AÇAI: Ascent Similarity Caching with Approximate Indexes

Tareq Si Salem
Université Côte d'Azur, Inria
tareq.si-salem@inria.fr

Giovanni Neglia
Inria, Université Côte d'Azur
giovanni.neglia@inria.fr

Damiano Carra
University of Verona
damiano.carra@univr.it

***Abstract*—Similarity search is a key operation in multimedia retrieval systems and recommender systems, and it will play an important role also for future machine learning and augmented reality applications. When these systems need to serve large objects with tight delay constraints, edge servers close to the end-user can operate as *similarity caches* to speed up the retrieval. In this paper we present AÇAI, a new similarity caching policy which improves on the state of the art by using (i) an (approximate) index for the whole catalog to decide which objects to serve locally and which to retrieve from the remote server, and (ii) a mirror ascent algorithm to update the set of local objects with strong guarantees even when the request process does not exhibit any statistical regularity.**

I. Introduction

Mobile devices can enable a rich interaction with the environment people are in. Applications such as object recognition or, in general, augmented reality, require to process and retrieve in real time a set of information related to the content visualized by the camera. The logic behind such applications is very complex: although mobile devices' computational power and memory constantly increase, they may not be sufficient to run these sophisticated logics, especially considering the associated energy consumption. On the other hand, sending the data to the cloud to be processed introduces additional delays that may be undesirable or simply intolerable [1]. Edge Computing [2], [3] solves this dichotomy by providing distributed computational and memory resources close to the users. Mobile devices may pre-process locally the data and send the requests to the closest edge server, which runs the application logic and provides quickly the answers.

Augmented reality applications often require to identify similar objects: for example an image (or an opportune encoding of it) can be sent as a query, and the application logic finds similar objects to be returned to the user [3]–[6]. For instance, a recommendation system may suggest similar products to a user browsing shop windows in a mall, or similar artists to a user enjoying a street artwork. The search for similar objects is based on a k-nearest neighbor (kNN) search in an opportune metric space [7]. The flexibility of kNN search comes at the cost of (i) high computational complexity in case of high dimensional spaces, and (ii) large memory required to store the instances. The first issue has been solved in recent years with a set of techniques used to index the collection of objects that provide *approximate answers* to kNN searches, i.e., they trade accuracy for speed. Searches over large catalogs (billions of entries) in high dimensional spaces may be executed now in less than a millisecond [8]. Still, the issue about the memory required to store the objects remain, especially in a distributed edge computing scenario, as edge servers have limited memory resources compared to the cloud.

The selection of which objects a specific edge server should maintain remains an open issue. Requests coming from the users often exhibit spatial and temporal correlation—e.g., the same augmented reality application will recover different information in different areas, and this information can change over time as the environment changes and users' interests evolve. This observation suggests that we may use the request pattern to drive the object selection. In other words, the edge server can be seen as a *cache* that contains the set of objects required to reply efficiently to the local requests avoiding to forward them to the cloud.

In this paper, we study how to optimize the use of the edge server memory for similarity searches. To this aim, we consider the *costs* associated to the replies, which capture both the quality of the reply (that is how similar/dissimilar to the request the objects provided are), as well as system costs like the delay experienced, the load on the server or on the network. The aim of our study is to design an online algorithm to *minimize such costs*. We provide the following contributions:

- We formulate the problem of kNN optimal caching taking into account both dissimilarity costs and system costs.
- We propose a new similarity caching policy, AÇAI, that 1) relies on fast, approximate similarity search indexes to decide which objects to serve from the local datastore and which ones from the remote repository and 2) uses an online mirror ascent algorithm to update the cache content in order to minimize the total service cost. AÇAI offers strong theoretical guarantees without any assumption on the traffic arrival pattern.
- We compare our solution with state-of-the-art algorithms for similarity caching and show that AÇAI consistently improves over all of them under realistic traces.

The remainder of the paper is organized as follows: we present similarity caches in Sec. II and other relevant background in Sec. III. We introduce AÇAI in Sec. IV and our experimental results in Sec. V. An extended version is available online [9]. It contains additional results and more detailed proofs.

ISBN 978-3-903176-43-0

II. Similarity Caches

Consider a remote server that stores a catalog of objects $\mathcal{N} := \{1, 2, \ldots, N\}$. A similarity search request r aims at finding the k objects $o_1, o_2, \ldots, o_k \in \mathcal{N}$ that are most similar to r given an application-specific definition of *similarity*. To this purpose, similarity search systems rely on a function $c_d(r, o)$, that quantifies the dissimilarity of a request r and an object o. We call such function the *dissimilarity cost*.

In practice, objects and requests are mapped to vectors in $\mathbb{R}^d$ (called *embeddings*), so that the dissimilarity cost can be represented as (a function of) a selected distance between the corresponding embeddings. For instance, in case of images, the embeddings could be a set of descriptors like SIFT [10], or ORB [11], or the set of activation values at an intermediate layer of a neural network [12], [13]. Examples of commonly employed distances are the p-norm, Mahalanobis or cosine similarity distances.

The server replies to each request r with the k most similar objects in the catalog $\mathcal{N}$. As the dissimilarity is captured by the distance in the specific metric space, these objects are also the k closest objects (neighbors) in the catalog to the request r ($k\mathrm{NN}(r, \mathcal{N})$).[1] The mapping translates the similarity search problem in a kNN problem [14], [15].

We can also associate a dissimilarity cost to the reply provided by server (e.g., by summing the dissimilarity costs for all objects in $k\mathrm{NN}(r, \mathcal{N})$). This cost depends on the catalog $\mathcal{N}$ and we do not have control on it. In addition, there is a *fetching cost* to retrieve those objects. The fetching cost captures, for instance, the extra load experienced by the server or the network to provide the objects to the user, the delay experienced by the user or a mixture of those costs.

In the Edge Computing scenario we consider, we can reduce the fetching cost by storing at the edge server a subset of the catalog $\mathcal{N}$, i.e., the edge server works as a *cache*. When answering to a request, the cache could provide just some of the objects the server would provide. The seminal papers [16], [17] proposed a different use of the cache: the cache may reply to a request using a local subset of objects that are potentially *farther* than the true closest neighbors to reduce the fetching cost while increasing—hopefully only slightly—the dissimilarity cost. They named such cache a *similarity cache*. The original applications envisaged were content-based image retrieval [16] and contextual advertising [17]. But, as recognized in [18], the idea has been rediscovered a number of times under different names for different applications: semantic caches for object recognition [3]–[6], soft caches for recommender systems [19], [20], approximate caches for fast machine learning inference [21].

A common assumption in the existing literature is that the cache can only store h objects and the index needed to manage them has essentially negligible size. We also maintain this assumption that is justified in practice when objects have a size of a few tens of kilobytes (see the quantitative examples in Sec. III).

