École Polytechnique Fédérale de Lausanne

Bachelor Project Report

Predicting the Performance of a Vector Database Cache on Unknown Hardware Platforms

presented by

Zacharie Perego

Supervisors:

Dr. Rafael Pires

Mathis Randl

June 2025

Contents

1 Abstract **1**

2 Introduction **2**

3 Preliminaries **3**

3.1 Proximity 3

3.2 Definitions 3

3.3 Metrics 4

4 Core **6**

4.1 Overview 6

4.2 Methodology 6

4.3 Adaptation of the standard Proximity usage 7

4.3.1 Original usage of Proximity 7

4.3.2 Adapted usage of Proximity 7

4.4 Creating the Dataset 8

4.4.1 Test design 8

4.4.2 Features selection 10

4.4.3 Building the dataset 10

4.5 Training Models 11

5 Evaluation **12**

5.1 Experimental Setup 12

5.2 Model results 13

5.2.1 Analysis 14

5.3 Model Perfomance on unknown machine 15

5.3.1 Analysis 16

5.4 Feature Importance 16

5.4.1 Analysis 17

6 Discussion **19**

7 Conclusion **20**

8 Complementary Material **21**

8.1 Second Test Machine 21

8.2 Benchmark results for machines in the dataset 22

Abstract

Large language models (LLMs) are becoming increasingly popular, to precisely answer more complex and domain-specific queries, they are often used in combination with vector databases to access accurate information. This combination is called Retrieval-Augmented Generation (RAG).

One of the main challenges of RAG is the latency introduced by the queries performed by the LLM to the vector database. To reduce this latency, Proximity [2], a vector database cache, has been developed. It allows to store recently queried results and return them without consulting the database when the query is similar to a previously cached one. Proximity is highly dependent on the hardware it runs on and its performance can vary significantly across different machines. Being able to predict the performance of Proximity on computers we do not have access to would be highly valuable to optimize the overall performance of the system.

This work studies how to predict the performance of Proximity on machines we do not have physical access to, using only their technical details (datasheet). To achieve this, we built a machine learning model capable of predicting the cache performance.

We evaluated it on two unseen computers and the model produced predictions with all errors below 35%. It also allowed us to identify that memory is the hardware component having the most significant impact on Proximity's performance.

Introduction

Proximity [2] is a high performance Python library written in Rust that provides vector database caching functionality optimized for Retrieval Augmented Generation (RAG) systems. It is an intermediary layer that can significantly reduce latency by avoiding index searches if queries are sufficiently similar to previously cached results.

To retrieve a result in the cache, Proximity iterates over all the cached values. If no vector in the cache is enough similar to the requested one, the system has to search in the indexes, making the cache search an extra step that adds delay. For this reason, the execution speed of Proximity is especially important to make its use worthwhile. However, Proximity's performance can vary significantly depending on the hardware it runs on. Predicting how well Proximity will perform on a new machine is challenging, especially if we do not have a physical access to it.

In this work, we propose a solution using a machine learning approach to predict how well the cache will work on unknown computers. To do so, we collected data by testing Proximity on various hardware configurations and then train a machine learning model to predict three key metrics depending on the cache size and tolerance parameters: (1) the time needed to return a result from the cache (cache hit), (2) the time needed to return a result from the indexes when the cache fails (cache miss), and (3) the number of queries per second the process can handle. It enabled us to generate heatmaps displaying the computer's performance across multiple cache sizes and cache tolerances.

Preliminaries

3.1 Proximity

Proximity [2] is a Python Library written in Rust. It implements a Least Recently Used (LRU) policy and uses the following parameters:

- **Cache size:** Maximum number of vectors that can be stored in the cache, before it starts removing the least recently used ones.
- **Cache tolerance:** The maximum Euclidean (L2) distance between the queried vector and cached vectors to consider them similar.

3.2 Definitions

Vector Database

"A vector database is any database that can natively store and manage vector. [...] Vectors that are mathematically close to one another tend to describe objects with similar features, so you can quickly compare or search them and return objects that are alike." [8]

IVF-PQ indexing

"IVF-PQ is a composite index that combines inverted file index (IVF) and product quantization (PQ)."[6]

- "**Product quantization (PQ)** works by dividing a large, high-dimensional vector of size into equally sized subvectors. Each subvector is assigned a "reproduction value" that maps to the nearest centroid of points for that subvector. The reproduction values are then assigned to a codebook using unique IDs, which can be used to reconstruct the original vector." [6]

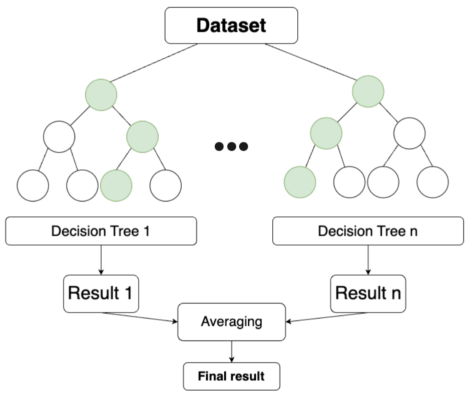
- ”In **IVF**, the PQ vector space is divided into [...] partitions that consist of all the points in the space that are within a threshold distance of the given region’s seed point. [...] These regions are then used to create an inverted index that correlates each seed point with a list of vectors in the space, allowing a search to be restricted to just a subset of vectors in the index.” [6]

