of 2 and final fully-connected and output layer with 100-16 neurons, trained for 10 epochs optimized with Adam with learning rate of 0.001

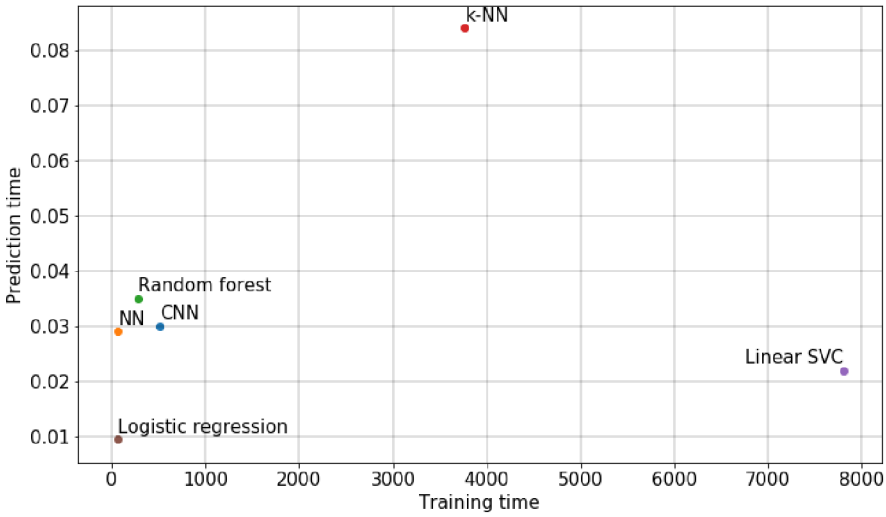

Figure 4.2: Training and prediction time of various classifiers. Training time refers to traning of the whole LMI – multiple classifiers, each for every tree node and prediction time is measured from the root of the tree to the first bucket. Time is stated in seconds.

classifier are slower to train, making them less suitable to continue with. On the other hand, the cheapest model in this measurement is Logistic Regression, followed by slight longer prediction times for Neural networks (NN, CNN) and Random forest.

Figure 4.3 shows the training and testing accuracy when trained on 80% of the dataset as opposed to 100%. Most of the classifiers keep training and testing accuracy similarly at around 50% with testing accuracy scoring slightly worse. One notable exception is Random forest, which can clearly overfit the training examples with no specific effort in terms of parameter tuning. However, overfitting on 1-NN labels does not guarantee quick retrieval of the remaining 29-NN. To evaluate the true objective function, a 30-NN search needs to be performed.

The outcome of the 30-NN search experiment is depicted in Figure 4.4. As we can see Logistic regression beats the other classifiers in terms of speed considering the first few stop conditions, k-NN

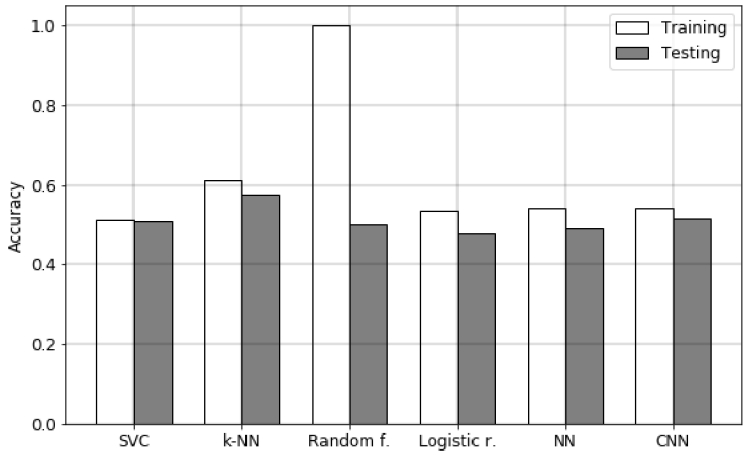

Figure 4.3: Training and testing accuracy of the root model of LMI of various classifiers.

classifier's rather slow prediction time marks it as the slowest in the 30-NN search as well. In terms of recall, there is a group of estimators that performs well – NN, CNN, and Random forest. The neural network used was a simple 2 layer fully-connected one, its convolutional counterpart was more robust. Results of this experiment suggest that using a network with more parameters may not be very beneficial, and a simpler model may suffice. Furthermore, there does not seem to be a benefit of using CNN to a fully-connected NN despite the data descriptors being extracted from images. Finally, Random forests look like a reasonable option in regards to the speed-recall trade-off.

As the results of this initial exploration, the selected machine learning models to test the viability of the LMI in the full experimentation setup were the following:

- logistic regression
- fully-connected neural network
- random forests

All three of these classifiers in their basic setups showed promising performance in terms of recall and time as shown in Figure 4.4. Investigating Logistic regression may be additionally of interest due to its

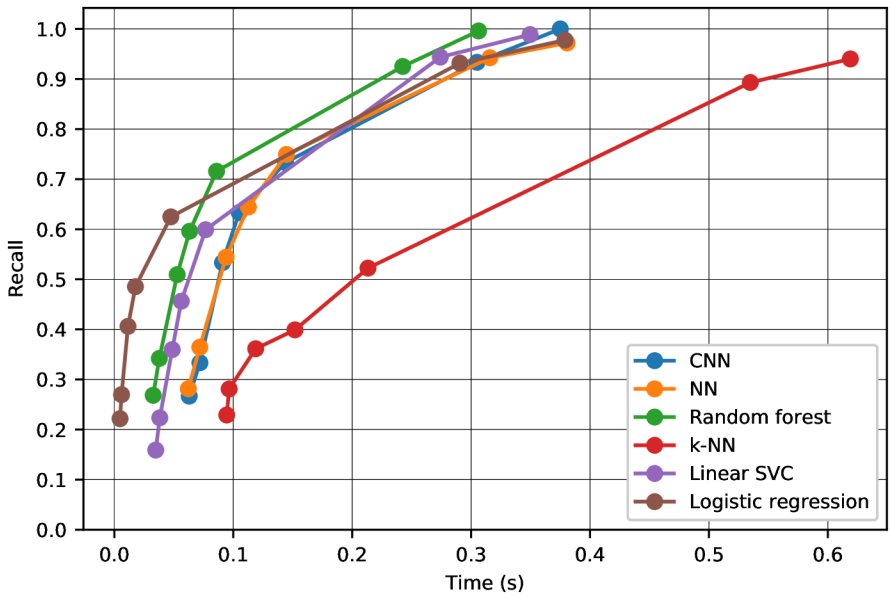

Figure 4.4: Comparison of 30-NN search performance of various classifiers on M-tree CoPhIR 100k dataset, mean of 100 randomly chosen queries. Points represent thresholds of increasing stop-conditions: 0.5k, 1k, 3k, 5k, 10k, 30k and 50k objects in the visited buckets

simplicity as a linear model [20], the same goes for Neural networks. Random forests represent a different type of algorithm, conceptually similar to how tree-like database structures, such as B-Trees, work by (simply put) dividing the feature space by using multiple if-else statements. This may or may not be an advantage over other algorithms, in any case, it is interesting to investigate.

A common feature of all three classifiers is that they train on 1-NN ground truth labels. This could be an issue because the overall goal is to answer 30-NN queries well, not merely find the target bucket of a query. At the same time, the 30-NN search recall was quite reasonable, reaching, on average, 55% of k-NNs found at 3% of the structure searched in some cases (see Figure 4.4). Optimizing for a better performance in k-NN search while in the training phase is difficult, though.

Overtraining does not seem like the desired strategy, on the other hand, undertraining can surely occur too and there is no way to figure out at which stage a model is while training – only when the whole LMI is constructed and can be searched.

One way to mitigate this uncertainty is to redefine the training setup and use all 30-NN labels as ground truth instead of a single one. Therefore, as a separate model, I chose to try a neural network with Multilabel classification [21], where the ground truth labels are a tuple of 30-NN categories instead of 1-NN. These categories are encoded by a technique similar to one-hot encoding, where frequencies of classes are considered — e.g., categories `[2,3,3]` are encoded as `[0,0,1,2]` to reflect "stronger" groups of nearest neighbors appropriately.

The final list of machine learning models to train the LMI with is therefore the following:

- logistic regression
- fully-connected neural network
- fully-connected neural network with Multilabel classification
- random forests

All of the algorithms were set up in a way that outputs the probability distribution of predicted classes. One exception is Random forests, which, as an ensemble technique, constructs its output by considering the proportion of the votes in the tree ensemble.

4.3 Space and time requirements: A case study

To give the reader an idea of how LMI behaves in the scope of individual classifiers and in terms of changing dataset size, I set up an experiment[6] to compare three properties of each model: *the training time*, *the prediction (inference) time* and *the space occupancy* of the said model. This analysis can be used as a primer for future optimizations, as described in Subsection 6.4.4.

6. The experiment was run on a machine with following hardware specification: 3x Intel(R) Xeon(R) Gold 5218 CPU @ 2.30GHz, GPU nVidia Tesla T4 16GB

Figure 4.5 shows the comparison of two different classifiers – Random forests and Neural network. Neural network exhibits higher levels of flexibility as the amount of data increases mainly in terms of training time and space occupancy, though the prediction time slightly worsens. On the other hand, Random forests have a much harder time adapting to increasing amounts of data in terms of both the training time and space occupancy. This fact stems from the nature of the algorithm – the more data there is, the bigger the trees, that make up Random forests, are. One other factor that hugely influences the size of the trees and, as a result, the training speed and space, is the *depth* hyperparameter, as seen in the last subplot of Figure 4.5. A similar configuration element in the realm of Neural networks is their complexity – the number of layers and neurons within them, though its effect on these 3 properties is not as significant as depth, in the case of Random forests.