Caching policies. The performance of the cache depends heavily on which objects the cache stores. Among the papers mentioned above, many (e.g., [19], [20]) consider the offline object placement problem: a set of objects is selected on the basis of historical information about object popularity and prefetched in the cache. But object popularity can be difficult to estimate and can change fast, specially at the level of small geographical areas (as those that can be served by an edge server) [22]. Other papers [3]–[6], [21], [23] present more a high-level view of the different components of the specific application system, without specific contributions in terms of cache management policies (e.g., they apply minor changes to exact caching policies like LRU or LFU). Some recent papers [18], [24], [25] propose online caching policies that try to minimize the total cost of the system (the sum of the dissimilarity cost and the fetching cost), but their schemes apply only to the case $k = 1$, which is of limited practical interest.

To the best of our knowledge, the only dynamic caching policies conceived to manage the retrieval of $k > 1$ similar objects are SIM-LRU, CLS-LRU, and RND-LRU proposed in [17] and QCache proposed in [26]. Next, we describe in detail these policies to highlight AÇAI's differences and novelty.

All these policies maintain an ordered list of key-value pairs where the key is a previous request and the value is the set of k' closest objects to the request in the catalog (in general $k' \geq k$). The cache, whose size is h, maintains a set of h/k' past requests. This approach allows to decompose the potentially expensive search for close objects in the cache (see Sec. III) in two separate less expensive searches on smaller sets. Upon arrival of a request r, the cache identifies the l closest requests to r among the h/k' in the cache. Then, it merges their corresponding values and looks for the k closest objects to r in this set including at most $l \times k'$ objects. If this answer is evaluated to be good enough, then an *approximate hit* occurs and the answer is provided to the user, otherwise the request r is forwarded to the server that needs to provide all k closest objects. The cache state is updated following a LRU-like approach: upon an approximate hit, all key-value pairs that contributed to the answer are moved to the front of the list; upon a miss, the new key-value pair provided by the server is stored at the front of the list and the pair at the end of the list is evicted.

This operation is common to SIM-LRU, CLS-LRU, RND-LRU, and QCache. They differ in the choice of the parameters k' and l and in the way to decide between an approximate hit and a miss. As they assume no knowledge about the catalog at the server, they cannot compare the quality (i.e., the dissimilarity cost) of the answer the cache can provide with the quality of the answer the server can provide. They need then to rely on heuristics.

SIM-LRU considers $k' \geq k$ and $l = 1$. Upon a request for r, SIM-LRU selects the closest request in the cache

[1] More precisely, these are the k objects whose embeddings are closer to the embedding of r. From now on we identify objects and their embeddings.

and decides for an approximate hit (resp. a miss) if their dissimilarity is smaller (resp. larger) than a given threshold C_θ. Every stored key r' *covers* then a hypersphere in the request space with radius C_θ.

SIM-LRU has the property that no two keys in the cache have a dissimilarity cost lower than C_θ, but the corresponding hyperspheres may still intersect. CLS-LRU [17] is a variant of SIM-LRU, that can update the stored keys (the centers of the hyperspheres) and push away intersecting hyperspheres to cover the largest possible area of the request space. To this purpose, CLS-LRU maintains the history of requests served at each hypersphere and, upon an approximate hit, moves the center to the object that minimizes the distance to every object within the hypersphere's history. When two hyperspheres overlap, this mechanism drives their centers apart, which in turn reduces the overlapping region.

RND-LRU [17] is a random variant of SIM-LRU that determines the request r to be a miss with a probability that is increasing with the dissimilarity cost between r and the closest request in the cache.

Finally, QCACHE [26] considers $k' = k$ and $l > 1$. The policy decides if the k objects selected from the cache are an approximate hit if (1) at least two of them would have been provided also by the server—a sufficient condition can be obtained from geometric considerations—or (2) the distribution of distances of the k objects from the request looks similar to the distribution of objects around the corresponding request for other stored key-value pairs.

These policies share potential inefficiencies: (i) the sets of closest objects to previous queries are not necessarily disjoint (but CLS-LRU tries to reduce their overlap) and then the cache may store less than h distinct objects; (ii) the two-level search may miss some objects in the cache that are close to r, but are indexed by requests that are not among the l closest requests to r; (iii) the policy takes into account the dissimilarity costs at the caches but not at the server; (iv) objects are served in block, all from the cache or all from the server, without the flexibility of a per-object choice. As we are going to see, AÇAI design prevents such inefficiencies by exploiting new advances in efficient approximate kNN search algorithms, which allows us to abandon the key-value pair indexing and to estimate the dissimilarity costs at the server. Also AÇAI departs from the LRU-like cache updates, considering gradient update schemes inspired by online learning algorithms [27].

III. Other Relevant Background

Indexes for approximate kNN search. Indexes are used to efficiently search objects in a large catalog. In case of kNN, one of the approaches is to use tree-based data structures. Unfortunately, in high dimensional spaces, e.g., $\mathbb{R}^d$ with $d > 10$, the computational cost of such search is comparable to a full scan of the collection [28]. *Approximate Nearest Neighbor* search techniques trade accuracy for speed and provide k points close to the query, but not necessarily the closest, sometimes with a guaranteed bounded error. Prominent examples are the solutions based on locality sensitive hashing [29], product quantization [8], [30], and graphs [31]. Despite being approximate, these indexes are in practice very accurate, as showed over different benchmarks in [32].

As we are going to describe, AÇAI employs two approximate indexes (both stored at the edge server): one for the content stored in the cache, and one for the whole catalog $\mathcal{N}$ stored in the remote server. For the former, since cache content varies over time, we rely on a graph-based solution, such as HNSW [31], that supports dynamic (re-)indexing with no speed loss. On various benchmarks [32], HNSW results the fastest index, and it is able to answer a 100NN query over a dataset with 1 million objects in a 128-dimensional space in less than 0.5 ms with a recall greater than 97%.[2] As for the memory footprint, a typical configuration of the HNSW index requires $O(d)$ bytes per objects, where d is the number of dimensions. For instance, in case of $d = 128$ dimensional vectors, the memory required to index 10 million objects is approximately 5 GB. As the server catalog changes less frequently, AÇAI can index it using approaches with a more compact object representation like FAISS [8]. FAISS is slightly slower than HNSW and does not support fast re-indexing if the catalog changes, but it can manage a much larger set of objects. With a dataset of 1 billion objects, FAISS provides an answer in less than 0.7 ms per query, using a GPU.[3] As for the memory footprint, for a typical configuration (IVFPQ), FAISS is able to represent an object with 30 bytes (independently from d): only 3 GB for a dataset with 100 million objects!

Summing up our numerical example, if each object has size 20 KB, an edge server with AÇAI storing locally 10 million objects from a catalog with 100 million objects, needs 200 GB for the objects and only 8 GB for the two indexes. The larger the objects, the smaller the index footprint: for example, when the server has a few Terabytes of disk space to store large multimedia objects, the indexes' size can be ignored.

Gradient descent approaches. Online caching policies based on gradient methods have been studied in the stochastic request setting for exact caching, with provable performance guarantees, [33], [34]. More recently, the authors of [25] have proposed a gradient method to refine the allocation of objects stored by traditional similarity caching policies like SIM-LRU. Similarly, the reference [35] considers a heuristic based on the gradient descent/ascent algorithm to allocate objects in a network of similarity caches. In both papers, the system provides a single similar content ($k = 1$).