Retrieval-Augmented Generation (RAG) systems

”Retrieval-Augmented Generation (RAG) is the process of optimizing the output of a large language model, so it references an authoritative knowledge base outside of its training data sources before generating a response.” [1]

Random Forest

”Random forest is a machine learning algorithm [...] that combines the output of multiple decision trees to reach a single result.” [5]



Permutaion Importance

”Permutation feature importance is a model inspection technique that measures the contribution of each feature to a fitted model’s statistical performance on a given tabular dataset. This technique is particularly useful for non-linear or opaque estimators, and involves randomly shuffling the values of a single feature and observing the resulting degradation of the model’s score. By breaking the relationship between the feature and the target.” [10]

3.3 Metrics

- **Query latency:** Time (in milliseconds) to process a search request.
- **Hit rate:** Percentage of queries for which results are found in the cache.

- **Cache hit latency:** Time $[ms]$ to process a search request when the result is found in the cache.
- **Cache miss latency:** Time $[ms]$ to process a search request when the result is not found in the cache.
- **Query throughput:** Number of queries handled per second by the process $[1/s]$.
- **Machine learning model evaluation metrics: [11]**
  - Mean Absolute Error: $\text{MAE}(y, \hat{y}) = \frac{1}{n_{\text{samples}}} \sum_{i=0}^{n_{\text{samples}}-1} \left| y_i - \hat{y}_i \right|.$
  - Mean Absolute Percentage Error: $\text{MAPE}(y, \hat{y}) = \frac{1}{n_{\text{samples}}} \sum_{i=0}^{n_{\text{samples}}-1} \frac{|y_i - \hat{y}_i|}{\max(\epsilon, |y_i|)}$

    ϵ is an arbitrary small strictly positive number to avoid undefined results.
  - Mean Squared Error: $\text{MSE}(y, \hat{y}) = \frac{1}{n_{\text{samples}}} \sum_{i=0}^{n_{\text{samples}}-1} (y_i - \hat{y}_i)^2.$
  - R-squared: $R^2(y, \hat{y}) = 1 - \frac{\sum_{i=1}^{n} (y_i - \hat{y}_i)^2}{\sum_{i=1}^{n} (y_i - \bar{y})^2}$

    ”It provides an indication of goodness of fit and therefore a measure of how well unseen samples are likely to be predicted by the model”

Core

4.1 Overview

To predict the performance of Proximity on various hardware machines, we propose a machine learning approach based on the hardware specifications of the machine and the cache configuration. The model is trained to predict three metrics (c.f. 3.2 for details):

(1) Cache hit latency **(2)** Cache miss latency **(3)** Query throughput

The model is trained on a dataset built from the results of running Proximity on various hardware configurations, using the *BigANN* dataset [4] providing also the queries and the indexes.

4.2 Methodology

Our approach to predict the performance of the cache on various hardware machines follows these key steps:

1. Adapt the standard Proximity usage to focus only on the vector searching functionality, removing components that require a lot of computing resources (section 4.3).

2. Design a test to measure performance metrics across different machines (section 4.4.1).

3. Identify relevant hardware features that could impact cache performance and build a dataset (section 4.4.2).

4. Use this dataset to train machine learning models and predict cache performance metrics (hit latency, miss latency, and query throughput).

5. Evaluate the model's performance and its ability to generalize to unknown computers.

6. Isolate the most significant features that impact Proximity's performance.

4.3 Adaptation of the standard Proximity usage

4.3.1 Original usage of Proximity

Originally, with the LLM part and the indexes, running the process required at least 200GB of RAM, 400GB of disk space and one performant GPU.

```
Algorithm 1: Original usage of Proximity
   Input: A user query
   Output: The text passage from the database added to the user query
1  Convert the user query into a vector using a retriever
2  Check if a similar vector is present in the cache (cache tolerance)
3  if a similar vector is found in the cache then
4      index ← Cached result
5  else
6      index ← Search for a result in the retriever
7      Add index in the cache
8  passage ← database.getPassageFromIndex(index)
9  return The original user's query + passage
```

Note1: the returned passage is then used by the LLM to formulate a response. **Note2:** In practice, there is more than one vector to retrieve, resulting in a mix of cache hits and retriever searches.

4.3.2 Adapted usage of Proximity

Our goal was to run Proximity on smaller machines (laptops, etc.) to be able to collect enough data and build a sufficiently large dataset. To achieve this, we removed the LLM part. It was only used to generate the vectors to be searched. Therefore, it was not relevant for evaluating Proximity's performance.(cf. section 4.3.1). Additionally, fetching the passage from the database using the retrieved index was also removed, since the database was taking too much space and, in the absence of the LLM, no longer needed.

