Dirty Entity Resolution staring Locality-Sensitive Hashing on Incoherent and Incomplete Distributed Big Data: A Feasibility Study

Master Thesis

Colin Geukes

TU Delft

Dirty Entity Resolution starring Locality-Sensitive Hashing on Incoherent and Incomplete Distributed Big Data: A Feasibility Study

Master Thesis

by

Colin Geukes

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Wednesday December 13, 2023 at 13:00.

Student number:	4571606	
Project duration:	March 13, 2023 – December 13, 2023	
Thesis committee:	Dr. A. (Asterios) Katsifodimos,	TU Delft, chair, supervisor
	Dr. H. (Huijuan) Wang,	TU Delft
	Ir. B. (Bastijn) Kostense,	Exact, supervisor
Company supervision:	Dr. S. (Soude) Fazeli,	Exact, supervisor

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

TU Delft Delft University of Technology

= exact

Preface

Before you lies the master thesis “Dirty Entity Resolution starring Locality-Sensitive Hashing on Incoherent and Incomplete Distributed Big Data: A Feasibility Study”. It has been written under the Web Information Systems research group to fulfill the graduation requirements of the Data Science program at the Delft University of Technology. I was engaged in researching and writing this thesis from March to December 2023.

The intended goal of applying for a thesis at a company is to start my transition from a student life to a working life, marking the end of my educational career as a student. I am grateful that Exact gave me this opportunity. Since elementary school, I always had the dream to one day graduate from the Delft University of Technology, and I am proud to say that this goal has been achieved with the handing in of this manuscript.

Now, I want to sincerely thank everyone involved in creating this masterpiece. Thank you Dr. Asterios Katsifodimos, for finding a subject that suits my interests and the help along the way. Thank you, Ir. Bastijn Kostense, for your time and guidance throughout the entire process, you were always available to provide invaluable input, also, I would like to give an honorable mention for your knowledge sharing of your barista and artistic skills. Thank you, Dr. Soude Fazeli, for sharing your invaluable input and knowledge with me; it is much appreciated. I will be forever in debt to you.

Finally, I want to thank my family, particularly my father Kees, mother Angelien, sister Daisy, and all my friends for supporting me throughout my entire educational process. A big thank you to my lovely colleagues at Exact, with whom I spent countless hours working but, more importantly, with whom I did plenty of fun activities; you are all so much more than just colleagues, making these months a breeze. I will conclude by thanking you, my reader: I hope you enjoy your reading.

Colin Geukes
Leiden, December 2023

Disclaimer

The information made available by Exact for this research is provided for use of this research only and under strict confidentiality.

Abstract

In real-world scenarios, users provide invaluable data; however, this data is inherently incoherent, incomplete, and duplicated, i.e., different data rows refer to the same real-world object. Merging duplications to a single entry broadens the knowledge of a given real-world object represented within a data set. Applying a straightforward cross-join operation to check for duplications is infeasible and intractable in big data scenarios. Instead of comparing all rows, the similarity can be approximated by Locality-Sensitive Hashing (LSH). LSH's MinHash creates colliding hashes on common tokens. With enough matching hashes, two rows can be deemed similar. The applications of LSH within an Entity Resolution (ER) pipeline on incoherent and incomplete big data are thoroughly investigated in this work.

An ER pipeline is introduced to make the deduplication problem tractable. This pipeline consists of four phases: preprocessing, blocking, matching, and clustering. In the preprocessing, the data is made coherent. In the blocking, the data is divided into smaller main blocks by blocking on complete properties. These blocks contain a majority of True-Matches. The created blocks can be augmented with additional blocks that rely on incomplete properties. In the matching phase, only the Matches found within the blocks are compared, significantly reducing the required number of comparisons. The incomplete properties cannot be transformed into features for every Match; therefore, classifiers are trained per property combination, allowing for optimal Match classification. In clustering, the duplicated rows are merged into a single Entity. These Entities are created with disconnected sub-graphs of Positive-Matches. These sub-graphs are particularly susceptible to noise. Therefore, novel implementations of MinHash are used for noise removal and finding missed True-Matches. The created Entities were tested on an existing model.

Blocks constructed with MinHash on N-gram tokens allow spelling mistakes to be overcome within the user data, which improves the Pairs Completeness (PC) at the cost of Pairs Quality (PQ) and Reduction Ratio (RR). Introducing augmentative blocks increases the PC significantly at the cost of PQ and a low cost of RR. Multiple runs are investigated throughout the pipeline. The runs relying on MinHash showed increased PC at the cost of PQ and RR. A novel concept of a double-pass pipeline is introduced, in which the baseline ER pipeline should be initially applied, followed by a more expensive error-corrective MinHash ER pipeline that can be used on Entities created by the first pass, combining the strengths of the investigated runs.

Keywords: Big Data, Dirty Entity Resolution (D-ER), Locality-Sensitive Hashing (LSH), MinHash, Entity Resolution (ER) Pipeline, N-grams

Contents

List of Figures

List of Tables

1 Introduction

The general trend is that more and more data is being created and collected each year, some even expressing it as a data tsunami [1, 17]. Many who deal with data collection are interested in techniques that can efficiently harness the information within the data [1, 7]. These large quantities of data, henceforth referred to as big data, are associated with different challenges [17]. The data is massive in size, making traditional databases not applicable, and the velocity of the data is too significant for traditional database schemes. In database management, a common practice is to merge multiple data sets and achieve a larger and higher quality data set [17]. Properly combining these data sets requires finding duplicates and merging these duplicates into a single entry, initially coined as the merge-purge problem [18]. In the merge-purge problem, advertisers bought databases containing personal addresses and joined them with their database [18]. If the person already existed in the original database, then joining the new database could create a duplicate entry for the given person, resulting in the problem that a single person could receive multiple instances of the same advertisement magazines [18]. Databases should be joined in a deduplicated manner, i.e., records referring to the same real-world object should be merged, removing duplications in the joined data collection. There are many different names to address this challenge: merge-purge[18], record linkage[23], entity matching[12], entity deduplication[6], or Entity Resolution (ER)[8, 15, 32].

Entity Resolution (ER) is more complex than it sounds due to how diverse the data between multiple collections could be. The data quality inside a database could be deficient for various reasons [23]. Entering the values into the system could introduce writing errors or differences in word selection [23]. Optional fields are only sometimes provided, making it difficult to rely on them for real-world scenarios. Missing fields for the users could result in data being appended into another field. The most important aspect of ER is using the data afterward. Creating links between different equal entities will provide information not available beforehand by increasing the knowledge base of the real-world entity represented within the data set [7]. ER generates data that is generally too expensive to acquire [7]. The generated data is information-rich and high-quality, and its generation is cost-efficient [7]. The comprehensive data is invaluable and can enhance other data-driven tasks [7]. Although much research has already been conducted in the field of ER, the application of ER on incoherent and incomplete big data is still in its infancy [23].

There are different focuses within ER. Data sets can be either clean or dirty; in the former, the data set has no duplications within the collection, while in the latter, the data set contains duplications [23, 36]. With these focuses, merging two data sets containing no duplicates is called Clean-Clean Entity Resolution (CC-ER) [38]. It is also possible to merge a clean set with a dirty set. Or even two dirty sets with each other. This research applies ER to a single dirty data set called Dirty Entity Resolution (D-ER), sometimes called entity deduplication.

ER is a quadratic comparison problem, as the idea is to compare each row with all the other rows of the collection [7, 8]. Comparing each entity pair will provide the highest deduplication factor possible. However, a complete pairwise comparison is only feasible for merging miniature databases as the quadratic time complexity will quickly become intractable. This research is conducted on the Account data set of Exact, consisting of approximately 300 million rows. Even though this size is already significant, it will grow faster after each passing year. Conducting comparisons between each unique pairwise combination will result in processing 45 quadrillion comparisons, taking several years of computation time while running large computational clusters, which is not feasible in time or money. Proposed ER solutions[19, 25, 28] for this data set

only utilized a subset and quickly became intractable due to the vast increase in entries.

Applying record linkage on two clean data sets yields a maximum number of comparisons equal to $C_{CC} = |DB_1| \times |DB_2|$ [6–8]. If record linkage were to be applied on a single dirty data set, then the maximum number of comparisons would be equal to $C_D = |DB| \times \frac{|DB|-1}{2}$ [6, 8], because comparing row_A with row_B, will have the same outcome as comparing row_B with row_A. Henceforth, the term 'ER' will refer to the single dirty data set scenario D-ER.

A matching function takes two different entities and returns a binary value of either 0 or 1 for a Negative- and Positive-Match, respectively. The matching function relies on expensive similarity measurements [36]. The most computation-intensive task is the comparison of entities and, therefore, the bottleneck of any ER task [11]. A blocking technique should be implemented to divide the data set into blocks of similar entities to reduce this bottleneck [7, 11]. By applying a blocking technique, only the entities within a block must be cross-compared, vastly reducing the number of comparisons required [7, 11]. Comparing only those pairs within the created blocks will have a time complexity of $O(n * b)$, in which b is the maximum number of entities in a bucket, and b should be much smaller than n ($b << n$) [7, 24]. If the collection can be split into smaller subsets, then it is feasible to execute quadratic time algorithms to achieve greater accuracy [18]. The main focus of the blocking should be to place different equal entities, i.e., referring to the same real-world entity, in a joint block [7]. The largest block will dominate the merging step's time requirement due to the quadratic comparison problem. Ideally, these blocks should be uniformly distributed [7, 18]. Choosing the perfect block size will always be a trade-off [7]. Since the blocking process is fully made-to-order, one can opt for larger blocks, resulting in fewer matches missed while costing more computational time to compare the additional matches [7]. On the other hand, one can use smaller fast blocks, which lowers the effectiveness of the blocks [7].

Matching two rows can be done via certain similarity features [7]. These features calculate, for example, the edit distance between two strings or the number of overlapping words [3, 23]. Generally speaking, ER is applied to find matches within the data. The algorithm has reached completion after these matches are discovered. However, after finding the Positive-Match, there is the option to merge these two entities and place them back in the ER pipeline [23]. The match-merge problem requires recalculating the steps required for matching until it finally results in a solution [23]. Pairs of rows that have not been found in joint blocks will not be matched with each other and are, therefore, placed in the Negative-Match category [7]. The advantage of doing so is that an extensive quadratic matching operation between all the different entities within a collection is not required [7]. Not all rows are compared with each other, resulting in missed Positive-Matches. Clustering methods can, and should, be implemented to retrieve these Positive-Matches based on the spatial location and connectivity between clusters of rows. However, clustering implemented on a data set with the mentioned characteristics needs to be investigated appropriately, meaning there is a gap in the currently available methods for clustering distributed big data.

The universal ER pipeline consists of four different phases. First, the data is preprocessed, in which the incoherent nature of real-world data is treated [8]. Secondly, the data is split into blocks, creating Matches of rows located in a joint block. Thirdly, these comparisons are classified as Positive- or Negative-Match. In the ideal situation, all the Matches should be Positive-Matches, meaning the block creation was perfect. However, in practice, this is never the case and should not be the true aim of blocking.

According to Christen et al., research on the blocking phase ER has two main focuses. The first is creating new or updating existing blocking algorithms [7]. The second focus is creating the most effective blocking keys [7]. Designing the blocking keys was a task for system experts, while state-of-the-art research was conducted on making that process automatic [7, 36]. However, it will require training sets that are most likely unavailable or very costly to generate with real-world data sets [7].

The utilization of Locality-Sensitive Hashing (LSH) throughout the ER pipeline will be investigated. LSH is an approximation algorithm in which hashes are generated for every row [7, 20]. Usually, when creating hashes, one wants to have as little overlap as possible, but in LSH, having as much overlap as possible between similar entities is the desired outcome. The rows are represented in fewer dimensions, making it more likely that similar rows are given equal hashes [37]. The similarities of different rows can efficiently be approximated with this technique, making it a perfect fit for application in big data scenarios [40, 41].

1.1. Problem Definition and Research Questions

This thesis applies a D-ER framework to create clusters of companies referring to the same real-world company based on Exact's[1] Account data set using Apache Spark's[2] distributed environment. Each step of the D-ER pipeline will be addressed and tackled as efficiently as possible while still yielding results of high correctness. The thesis will be a feasibility study on applying D-ER in a real-world incoherent and incomplete distributed big data scenario. The following Research Question (RQ) must be answered for conducting this feasibility study:

> **RQ:** Can Locality-Sensitive Hashing techniques help to improve the Dirty Entity Resolution efficiency and accuracy for incoherent and incomplete distributed big data scenarios?

Several Sub-Questions (SQs) are formulated to answer the main research question. To answer if D-ER can be applied in a tractable amount of time while still having high correctness, three phases of ER have their own dedicated SQ, followed by a final SQ regarding the generated deduplicated data set.

Sub-Question 1

> **SQ1:** Can the blocking phase of Dirty Entity Resolution significantly reduce the number of required comparisons associated with big data, making it tractable while still producing blocks with high Pairs Completeness and Pairs Quality?

The blocking phase aims to remove all incorrect comparisons before the matching phase, causing a significant improvement in time requirement. However, the question of SQ1 is, can a blocking scheme be applied to distributed big data that is incoherent and incomplete? Moreover, do the blocking schemes allow for creating blocks high in unique True-Matches and with limited unique False-Matches?

Sub-Question 2

> **SQ2:** Can the matching phase of Dirty Entity Resolution correctly classify comparisons between incoherent and incomplete rows?

The matching phase will classify comparisons as either referring to the same real-world company or not. However, users provide the data of the investigated data set, causing it to be incoherent and incomplete, creating this SQ, SQ2, if it is feasible to classify the comparisons of the blocked data set correctly.

Sub-Question 3

> **SQ3:** Can the clustering phase of Dirty Entity Resolution be applied to cluster distributed big data with high correctness?

The distributed nature of the data could cause complications with using established clustering methods. For this SQ, SQ3, it is unclear if clustering can and should be applied due to this distributed nature.

Sub-Question 4

> **SQ4:** Can the created deduplicated data set be used as a solution in existing models to improve their results?

The last SQ, SQ4, is not regarding the feasibility of applying D-ER; it tries to answer the latter high correctness part of the RQ. If the resulting deduplicated data set is high in quality, then it should improve existing models of Exact based on the Account data set.

[1] Exact (exact.com)
[2] Apache Spark™ (spark.apache.org)

1.2. Methodologies

All the ER phases were applied to answer the research questions, which are based on several fields of the Accounts data set, which are: Name, Website, Postcode, Email, Phone, City, State, AddressLine1Street (AL1S), and AddressLine1Number (AL1N). These fields are utilized to create blocks within the data set, making the data set tractable, which allows to answer SQ1. In the matching phase, these fields are used to check if two rows refer to the same real-world object, allowing to answer Sub-Question 2 (SQ2). In the clustering phase, similar clusters resembling the same real-world company will be merged. The completion of the clustering phase marks the completion of the ER pipeline, which provides an answer to SQ3.

The outcome of the ER pipeline is a data set that maps the original ID to a newly created EntityID. Rows that refer to the same real-world company should have the same EntityID. To investigate the correctness of these EntityIDs, the results are implemented in an existing Late Payment Prediction (LPP) model predicting if an instance of Accounts will pay its invoices on time. The model relies on the Accounts data set, allowing the deduplicated data set to be used without any changes to the model itself. This model and its outcomes will answer the final SQ, SQ4. The main research question can be answered after all these SQs are answered in succession. The hypothesis is that the outcome of LPP will be improved when using the deduplicated Account data set. Because merging Accounts removes a cold-start problem associated with new Accounts, historical data of other Accounts referring to the same real-world company can be used. Also, it ensures the most up-to-date information of a real-world company within the Exact's data sets can be found.

1.3. Contributions

This research has multiple contributions to the ER field. Even though ER is an established field, there needs to be more information available on this kind of data set; most written research is available on standardized data sets, which are small to medium in size with complete information and are not distributed. This research hopes to fill the gap in available information on the matter. The main focus of the research is a feasibility study, finding out whether it is possible and making this study open-ended. New methods can be investigated and developed on this base framework. The main contributions are listed as follows:

- A blocking schema dependent on LSH is applied and studied.
- Different additions to MinHash are proposed and used within this thesis; these include hash manipulations and novel concepts for token creation.
- A novel multi-model classification setup is applied to classify comparisons.
- A novel concept of using LSH to efficiently transform a graph into clusters using spatial information of the graph itself.
- Multiple distinct benchmark runs on a data set of incoherent and incomplete distributed big data. Applying ER to this kind of data set is poorly studied in the literature. This conducted study is a solid contribution and could serve as a sound basis for similar studies.
- Extensive future work possibilities are addressed, which could all be investigated in future studies.

