Universidad Pontificia de Comillas

Master's Thesis

Máster en Big Data: Tecnología y Analítica Avanzada

Enhancing Supply Chain Data Harmonization through Large Language Models

Author: Fernando Arnal Escudero

Supervisor: Gonzalo Pablo España-Heredia Llanza

Madrid
June 2025

Fernando Arnal Escudero, declara bajo su responsabilidad, que el Proyecto con título **Enhancing Supply Chain Data Harmonization through Large Language Models** presentado en la ETS de Ingeniería (ICAI) de la Universidad Pontificia Comillas en el curso académico 2024/25 es de su autoría, original e inédito y no ha sido presentado con anterioridad a otros efectos. El Proyecto no es plagio de otro, ni total ni parcialmente y la información que ha sido tomada de otros documentos está debidamente referenciada.

Fdo.: [illegible] **Fecha:** 16 / 06 / 2025

Autoriza la entrega:

El Director del Proyecto

Gonzalo España-Heredia Llanza

Fdo.: [illegible] **Fecha:** 16 / 06 / 2025

V. B. del Coordinador de Proyectos

Carlos Morrás Ruiz-Falcó

Fdo.: .... **Fecha:** .... / .... / ....

Resumen

En los últimos años, el avance de la Inteligencia Artificial Generativa —y en particular, de los Modelos de Lenguaje de Gran Escala (LLMs)— ha transformado radicalmente la forma en que las organizaciones procesan y entienden el lenguaje natural. En este contexto de innovación tecnológica acelerada, uno de los retos persistentes en los entornos empresariales sigue siendo la calidad y coherencia de los datos maestros. Este aspecto está estrechamente relacionado con el proceso de armonización de datos, cuyo objetivo es garantizar la consistencia entre fuentes de información heterogéneas [1]. Este problema resulta especialmente crítico en sectores logísticos como el de la cadena de suministro, donde decisiones operativas y estratégicas dependen directamente de la fiabilidad de la información disponible.

Este Trabajo de Fin de Máster se enmarca en el equipo de Supply Chain Data & AI de Accenture y aborda precisamente este desafío, proponiendo una solución basada en LLMs para automatizar la identificación y corrección de errores en Master Data Tables (MDT), un elemento esencial en la metodología de trabajo de cualquier proyecto dentro del grupo. El objetivo de reducir la intervención manual, mejorar la escalabilidad de los procesos de validación de datos y aumentar la precisión en la toma de decisiones operativas.

A partir de un caso de uso representativo, se ha desarrollado una herramienta capaz de detectar registros duplicados o inconsistentes dentro de grandes volúmenes de datos estructurados, utilizando técnicas avanzadas de representación semántica, búsqueda por similitud y generación aumentada por recuperación (RAG). La solución propuesta combina la potencia de modelos como CANINE-C para la creación de embeddings, con un sistema orquestado mediante LangGraph que emplea agentes generadores y evaluadores basados en LLMs, coordinados a través de flujos de trabajo iterativos que permiten la autoevaluación del sistema. El proceso se completa con la integración de los resultados en bases de datos externas para su validación por expertos, cerrando así un ciclo completo de detección y trazabilidad de errores.

Por lo tanto, el desarrollo se apoya en una arquitectura monolítica dividida en tres etapas funcionales y se enfrenta a desafíos como la optimización del rendimiento computacional y la consistencia de las respuestas generadas por los modelos. Los resultados obtenidos confirman el potencial de los LLMs como herramienta eficaz y adaptable para mejorar la calidad del dato en entornos empresariales complejos, sentando las bases para futuras aplicaciones en proyectos de armonización de datos a gran escala.

Abstract

In recent years, the advancement of Generative Artificial Intelligence — and particularly of Large Language Models (LLMs) — has radically transformed the way organizations process and understand natural language. In this context of accelerated technological innovation, one of the persistent challenges in business environments remains the quality and consistency of master data. This is particularly relevant to the process of Data Harmonization, which seeks to ensure coherence across heterogeneous sources of information [1]. The issue becomes especially critical in logistics-related sectors such as supply chain management, where both operational and strategic decisions depend directly on the reliability of the available information.

This Master's Thesis is carried out within the Supply Chain Data & AI team at Accenture and directly addresses this challenge by proposing a solution based on LLMs to automate the identification and correction of errors in Master Data Tables (MDT), a core element in the working methodology of any project within the group. The main objective is to reduce manual intervention, improve the scalability of data validation processes, and increase accuracy in operational decision-making.

Based on a representative use case, a tool has been developed that can detect duplicated or inconsistent records within large volumes of structured data, using advanced techniques such as semantic representation, similarity search, and Retrieval-Augmented Generation (RAG). The proposed solution combines the power of models like CANINE-C for embedding generation with a system orchestrated using LangGraph, which leverages generator and evaluator agents based on LLMs, coordinated through iterative workflows that enable self-evaluation of the system. The process concludes with the integration of the results into external databases for expert validation, thereby completing a full cycle of error detection and traceability.

The development is supported by a monolithic architecture divided into three functional stages and addresses key challenges such as the optimization of computational performance and the consistency of the model outputs. The results obtained confirm the potential of LLMs as an effective and adaptable tool to improve data quality in complex business environments, laying the groundwork for future applications in large-scale data harmonization projects.

Acknowledgements

The development of this project would not have been possible without the trust placed in me by the Supply Chain Data & AI team at Accenture. I would especially like to thank Gonzalo España-Heredia and Rubén Balbastre, who guided me throughout this journey and from whom I've learned—far beyond the technical realm—what dedication, effort, and excellence truly mean.

Of course, I cannot miss the opportunity to express my deepest gratitude to my family, who have supported me unconditionally and to whom I owe every achievement. I also want to thank my friends, who I'm fortunate to count on every day. I am proud of their successes, and it brings me joy to know they celebrate mine.

Above all, I would like to thank my sister Cristina, someone I think about every day and who is a true example of perseverance for me. Whenever I struggle, feel afraid, or hesitate to face a challenge, thinking of her helps me push forward and become the version of myself I want her brother to be.

Contents

List of Figures

List of Tables

Acronyms

AI	Artificial Intelligence.
BERT	Bidi-rectional Encoder Representations from Transformer.
CEO	Chief Executive Officer.
CFO	Chief Financial Officer.
COO	Chief Operations Officer.
DBSCAN	Density-Based Spatial Clustering of Applications with Noise.
ERP	Enterprise Resource Planning.
FAISS	Facebook AI Similarity Search.
GenAI	Generative Artificial Intelligence.
GIL	Global Interpreter Lock.
GPT	Generative Pre-trained Transformer.
ISCP	Intelligent Asset Management Services.
LLM	Large Language Model.
MDT	Master Data Table.
NLP	Natural Language Processing.
PCA	Principal Component Analysis.
PK	Primary Key.
RAG	Retrieval-Augmented Generation.
WCSS	Within-cluster Sum of Squared Distances.

Chapter 1

Introduction

In recent years, the advancement of Generative Artificial Intelligence (GenAI), and in particular of Large Language Models (LLMs), has opened new possibilities in the processing and understanding of natural language. These technologies make it possible to automate complex tasks such as information extraction, pattern detection, and text generation, with a level of semantic understanding that was previously unattainable. In this context, the present work explores how to leverage the capabilities of LLMs to address a common problem in data management within supply chains: the identification and correction of errors in Master Data Tables (MDTs). By applying techniques based on GenAI, the goal is to reduce manual intervention, improve the scalability of the solutions, and increase the accuracy in handling critical data used for business decision-making.

This study is applied within a real-world business context, specifically in the *Supply Chain Data & AI* team at Accenture, where data quality plays a crucial role in ensuring operational excellence. The proposed approach involves designing a workflow powered by LLMs to detect anomalies in structured data tables, with a focus on minimizing human intervention. The expected contribution is to demonstrate the viability of using LLMs as a scalable, accurate, and flexible tool for data harmonization in supply chain environments.

1.1 Motivation

With the rise of automation techniques based on artificial intelligence, many routine and repetitive tasks that were once well-established across various businesses, companies, and work environments are now being replaced. This is not just a personal perspective; according to a recent report by the World Economic Forum [4], more than 88% of C-suite executives (Chief Executive Officer (CEO), Chief Financial Officer (CFO), Chief Operations Officer (COO), etc.) believe that the imminent implementation of Artificial Intelligence (AI) in the workplace is imperative. They argue that these new technologies will foster more creative and innovative teams, allowing them to move beyond stagnant methodologies.
Despite having limited professional experience and only a basic understanding of the subject when I began this project, I viewed it as a valuable opportunity for personal and academic growth. Although I initially lacked deep knowledge about LLMs and the emerging use of GenAI in business environments, I quickly recognized the relevance of developing a solution based on these technologies to address a problem marked by its manual, monotonous, and non-scalable nature.

1.2 Project's Objectives

The development objectives addressed in this Master's Thesis are as follows:

a) **Identification and traceability of potential duplicates:** To identify and store, in an external database, the identifiers of records flagged as potential duplicates. This allows for manual review and subsequent analysis and cleaning. The external database will support change traceability and enable human oversight in cases where ambiguities or inconsistencies arise that automated models cannot confidently resolve.

b) **Evaluation of detection techniques:** To study which techniques, methods, or tools are most effective for extracting similar records or potential duplicates within large datasets. Both statistical and machine learning-based approaches will be analyzed, including similarity search, clustering, and dimensionality reduction methods. The suitability of each technique will be assessed based on the type and structure of the data, as well as on performance criteria such as accuracy, coverage, and computational cost.

c) **Implementation of LLMs through prompt design:** To explore the different possibilities for applying LLMs in data review and extraction tasks. This includes designing and implementing precise prompts to guide the models effectively. Various prompt engineering strategies will be studied to adapt LLMs to specific tasks such as duplicate detection, normalization, and entity extraction.

d) **Impact assessment of LLMs in data quality workflows:** To evaluate the impact of using LLMs on the quality and efficiency of the data cleaning process, comparing their performance with traditional methods. A set of evaluation metrics will be defined to assess not only the reduction of duplicates but also semantic coherence and the minimization of manual errors. This analysis will also highlight current limitations of LLMs in this context and propose potential improvements or hybrid strategies.

1.3 Resources Used

In order to fulfill the defined objectives and carry out the project efficiently, the resources employed can be categorized as follows:

- **Human support:** The guidance provided by both my internship tutor and project supervisor is a critical resource. Their experience in code review, project structuring, and validation of each stage has played a key role in ensuring successful project development.

- **Hardware:** The development is carried out on a company-provided laptop with 16GB of RAM, a detail worth noting due to its potential impact on certain computational tasks. The project is conducted in parallel with an internal initiative at Accenture. Naturally, the setup also includes essential peripherals such as a charger and a mouse.

- **Software and tools:**

  - An **IDE (Integrated Development Environment)** is used to write, edit, and run code. Throughout the project, both Visual Studio Code and PyCharm have been utilized. The core of the implementation is based on Python 3.19, along with several libraries introduced throughout the thesis.
  - A **code repository**, used for version control and collaboration. While GitHub is commonly used in academic projects, this project employs BitBucket—an enterprise Git solution integrated with Jira, a project management tool aligned with the Scrum methodology.
  - A brief mention is due for **DBeaver**, the database client used as the testing environment for managing and visualizing database content.

- **Other supporting resources:** Additional support is provided through technical documentation, specialized online forums, and collaboration tools such as Microsoft Teams. Although these resources are less tangible, they are essential for enabling agile communication, resolving technical challenges, and adhering to best practices in software development.

1.4 Methodology

With the project objectives and available resources already defined, this section outlines the methodology followed to address the problem and develop the project in a structured and professional manner. The methodology is aligned with the working environment and expectations established by Accenture, combining technical rigor with practical applicability.

The project adopts an experimental methodology, based on the iterative design, implementation, and validation of an end-to-end system. The approach integrates practices such as data analysis, prompt engineering, and quantitative evaluation, aiming to build a replicable and measurable solution. The development is structured in the following stages:

- **Problem familiarization and system design:** The initial phase focuses on understanding the problem, adapting to the team environment, and outlining a high-level system architecture. This stage involves collaboration with supervisors to define the development goals and constraints.

- **Research and preliminary implementation:** Once the system architecture is outlined, the next step involves researching tools, algorithms, and techniques that could support the project's objectives. Selected methods are tested through early-stage implementations to assess their feasibility and effectiveness.

- **Similarity detection development:** A core part of the project involves developing mechanisms for detecting similar records in large datasets. This is achieved through vector-based techniques, supported by dimensionality reduction and clustering strategies when needed.

- **Integration of LLMs):** The system is then extended by incorporating LLMs to support data validation and enhancement. This includes designing and iterating over prompt-based instructions, selecting appropriate libraries, and optimizing the pipeline for task-specific applications.

- **System integration and evaluation:** The final stage consists of connecting the system output to an external database, organizing the codebase according to internal quality standards, and defining evaluation metrics. The goal is to assess not only technical performance (e.g., duplicate reduction, semantic consistency) but also the system's maintainability and scalability.

Chapter 2

State of the Art

To understand the problem and the potential solutions to be implemented, it is necessary to understand its context.

As mentioned in the introduction, the work presented here is part of the Supply Chain Data & AI team within the Strategy & Consulting department at Accenture. As its name suggests, this department is responsible for studying, managing, optimizing, and making decisions regarding clients' supply chains, with a strong focus on data analysis and usage.

In each client's supply chain, the proposed solutions rely on what is referred to as "the network": a complete digital representation (digital twin) of their operations—from suppliers to customers—including factories, distributors, and other operational locations.