Algorithm 2: Adapted usage of Proximity for this work

```
Input: A random vector query
Output: nothing
1 Check if a similar vector (cache tolerance) is present in the cache
    (Proximity)
2 if a similar vector is found in the cache then
3     index ← Cached result
4 else
5     index ← Search for a result in the indexes
6     Add index in the cache
```

We selected the *BigANN* dataset [4] as it is a well-known dataset for vector search and is accompanied by a set of 10k queries, replacing the ones provided by LLM. We used the *Sample data* composed of 100 million points instead of the full one composed of 1 billion base points. To build the indexes, we had two considerations: (1) the indexes should be stored in the memory to avoid disk access, which would be very inconsistant on different machines, and (2) the indexes should be small enough to fit on a 8GB RAM machine. The precision of the indexes is not critical, as we are not interested in the results, but only in the time it takes to retrieve them. We used *faiss* [3] to build *Inverted File Product Quantization* (IVF-PQ) indexes (c.f. 3.2) allowing to compress a 128-dimensional float32 vector (512 bytes) into a 32-dimensional 8-bits vector (32 bytes). It allowed us to build indexes of 2.24GB, which is small enough to fit on a 8GB RAM machine and should not be affected by other processes running on the machine. [1]

To summarize

To be able to benchmark relatively small computers we:

- Removed the LLM part and the database usage.
- Used the *BigANN* dataset [4] with 100M points and built 2.24GB IVF-PQ indexes.

4.4 Creating the Dataset

4.4.1 Test design

To facilitate testing across multiple computers, we created a folder containing the indexes, query files, benchmark Python script, and a shell script. The script

[1]Given that no other resource-intensive process is running on the computer at the same time.

performs the following steps: **(1)** creates a Python virtual environment, **(2)** Build Proximity with *Maturin* [7], **(3)** installs *Faiss* to read the raw indexes, **(4)** runs the test Python file **(5)** saves the results in a file.

To benchmark a machine, simply run the script and collect the resulting file.

Python benchmark script

The script follows the idea presented in the algorithm 2 (c.f. section 4.3.2).

The benchmark script randomly selects 7,000 queries from the 10,000 available in the *BigANN* queryset, using a fixed random seed. For each combination of cache size and cache tolerance (c.f. algo. 3), it iterates through these queries, measuring whether each query results in a cache hit or miss and the corresponding latency. After processing all queries for a given cache configuration, it computes and saves the *mean overall latency* [2], *mean cache hit latency*, and *mean cache miss latency* to an output file. The file contains 30 rows at the end of the script (length of cache capacity · length of cache tolerance).

Algorithm 3: Benchmark script

```
Input: None
Output: File containing the measurements
1  foreach cacheCapacity ∈ [10, 50, 100, 200, 300] do
2      foreach cacheTolerance ∈ [0, 200, 300, 350, 400, 500] do
3          foreach query ∈ 7000 queries do
4              Check if a similar vector (within cacheTolerance) is present
                 in the cache
5              if a similar vector is found in the cache then
6                  index ← Cached result
7                  Record cache hit latency
8              else
9                  index ← Search for a result in the indexes
10                 Add index in the cache
11                 Record cache miss latency
12         Write: mean latency, mean cache hit latency, mean cache miss
             latency
13 Return the 30 rows file
```

The selection of the **cache capacities** is based on the *Proximity* paper [2], while the **cache tolerances** were empirically chosen to span the range of cache hit rate, from 0% (no cache hits) to 99%.

[2]Overall latency allows to compute the cache queries per second ($\frac{1}{\text{latency}}$)

4.4.2 Features selection

Selecting the most relevant features from a machine's datasheet to predict the performance of Proximity is a crucial part of our approach. These features must be impactful and should not mislead the model. They must be able to describe a maximum amount of machine types and architectures (server, laptop, macbook, etc.).

Selected features

- **Machine type**: Architecture of the CPU (`x86_64` or `arm64`).
- **Total Cores**: Total CPU cores including logical/virtual cores.
- **Max CPU Frequency**: Maximum processor frequency $[MHz]$.
- **CPU year**: The year of the CPU release.
- **Memory MT/s**: Memory transfer rate in MT/s (million transfers per second).
- **Memory Type**: Type of RAM (DDR3, DDR4).
- **Memory in SoC**: Whether memory is integrated within the System on Chip. The transfer latency is lower if the memory is in the SoC.

Not selected features

- **CPU Manufacturing Process**: The manufacturing process (e.g. 7nm, 14nm) affects mainly the power efficiency and thermal characteristics.
- **CPU Cache**: The characteristics of L1, L2, and L3 CPU caches are relevant for evaluating processor performance. However, due to significant variability in architectural designs across different CPUs, establishing consistent comparisons is a complex challenge.
- **Disk specs**: As everything is stored in memory, everything related to disks is not relevant.

4.4.3 Building the dataset

The model is trained to predict three target variables: cache hit latency, cache miss latency, and query throughput (defined as 1/cache latency). To ensure the model is not biased by configurations when the cache is not used, all rows with zero cache hits (7 per machine) are excluded from the dataset. It happens when the cache tolerance is too small. As a result, each machine contributes for a total of 23 rows for each combination of cache size and cache tolerance where at least one cache hit occures.