1.4. Report Structure

This report is structured with multiple chapters. First, the related work is investigated in Chapter 2. Afterward, the concept of ER is explained in Chapter 3. Each of the phases of the ER pipeline will be documented in a separate chapter, starting with the preprocessing phase in Chapter 4. In this chapter, the data set of Exact is taken under the loop, what it consists of, and how it should be preprocessed. In the following chapter, Chapter 5, the blocking phase of ER, a significant part of the ER pipeline, will be discussed in-depth. This elaborated chapter discusses how to create blocks within the data set. Afterward, the matching phase is discussed in Chapter 6. This chapter will classify the Matches as Negative- or Positive-Match, filtering the data within the created blocks. The final chapter of the ER pipeline is the clustering chapter, as can be read in Chapter 7. Novel clustering methods relying on LSH are discussed in this chapter. The created data set is applied to an existing Exact prediction model in Chapter 8; the outcome of this application is discussed in this chapter. Finally, Chapter 9 presents this study's conclusion, and Chapter 10 provides the final discussion and ventures for future work.

2

Related work

In this chapter, we investigate the related work. We are starting in Section 2.1 with what has been done by prior research on the Exact data set. Afterward, a brief overview of research papers on the ER pipeline is listed in Section 2.2. Section 2.3 is a small section dedicated to noteworthy preprocessing practices. Next, we investigate different methods to block the data set in Section 2.4. Followed up with Section 2.5 on measuring the effectiveness of the applied blocking methods. The pruning of blocks and rows is investigated in Section 2.6 and Section 2.7, respectively. Afterward, the methods for comparing texts are found in Section 2.8. Finally, we present existing studies on clustering approaches in Section 2.9.

2.1. Exact's Prior Research

In Exact, there exist three preceding conducted studies with a similar focus and using the same data set used in this thesis [19, 25, 28]. Three prior publications worked on the same Exact data set, and this research will build upon what has already been achieved. In the work of Hovanesyan (2019), it was shown that ER is a crucial aspect in order to work with the data. The data is heterogeneous, as the data are real-world entries with only a required Name, and the other properties are optional to provide. An important aspect is the preprocessing of the data. Cleaning the data and allowing it to fit a more strict form. Afterward, blocking was applied to make their ER algorithm tractable. It was noticed that once these blocks become too large, the algorithm will become unscalable. In order to work with the data, they opted only to use entries with either a Chamber of Commerce (CoC) or VATNumber, which is roughly 16% of the total size [19]. Each entity has a Blocking Key Value (BKV) corresponding to their CoC or VATNumber. Entries that have both are placed in two blocks. The downside of relying on optional data to block the data set is that most of the valuable data is unused.

Two years after the finalization of Hovanesyan's MSc thesis, Kostense (2021) utilized Hovanesyan's proposed algorithm. However, the data set had already more than doubled in size, and it was deemed that the proposed algorithm was too complex and not scalable enough to be run in a tractable time frame [25]. Kostense contributed by making a lightweight algorithm. Kostense had the clever idea to utilize the type of the Accounts field. Users of Exact create their own unique type D Account in Accounts representing their company. Since the type D Account is bound to a unique real-world company, it is not required to match type D Accounts with other type D Accounts, as it will always result in a False-Match. It is only required to match each type A Account with a single type D Account. Three distinct techniques were applied to create a multi-attribute ER algorithm: Matching on CoC, matching on VATNumber, and fuzzy matching on Postcode and AL1N in combination with Email. Multiple type D Accounts may share the same CoC, VATNumber, or even a Postcode, AL1N, and Email tuple. For this reason, Kostense agreed with system experts to drop entries of type D that shared a common field more than ten times [25]. Not all rows of the Accounts set are used in the ER pipeline, as a row must have one of these three BKVs to be indexed in at least one block.

The latest MSc thesis by Mao (2022) tries to embrace the entire data set; entities that have few provided fields still need to be matched [28]. The Postcode was used as their blocking algorithm's only BKV. The reasoning is that the majority (roughly 80%) of the entities have a Postcode provided. Only losing roughly 20% of the entities is already a significant improvement. The blocks created based on these BKVs are disjointed as only a single Postcode is provided per row. After dividing the data set into disjoint blocks, graph merging is

		Entity Resolution Phase		
Paper	**Year**	**Blocking**	**Matching**	**Clustering**
Hernández and Stolfo [18]	1995	✓		
McCallum et al. [29]	2000	✓		
Bilenko et al. [3]	2003		✓	
Chaudhuri et al. [5]	2003		✓	
Christen et al. [7]	2007	✓		
Christen [6]	2011	✓		
Draisbach and Naumann [11]	2011	✓		
Papadakis et al. [33]	2011	✓		
Papadakis et al. [34]	2012	✓		
Fisher et al. [15]	2015	✓		
Papadakis et al. [35]	2015	✓		
Christophides et al. [8]	2019	✓	✓	✓
Li et al. [27]	2020	✓		
Papadakis et al. [37]	2020	✓	✓	
Jafari et al. [21]	2021	✓		
Niknam et al. [32]	2021	✓		

Table 2.1: Extensive research papers on various phases of the ER pipeline

applied. Experiments within graph merging experiments were conducted. In these experiments, edges found in a joint block were tested by merging them randomly vs merging them by a method with a threshold. The experiments show that matching inside the created blocks based on methods outperforms the merging of random edges, showing the need to clean the generated blocks.

2.2. Paper Overview

Multiple survey papers are read to gather significant knowledge on constructing the ER pipeline. Important papers to gain insights are listed in Table 2.1. The table links the papers to which part of the ER pipeline they cover. The preprocessing phase is excluded as it is made to fit per data set and is poorly represented in the significant papers. The selected papers offer a wide range of cutting-edge implications within ER. Noticeably, the papers investigating the matching phase have been around for quite a while. These papers are not necessarily dedicated to the ER problem but can still be implemented in an ER pipeline. The papers dedicated to the ER problem mainly examine the blocking phase of the ER pipeline. Most research is done on smaller data sets, allowing a blocking stage that essentially can create a near-perfect block of entities all referring to the same real-world entity. Using a smaller data set has the effect of making the matching and clustering stage unnecessary. There is a significant gap in knowledge in the clustering phase; more research needs to be done on clustering concerning the ER pipeline. The clustering phase is heavily dependent on the workings and outcome of the previous phases, which can cause each situation requiring clustering to be unique.

2.3. Preprocessing

The preprocessing phase of the ER pipeline is mainly left out of scientific research, as it is an often deemed trivial step [2]. Hernández and Stolfo used a database in which synonyms of names were stored [18]. In the real world, there are numerous manners to express the same thing. For their input, they also applied a spelling correction program on top of the data, allowing them to correct spelling mistakes and improve their final matching percentage [18]. Applying such a correction beforehand could make using spelling-correcting indexing methods unnecessary.

2.4. Indexing Algorithms

Comparing all the entities will always give the optimal result; however, in a big data scenario, it cannot be calculated in tractable time complexity as entity comparisons suffer from quadratic complexity [7, 8, 11, 24, 35]. The most common way to tackle this problem is to block the collection into multiple disjoint subsets [35]. Splitting or blocking the data set into blocks will lower the number of comparisons at the cost of the quality of the indexing algorithm [7, 35]. Some rows of True-Matches are not placed in a common joint block and, therefore, no longer compared [7]. There are many different manners of creating blocks from a collection. The newer techniques build on top of a predecessor. Knowing how the more advanced blocking techniques came into existence relies on knowing the rudimentary predecessors of a given blocking technique. Even though not all techniques are applicable anymore on a big data set, it still is possible to use specific indexing algorithms for underlying structures within the data set in a divide-and-conquer methodology. The blocking algorithms rely on BKVs, a field carefully crafted by domain experts to index the rows efficiently. The majority of the blocking schemes also require parameters that need to be fine-tuned [34]. Research has been conducted in creating the BKVs and setting the parameters via machine learning, relieving the need to have domain experts thoroughly investigate an implementation [34].

The most important characteristics of the blocking algorithm are twofold [6, 8]. Firstly, the algorithm should separate the collection into blocks of closely related entities. Secondly, the number of entities in the blocks should be as minimal as possible. The former provides information on the fitness of the blocking scheme (effectiveness), and the latter depicts if the algorithm will be tractable for computation (efficiency). The problem is that it is not possible to maximize both criteria for a given blocking algorithm [8]; there is always a trade-off. If all the entities are matched in a large common block, comparing them will take more time [6]. On the other side, if all the entities are divided into multiple smaller blocks, comparing them will take less time, but the effectiveness will be lower [6].

Each blocking method is different, according to Christophides et al., they can be classified using the following taxonomy in Table 2.2 [8]. In this project, the data is structured; therefore, only scheme-based indexing algorithms will be utilized, as these algorithms have a higher efficiency and effectiveness than scheme-agnostic indexing algorithms [31]. The algorithm could either be partitioning or overlapping [8]. In the former, the entity collection is divided into partitions by using a single key from an entity. In the latter, multiple keys can be constructed from an entity placing it in multiple blocks, causing an overlapping effect. One of these methods is not necessarily better than another; both can be used depending on the application of the blocking algorithm. Overlapping will create more keys, allowing it to handle noisier data better [7]. The downside is that it will create more comparisons. In general, however, overlapping blocking algorithms tend to create more robust blocks regarding efficiency and effectiveness [6, 8].

Standard Blocking (1969) In Traditional or Standard Blocking, each row in the collection gets a single BKV assigned, and the collection is partitioned on their BKVs [6, 8, 14]. This results in partitioned blocks, allowing for higher efficiency but lower effectiveness. Real-world data that is generally noisy will likely fail to be partitioned into the same block. Standard Blocking is the technique on which all the blocking algorithms rely; the newer techniques try to improve the traditional blocking by extending its method [35]. With Standard Blocking, it is possible to generate overlapping blocks using multiple Standard Blocking steps, creating different BKVs per row [35].

Sliding Window (1995, 2007) In the Sliding Window [18], sometimes called Sorted Neighborhood Indexing, the entities are sorted on their BKV [6]. Creating the row's BKV in a Sliding Window is the same as in Standard Blocking [35]. Afterward, a window is used over the collection, and each entity in a given window is put in a block [18]. The algorithm relies on how well the BKV can be sorted in order to have an effective neighborhood. For example, if names were used as BKV, then spelling mistakes at the end will more likely result in a joint subset while spelling mistakes at the start will most likely result in a disjoint subset [7]. Multiple BKVs per row will increase the chance that different word spellings will still result in the entities being in joint blocks [7, 11]. The algorithm's accuracy for smaller window sizes is insufficient [18]. For the Sliding Window to retrieve a high accuracy, a more extensive set of neighborhoods needs to be compared, which defies the point of indexing [18]. Smaller window sizes benefit from fewer False Positives (FPs), thus increasing the accuracy [18]. There are two approaches to this indexing technique [6]. The first is the sorted array-based approach; each entity corresponds to a unique windowing row in the collection [18]. The second approach is the inverted index-based or extended sorted neighborhood approach [7, 35]. In this approach, the entities are grouped

Schema-awareness	
schema-based	Keys can be created on specific fields on the structured data
schema-agnostic	There is no underlying scheme, the data is unstructured
Indexing Function Complexity	
atomic	Single key is generated to use in the indexing function
composite	Generating multiple keys per row to use in the indexing function
Indexing Function Definition	
hash-based	Function uses hash generation for the creation of the BKVs
sort-based	Function uses sort method to generate the BKVs
learning-based	Function uses a machine-learning method to create BKVs
Redundancy attitude	
partitioning	Creation of disjoint blocks
overlap-positive	Creation of joint blocks in which the number of blocks in common is corresponding to show equality
overlap-neutral	Creation of joint blocks, equality of rows is independent on the number of blocks in common

Table 2.2: Taxonomy of Blocking Methods

by their BKV, and each unique BKV corresponds to a windowing row. The sorted neighborhood is an expensive indexing algorithm due to the required sorting step [18]. This inverted index approach achieves higher performance [35].

N-gram Blocking (2001) In this blocking algorithm, the BKVs are altered to achieve higher robustness against noise in the BKV [35]. This algorithm changes the BKVs to a N-gram representation [16]. These are subsets of the key with a size of N. Each N-gram will form a new block [16, 35]. For example, the BKV of 'Exact' with $N = 2$ will create {'Ex', 'xa', 'ac', 'ct'} as its token set. There is an extended version of N-gram blocking. In this version, the N-grams of a single BKV are combined and used as the final key [35]. To combine the N-grams, the number of N-grams per token is calculated as $l = \lfloor k \times t \rfloor$, with k being the number of N-grams from a BKV and t is a threshold [6]. If a threshold of 0.9 is applied in the example, then the number of N-grams per token will be $\lfloor 4 \times 0.9 \rfloor = 3$ [6]. All tokens that are possible to be created containing three N-grams are as follows: {('Ex', 'xa', 'ac'), ('xa', 'ac', 'ct'), ('Ex', 'ac', 'ct'), ('Ex', 'xa', 'ct')}. To create blocks from these tokens, the N-grams are merged, resulting in {'Exxaac', 'xaacct', 'Exacct', 'Exxact'} as the set of BKVs. If $l = 2$, the same algorithm recursively applies to the sublists created with $l = 3$ [6]. Applying the extended version will make the blocks more correlated and smaller [35]. However, if the threshold is low and the length of the BKV is large compared to q, the number of tokens increases exponentially.

Canopy Clustering (2000) Similar to N-gram blocking, but instead of creating blocks purely on equality, it creates blocks based on similarity [35]. The data is clustered into overlapping subsets in canopy clustering. These clusters are based on their spatial location [29, 31]. First, overlapping subsets are created using simple comparison algorithms that do not require a significant computational cost [29]. A method that can achieve this is based on the inverted index; each unique word represented in the row will be attached to all the rows in which the word is present [29]. These lists can be used to efficiently cluster the rows, as when the rows have no words in common, they will not be compared. These created subsets are called canopies, and each row has a certain distance above a threshold from the central point of the canopy [29]. Every entity needs to be in at least one canopy, and entities can be in multiple subsets; therefore, canopy clustering generates overlapping clusters [29]. After creating these canopies, algorithms with higher computation costs can be applied, as it is not required to do pairwise comparisons between every entity, only the entities within a canopy [29]. Canopy Clustering applies both the blocking and matching stages in a single method.

Suffix Array (2005) In suffix array, tokens are created from the BKV [6]. These tokens are subsets of the original BKV by removing the left characters to a minimum length of l_m [6, 35]. Creating multiple tokens from a single BKV has the benefit that the scheme is more tolerant of noise in the data. For example, the BKV 'Exact' with $l_m = 3$ will create {'Exact', 'xact', 'act'} as its set of tokens. The number of tokens created per BKV is $T_{SA} = max(1, l_k - l_m + 1)$, given the length of a BKV as l_k. Creating blocks based on these tokens will result in a very skewed distribution of the entities. To address this problem, the blocks with proportionally too many entries are discarded [6, 35]. Noticeably, in the example, only noise at the left of the BKV will be addressed [6]. Extended Suffix Array is created to help overcome noise at every location of the BKV [35]. This is done by creating all the possible tokens of a BKV with a length larger than the set minimum of l_m. If the example were to be applied for the extended suffix array, then {'Exact', 'Exac', 'xact', 'Exa', 'xac', 'act'} would be the set of tokens. The number of tokens generated per BKV is $\sum_{n=1}^{T_{SA}} n$ [35]. Entities with a large l_k compared to l_m will create many tokens and are placed in many different blocks. Limiting the maximum length of a BKV is advisable to overcome this issue. Like the standard Suffix Array, the most frequent tokens are discarded based on a set limit [35]. The Extended Suffix Array method resembles the N-gram blocking. The only difference compared to N-gram is that only the tokens with a maximum length of l_m are added [35].

Locality-Sensitive Hashing (2000) LSH is an approximated algorithm in which a higher dimensional input space is mapped to a lower dimensional space [21]. A data indexing solution can be constructed for search spaces with a low dimensionality. However, these solutions quickly become unscalable for the high-dimensional search space due to the curse of dimensionality [21]. To circumvent this curse, approximations can be used to retrieve solutions that are near the exact solution but consume only a fraction of the otherwise required resources. The nature of the solutions given by the LSH approach is beneficial in spaces that do not require an exact solution [21]. LSH is an Approximated Nearest Neighbor (ANN) algorithm and, according to Jafari et al., the most popular one [21].

The concept of LSH is to use randomly created n hash functions to assign entities to a unique hash bucket created from these hash functions [21]. High-dimensional entities referring to the same real-world object will have some characteristics in common and are located in the same neighborhood. Applying a hash function, which maps the entity to a lower dimensional representation, retains this relationship [21].

There are two methods of applying LSH. The first option is to use Bucketed Random Projection (BRP). For this method, dense vectors must represent the data set, which can be done by training a Word2Vec representation. The other method, MinHash, relies on sparse vector representations; these sparse vector representations can be created equal to other blocking techniques, such as N-gram blocking.

In HARRA[23], an LSH MinHash approach is used to block data sets, and they implement their approach called Iterative-LSH, in which LSH is iteratively applied to large blocks, which will split, depending on the elements inside each block, the large blocks into smaller blocks. Eventually, this results in hash blocks that will be similar in size [23].

LSH can be used as the first method in canopy clustering, creating canopies with a low computational cost. Afterward, a method with more computational cost can clean the canopies of invalid edges [29].