Increasing efficiency in regards to time and space occupancy is crucial, mainly in systems where space is sparse and that require fast responses/training times. In the scope of this work, we did not specifically optimize for the best results on this front, but we realize this is a point that needs to be addressed to make LMI as fast and lightweight as possible. Also note, that the specific cost is tightly associated with the framework implementing the classifier and may differ from implementation to implementation.

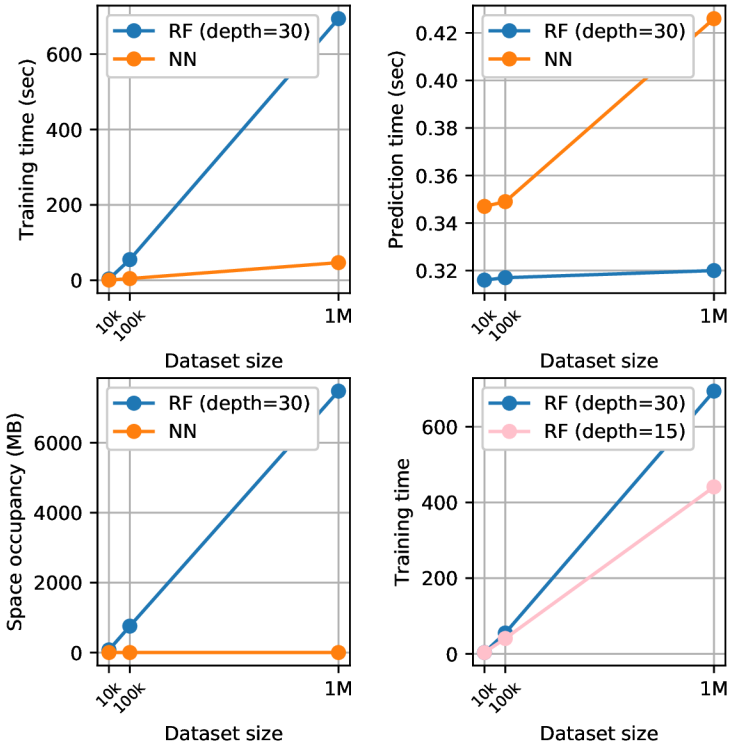

Figure 4.5: Comparison of a single Neural network model[a] and a Random Forest model[b] in terms of training time (top left subgraph), prediction time (top right subgraph), and space occupancy (bottom left subgraph). The difference within the same classifier with different depth hyperparameter values is shown on the bottom right subgraph.

a. Fully-connected Neural network with 3 layers of 282-128-18 neurons.
b. RF with maximum depth=30, number of estimators=200

4.4 Hyperparameter search

There are a number of best-practices in the machine learning community for how to organize a hyperparameter search process. These often involve using a subset of the data (to speed up the tuning) in combination with techniques such as grid search or random search while evaluating the suitability of a combination of hyperparameters on an objective function, like validation loss or accuracy.

Due to the nature of this problem, my hyperparameter searching options were limited. Simply put, the goal is to find as many k-NNs as soon as possible while maintaining a favorable distribution of objects in buckets. To the best of my knowledge, this objective function can be only evaluated at inference time, since LMI involves a combination of multiple learned models visited as a response to a specific query. Shrinking the size of the dataset for tuning purposes is problematic as well because the target k-NN objects would need to be recomputed for the smaller dataset to evaluate the performance of LMI. Another thought should be then given to finding the optimal value for k at the risk of this setup not being representative enough for the original dataset and target k-NNs.

At the same time, there undoubtedly exist optimal hyperparameters for each of the classifiers for this problem, so some form of hyperparameter searching is needed. Therefore instead of a robust hyperparameter search, I opted for simpler tuning, training the entire LMI with a small set of values for each major hyperparameter. The full list of hyperparameters is described below:

- logistic regression: number of epochs
- neural network: complexity – number of layers and neurons, number of epochs
- random forest: depth, number of estimators
- neural network (multilabel training): see seural network

Additionally, since various models within LMI work with subsets of the data that might differ in size and object distribution, I also considered a setup that combined different values of hyperparameters at different levels of LMI. This setup was run last.

Broadly speaking, there are two distinct groups of hyperparameters that contribute to different aspects of the training. The first group controls overtraining and consists of the number of epochs (Logistic regression, Neural network) or the depth of decision trees (Random forest) for models trained on 1-NN labels (exact hit). The training phase needs to be long enough so that the model learns the important features of the search space, but not so long that it becomes overfit for 1-NN search – this would lead to poor performance on 30-NN queries. This does not apply to Multilabel (30-NN) neural network – in this case, performance does not decline with overtraining, although after a certain number of epochs the performance does not improve anymore.

The second group of hyperparameters relates to model complexity and is represented by the number of layers/neurons (Neural network) and estimators (Random forest). Generally, the more complex the models are, the better they are at capturing diverse data, making them more suitable for larger subsections of the dataset. Simpler models, on the other hand, are more appropriate for smaller subsections. Complex models also tend to be slower at inference (as well as training), which has a negative impact on the overall performance of the index (evaluated as a trade-off between speed and recall).

The following Tables show the final hyperparameters chosen for each of the classifiers to run the final experiments with. The columns represent varying setups for every level of the LMI tree to take into account the decrease in input dataset size with a deeper position within the tree. The rest of the hyperparameters are the same as in the default implementations shown in Section 4.2.

Table 4.1: Hyperparameter *Epochs* chosen after the tuning for Logistic Regression and its values for classifiers at different LMI levels.

Hyperparameter	root	L1	L2	L3	L4	L5	L6	L7	L8
Epochs	10	10	5	5	5	5	2	2	2

Table 4.2: hyperparameters *Number of estimators* (*trees*) and *Depth* chosen after tuning for Random Forests and its values for classifiers in different LMI levels.

Hyperparameter	root	L1	L2	L3	L4	L5	L6	L7	L8
N. of estimators	300	300	300	200	200	200	100	100	100
Depth	25	25	25	25	20	20	20	20	15

Table 4.3: Hyperparameters *Complexity* (number of hidden layers and neurons within them), *Number of epochs*, *Learning rate*, and *Optimizer* chosen after tuning for Neural network.

Hyperparam.	root	L1	L2	L3	L4	L5	L6	L7	L8
Complexity	282-128	282-128	282-128	282-128	100	100	100	100	100
N. of epochs	20	10	10	10	5	5	5	5	5
Learning rate	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001
Optimizer	Adam	Adam	Adam	Adam	Adam	Adam	Adam	Adam	Adam

Table 4.4: Hyperparameters *Complexity* (number of hidden layers and neurons within them), *Number of epochs*, *Learning rate*, and *Optimizer* chosen after tuning for Multilabel neural network. Levels 5-8 are the same as level 4, they were omitted for brevity purposes.

Hyperparam.	root	L1	L2	L3	L4
Complexity	282-1024-256	282-1024-256	282-1024-256	282-1024-256	282-1024-256
N. of epochs	20	10	5	5	3
Learning rate	0.001	0.001	0.001	0.001	0.001
Optimizer	Adam	Adam	Adam	Adam	Adam

5 Experimental results

This chapter is dedicated to the results of experiments with LMI, together with M-tree and M-index.

5.1 Experiment setup

The experiment setup is the following: A 30-NN search for 1000 randomly chosen queries from 8 different dataset configurations. Each experiment is contains 4 LMI setups compared to either M-tree or M-index (depending on the configuration). In order to directly compare the speed and accuracy of our new indexing methods to the state of the art, I have also implemented the functions for creation and processing of the priority queue of the original indexes – M-tree and M-index (implementation with PPP-codes) – in Python.

Seven dataset configurations were run, each was a combination of the properties in Table 5.1 and each contains a million objects. One omitted combination was M-tree Profiset 200.

Table 5.1: Different properties of the base datasets.

Dataset type	Leaf node capacity	Index structure type
CoPhIR / Profiset	2000 / 200	M-tree / M-index

For clarification, *leaf node capacity* is analogous to the maximum number of objects allowed in the buckets. This property directly translates to how deep and wide the structure is – the smaller value of leaf node capacity, the deeper and wider the index is.

For completeness purposes, I include also Table 5.2 with the number of nodes and levels in each dataset.

5.1.1 Visualizing experiments

Significant time and thought were given to the way of presenting the experimentation results since there is a lot of information to be given to the reader. There are 3 dimensions to the experiments to consider: *Time*, *Recall*, and *Number of objects accessed*. Recall represents

Table 5.2: Number of buckets and levels in each of the datasets.

Dataset	Original index	Max. level	Leaf node cap.	Buckets
CoPhIR	M-tree	2	2 000	1 124
CoPhIR	M-tree	3	200	11 571
CoPhIR	M-index	8	200	62 049
CoPhIR	M-index	6	2 000	10 943
Profiset	M-tree	2	2 000	2 634
Profiset	M-index	2	2 000	4 288
Profiset	M-index	8	200	61 583

the percentage of the objects found within the 30 most similar. Time is self-explanatory: Time taken for an experiment to run. Note that all the experiments within one dataset configuration were run on the same hardware to ensure we can make time comparisons among them[1]. *Number of objects accessed* represents the cumulative sum of objects in the buckets visited. This number is important because in practice, after accessing a bucket, all objects in that bucket need to be sequentially scanned, which is a considerable computational overhead.