4.5 Training Models

We used the *scikit-learn* [9] library to train a Random Forest model (c.f. 3.2) on the dataset we created. The model pipeline includes the following steps:

1. **Data Preprocessing:** Convert categorical features into numerical values using one-hot encoding and scale numerical feature values to a standard range.
2. **Model Training:** A Random Forest model is trained using GridSearch[3] which evaluates different combinations of hyperparameters using cross-validation to identify the best configuration.
3. **Model Testing:** Test the model on two totally unseen machines to evaluate its generalization capabilities.
4. **Feature Importance:** Find the importance of each feature in the model using the *Sklearn* permutation importances function[4] (c.f. 3.2 for details on permutation importance).

[3]GridSearchCV documentation
[4]sklearn permutation_importance documentation

Evaluation

5.1 Experimental Setup

The benchmark script has been run on 8 different machines, each with a different hardware configuration. No other ressources-intensive processes were running on the machines during the tests to ensure that the results are not affected by other workloads.

The machines were selected to cover a wide range of hardware specifications, the only requirement being that they have at least 8GB of RAM.

This setup resulted in a dataset of 224 samples.

Machine	**Arch.**	**Cores**	**Freq.**	**Year**	**MT/s**	**Mem.**	**SoC**
Macbook pro M1	arm64	8	3204	2020	4266	DDR4	Yes
Labostrex 119	x86_64	32	3200	2014	2133	DDR4	No
Thinkpad t14s gen 2	x86_64	8	3200	2020	3267	DDR4	No
Macbook pro 2019 (intel)	x86_64	16	4700	2019	2400	DDR4	No
Victus by HP 16d0900nz	x86_64	16	4600	2021	3200	DDR4	No
Macbook air M1	arm64	16	3200	2020	4266	DDR4	Yes
sacs003	x86_64	16	5000	2019	2666	DDR4	No
Balélec	x86_64	16	3400	2013	1600	DDR3	No

Table 5.1: Hardware specifications of the machines employed to construct the dataset (c.f. 4.4.2 for details on the features).

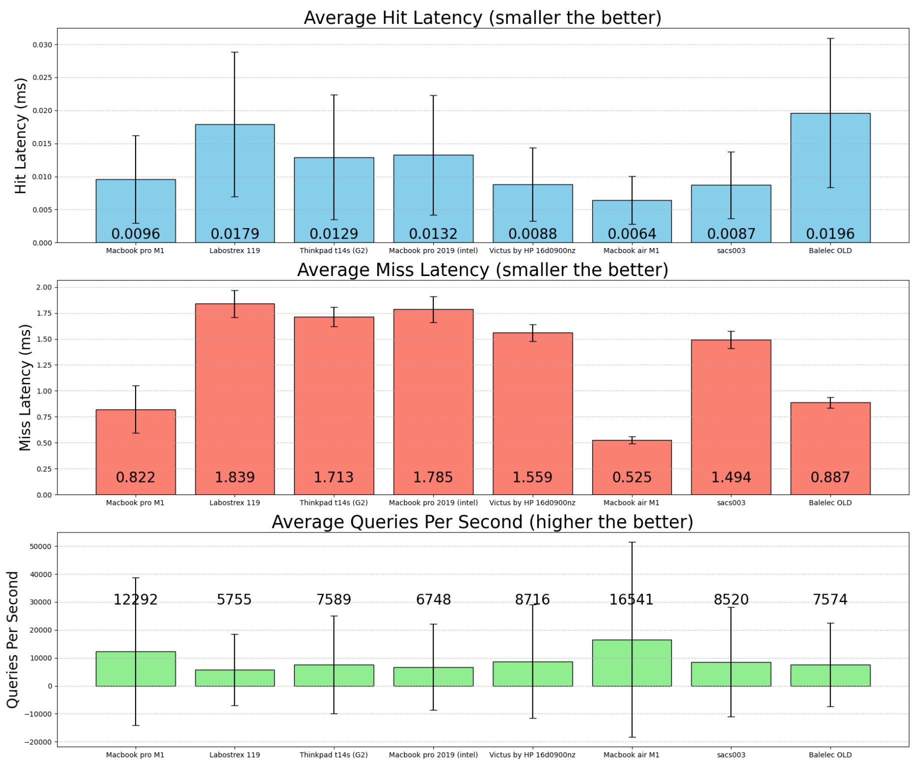


Figure 5.1: Performance comparison across all dataset's machines showing hit latency, miss latency, and queries per second. (Mean and standard deviation across all cache configurations)

5.2 Model results

GridSearchCV parameter grid and selected values

In order to optimize the results of the Random Forest Regressor, we performed a hyperparameter search using GridSearchCV. The table below summarizes the range of values tested for each parameter and the selected ones based on the best cross-validation score.

Parameter	**Tested Values**	**Selected Value**
`regressor bootstrap`	True, False	**False**
`regressor n estimators`	10, 50, 80, 100, 120, 200	**50**
`regressor max depth`	None, 5, 10, 20, 30	**None**
`regressor max features`	sqrt, log2, None	**sqrt**

Table 5.2: GridSearchCV parameter grid for the Random Forest regressor. Selected values are highlighted in bold.