This network is constructed using MDTs and transactional data from systems such as SAP or SIEVO. The MDTs contain key information about entities within the supply chain: materials, locations, specific characteristics of materials per location, customers, and more.
The accuracy of this digital representation directly depends on the quality of the data used. Poorly labeled or duplicated data can lead to errors that negatively affect decision-making.
For instance, in supplier master data, it is common to find duplicate entries. This may be due to the existence of multiple Enterprise Resource Planning (ERP) systems, manual updates introducing redundancies, or the consolidation of information from different business units across countries dealing with the same supplier. Even when the Primary Keys (PKs) are unique, the attributes may indicate that the same supplier has been registered multiple times.

Such inconsistencies can lead to serious consequences. If a supplier is duplicated under different PKs, it might appear that certain materials have multiple sources (multi-sourced), when in fact they depend on a single supplier (single-sourced). This false perception could mask critical risks to the supply chain's resilience by underestimating the vulnerability to disruptions from that supplier.

Given the importance of this issue, the team developed a series of measures to correct potential errors, initially based on manual data review or the application of basic Natural Language Processing (NLP) techniques. However, these approaches are often manual, inefficient, and, most importantly, not scalable across different clients. As a result, each time a new client is onboarded, the team must invest additional resources to ensure data quality.

This project aims to explore and implement novel techniques for error and duplicate detection using LLMs, thereby saving valuable time and resources for both the company and its clients. Moreover, it seeks to contribute to the broader field of artificial intelligence and its expanding applications.
Before the development of this project, the problem had been addressed using different approaches:

- Initially, the team was unaware of the severity of the issue and relied on simple manual data reviews. This method led to frequent forecasting errors, prompting the development of more advanced solutions.
- The second approach involved more sophisticated language processing techniques, such as text mining and regular expressions (regex). However, these methods were highly complex and tailored to specific types of master data, limiting their scalability.

With the advancement of LLMs, the proposed solution not only surpasses previous methods but also significantly improves the team's efficiency in managing multiple projects concurrently.

Chapter 3

Theoretical Framework

3.1 What is an LLM / foundation model?

Large Language Models (LLMs), also known as foundation models, are advanced deep learning architectures trained on massive datasets to understand and generate human-like language [5]. They are built upon the transformer architecture introduced by Vaswani et al. in 2017, which uses self-attention mechanisms to capture long-range dependencies and contextual information in text [6].
These models are typically pre-trained on vast bodies of text, encompassing sources such as books, websites, and academic articles. Pre-training involves predicting masked or next tokens, allowing the model to learn the statistical patterns and semantic relationships present in natural language. Once pre-trained, these models can be fine-tuned on specific tasks (like text classification, question answering, or summarization) or used in zero-shot and few-shot settings by leveraging prompt engineering techniques.

The term *foundation model* was popularized by researchers at Stanford, who emphasized that these models serve as a versatile basis for a wide range of downstream applications [7]. Foundation models exhibit emergent capabilities, such as in-context learning and multilingual understanding, that make them adaptable to diverse domains.

LLMs, including models like Generative Pre-trained Transformer (GPT), Bidi-rectional Encoder Representations from Transformer (BERT), and T5 (Text-to-Text Transfer Transformer), have demonstrated state-of-the-art performance across numerous NLP benchmarks [8]. Their size — often measured in billions of parameters — allows them to capture intricate patterns in language, but also raises challenges related to computational cost and environmental impact, which has been marked popularly as one of the main issues to be solved.

Moreover, these models have become the backbone of Retrieval-Augmented Generation (RAG) systems, chatbots, and other generative AI applications. Despite their remarkable capabilities, LLMs are not without limitations. They can sometimes generate plausible but incorrect information, known as hallucinations, and they may encode biases present in their training data.
To mitigate these challenges, research continues on topics such as model alignment, interpretability, and efficient fine-tuning strategies [9].

Overall, LLMs represent a significant breakthrough in AI, opening avenues for improved human-computer interaction, knowledge discovery, and content generation. Having mentioned it, let's see what a RAG is.

3.2 Embedding

Embeddings are dense numerical representations of data elements (e.g., words, phrases, or documents) in a continuous vector space of low or moderate dimensionality [10]. Each word or text is assigned a real-valued vector, whose relative position captures semantic and contextual relationships. For example, in a vector space, similar words are located close to each other (e.g., "football" and "soccer" vectors will have similar close distances). This allows NLP models to process language more effectively. Furthermore, by reducing the dimensionality of the data, we improve the computational efficiency of the agents. However, this is not always the case, as we have encountered instances where the dimensionality of the embeddings was higher than that of the original data — though these cases are exceptions.
Some of the most common techniques related to create embeddings are:

- **Word2Vec**: Developed in 2013, Google's neural network model learns word vectors from a large corpus of text. This model includes two main architectures: Continuous Bag of Words (CBOW), which predicts a target word based on its context; and Skip-gram, which predicts the surrounding context given a single word. Although it is effective, its embeddings are static, meaning that each word has only one fixed vector, with no distinction between its possible acceptations or meanings. [11].

- **FastText**: This algorithm expands Word2Vec by taking n-grams into consideration. Instead of learning a vector for each whole word, FastText learns vectors for subword units or fragments. This allows for more flexible handling of rare words and better performance with morphological variations [12] [13].

- **Contextual Embeddings**: Some examples like BERT, ELMo or GPT, are models based in the 'transformer' architecture, which generate vectors from the context of a word [14]. Hugging Face has popularized the use of these models to obtain full-text embeddings. More recent techniques (such as Sentence-BERT) fine-tune BERT to generate sentence- or paragraph-level vectors, which are useful for semantic similarity comparison.

Each technique has its advantages: static embeddings like Word2Vec or GloVe are efficient and capture general relationships, while contextual embeddings based on transformers (BERT, GPT, etc.) provide richer representations that take the full context of the text into account. The choice between one or the other depends on the application: in general, transformer-based models have significantly improved performance in many modern NLP tasks. Later in the implementation, we will discuss which embedder suits the best for our purpose.

3.3 Vector Stores

A vector store (or vector database) is a specialized system designed for storing and indexing high-dimensional embeddings, enabling efficient similarity search over unstructured data such as text, images, or audio [15, 16]. These systems support operations like adding, querying, and deleting vectors, often accompanied by metadata filtering and horizontal scaling capabilities [16]. They implement optimized nearest-neighbor search algorithms—like HNSW, IVF, or product quantization—to rapidly retrieve the most semantically similar vectors based on distance metrics (e.g., Euclidean or cosine) [15]. In modern AI applications, including RAG, vector stores serve a crucial role by bridging embeddings from language models to downstream retrieval tasks with low latency and high throughput.

3.4 RAG

Retrieval-Augmented Generation (RAG) is a technique that combines the generative capabilities of LLMs with the precision of external information retrieval. Instead of relying solely on the pre-trained knowledge of LLMs, a RAG system dynamically retrieves relevant content from external sources — such as document collections or knowledge bases — and uses it to improve the accuracy and relevance of the generated outputs [17].

The typical workflow of a RAG model involves two main stages. First, a retriever component uses similarity search techniques to identify the most relevant documents or passages based on a given query or prompt. Then, these retrieved documents are provided as context to the generative model, which produces a final output that is better grounded in up-to-date or domain-specific knowledge [18]. This approach improves the factual accuracy and relevance of generated responses, addressing limitations of purely generative models. Moreover, RAG systems enable dynamic adaptation to new information without requiring expensive retraining of the entire language model.

Facebook AI Similarity Search (FAISS): FAISS is one of the main libraries used in this project. Developed by Meta (previously known as Facebook) to perform efficient similarity searches and clustering of dense vectors. It is widely used in applications where vector embeddings represent data, such as text, images, or audio. FAISS provides a suite of algorithms and data structures that enable fast nearest-neighbor search, even for very large datasets.
According to its official documentation [19], FAISS supports both exact and approximate search methods, allowing users to balance between accuracy and speed depending on their needs. It is optimized for high-dimensional data and offers support for CPU and GPU acceleration, making it suitable for large-scale machine learning and AI pipelines.
FAISS's modular design allows researchers and engineers to choose between different index types and quantization strategies, tailoring the search process to specific applications and data characteristics. This flexibility has made it a key component in modern RAG systems and other AI-driven information retrieval tasks.

3.5 Dimensionality Reduction Methods

When working with huge amounts of data, and more precisely, vectors, it is common to use dimensionality reduction methods. This process is characterized for reducing the number of input variables (features) in a dataset while preserving as much important information as possible [20]. In our case, we have used several methods to optimize the execution of our processes, as we will see later. Let's briefly define some of these methods:

3.5.1 Principal Component Analysis (PCA)

PCA is a statistical technique used to reduce the dimensionality of datasets while preserving as much variability as possible. It transforms a set of possibly correlated variables into a set of linearly uncorrelated variables called principal components. This transformation is achieved through an orthogonal linear transformation, ensuring that the first principal component accounts for the largest possible variance in the data, with each succeeding component accounting for the remaining variance under the constraint of being orthogonal to the preceding components [21].

PCA is widely applied in fields such as pattern recognition and computer vision, notably in face recognition systems. In such applications, PCA helps in extracting the most significant features from facial images, facilitating efficient and accurate recognition. The method involves computing the eigenvectors and eigenvalues of the covariance matrix of the dataset, which are then used to project the original data into a lower-dimensional space. This projection retains the most critical information, enabling effective data analysis and interpretation [21].

The following image represents a small example on how does PCA work in a 2 dimension (two variables) dataset.

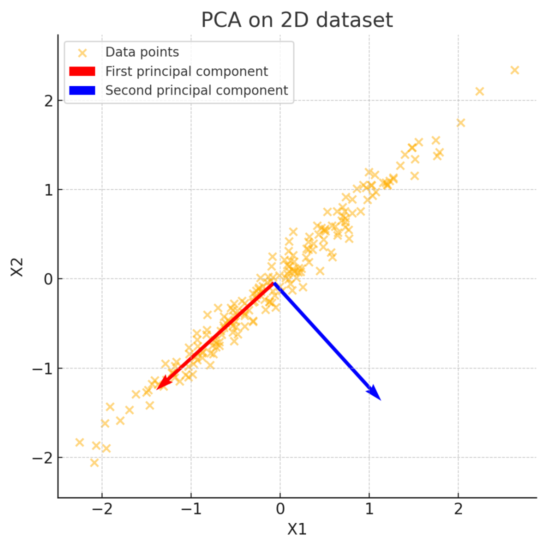


Figure 3.1: Example of principal components and data variance in a small dataset

3.5.2 Clustering

Clustering is an unsupervised machine learning technique that groups elements based on similarity, without relying on predefined labels. As IBM Research highlights [22], clustering enables the discovery of hidden structures within data by creating groups (clusters) of similar records. There are several clustering algorithms, including:

- **Non-Hierarchical Clustering (K-Means)**: This algorithm assigns data points to k clusters by minimizing the variance within each cluster.
- **Hierarchical Clustering**: This approach builds a tree-like structure of clusters, either through agglomerative (bottom-up) or divisive (top-down) processes.
- **Centroid-Based Clustering**: Algorithms like K-Means are examples of centroid-based clustering, where each cluster is represented by the center (centroid) of its data points.
- **Distribution-Based Clustering**: These methods assume data is generated from a mixture of distributions, typically Gaussian distributions, and group data based on this assumption.

In our study, when identifying the most similar records or those with the highest probability of being duplicates, we considered not only similarity search by score, but also the use of clustering algorithms. Specifically, clustering provides a complementary approach to similarity search by grouping data points that exhibit similar characteristics, improving the detection of potential duplicates or errors.

Density-Based Spatial Clustering of Applications with Noise (DBSCAN): A commonly used algorithm in density-based clustering is DBSCAN. It relies on two key hyperparameters:

- **Minimum Points**: This represents the minimum size of each cluster, essentially defining the core of a cluster. Since our goal is to detect potential data errors, we consider a minimum size of two elements.
- **Epsilon Value**: This is the maximum distance between two points to be considered as neighbors belonging to the same cluster.

To optimize the value of epsilon, we can employ HDBSCAN (Hierarchical Density-Based Spatial Clustering of Applications with Noise). HDBSCAN performs DBSCAN over varying epsilon values and integrates the results to identify the clustering that provides the best stability across epsilon values [23] [24].

KMEANS: Usually called unsupervised feature learning, K-Means is a classical and widely used unsupervised clustering algorithm that partitions a dataset into k distinct clusters[25].
It operates iteratively by assigning each data point to the nearest cluster centroid and then recalculating the centroids based on the current assignments. The objective is to minimize the Within-cluster Sum of Squared Distances (WCSS) between data points and their respective centroids. Despite its simplicity and efficiency, K-Means has limitations, such as sensitivity to the initial choice of centroids and the need to predefine the number of clusters k. In the Results Chapter, we will discuss and dive deeper into how we can implement this kind of clustering, and choose between algorithms.

Overall, clustering provides a powerful tool to organize unstructured data into meaningful groups, playing a vital role in applications such as anomaly detection, record linkage, and data cleaning.

3.6 Other Techniques used in the project

Multiprocessing

The `multiprocessing` module in Python provides support for concurrent execution using multiple processes. Unlike multithreading, which operates within a single process and is constrained by the Global Interpreter Lock (GIL), `multiprocessing` bypasses the GIL by creating separate memory spaces for each process, enabling true parallelism on multi-core systems. This makes it especially useful for CPU-bound tasks that require heavy computation, which will be a essential feature, as we will see in the Implementation chapter.