2.4.1. Indexing Algorithms Classification

The indexing algorithms of Section 2.4 can be classified using the taxonomy of Table 2.2. The classification of the researched indexing algorithms is shown in Table 2.3. Since it was known that the data was structured, only schema-based indexing techniques were required. The majority of the indexing techniques are based on a composite complexity. All except one technique generates the sets based on hash, and only one sorted indexing technique is investigated. No learning-based blocking techniques will be investigated; this study is a feasibility study. More straightforward techniques with low cost should be investigated first. Learning-based methods could be applied after ER for this type of data set is deemed feasible for the given application. All blocking techniques except Standard Blocking will create overlap between blocks. For LSH and N-gram blocking, the more blocks two rows have in common, the more likely they refer to the same real-world entity. For the overlap-neutral methods, a single block in common already means that they likely refer to the same real-world entity.

2.5. Measuring Block Effectiveness

The effectiveness and efficiency of the blocking algorithm can be evaluated with the following three established measurements: Pairs Completeness (PC), Pairs Quality (PQ), Reduction Ratio (RR) (Eq. (2.1), Eq. (2.2),

	Indexing Function Complexity		Indexing Function Definition		Redundancy Attitude		
Indexing Technique	atomic	composite	hash-based	sort-based	partitioning	overlap-positive	overlap-neutral
Standard Blocking	✓		✓		✓		
Sliding Window	✓			✓			✓
N-gram Blocking		✓	✓			✓	
Canopy Clustering		✓	✓				✓
Suffix Array		✓	✓				✓
LSH		✓	✓			✓	

Table 2.3: The classification of the investigated indexing algorithms

Eq. (2.3), respectively) [35]. In Pairs Completeness (PC), the recall is measured, which is the number of matches in the blocked data set depicted as $|D(B)|$ divided by the number of matches of the complete data set $|D(C)|$. In Pairs Quality (PQ), the precision is measured, and the precision in blocking means the number of matches we have in $D(B)$ divided by the total size of the blocked data set depicted as $||B||$. Finally, we compare computation costs in the Reduction Ratio (RR), which measures the computational cost of the comparisons in the blocked data set versus the complete set. Since the focus is on Dirty-ER, the number of comparisons in the complete set equals $|C| \cdot (|C|-1)/2$. Each of these measurements will give a fraction in the $[0,1]$ range, allowing us to compare different blocking methods with each other in terms of effectiveness (PC) and efficiency (PQ, RR), which are the measurements depending on $||B||$. The aim is to maximize the blocking algorithm's effectiveness and efficiency [35]. However, since blocking reduces the total number of comparisons required and comparing all the entities will always result in the highest possible recall, it is a trade-off between effectiveness and tractability [35]. According to Papadakis et al., achieving a high PC PC (> 0.8) is essential. Without a high PC, the next methods in the ER pipeline cannot retrieve numerous matches [35]. It would be unfair to place an algorithm that perfectly blocks but takes cubic time to evaluate as the best possible solution; therefore, blocking time should also be compared [35].

$$PC = \frac{|D(B)|}{|D(C)|} \qquad (2.1) \qquad PQ = \frac{|D(B)|}{||B||} \qquad (2.2) \qquad RR = 1 - \frac{||B||}{||C||} \qquad (2.3)$$

Essentially, PC and PQ resemble recall and precision, respectively. Balancing these measurements is important to find the optimal solution since these two factors are inherently caught in a trade-off. The F-score can be utilized to find a balance between these factors, as provided in Eq. (2.4). In this equation, β is a value to address that recall is β times more important than precision. In standard cases, a value of 1 is used for β, as this is a fair comparison between recall and precision since both are equally important. However, in the case of ER and especially in the blocking phase, it is advisable to have a high recall at the cost of having a lower precision. The blocking pipeline is the ER pipeline's first computational major alteration step. If pairs cannot be found in this step, it will be more challenging to cluster the matches in the clustering phase of the ER pipeline. It is also possible to calculate F-scores between PC and RR, as RR and PQ are both considered to be the efficiency of the blocking phase and thus have the same trade-off relationship with PC.

$$F_\beta = (1+\beta^2) \cdot \frac{precision \cdot recall}{(\beta^2 \cdot precision) + recall} \qquad (2.4)$$

2.6. Block Pruning

Fisher et al. researched ER blocking (indexing) techniques [15]. They applied the standard blocking technique, transforming rows into a single BKV and blocking the collection based on these BKVs. According to Fisher et al., the blocks should have minimum and maximum sizes [15]. If the block size is smaller than the minimum, it should be merged with a block nearby in Euclidean space based on the BKV. The creation of their BKVs consisted of three steps. The first step is to block the collection based on the first two characters of the name. The second step is split on the Soundex[10] representation of the surname. Soundex is a phonetic

resemblance of a word. The final step is to add the last four digits of the ZIPcode. In between these steps, the blocks are split into ($B-$, $B*$, and $B+$) based on their size compared to the minimum and maximum allowed block size. In which $B-$ has fewer than the minimum rows within a block, $B*$ is within the allowed range, and $B+$ has more rows than the maximum allowed. The $B-$ blocks should be merged with the $B*$ blocks. This merging is done by creating a priority queue of descending similarity between the $B-$ and $B*$ blocks. Only the $B+$ blocks will continue to be blocked on the next step in the blocking process in order to get these $B+$ blocks in the desired $B*$ range. Eventually, blocks are within the specified range of allowed block size, or the algorithm runs out of possible blocking keys. The final results are blocks of multiple similar BKVs. This applied method cannot rely on incomplete data, as this would mean that rows with different fields available are not a match.

The created blocks are not uniform in their size. The larger the block is, the larger the clique inside the block, and the number of comparisons required per block grows exponentially to the size of the given block. Therefore, the largest blocks will consume the most resources. There are some options to mitigate the resource dependency of large blocks. These options are either static or dynamic [37]. In static blocking, the block cleaning methods do not depend on the matching results, while in dynamic block cleaning, the method is mingled within the matching phase [37].

The idea behind the static methods is that large blocks have less valuable information within them [37]. These blocks could be created during the blocking phase because of common stopwords [37]. Blocks that rely on stopwords primarily do not refer to a single real-world entity but rather to multiple real-world entities, as most real-word entities can contain stopwords that are shared with other real-world entities. These largest blocks could be discarded without negatively impacting the matching phase [8, 37]. With overlapping blocks, the matches in the largest blocks could still be found in the other overlapping blocks while significantly reducing the number of required computations. For the partitioning blocking methods, removing the largest blocks will cause these rows not to be matched with any other row.

It is also possible to have blocks that are near a fixed size. Creating blocks of a fixed size can be done by splitting large blocks into smaller blocks if they surpass a specified maximum size. The blocks below a specified minimum size can be merged with other smaller blocks. Merging of small blocks should only be applied if these blocks have the majority of rows in common or if their BKVs are nearly identical.

The final static method transforms the blocks into a graph in which the blocks are vertices. Edges are made between the vertices based on their BKV and the similarities between the different blocks; if they are similar above a specified threshold, an edge is created between the nodes with various agnostic functions used to give the edge weight [37]. The edges are then sorted on their weight to combine the highly interconnected blocks while skipping the block pairs with low connectivity [37].

Dynamic methods can also be applied to clean the blocks. In Iterative Blocking, these rows are merged once a True-Match is found between two rows, r_1 and r_2 [37]. Merging rows has the benefit of expanding information; Rows previously determined to be a Negative-Match with r_1 and r_2 can be matched again with $r_{1,2}$ and possibly result in a Postive-Match [37]. Expanding the knowledge base of single rows within the matching process can yield a larger PC. The matching phase is concluded when no Positive-Matches can be found. This method can only be applied to D-ER, as the data set has many duplications. Using Iterative Blocking for CC-ER yields no improvement; at maximum, only a single duplication exists per real-world entity. Duplications should be located and merged as early as possible to negate redundant comparisons [37]. Blocks should be weighted on the probability of Positive-Matches in the block. Blocks with the highest probability of containing Positive-Matches should be processed first; This process is called Block Scheduling [37].

2.7. Row Pruning

Instead of pruning the blocks, it is also possible to prune comparisons between entities that are unlikely to be True-Matches of each other. The method of performing such an action is to inverse the index of the blocked data set, in which the RowID is the primary key, instead of the BlockID. By swapping the primary keys, it is possible to retrieve the number of blocks rows have in common [37]. Comparisons between two rows with less than a given threshold of blocks in common are removed from the pool, as these comparisons are not likely to be a Positive-Match [37]. Cleaning the comparisons will positively impact the PQ and RR, with a marginally negative impact on PC [37].

A comparison between two rows should, at maximum, only be done once. The simplest method to discard these duplicated comparisons is comparison propagation [33]. Discarding duplicated comparisons can be performed by changing the structure of the blocks to a graph structure. In this graph structure, the rows are

vertices, and the comparisons are the edges. Performing this transformation will automatically discard duplicated comparisons, as only a single edge between two distinct vertices can exist. The number of duplicated comparisons can also be the weight of the vertices [8]. If there are multiple overlapping comparisons between two vertices, they are more likely equal than if a single comparison existed between them. The weight will, therefore, give an evidence estimator of probable likeliness to the edge [8]. Building up the Matched data set should be done by first performing all the comparisons that have a high likelihood of being a Positive-Match, i.e., in which the vertices have a high weight assigned to them. It is even possible to discard all the entities with low evidence of being a Positive-Match [8]. According to Christophides et al., these are the pruning methods possible:

1. Weighted Edge Pruning (WEP): Remove all edges with a weight lower than the given threshold [8, 37].
2. Cardinality Edge Pruning (CEP): Keep the top K edges with the highest weight and prune the edges with a weight lower than the K^{th} weight [8, 37].
3. Weighted Node Pruning (WNP): For each node neighborhood, remove the edges with a lower weight than the threshold [8, 37].
4. Cardinality Node Pruning (CNP): Keep the top K edges with the highest weight for each node neighborhood [8, 37].

Removing entities from a block instead of the edges themselves is also possible. In Low Entity Co-occurrence Pruning (LECP), the entities with the lowest average edge weights, among others, are pruned [37]. LECP can always be applied, or it can be applied only on the largest blocks, removing entities from the block that are poorly connected, resulting in a manageable block in size to perform comparisons on [37]. These large blocks could also be split into smaller blocks by finding communities and creating new blocks out of them [9, 37].

According to Papadakis et al., no block pruning method will consistently outperform the other methods [37]. Instead, combining multiple block pruning methods can create a more robust block pruning process [37]. However, combining multiple row pruning methods is not possible [37]. Only a single row pruning method can be part of the cleaning process [37]. Generally, these row-pruning methods have a higher computational cost than block-pruning methods [37].

2.8. Fuzzy Matching

Different rows need to be linked together; this process is called record linkage. It is impossible to compare two bodies of text directly with each other; they need to be transformed into values that a machine can interpret. The distance or similarity between texts can be measured in different ways, called fuzzy matching. In fuzzy matching, the texts are transformed into feature values, which correlate to the similarity of the two texts in question. These features are in the range of 0 to 1, dissimilar to similar, respectively.

In most ER implementations, the correctness of the applied record linkage needs to be investigated. According to Niknam et al., most ER blocking evaluations rely mostly on whether equal entries are placed in the same bucket, regardless of the linkage[32]. However, they claim that blocking in combination with linkage plays a significant role in ER and should be evaluated together.

2.8.1. Edit-Distance Measures

The number of transformations required to transform one string to the other corresponds to the distance [5]. This distance can be used to compare two strings, s_1 and s_2, with each other [3].

Levenshtein Levenshtein similarity relies on strings [3, 26]. It uses the Levenshtein distance, the number of character transformations required to transform one text to the other. In Levenshtein, The allowed transformations are addition, substitution, and deletion [26].

Jaro similarity In Jaro similarity, three parts make up the similarity. First, the characters in common are calculated for the two strings s_1 and s_2, denoted as m. The characters in common are then divided by the length of the two strings separately. Secondly, the transposition is calculated, which is done by calculating the edit distance on the substrings of the characters in common for both strings. The transposition t is the

halved number of the edit distance required. These three similarities are summed and divided by three. The Jaro similarity equation is shown in Eq. (2.5).

$$\text{Jaro}(s_1, s_2) = \begin{cases} 0 & \text{if m} = 0 \\ \frac{1}{3}(\frac{m}{|s_1|} + \frac{m}{|s_2|} + \frac{m-t}{m}) & \text{otherwise} \end{cases} \tag{2.5}$$

Jaro-Winkler similarity The same Jaro similarity is used in Jaro-Winkler, but there is an addition to the prefix both strings have in common. Winkler utilizes the similarity of the start of both words. The prefix similarity, denoted as l, is measured starting from the first character up to four characters [3]. The prefix similarity is divided by a constant factor; the standard is 10. Afterward, it is multiplied by the inverse of Jaro. The Jaro-Winkler similarity is shown in Eq. (2.6). The Jaro-Winker similarity is widely used in ER applications [39]. According to Bilenko et al., the similarities between Jaro and Jaro-Winkler are intended for smaller strings [3]. Their research showed that Jaro-Winkler performed worse than Jaro for the given data set [3].

$$\text{Jaro-Winkler}(s_1, s_2) = \text{Jaro}(s_1, s_2) + \frac{l}{10}(1 - \text{Jaro}(s_1, s_2)) \tag{2.6}$$

2.8.2. Token-Based Measures

Two texts are split into sets of tokens, S_A, S_B; afterward, these sets can be compared, and the similarity can be calculated [3]. The number of tokens in common directly represents the similarity. An advantage of token-based measurements is that the similarity is independent of word order.

Jaccard Jaccard similarity relies on sets of tokens. With Jaccard similarity, the similarity of these two sets of strings is calculated as shown in Eq. (2.7).

$$\text{Jaccard}(S_A, S_B) = \frac{|S_A \cap S_B|}{|S_A \cup S_B|} \tag{2.7}$$

TF-IDF In Jaccard similarity, all the words have the same weight, independent of their rarity. Jaccard essentially acts as the Term Frequency (TF). A weight that will be lower if it is a frequent term can be assigned to this TF. Inverse Document Frequency (IDF) is the number of rows where the term occurs, but the inverse is taken to give the most frequent terms the least weight. Often, TF-IDF performs better than methods that do not consider term frequency, but not always [3].

2.9. Entity Clustering

As shown in Table 2.1, more research is needed on clustering the resolved entities. Only Christophides et al. have a section dedicated to this phase [8]. The missed connections between different created entities can be found with a clustering phase, in which implicit connections between different vertices are made [8]. Transforming the graph into a set of disjoint sub-graphs, also known as Connected Components, can be used to infer these implicit relationships [8]. Noise within these Connected Components is troublesome as they will merge vastly different sub-graphs if a single noisy link exists between them, causing the recall to go up but diminishing the precision [8].

Different graph nodes can be selected as the center of a cluster, and then other nodes are connected to their nearest center [8]. This application can be improved by merging cluster centers that are similar together [8]. Another method is Star Clustering, in which all the vertices are put on a stack, sorted from highest to lowest degree [8]. Then, pop the vertex with the highest degree of the stack and create a cluster of the said vertex and all of its neighbors. These neighbors are also removed from the available stack. Continue until the stack is empty and disjoint clusters are generated. After Star Clustering, it is possible to apply additional clustering techniques to merge the disjoint clusters and find more implicit connections [8].

3

Entity Resolution

In this chapter, we will concisely show the ER pipeline and its division into multiple phases: preprocessing (Section 3.1), blocking (Section 3.2), matching (Section 3.3), and clustering (Section 3.4). The concepts of these phases are written here, and the elaborated application of each phase will be described in their own chapter. A depicted overview of the pipeline process is given in Fig. 3.1. To measure the performance of the ER implementation, a Ground Truth (GT) is required, which is discussed in Section 3.5.

3.1. Preprocessing

The first step in the ER pipeline is the preprocessing of the data. In this thesis, unprocessed (raw) data suffers from several issues commonly reported for big data in most real-world scenarios [18]. Since the data is not uniform, the same field can be created in many different manners. Missing parts, misspellings, synonyms, and additional information can cause inconsistencies within the data [18]. If the subsequent phases are generated from unaligned data, then it will cause similar entities to be placed in disjoint clusters [7, 18]. Therefore, it is essential to apply a well-defined preprocessing phase. The fully detailed preprocessing phase is explained in Chapter 4.