Each experiment is evaluated at certain checkpoints, that are set based on the number of objects visited since the number of buckets is variable from model to model. We chose to use this subset of checkpoints: *0, 500, 1 000, 3 000, 5 000, 10 000, 50 000, 100 000, 200 000, 300 000, 500 000* and optionally *750 000*.

1. The computational resources used belong to a grid computing provider Metacentrum (see Acknowledgements for details). Since it is a system where many users have access to the same machines, then even though allocated resources are assigned only to a user that requested them, it may have happened that computations were negatively impacted by some other user on the same machine through e.g., operations on the bus. To eliminate these nuances, I ran each of the experiments ten times or used exclusive access to a machine to ensure the time results are as faithful as possible.

5.2 Datasets

Throughout the experimentation with LMI, two different datasets were used – CoPhIR and Profiset. I worked with both of them as a subset of 1 million objects to have sufficiently large data distribution while being reasonably small to allow for fast training and experiments. I present them both in the following subsections.

5.2.1 CoPhIR dataset

The CoPhIR collection [22] is a well-known dataset in the similarity searching domain, designed for testing various Content-based Image Retrieval approaches. It consists of 100 million images extracted from Flickr, a website for sharing pictures. The dataset comes with a set of accompanying metadata. However, the only useful part of the dataset for the sake of content-based retrieval is its MPEG-7 descriptors. MPEG-7 is composed of five visual properties of each image:

1. scalable color
2. color structure
3. color layout
4. edge histogram
5. homogenous texture

and each of them is represented as a vector of specific values within the dataset. These properties are concatenated into 282-dimensional vectors that form the dataset attributes.

The ground truth for the distance between 2 objects is computed separately for each of the visual properties and then combined together. For some of the properties, the distance function is straightforward (e.g., *Euclidean distance*), some have a more complicated one. The specifics of all the distance computations are described in *CoPhIR Image Collection under the Microscope* [23].

5.2.2 Profiset dataset

Profiset image collection[2] [24] consists of 20 million images stock images. It is supplied together with DeCAF [25] descriptors of these images, which are 4096-dimensional vectors that were extracted from the layers of DeCAF Convolutional Neural Network.

The ground truth for the distance between 2 objects is in this case obtained from the *Euclidean distance* measurement.

5.2.3 Data preprocessing

Generally, when working with an arbitrary machine learning dataset, it is good practice not to rely on the suitability of the original features. The reason being that some algorithms have hidden assumptions about the data distribution, perform better when they are met and most datasets are not collected with this constraint in mind [26].

For the problem of training LMI I considered *Z-score normalization*, also known as *Standardization*. Z-score normalizes the data so that they have the properties of the standard normal distribution, so:

$$\mu = 0 \text{ and } \sigma = 1$$

Z-score helps especially when the data consists of a great range of values. The CoPhIR dataset was a clear candidate to normalize for this very reason. Profiset's attributes consist of a 4096-dimensional vectors that are an average of the last-but-one layer outputs from a Convolutional neural network [27]. Because of the unique type of attributes, that may be beneficial mainly for neural networks in LMI, I did not include feature scaling in this dataset at the beginning, but later checked for the difference with z-score normalization and found that it did not help the performance.

5.3 Experiments

In the following subsections, I will separately address the experiment results. The rest of the experiment runs will be included in Appendix A. All of the experiments followed the experiment design described in

2. `http://disa.fi.muni.cz/research-directions/software/profiset/`

Section 5.1 and all of them used Algorithm 1, described in Section 3.3, except for the CoPhIR M-tree 2000 setup[3], which used Algorithm 2. The hardware specifications of the machines that these experiments were run on are included in Appendix B.

5.3.1 CoPhIR

Four setups within the CoPhIR dataset were run altogether, in this Subsection I will comment on two of them – the basic **M-tree with leaf node capacity 2000** (2 tree levels) and the more complex **M-index with leaf node capacity 2000** (6 tree levels).

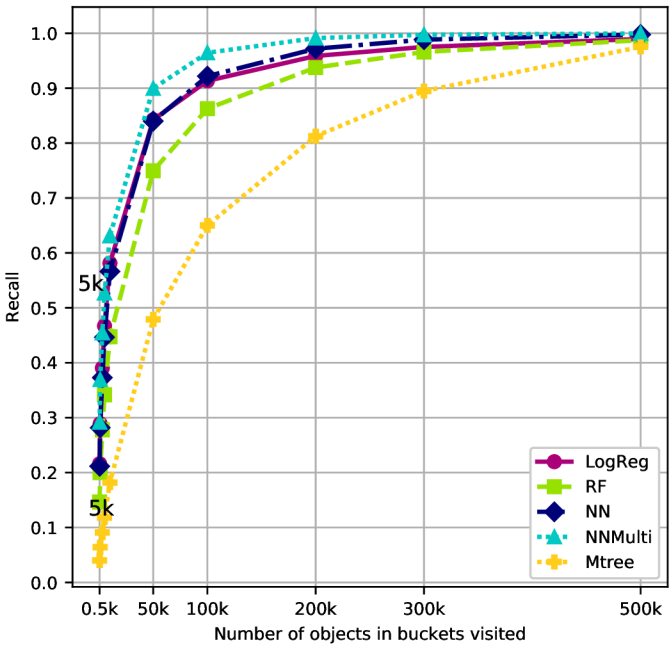

Figure 5.1: Comparison of 4 LMI models and M-tree using the CoPhIR M-tree dataset of leaf node capacity 2000 (2 tree levels). Trade-off between recall and the cumulative number of objects in the visited buckets.

3. Algorithm 2 was introduced quite late in the experimentation phase and showed improvements over Algorithm 1 only in some cases. We plan to re-investigate it for all the experiments in the scope of our paper.

In Figure 5.1, the better the model performs, the more to the left (not many objects seen) and up (higher recall) it is situated. The original M-tree index has initially problems finding many nearest neighbors, settling at 10% recall for 5k objects seen. In comparison, the best LMI model, Multilabel Neural network reaches more than 50% recall in the same threshold. Eventually, after 500k visited objects (50% of the whole dataset), every model converges to almost 100% recall. Random forest, Neural Network, and Logistic Regression are overshadowed by Multilabel, but still, perform considerably better than M-tree.

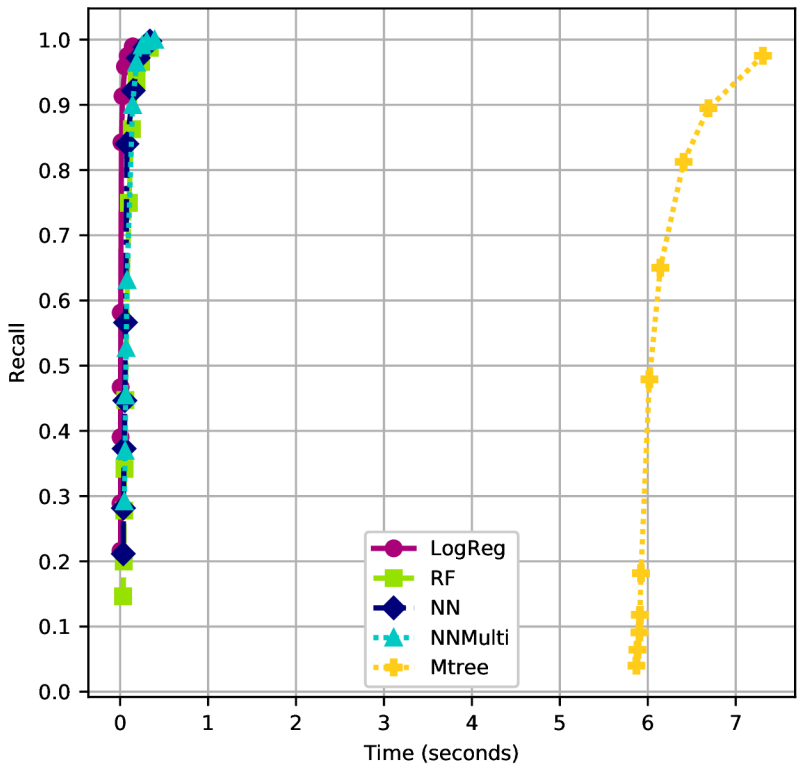

Figure 5.2: Comparison of 4 LMI models and M-tree using the CoPhIR M-tree dataset of leaf node capacity 2000 (2 tree levels). Trade-off between time and recall.

Figure 5.2 shows the same experiment setup, this time in terms of the trade-off between time and recall. Again, we would like to see the

points having as much recall for as little time possible (top left corner). Notice that M-tree starts off with a big delay of almost 6 seconds in comparison to the LMI models. This is mostly due to the fact that M-tree traverses the index structure breadth-first, as opposed to depth-first (which is typical for LMI and also M-index), so it performs a lot of calculations before even arriving at the first bucket.