The module offers a similar interface to the `threading` module, making it accessible for developers familiar with multithreaded programming. It supports process creation, synchronization primitives, shared memory, and inter-process communication via queues and pipes [26].

Multithreading

On the other hand we can find Multithreading, which is the way of achieving multitasking by dividing a task into threads. A thread of execution is the smallest sequence of programmed instructions that can be managed independently by a scheduler, which is typically a part of the operating system [27]. In many cases, a thread is a component of a process. As we mentioned, multithreading also has a native Python module. However, after some research, we discovered that the FAISS library also includes a multithreading module. This allowed us to centralize several utilities we were looking for within a single library, contributing to clearer and more efficient code. Later on, we will present the implementation in detail and discuss the reasoning behind choosing one technique over the other.

When to Use Multithreading vs. Multiprocessing

The choice between multithreading and multiprocessing depends largely on the nature of the task. Multithreading is typically used for I/O-bound operations, such as reading files, handling network requests, or waiting for user input. These tasks spend a significant amount of time idle or waiting, which allows multiple threads to operate efficiently within a single process, despite the GIL in Python. On the other hand, multiprocessing is better suited for CPU-bound tasks that require heavy computation. Since each process has its own Python interpreter and memory space, multiprocessing bypasses the GIL and can take full advantage of multiple CPU cores. This makes it ideal for parallelizing tasks like data processing, mathematical simulations, or machine learning model training.

3.7 Hugging Face

Hugging Face is a company and open-source community that has become a central hub for modern NLP and machine learning tools. It is best known for the numerous `Transformers` libraries, which provides pre-trained models for a wide range of tasks, including text classification, question answering, summarization, and translation [28].

Hugging Face simplifies the use of powerful models such as BERT, GPT, RoBERTa, and T5, making them accessible through intuitive APIs and a standardized interface. The platform also includes tools like datasets for standardized and shareable datasets, and tokenizers for efficient text preprocessing.

In addition to code libraries, Hugging Face offers a **Model Hub** — a collaborative repository of thousands of pre-trained models contributed by the research and developer community. This ecosystem promotes reproducibility, transparency, and rapid experimentation in NLP and beyond, helping us in a great way on the task of finding the right embedding model.

By lowering the entry barrier to using state-of-the-art models, Hugging Face has played a key role in democratizing AI development and accelerating research in natural language understanding and generation.

3.8 Software Architecture

Having seen most of the techniques that we are about to use, let's analyze the code architecture from a high-level perspective before diving into the detail, and search for these techniques implementations.

Since the beginning, we adopted a monolithic design, despite being aware of its limitations compared to a microservices-based architecture [29]. The architecture design was constrained by a set of library choices and software versions defined by an internal guideline within the Accenture team, known as Intelligent Asset Management Services (ISCP). This framework acts as a common ground across projects, enabling the sharing of tools and methods, and ensuring the performance and maintainability of all code developed within this environment [30].

The resulting architecture is composed of three sequential stages, each governed by a distinct philosophy and purpose. Now we will briefly outline the main principles and goals that define them.:

a) **Data Rearrange and Similarity Extraction:** This first step aims to reduce the number of rows or records that the LLMs must compare. Although it might seem irrelevant at first glance, the performance of the agents is directly related to the number of records they must analyze, as we mentioned earlier. We achieve this reduction by working within a vector space, which is built from the original data and an embedding. Then, for each record, we simply retrieve the desired number of most similar vectors. In this implementation, we specifically aim to retrieve the 10 most similar vectors for each record, retrieving, in the end, a list formed by 10-element lists, being these elements the identification of the record or row, and its distance with the original row. This process of searching for the most similar vectors is known as "Similarity Search".

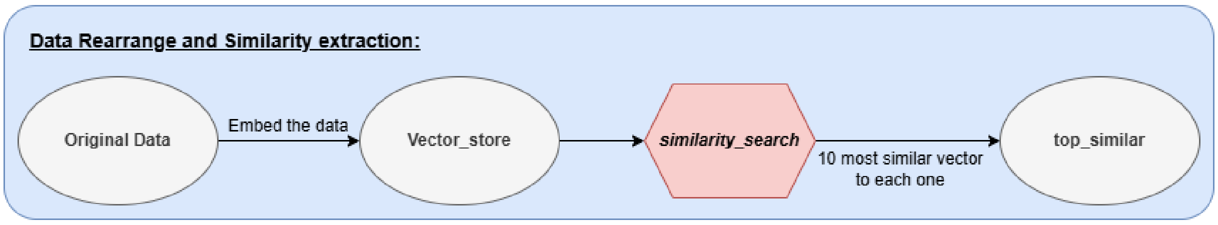

Data Rearrange and Similarity extraction:
Original Data
Embed the data
Vector_store
similarity_search
10 most similar vector to each one
top_similar

Figure 3.2: Diagram illustrating the architecture of the first step

b) **Output from duplicates detection:** The second and main step represents the core of the project. In this stage, we aim to send blocks of text to a LangChain workflow. This workflow attempts to detect duplicates or similarities between records and evaluate the resulting output. The output obtained is a list of two possible elements, an array made up of the ids very possible duplicated rows, or the string 'None'. It is imperative that the prompts and the workflow work correctly so the output structure does not vary from one iteration to another. This is the step where we believe the most innovative advances are being made, as we are exploring new ways of interacting with LLMs. This involves studying parameters such as temperature, the prompts used, and how the information is structured and provided.

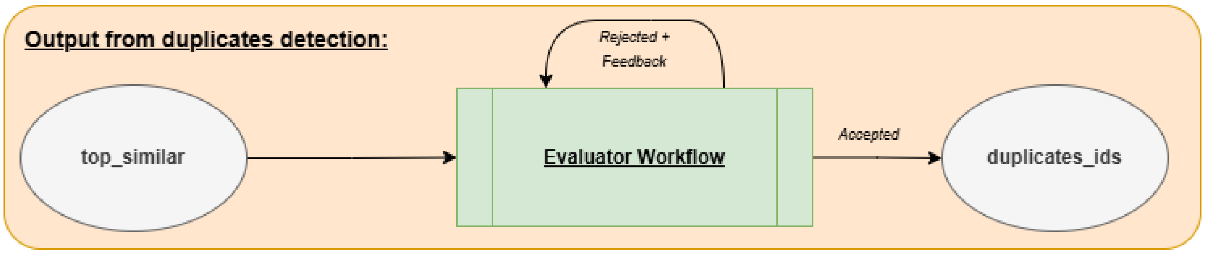


Figure 3.3: Diagram illustrating the architecture of the duplicate detection by LLMs step

c) **Results preparation for visualization:** This final step is the least critical, but not necessarily the easiest. After retrieving the list of IDs corresponding to the duplicated rows, we need to publish them to an external database so that the user can manually review any potential errors present in the original data. To do this, we must first link the retrieved IDs to their corresponding records. Then, we simply need to create a table, define its columns, and write the gathered information into it.

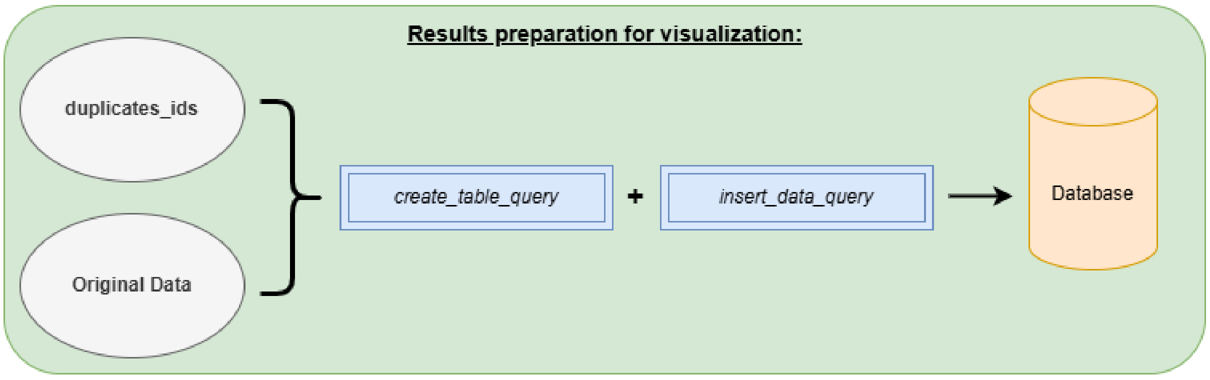


Figure 3.4: Diagram illustrating the architecture of the retrieving data step

Chapter 4

Implementation

In this chapter, which constitutes the core of the thesis, we present all the details regarding the application and its successful implementation. We begin by analyzing the defined architecture, outlining its three main components, their respective purposes, and a high-level overview of their interactions.

Subsequently, we will introduce the data used, providing a brief description and the rationale behind its selection.
Once already understood the general structure of the tool in the previous chapter, we delve into each of the three components in detail, reviewing the corresponding code and the modifications it has undergone throughout development. Finally, we discuss the main challenges encountered during implementation and the strategies adopted to address them.

There are numerous possible implementations tailored to a wide variety of problems and business needs [1], but this implementation can serve to numerous situations. We believe this approach offers a clear and concise path for understanding the tool in a structured and accessible manner.

4.1 Input Data

The main dataset used for experimentation, development, and analysis in the *Results* chapter is a synthetic dataset that replicates the typical structure of a MDT related to company suppliers. This dataset was provided by Accenture and does not contain any real client information.
The following table presents a sample of the data. It consists of eight columns:

- `LIFNR`: which represents the position or identifier of the supplier in the data table.
- `NAME1`: the name of the supplier company.
- `LAND1`: a short code representing the country of the supplier.
- `ORT01`: a more explicit location detail, which may be a name or a code.
- `PSTLZ`: the postal code.
- `STRAS`: the street address of the supplier.
- `TELF1`: the telephone number of the supplier.

- `REGIO`: code of the region where the supplier is set

We must remember that the values are fictional, so the data does not exist, and the name of the locations might not exist, and so the region and the telephone numbers.
Having defined the data columns, lets see a small representation of the data:

LIFNR	NAME1	LAND1	ORT01	PSTLZ	STRAS	TELF1	REGIO
1000026	Walls Group	ZV	Y7Q0I	51330	720 Valencia Falls Apt. 838	438-655-0800x09277	WY
1000671	Sheppard-Garcia	CY	Warnermouth	98108	2982 Perez Forks	001-684-563-4845x158	VA
1000003	Herrera Inc	IQ	Mooreville	29678	80384 Greene Forge Suite 468	001-404-267-3992x964	NJ
1000013	Walls Group	ZW	Y7Q0I	51330	720 Valencia Falls Apt. 838	438-655-0800x09277	WYO
1000001	Williams-RoberM	CL	New Ryan	93050	177 David Ma5	(322)137-6078x77087	MT
1000008	Hicks PLC	UZ	South Wesley	83853	8247 Kristen Park	345.357.1579x617	KY
1000015	Zhang PLC	IS	Haydenstad	14090	193 Howell Grove	444-743-0952	KY

Table 4.1: Example of input data used in the simulation.

With a closer inspection, we can identify two very similar rows: both `1000026` and `1000001` share many fields in common. These examples illustrate precisely the kind of cases this project aims to address. When searching for potential duplicates, we focus on records where the information is nearly identical but may have been overlooked due to small differences—such as in this case, where the values differ in the `LAND1` and `REGIO` fields. These types of errors originate from the automatic transfer of data into the MDT. In this process, we can assume that data sharing the same identifier (in this case, `LIFNR`) should be consolidated into a single record or row. However, if the associated information differs, two entries that should represent the same entity may end up generating two separate rows.
Other cases, which we will examine later, may involve misspelled words or minor grammatical differences, although these are less frequent.

Now that we understand how the input data is structured, we can begin the process. Before proceeding, it is important to note that this implementation is specifically designed for this type of input, which is appropriate given that the tool is focused on Supply Chain methodology.

4.2 Data Reduction Step

4.2.1 Process Data

Once the input data is available, it must go through several preprocessing steps to reach an optimal working state.

We begin by removing the first row of the table, as the column names are not needed at this stage. Then, we create a list in which each element is a string representing an entire row. This list will be useful not only for obtaining the vectors but also as the input passed to the LLM workflow.
As a small enhancement we discovered, we separate the fields within each row using a pipe symbol (`|`). This change improves the LLM's ability to distinguish between fields, helping the agents avoid blending information from adjacent columns and enabling more

accurate analysis.

The next step is to generate vector representations from the data. To that end, we must import the embedding model from Hugging Face that we will use. But before this, we highly recommend to divide the list into several data chunks, we do this to reduce the computer work stress and time invested in embedding. After extensive research, we found that the model which best suited our needs was **Google's CANINE-C** [31].
This model stands out for operating at the character level, rather than tokenizing each word or sentence (as we observed with other models in the previous chapter). It was designed with two main objectives: predicting a hidden word in a sequence (Masked Language Modeling, MLM) and determining whether two sentences belong to the same text sequence (Next Sentence Prediction, NSP). These defined objectives are not the ones we are targeting, but this does not mean that the model does not perform well for our task. Under this premise, the model is designed to use the entire sentence to make decisions, which makes it well-suited for tasks such as sentence or token classification, or question answering — but not for text generation.

For this reason, it fits perfectly with the objective we are pursuing. By operating at the character level, it becomes easier to capture small lexical differences between records. It is also worth noting that the resulting embedding vectors have a dimensionality of 384, which will be relevant afterwards, when we will analyze how to optimize the code and the problems encountered.