3.2. Blocking

Since comparing each row with all the other rows is infeasible, the data is divided into multiple blocks. Similar rows should be mapped to the same block. Blocking is a significant aspect of the ER pipeline and crucial for dealing with big data sets. Without proper blocking, the next phases are intractable. The blocking algorithms applied should be cheap in computational cost. Blocking aims to construct blocks so that each block only has rows referring to the same real-world entity. In practice, however, this is unrealistic; multiple similar real-

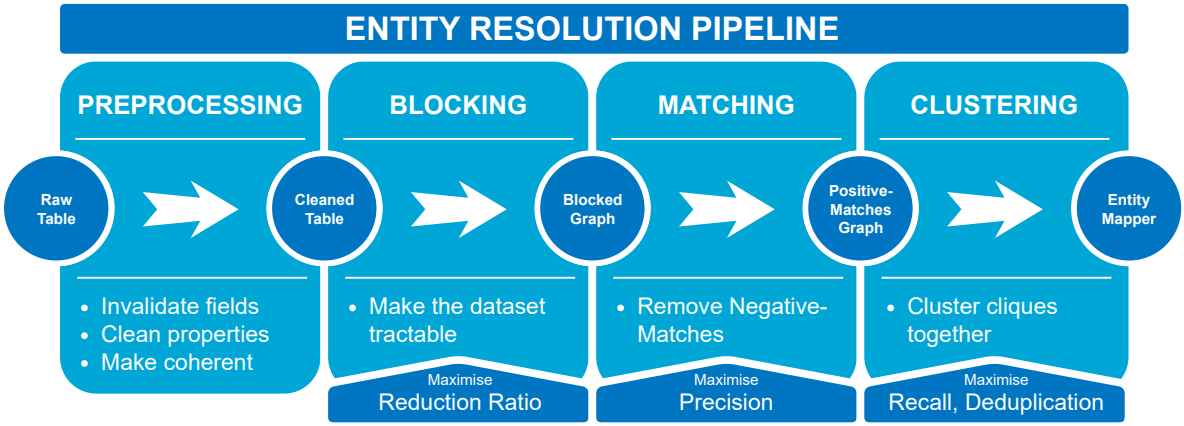

Figure 3.1: Diagram of the ER pipeline and the characteristics per phase

world entities will be mapped to the same blocks, which will be classified in the subsequent phase. Decisions must be made on whether the blocks should favor precision or recall. In the ideal situation, one should construct the BKV from sufficiently available properties with the most negligible error, i.e., the highest quality possible [7]. The aim should be to create many correct buckets with small sizes, as cross-comparisons within buckets are quadratic per bucket size. This should be possible for CC-ER, as real-world entities are unique for both data sets. However, in the case of D-ER, this is impossible with real-world data due to some real-world entities dominating the data set. As a result, some buckets will be huge, while others will be almost empty [7, 18]. The applied blocking phase is detailed in Chapter 5.

3.3. Matching

Each BKV is transformed into a complete undirected graph in the matching phase. Afterward, all the edges are stored in a distinct data set, dropping duplicated block Matches. Since the number of possible Matches is significantly smaller than a total data set comparison, it can be possible to use more expensive algorithms to perform Matching inside the blocks. In the matching phase, the idea is that the same recall remains while increasing the precision. The entire applied matching phase is provided in Chapter 6. Throughout the project, multiple definitions exist regarding 'Match'. Table 3.1 explains all these different definitions and will be used henceforth.

	Meaning	**Created by**
Match	A comparison between two Accounts that have a block, created by the blocking phase, in common. The existence of a comparison creates an edge between the two IDs. However, it has yet to be discovered if this match is correct.	blocking-output
True-Match	A comparison between two Accounts, which both have a valid label provided by the user and these two labels are equal.	user-input
False-Match	Similar to True-Match, however, the two labels are unequal.	user-input
Positive-Match	The classifier predicted that the Match is a True-Match, which is not necessarily correct.	classifier-output
Negative-Match	The classifier predicted that the Match is a False-Match, which is not necessarily correct.	classifier-output

Table 3.1: Match Terminology

3.4. Clustering

The matching phase will create a large matched graph containing Positive-Matches while discarding all the Negative-Matches. These Negative-Matches are only determined by comparing the two Accounts of a Match. In the clustering phase, spatial information of the Positive-Match within the graph can be considered and used to determine the correctness of the Positive-Match. Matches are created by Accounts with joint BKVs. Due to the blocking, not all the rows referring to the same real-world entity can be blocked together. There are most likely still Positive-Matches that were missed. These missing Matches will be searched and added to the total Positive-Match pool. Finally, Entities can be created, which are groups of disjoint Accounts. Each unique Account will be mapped to a single Entity, giving an EntityMapper data set as the final output of the ER pipeline. These EntityMappings are theoretically maximized in terms of recall and precision while having, given the data size, a relatively lightweight algorithm. The clustering of the data set is detailed in Chapter 7.

3.5. GroundTruth

It is important to consider what the resolved data set should represent, i.e., when two different rows are a True-Match or a False-Match, meaning they both refer to the same real-world entity. In order to test the performance of the applied ER pipeline, all the True-Matches should be found. Even for a small subset of 100,000 entities, it is still required to label 5 billion possible matches if they are a True- or False-Match. It is infeasible to label these matches manually.

The use of unique identifiable properties logically retrieves True-Matches. For Exact's data set, multi-

ple properties uniquely describe a real-world entity. The most noticeable are VATNumber and CoC. Both fields are unique to real-world companies. Therefore, these properties can be used to construct a GT data set with high confidence that the True-Matches are valid True-Matches [19, 25, 28]. Theoretically, using a single unique identifiable property is sufficient to create a GT data set, but in the case of Exact, it is possible to use both the VATNumber and CoC property to create the GT data set. The ActiveAccounts subset of the Accounts data set has approximately 85 million unique IDs. If the data set is filtered on entries with both a valid VATNumber and CoC, it results in a data set of roughly 7.6 million unique IDs with both fields provided, as seen in Table 3.2. Such a large data set is sufficient for measuring the performance of applied techniques. The creation of the GT is simply grouping by the VATNumber and CoC; afterward, all the unique IDs are then collected, and the rows grouped together form a clique, which makes up the GT. It can be seen that the GT primarily consists of False-Matches, while there are only some True-Matches. This imbalance of True- and False-Matches throughout the pipeline should be considered per pipeline phase.

Type	**Amount**
Labeled Accounts	7,617,382
Labeled Comparisons	29,012,250,458,271
Labeled True-Matches	7,256,780,795
Labeled False-Matches	29,004,993,677,476

Table 3.2: Label Distributions in GT

4

Preprocessing

In this chapter, we investigate the first phase of the ER pipeline, as detailed in Section 3.1. We start by familiarizing ourselves with the Accounts data set and applying transformations to make the data coherent in Section 4.1. Afterward, we investigate the characteristics per property of the Accounts data set in Section 4.2. Next, in Section 4.3, we investigate the GT. There are also comparisons known as False-Matches without applying ER, discussed in Section 4.4. Finding common words is more complex for D-ER, and the problem is explained in Section 4.5. Lastly, we show that inverting the index gives a significant advantage for limited costs in Section 4.6.

4.1. Exact Account Data

First, it is required to get familiar with the data and how it is structured. The users are only required to provide a Name for creating an Account; all the other fields are optional. For this reason, the data inside the Accounts table is heterogeneous. The data is ambiguous due to the lower quality of incomplete fields. The fields are shown in Table 4.1. It is in Exact's interest to use the active Accounts. Accounts with at least a single transaction assigned to them in the last two years are deemed active. The ActiveAccounts data set has a size of roughly 85 million rows. Each reference to the Accounts data set hereafter refers to the ActiveAccounts data set. To work with the data, the data must be cleaned in the preprocessing step of the ER. For each property, the accents of the accented letters are removed, making the strings more coherent.

Name All letters are changed to lowercase. Afterward, all common words that indicate business structures, such as 'bv' and 'GmbH', are removed [19, 25, 28]. The regex to remove these common words can be found in Appendix A.1. Also, words consisting of a single letter and prepositions are removed.

Postcode All the whitespace is removed, and only the digits and letters are kept. Secondly, the letters are capitalized. Thirdly, there is a match if the value starts with four digits and ends with two capital letters. Postcodes starting with a '0' and those that have prohibited suffixes are removed. Entries not following this match are discarded.

VATNumber First, all the whitespaces are removed. Secondly, we capitalize the letters. Thirdly, we check if the entry follows the format of starting with 'NL' followed by nine digits, followed by a 'B' and two digits. All the entries in which the first nine digits are '0' are removed.

ChamberOfCommerce First, we remove everything that is not a digit. Afterward, we keep all entries with either eight or twelve digits. All the entries in which the first eight digits are '0' are removed since that is not a valid CoC.

AddressLine1 The AddressLine1 (AL1) is transformed to initcap format; in this format, the first letter of every word is capitalized. Words that consist only of '0' are removed. Afterward, we extracted words that contained a number from the AL1 and inserted them into a new column called AL1N. The words that are not

Key	Description	Flag
Name	The name of an Account	**Required**
Postcode	The postal code of the Account	optional
Email	The email address associated with the Account	optional
Phone	The phone number associated with the Account	optional
Fax	The fax number associated with the Account	optional
Website	The website of the Account	optional
State	The state where the Account is located	optional
City	The city where the Account is located	optional
AddressLine1	The main address of the Account	optional
AddressLine2	Additions to the main address	optional
AddressLine3	Additions to the main address	optional
VATNumber	Tax number given by the government to track a company and their tax payments	optional
ChamberOfCommerce	Number that is given by the Dutch Chamber of Commerce which deals company registration within the Netherlands	optional
Type	Either D, A, which refers to a division and an account in a division, respectively	*generated*

Table 4.1: The data structure of Accounts

extracted are part of the street name and are also placed in a new column called AL1S. The same actions are applied to AL2 and AL3.

Email After examining the data, it was shown that some users enter multiple Email addresses per field. The preprocessing should take into consideration that this field can have multiple entries. The applied preprocessing steps were as follows: First, all the accents are removed from the letters. Secondly, when multiple Emails are filled in, different separators are used, so they need to be mapped to a uniform separator. Thirdly, not allowed characters for emails are removed. Fourthly, a simple regex is used to extract the valid Emails, which can be found in Appendix A.3. Finally, Emails can contain a '+' with some additional information, and the additional information is removed.

Website First, change all the letters to lowercase. Secondly, some divisions use a comma where a dot should be, so replace those with a dot. The whitespace around dots is also removed. Thirdly, the Website prefixes are removed with a regex, as seen in Appendix A.2.1. Afterward, the addresses and their domain are extracted from the list, removing paths and queries from the fields. Some users provide multiple Website addresses per field, which should be considered; the regex is in Appendix A.2.2.

Phone, Fax A row could consist of multiple Phone numbers, which should be considered. Users use different separators to fill in multiple phone numbers, and the different separators should be converted to a uniform one. Next, the invalid characters are removed from the field. Afterward, the Phone numbers are extracted with a regex, as seen in Appendix A.4.

City, State, Country Generic preprocessing is applied to these properties. The letters' accents are removed and then transformed to initcap format.

The preprocessing step applied to the Accounts data set is a step that is also performed in the previous theses and their handling of the data [19, 25, 28]. The additional preprocessing contributions that are applied in this implementation are the following:

- Accents from accented letters are removed, increasing text similarity as accents are used sporadically.

- Common words other than business structure, such as prepositions, are removed from the Name property.
- Fields that can hold multiple entries are considered, and their different values are extracted, including but not limited to Website, Email, and Phone.
- A more forgiving Phone validation is used.
- The use of initcap to emphasize the importance of the starting letters for words.
- Preprocessing on more properties of Accounts.
- AL1S and AL1N are automatically extracted from AL1.

4.2. Property Characteristics

Preprocessing is an essential step in order to work with ambiguous big data. More insights into the available raw and preprocessed data can be seen in Fig. 4.1. First, the availability of each property is depicted as 'raw', representing the ratio of how many times that property is filled-in by the user. It can be seen that the availability of the properties differs. The always available property is the Name property. For availability reasons, the Name is the base to apply further methods on. Phone, Email, Postcode, AL1, and City are all reasonably available to be utilized. The column 'Preprocessed' shows the total ratio of valid properties after applying the rules described in Section 4.1, while the 'Validness' column shows the ratio of valid filled-in properties. Some properties do not have any rules that can falsify the filled-in field, for example, AL1. Other properties can be falsified if the filled-in property does not follow the generic pattern; for example, the Postcode has a generic pattern that must be followed. The Accounts is a data set that has multinational data in it. Some multinational data is universal, such as a Name or Email address. However, some data requires a format dependent on the country of origin. It can be seen from Fig. 4.1 that Postcode and VATNumber have a lower validity. This lower validity is because only the Dutch formats for Postcode and VATNumber are allowed in the preprocessing. Thereby, fields that might be valid for the given country are invalidated; however, ER is only applied with Dutch companies in mind for this project's scope. The ratio of adjusted characters is shown as 'Adjustments' to show that preprocessing makes the data uniform. The Levenshtein distance between the raw and preprocessed valid fields is calculated to retrieve this ratio. The Levenshtein distance is essentially an edit distance in which the number of adjustments is counted [26]. These adjustments can either be an insertion, deletion, or substitution [26]. Levenshtein distance is the minimal number of characters that must be changed to transform the raw string into the preprocessed one [26]. To get the ratio per property, the Levenshtein distance is divided by the length of the largest string, and the average is taken per property. Most adjustments done to the strings are in the form of removing invalid letters and changing letters from uppercase to lowercase, so entries mainly consisting of numbers, such as VATNumber and CoC, score a low adjustment ratio. The State property has a high adjustment ratio as the state is provided in upper case and will be transformed to initcap as the preprocessing step.

4.3. GroundTruth availability

Creating the GT, as described in Section 3.5, has the benefit of being cheap, fast, and highly accurate. However, the only notable disadvantage is that the data set is essentially filtered on the rows of the data set that are adequately filled in. If the rows have a VATNumber and CoC filled in, there is a higher chance that the other fields are also filled in, as users who correctly fill in multiple fields tend to make their entries as complete as possible. This results in the GT measuring the performance of the entire ER pipeline on the best rows the data set has to offer, which is a problem that might occur while dealing with incomplete data sets. A robust ER pipeline does not solely rely on these properties; neither will be the case for this implementation. However, it is important to consider the availability of the properties of the GT set, as depicted in Fig. 4.2. Since the GT is created from entries with a valid VATNumber and CoC, they are always available in the GT data set. The distribution of some properties increased while others decreased. The Website, State, Postcode, AL1, and City all have their availability increased in the GT data set, while Phone and Email got their availability decreased. Even though the distribution of available properties is changed when taking the GT, it is still a fair comparison to use the GT to resemble the entirety of the Accounts data set.

Since the AL3, Fax, and AL2 properties are inadequately represented in the preprocessed set and the GT, these properties are discarded for further use within the ER pipeline. It can be noticed that Phone and Email

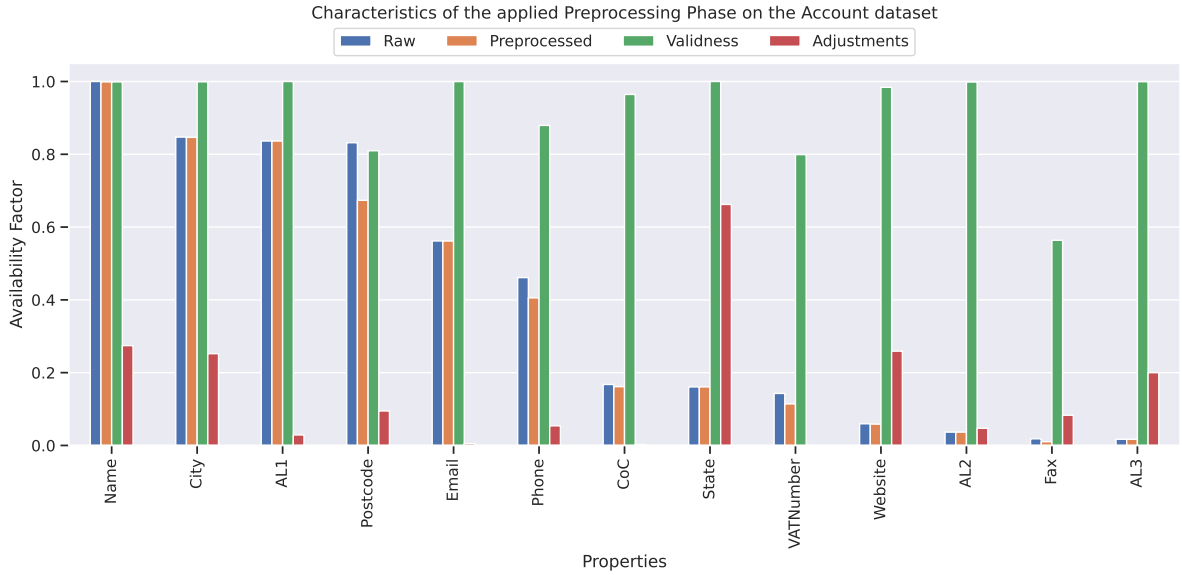

Figure 4.1: Insights of the applied preprocessing to the Accounts data set

are the fields that are less represented in the GT; this is most likely due to the Accounts data set containing end-users of companies. The end-users have their contact information put into the Accounts data set, and these end-users most likely do not have their own company, thus lacking a CoC. Accounts without a CoC are not part of the GT, thereby explaining the lower availability factor for Phone and Email in the GT. The other properties have higher availability factors in the GT than the overall property availability. This higher availability can be explained by the fact that if users are filling in the VATNumber and the CoC, they are more likely to provide the other properties.

4.4. Known False-Matches

It is possible to deduce False-Matches before applying ER. The Accounts data set has a generated property called Type. Type can either be A(ccount) or D(ivision). Type D are users of Exact, and type A are clients of the users. Therefore, Accounts with type D are unique real-world entities, as real-world companies do not sign up multiple times for Exact software. Resulting in the fact that a row of type D can never be a True-Match with another row of type D; they are all False-Matches. There are roughly 700.000 type D Accounts, resulting in 250 billion False-Matches known before the ER pipeline.