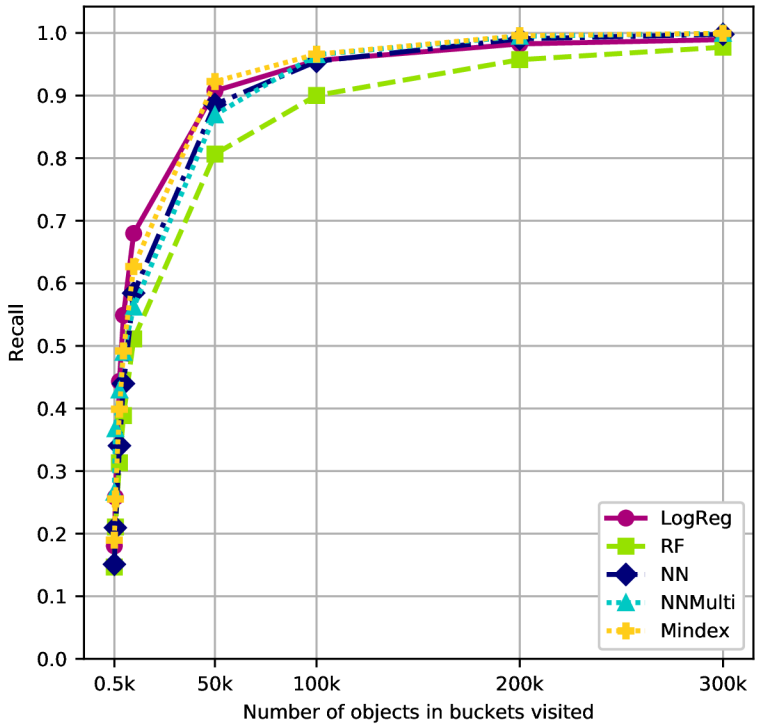

Figure 5.3: CoPhIR M-index dataset of leaf node capacity 2000 (6 tree levels): Comparison of 4 LMI models and M-index in terms of recall in proportion to number of objects in visited buckets.

As you can see in Figure 5.3, the performance of M-index and various LMI models is much more similar than that of an M-tree. M-index manages to find more k-NNs for a smaller amount of searched objects. However, all of the LMI models are solid competitors, with Logistic regression and/or Multilabel neural net surpassing M-index at the majority of the thresholds. M-index' performance is slightly

better at 50k and 100k stop conditions and all of the models reach comparable performance at the higher thresholds.

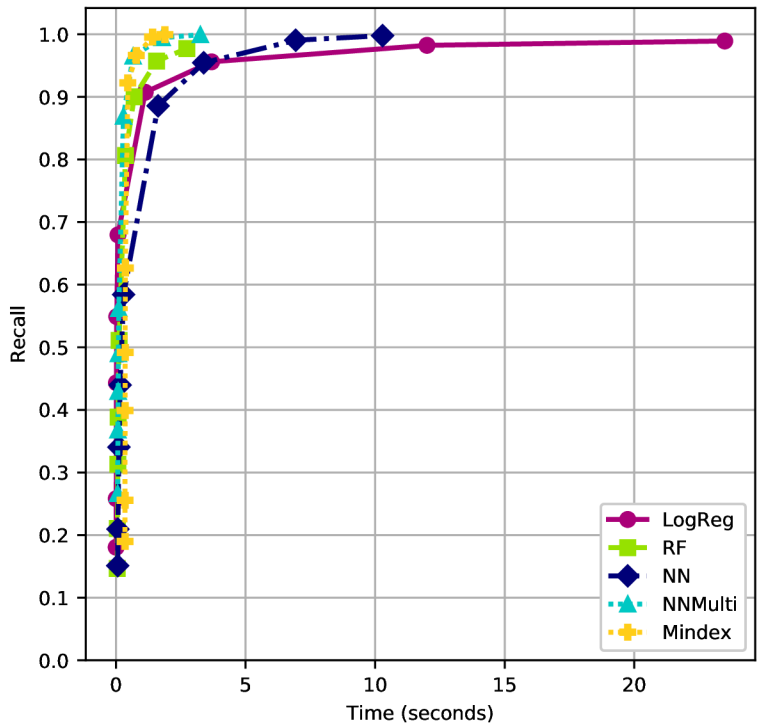

Figure 5.4: CoPhIR M-index dataset of leaf node capacity 2000 (6 tree levels): Comparison of 4 LMI models and M-index in terms of recall in proportion to time.

Figure 5.4 depicts the performance of M-index and LMI models in relation to the time taken to evaluate various thresholds. All of the models respond in terms of milliseconds in lower thresholds, the response time increases to lower tens of seconds for higher ones. In the case of Logistic regression, the running time is slightly higher.

5.3.2 Profiset

Here I include one Profiset experiment, out of the total of four runs, for brevity purposes. The rest are included in Appendix A.

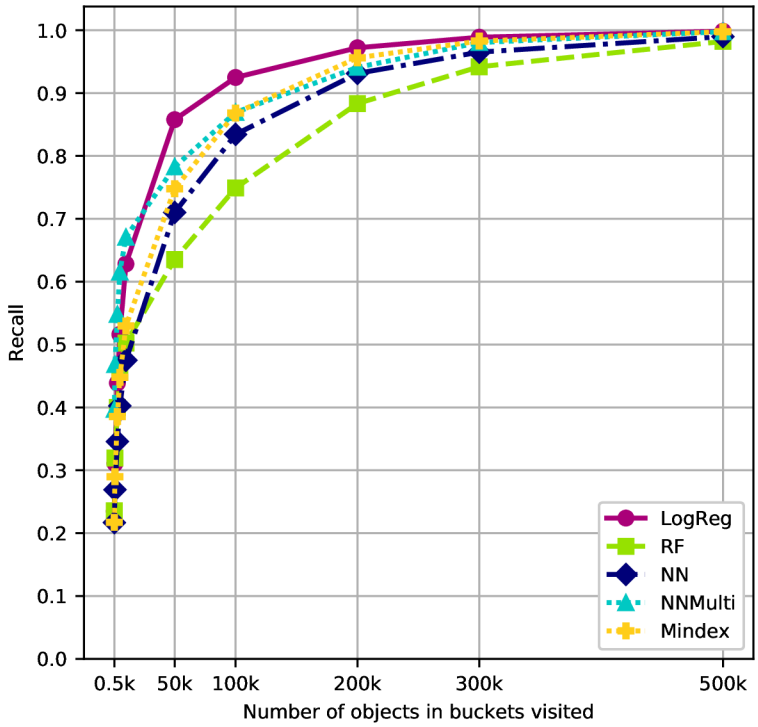

Figure 5.5: Profiset M-index dataset of leaf node capacity 2000 (2 tree levels): Comparison of 4 LMI models and M-index in terms of recall in proportion to number of objects in visited buckets.

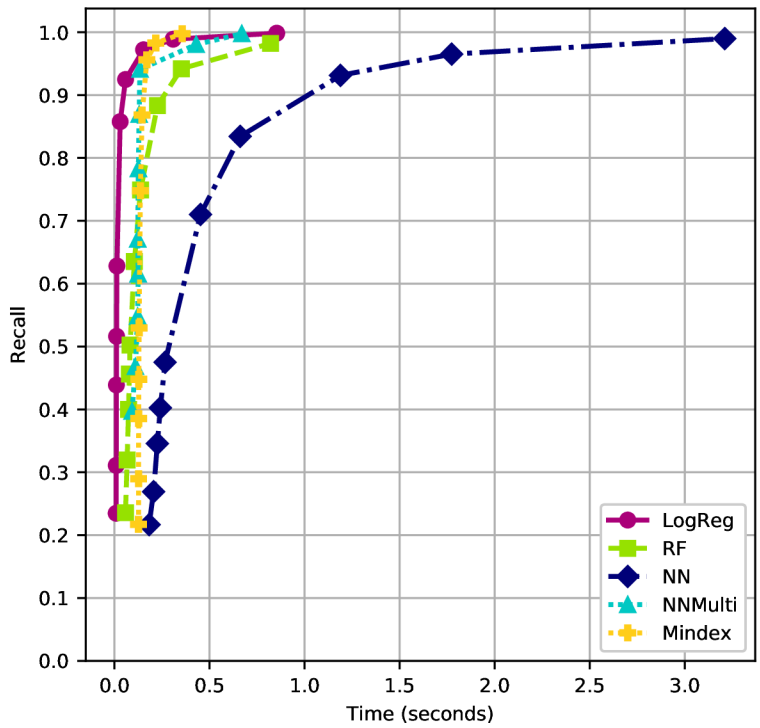

Figure 5.6: Profiset M-index dataset of leaf node capacity 2000 (2 tree levels): Comparison of 4 LMI models and M-index in terms of recall in proportion to the time elapsed.

5.4 Experiments with alternative approaches

This final section is dedicated to a brief exploration of other approaches in training and evaluating LMI. These could be used for inspiration in possible future research in this domain but were out of scope for this thesis. The first part documents how well LMI generalized on unseen data, which is the precursor for making the insert operation possible. The second part focuses on removing the need for using an existing index beforehand and poses the problem as an unsupervised one.

5.4.1 Queries out of dataset (OOD)

As mentioned in Section 3.1, the focus of our research and this thesis is searching performance, therefore we train on its entirety and the queries are equally selected from this trained set. A very natural question may arise: "What if we chose queries outside of the training portion?". In other words – if we do not give LMI any benefit of memorizing the query and its proximity to other objects, how much would the performance suffer?

To answer this question, I set up an experiment with CoPhIR M-index 2000 dataset configuration, where I compared the k-NN recall of trained models when evaluating 1000 queries from the dataset with 1000 queries that were out of the training dataset.