We deviate from these works by considering $k > 1$ and the more general family of online mirror ascent algorithms (of which the usual gradient ascent method is a particular instance). Also our policy provides strong performance guarantees under a general request process, where requests can even be selected by an adversary. Our analysis relies on results from online convex optimization [36] and is similar in spirit to

[2]Experiments on a 4-core Intel Core i7-4790, 32 GB RAM, 3.6 GHz.

[3]Experiments on 2x2.8GHz Intel Xeon E5-2680v2, 4 Maxwell Titan X GPUs, CUDA 8.0.

what done for exact caching using the classic gradient method in [27] and mirror descent in [37].

IV. AÇAI Design

A. Cost Assumptions

Many of the similarity caching policies proposed in the literature (including SIM-LRU, CLS-LRU, RND-LRU, and QCache) have not been designed with a clear quantitative objective, but with the qualitative goal of significantly reducing the fetching cost without increasing too much the dissimilarity cost. Because of such vagueness, the corresponding papers do not make clear assumptions about the dissimilarity costs and the fetching costs. On the contrary, AÇAI has been designed to minimize the total cost of the similarity search system and we make explicit the corresponding hypotheses.

Our main assumption is that all costs are additive. The function $c_d(r, o)$ introduced in Sec. II quantifies the dissimilarity of the object o and the request r. Let $\mathcal{A}$ be the set of objects in the answer to request r: it is a natural to consider as dissimilarity cost of the answer $\sum_{o\in\mathcal{A}} c_d(r, o)$.

In addition, if fetching a single object from the server incurs a cost c_f, the fetching cost to retrieve m objects is $m \times c_f$. This is an obvious choice when c_f captures server or network cost. When c_f captures the delay experienced by the user, then summing the costs is equivalent to consider the round trip time negligible in comparison to the transmission time, which is justified for large multimedia objects. It is easy to modify AÇAI to consider the alternative case when the fetching cost does not depend on how many objects are retrieved. Finally, as common in other works [18], [25], we assume that both the dissimilarity cost and the fetching cost can be directly compared (e.g., they can both be converted in dollars). Under these assumptions, when, for example, the k nearest neighbors in $\mathcal{N}$ to the query r ($k\text{NN}(r, \mathcal{N})$) are retrieved from the remote server, the total cost experienced by the system is $\sum_{o\in k\text{NN}(r,\mathcal{N})} c_d(r, o) + kc_f$.

B. Cache Indexes

AÇAI departs from the key-value indexes of most the similarity caching policies. As discussed in Sec. II, such approach was essentially motivated by the need to simplify kNN searches by performing two searches on smaller datasets (the set of keys first, and then the union of the values for l keys), and may lead to potential inefficiencies including sub-utilization of the available caching space.

The two-level search implemented by existing similarity caching policies can be seen as a naive way to implement an approximate kNN search on the set of objects stored locally (the *local catalog* $\mathcal{C}$). Thanks to the recent advances in approximate kNN searches (Sec. III), we have now better approaches to search through large catalogs with limited memory and computation requirements. We assume then that the cache maintains two indexes supporting kNN searches: one for the local catalog (the objects stored locally) and one for the remote catalog (the objects stored at the server). A discussion about which approximate index is more appropriate for each catalog is in Sec. III.

The local catalog index allows AÇAI to (i) fully exploit the available space (the cache stores at any time h objects and can perform a kNN search on all of them), (ii) potentially find closer objects in comparison to the non-optimized key-value search. Instead, the remote catalog index allows AÇAI to evaluate what objects the server would provide as answer to the request, and then to correctly evaluate which objects should be served locally and which one should be served from the server, as we are going to describe.

C. Request Serving

Differently from existing policies, AÇAI has the possibility to compose the answer using both local objects and remote ones. Upon a request r, AÇAI uses the two indexes to find the closest objects from the local catalog $\mathcal{C}$ and from the remote catalog $\mathcal{N}$. We denote the set of objects identified by these indexes as $k\text{NN}(r, \mathcal{C})$ and $k\text{NN}(r, \mathcal{N})$ respectively. AÇAI composes the answer $\mathcal{A}$ by combining the objects with the smallest costs in the two sets. For an object o stored locally ($o \in \mathcal{C}$), the system only pays $c_d(r, o)$; for an object o fetched from the remote server ($o \in \mathcal{N} \setminus \mathcal{C}$), the system pays $c_d(r, o) + d_f$. The total cost experienced is

$$C(r, \mathcal{A}) \triangleq \sum_{o\in\mathcal{A}\cap k\text{NN}(r,\mathcal{C})} c_d(r, o) + \sum_{o\in\mathcal{A}\setminus k\text{NN}(r,\mathcal{C})} (c_d(r, o) + c_f)\,. \tag{1}$$

The answer $\mathcal{A}$ is determined by selecting k objects that minimize the total cost, that is

$$\mathcal{A} = \underset{\substack{\mathcal{B}\subset k\text{NN}(r,\mathcal{C})\cup k\text{NN}(r,\mathcal{N}) \\ |\mathcal{B}|=k}}{\arg\min} C(r, \mathcal{B}). \tag{2}$$

D. Cache State and Service Cost/Gain

In order to succinctly present how AÇAI updates the local catalog and its theoretical guarantee, it is convenient to express the cost in (1) as function of the current cache state and replace the set notation with a vectorial one.

First, we define the *augmented catalog* $\mathcal{U}$ to be the set $\mathcal{N} \cup \{N + 1, N + 2, \ldots, 2N\}$ and define the new costs

$$c(r, i) = \begin{cases} c_d(r, i), & \text{if } i \in \mathcal{N}, \\ c_d(r, i - N) + c_f, & \text{if } i \in \mathcal{U} \setminus \mathcal{N}. \end{cases} \tag{3}$$

Essentially, i and $i+N$ (for $i \in \{1, \ldots, N\}$) correspond to the same object, with i capturing the cost when the object is stored at the cache and $i + N$ capturing the cost when it is stored at the server. From now on, when we talk about the closest objects to a request, we are referring to $c(., .)$ as distance.

It is also convenient to represent the state of the cache (the set of objects stored locally) as a vector $\boldsymbol{x} \in \{0, 1\}^{2N}$, where, for $i \in \mathcal{N}$, $x_i = 1$ (resp., $x_i = 0$), if i is stored (resp., is not

stored) in the cache, and we set $x_{i+N} = 1 - x_i$.[4] The set of valid cache configurations is given by:

$$\mathcal{X} \triangleq \left\{ \boldsymbol{x} \in \{0,1\}^{2N} : \sum_{i \in \mathcal{N}} x_i = h,\ x_{j+N} = 1 - x_j, \forall j \in \mathcal{N} \right\}. \tag{4}$$

For every request $r \in \mathcal{R}$ we define the sequence π^r as the permutation of the elements of $\mathcal{U}$, where π^r_i gives the i-th closest object to r in $\mathcal{U}$ according to the costs $c(r,o), \forall o \in \mathcal{U}$. The answer $\mathcal{A}$ provided by AÇAI (Eq. (2)) coincides with the first k elements of π^r for which the corresponding index in $\boldsymbol{x}$ is equal to 1. The total cost to serve r can then be expressed directly as a function of the cache state $\boldsymbol{x}$:

$$C(r, \boldsymbol{x}) = \sum_{i=1}^{2N} c(r, \pi^r_i) x_{\pi^r_i} \mathbb{1}_{\left\{\sum_{j=1}^{i-1} x_{\pi^r_j} < k\right\}}, \forall \boldsymbol{x} \in \mathcal{X}. \tag{5}$$

where $\mathbb{1}_{\{\chi\}} = 1$ when the condition χ is true, and $\mathbb{1}_{\{\chi\}} = 0$ otherwise.