One of the techniques considered and tested was dimensionality reduction using PCA. Through PCA, we aimed to reduce the dimensionality of each vector, thereby decreasing the execution time of the similarity search.
Although this approach had the potential to significantly speed up the process, it ultimately rendered the similarity search ineffective. Further details on this outcome are provided in the Results chapter.

4.2.2 Data To Vector Store

Once we have the data chunks and the embedding model loaded, we proceed to vectorize the data using multiprocessing.
With multiprocessing, we can divide the CPU into multiple cores, allowing us to execute the vectorization process in parallel. In our case, after some testing, we decided to use 3 parallel processes. This enables a more efficient execution of the embedding step, which accounts for the majority of the total processing time. In the next chapter, we will analyze how execution time changes depending on the number of rows processed, and whether multiprocessing is applied or not.

Once the list of vectors is generated, we aim to create a vector store. This vector store is based on a FAISS index, which stores the vectors and organizes them according to their Euclidean distance.

While developing the code, we designed a possible approach to manage the creation of the vector store by clustering the vectors. With that purpose in mind, we performed clustering using the `KMeans` method provided by FAISS, setting the number of clusters we considered optimal. In the next chapter, we will further explore the differences between both approaches, and discuss the decisions made.

4.2.3 Similarity Search

Once the vector store has been created, let's explain how the similarity search method works for both of the approaches we designed: the vector store-based approach and the clustering-based approach.

1. In the case of the vector store created, we simply execute a multiprocessing method that returns a list of elements with the shortest distance to a given vector. In our case, we set this number to 10 elements, although later we will analyze the differences that arise when varying this parameter.

   We must also highlight a very important parameter when performing similarity search: the `score_threshold`. This threshold defines the maximum allowable distance — any records with a greater distance are excluded from the results. According to the FAISS documentation, applying a score threshold can significantly improve the quality of nearest neighbor results, particularly when working with dense and high-dimensional vector spaces [32].

   As a result, our worker returns a list of sublists, each containing the indices of the most similar records. Furthermore, to assist the LLMs, we specify that if a sublist contains fewer than two elements, it should be discarded. This allows us to avoid evaluating records that are not likely to be duplicates.

2. On the other hand, to speed up the process, we can compare each vector only with the vectors belonging to the cluster whose centroid is closest. In other words, the task can be simplified: instead of comparing each vector with all others, we associate each vector with its nearest centroid and compare it only with the vectors within that cluster.

   While this approach reduces execution time, is it really the best solution? Later, we will analyze the performance differences between both methods.

3. A third option was considered and designed, based on the idea of a matrix operation. Thanks to the FAISS library, we had the ability to generate an $N \times N$ matrix containing the pairwise distances between all vectors. This approach could lead to a faster, simpler, and more straightforward way of retrieving the most similar vectors, compared to the other two methods previously discussed — especially if combined with the use of the multithreading module to execute multiple operations in parallel. In this case, we would simply need to extract the desired number of closest distances for each element in the matrix.

   Although this idea had a lot of potential, it came in a very late state of the development and we decided to carry on, and maybe get back to it in a future imporvement or update.

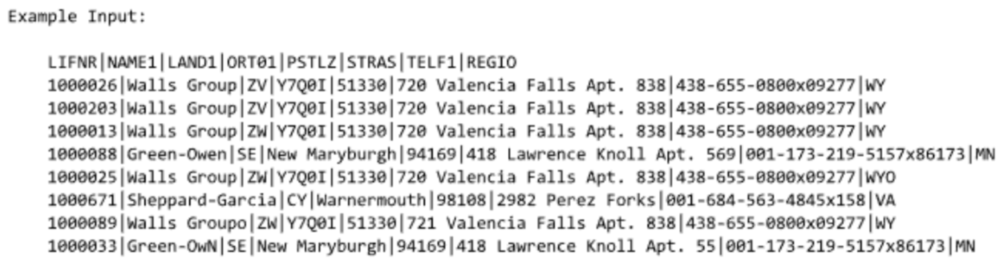
In conclusion regarding the similarity search, regardless of the approach chosen, the output is a list of sublists. Each sublist consists of ten tuples, each containing the ID that links the vector to the original row and the computed distance. It is important to note that the first element of each sublist will always be the vector itself, as it has zero distance to itself. We will refer to this first record as the **"original identifier"**, to distinguish it from the possible duplicates.

4.3 Harmonization Workflow

As mentioned earlier, the last step results in a list composed of the ten most similar records for each row. By reducing the number of records that the LLMs need to analyze to a smaller subset, we can now build a LangChain-based workflow aimed at performing self-evaluation.
This process begins with the definition of the workflow, where we establish the two agents to be used. Although we could have worked with other, less expensive alternatives models, we chose to use the GPT-4o API to ensure the highest possible quality of the results.
Lets take a look to both agents:

4.3.1 Generator Agent:

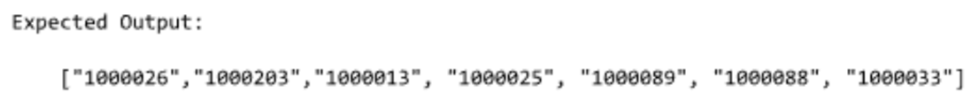
The purpose of this agent is to receive the initial input and search within the list of similar records for those that are either exact matches or contain slight differences, which could indicate potential errors (as the example shown before).
Once the duplicates have been identified, the agent returns a list containing the IDs of the matching records. If no duplicates are found, the agent returns `"None"`. Lets look to an example of how does this agent work:



```
Example Input:

    LIFNR|NAME1|LAND1|ORT01|PSTLZ|STRAS|TELF1|REGIO
    1000026|Walls Group|ZV|Y7Q0I|51330|720 Valencia Falls Apt. 838|438-655-0800x09277|WY
    1000203|Walls Group|ZV|Y7Q0I|51330|720 Valencia Falls Apt. 838|438-655-0800x09277|WY
    1000013|Walls Group|ZW|Y7Q0I|51330|720 Valencia Falls Apt. 838|438-655-0800x09277|WY
    1000088|Green-Owen|SE|New Maryburgh|94169|418 Lawrence Knoll Apt. 569|001-173-219-5157x86173|MN
    1000025|Walls Group|ZW|Y7Q0I|51330|720 Valencia Falls Apt. 838|438-655-0800x09277|WYO
    1000671|Sheppard-Garcia|CY|Warnermouth|98108|2982 Perez Forks|001-684-563-4845x158|VA
    1000089|Walls Groupo|ZW|Y7Q0I|51330|721 Valencia Falls Apt. 838|438-655-0800x09277|WY
    1000033|Green-OwN|SE|New Maryburgh|94169|418 Lawrence Knoll Apt. 55|001-173-219-5157x86173|MN
```

Figure 4.1: Example of the input format used in the workflow.



```
Expected Output:

    ["1000026","1000203","1000013", "1000025", "1000089", "1000088", "1000033"]
```

Figure 4.2: Example of the output after duplicate detection.

As we can observe, the agent retrieves the `LIFNR` identifiers of the records that share most of their values but differ in the ID. One detail that led us to modify part of the implementation (specifically the similarity score threshold) was the agent's tendency to retrieve pairs of records that were similar to each other, but not to the original input record — as seen in the case of "1000088" and "1000033".

This agent only receives the prompt and the input text as the context. This prompt is one of the main fields of study in this project, and we will detail the research afterwards, along the prompt of the Evaluator agent.

4.3.2 Evaluator Agent:

On the other hand, the evaluator agent is responsible for reviewing and determining whether the generator's output is correct in both form and content. If the output is deemed correct, the list is saved locally. However, if the response is incorrect, the agent sends feedback to the generator and requests a new response. The output can be wrong because of three main reasons: Wether the answer does not contain one record that is a duplicate, or the opposite, and if the structure of the answer is not a list or a `"None"`.

This structure requires the model to be configured with two prompts: one for analyzing the output, and another for providing appropriate feedback to the generator. It also needs to receive both the response from the previous agent and the original input.
Although this might seem somewhat redundant — *why give two models the same input?* — the presence of an additional agent evaluating the output significantly increases the number of successful results compared to using a single agent.
On the downside, the execution time nearly doubles, and consequently, the energy consumption also increases.

4.3.3 Best practices for Prompt Engineering

Despite the fact we are not able to show directly the prompts used, we the consider it important to highlight some key steps in prompt engineering, which has been essential for developing the prompts required by both models.
It is crucial that the prompt be brief and concise — including unnecessary details will only create confusion in the model's response. Furthermore, it is essential to clearly separate specific instructions using annotations such as quotation marks ("") or dashes (-).
It is also highly recommended to include an example execution, showing both an input case and the corresponding output, to guide the model's behavior more effectively. These few tips, along with other techniques, have significantly improved our results — making the difference between a useful tool and a meaningless one.

Some of the prompt engineering strategies applied in this project were learned through the course *ChatGPT Prompt Engineering for Developers*, offered by DeepLearning.AI in collaboration with OpenAI [33]. This training provided practical guidance on crafting effective prompts for large language models, which proved essential for designing the instructions used by both agents.

In addition, we adjusted a series of parameters that control the behavior of the model [2]:

- `temperature`: This parameter determines the randomness of the model's responses, influencing how it selects the next word. The higher the temperature, the more creative and varied the output tends to be.

  Temperature works in such a way that, when generating a token, the model predicts which tokens are likely to follow. This selection is then modulated by the temperature: with a low temperature, the model favors high-probability tokens (more deterministic behavior); whereas with a high temperature, it gives more weight to lower-probability tokens, resulting in more diverse outputs.

- `top_p` (nucleus sampling): This parameter defines the subset of tokens to be considered during generation. The model selects from the smallest set of tokens whose cumulative probability exceeds the value of `top_p`.

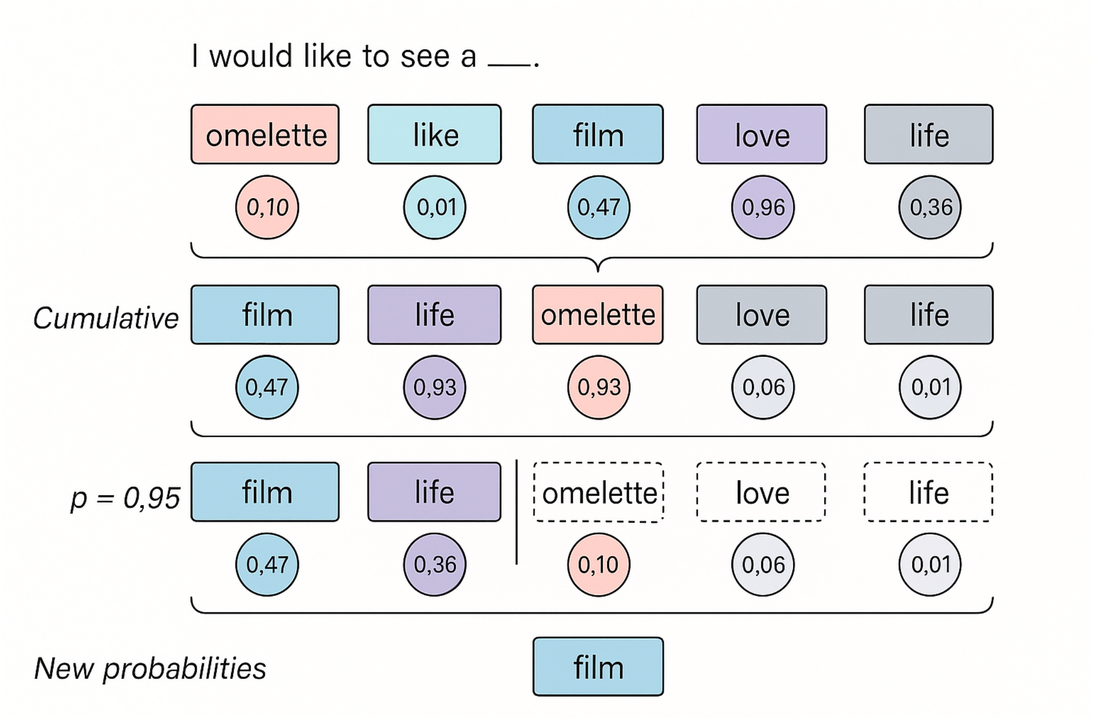


Figure 4.3: Illustration of nucleus sampling ($p = 0.95$) applied to the sentence *"I would like to see a ___ "*. Inspired in the Vellum article [2]

4.3.4 Workflow Architecture

After understanding how the LLM agents operate, we now turn to the structure of the workflow: its components and how it functions.

Before explaining the architecture itself, we must highlight a small but important detail regarding data distribution. For the workflow to execute correctly, it is essential to associate the vectors extracted during the similarity search step with the corresponding original text entries. As mentioned earlier in the development, we had already saved the original records as string representations, making it easy to replace the list of vectors with a list of the corresponding texts.

Another important detail is that we discovered that, for improved understanding and performance of the LLMs, it is beneficial to add a header line to each list, containing the names of the fields (the column names).

The implemented workflow follows the "Evaluator–Optimizer" structure defined in the LangGraph library [34]. This workflow consists of two nodes—corresponding to the previously defined agents (Generator and Evaluator)—and four edges. Another important element in the workflow worth mentioning is the *state*. The `state` is a defined class that acts as a typed dictionary, storing various variables essential for the correct execution of the workflow. These variables are continuously updated and reset in each loop of the workflow. In our case, the `state` is used to store:The input text with its possible duplicates, the output generated by the generator agent, the review and feedback provided by the evaluator agent, and the three different prompts previously mentioned. The `state` class offers great flexibility and ease of implementation.