The performance of out-of-dataset experiments shown in Figure 5.7 is lower than the one evaluated on queries from the dataset, as expected. However, once we allow the search to continue to higher stop conditions, the difference between the two setups gets smaller. Models that were penalized the least seem to be both Neural networks, suggesting that they learned to capture the general properties of the data better without overtraining on the noise specific to the training data, rendering them promising candidates for optimization for this problem.

For comparison purposes, I included the corresponding M-index model, which showed performance loss that was fairly similar to the LMI models.

These results demonstrate that LMI models are indeed overtrained on the training data, which is expected and, after all, beneficial for our problem statement – optimizing for the best search performance

within a fixed dataset. However, the performance on testing (OOD) data is not significantly worse, even without actively optimizing for it e.g., Performance of OOD data of Neural network in Multilabel setup is still better on several thresholds than that of M-index with training data.

This suggests that there is room for improvements, e.g., in terms of efficiency of building LMI, where we could take just a subset of the dataset and consider it a reasonable enough representation of the whole dataset. The other potential application may be optimizing for insert performance.

5.4.2 Unsupervised learning

A natural objection may arise toward the supervised training approach introduced in Section 3.1. Why create an index, if it needs to be trained on another existing index? As will be explained in the following chapter, we chose to use an existing index in order to make a meaningful comparison with the original structures. However, we are aware that to be truly a serious competition to the state-of-the-art indexing schemes, we must explore ways to construct it independently.

To test how would unsupervised training perform in comparison to the supervised one, I set up an experiment with *Gaussian mixture models* (*GMM*) [4] algorithm, which is a probabilistic generalization of better-known K-Means clustering algorithm. During the training, I set the desired number of clusters to correspond to the training dataset configuration – CoPhIR Mtree 2000. Algorithm 1 (Naive one) was used for the building of LMI to allow as much freedom in bucket creation as possible.

Figure 5.8 shows a comparison of this unsupervised approach with one supervised LMI setup, the Neural networks model. As expected, the fine-tuned LMI model performs visibly better over the unsupervised approach. However, the GMM model appears to have learned how to cluster the data somewhat effectively without any further fine-tuning or data pre-processing. Interestingly enough, due to the breadth-first search nature of M-tree, the recall of the unsuper-

4. Scikit-learn implementation with 'diag' covariance type: `https://scikit-learn.org/stable/modules/generated/sklearn.mixture.GaussianMixture.html`

vised GMM is still higher on the first 5 stop conditions than that of an M-tree.

This is another interesting future research direction, free of any initial learning template, that could be a serious competitor of the existing indexing structures. I elaborate more on this approach in the following chapter.

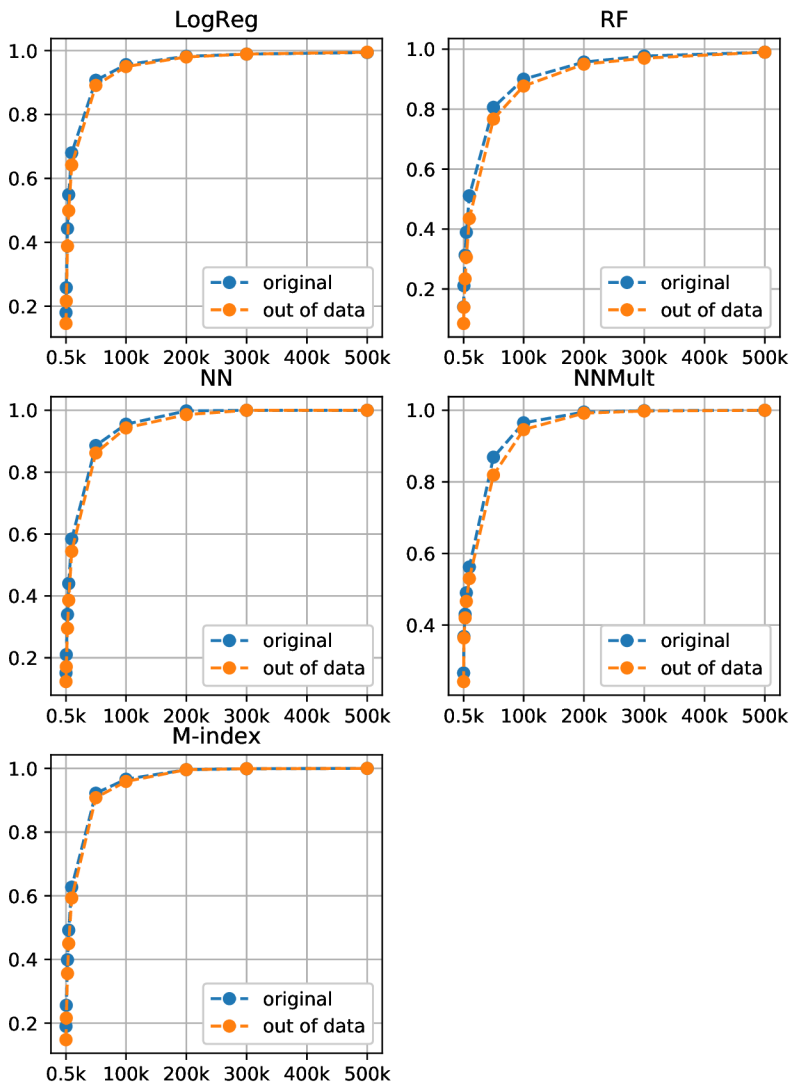

Figure 5.7: Comparison of average recall collected from running k-NN search on 1000 random queries from and out of the dataset (OOD), at increasing stop conditions.

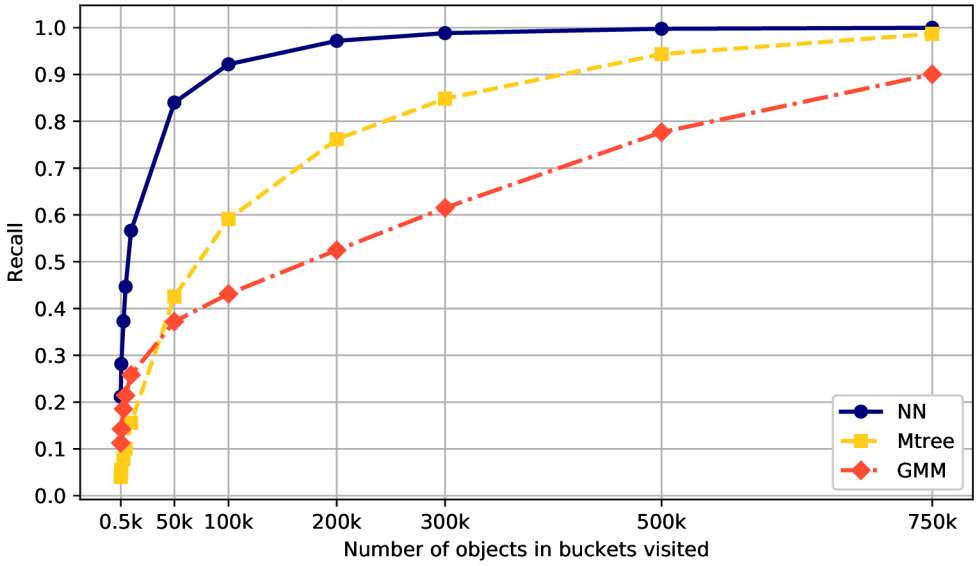

Figure 5.8: Performance comparision of a supervised LMI model – Neural networks (NN) and an unsupervised one – Gaussian mixture models (GMM)

6 Discussion

I dedicate this chapter to discuss the contributions of my work, offer an interpretation of the results, and reflect on the small successes and failures that happened while I was immersed in this problem. Summary and future work will be included in Conclusion (Chapter 7).

6.1 Hypothesis statement and LMI design

The initial goal that I set out to explore was whether machine learning-based models can indeed capture inherent features of complex data. This research idea was inspired by a paper with a similar setup in structured data, which showed promising results. The hypothesis was that if we can indeed learn something about the data and create an index based on this knowledge, that would give it an advantage over other indexing structures.

My task was to apply suitable learning models for this task, coming from a machine learning background. From the similarity searching side, I needed to understand what an indexing structure does, how can a static tree node be replaced by a machine learning model, and what it means to train a hierarchy of models that would cooperate at inference (prediction) time to give reasonable answers.

I approached the whole problem as any other standard categorization problem, i.e., given a dataset, train multiple models on different subsections of the data and 'connect' them by their predictions. I explored all the steps that should be present in any dataset-exploration model-building process, such as normalization, feature selection, class weighting (considering that this is a categorization problem with imbalanced classes), classifier selection, and finally evaluation using multiple metrics. I returned to this process later for hyperparameter tuning.

6.2 Experiments' setup

Regarding the setup of the experiments, we dealt with one fundamental question: *How do we compare the performance of LMI to the original indexes?*

The first idea was to evaluate the cost of both indexing schemes in terms of computational complexity. This proved itself to be a dead-end and we opted for a more straight-forward time comparison (complemented with performance comparison).

As easy for interpretation as a time comparison is, there are a few technical difficulties that I needed to take care of. Following are the variables I controlled for, to obtain the most reliable results possible:

- Searching in both indexing structures is implemented in the same language and environment
- Both searching procedures are run on the same hardware and multiple times to rule out interference from other users of the computational grid

My other task was, therefore, to re-write a Java implementation of M-tree and M-index searching in Python 3.6, with M-tree being the more straightforward, though its distance computation is rather complicated. The implementation of M-index searching was a bit more tricky since the algorithms included in the papers describing it are lacking a key optimization that was available only in the Java implementation[1].