Instead of working with the cost $C(r, \boldsymbol{x})$, we can equivalently consider the *caching gain* defined as the cost reduction due to the presence of the cache (as in [38]–[40]):

$$G(r, \boldsymbol{x}) \triangleq C(r, (\underbrace{0, 0, \ldots, 0}_{N}, \underbrace{1, 1, \ldots, 1}_{N})) - C(r, \boldsymbol{x}), \tag{6}$$

where the first term corresponds to the cost when the cache is empty (and then requests are entirely satisfied by the server). The theoretical guarantees of AÇAI are simpler to express in terms of the caching gain (Sec. IV-G).

The caching gain has the following compact expression [9, Lemma 6]:

$$G(r, \boldsymbol{x}) = \sum_{i=1}^{K^r - 1} \alpha^r_i \min \left\{ k - \sigma^r_i, \sum_{j=1}^{i} x_{\pi^r_j} - \sigma^r_i \right\}, \tag{7}$$

where

$$\sigma^r_i \triangleq \sum_{j=1}^{i} \mathbb{1}_{\left\{\pi^r_j \in \mathcal{U} \setminus \mathcal{N}\right\}}, \qquad \forall (i, r) \in \mathcal{U} \times \mathcal{R}, \tag{8}$$

K^r is the value of the minimum index $i \in \mathcal{N}$ such that $\sigma^r_i = k$, and $\alpha^r_i \triangleq c(r, \pi^r_{i+1}) - c(r, \pi^r_i) \geq 0$. Let $\mathrm{conv}(\mathcal{X})$ denote the convex hull of the set of valid cache configurations $\mathcal{X}$. We observe that $G(r, \boldsymbol{y})$ is a concave function of variable $\boldsymbol{y} \in \mathrm{conv}(\mathcal{X})$. Indeed, from Eq. (7), $G(r, \boldsymbol{y})$ is a linear combination, with positive coefficients, of concave functions (the minimum of affine functions in $\boldsymbol{y}$).

E. Cache Updates

We denote by $r_t \in \mathcal{R}$ the t-th request. The cache is allowed to change its state $\boldsymbol{x}_t \in \mathcal{X}$ to $\boldsymbol{x}_{t+1} \in \mathcal{X}$ in a reactive manner, after receiving the request r_t and incurring the gain $G(r_t, \boldsymbol{x}_t)$. AÇAI updates its state $\boldsymbol{x}_t$ with the goal of greedily maximizing the gain.

The update of the state $\boldsymbol{x}_t$ is driven from a continuous fractional state $\boldsymbol{y}_t \in [0,1]^{2N}$, where $(\boldsymbol{y}_t)_i$ can be interpreted as the probability to store object i in the cache. At each request r_t, AÇAI increases the components of $\boldsymbol{y}_t$ corresponding to the objects that are used to answer to r_t, and decreases the other components. This could be achieved by a classic gradient method, e.g., $\boldsymbol{y}_{t+1} = \boldsymbol{y}_t + \eta \mathbf{g}_t$, where $\mathbf{g}_t$ is a subgradient of $G(r_t, \boldsymbol{y}_t)$ and $\eta \in \mathbb{R}_+$ is the learning rate (or stepsize), but in AÇAI we consider a more general online mirror ascent update OMA [41, Ch. 4] that is described in Algorithm 1.[5] OMA is parameterized by the function $\Phi(\cdot)$, that is called the *mirror map*. If the mirror map is the squared Euclidean norm, OMA coincides with the usual gradient ascent method, but other mirror maps can be selected. In particular, our experiments in Sec. V show that the negative entropy map $\Phi(\boldsymbol{y}) = \sum_{i \in \mathcal{N}} y_i \log y_i$ achieves better performance.

Periodically, every M requests, AÇAI use the randomized rounding scheme DEPROUND [42] to generate a cache allocation $\boldsymbol{x}_{t+1} \in \{0,1\}^{2N}$ from $\boldsymbol{y}_{t+1} \in [0,1]^{2N}$, and the cache can fetch from the server the objects that are in $\boldsymbol{x}_{t+1}$ but not in $\boldsymbol{x}_t$. In [9] we discuss also an approach based on coupling the random variables $\boldsymbol{x}_t$ and $\boldsymbol{x}_{t+1}$ to significantly reduce the number of fetched objects to update the cache state.

F. Time Complexity

AÇAI uses OMA in Algorithm 1 coupled with a rounding procedure DEPROUND. The rounding step may take $\mathcal{O}(N)$ operations (amortized every M requests). In practice, AÇAI quickly sets irrelevant objects in the fractional allocation vector $\boldsymbol{y}_t$ very close to 0. Therefore, we can keep track only of objects with a fractional value above a threshold $\epsilon > 0$, and the size of this subset is practically of the order of h.

Similarly, subgradient computation may require $\mathcal{O}(N)$ operations per each component and then have $\mathcal{O}(N^2)$ complexity, but in practice, as the vector $\boldsymbol{y}_t$ is sparse, calculations in [9, Eq. (55)] require only a constant number of operations and complexity reduces to $\mathcal{O}(N)$.

Finally, we adapted the Euclidean projection algorithm in [43] to obtain a neg-entropy Bregman projection (line 6 of Algorithm 1) that has $\mathcal{O}(N \log N)$ time complexity. The $\mathcal{O}(N \log N)$ is due to a sorting operation while the actual projection takes $\mathcal{O}(h)$. Again, most of the components of $\boldsymbol{y}_t$ are equal to 0, so that we need to sort much less than N points.

G. Theoretical Guarantees

We observe that maximizing the gain (6) is NP-hard in general even for $k = 1$ under a stationary request process [18]. Nevertheless, AÇAI provides guarantees in terms of the ψ-regret [44]. In this scenario, the regret is defined as gain loss in comparison to the best static cache allocation in hindsight $\boldsymbol{x}_* \in \arg\max_{\boldsymbol{x} \in \mathcal{X}} \sum_{t=1}^{T} G(r_t, \boldsymbol{x})$. The ψ-regret discount the best

[4]The vector $\boldsymbol{x}$ has clearly redundant components, but such redundancy leads to more compact expressions in what follows.

[5]Properly speaking OMA, only refers to the update of $\boldsymbol{y}_t$ and does not include the randomized rounding scheme DEPROUND in lines 7–9.