A similar scheme can be deduced for the division. Rows with the same division should not be compared, as users do not create multiple Accounts for the same real-world entity. The little or no edge cases that are doing so are not worth the extra calculations. Not comparing rows created by the same user results in roughly 21 trillion known False-Matches to be excluded.

If both of these known False-Matches are considered and removed from the possible matching pool, then roughly 22 trillion required comparisons are already eliminated. Even though that is a lot of eliminated comparisons, the entire matching pool consists of 44,630 trillion possible matches. Lowering the total number of the matching pool by 0.04 percent is insignificant, but these known False-Matches could also be used for heuristics throughout the different stages of the ER pipeline.

4.5. Common words

Numerous common words can be found throughout the filled-in fields of the Accounts data set. These common words can take on many different forms; the Name field consists of different business structures that are common throughout the different unique names that make up the Name column or fields in the Phone column that have, for example, the value '06-12345678'. Common words mostly provide little context about the meaning of a sentence, as vastly different entities can share the same common words [30]. In a CC-ER, finding and removing these common words from the data set is almost trivial. Term Frequency - Inverse Document Frequency (TF-IDF) can be applied to calculate the uniqueness of each word in the document. Each

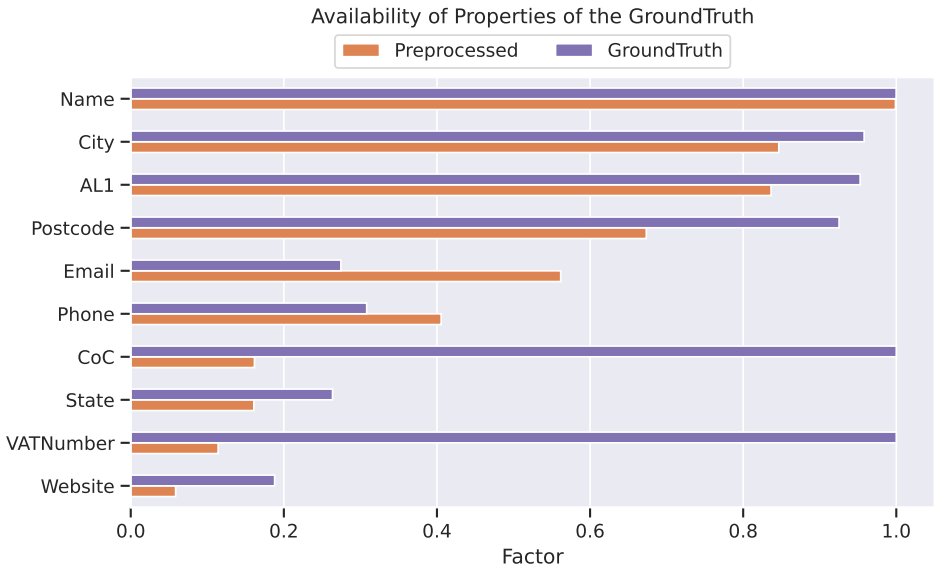

Figure 4.2: The distribution of available properties of the GT compared to Accounts

row in the data set is considered as a document. TF-IDF compares the number of times a word appears in the document to the number of documents in which the word appears. In CC-ER, TF-IDF can be applied to eliminate the common words from the data set, as there exists at most only one duplication per real-world entity. In D-ER, however, it is not advisable to be used. Entities can be duplicated many times in such a data set, and as a result, unique words can even be more common than common words. If, for example, many entities refer to the real-world entity 'Exact', then by using TF-IDF, 'Exact' is considered a common word. Marking unique identifiable words as common words is faulty behavior and should be avoided. One can manually seek and remove the most common words by researching the words and their count. However, this will most likely result in only the most frequently occurring common words being exterminated, leaving the numerous infrequent common words in the data set.

To automatically retrieve the list of common words for D-ER, one can use their GT. A method similar to TF-IDF can be applied to find the common words in D-ER. The TF remains the same; however, the IDF changes to work with the GT instead. The number of unique GT pairs linked to a word are retrieved. In the case of 'Exact', only a couple of GT pairs exist in which that word is found. A common word such as 'BV' can be found in many Names, while there are also many GT pairs in which that word can be found. Thus, by applying some thresholds, it is possible to automatically find and remove the common words from the pool to enrich the context of the words and allow for fewer misclassifications, reducing the overall noise of the ER and positively impacting the recall and precision. Applying this technique to shared properties between different real-world entities will not work. A unique Postcode could be shared between many entities while referring to different real-world entities; as a result, the majority of unique Postcodes can be tagged as 'common words' if this concept is applied. Therefore, one should use this technique on identifiable properties per real-world entities such as Name, Phone, Fax, Email, and Website. It is also possible to apply this technique on a combination of columns, for example, Postcode in combination with AL1S, to remove combinations with little context from the data set.

4.6. Inverting the Index

Redundant comparisons should be limited as much as possible. If there are identical rows, for example, $row_1 = row_2$, then comparing row_3 to row_1 will give the same result as comparing row_3 to row_2. If the rows are truly equal, then by default, they will be referring to the same real-world company, as it is impossible to distinguish different real-world companies from identical rows. It is important to note that the effects of inversing the index are linked to the degree of duplication within the used data set. A high degree of deduplication will result in a high comparison reduction when the indices are inverted. On the other hand, a low deduplication degree results in a limited reduction in comparisons when the indices are inverted. The cost of inverting the index is negligible, while the improvement can be significant.

Type	Amount
ActiveAccounts Count	84,597,242
InverseIDs Count	63,201,285
ActiveAccounts Comparisons	3,578,346,634,704,661
InverseIDs Comparisons	1,997,201,181,224,970

Table 4.2: InverseIDs Comparison Improvement

In the InverseID, the rows are grouped by all their properties except their ID. These properties are shown in Fig. 4.2, which are AL1, City, Email, Name, Phone, Postcode, State, and Website. If the selected fields of the rows are equal, they will receive the same InverseID. Inverting the index can be done with a mapper, in which ID will be mapped to InverseID in a n-to-1 relationship, as in multiple unique IDs can be mapped to a unique InverseID. The significance of using InverseIDs in D-ER can be seen in Table 4.2. Applying InverseIDs decreases the total number of rows by roughly 25 percent. Due to the exponential nature of cross-comparing all the rows, applying InverseIDs decreases the total number of comparisons by roughly 45 percent. With little cost, applying InverseIDs to D-ER has a significant impact.

5
Blocking

In this chapter, we investigate the second phase of the ER pipeline, as detailed in Section 3.2. This phase builds on the preprocessed data as described in Section 2.3. Reducing the number of comparisons is the blocking phase's key characteristic, as discussed in Section 5.1. Next, in Section 5.2, we investigate how to apply blocking on the Accounts data set. Afterward, the different applications of LSH are investigated in Section 5.3. After selecting MinHash as a promising LSH method, we dive deeper into the details of MinHash, what it is, and how to apply it, as discussed in Section 5.4. After explaining MinHash, we discuss which parameters are optimal and which should be fine-tuned in Section 5.5. Afterward, in Section 5.6, we show an improvement to the Sliding Window (Section 2.4). Applying multiple blocking schemes, as discussed in Section 5.7, is possible. The addition of different schemes to the proposed MinHash blocking scheme is investigated in Section 5.8. Experiments are conducted on different augmentation setups with a smaller GT, allowing us to find the most optimal block creation for the data set. Lastly, four different runs are introduced in Section 5.9; these different runs are used throughout the thesis to show the differences in applying LSH to the created deduplicated data set.

5.1. Blocking Measurements Focus

The most important aspect of the blocking phase is not necessarily the PC or the PQ; the key characteristic of applying a blocking scheme is having a high RR, with only a limited impact on the PC and PQ. However, even though RR and the PQ are correlated in terms of the efficiency in the blocking phase, they both address different aspects of efficiency.

It is also crucial to keep the PC as high as possible, as lowering the PC in the blocking phase will have deteriorating effects on the later phases of the ER pipeline. The data set is enormous, and cross-comparing everything will result in many comparisons, as seen in Table 4.2. Therefore, the aim is to perform the comparison with a relatively small number of comparisons. However, this will cause the RR to be a value near 1. In order to gain more valuable insights into the differences between applied schematics, a transformation is applied to the RR, which allows distinguishing differences between RRs more clearly. This transformation is called the Theoretical SpeedUp (TSU), as seen in Eq. (5.1). It represents the speedup that should theoretically be achieved if the time requirement is linear to the number of comparisons. The RR will be asymptotically blown up near the maximum value of 1, equivalent to the theoretical speedup achieved with blocking compared to not applying any blocking.

$$\mathrm{TSU} = 1/(1-\mathrm{RR}) - 1 \tag{5.1}$$

5.2. Blocking Specifics for Accounts

In order to block every row of the Accounts data set, each row must have at least one BKV. If different rows share the same BKV, they will belong to the same block and, therefore, be compared. For this reason, the Name property is used as the basis for creating blocks, as the user is required to fill it in.

The traditional blocking methods are created to merge data sets in a clean-clean manner, i.e., the data sets do not contain any duplicates within them. Such a merge will at most result in a single duplication for each real-world entity. The case for D-ER is to find duplicated real-world entities in a given set, normally this

will result in some duplications to be found, e.g., 1 to 10 duplicated rows for a single unique real-world entity. However, in the case of Exact, some rows are highly duplicated. Since all the users use the Exact software, the majority of them created an Account for Exact. This results in a highly skewed distribution of duplicated rows. If the entities were perfectly blocked, a block with the entities referring to Exact will still be populated with approximately 300.000 rows, resulting in $C_D \approx 4.5 \cdot 10^{10}$ comparisons.

Most blocking techniques create tokens from a BKV to address the noise in the data, resulting in the row being placed in many disjoint blocks. If there is a high degree of exploratory for finding similar entities, then the blocking algorithm could create many erroneous blocks with only a few correct blocks. Such an output will negatively impact the PQ while only marginally improving the PC. Blocking techniques that create many disjoint blocks reason that one can drop the most populated blocks to increase the PQ and only minimally impact the PC. In the case of Exact, however, dropping these blocks will significantly lower the PC, as the most populated blocks are all blocks that could refer to highly deduplicated entities within the Accounts data set. The number of comparisons quadratically grows per the size of the block. If blocks containing the highly duplicated rows were dropped, then the many connections between rows of a correct block would not be found anymore and significantly affect the PC. For this reason, blocking techniques that create many disjoint blocks per row are not feasible for the Accounts data set. These include, as discussed in Section 2.4, N-gram Blocking, Canopy Clustering, and Suffix Array. The created BKVs are inverted on their index. Inverting is done by partitioning per BKVs and discarding the BKVs with only a single InverseID in them. These partitioned BKVs are the created blocks of the blocking pipeline.

5.3. Locality-Sensitive Hashing

Locality-Sensitive Hashing (LSH) is an ANN method; hashes are created from each row, and these hashes approximate the similarities between different rows. There are two main implementations of LSH, which are BRP and MinHash. To apply LSH, the rows must be represented in a machine-readable vector format. The former relies on dense vector representations, and the latter relies on sparse vector representations. Both implementations are suitable for different applications.

In BRP, the Euclidean distance between different rows of $D(A)$ is approximated. A random vector representing a hyperplane is formed to create a hash in BRP, and the dot product between said vector and every row is calculated. Sparse vector representation will not work with BRP, as the idea behind a sparse vector is that the vector consists mostly of zeros. If both vectors consist mostly of zeros, then most of the vector is the same since all these zeros match. This results in the vectors almost being an exact match, even though the two vectors do not share a single item. This incorrect match is, of course, not the required behavior. Therefore, a smaller dense vector should be constructed that represents each item in a smaller dimension.

MinHash is an algorithm for approximating the Jaccard similarity between two sets of objects. Jaccard similarity is calculated by dividing the number of elements in common with the number of unique elements in the combined set, as seen in Eq. (2.7). In the Jaccard similarity, the dense vector representation is not applicable, as these vectors are learned representations of words in the rational space. It is improbable that close words share the same rational number, causing almost no buckets to be created from a dense vector. Instead, a sparse vector should be used. In Jaccard similarity, only the elements of the rows are compared. None of the zeros of the sparse vectors will influence the outcome, as they do not represent any token.

5.3.1. Name Context Investigation

In order to decide which variation of LSH to apply, the context of the words must be investigated. BRP uses word embeddings to create vectors per row. An example of word embeddings is Word2Vec, which creates dense vector representations of words that make up an Account Name. Each word gets a random dense vector within such a Natural Language Processing (NLP) algorithm. Then, feeding the algorithm context around the words will transform the random dense vector into a representation that closely resembles the word based on the context. Pre-trained Word2Vec embeddings exist on, for example, the English language. However, the use case within this application is specifically to embed companies in the space. These pre-trained models are trained for generic linguistical use cases, and using them for specific environments will not be effective. No pre-trained Word2Vec embeddings exist specifically for the Dutch company space, so using a pre-existing model is out of the question; a custom embedding must be learned. For BRP, context is required to learn a vector representation per word. The context of a word is based on the surrounding words. Typically, the context is learned from descriptions with multiple sentences of many words. However, no such description exists for the rows in Account. Only the Name column can be used to learn Word2Vec embeddings. However,

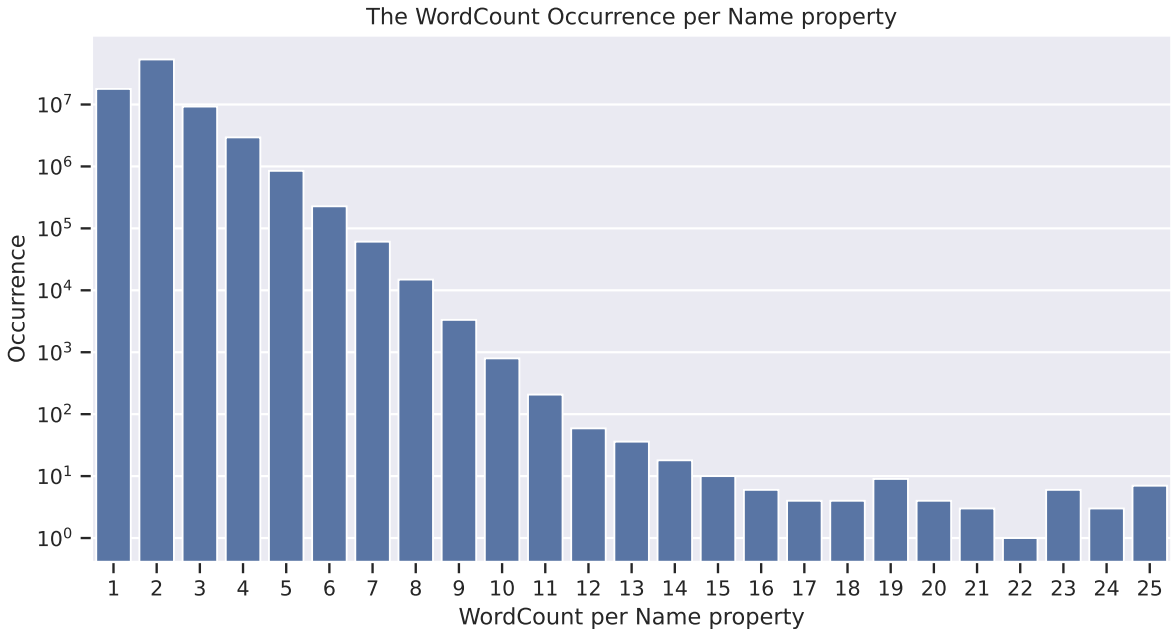

Figure 5.1: The number of words per Name of Accounts

with limited context, this might be an issue.

To investigate the count of words per Name row is investigated with their distribution in Fig. 5.1. It can be noticed that the majority of the Names only consist of one up to three words. If the context only consists of a single word, learning a Word2Vec embedding is impossible, as it has no context. The context might be limited within Name. If there is just a single word, then it is likely that the word will uniquely identify the company. However, if an additional word is added, the additional word might be shared among other distinctive companies. These additional words could express city names or professions, for example. Thus, with more words added, it is more likely that the second word will be a common word that does not necessarily distinguish the company, as explained in Section 4.5. Labels attached to words can be used to compare the strength of words.

Each row in Accounts with a label attached is split into their word representations. Then, it is possible to see the number of different labels attached to each word. Some words are shared between distinct companies; these should have a lower context strength, while other words will uniquely describe a company and, therefore, have a high context strength. To express the context strength of a word, the Mean Squared Error (MSE), as seen in Eq. (5.2), is used.

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_2)^2 \tag{5.2}$$

The MSE has a few variables. n is the number of samples, Y_i is the observed value, and $\hat{Y}_2$ is the predicted value. Some changes must be made to use MSE to express WordRichness. The WordRichness is expressed using the formula shown in Eq. (5.3). First, the MSE is calculated per word. Instead of n being the size of all samples, they are now $n(w)$ the size of labeled samples for a given word. Y_i represents the error-less case, meaning this should be the highest achievable value for the WordRichness, which is 1. $\hat{Y}_2$ will now represent the distribution of labels over w. The words are grouped by the label; the fraction of the current label over the entire set of labels attached to the word will represent $\hat{Y}_2$. Every word has a label attached; first, the number of repetitions of the current label within w will be retrieved as denoted as $w(l_i)$ and divided by the total number of labeled words, giving a fraction of how many labels make up the entire label knowledge base of w. In this case, the MSE ranges from 0 to 1, from no errors to only errors. In the context richness, 1 represents a word that uniquely describes a real-world company, while values near 0 represent a common word shared between many real-world companies.