We present these elements below by describing the overall workflow process:

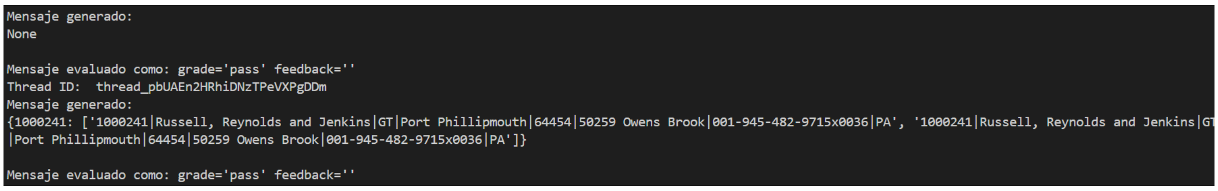
1. Initially, the first edge represents the input of the text data into the generator agent, with the `state` variables being reset.

2. Once a response is generated, it is passed to the evaluator agent. Depending on the evaluation result (stored in the `state`), two possible edges can be followed:

   (a) If the value of `evaluation` is `"pass"`, the output is saved locally, and the loop begins again with a new sublist from the main list.

   (b) If the value of `evaluation` is not `"pass"`, the evaluator sends feedback to the generator, and the loop restarts with the same input until the `evaluation` value becomes `"pass"`.

In conclusion, in every iteration of the loop we send a list of texts that might be duplicate, or not. The generator model retrieves a list of the identifiers of those records that it considers a duplicate or an error. And the evaluator model reviews that response and gives an evaluation and a feedback. The following image is an example of this loop:



```
Mensaje generado:
None

Mensaje evaluado como: grade='pass' feedback=''
Thread ID:  thread_pbUAEn2HRhiDNzTPeVXPgDDm
Mensaje generado:
{1000241: ['1000241|Russell, Reynolds and Jenkins|GT|Port Phillipmouth|64454|50259 Owens Brook|001-945-482-9715x0036|PA', '1000241|Russell, Reynolds and Jenkins|GT
|Port Phillipmouth|64454|50259 Owens Brook|001-945-482-9715x0036|PA']}

Mensaje evaluado como: grade='pass' feedback=''
```

Figure 4.4: Workflow iteration display example

The final result from the workflow is another list, much smaller, with the lists of the identifiers that are selected as duplicates. The list contains `"None"` elements, but these will be deleted in the moment the list is finished.

For a better understanding of this simple workflow, lets have a look to a diagram that resumes it.

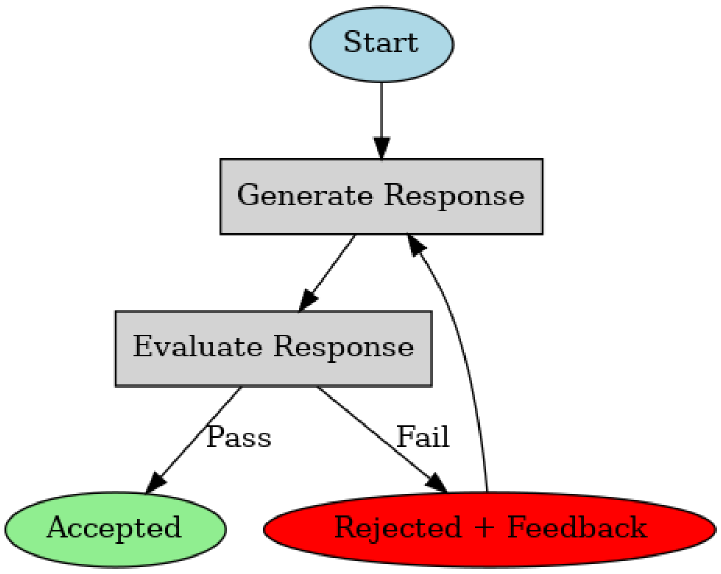


Figure 4.5: Workflow implemented using LangGraph, following the Evaluator–Optimizer pattern.

4.4 Database Results Insertion

Finally, we just need to export our data to an external database. In our case, we used SQLite to test whether the execution was working correctly.
The table created to store the data contains a column called `MASTER_ID`, which represents the original identifier, followed by the values of all the fields from the record identified as a duplicate.
To achieve this, we had to go through a somewhat tedious process of rearranging the data and associating the identifiers with the corresponding values from the original source.
We found that the cleanest way to build the final table was by creating a DataFrame from the processed data and converting it directly into a database table.

In the initial approach, the result returned by the generator agent was a JSON object containing the original identifier as the key and the duplicate identifiers as the values. However, we changed this to a simple list so that both agents would have fewer difficulties interpreting the given instructions. When we had the JSON object the DataFrame was easier to create, but with a list of identifier, we had to separate the first value and associate the rest with their information.

4.5 Challenges Encountered

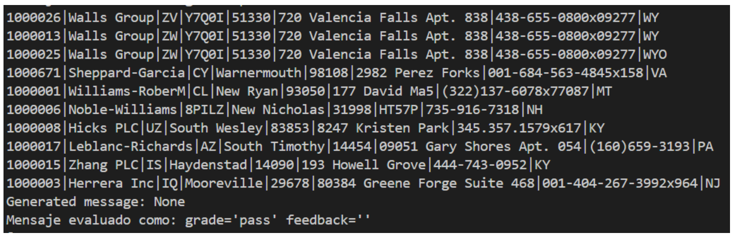
Embedding Times

The main issue we encountered during the development of this project was the significant amount of time required to execute the entire process.
To give an idea, we aimed to process files containing around one hundred thousand records. The similarity search alone took approximately 1581.03 seconds — roughly 26 minutes. When including the execution time of the full workflow, the total duration could easily be ten times longer than that.

As we could hardly improve the time spend by the workflow, all out attention was focused into improving the time spend into embedding the data and doing the similarity search. After some researching and tests, the only improvement we could established was the introduction of multiprocessing with three cpu, reducing the time to approximately 14 minutes.

Errors in the responses

When working with LLMs, expecting consistently correct responses is more than just challenging.
Errors may appear not only in the output produced by the generator agent, but also in the evaluation and feedback provided by the evaluator agent.
This requires us to be extremely precise with the prompts used and to pay close attention to the wording of the instructions — often needing to split hairs. Despite the fact that we could show a great list of errors shown, we are only going to display one for a better understanding.



```
1000026|Walls Group|ZV|Y7Q0I|51330|720 Valencia Falls Apt. 838|438-655-0800x09277|WY
1000013|Walls Group|ZW|Y7Q0I|51330|720 Valencia Falls Apt. 838|438-655-0800x09277|WY
1000025|Walls Group|ZW|Y7Q0I|51330|720 Valencia Falls Apt. 838|438-655-0800x09277|WYO
1000671|Sheppard-Garcia|CY|Warnermouth|98108|2982 Perez Forks|001-684-563-4845x158|VA
1000001|Williams-RoberM|CL|New Ryan|93050|177 David Ma5|(322)137-6078x77087|MT
1000006|Noble-Williams|8PILZ|New Nicholas|31998|HT57P|735-916-7318|NH
1000008|Hicks PLC|UZ|South Wesley|83853|8247 Kristen Park|345.357.1579x617|KY
1000017|Leblanc-Richards|AZ|South Timothy|14454|09051 Gary Shores Apt. 054|(160)659-3193|PA
1000015|Zhang PLC|IS|Haydenstad|14090|193 Howell Grove|444-743-0952|KY
1000003|Herrera Inc|IQ|Mooreville|29678|80384 Greene Forge Suite 468|001-404-267-3992x964|NJ
Generated message: None
Mensaje evaluado como: grade='pass' feedback=''
```

Figure 4.6: Workflow error example.

It is clear that the example showed should retrieve a list of three identifier, the ones detected as duplicates, but none are displayed. Neither does the evaluator detect this error in the display.

Iterations limit

Occasionally, the code raises an error related to the maximum number of allowed iterations. This occurs when the generator model and the evaluator fail to reach a conclusion. To address this issue, we can temporarily increase the maximum number of iterations from the default value of 25 to 100. However, this is only a temporary workaround until the root cause is fixed. One possible reason for the process entering this loop is the lack of precision in the prompts, which may cause confusion for the model. To better understand and resolve this issue, we can insert `print` statements to inspect what each node is generating, allowing us to identify at which point in the workflow the loop is occurring. Looking at the points were the loop is generated, we can try to understand why does this loop appear, and stablish two possible corrections for it:

Text input cleaning

While reviewing each step of the loop, we noticed that an object called `AIMessage` was being printed to the screen. This object contained the message itself, along with the number of tokens used and other metadata.
To prevent the evaluator model from having to process the entire content of the object, we modified the generation step so that it returned only the message.

Feedback not received

Another potential issue in the loop was that the generator agent might not take into account the feedback provided by the evaluator agent.
To ensure that the generator received the feedback, we manually added it to the `state` class. One possible drawback of this approach was that the feedback could accumulate in the `state`, potentially hindering the generator's performance.
Fortunately, the `state` is reset in each iteration, meaning that the feedback only influences the next message.
However, this implies that in cases where more than two corrections are required, the generator could enter a loop of repeating previous errors that had already been addressed in earlier feedback.

So, our final observation, is that the best way to fix the loop problem is by modifying the prompts, with the advices mentioned earlier.

Chapter 5

Results

In this chapter, we present the results and the various tests conducted on the implemented steps.
To work in a controlled scenario and obtain more precise insights, we modified the original dataset to contain only 500 records. Among these, 50 were manually altered to include errors following the specific structure we aimed to detect. This configuration allowed us to systematically adjust different aspects of the code in order to correct the introduced errors. The newly modified dataset is named `DQ_checks_modified`, although this information is essentially irrelevant for understanding the development presented.

5.1 Similarity Search

In a first approach where we were using PCA and the embedding model 'grammarly/coedit-large', we modify into two types of errors tests:

1. **Simple test with 5 entries containing varied errors:**

   By introducing 5 errors of different types—such as minor grammatical mistakes (capitalization, missing spaces, etc.), character omission or transposition, and modifications in phone numbers or postal codes—we aim to verify whether the workflow is capable of detecting small or specific cases.

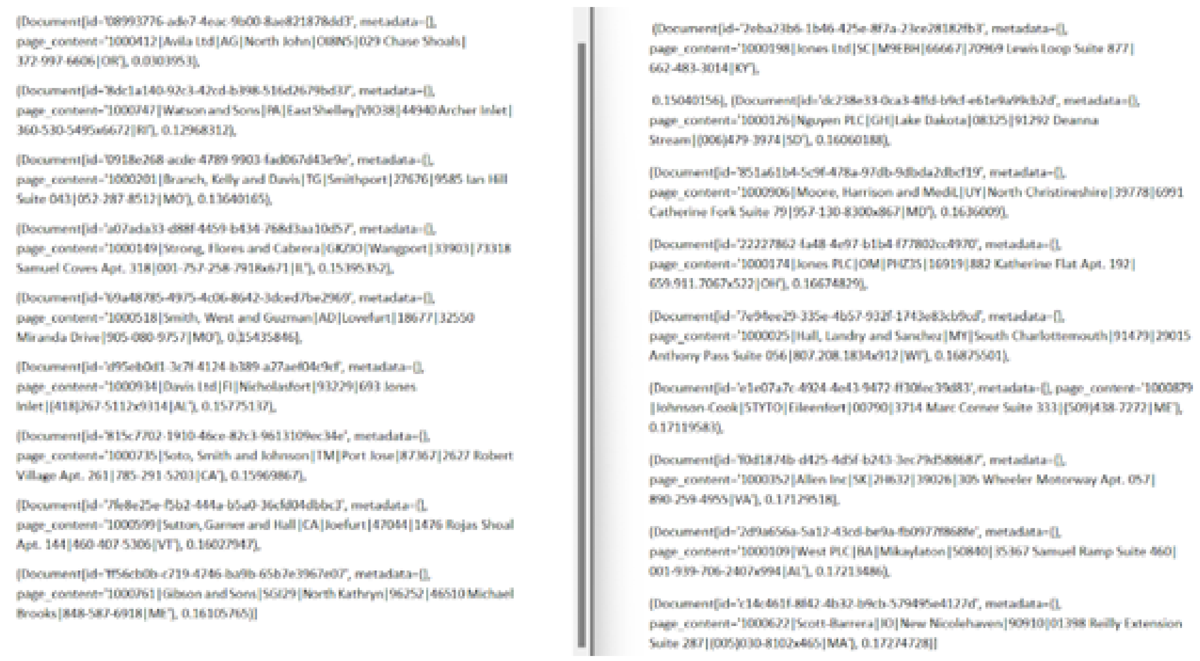
2. **Relocation test:**

   This test is slightly more complex than the previous one, as it aims to assess whether the model can identify potential cases where a company has changed its location or phone number, and may therefore be duplicated in the Master Data Table.

The results given by the tests were awful. For example, in a case with some characters modify like 'Zimbabwe' to 'Zimvavwe', or '2014' to '3014', the score or distances from the vectors were of 80 points, which is a very bad result, as we will see later. After deleting the PCA step (with the cost of more executing time), the distance retrieved was more optimistic, about 2.5 points.
But this results were far from correct, so we decided to change the perspective and try with another embedding model.