Algorithm 1 Online Mirror Ascent (OMA_Φ)

procedure ONLINEMIRRORASCENT($\boldsymbol{y}_1 = \arg\min_{\boldsymbol{y}\in\mathcal{X}\cap\mathcal{D}} \Phi(\boldsymbol{y}), \boldsymbol{x}_1 =$ DEPROUND$(\boldsymbol{y}_1), \eta \in \mathbb{R}_+$)
  for $t = 1, 2, \ldots, T$ **do** ▷ Incur a gain $G(r_t, \boldsymbol{y}_t)$, and receive a subgradient $\boldsymbol{g}_t \in \partial_{\boldsymbol{y}} G(r_t, \boldsymbol{y}_t)$ using [9, Eq. (55)]
    $\hat{\boldsymbol{y}}_t \leftarrow \nabla\Phi(\boldsymbol{y}_t)$ ▷ Map primal point to dual point
    $\hat{\boldsymbol{z}}_{t+1} \leftarrow \hat{\boldsymbol{y}}_t + \eta \boldsymbol{g}_t$ ▷ Take gradient step in the dual
    $\boldsymbol{z}_{t+1} \leftarrow (\nabla\Phi)^{-1}(\hat{\boldsymbol{z}}_{t+1})$ ▷ Map dual point to a primal point
    $\boldsymbol{y}_{t+1} \leftarrow \prod^{\Phi}_{\mathcal{X}\cap\mathcal{D}}(\boldsymbol{z}_{t+1})$ ▷ Proj. new point onto feasible region
    if $M \mid t$ **then** ▷ Round the fractional state every M requests
      $\boldsymbol{x}_{t+1} \leftarrow$ DEPROUND$(\boldsymbol{y}_{t+1})$
    end if
  end for
end procedure

static gain by a factor $\psi \leq 1$. Formally,

$$\psi\text{-Regret}_{T,\mathcal{X}}(\text{OMA}_\Phi) = \sup_{\{r_1, r_2, \ldots, r_T\}\in\mathcal{R}^T} \left\{ \psi \sum_{t=1}^{T} G(r_t, \boldsymbol{x}_*) - \mathbb{E}\left[\sum_{t=1}^{T} G(r_t, \boldsymbol{x}_t)\right] \right\}, \tag{9}$$

where the expectation is over the randomized choices of DEPROUND. Note that the supremum in (9) is over all possible request sequences. This definition corresponds to the so called *adversarial analysis*, imagining that an adversary selects requests in $\mathcal{R}$ to jeopardize cache performance. This modeling approach is commonly used to characterize system performance under highly volatile external parameters (e.g., the sequence of requests r_t) and has been recently successfully applied to caching problems [27], [37], [45].

Obviously, regret bounds in the adversarial setting provide strong robustness guarantees in practical scenarios. AÇAI has the following regret guarantee:

Theorem IV.1. *Algorithm 1 with $M = 1$ has a sublinear $(1 - 1/e)$-regret in the number of requests, i.e., there exists a constant A such that:*

$$(1 - 1/e)\text{-Regret}_{T,\mathcal{X}}(\text{OMA}_\Phi) \leq A\sqrt{T}. \tag{10}$$

where $A \propto c_d^k + c_f$. c_d^k is an upper bound on the dissimilarity cost of the k-th closest object for any request in $\mathcal{R}$.

Proof. (sketch) We first prove that the expected gain of the randomly sampled allocations $\boldsymbol{x}_t$ is a $(1-1/e)$-approximation of the fractional gain. Then, we use online learning results [41] to bound the regret of OMA schemes operating on a convex decision space against concave gain functions picked by an adversary. The two results are combined to obtain an upper bound on the $(1-1/e)$-regret. We fully characterize the regret constant A in [9]. □

A consequence of Theorem IV.1 is that the expected time-average $(1 - 1/e)$-regret of AÇAI can get arbitrarily close to zero for large time horizon. Hence, AÇAI performs on average as well as a $(1 - 1/e)$-approximation of the optimal configuration $\boldsymbol{x}_*$. This observation also suggests that our algorithm can be used as an iterative method to solve the NP-hard static allocation problem with the best approximation bound achievable for this kind of problems [46].

Corollary IV.1.1. *(offline solution) Let $\bar{\boldsymbol{y}}$ be the average fractional allocation $\bar{\boldsymbol{y}} = \frac{1}{T}\sum_{t=1}^{T} \boldsymbol{y}_t$ of* AÇAI*, and $\bar{\boldsymbol{x}}$ the random state sampled from $\bar{\boldsymbol{y}}$ using* DEPROUND*. $\forall \epsilon > 0$ and over a sufficiently large time horizon T, $\bar{\boldsymbol{x}}$ satisfies*

$$\mathbb{E}\left[\frac{1}{T}\sum_{t=1}^{T} G(r_t, \bar{\boldsymbol{x}})\right] \geq \left(1 - \frac{1}{e} - \epsilon\right)\frac{1}{T}\sum_{t=1}^{T} G(r_t, \boldsymbol{x}_*).$$

where $\boldsymbol{x}_* = \arg\max_{\boldsymbol{x}\in\mathcal{X}} \sum_{t=1}^{T} G(r_t, \boldsymbol{x})$.

V. EXPERIMENTS

We evaluate AÇAI under different real world traces. We compare our solution with state of the art online policies proposed for kNN caching, i.e., SIM-LRU [17], CLS-LRU [17], and QCACHE [26] described in Sec. II.

A. Data-sets Description

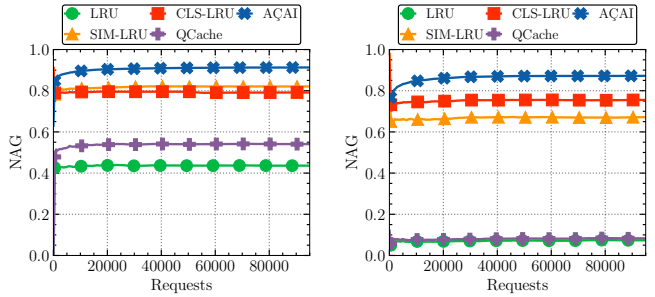
SIFT1M trace. SIFT1M is a classic benchmark data-set to evaluate approximate kNN algorithms [47]. It contains 1 million objects embedded as points in a 128-dimensional space. SIFT1M does not provide a request trace so we generated a synthetic one according to the Independent Reference Model [48] (similar to what done in other papers like [16], [23]). Request r_t is for object i with a constant probability λ_i independently from previous requests. We spatially correlated requests by letting λ_i depend on the position of the embeddings in the space. In particular, we considered the barycenter of the whole dataset and set λ_i proportional to $d_i^{-\beta}$, where d_i is the distance of i from the barycenter. The parameter β was chosen such that the tail of the ranked objects popularity distribution is similar to a Zipf with parameter 0.9, as observed in some image retrieval systems [26]. We generated a trace with 10^5 requests. The number of distinct objects requested in the trace is approximately 2×10^4.

Amazon trace. The authors of [49] crawled the Amazon webstore and collected a dataset to model relationships among products and provide user recommendations. They took as input the visual features of product images obtained from a machine learning model pre-trained on 1.2 million images from ImageNet. They also collected information about which users purchased or viewed the objects. The authors of [25] built a request trace from the timestamped user reviews for objects in the category Baby embedded in a 100-dimensional space. Two products o and o' are considered similar if they have been viewed by the same users. We use the request trace from [25], and in particular the interval $[2 \times 10^5, 3 \times 10^5]$.[6] The number of distinct objects requested in this trace is approximately 2×10^4.