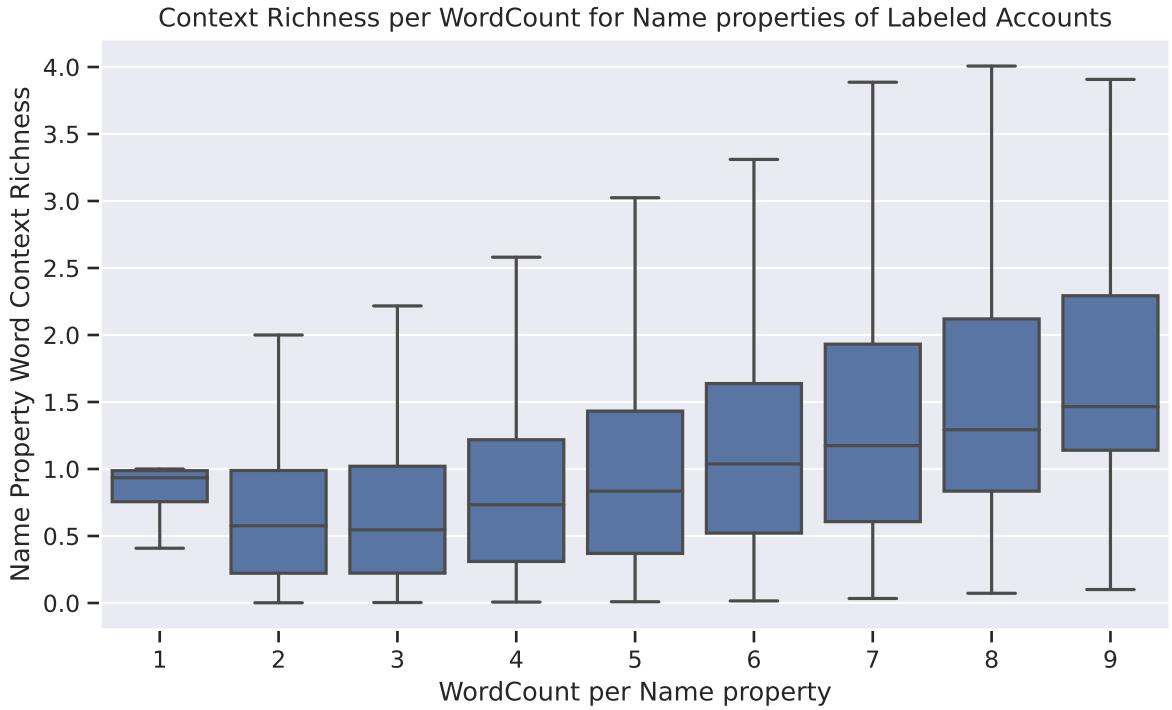

Figure 5.2: The Word Context Richness on the Name property per WordCount of GT

$$\text{WordRichness(w)} = 1 - \frac{1}{|w|}\sum_{i=1}^{|w|}(1 - \frac{|w(l_i)|}{|w|})^2 \tag{5.3}$$

The WordRichness of every word that makes up the Name property of GT can be calculated. Every Name property consists of a variable number of words; the ContextRichness of a row is the summation of all the WordRichnesses per row. This means the maximum ContextRichness per Name row is the number of words the row consists of. Therefore, it is possible to investigate if there is more ContextRichness when a Name entry consists of more words. The results can be seen in Fig. 5.2. It shows that if the Name consists of a single word, it mostly has a high ContextRichness. If the row consists of multiple words, then the ContextRichness increases per number of words; however, the median ContextRichness increments significantly slower than the maximum ContextRichness when the number of words per row increases. This slow increment means that when more words are added, they are mostly common words, which will not provide context for learning a representation. Because of the limitations with the context of the Name property, learning a Word2Vec embedding will not be effective. Instead, a MinHash implementation should be investigated and implemented.

5.4. MinHash Implementation

For MinHash, a sparse vector representation is needed for the rows of the data set. To create this sparse vector representation, a vocabulary is required. For data set $D(A)$, the vocabulary $v(A)$ is created, which consists of all the possible tokens that make up $D(A)$. For the creation of a vocabulary, different parameters can be tweaked. A minimum TF can be applied, meaning that the token must be duplicated several times within a document itself. A minimum and maximum Document Frequency (DF) can be set. A unique token must be duplicated more than the minimum and less than the maximum DF in the corpus. A maximum size can be assigned to the vocabulary; a larger size contains more knowledge about the documents but increases the time and space requirements.

The created vocabulary model assigns a unique index to each token that follows the specified parameters. This model allows the transformation of the documents into a vector representation. The TF per token of the vocabulary is stored in a vector per document, which allows each row of $D(A)$ to be transformed into an array of indices of size $|v(A)|$ representing the $v(A)$. Each row of $D(A)$ only consists of a few tokens with a TF above 0 from $v(A)$. Creating a vector for each row will consist primarily of zeros, as it only contains a few

tokens of $v(A)$. Therefore, sparse vectors should be used for the MinHash method. Only keep track of all the tokens that the document contains. Using sparse vectors results in enhanced space efficiency. Instead of representing the TF, opting for a one-hot encoder is also possible, which will set all the TF's above 0 to 1. In the thesis, one-hot encoding is not applied, as this will lose the TF information and could cause the creation of more erroneous blocks. Generating a hash is based on the token it contains and the TF of said token in the document.

5.4.1. Vocabulary Token Creation

To create the tokens of the vocabulary, one can split the text by whitespace, creating lists of words. Each unique word will be an entry in the vocabulary. Doing so has the problem that spelling mistakes or near-identical words will be given a different index in the vocabulary, making it impossible for them to become a match. Tokens unknown in the vocabulary cannot be processed and are therefore lost. The text can be split into N-grams, as discussed in Section 2.4, representation to overcome these issues. The splitting of the text into this representation has the benefit of overcoming spelling mistakes by using the Jaccard similarity, as only the tokens containing the spelling mistake will be misaligned, while the correctly spelled tokens are aligned. There is only one side-effect: the longer words will be transformed into more tokens, giving the longer words more overall weight within the Jaccard similarity. Instead of using a single N to construct the N-grams, it is possible to use a range of N. This addition to N-gram is already implemented in FastText[1] [4, 22]. It transforms the text into N-grams from Min$N$ up to Max$N$. With a broader range of N-grams, more structure is preserved in the token representation. If the alphabet consists of a characters, then the number of tokens that could be created per N are a^N. This means that a large N will create many unique tokens, dominating the fewer tokens created by lower values of N. Larger N are also more impacted by spelling errors. Using a larger N creates fewer tokens per word as the size of the token increases. A larger N will create more misaligned tokens between the correct word and the word with a single spelling mistake, up to a maximum of N different tokens for a single spelling mistake. A selected N that is too large will make it more challenging to create buckets of words containing spelling errors.

In addition to the range of N, extra weight is also given to the start and end of a word. FastText does this by adding angular brackets around the words. In the tried implementation, the brackets are not added around each word but around each document. FastText works per word, but with the MinHash implementation, it is also possible to use the whitespace segments of the documents; two adjacent words are connected by whitespace, transforming the document into N-grams will also create tokens containing said whitespace. Therefore, extra tokens are already created for words ending and starting in the middle of the document. So, only adding brackets around the start and end of the document will create extra tokens for equal starting and ending combinations.

In the first pass, all the unique tokens are given an ID in the vocabulary. With the vocabulary, a document can be transformed into a sparse vector; all the IDs making up the N-gram will be stored in this sparse vector. New unforeseen words that were not part of the vocabulary creation still have a high chance that tokens of the vocabulary will represent them, as the unforeseen word will consist of different N-grams already contained in the vocabulary. With the use of FastText, it is possible to overcome the issue of spelling mistakes and the problem of out-of-vocabulary words.

5.4.2. Hash Generation

MinHash algorithm uses the created vocabulary to generate its hashes. Multiple hashes per row can be generated. To generate hash_i for a given row, an array from values 0 up to $|v(A)|$ representing all the tokens of $v(A)$ is required. A random permutation p of the index array approximates the Jaccard similarity of multiple rows. Now, p_i is the permutated index array used to generate hash_i. Each row gets the hash equal to the id of the first token of p_i contained within the row's tokens. If two different rows have the same hash, they have a token in common and do not contain all the tokens earlier in the p_i; thus, the generated hash contains more information than just an index within the vocabulary alone. So, each row gets a hash value directly related to its list of tokens. Therefore, rows not sharing a common token could never receive the same hash value. With more hashes, comparing two different lists of hashes, the closer it resembles the Jaccard similarity between the two documents. In the case of removing spelling mistakes, this approximation is beneficial, as this allows for spelling correctness. Using LSH in the blocking phase should create almost identical but not truly identical blocks.

[1] `https://github.com/facebookresearch/fastText`

Multiple permutations generate different hashes of the rows in $D(A)$. So far, a list of hashes can be generated that resemble the Jaccard similarity for $D(A)$ without comparing each row with another row of $D(A)$ as it is only required to compare rows that have at least a single hash in common for the Jaccard similarity. Retrieving the matches is just a matter of grouping the rows with equal hashes, ensuring they have tokens in common. A group of n hashes is called a hash table. A hash table is an AND-clause of n hashes. For two rows to be grouped, they must share n identical hashes. However, it is certainly possible that creating a single hash table will not give the desired result, as if one of the n hashes is misaligned, then both rows will be placed in disjoint blocks. Therefore, m different hash tables are generated per row, giving more chance for near identical rows to be placed in at least a single joint block. These additional hash tables resemble an OR-clause. The total number of required hashes is nm. The combination of these AND- and OR-clauses is called the AND-OR amplification; an example of the amplification for n = 3 and m = 2 is given in Eq. (5.4)

$$(\text{hash}_1 \wedge \text{hash}_2 \wedge \text{hash}_3) \vee (\text{hash}_4 \wedge \text{hash}_5 \wedge \text{hash}_6) \tag{5.4}$$

5.4.3. Hash Manipulations

The result of the MinHash algorithm is an array of hashes for each row in the data set. These hashes could be manipulated to gather the correct neighborhood of similar rows as efficiently as possible. The computational cost of the MinHash significantly relies on the number of generated hashes. In the applied method, the only manipulation performed was the AND-OR amplification, in which the array of hashes is split into multiple disjoint subarrays of hashes. However, three other techniques were tried to create more combinations of subarrays based on the initial array.

Hash Expansion On top of the AND-OR amplification of the generated hash array, it is also possible to expand the hash tables. Expanding a hash table allows for error within the hash table itself. Expanding a hash table transforms the table of size n, $a(n)$ to $n \cdot a(n-1)$. Creating n new sub-tables from the table in which each sub-table has a single element removed from the initial table, creating n unique sub-tables based on the initial array. A single hash table $(1,2,3)$ will be expanded into the sub-tables of $((1,2),(1,3),(2,3))$. Grouping rows by their sub-tables will, in this case, allow for a $\frac{1}{3}$ fault tolerance between the different hash arrays, as each sub-array misses a single element from the initial hash array of size 3.

Generally, one would alter the numHashTables to create more hash tables, but expanding the hash table into sub-tables is time-efficient compared to generating more hashes. Therefore, expanding should be considered for manipulating hash arrays, or arrays in general, to reduce the possible time requirements of the blocking phase. Hash Expansion showed that it worked on creating more blocks without too much computational cost for $n = 1$. However, if the hash expansion were to be applied recursively, it would create too many blocks depending on the numHashesPerTable, consuming too much space and causing the pipeline to fail.

Hash Permutation Sampling In hash sampling, it is possible to create overlapping subsets from the generated hashes, unlike the AND-OR amplification. In essence, random samples of a fixed size are taken from the main hash array; the random sampled array is smaller than the main hash array. The same random sample is retrieved from each row. Theoretically, this is an excellent strategy to handle the generated hash arrays efficiently, cheaply creating blocks from just a few hashes. One way to create hash sampling is to permutate the hash arrays of each row in the same pattern and take the first n hashes as the block, resulting in a single smaller permutated hash array created out of the larger array. One can repeat the same process to create many smaller hash arrays from a single larger hash array. However, it was impossible to effectively implement such a concept with the provided tools, as the idea behind other strategies is to reduce the total number of resources required to approximate the alignment of different hashes. With a built-in function to permutate the hashes equally for each row, it should be possible to have an as accurate but faster method than the generic AND-OR amplification.

Hash N-gram Sampling Implementing hash sampling efficiently in the PySpark environment was impossible. However, a concept already used is N-grams. N-grams can also be created for the hash arrays on the hash level. A window is slid over the hashes to create highly overlapping sub-tables of the primary hash array. This method was not computationally intensive and resulted in many blocks compared to the AND-OR amplification. The generated hash tables via the N-gram method are a superset of the ones generated with AND-OR amplification. The problem was that the generated hash tables were highly overlapping, as each sub-table

only differed in a single hash with the next sub-table in the window. Without much difference between the generated hash tables, only a few extra rows are compared, even though many new blocks are generated. Therefore, implementing N-gram for hash sampling is not efficient or ineffective.

5.5. MinHash Parameter Selection

The optimal parameters are discussed for the MultiGram MinHash implementation as described in The parameters discussed in Sections 5.4.1 and 5.4.2. Each ER implementation will require slightly different parameters. One can approximate the best parameters using the GT, but the GT only considers labeled rows; the entire corpus might behave differently regarding PC and PQ. All of the possible parameters will now be discussed below per parameter.

MinN The MinN determines the smallest N of the N-gram for the FastText representation. It is an important aspect of what the smallest vocabulary tokens should represent. With a low N, less subsequent structure is required for rows to become a match. The MinN should ideally always be set to 2, as tokens consisting of just two characters will serve as basic tokens in the vocabulary.

MaxN The MaxN determines the largest N of the N-gram for the FastText representation. MaxN is an important parameter that enforces the quantity of structure within different rows that needs to be aligned for them to become a match. If set too large, too much equal structure is required between two different rows to become a match. Therefore, the MaxN should not be set to a value that is too high, as that would create too many unique tokens and negatively impact spelling correctness. With this this information, a decision is made to select 3 as MaxN, as it will create more tokens with additional structure on the document while not creating too many unique tokens. If one wants to enforce having more shared structure between the rows in a block, then the MaxN can be increased.

MinTF The MinTF represents the minimum number of token occurrences within the document itself. This means that if set to anything other than 1, there should be duplicated N-grams in each document for all the rows of the Account data set. The majority of the documents will not contain duplicated tokens. Creating the vocabulary on these duplicated tokens will cause the creation of very strict and poor vectors. Therefore, the TF will always be set to 1.

MinDF The MinDF is the minimum number of occurrences per token across the rows to include the token in the vocabulary. MinDF enforces that there is, for each vocabulary token, a minimum number of occurrences in the Account data set. If the minimum is set to 1, it will increase the likelihood of creating blocks with only a single row inside, as only a single row has that unique token. For a higher minimum, fewer unique tokens will make up the vocabulary, meaning that the tokens created by misspellings seen infrequently in the corpus will not be valid tokens of the vocabulary. One could argue that there should be at least two occurrences of a given element in the vocabulary because the elements with only a single occurrence could never create a match between two rows. However, unique tokens that do not match also provide valuable information to block rows on; therefore, MinDF is set to 1, the strictest form for creating the vocabulary. It does create a large vocabulary, but eventually, it will result in the least number of comparisons between the rows possible. This is because rows of these one-of-a-kind tokens will likely end up alone in a block. A low MinDF causes less collision within the generated hashes, increasing the RR of the MinHash blocks. If more collisions are required for rows containing these low DF tokens, then one can increase the number of numHashTables to create more blocks per row instead.

MaxDF The MaxDF is the maximum number of occurrences per token to include the token in the vocabulary. A few highly frequent tokens can dominate the vocabulary; the MaxDF can be tweaked to address that. Since MinHash utilizes a random permutation of the list of indices from the vocabulary, the sporadic and the frequent tokens are equal points of interest for MinHash. This means the most occurring N-grams do not dominate the generated hashes. The MaxDF is maximized in the runs, ensuring the vocabulary consists of all the possible tokens. The reasoning behind this is that if set to a number, it might cause problems because no tokens could be created for the heavily duplicated rows, as expressed in Section 5.2, causing these heavily

duplicated rows not to be included in the MinHash. Including terms with a high DF will increase the possibility of creating large blocks. However, if this is the case, then the numHashesPerTable could be increased to make the blocks rely on more hashes, creating blocks of a smaller size.