5.1.1 Embedding performance

To verify whether the embedding is the component causing issues during execution, we tested the same scenario using both embedding models: **google/canine-c** and **grammarly/coedit-large**. In the following image, we can observe that the results remain constant, as the vectors generated through the embedding process and the Euclidean distance calculation are always the same.



```
[(Document(id='08993776-ade7-4eac-9b00-8ae821878dd3', metadata={},
page_content='1000412|Avila Ltd|AG|North John|O8BN5|029 Chase Shoals|
372-997-6606|OR'), 0.0303953),

(Document(id='8dc1a140-92c3-42cd-b898-516d2679bd37', metadata={},
page_content='1000747|Watson and Sons|PA|East Shelley|VIO38|44940 Archer Inlet|
360-530-5495x6672|RI'), 0.12968312),

(Document(id='0918e268-acde-4789-9903-fad067d43e9e', metadata={},
page_content='1000201|Branch, Kelly and Davis|TG|Smithport|27676|9585 Ian Hill
Suite 043|052-287-8512|MO'), 0.13640165),

(Document(id='a07ada33-d88f-4459-b434-768d3aa10d57', metadata={},
page_content='1000149|Strong, Flores and Cabrera|GKZ0O|Wangport|33903|73318
Samuel Coves Apt. 318|001-757-258-7918x671|IL'), 0.15395352),

(Document(id='69a48785-4975-4c06-8642-3dced7be2969', metadata={},
page_content='1000518|Smith, West and Guzman|AD|Lovefurt|18677|32550
Miranda Drive|905-080-9757|MO'), 0.15435846),

(Document(id='d95eb0d1-3c7f-4124-b389-a27aef04c9ef', metadata={},
page_content='1000934|Davis Ltd|FI|Nicholasfort|93229|693 Jones
Inlet|(418)267-5112x9114|AL'), 0.15775137),

(Document(id='815c7702-1910-46ce-82c3-9613109ec34e', metadata={},
page_content='1000735|Soto, Smith and Johnson|TM|Port Jose|87367|2627 Robert
Village Apt. 261|785-291-5203|CA'), 0.15969867),

(Document(id='7fe8e25e-f5b2-444a-b5a0-36cfd04dbbc3', metadata={},
page_content='1000599|Sutton, Garner and Hall|CA|Joefurt|47044|1476 Rojas Shoal
Apt. 144|460-407-5306|VT'), 0.16027947),

(Document(id='ff56cb0b-c719-4746-ba9b-65b7e3967e07', metadata={},
page_content='1000761|Gibson and Sons|SGI29|North Kathryn|96252|46510 Michael
Brooks|848-587-6918|ME'), 0.16105765)]
```

```
[(Document(id='2eba23b6-1b46-425e-8f7a-23ce28182fb3', metadata={},
page_content='1000198|Jones Ltd|SC|M9EBH|66667|70969 Lewis Loop Suite 877|
662-483-3014|KY'),

0.15040156), (Document(id='dc238e33-0ca3-4ffd-b9cf-e61e9a99cb2d', metadata={},
page_content='1000126|Nguyen PLC|GH|Lake Dakota|08325|91292 Deanna
Stream|(006)479-3974|SD'), 0.16060188),

(Document(id='851a61b4-5c9f-478a-97db-9dbda2dbcf19', metadata={},
page_content='1000906|Moore, Harrison and Medil|UY|North Christineshire|39778|6991
Catherine Fork Suite 79|957-130-8300x867|MD'), 0.1636009),

(Document(id='22227862-fa48-4e97-b1b4-f77802cc4970', metadata={},
page_content='1000174|Jones PLC|OM|PHZ35|16919|882 Katherine Flat Apt. 192|
659.911.7067x522|OH'), 0.16674829),

(Document(id='7e94ee29-335e-4b57-932f-1743e83cb9cd', metadata={},
page_content='1000025|Hall, Landry and Sanchez|MY|South Charlottemouth|91479|29015
Anthony Pass Suite 056|807.208.1834x912|WY'), 0.16875501),

(Document(id='e1e07a7c-4924-4e43-9472-ff30fec39d83', metadata={}, page_content='1000879
|Johnson-Cook|STYTO|Eileenfort|00790|3714 Marc Corner Suite 333|(509)438-7272|ME'),
0.17119583),

(Document(id='f0d1874b-d425-4d5f-b243-3ec79d588687', metadata={},
page_content='1000352|Allen Inc|SK|2H632|39026|305 Wheeler Motorway Apt. 057|
890-259-4955|VA'), 0.17129518),

(Document(id='2d9a656a-5a12-43cd-be9a-fb0977f8686e', metadata={},
page_content='1000109|West PLC|BA|Mikaylaton|50840|35367 Samuel Ramp Suite 460|
001-939-706-2407x994|AL'), 0.17213486),

(Document(id='c14c461f-8842-4b32-b9cb-579495e4127d', metadata={},
page_content='1000622|Scott-Barrera|JO|New Nicolehaven|90910|01398 Reilly Extension
Suite 287|(005)030-8102x465|MA'), 0.17274728)]
```

Figure 5.1: Comparative image between Canine-c (left) and Coedit-large (right)

The image does not clearly indicate which result is correct or why. However, if we examine the distances or scores of the first element (the closest vector), we observe that the score is approximately 0.03, whereas the opposite one is around 0.15. This difference provides a useful reference for the distance range we can expect when identifying potential duplicate records.

5.1.2 Similarity Search time scale variations

An empirical analysis was conducted to measure the execution time of the semantic search system as a function of the number of processed records. As the number of elements increases, the time required to generate the vector store and perform similarity comparisons grows significantly. The complete results are presented in **Appendix A**.

In conclusion, we ended up using a 3-CPU multiprocessing approach for both the embedding and the similarity search. This means that, in a final implementation, the time required to embed the documents, create a vector store from these vectors, and perform the similarity search is approximately 756 seconds or almost 13 minutes for a hundred thousand documents.

Although no visual proof is provided in this document, our experiments consistently demonstrated that the similarity search method outperforms alternative approaches in terms of accuracy, reliability, and execution time. In particular, it proved more effective than clustering-based methods when it came to correctly grouping or identifying duplicate records.

This conclusion is supported by:

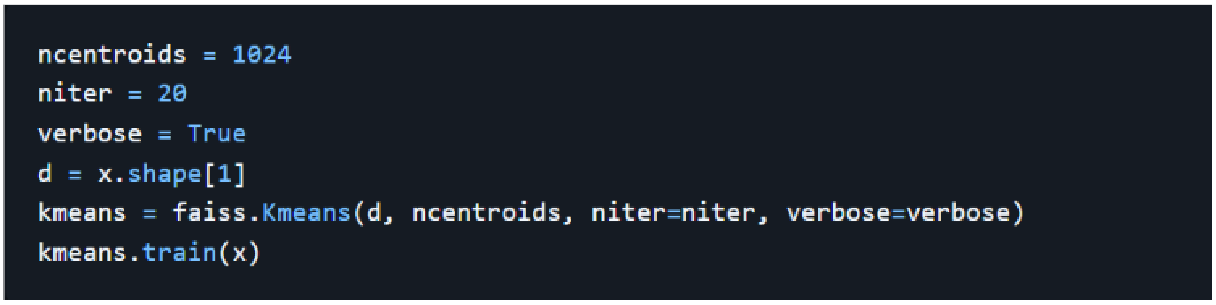
- Its consistent performance across multiple dataset sizes.
- Its ability to handle highly heterogeneous data without requiring predefined parameters (such as the number of clusters).
- The comparative under-performance of clustering, as shown by the low silhouette scores and the dispersion of duplicate records across clusters.

As a result, similarity search was selected as the core mechanism for detecting potential duplicates in our final implementation.

5.1.3 Clustering

This subsection is quite more detailed than the others, because we think that the study of this technique required a lot of time and tests to useful not be written up. Unfortunately, we decided not to use clustering for the reasons explained

After some tests and researching we decided to try on the KMEANS algorithm with the Faiss implementation:



```
ncentroids = 1024
niter = 20
verbose = True
d = x.shape[1]
kmeans = faiss.Kmeans(d, ncentroids, niter=niter, verbose=verbose)
kmeans.train(x)
```

Figure 5.2: Example of Kmeans clustering implementation from Faiss Documentation [3]

This approach would allow for seamless integration between the existing similarity search methodology and the new clustering functionality. However, using K-Means introduces a significant limitation: the requirement to define the exact number of clusters in advance. In a real-world scenario, it is not feasible to determine this number accurately, as doing so would imply prior knowledge of the exact amount of errors or duplicates present—making much of this effort unnecessary.

In the case of implementing this approach with the existing codebase, we propose the following pipeline:

1. All records are passed through the embedding model to generate vectors. This set of vectors is denoted as V.
2. We apply clustering using an algorithm (for simplicity, K-Means):

   (a) Perform a sweep over the number of clusters $n \in \{2, 5, \ldots, 50\}$ and compute evaluation metrics for each run.
   (b) Select the best run, e.g., for $n = 10$.
   (c) Divide V into $n = 10$ clusters. That is, partition V into subsets V_n with corresponding centroids C_n.

3. We define the search pipeline as follows:
   (a) Given a vector $a \in V$, compute its similarity to all centroids C_n. For example, assume C_1 and C_3 are the closest.
   (b) Search for the most similar vectors within the corresponding subsets V_1 and V_3.
   One possibility is to combine V_1 and V_3 into a single vector store for the search. Whether this is viable depends on the complexity of implementation.

To optimize the aforementioned hyperparameters—specifically, the number of clusters—we developed a script that calculates both the silhouette score and the Calinski score for each iteration. Also known as the Variance Ratio Criterion, the Calinski and Harabasz score is defined as ratio of the sum of between-cluster dispersion and of within-cluster dispersion. In each run, the number of clusters is varied. The results of these tests can be observed in the following figures:

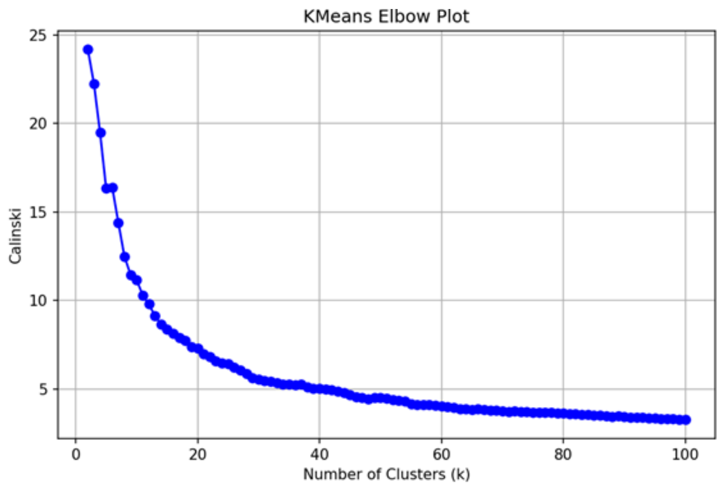


Figure 5.3: Calinski score through 100 iterations of different size clustering

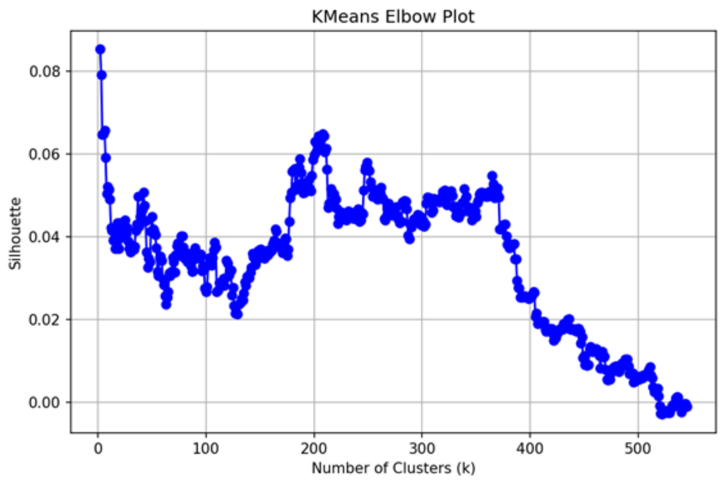


Figure 5.4: Silhouette score across 500 iterations of different size clustering.

These graphs do not show promising results regarding the clustering approach, but we should not give up just yet. Since we needed additional evidence to confirm that clustering was not a better alternative to the already established similarity search method, we decided to analyze how many duplicates were actually grouped by the clusters.

To this end, we created a new table that included an additional column, `ID_OR`, which indicates the original identifier for each row. For example, a duplicate found in position 235 would have the `ID_OR` value corresponding to the position of the correct (original) record. To be more specific, let us consider a particular value of `ID_OR`. By examining the cluster IDs associated with this value, we often observe that the duplicated records are distributed across multiple clusters. Therefore, we only consider as valid groups those sets of records that share the same cluster identifier at least twice.

After manually modifying the entire dataset, adding all values to the new column, and counting the number of duplicated records, we can confidently state that the total number of duplicate groups is 26.