B. Settings and Performance Metrics

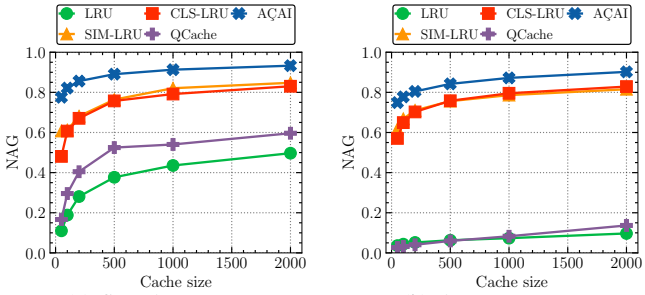
For AÇAI, unless otherwise said, we choose the negative entropy $\Phi(\boldsymbol{y}) = \sum_{i\in\mathcal{N}} y_i \log(y_i)$ as mirror map (see Fig. 6 and the corresponding discussion for other choices). The learning rate is set to the best value found exploring the range $[10^{-6}, 10^{-4}]$.

[6] We discard the initial part of the trace because it contains requests only for a small set of objects (likely the set of products to crawl was progressively extended during the measurement campaign in [49]).



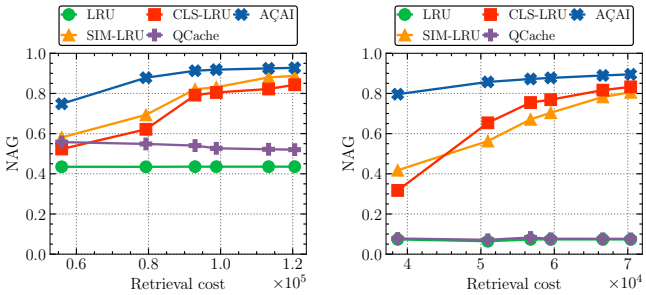
(a) SIFT1M trace (b) Amazon trace

Fig. 1: Caching gain for the different policies. The cache size is $h = 1000$ and $k = 10$.



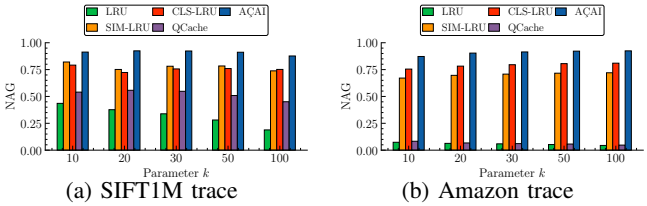
(a) SIFT1M trace (b) Amazon trace

Fig. 2: Caching gain for the different policies, for different cache sizes $h \in \{50, 100, 200, 500, 1000, 2000\}$ and $k = 10$.



(a) SIFT1M trace (b) Amazon trace

Fig. 3: Caching gain for the different policies and different retrieval cost. The retrieval cost c_f is taken as the average distance to the i-th neighbor, $i \in \{2, 10, 50, 100, 500, 1000\}$. The cache size is $h = 1000$ and $k = 10$.



(a) SIFT1M trace (b) Amazon trace

Fig. 4: Caching gain for the different policies. The cache size is $h = 1000$, and $k \in \{10, 20, 30, 50, 100\}$.

As for the state-of-the-art caching policies, SIM-LRU and CLS-LRU have two parameters, C_θ and k', that we set in each experiment to the best values we found exploring the ranges $[c_f, 2c_f]$ for C_θ and $[1, h]$ for k'. For QCACHE we consider $l = h/k$: the cache can then perform the kNN search over all local objects.

We also consider a simple similarity caching policy that stores previous requests and the corresponding set of k closest objects as key-value pairs, and manages the set of keys according to LRU. The cache then serves locally the request if it coincides with one of the previous requests in its memory, it forwards it to the server, otherwise. The ordered list of keys is updated as in LRU. We refer to this policy simply as LRU.

For every caching policy $\mathcal{P}$, we can define its caching gain $G_{\mathcal{P}}(r, \boldsymbol{x})$ as in Eq. (6). We can then compare the policies in terms of their normalized average caching gain per-request, where the normalization factor corresponds to the caching gain of a cache with size equal to the whole catalog. In such case, the cache could store the entire catalog locally and would achieve the same dissimilarity cost of the server without paying any fetching cost. The maximum possible caching gain is then kc_f. The normalized average gain over T requests can then be defined as:

$$\mathrm{NAG}(\mathcal{P}) = \frac{1}{kc_f T} \sum_{t=1}^{T} G_{\mathcal{P}}(r_t, \boldsymbol{x}_t). \tag{11}$$

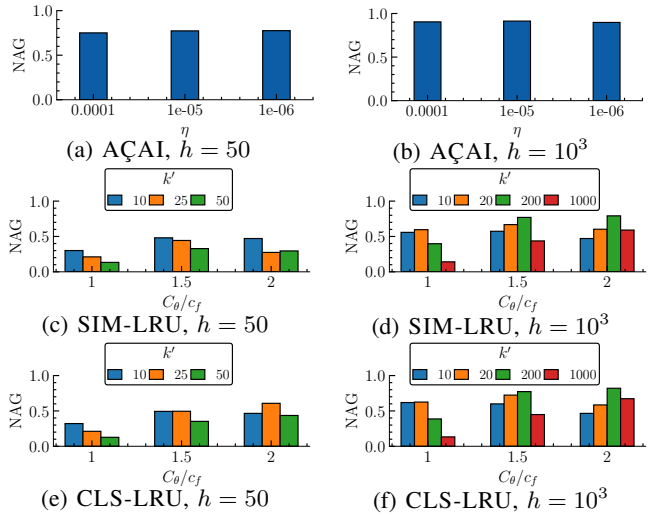
C. Results

We consider a dissimilarity cost proportional to the squared Euclidean distance. This is the usual metric considered for SIFT1M benchmark and also the one considered to learn the embeddings for the Amazon trace in [49].

The numerical value of the fetching cost depends on its interpretation (delay experienced by the user, load on the server or on the network) as well as on the application, because it needs to be converted into the same "unit" of the approximation cost. In our evaluation, we let it depend on the topological characteristics of the dataset in order to be able to compare the results for the two different traces. Unless otherwise said, we set c_f equal to the average distance of the 50-th closest neighbor in the catalog $\mathcal{N}$.

Figure 1 shows how the normalized average gain changes as requests arrive and the different caching policies update the local set of objects (starting from an empty configuration). The cache size is $h = 1000$ and the cache provides $k = 10$ similar objects for each request. All policies reach an almost stationary gain after at most a few thousand requests. Unsurprisingly, the naive LRU has the lowest gain (it can only satisfy locally requests that match exactly a previous request) and similarity caching policies perform better. AÇAI has a significant improvement in comparison to the second best policy (SIM-LRU for SIFT1M and CLS-LRU for Amazon).