- `regressor bootstrap`: If bootstrap samples are used when building trees (False = use the whole dataset for each tree).
- `regressor n estimators`: The number of trees in the forest.
- `regressor max depth`: The maximum depth of each tree (None = nodes are expanded until all leaves are pure).
- `regressor max features`: The number of features to consider when looking for the best split.

Results of the model evaluation on the test set

The model was evaluated using a test set representing 15% of the full dataset. The table below summarizes the performance metrics for each target variable.

Note that these results reflect the model's performance on data from the same machines used for training, not on entirely new machines. (See 5.3 for a test on computers that are not part of the training set)

Target	**MSE**	**MAE**	**MAPE (%)**	**R^2**
hit latency	$9.0525 \cdot 10^{-6}$	$1.70 \cdot 10^{-3}$	9.01	0.857
miss latency	$2.38 \cdot 10^{-2}$	$9.56 \cdot 10^{-2}$	8.24	0.904
queries/sec	$1.84 \cdot 10^{8}$	$4.76 \cdot 10^{3}$	17.27	0.781

Table 5.3: Performance metrics for each target on the test set.

5.2.1 Analysis

The results shows that the random forest model achieves good predictive performance on the test set, with MAPE values below 10% for both hit and miss latency, and below 20% for queries per second. The R^2 scores is above 0.87 for all targets which indicates that the model explains a large portion of the variance in the data.

5.3 Model Perfomance on unknown machine

The model was tested on two machines that were not part of the dataset. The model has no a-priori informations except the labels in the table below 5.4.

The first machine is a **Dell OptiPlex 7770 AIO** A desktop computer (the ones used in EPFL INF rooms), and the second one is a **Dell PowerEdge R440** server machine.[1]

Machine	**Arch.**	**Cores**	**Freq.**	**Year**	**MT/s**	**Mem.**	**SoC**
Dell OptiPlex 7770	x86_64	8	4700	2018	2666	DDR4	No
Dell PowerEdge R440	x86_64	8	3700	2017	2666	DDR4	No

Table 5.4: Hardware specifications of the two test machines.

Results

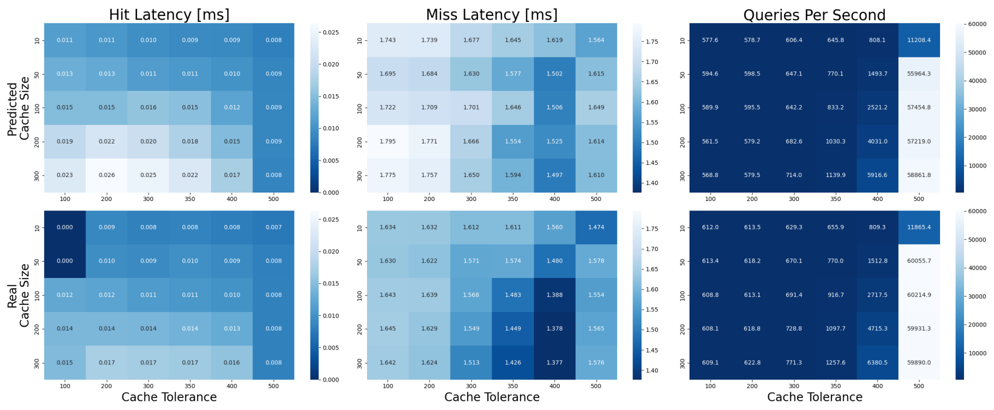


Figure 5.2: Heatmaps of the model predictions (top row) and the measured values (bottom row) for the Dell OptiPlex 7770.

[1]For readability reasons, only the plots for the Dell OptiPlex 7770 are showm here, the ones for the second machine are in the complementary material (8).

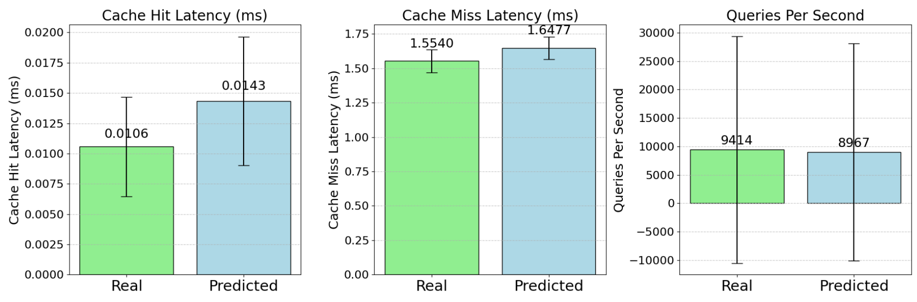


Figure 5.3: Mean and standard deviation of the model predictions (blue bars) and the measured values (green bars) for the Dell OptiPlex 7770.

5.3.1 Analysis

Using the model on two previously unseen machines (Dell OptiPlex 7770 and Dell PowerEdge R440), we see that the relative errors ($|\frac{\text{real value}-\text{predicted value}}{\text{real value}}| \cdot 100$) are all bellow 30% which is acceptable.