VocabSize The VocabSize is the maximum allowed size of the vocabulary, allowing it to reduce a vocabulary that is too large to a smaller format by dropping elements of the vocabulary. Setting it to a specific smaller size will decrease the time complexity but at the cost of the created block quality. No maximum size was specified for the implementation, meaning that the vocabulary would contain all the tokens that follow the set parameters.

numHashesPerTable The number of hashes per table created by the MinHash. If set to a small value, only a few hashes must be aligned for two rows to become a match. If set to a large value, more tokens must be equal. This parameter directly impacts the PQ and the RR of the MinHash. Setting numHashesPerTable too low for the given corpus will create more noisy blocks, negatively impacting the PQ and the RR. If set too high, it will negatively impact the PC, as it will be too difficult for two rows to have many shared hashes and require a larger numHashTables. First, the numHashesPerTable should be selected, and the created blocks should be investigated on the quality. When the quality is on par, the numHashTables can be selected.

numHashTables The number of hash tables created by the MinHash. This parameter directly relates to PC, as more hash tables are generated per token, which means more blocks are created per row, and these extra blocks could provide new comparisons. If the numHashesPerTable is too low, it could create more noisy blocks and lower the PQ. If numHashesPerTable is increased, the blocks become stricter, and therefore, numHashTables should also be increased to create more of these stricter blocks. MinHash requires numHashesPerTable · numHashTables hashes to be generated. Selecting the parameters for the hash generation is a trade-off between effectiveness and efficiency.

5.5.1. Parameter Tuning

The theoretical part of Section 5.4 is implemented, and the tunable parameters will be investigated to allow for optimal block creation of rows on their Name property. Creating experiments on the entire $D(A)$ corpus is not feasible; there are too many required comparisons to create the runs. Therefore, a smaller subset of the GT is used, denoted as Ground sub-Truth (GT_1). Each row in GT_1 is in at least one True-Match with another row of GT_1. The composition of the GT_1 compared to GT is shown in Table 5.1. The Ratio of the False-Matches compared to True-Matches is almost identical between the different GT versions. Since the GT_1 is significantly smaller than the GT, they cannot be compared directly. The results of GT_1 should only be compared with other results relying on GT_1.

Type	GT	GT_1
Labeled Accounts	7,617,382	100.000
Labeled Comparisons	29,012,250,458,271	4,999,950,000
Labeled True-Matches	7,256,780,795	1,255,398
Labeled False-Matches	29,004,993,677,476	4,998,694,602
False-Match Ratio	3996.95	3981.76

Table 5.1: Composition of GT_1 compared to GT

The main focus of interest is the MinN and MaxN of the vocabulary, which are essential to shape the created vocabulary. An exhaustive parameter tuning is performed for MinN=2 and MinN=6, with the MinHash parameters set to numHashTables=2 and numHashesPerTable=6 on the Name properties of rows in GT_1. These runs are compared to a baseline of Standard Blocking on the Name property. MinHash is the technique in question for these runs. Applying more advanced techniques will always reduce the RR and should increase the PC; a trade-off exists between these characteristics. F-scores are applied to gain insights into optimal solutions by balancing this trade-off. The main characteristic of the blocking phase is a high RR, and since the differences in RR are minimal between different runs, it is good to factor in the more significant importance of RR as a large β in the F_β-score, making RR β times more important than PC. If a small β is used, it favors

creating more edges between the rows. Otherwise, if a large β is used, fewer edges between rows are favorable, as can be seen in Figs. 5.3 and 5.4 respectively.

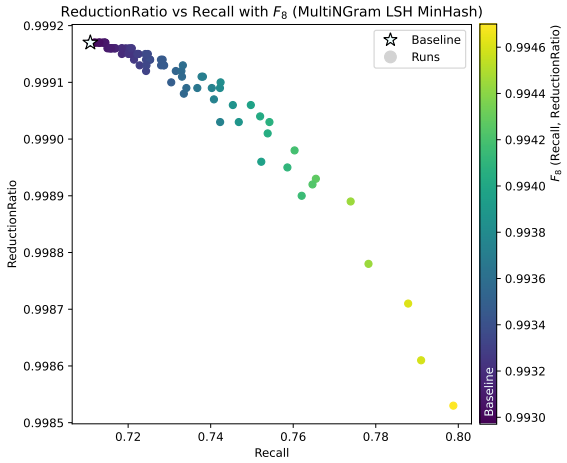

Figure 5.3: MinHash runs with a low $\beta = 8$, making RR 8 times more important than PC

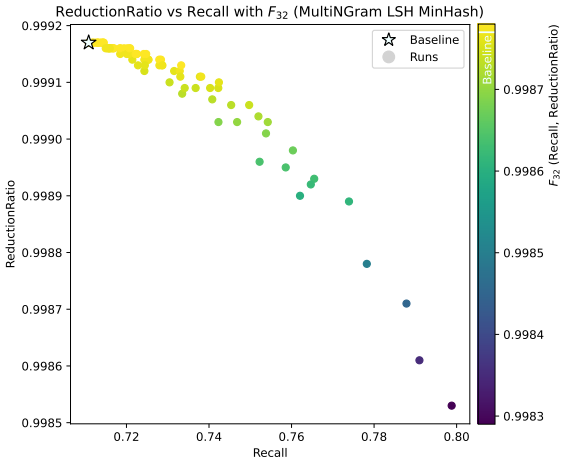

Figure 5.4: MinHash runs with a high $\beta = 32$, making RR 32 times more important than PC

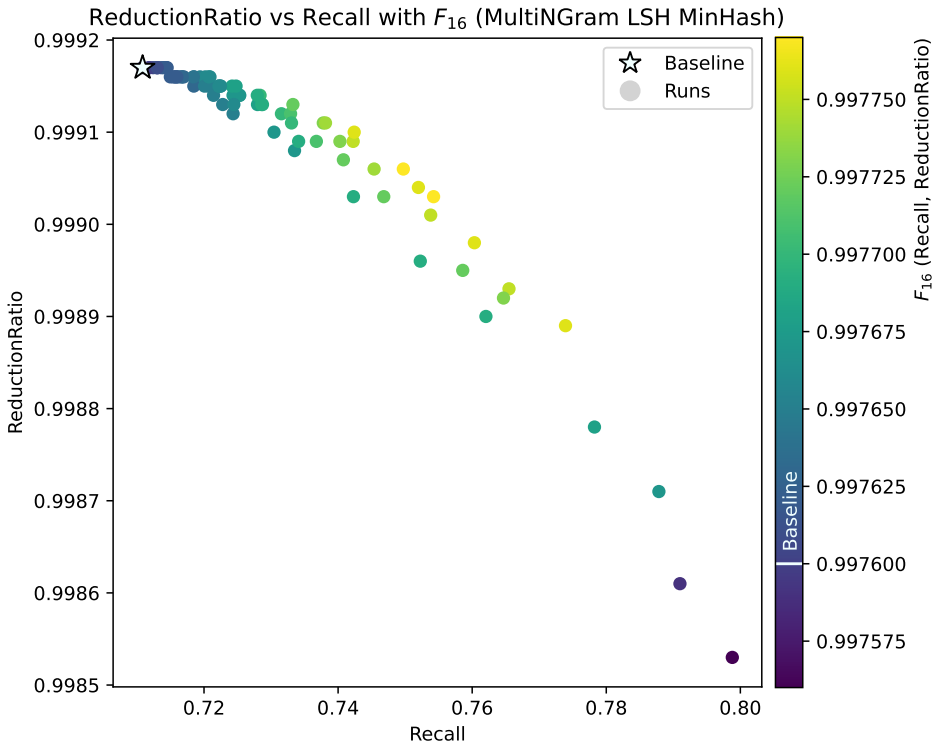

Figure 5.5: MinHash runs with a fair $\beta = 16$, making RR 16 times more important than PC

With a β of 8 and 32, the outliers will be selected as the best options. A fairer trade-off factor is required to ensure high RR while also improving the PC. If the PC is high in the blocking phase, the recall will also be high by the end of the ER pipeline. A fair F-score of these factors would be F_{16}. Since the differences in RRs between the runs are small, and the difference between PCs is higher. The most important aspect of the blocking phase is the RR; thus, it should be factored in to have more weight bound to it. The results of this setting can be seen in Fig. 5.5. It is visible that some runs in terms of F_{16} perform worse than the baseline, while some perform better than the baseline. It seems like the differences in F_{16} score are minimal, but one needs to consider that the RRs are close together; a small change in the F-score can have a big impact on the

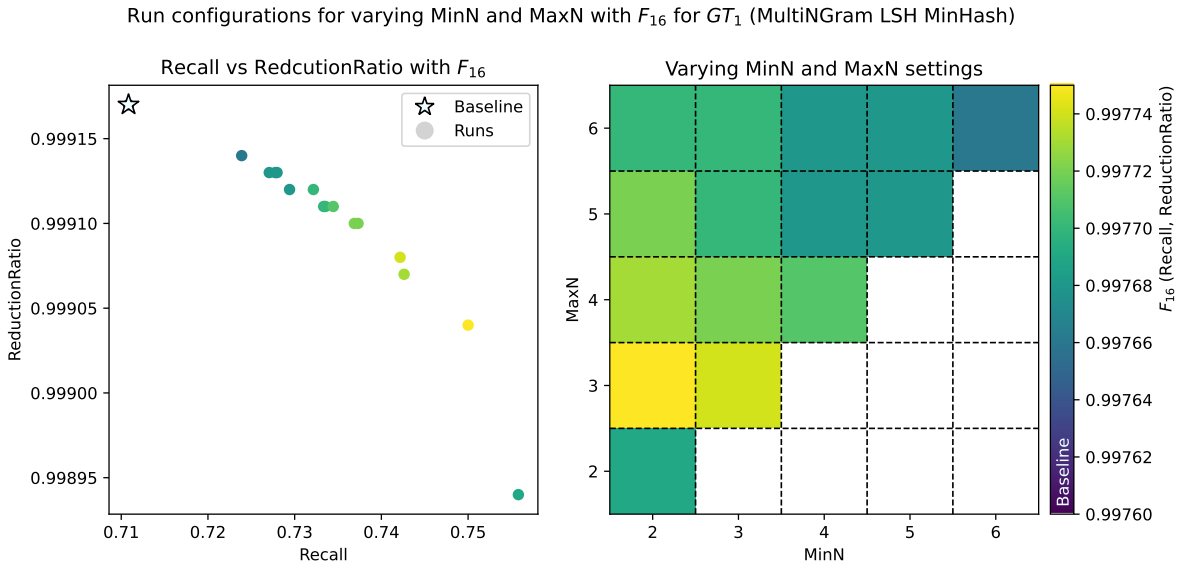

Figure 5.6: PC vs RR for the varying MinN and MaxN for GT_1 and their accompanied F_{16}-Score

final result, especially if the entire corpus is considered instead of the scaled-down GT_1.

The exhaustive search results can now be compared with the balanced F_{16}-score of the trade-off between PQ and RR. MinHash is an ANN; therefore, randomness is involved in calculating the results. To reduce randomness throughout the runs, a repetition factor of 5 is applied, and the average PC and RR are used as the run metrics for the specific configurations. These exhaustive parameter tuning results are depicted in Fig. 5.6. It is shown that the discussed MultiGram parameters are the optimal configuration for GT_1 with a given F_{16}-score between PQ and RR. Namely, the MinN of 2 and the MaxN of 3 have the highest F_{16}-score.

After investigating the created blocks on the entire corpus, including the unlabeled rows, it showed with a manual inspection that numHashesPerTable of 6 is too strict for the selected MultiGram settings; thus, it was lowered to 5 to make the blocks more lenient on different spelled fields. Afterward, the numHashTables is increased to 5 to create more blocks, allowing for more matches. If the data set is fully labeled, then the PC and PQ address the entire corpus, which can be used to determine the perfect numHashesPerTable and numHashTables. For this implementation, a value of numHashesPerTable is set first with domain knowledge in mind, and afterward, one can increase the numHashTables to increase the number of generated matches at the cost of time and space requirements.

5.6. Sliding Window Optimizations

As described in Section 2.4, the Sliding Window has the drawback of making large blocks for the BKVs depending on the N. If the entire neighborhood is inserted into the list of IDs in a BKV, then the blocks would heavily overlap between BKVs in a similar neighborhood. If the sorted neighborhood exists out of three elements (A, B, C), then in the neighborhood of A, the elements B and C should not be compared. Applying this addition makes the blocks smaller and more resource-friendly due to the exponential number of comparisons a larger block requires. In addition, the Sliding Window can also be transformed into a Sliding Comparison Window, in which instead of only creating a match on the neighborhood, the BKVs themselves should be compared if they are close enough to match. In the implementation, the comparison is based on the Levenshtein distance between the two BKVs. A threshold will help increase the PQ and RR while allowing for a larger N, increasing the PC. Applying these optimizations makes the Sliding Window an effective and fast method that is an improvement to Standard Blocking and Sliding Window with the cost of extra computations. It should also be noted that only properties that can use a sorting method should be used in a Sliding Window approach. Using the Sliding Comparison Window on meaningless BKVs, such as a generated hash BKVs, will create contextless blocks full of noisy comparisons.

5.7. Multi-Disciplinary Blocking

There is not a single 'best' blocking scheme. It is always a trade-off between PC, PQ, and RR. However, it is possible to augment a single blocking scheme with another blocking scheme. As pointed out by Papadakis et al., it is possible to transform each property field of a row into a BKV [35]. Multiple BKVs based on different methodologies can be created, harnessing the characteristics of a given blocking scheme with the data. For example, the Sliding Window relies on how strings are sorted into neighborhoods. If two entities refer to the same real-world object, these entries will be put in the same neighborhood if noise exists at the end of the string. If the noise is at the start of the string, then it is likely that these rows are not put in a joint block. Allowing to adjust for noise at the end of a string is a strong characteristic of merging data types such as websites or email addresses. A Sliding Window approach will create blocks of rows that only differ in the top-level domain. For example, rows that end in 'exact.com' and 'exact.nl' will be put in a joint block.

Only some of the rows of Accounts have a filled-in Email. So, it is not possible to rely only on a scheme that blocks purely based on Email, but it can be used as an additional blocking methodology to improve the blocking pipeline's results for a relatively low cost. In the matching step, the algorithm will only calculate if there is a match between two unique IDs; thus, if there is a link between ID_1 and ID_2 in multiple blocks, then the comparison between ID_1 and ID_2 is only done once. If a set of matches created by a blocking scheme is a subset of another collection and these collections are joint, then the number of required comparisons will remain the same. For this reason, it should be beneficial to create a few sparse but dedicated blocks and join them in the collection of matches.

5.8. Account Blocking Augmentation

Augmenting the created blocking data set with multiple properties other than the Name property is possible. Other blocking pipelines can be appended to the created blocking data set. Duplicated comparisons in the blocked data set do not significantly impact the computation time in the further phases of the ER pipeline, as they are filtered out of the set. Multiple properties could augment the created data set to improve PC, as more possible matches can be found. The RR will be lower as more comparisons are added, thus lowering the ratio described in Eq. (2.3). It is impossible to claim anything about the impact on precision by applying augmentations, as it depends on how well the augmentations are created; augmenting with blocks of higher PQ than the baseline will increase the PQ, while a lower PQ augmentation might create more matches, but the blocks will be of lower quality. Therefore, the augmentations are a trade-off between PC, PQ, and RR.

The properties considered to be added as an augmentation to the blocked data set are high in quality but only sometimes available. If a row does not have a filled-in property used in the augmentation, then that row is not considered for said augmentation. Only rows with all the required properties can augment the blocked data set of a given augmentation scheme. The following properties and their applied augmentation scheme are investigated:

1. A Sliding Comparison Window approach on the Website property, as some domains, are exactly the same, but they only differ in their extension, i.e., 'exact.nl' and 'exact.com'. A Sliding Window approach is perfect for using sorting to its advantage. Standard Blocking of websites by just removing the top-level domains is also possible, so removing '.nl' or '.com' would create similar blocks. It is opted to use the Sliding Comparison Window instead.

2. A Sliding Comparison Window approach on the Email property, such an approach has potential for this property similar to the Website property. Different domain extensions could be used in an email address.

3. A Sliding Comparison Window approach on the Phone property. The Phone fields differ in their notation at the start of the entry, i.e., '+316-12345678' and '06 12345678'; therefore, the sorting should be based on the end of the field. This can be achieved by reversing each Phone field before applying the Sliding Comparison Window, allowing the properties to be alphabetically sorted at the end instead.

4. As Kostense mentioned, the Postcode can be enhanced by the AL1N [25]. A Standard Blocking technique is applied to block these properties. Since AL1N can have multiple numbers of the AL1 in them, the AL1N should be split before merging with the Postcode, and the fields should be concatenated as a BKV in Standard Blocking.