This advantage of AÇAI is constantly confirmed for different cache sizes (Fig. 2), different values of the fetching cost c_f (Fig. 3), and different values of k (Fig. 4). The relative improvement of AÇAI, in comparison to the second best policy, is larger for small values of the cache size ($+30\%$ for SIFT1M and $+25\%$ for Amazon when $h = 50$), and small values of the fetching cost ($+35\%$ for SIFT1M and $+100\%$ for c_f equal to the average distance from the second closest object). Note how these are the settings where caching choices



(a) AÇAI, $h = 50$ (b) AÇAI, $h = 10^3$

(c) SIM-LRU, $h = 50$ (d) SIM-LRU, $h = 10^3$

(e) CLS-LRU, $h = 50$ (f) CLS-LRU, $h = 10^3$

Fig. 5: Caching gain for AÇAI for different values of η (top). Caching gain for SIM-LRU(middle) and CLS-LRU (bottom) for different values of the parameters (k', C_θ). SIFT1M trace.

are more difficult (and indeed all policies have lower gains): when cache storage can accommodate only a few objects, it is critical to carefully select which ones to store; when the server is close, the costs of serving requests from the cache or from the server are similar and it is difficult to correctly decide how to satisfy the request. Caching policies performance are in general less dependent on the number k of similar objects to retrieve and AÇAI achieves about 10% improvement for k between 10 and 100 when $h = 1000$ (Fig. 4).

Sensitivity analysis. We now evaluate the robustness of AÇAI to the configuration of its single parameter (the learning rate η). Figure 5 shows indeed that, for learning rates that are two orders of magnitude apart, we can achieve almost the same normalized average gain both for $h = 50$ and for $h = 1000$.[7]

In contrast, the performance of the second best policies (SIM-LRU and CLS-LRU) are more sensitive to the choice of their two configuration parameters k' and C_θ. For example, the optimal configuration of SIM-LRU is $k' = 10$ and $C_\theta = 1.5 \times c_f$ for a small cache ($h = 50$) but $k' = 200$ and $C_\theta = 2 \times c_f$ for a large one ($h = 1000$). Moreover, in both cases a misconfiguration of these parameters would lead to significant performance degradation.

Choice of the mirror map. If the mirror map is selected equal to the squared Euclidean norm, the OMA update coincides with a standard gradient update. Figure 6 shows the superiority of the negative entropy map: it allows to achieve a higher gain than the Euclidean norm map or the same gain but in a shorter time. To the best of our knowledge, ours is the first paper that shows the advantage of using non-Euclidean mirror maps for caching problems. It is possible to justify theoretically this result, considering the difference of subgradients norms for

[7]Under a stationary request process, a smaller learning rate would lead to converge slower but to a solution closer to the optimal one. Under a non-stationary process, a higher learning rate may allow faster adaptivity. In this trace, the two effects almost compensate, but see also Fig. 6.

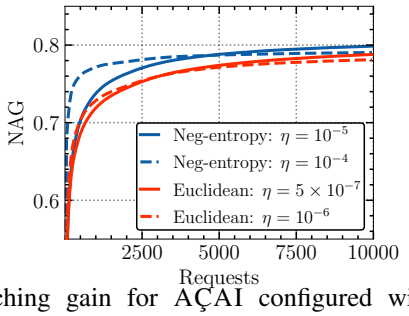
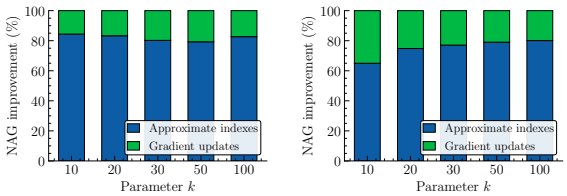


Fig. 6: Caching gain for AÇAI configured with negative entropy and Euclidean maps (SIFT1M trace). The cache size is $h = 100$ and $k = 10$.



(a) SIFT1M trace (b) Amazon trace

Fig. 7: AÇAI caching gain improvement in comparison to the second best state-of-the-art similarity caching policy: contribution of approximate indexes and gradient updates. The cache size is $h = 1000$, and $k \in \{10, 20, 30, 50, 100\}$.

similarity caching and exact caching problems [9, Appendix D].

Dissecting AÇAI performance. In comparison to state-of-the-art similarity caching policies, AÇAI introduces two key ingredients: (i) the use of fast, approximate indexes to decide what to serve from the local catalog and what from the remote one, and (ii) the OMA algorithm to update the cache state. It is useful to understand how much each ingredient contributes to AÇAI improvement with respect to the other policies.

To this aim, we integrated the same indexes in the other policies allowing them to serve requests as AÇAI does, combining both local objects and remote ones on the basis of their costs (see Sec. IV-C), while leaving their cache updating mechanism unchanged. We then compute in the setting of Fig. 4 how much the gain of the second best policy (SIM-LRU for SIFT1M and CLS-LRU for Amazon) increases because of the use of AÇAI request service mechanism. This is the part of AÇAI improvement attributed to the use of the indexes, the rest is attributed to the cache update mechanism through OMA.

We observe from Fig. 7 that most of AÇAI gain improvement over the second best caching policy is due to the use of approximate indexes, but OMA updates are still responsible for 15–20% of AÇAI performance improvement under SIFT1M trace and for 20–35% for the Amazon trace.

VI. Conclusion

Edge computing provides computing and storage resources that may enable complex applications with tight delay guarantee like augmented-reality ones, but these strategically positioned resources need to be used efficiently. To this aim,

we designed AÇAI, a content cache management policy that determines dynamically the best content to store on the edge server to reply to similarity search queries. Our solution adapts to the user requests, without any assumption on the traffic arrival pattern. AÇAI leverages on two key components: (i) new efficient content indexing methods to keep track of both local and remote content, and (ii) mirror ascending techniques to optimally select the content to store. The results show that AÇAI is able to outperform the state-of-the-art policies and does not need careful parameter tuning.

As future work, we plan to evaluate AÇAI in the context of machine learning classification tasks [50], in which the size of the objects in the catalog is comparable to their d-dimensional representation in the index, and, as a consequence, the index size cannot be neglected in comparison to the local catalog size.

REFERENCES

[1] T. Y.-H. Chen *et al.*, "Glimpse: Continuous, Real-Time Object Recognition on Mobile Devices," in *Proc. of the ACM SenSys*, 2015, pp. 155–168.

[2] Z. Zhou *et al.*, "Edge intelligence: Paving the last mile of artificial intelligence with edge computing," *Proc. of the IEEE*, vol. 107, no. 8, pp. 1738–1762, 2019.

[3] S. Venugopal *et al.*, "Shadow puppets: Cloud-level accurate AI inference at the speed and economy of edge," in *USENIX HotEdge* , 2018.

[4] U. Drolia *et al.*, "Cachier: Edge-caching for recognition applications," in *Proc. of the IEEE ICDCS*. IEEE, 2017, pp. 276–286.

[5] U. Drolia *et al.*, "Precog: Prefetching for image recognition applications at the edge," in *Proc. of ACM/IEEE Symposium on Edge Computing*, 2017, pp. 1–13.

[6] P. Guo *et al.*, "Foggycache: Cross-device approximate computation reuse," in *Proc. of the MobiCom*, 2018, pp. 19–34.

[7] A. Bellet *et al.*, "Metric learning," *Synthesis Lectures on Artificial Intelligence and Machine Learning*, vol. 9, no. 1, pp. 1–151, 2015.