Machine	**Hit Latency (%)**	**Miss Latency (%)**	**QPS (%)**
OptiPlex	35.61	6.03	4.75
PowerEdge	23.11	26.74	27.47

Table 5.5: Relative error for the model on the two test machines.

The errors are higher for predictions on previously unseen machines than for the test set, which is expected since the test set consists of data from machines already present in the training data.

If we look at the results of the heatmaps (fig. 5.3), we observe that the overall patterns are similar between the predicted and measured cells. This suggests that the model has mostly captured the relationship between cache size, cache capacity and system performance.

5.4 Feature Importance

We used permutation importance to evaluate the impact of each feature on the model's predictions (c.f. 3.2 for details about permutation importance). This process allowed us to identify the features that contribut to the model predictions.

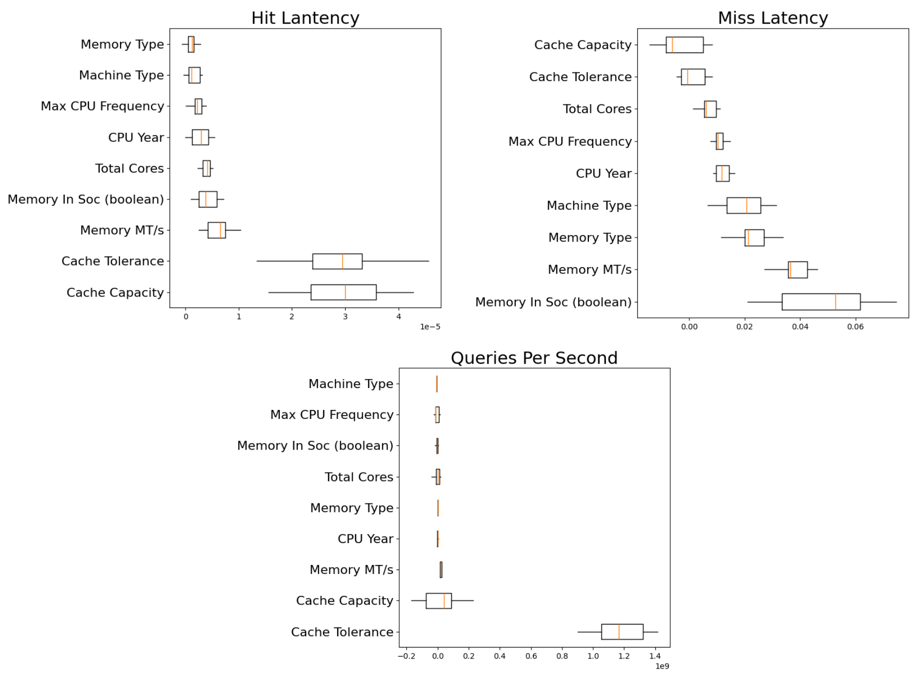


Figure 5.4: Permutation importances for the three performance metrics.

5.4.1 Analysis

Hit Latency

The hit latency plot shows that the most important features are the cache size and the cache tolerance, which makes sense given that the cache speed is mainly affected by the number of elements it has to seach through (the fewer elements there are, the faster it is) and the distance between the queried vector: more tolerant is the cache, higher is to probability to find a matching vector in the first values stored in the cache.

Other important features are the memory transfer speed (MT/s) and the boolean feature indicating whether the machine is a System on Chip (SoC) or not. It also makes sense as SoC machines have a faster memory access speed due to the fact that the RAM is integrated into the same chip as the CPU. Both of these features are related to the memory speed, wich suggests that the memory speed is the key factor in the cache's performance.

Miss Latency

The second plot shows that the most important features that affect the miss latency are all related to the memory(SoC, MT/s, and memory type). It also makes sense, since miss latency corresponds to the time required to retrieve a result from the indexes when the cache misses. As these indexes are stored in the memory, memory-related features (SoC, MT/s, and memory type) have the greatest impact. In contrast, cache capacity / tolerance are less relevant here because browsing the cache is way faster than searching in the indexes, even if the whole cache is iterated (and missses) before using indexes.

Queries per second

We see on the plot that the only feature which has a significant impact on the queries per second is the cache tolerance. It makes sense, since the greater the tolerance, the higher the probability is to find a matching vector inside and more queries it will handle witout having to use the indexes. When the cache tolerance is set to 500, the cache hit rate exceeds 99%. It means that the indexes are used less than 1% of the queries (c.f. 8.2). Given that using indexes is significantly slower, the number of queries per second grows exponentially with the cache tolerance (c.f. figure 5.3).

Observation

The permutation importance results indicate that, among hardware features, memory-related characteristics, such as memory MT/s and integration in the SoC are the primary factors influencing Proximity's performance.

When considering all features together, cache settings (size and tolerance) are the most influential factors. Therefore, the most efficient way to optimize Proximity's performance is to tune the hyperparameters to find a good tradeoff between precision and the need to perform expensive searches in the indexes to retrieve more precise results.