6.3 Optimization of achieved results

The early results with basic models seemed promising, but I became slightly dissatisfied with the style of training and the style of evaluating – we could not overtrain the models on their categories (labels), since we trained on their exact hit (1-NN) and evaluated on their respective 30-NN labels. In other words, we did not train exactly for the objective we were trying to achieve.

1. DISA Lab's M-index code repository: `https://bitbucket.org/disalab/m-index/src/devel/`

That is why **I introduced a new training scheme – Multilabel training**, that is based on training with labels, which we are actually trying to optimize for – vectors of the full 30-NN bucket labels rather than just the one bucket label of each object. The results showed its dominance over other methods in several experiments. However, other models did not lack behind significantly, suggesting that a model can learn a fair amount of information needed to do a reasonable search just from the one bucket label of the query.

In the spirit of optimization, **I also set up a simple hyperparameter optimization scheme**, where for each model and a major hyperparameter, I tried to find an optimal value. However, I believe that despite the best of our efforts, there is a potential for improvement in model tuning by doing an extensive search with all the possible hyperparameters. We chose a simpler setup due to the limitations mentioned in Section 4.4.

6.4 Future research directions

Since we have encountered a lot of novel ideas not necessarily closely connected to our goal, but worth exploring anyway, we also outlined other possible future research directions.

6.4.1 Unsupervised learning

I consider learning the index in an unsupervised manner our top future research priority. The reason is simple – as long as we continue to use a template index for training, we naturally can not consider LMI being a fully-developed competitor for it. In Subsection 5.4.2 I showed a quick non-optimized proof of concept of how the future research may start and I believe that with proper algorithms, training approaches, and fine-tuning we will be able to construct an unsupervised LMI with a much better performance.

My colleagues and I plan to start to work on this unsupervised LMI by the start of 2021 together with a student whose upcoming master's thesis will revolve around this very problem.

6.4.2 Queries out of the dataset

Another interesting direction, which was unfortunately out of scope for this thesis is performance on queries out of dataset (testing queries). As outlined in Subsection 5.4.1, optimizing for this testing dataset may potentially lead to shorter index building time, where it would be sufficient to train the LMI on a smaller portion of the whole dataset given that the testing data is from the same data distribution.

6.4.3 Strategies for forming a priority queue

Lastly, a series of optimizations took place, that introduced Algorithm 2 and we briefly explored other approaches to forming a priority queue while searching. These approaches did not help to improve the results overall (only in some specific scenarios in particular thresholds), so they are not included in this thesis, but will be explored in the future.

6.4.4 Optimizing building cost

The time it takes to build an index and its space occupancy are important factors that influence how usable the index is in real-world applications. In the scope of our work, we decided not to directly optimize for these properties as our research is more proof-of-concept focused. Nevertheless, the building cost of LMI will need to be improved in the future. Just to give the reader an idea of what is the current situation of LMI on this front like, I included the best case and the worst-case scenario for building costs both in terms of time and resources:

Table 6.1: Building cost – best and worst case scenarios

	Dataset	Buckets	ML classifier	CPUs	RAM (GBs)	Time
Best	CoPhIR	1k	Logistic Regression	1	20	0.5h
Worst	Profiset	62k	Random Forests	10	700	6h

The searching cost is negligible in terms of resources and from the perspective of time, it is visible in the experiment graphs, that is why I am not explicitly including it here.

The cost as shown in Table 6.1 is highly dependent on the number of features in the dataset. Profiset, which is more than 14-times larger in terms of the number of features than CoPhIR is more costly to train both in terms of RAM and in terms of time and CPUs. The high RAM usage is also partly caused by the fact that both the LMI and the dataset are loaded into the main memory, stemming from the implementation of the Python libraries used (pandas, scikit-learn).

6.4.5 Resource efficiency

The main goal of our research and of this thesis was to verify our initial hypothesis – that machine learning models can be used to form a reasonable indexing structure. Though we come inspired from *The case for learned index structures*, no paper of this sort was published in the field of similarity searching, as far as we know. Therefore, the biggest strength of this thesis, in my view, is the potential for future research it provides.

With this in mind, our research philosophy revolved around gathering evidence for this hypothesis in as broad of a scope (different datasets, indexing structures, LMI models, approached to training, etc.) as possible, not necessarily fine-tuning for the best performance (in terms of time/accuracy).

Many of the implementation steps would have been done differently were we trying to optimize for the fastest most resource-efficient solution possible and I expect to work on improving LMI in this realm in the future. Firstly Python is not a first language choice in applications, where time is a critical component, and even though the backend of the machine learning models is programmed in more efficient lower-level languages, the navigation through the structure and sorting of the priority queue is not particularly fast (usually in terms of seconds for tens of thousands of objects, depending on the structure size). This applies to the original M-tree and M-index, whose searching mechanisms were re-written in Python.

Second consideration is that the training is rather resource-costly, requiring tens to hundreds of GigaBytes of RAM both the data and the models (in the case of scikit-learn models) are stored in the main memory. I briefly explored other techniques of data manipulation and index building that would be less of a burden resource-wise

but abandoned them because of their technical difficulty. However, I consider this a major impediment in the potential future applications.

The final space for optimization lies in the speed of query answering, which is connected to inference of machine learning models. As documented by other people[2] working in the field of query optimization, many machine learning frameworks do not have predict methods optimized for predicting the category of one object, as opposed to a whole dataset. This leads to so the prediction times that are rather slow during searching and some pre-predicting bulk operation that would include many queries may significantly improve the time performance.

2. https://maxhalford.github.io/blog/speeding-up-sklearn-single-predictions/

7 Conclusion

This thesis documented my contribution to creating a brand new approach to the indexing of complex data – *Learned metric index* (*LMI*). In Chapters 1 and 2 I introduced the problem of indexing in general and specifically in the similarity searching domain and outlined how machine learning could be used to improve it. Chapter 3 contained the specific problem statement and descriptions of how to go about it. Chapter 4 documented how I developed an environment for building this indexing structure and functionality for searching within it. LMI is built via supervised learning based on an existing index – M-tree or M-index. The datasets used in the experiments were CoPhIR and Profiset. Experimental results with all eight configurations of these datasets were presented and commented in Chapter 5. The machine learning models forming this structure included simpler Logistic regression and Random Forests (from scikit-learn) and more complex Neural networks with dual mode of training (from Tensorflow/Keras). I reflected on the research process in Chapter 6.

Since using indexes based on machine learning in similarity searching is a novel idea, the main contribution of this work is, in my view, opening the door for subsequent research and the solid proof of concept provided, showing that this approach is indeed viable. As for future research directions, in this thesis I hinted at some of the options that could take this idea forward: *Indexes based on unsupervised training*, though they mean a departure from *The case for learned index structures*, but make much more sense in the similarity searching domain, *optimizing for even better k-NN recall* with complex learning models and robust hyperparameter searching, *exploring the potential for reliable insert operations*, whose precursor, the performance of queries out of dataset, was presented as well and *reducing the cost of building and querying* by more effective implementation.

Learned metric index (*LMI*) is a probabilistic tree-like index, that takes advantage of the data distribution in the building phase without relying on computing distance measurements between objects in the querying phase, unlike other indexes in this field. The preliminary experimental results showed a clear dominance of this indexing approach over M-tree in terms of the number of nearest neighbors found

with a small percentage of the structure visited. Equally, LMI showed a great ability to compete with the newer M-index.

My work on this problem does not end with this thesis' deadline, since further experiments are needed for an upcoming paper that I am co-authoring together with my colleagues. I also plan to advise on an upcoming master's thesis regarding the topic of *Unsupervised LMI* and I hope to be a part of the research team that will develop more of these ideas forward.