Now that we know the exact rows with errors and the full composition of each group, we can begin evaluating how well the clustering process captures these groups. In the following graph, we present a comparison between the silhouette score and the percentage of total duplicate groups correctly grouped within any cluster:

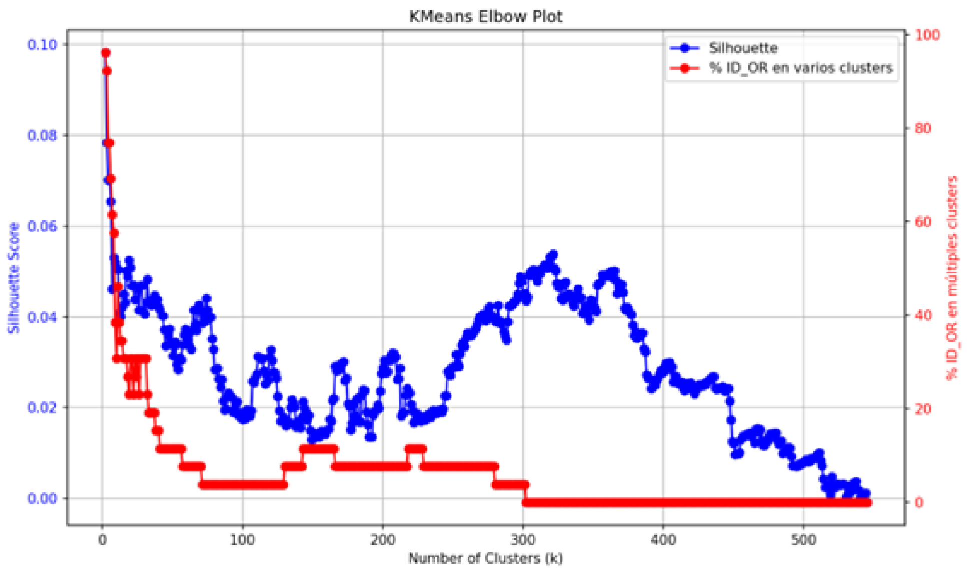


Figure 5.5: Comparison between the silhouette score and the percentage of detected duplicate groups (till 500 clusters).

Again, the result is not as god as expected. We can discard those number of clusters whose percentage of groups gathered is less than the 20%. With this approach, the next graph gives us a new perspective of the problem:

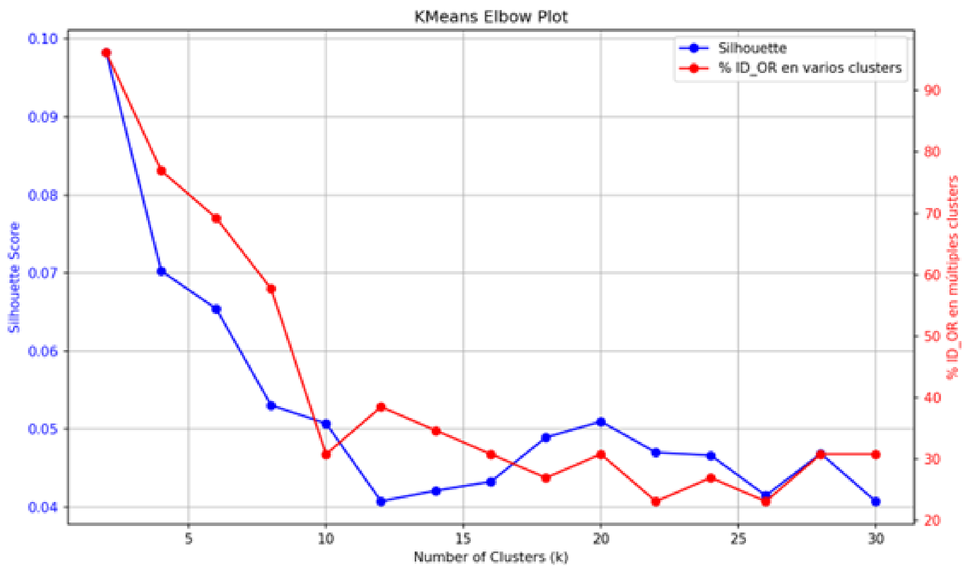


Figure 5.6: Comparison between the silhouette score and the percentage of detected duplicate groups (till 50 clusters).

Based on the results, we can conclude that we are unable to achieve the minimum level of quality or reliability required to implement this approach into our pipeline. Therefore, we will explore alternative ways to improve the results or make this method viable. Although we can see how the percentage improves with less number of clusters, we must not forget that the main objective of using this technique, was to reduce the time invested on the similarity search. But in the **Appendix B**, we can find some graphs that represent the comparison between the time spent on clustering searching and the groups of errors found.

As a conclusion for discarding the clustering approach, our main hypothesis for its poor performance is the high dimensionality introduced by the embedding process. This high-dimensional space likely hinders the ability of clustering algorithms to form well-defined and compact groups, especially when the number of true duplicates is small relative to the dataset size. The following image is a representation we made of the clustering distribution while we were researching for HDBSCAN performance, it can visually explain why clustering may not be the most effective technique to use in this project:

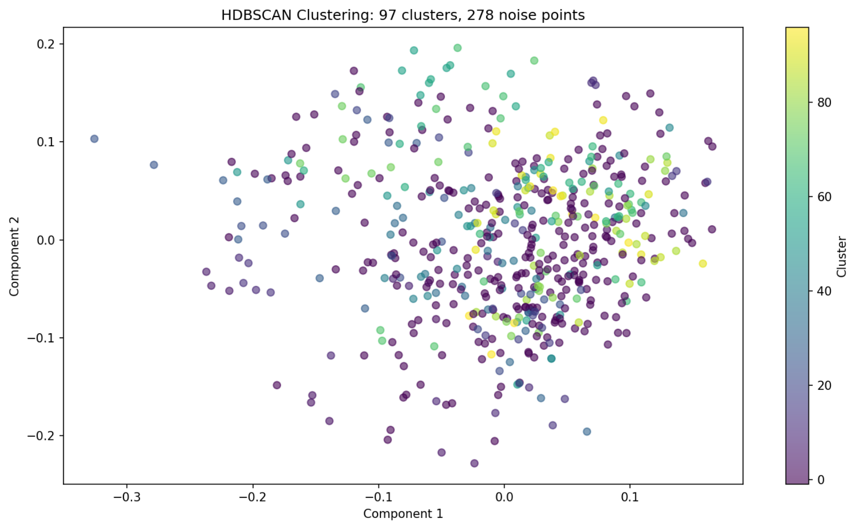


Figure 5.7: HDBSCAN Clustering distribution for 500 records

5.2 Workflow

To test the precision and accuracy of the workflow outputs, we developed a dedicated Python script to evaluate the results. This script took advantage of the column `ID_OR`, originally created for assessing clustering performance, and we introduced a new column, `IS_OR`, which indicates with a boolean value whether a given record is the original one. From the clustering development, we also knew that the number of total error groups was 26. With this addition, we were able to filter the retrieved lists and retain only those where the first identifier corresponds to the original record. This allowed for a more precise and accurate evaluation of the detected duplicate groups. In previous tests, we lacked information about the exact structure of the groups (e.g., whether they contained 1, 2, 3, or 4 errors).

In a first approach, the script just went through the lists obtained and crossed out those identifier we knew were errors. But this solution did not provide the precise information we were looking for, maybe a list erroneously collected a identifier that should not be there, or in a list we could have the same identifier multiple times.

In order to perform a more detailed analysis of the process performance, we designed

a grouped evaluation, where both the retrieved lists and those grouped by the common `ID_OR` values were converted into sets using `frozenset`. This approach ensured that the order of elements within each group would not affect the comparison— meaning that two lists would be considered equivalent if they contained exactly the same elements, regardless of order.
In this process, we calculated the Exact Match Ratio (EMR), which measures the proportion of correctly predicted duplicate groups out of the total number of real groups. We also computed a weighted version of the EMR, which assigns greater importance to larger clusters by weighting the matches based on the number of elements in each group. Finally, we generated a breakdown of the EMR by group size, allowing us to identify which group sizes were more prone to errors.

For a last test, we used the mentioned new column to retrieve those listed identifier that had no errors or duplicates founded. This last and more explicit validation, confirmed once again that the vast majority of known duplicates were succesfully grouped, further supporting the reliability of this approach.

Chapter 6

Conclusion

With the aim of improving the Data Harmonization process—an essential and common step across all projects within the Supply Chain Data & AI department at Accenture—this project sought to detect and remove potential duplicate records from the Master Data Tables, which are often caused by inconsistencies arising from merging data from various sources within a company.
With this objective in mind, we designed an automated system that leverages NLP techniques and LLM implementations to avoid the tedious task of manual data review, as was previously done. The architecture of the proposed process consists of three clearly defined phases, each with its own objectives, but following a linear progression. The phases are:

1. **Data Embedding and Similarity Search:** In this step, we aim to reduce the amount of raw data that LLMs need to analyze in each iteration. To do this, we group together records that are detected as similar. This is achieved through embeddings, which transform each record into vectors of a specific dimensionality. Then, by computing the Euclidean distance between vectors, we generate lists of the most similar records for each one.

2. **LangGraph Self-Evaluated Workflow:** The LLM-based loop is built using an architecture with two interacting agents. The first agent, the Generator, analyzes each list of similar records (in text format) and returns the identifiers of those records that are likely to be duplicates or have high semantic similarity. The second agent, the Evaluator, checks whether the Generator's output is correct based on a set of prompt rules, which define the expected structure of the answer. It determines whether the Generator may have made a mistake. The final output of this workflow is a list of lists containing the identifiers of the records detected as duplicates.

3. **Database Results Insertion:** Finally, for each list of detected duplicates, we link the identifiers back to their corresponding records and insert them into a database table that aggregates all potential duplicates for each original identifier.

Therefore, after executing this process and allowing sufficient time for completion, the user only needs to review the flagged records in the Master Data Table—improving not only execution time but also the overall scalability of the process.

Future Improvements

Although the results have been very favorable and the implementation of the tool can be considered a success, there are still some unexplored areas worth investigating:

- **Matrix multiplication:** One potential approach to reduce the execution time of the similarity search was to compute a full distance matrix between all vectors using matrix multiplication. This would allow us to directly extract the top X closest distances for each record without iterating through pairwise comparisons.
- **Prompt refinement:** LLMs may occasionally fail due to inherent randomness in their responses. To minimize such failures, it is essential to refine and optimize prompt design. Even when randomness is present, well-crafted prompts can help ensure consistent and high-quality outputs.
- **Code optimization:** This thesis does not include detailed explanations of the code structure or implementation, primarily due to confidentiality reasons, as the code is proprietary to Accenture. However, it is important to note that the codebase used in this project could still be optimized—for example, by improving function definitions, adjusting variables, or modifying parameters that may slow down the execution process.

Appendix A

RAG System Execution Time

The following list details the measured times for vector store creation and similarity comparison, using different input sizes and configurations, both with and without multiprocessing:

- **1 element:**
  Embedding and Vector store creation: 0.54 s
  Similarity comparison: 0.08 s
- **5 elements:**
  Embedding and Vector store creation: 0.34 s
  Similarity comparison: 15.85 s
- **10 elements:**
  Embedding and Vector store creation: 0.68 s
  Similarity comparison: 0.07 s
- **100 elements:**
  Embedding and Vector store creation: 0.64 s
  Similarity comparison: 0.53 s
- **1K elements:**
  Embedding and Vector store creation: 1.41 s
  Similarity comparison: 4.38 s
- **10K elements:**
  Embedding and Vector store creation **without** multiprocessing: 359.30 s
  Embedding and Vector store reation **with** multiprocessing: 102.35 s

  Similarity Search comparison **without** multiprocessing: 51.26 s
  Similarity Search comparison **with** multiprocessing: 20.52 s
- **100K elements:**
  Embedding and Vector store creation **without** multiprocessing: 6030.02 s
  Embedding and Vector store creation **with** multiprocessing (2 CPUs): 737.61 s
  Embedding and Vector store creation **with** multiprocessing (3 CPUs): 675.63 s
  Embedding and Vector store creation **with** multiprocessing (4 CPUs): 684.03 s

Similarity Search comparison **with** multiprocessing (3 CPUs): 81.01 s

- **1M elements:**
  Embedding **with multiprocessing** (3 CPUs): 6458.42
  Vector store creation: 1154.85
  Similarity Search comparison **with multiprocessing** (3 CPUs): 4708.96
  Note: some executions triggered a `MemoryError`.

Appendix B

Clustering Comparison Between Time and Quality

The following graphs illustrate the comparison between the time spent on similarity search within clusters and the number of error groups detected. We present, in descending order, three different cases: the first with 1,050 records, the second with 10,000 records, and the last with 100,000 records.