It is also important to note that searching in the indexes is approximately 50 times slower than searching in the cache (c.f. figure 5.1). This shows that, by considering the overall process execution time, hardware characteristics play a more critical role in index search performance than in cache performance.

Discussion

We faced a significant challenge due to the relatively small size of the dataset for a machine learning approache (225 samples and 8 machines) and the limited diversity of hardware. Almost all the machines where built between 2018 and 2021, which implies similar CPU and memory performance.

Another limitation is the lack of control over other background processes on the test machines. We took care to close all other applications but other processes may have introduced variability in the measurements.

It should also be noted that for the feasability of this projects, the of the system was scaled down compared to a typical usage of Proximity in a RAG system. As a result, the observed performance may not fully reflect real world deployments.

Additionally, the model is dependent on the indexes and dataset used. Variations in indexes construction, differences in vector distributions or distance in the dataset can influence retrieval times and, consequently, the model's predictions.

Conclusion

In this project, we demonstrated the feasibility of using a machine learning approach, specifically a random forest model, to predict the performance of Proximit, a vector data base cache, built to improve speed of RAG systems.

By training our model on data collected from eight different machines, we achieved reasonably accurate predictions with a relative error below 35% on unseen hardware for tree metrics (hit latency, miss latency, and query throughput) only using the machine's datasheet.

We also identified Proximity as a memory-bound process, indicating that improving the memory access speed could improve the performance of Proximity.

However, searching for vectors in the indexes is approximately 50 times slower than retrieving them from the cache. Therefore, tuning the cache settings (size and tolerance), by finding an optimal balance between accuracy and performance to minimize index searches, would have a greater impact on overall processing time than hardware optimization specifically designed for Proximity's performance.

Complementary Material

Project Repository

https://gitlab.epfl.ch/perego/bachelor-project

8.1 Second Test Machine

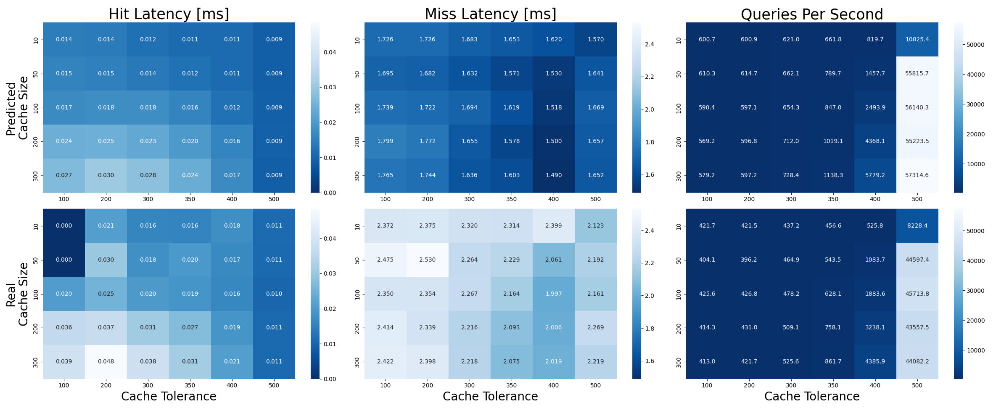


Figure 8.1: Heatmaps of the model predictions (top row) and the measured values (bottom row) for the PowerEdge 440.

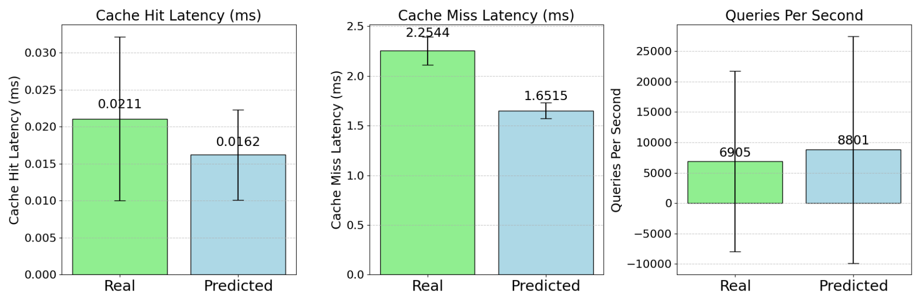


Figure 8.2: Mean and standard deviation of the model predictions (blue bars) and the measured values (green bars) for the PowerEdge 440.

8.2 Benchmark results for machines in the dataset

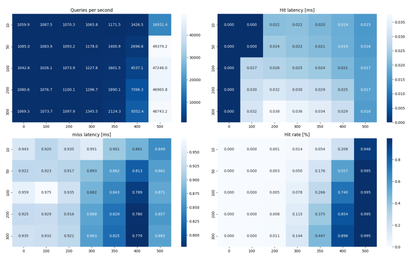


Figure 8.3: Benchmark results for Balelec OLD.

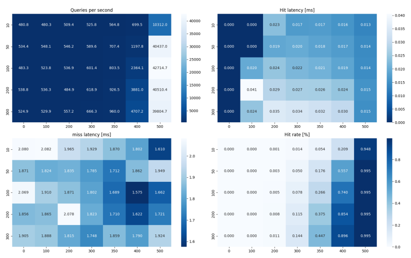


Figure 8.4: Benchmark results for Labostrex 119.