A Remaining experimental results

A.1 CoPhIR experiments

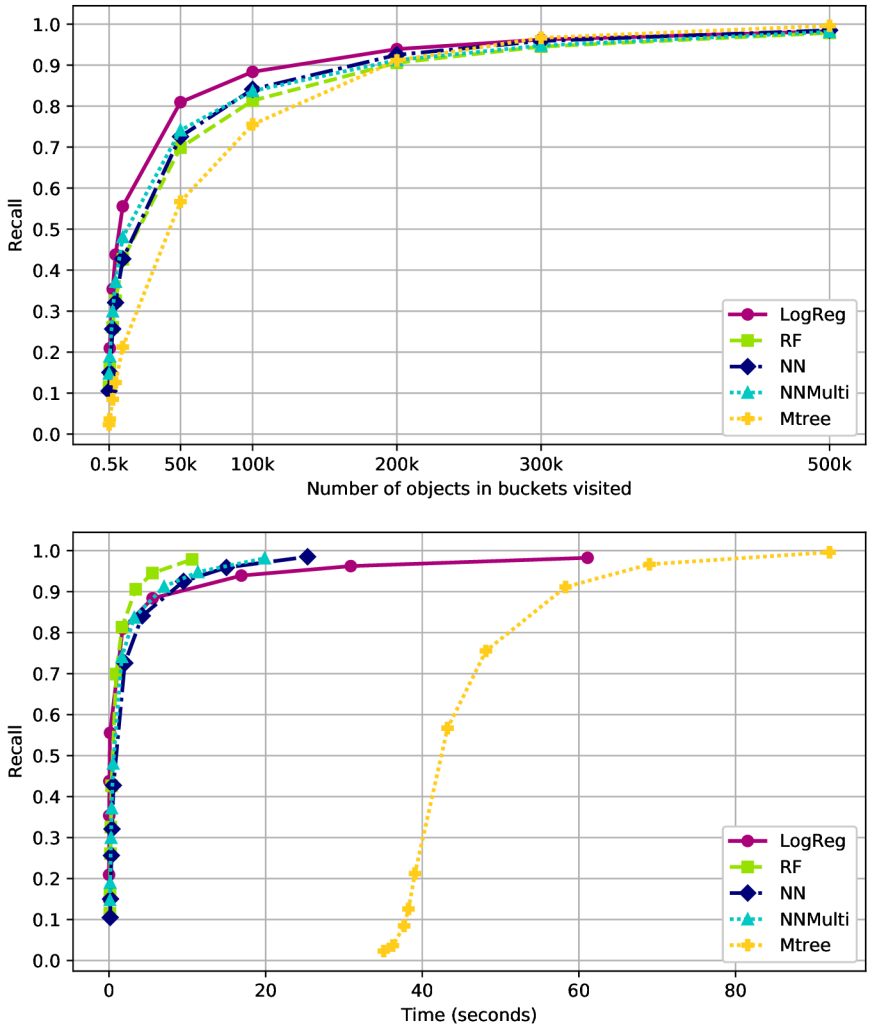

Figure A.1: CoPhIR M-tree 200 (3 index levels).

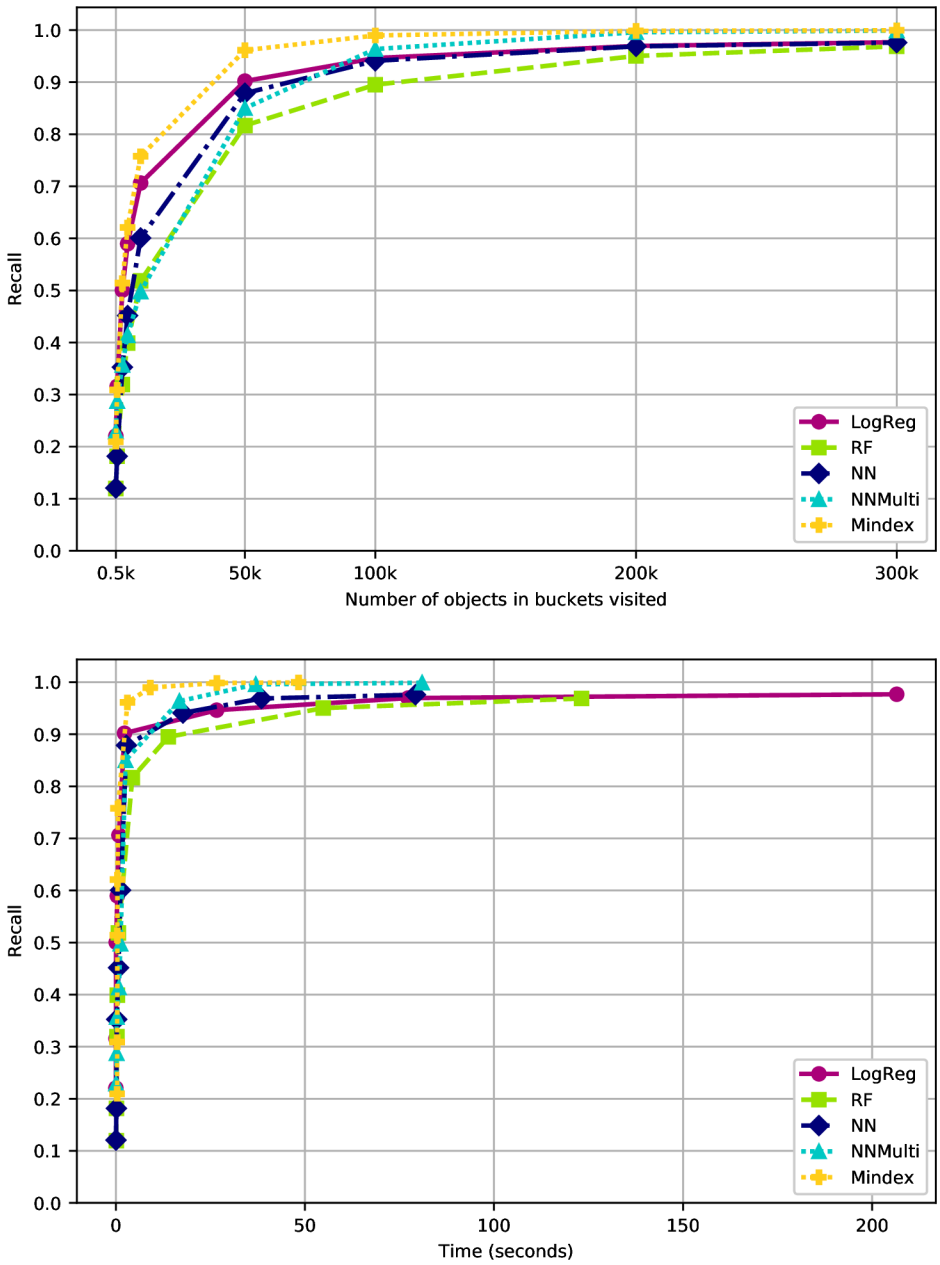

Figure A.2: CoPhIR M-index 200 (8 index levels).

A.2 Profiset experiments

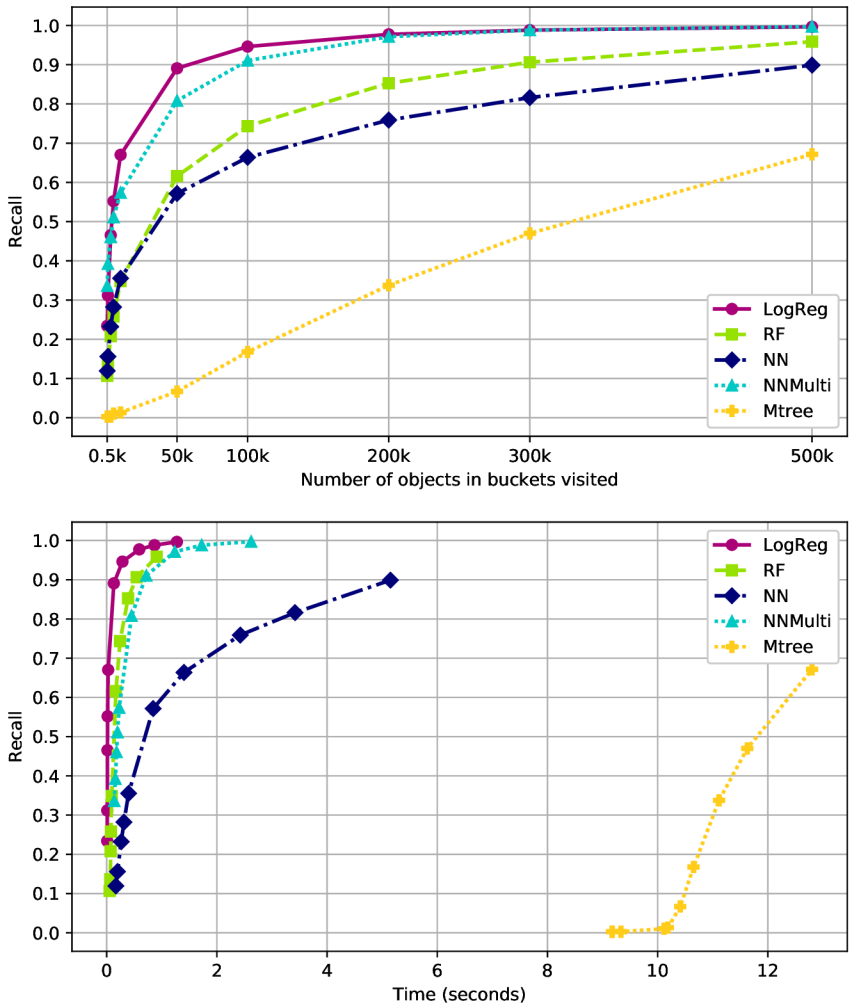

Figure A.3: Profiset M-tree 2000 (2 index levels).