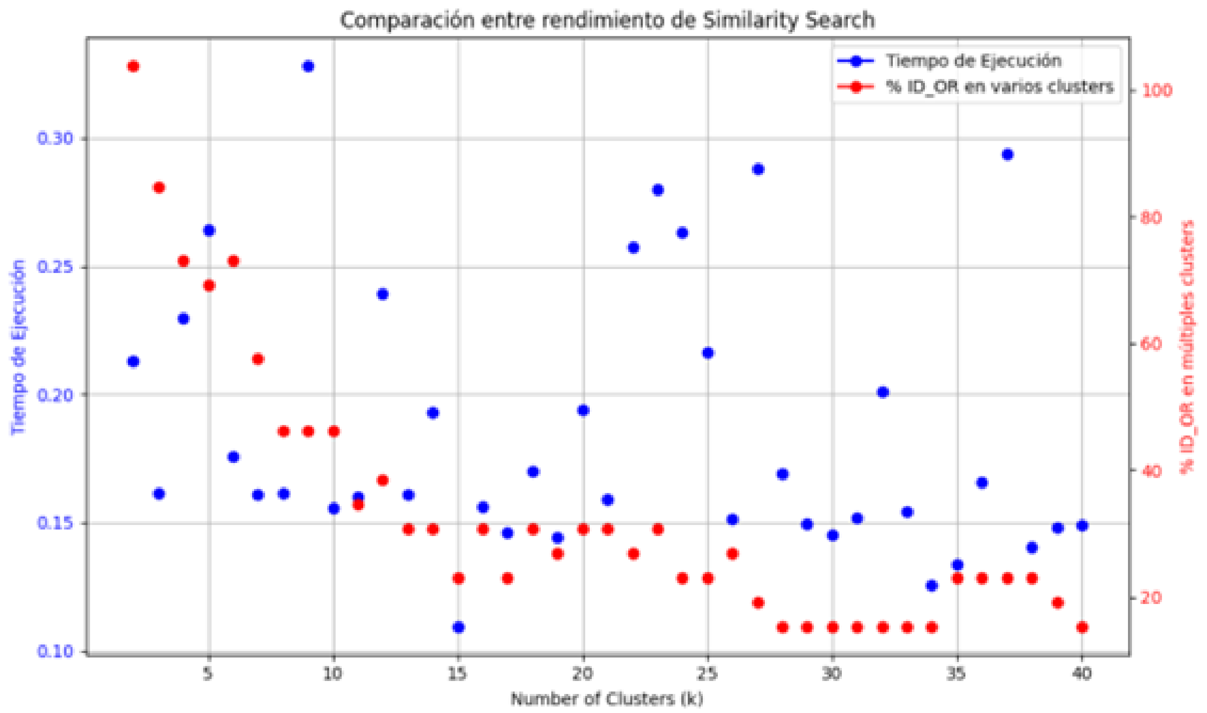


Figure B.1: Similarity search time vs. detected error groups — Dataset of 1,050 records.

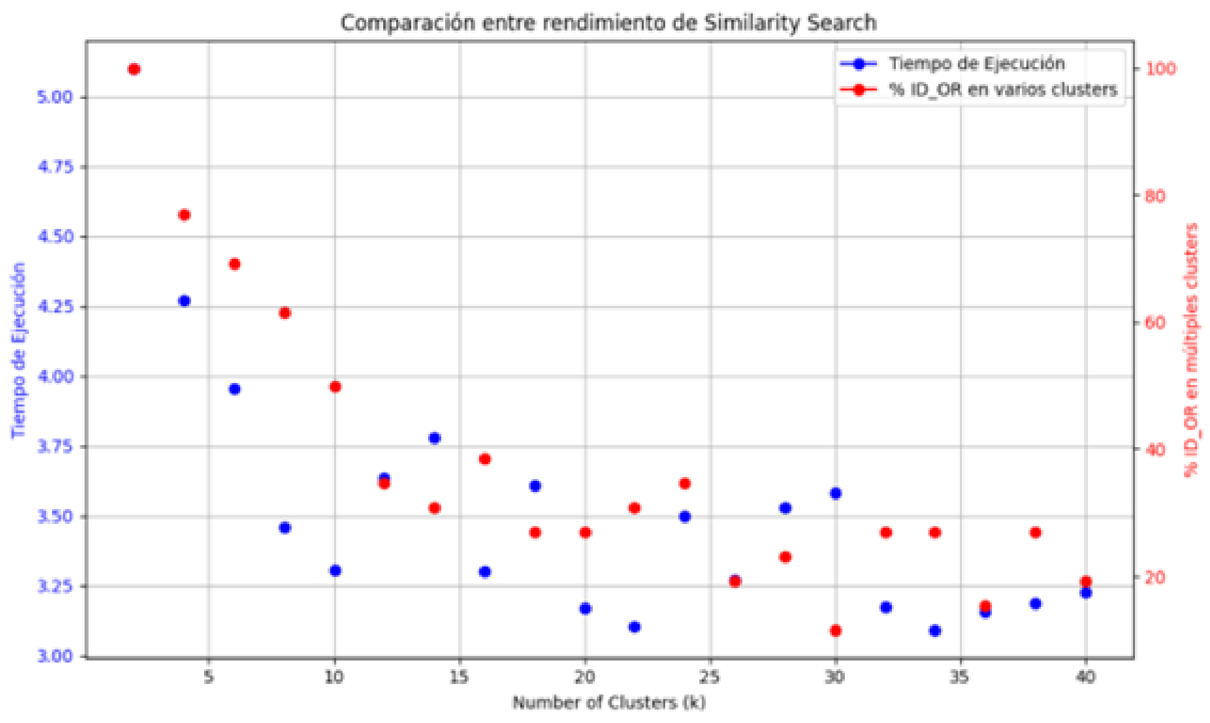


Figure B.2: Similarity search time vs. detected error groups — Dataset of 10,000 records.

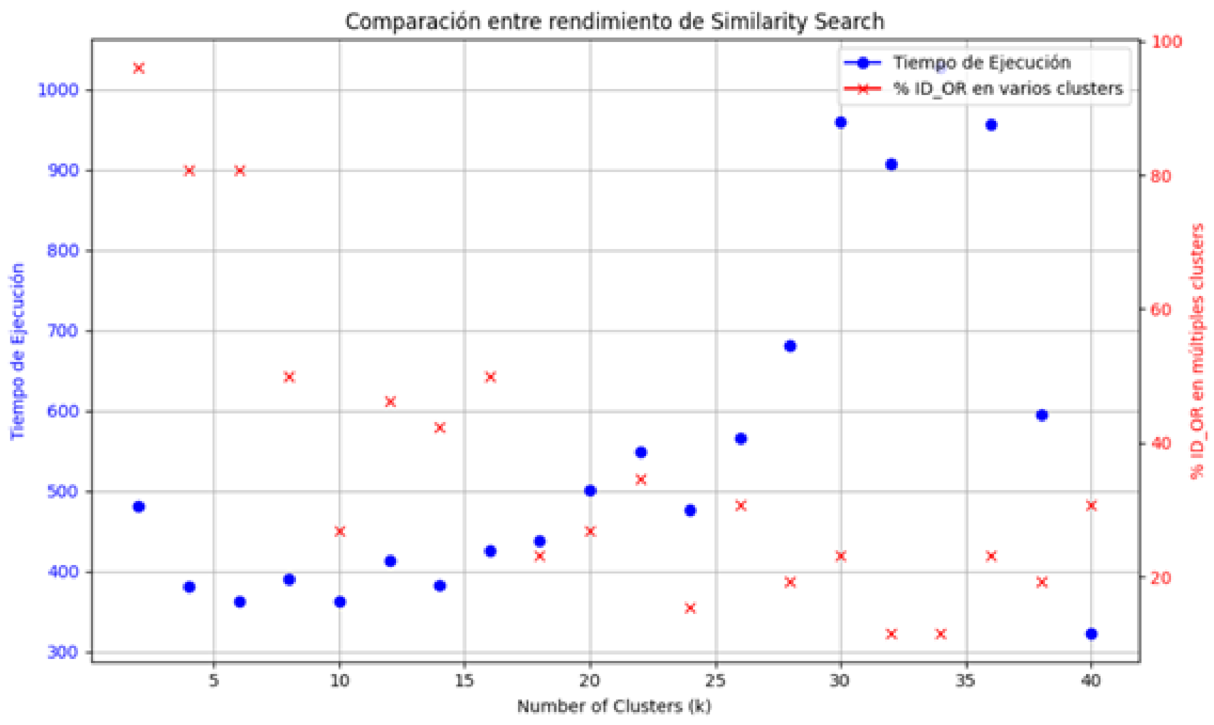


Figure B.3: Similarity search time vs. detected error groups — Dataset of 100,000 records.

These results were surprising, as we would not expect the fastest similarity check to occur when the cluster has the largest size — this outcome appears counterintuitive. A deeper research should help us to light the possible cause of this effect.

Bibliography

[1] TIBCO Software Inc., “What is data harmonization?” 2024. [Online]. Available: https://www.tibco.com/glossary/what-is-data-harmonization

[2] V. AI, “Llm parameters: Temperature, top-p, frequency penalty, and more,” https://www.vellum.ai/llm-parameters, 2024.

[3] F. A. Research, “Faiss building blocks: Clustering, pca, quantization,” https://github.com/facebookresearch/faiss/wiki/Faiss-building-blocks:-clustering,-PCA,-quantization, 2024.

[4] World Economic Forum, “How ai could shape the workplace in 2025,” Jan. 2025. [Online]. Available: https://www.weforum.org/stories/2025/01/ai-2025-workplace/

[5] Hugging Face, “Large Language Models (LLMs),” https://huggingface.co, 2024.

[6] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Łukasz Kaiser, and I. Polosukhin, “Attention is all you need,” *arXiv preprint arXiv:1706.03762*, 2017. [Online]. Available: https://arxiv.org/abs/1706.03762

[7] Stanford HAI, “Introducing Foundation Models,” https://hai.stanford.edu/news/introducing-foundation-models, 2021.

[8] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, “Exploring the limits of transfer learning with a unified text-to-text transformer,” *arXiv preprint arXiv:2002.12327*, 2020. [Online]. Available: https://arxiv.org/abs/2002.12327

[9] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. L. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray, and et al., “Training language models to follow instructions with human feedback,” *arXiv preprint arXiv:2207.05221*, 2022. [Online]. Available: https://arxiv.org/abs/2207.05221

[10] IBM, “Word Embeddings: Dense representations of language,” https://www.ibm.com/mx-es/think/topics/word-embeddings, 2024.

[11] Data ILM, “Word2Vec — From Google Labs to Real-World Applications,” https://medium.com/@datailm/word2vec-from-google-labs-to-real-world-applications-a6b9c305572f, 2023.

[12] Meta AI, “FastText: Supervised Text Classification,” https://fasttext.cc/docs/en/supervised-tutorial.html, 2024.

[13] ——, “FastText: Unsupervised Word Representations,” https://fasttext.cc/docs/en/unsupervised-tutorial.html, 2024.

[14] Princeton University, “Lecture 3: Contextualized Word Embeddings,” https://www.cs.princeton.edu/courses/archive/spring20/cos598C/lectures/lec3-contextualized-word-embeddings.pdf, 2020.

[15] LangChain, “Vector Stores | LangChain,” https://python.langchain.com/docs/concepts/vectorstores/, 2025.

[16] Pinecone, “What is a Vector Database How Does it Work?” https://www.pinecone.io/learn/vector-database/, 2023.

[17] IBM Research, “Retrieval-Augmented Generation (RAG),” https://research.ibm.com/blog/retrieval-augmented-generation-RAG, 2023.

[18] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W. tau Yih, T. Rocktäschel, S. Riedel, and D. Kiela, “Retrieval-augmented generation for knowledge-intensive nlp tasks,” *arXiv preprint arXiv:2005.11401*, 2020. [Online]. Available: https://arxiv.org/abs/2005.11401

[19] Facebook AI Research, “Faiss: Facebook ai similarity search,” 2024. [Online]. Available: https://faiss.ai/index.html

[20] GeeksforGeeks, “Dimensionality Reduction,” https://www.geeksforgeeks.org/dimensionality-reduction/, 2023.

[21] S. Karamizadeh, S. M. Abdullah, A. A. Manaf, M. Zamani, and A. Hooman, “An overview of principal component analysis,” *Journal of Signal and Information Processing*, vol. 4, no. 3B, pp. 173–175, 2013. [Online]. Available: https://www.scirp.org/pdf/JSIP_2013101711003963.pdf

[22] IBM Research, “What is clustering?” https://www.ibm.com/es-es/think/topics/clustering, 2024.

[23] A. K. Jain, M. N. Murty, and P. J. Flynn, “Data clustering: a review,” *ACM computing surveys (CSUR)*, vol. 31, no. 3, pp. 264–323, 1999.

[24] scikit-learn, “HDBSCAN: Hierarchical density-based clustering,” https://scikit-learn.org/stable/modules/generated/sklearn.cluster.HDBSCAN.html, 2024.

[25] B. Chong, “K-means clustering algorithm: a brief review,” *Academic Journal of Computing & Information Science*, vol. 4, no. 5, 2021. [Online]. Available: https://www.francis-press.com/uploads/papers/mmzXrfO0RaPre0zbXGxwW1Er02OAYQumQDD78Hqp.pdf

[26] Python Software Foundation, “multiprocessing — Process-based parallelism,” https://docs.python.org/es/3.9/library/multiprocessing.html, 2024.

[27] Wikipedia contributors, “Thread (computing) – Wikipedia,” https://en.wikipedia.org/wiki/Thread_(computing), 2024, accessed: 2025-06-01.

[28] Hugging Face, “Hugging Face: The AI Community Building the Future,” https://huggingface.co, 2024.

[29] G. Blinowski, A. Ojdowska, and A. Przybyłek, "Monolithic vs. microservice architecture: A performance and scalability evaluation," *IEEE Access*, 2022, corresponding author: grzegorz.blinowski@pw.edu.pl.

[30] Accenture, "Intelligent asset management," 2024. [Online]. Available: https://www.accenture.com/be-en/services/digital-engineering-manufacturing/intelligent-asset-management

[31] J. H. Clark, D. Garrette, I. Turc, and J. Wieting, "CANINE: pre-training an efficient tokenization-free encoder for language representation," *CoRR*, vol. abs/2103.06874, 2021. [Online]. Available: https://arxiv.org/abs/2103.06874

[32] Meta AI, "FAISS Documentation: Filtering and score thresholds in similarity search," https://faiss.ai/, 2024.

[33] DeepLearning.AI and OpenAI, "Chatgpt prompt engineering for developers," https://www.deeplearning.ai/short-courses/chatgpt-prompt-engineering-for-developers/, 2023.

[34] LangGraph, "Langgraph - workflow orchestration tutorial," https://langchain-ai.github.io/langgraph/tutorials/workflows/#orchestrator-worker, 2024.