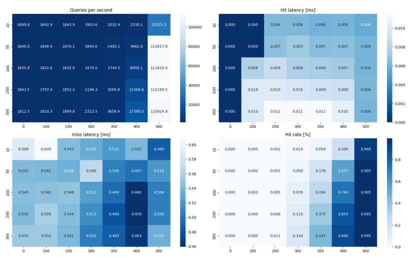


Figure 8.5: Benchmark results for Macbook air M1.

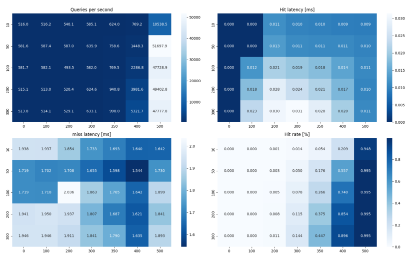


Figure 8.6: Benchmark results for Macbook pro 2019 (intel).

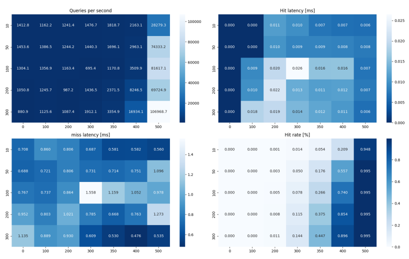


Figure 8.7: Benchmark results for Macbook pro M1.

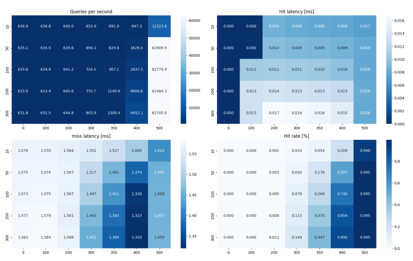


Figure 8.8: Benchmark results for sacs003.

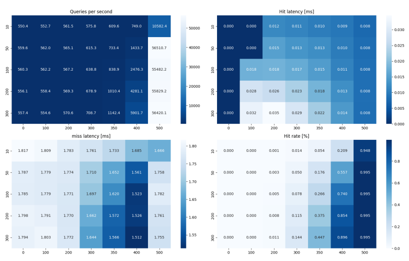


Figure 8.9: Benchmark results for Thinkpad t14s (G2).

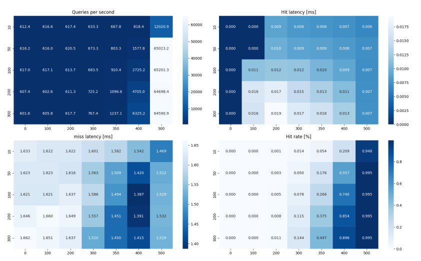


Figure 8.10: Benchmark results for Victus by HP 16d0900nz.

Bibliography

[1] Amazon Web Services. What is retrieval-augmented generation (rag)?, 2024. Accessed: 2025-06-03. URL: `https://aws.amazon.com/what-is/retrieval-augmented-generation/?nc1=h_ls`.

[2] Shai Aviram Bergman, Zhang Ji, Anne-Marie Kermarrec, Diana Petrescu, Rafael Pires, Mathis Randl, and Martijn de Vos. Leveraging approximate caching for faster retrieval-augmented generation. In *Proceedings of the 5th Workshop on Machine Learning and Systems*, EuroMLSys '25, page 66–73, New York, NY, USA, 2025. Association for Computing Machinery. `doi:10.1145/3721146.3721941`.

[3] Facebook. Faiss. `https://github.com/facebookresearch/faiss`, 2025.

[4] George Williams Harsha Vardhan Simhadri et al. Big ann benchmarks - neurips 2021. `https://big-ann-benchmarks.com/neurips21.html`, 2021. Accessed: 2025-05-19.

[5] IBM. What is random forest? Accessed: 2025-06-03. URL: `https://www.ibm.com/think/topics/random-forest`.

[6] LanceDB Contributors. Ivf-pq index — lancedb documentation, 2024. Accessed: 2025-06-03. URL: `https://lancedb.github.io/lancedb/concepts/index_ivfpq/`.

[7] konstin Messense et al. Maturin. `https://github.com/PyO3/maturin`, 2025.

[8] Oracle. What is a vector database?, 2024. Accessed: 2025-06-03. URL: `https://www.oracle.com/uk/database/vector-database/`.

[9] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David

Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011. URL: `https://jmlr.csail.mit.edu/papers/v12/pedregosa11a.html`.

[10] Scikit-learn. Permutation feature importance. Accessed: 2025-06-03. URL: `https://scikit-learn.org/stable/modules/permutation_importance.html`.

[11] Scikit-learn developers. *3.3. Model evaluation: quantifying the quality of predictions*. scikit-learn: Machine Learning in Python, 2024. Accessed: 2025-06-03. URL: `https://scikit-learn.org/stable/modules/model_evaluation.html#r2-score`.