[8] J. Johnson *et al.*, "Billion-scale similarity search with gpus," *IEEE Transactions on Big Data*, 2019.

[9] T. Si Salem *et al.*, "AÇAI: Ascent Similarity Caching with Approximate Indexes," *arXiv preprint arXiv:2107.00957*.

[10] D. G. Lowe, "Object recognition from local scale-invariant features," in *Proc. of the IEEE ICCV*, vol. 2. Ieee, 1999, pp. 1150–1157.

[11] E. Rublee *et al.*, "Orb: An efficient alternative to sift or surf," in *Proc. of the IEEE ICCV*. Ieee, 2011, pp. 2564–2571.

[12] G. E. Hinton *et al.*, "Reducing the dimensionality of data with neural networks," *science*, vol. 313, no. 5786, pp. 504–507, 2006.

[13] K. Lin *et al.*, "Learning compact binary descriptors with unsupervised deep neural networks," in *Proc. of the IEEE CVPR*, 2016, pp. 1183–1192.

[14] P. N. Yianilos, "Data structures and algorithms for nearest neighbor search in general metric spaces," in *Soda*, vol. 93, 1993, pp. 311–21.

[15] G. Navarro, "Searching in metric spaces by spatial approximation," *The VLDB Journal*, vol. 11, no. 1, pp. 28–46, 2002.

[16] F. Falchi *et al.*, "A metric cache for similarity search," in *Proc. of the ACM LSDS-IR*, 2008, pp. 43–50.

[17] S. Pandey *et al.*, "Nearest-neighbor caching for content-match applications," in *Proc. of the WWW*, 2009, pp. 441–450.

[18] M. Garetto *et al.*, "Similarity caching: Theory and algorithms," in *Proc. of the IEEE Infocom*, 2020.

[19] P. Sermpezis *et al.*, "Soft cache hits: Improving performance through recommendation and delivery of related content," *IEEE Journal on Selected Areas in Communications*, vol. 36, no. 6, pp. 1300–1313, 2018.

[20] M. Costantini *et al.*, "Impact of popular content relational structure on joint caching and recommendation policies," in *2020 18th International Symposium on Modeling and Optimization in Mobile, Ad Hoc, and Wireless Networks (WiOPT)*, 2020, pp. 1–8.

[21] D. Crankshaw *et al.*, "Clipper: A low-latency online prediction serving system," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, 2017, pp. 613–627.

[22] G. Paschos *et al.*, "Wireless caching: technical misconceptions and business barriers," *IEEE Communications Magazine*, vol. 54, no. 8, pp. 16–22, 2016.

[23] P. Guo *et al.*, "Potluck: Cross-application approximate deduplication for computation-intensive mobile applications," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, 2018, pp. 271–284.

[24] J. Zhou *et al.*, "Adaptive offline and online similarity-based caching," *IEEE Networking Letters*, vol. 2, no. 4, pp. 175–179, 2020.

[25] A. Sabnis *et al.*, "Grades: Gradient descent for similarity caching," in *IEEE Conference on Computer Communications (INFOCOM)*, 2021.

[26] F. Falchi *et al.*, "Similarity caching in large-scale image retrieval," *Information Processing & Management*, vol. 48, no. 5, pp. 803 – 818, 2012.

[27] G. S. Paschos *et al.*, "Learning to cache with no regrets," in *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*. IEEE, 2019, pp. 235–243.

[28] R. Weber *et al.*, "A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces," in *VLDB*, vol. 98, 1998, pp. 194–205.

[29] A. Andoni *et al.*, "Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions," in *IEEE FOCS*, 2006, pp. 459–468.

[30] A. Babenko *et al.*, "The inverted multi-index," *IEEE trans. on pattern analysis and machine intelligence*, vol. 37, no. 6, pp. 1247–1260, 2014.

[31] Y. A. Malkov *et al.*, "Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs," *IEEE trans. on pattern analysis and machine intelligence*, 2018.

[32] M. Aumüller *et al.*, "Ann-benchmarks: A benchmarking tool for approximate nearest neighbor algorithms," in *Proc. of the SISAP*. Springer, 2017, pp. 34–49.

[33] S. Ioannidis *et al.*, "Distributed caching over heterogeneous mobile networks," in *Proceedings of the ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, 2010, pp. 311–322.

[34] S. Ioannidis *et al.*, "Adaptive caching networks with optimality guarantees," *SIGMETRICS Perform. Eval. Rev.*, vol. 44, no. 1, pp. 113–124, 2016.

[35] J. Zhou *et al.*, "Adaptive offline and online similarity-based caching," *IEEE Networking Letters*, 2020.

[36] S. Shalev-Shwartz, "Online learning and online convex optimization," *Machine Learning*, vol. 4, no. 2, pp. 107–194, 2011.

[37] T. Si Salem *et al.*, "No-Regret Caching via Online Mirror Descent," in *IEEE International Conference on Communications (ICC)*, 2021.

[38] K. Shanmugam *et al.*, "Femtocaching: Wireless content delivery through distributed caching helpers," *IEEE Trans. on Information Theory*, vol. 59, no. 12, pp. 8402–8413, 2013.

[39] S. Ioannidis *et al.*, "Adaptive caching networks with optimality guarantees," *SIGMETRICS Perform. Eval. Rev.*, vol. 44, no. 1, pp. 113–124, Jun. 2016.

[40] G. Neglia *et al.*, "A swiss army knife for dynamic caching in small cell networks," *arXiv preprint arXiv:1912.10149*, 2019.

[41] S. Bubeck, "Convex optimization: Algorithms and complexity," *Found. Trends Mach. Learn.*, vol. 8, no. 3-4, pp. 231–357, Nov. 2015.

[42] J. Byrka *et al.*, "An improved approximation for k-median, and positive correlation in budgeted optimization," in *Proc. of ACM-SIAM symposium on Discrete algorithms*, 2014, pp. 737–756.

[43] W. Wang *et al.*, "Projection onto the capped simplex," 2015.

[44] A. Krause *et al.*, "Submodular function maximization," in *Tractability: Practical Approaches to Hard Problems*. Cambridge University Press, 2014, pp. 71–104.

[45] R. Bhattacharjee *et al.*, "Fundamental limits on the regret of online network-caching," *ACM Meas. Anal. Comput. Syst.*, Jun. 2020.

[46] G. L. Nemhauser *et al.*, "Best algorithms for approximating the maximum of a submodular set function," *Math. of Op. Res.*, 1978.

[47] H. Jegou *et al.*, "Product quantization for nearest neighbor search," *IEEE trans. on pattern analysis and machine intelligence*, vol. 33, no. 1, pp. 117–128, 2010.

[48] E. G. Coffman *et al.*, *Operating systems theory*. Prentice-Hall Englewood Cliffs, NJ, 1973, vol. 973.

[49] J. McAuley *et al.*, "Image-based recommendations on styles and substitutes," in *Proc. of the ACM SIGIR*, 2015, pp. 43–52.

[50] U. Khandelwal *et al.*, "Generalization through memorization: Nearest neighbor language models," in *Proc. of the ICLR*, 2020.