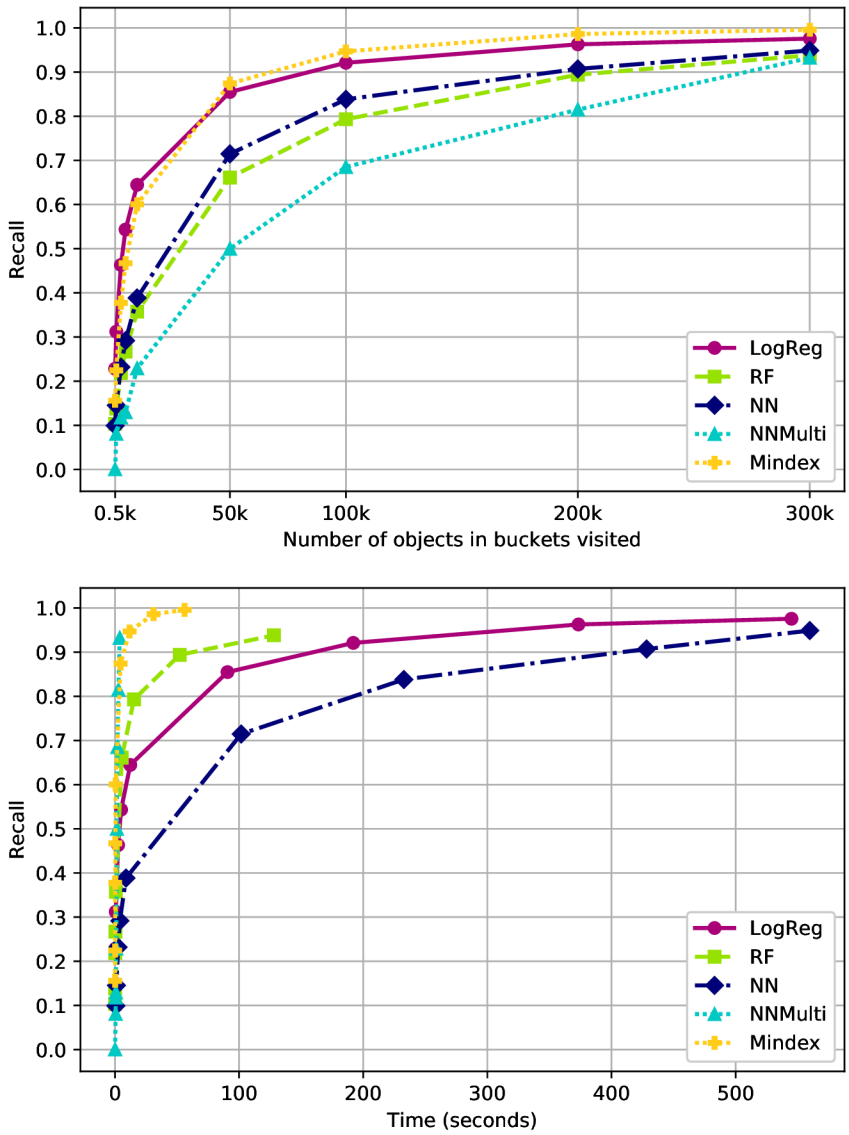

Figure A.4: Profiset M-index 200 (6 index levels).

B Hardware specification

Experiment setup	Cluster	CPU	CPUs used	GPU	GPUs used	Avg. RAM used
CoPhIR M-tree 2000	alfrid	Intel Xeon E5-2650v2	1	-	-	47.5 GB
CoPhIR M-tree 200	adan	Intel Xeon Gold 5218	2	Tesla T4	1	70 GB
CoPhIR M-index 200	adan	Intel Xeon Gold 5218	4	Tesla T4	1	97.5 GB
CoPhIR M-index 2000	adan	Intel Xeon Gold 5218	3	Tesla T4	1	50 GB
Profiset M-tree 2000	draba	Intel Xeon Gold 6230	5	-	-	350 GB
Profiset M-index 2000	draba	Intel Xeon Gold 6230	5	-	-	350 GB
Profiset M-index 200	draba	Intel Xeon Gold 6230	5	-	-	350 GB

C Acknowledgements

Computational resources were supplied by the project "e-Infrastruktura CZ" (e-INFRA LM2018140) provided within the program Projects of Large Research, Development and Innovations Infrastructures.

Bibliography

1. KRASKA, Tim; BEUTEL, Alex; CHI, Ed H; DEAN, Jeffrey; POLYZOTIS, Neoklis. The case for learned index structures. In: *Proceedings of the 2018 International Conference on Management of Data*. 2018, pp. 489–504.
2. GARCIA-MOLINA, Hector. *Database systems: the complete book*. Pearson Education India, 2008.
3. TARIQ, Ovais. *Understanding B+tree Indexes and how they Impact Performance*. 2018. Available also from: `http://www.ovaistariq.net/733/understanding-btree-indexes-and-how-they-impact-performance/#.XvriCJNKjRZ`.
4. KRASKAN, Prof. Tim. *Learned Index Structure - Prof. Tim Kraska*. 2018. Available also from: `https://youtu.be/qFbw4MgR6pI?t=267`.
5. ZEZULA, Pavel; AMATO, Giuseppe; DOHNAL, Vlastislav; BATKO, Michal. *Similarity search: the metric space approach*. Springer Science & Business Media, 2006.
6. PAOLOCIACCIA, MP. M-tree: An efficient access method for similarity search in metric spaces. In: *Proceedings of the 23rd VLDB Conference, Athenes, Greece*. 1997, pp. 357–368.
7. NAVARRO, Gonzalo; REYES, Nora. Dynamic Spatial Approximation Trees. *J. Exp. Algorithmics*. 2008, vol. 12. ISSN 1084-6654. Available from DOI: `10.1145/1227161.1322337`.
8. HOULE, Michael E.; NETT, Michael. Rank Cover Trees for Nearest Neighbor Search. In: BRISABOA, Nieves; PEDREIRA, Oscar; ZEZULA, Pavel (eds.). *Similarity Search and Applications*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 16–29. ISBN 978-3-642-41062-8.
9. NOVAK, David; BATKO, Michal; ZEZULA, Pavel. Metric index: an efficient and scalable solution for precise and approximate similarity search. *Information Systems*. 2011, vol. 36. ISSN 0306-4379. Available from DOI: `http://dx.doi.org/10.1016/j.is.2010.10.002`.

10. NOVAK, David; ZEZULA, Pavel. Rank Aggregation of Candidate Sets for Efficient Similarity Search. In: *Database and Expert Systems Applications: 25th International Conference, DEXA 2014, Munich, Germany, September 1-4, 2014. Proceedings, Part II*. Springer International Publishing, 2014, pp. 42–58. ISBN 978-3-319-10085-2. Available from DOI: `10.1007/978-3-319-10085-2_4`.

11. MIC, Vladimir; NOVAK, David; ZEZULA, Pavel. Binary Sketches for Secondary Filtering. *ACM Trans. Inf. Syst.* 2018, vol. 37, no. 1. ISSN 1046-8188. Available from DOI: `10.1145/3231936`.

12. ANTOL, Matej; DOHNAL, Vlastislav. Optimizing query performance with inverted cache in metric spaces. In: *East European Conference on Advances in Databases and Information Systems*. 2016, pp. 60–73.

13. COMER, Douglas. Ubiquitous B-tree. *ACM Computing Surveys (CSUR)*. 1979, vol. 11, no. 2, pp. 121–137.

14. MANNING, Christopher D; SCHÜTZE, Hinrich; RAGHAVAN, Prabhakar. *Introduction to information retrieval*. Cambridge university press, 2008.

15. CHOLLET, Francois et al. *Keras*. GitHub, 2015. Available also from: `https://github.com/fchollet/keras`.

16. PEDREGOSA, Fabian et al. Scikit-learn: Machine learning in Python. *Journal of machine learning research*. 2011, vol. 12, no. Oct, pp. 2825–2830.

17. OLIPHANT, Travis E. *A guide to NumPy*. Trelgol Publishing USA, 2006.

18. MCKINNEY, Wes et al. Data structures for statistical computing in python. In: *Proceedings of the 9th Python in Science Conference*. 2010, vol. 445, pp. 51–56.

19. KLUYVER, Thomas et al. Jupyter Notebooks – a publishing format for reproducible computational workflows. In: LOIZIDES, F.; SCHMIDT, B. (eds.). *Positioning and Power in Academic Publishing: Players, Agents and Agendas*. 2016, pp. 87–90.

20. BENGIO, Yoshua; GOODFELLOW, Ian; COURVILLE, Aaron. *Deep learning*. MIT press Massachusetts, USA: 2017.

21. TSOUMAKAS, Grigorios; KATAKIS, Ioannis; VLAHAVAS, Ioannis. Effective and efficient multilabel classification in domains with large number of labels. In: *Proc. ECML/PKDD 2008 Workshop on Mining Multidimensional Data (MMD'08)*. 2008, vol. 21, pp. 53–59.

22. BOLETTIERI, Paolo; ESULI, Andrea; FALCHI, Fabrizio; LUCCHESE, Claudio; PEREGO, Raffaele; PICCIOLI, Tommaso; RABITTI, Fausto. CoPhIR: a test collection for content-based image retrieval. *arXiv preprint arXiv:0905.4627*. 2009.

23. BATKO, Michal; KOHOUTKOVA, Petra; NOVAK, David. Cophir image collection under the microscope. In: *2009 Second International Workshop on Similarity Search and Applications*. 2009, pp. 47–54.

24. BUDIKOVA, Petra; BATKO, Michal; ZEZULA, Pavel. Evaluation platform for content-based image retrieval systems. In: *International Conference on Theory and Practice of Digital Libraries*. 2011, pp. 130–142.

25. DONAHUE, Jeff; JIA, Yangqing; VINYALS, Oriol; HOFFMAN, Judy; ZHANG, Ning; TZENG, Eric; DARRELL, Trevor. Decaf: A deep convolutional activation feature for generic visual recognition. In: *International conference on machine learning*. 2014, pp. 647–655.

26. GARCIA, Salvador; LUENGO, Julián; HERRERA, Francisco. *Data preprocessing in data mining*. Springer, 2015.

27. NOVAK, David; BATKO, Michal; ZEZULA, Pavel. Large-scale Image Retrieval Using Neural Net Descriptors. In: *Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval*. Santiago, Chile: ACM, 2015, pp. 1039–1040. SIGIR '15. ISBN 978-1-4503-3621-5. Available from DOI: `10.1145/2766462.2767868`.