The Sliding Comparison Windows uses a window size of 5 and a minimum Levenshtein similarity of 0.7. These parameters can also be tweaked but serve as a proof of concept for applying augmentations. These

	Isolated Runs								Combined Runs								
Blocking Methods	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
MinHash_Name	✓								✓	✓	✓	✓	✓	✓	✓	✓	✓
SB_Phone		✓							✓							✓	
SB_Email			✓							✓						✓	
SB_Website				✓							✓					✓	
SB_Postcode_AL1N					✓							✓				✓	✓
SCW_Phone						✓							✓				✓
SCW_Email							✓							✓			✓
SCW_Website								✓							✓		✓

Table 5.2: The Augmentation Run Configurations

	Blocking Time (s)	Pair Completeness	Pair Quality	Reduction Ratio	F_1-score (PC, PQ)
Run_1	28.57814	0.74639	0.88767	0.99880	0.81092
Run_2	0.81822	0.02942	0.88393	0.99950	0.05694
Run_3	0.80320	0.02402	0.89489	0.99933	0.04678
Run_4	0.58852	0.03995	0.84098	0.99859	0.07628
Run_5	0.86911	0.48981	0.82812	0.99922	0.61554
Run_6	1.34205	0.03404	0.82244	0.99938	0.06537
Run_7	1.37425	0.02444	0.73102	0.99916	0.04730
Run_8	1.01433	0.04050	0.81119	0.99852	0.07715

Table 5.3: Measurements of Isolated Augmentation Runs on GT_1 (5 Repetitions)

augmentations are made with domain knowledge in mind. Thus, these blocks are already high in quality. However, if blocks are made that have rows referring to many different real-world objects, then these blocks would negatively impact PQ. For example, if Standard Blocking with just the Postcode is applied as an augmentation, then the created blocks would have a high number of True Positives (TPs) but also many FPs, resulting in many unwanted comparisons [28].

Multiple augmentation runs can be made from these configurations as an isolated or combined augmentation run. To see the effect of Sliding Comparison Window over Standard Blocking, the aforementioned augmentation schemes based on a Sliding Comparison Window are also implemented as Standard Blocking. The created augmentation runs are shown in Table 5.2. Run_1 is the baseline for scheme augmentation with the optimal parameters discussed in Section 5.5.1. Run_{2-8} are runs where the augmentation is isolated, showing the characteristics of solely blocking based on the separate augmentation. Isolating gives insight into the performance difference based on blocking techniques. Run_{9-17} are runs that combine the baseline with other augmentations. Each run is applied on GT_1 with a repetition factor of 5, and the average is taken as the measurement.

5.8.1. Isolated Augmentation Runs

The run configuration of the isolated runs can be seen in Table 5.2. The results of these isolated augmentation runs are shown in Table 5.3. It can be noticed that the baseline run, Run_1, has a high PC and PQ compared to the other runs. Because $Runs_{2-8}$ rely on properties that are not always available, they will have a lower PC and a higher RR than Run_1, as fewer blocks are created since fewer rows can be used within these augmentation schemes. The performance differences between applying the Sliding Comparison Window and the Standard Blocking can be investigated with the isolated runs. Comparing $Run_{2,6}$, $Run_{3,5}$, and $Run_{4,8}$, there is a clear difference between the two blocking techniques. Standard Blocking is faster and has a higher PQ and RR but a lower PC. However, the Sliding Comparison Window consistently outperforms the Standard Blocking on the

	Blocking Time (s)	Pair Completeness	Pair Quality	Reduction Ratio	F_{16}-score (PC, RR)
Run_1	28.57814	0.74639	0.88767	0.99880	0.99749
Run_9	28.83880	0.75165	0.87443	0.99879	0.99751
Run_{10}	28.88144	0.74584	0.87167	0.99880	0.99748
Run_{11}	30.34928	0.75610	0.89026	0.99880	0.99755
Run_{12}	31.18664	0.82913	0.81800	0.99876	0.99797
Run_{13}	31.86963	0.74961	0.87936	0.99883	0.99754
Run_{14}	31.11874	0.74497	0.87584	0.99882	0.99750
Run_{15}	49.23674	0.75439	0.88018	0.99881	0.99755
Run_{16}	73.67435	0.83838	0.81918	0.99876	0.99802
Run_{17}	35.68965	0.83570	0.81373	0.99877	0.99801

Table 5.4: Measurements of Combinative Augmentation Runs on GT_1 (5 Repetitions)

trade-off between PC and PQ with a β of 1. The augmentation that relies on the physical address, Run_4 has a relatively high PC as the not-always-available property is still provided for most rows, as seen in Section 4.2. The PQ is also good, especially with such a high PC, giving the run a relatively high F_1-score.

5.8.2. Combinative Augmentation Runs

Combinative runs merge additional augmentation schemes (Run_{2-8}) with the scheme that relies on the MinHash scheme (Run_1). The run configuration of the combinative runs can be seen in Table 5.2. The results of these combinative augmentation runs are shown in Table 5.4. The F_{16}-score of PC versus RR can be directly compared, as all the runs include the MinHash scheme. The first thing that can be noticed is that when the physical address augmentation is added to the MinHash, then there is a significant improvement in the PC and the F_{16}-score at the cost of the PQ. Run_{10} is the only run with a lower F_{16}-score compared to Run_1, the other runs all are an improvement to the F_{16}-score. All the runs relying on the Sliding Comparison Window ($Run_{13,14,15}$) have a higher F_{16}-score than their Standard Blocking counterpart ($Run_{9,10,11}$), showing that the addition of a Sliding Comparison Window can be used as a way to improve the create blocks in comparison to simple partitioning. In the isolated runs, the Standard Blocking methods were faster, but in the augmentation runs, the combined Sliding Comparison Window methods were faster to compute, as seen in $Run_{16,17}$. Both of the augmentations applied in these runs are an excellent addition to improve the outcome of the blocking phase. The F_{16}-scores are so close that a decision can be made. In the matching phase, the PQ will be improved by classification. Increasing the PC in the subsequent phases is more complicated. Therefore, Run_{16} is selected as the best run despite having a slightly lower F_{16}-score. Choosing the augmentation run with an improvement in PC versus the drop in PQ is a good decision if the trade-offs are similar.

5.9. Benchmark Runs

Benchmark runs will be conducted throughout the ER pipeline to show the effect of implementing various schemes. These benchmarks showcase different characteristics of the blocking phase. The following benchmark runs are conducted:

1. **RunBaseline**: In this run, the blocks are created by simply grouping on the Name property, creating non-overlapping blocks of Accounts.

2. **RunLSH**: Blocks are created with the use of LSH, as described in Section 5.5.1, creating five different blocks for each Account row, causing overlapping blocks of Accounts. This run allows for correcting error mistakes.

3. **RunAugBaseline**: The baseline run is improved by the addition of the augmentations as described in Section 5.8, transforming the created blocks into overlapping blocks and allowing the linking of different spellings of real-world company names by relying on the other available properties.

4. **RunAugLSH**: The same augmentations are included as RunAugBaseline with the same LSH settings as RunLSH. This run creates the highest overlap of matches in Accounts of all the runs, resulting in the highest confidence of correctly deduplicating the Accounts.

Run	PC	PQ	RR	TSU	Comparisons
RunBaseline	0.70740	0.89753	0.999999187	1229959	2,909,319,298
RunLSH	0.73453	0.73436	0.999989479	95051	37,646,059,617
RunAugBaseline	0.81401	0.80092	0.999998693	765217	4,676,244,916
RunAugLSH	0.83425	0.69483	0.999987909	82709	43,264,011,035

Table 5.5: Blocking Benchmark Runs

The results of the benchmark runs are shown in Table 5.5. PC in the blocking phase means how often Accounts with the same labels are found in a joint block. The PCs of these runs are reasonably high. In the most simplistic run, the RunBaseline, the PC is already near 0.707, with a PQ of near 0.898 and a TSU that is theoretically 1,229,959 times faster to calculate than the complete cross-join comparison. The high PC means the entries with the same VATNumber and CoC mostly share the same Name. Exact has a feature that allows inserting equal fields if the company was selected via a company selector, which can explain the high PC. Since in RunBaseline, the blocks are created by just grouping on the Names, having the exact Name in common will only result in two Accounts sharing a block; therefore, it has a high PQ. In RunLSH, it can be seen that the PC goes up, while in comparison, the PQ has a higher negative impact; this can be explained by the larger generated blocks on very common parts within the Names of Accounts. Suppose a randomly permutated LSH block was created on common words, i.e., city names, or professions. In that case, the created block will mainly consist of False-Matches. The lower PQ should not be the primary concern; however, the number of added comparisons is a primary concern. As in the Matching phase, all these Matches will be compared and classified as a True- or False-Match. Increasing the PQ, but with the cost of the extra comparisons. The addition of the augmentations is a great addition, as seen in RunAugBaseline. Instead of non-overlapping blocks, there is now an interwoven connection between accounts residing in multiple blocks. The PC increases, while the PQ decreases, with limited costs for added comparisons. RunAugLSH has the highest PC yet the lowest PQ and RR, resulting in the highest number of comparisons that should be performed in the Matching phase. If no block cleaning is performed, applying the Matching phase on RunAugLSH should take approximately 15 times longer than that of RunBaseline.

6

Matching

In this chapter, we investigate the third phase of the ER pipeline, as detailed in Section 3.3. Different blocks containing False-Matches were created in the blocking phase. This phase builds on the benchmark result as described in Section 5.9, in which classifiers will determine if a Match is a Negative- or Positive-Match. All the Negative-Matches will be discarded, while the Positive-Matches are kept. In Section 6.1, we give each row of Accounts an identifier to express which properties are available for the given row. Allowing for a novel way to handle incomplete data. Afterward, in Section 6.2, we select the property features that will be used for classification. The setup for the classifiers is explained in Section 6.3. In Section 6.4, four different classification models are tried and compared with each other to select the best-fitting model. Finally, in Section 6.5, The best classification models found in the chapter are applied to the created blocks, and their results are discussed.

6.1. PropertyMask

Each filled-in row of the Account data set is not enforced to have all their properties filled in, as shown in Section 4.2; therefore, classification of the data set is not possible on its own. Missing properties form a tough challenge. When creating the features, some cannot be calculated and, therefore, will be missing. An imputer[1] can be put in place to fill in the missing values. However, there are only a few options to set the imputer to:

- **Static**: In static imputing, one can assign a static value as the feature value if one of the properties is missing. Simply setting it to a value of zero (Negative-Match) is incorrect, as an unknown property does not mean the Match is a Negative-Match.
- **Mean/Median:** Using the Mean or Median to fill in the missing feature value makes it more fair for unknown values. Using the mean or median will give it a value close to the boundary, making it easier to flip to a Negative-Match or Positive-Match. The only problem is that if all the properties except the Name property are unknown, it results in almost all values being on the boundary position, making it highly likely that the Match will be classified as a Positive-Match, which is erroneous.

As a result, creating a classifier with an imputer to correct the many missing features will not generate the desired outcome. For this reason, instead of a single classifier, it is possible to create multiple classifiers, and each of these classifiers is solely created for a unique set of known features. This splitting of the data set is ideal, as the data set is too large to work with as a single unit. Implementing a divide-and-conquer mentality will reduce the memory requirements while processing the data but can increase the time required to process the data. Therefore, a balance is required between splitting and not splitting the data set.

Since the Name property is always available, each classifier can always rely on that property. However, not all the other properties are always available. The number of classifiers we need to train is 2^n, in which n is the number of properties that are not always available. The more classifiers that are trained, the better the predictions will be on the data set. However, more classifiers come at the cost of additional training time. Training of these classifiers is only done once and can be used on the current and future versions of the

[1] `https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.ml.feature.Imputer.html`

data set. For this data set, it was decided to use 8 different properties, which will require $2^8 = 256$ different classifiers to be trained. These 8 properties could be used to classify the Match adequately in the data set. Using more properties will take considerably more time to train, and using fewer properties is not ideal as that will make the Accounts poorly represented. It is advisable to harness as much information from the data set as possible to make the result accurate. Lowering the n will make it so that not all valuable information is used.

It is also possible to use different classifier setups; for example, only train classifiers up to a certain number of available properties. By only using the m most important properties out of the n available properties, reducing the total number of required classifiers at the cost of some precision is possible.

The final properties selected for the classifiers are **Name**, **Website**, **Postcode**, **Email**, **Phone**, **City**, **State**, **AL1S**, and **AL1N**. To decide which of the 256 classifiers to use, each InverseID gets a PropertyMask. This PropertyMask uses bits as a boolean representation of the properties available for the given InverseID. Each property gets its own BitMask assigned $b(i) = 2^i$, in which i is the order of the property, and b is an integer representing the i-th bit. The Properties and their BitMasks in binary and decimal representations are shown in Table 6.1.

Index	**Property**	**BitMask$_2$**	**BitMask$_{10}$**
7	Website	10000000	128
6	Postcode	01000000	64
5	Email	00100000	32
4	Phone	00010000	16
3	City	00001000	8
2	State	00000100	4
1	AddressLine1Street	00000010	2
0	AddressLine1Number	00000001	1

Table 6.1: BitMask for each Property

The PropertyMasks depend on the available properties of the row, either available or unavailable. Therefore a two-case function $b(r,i)$ is implemented:

$$b(r,i) = \begin{cases} b(i) & \text{if } i \text{ available in } r \\ 0 & \text{else} \end{cases}$$

For each row, it is now possible to calculate their PropertyMask as follows $\sum_{i=1}^{n} b(r,i)$; since each of $b(i)$ represents a unique bit, summing the bits together yields the PropertyMask that represent the availability of properties for a given row r. For a comparison, features are created from the information of two different Accounts. To retrieve the classifier that is trained on the available features, it only takes a single bitwise-AND operation to retrieve the correct classifier as follows:

$$\text{ClassifierMask}(r_1, r_2) = r_1 \,\&\, r_2$$

Now, one can group the Matches based on the ClassifierMasks, creating 256 subsets in which all the properties are known for the given classifiers. Instead of imputing the missing values, different classifiers will be trained to work with actual values. Each Match created by the blocking phase can now be passed to their specialized classifier, and a decision can be made if it is a Positive- or Negative-Match.

6.2. Feature Selection

Classifiers are trained on Matches between two different Accounts; it can either result in a Positive- or a Negative-Match, classifying the comparison. However, a classifier cannot be applied to purely text data; the differences between the text of a comparison must be transformed into features.

Within the PySpark environment, the more advanced fuzzy matching features, as discussed in Section 2.8, were not provided. Implementing them would mean User Defined Functions (UDFs) need to be used, which are inherently slow methods and will not work for this big data setting. In this setting, it is crucial to engineer

fast and reliable features. The two implemented features satisfy these criteria while possible in the PySpark environment. These features are Levenshtein and Jaccard similarity, and both similarities will be in the range of 0 to 1, from dissimilar to similar, respectively. Even though these features are generic, they still provide good results [3]. The features are created from the properties of a comparison. If both Accounts of the comparison have the given property available, then that property can be transformed into a feature. The selected properties to become features are shown in Table 6.1.

6.2.1. Levenshtein Similarity

The Levenshtein distance is the number of characters, which does not have a precise range and is dependent on the size of the strings. In order to compare the Levenshtein distance, the distance is divided by the maximum string size of the two strings, as seen in Eq. (6.1), and this division will return the distance. The inverse of this distance is taken to get the similarity. The Levenshtein similarity allows for correcting spelling mistakes. The words with the most characters of the string will have the most impact on the Levenshtein similarity.

$$\text{lvsMax}(s_1, s_2) = 1 - \frac{\text{LevenshteinDistance}(s_1, s_2)}{max(|s_1|, |s_2|)} \tag{6.1}$$

6.2.2. Jaccard Similarity

Since the properties of the Accounts are strings, there are multiple ways to transform them into a set of tokens. One benefit of Jaccard is that two strings expressing the same words but are ordered differently will still have a high similarity, which is not necessarily the case for lvsMax.

The methods tried are splitting the strings into their word representation and splitting as an N-gram representation. Both implementations give different results. Splitting the string into words will give the same weight per word independent of the word size; however, it does not allow for spelling mistakes. If one character in a word differs, it will not be similar. If the text is split into N-grams, it will allow for spelling mistakes; however, the larger words will have more impact on the Jaccard similarity, as more N-grams can be created from the larger words compared to the smaller words, causing the larger words to dominate the smaller words in terms of importance.

Both options for splitting the strings into sets are valid. However, since both N-grams and split words rely on the same property, the feature value they produce will be heavily correlated. Therefore, one of these settings should be selected as the base for the Jaccard Similarity. Since lvsMax already works to address spelling corrections, it is opted to use Jaccard similarity on split words. If the N-gram option were selected, then large words would overshadow smaller words throughout the different features, which is not desired as words with just a few characters can give as much information as words with plenty of characters.

Jaccard similarity can also be used to check if two strings are exact matches of each other. If both texts only consist of a single word, after splitting the text into the word representation, there are two lists with both a single word; these two words can either be equal or not equal, essentially the same as checking for an exact match. Properties that always consist of a single word will, therefore, have a Jaccard similarity of 0 or 1.

6.2.3. Feature Importance

Finding the feature's importance is essential to see if the created feature is well-defined. To understand which features can be used for this problem, a simple logistic regression model is trained on the labeled Property-Mask$_{255}$ to find the feature's importance. All the properties are available in this model and bound to a label. The training and test sets are evenly distributed in True- and False-Matches in this model. In permutation feature importance, a single feature is permutated independently throughout the test set, and the difference in accuracy is measured. The isolated feature is important if the accuracy drops relatively much after permutating a given feature. A feature excluded in advance is lvsMax on AddressLine1Number (AL1N); as street numbers do not consist of many digits, applying error correction on top of a few digits will not make sense, as different numbers can still give a high lvsMax similarity.

In Fig. 6.1, the feature importance of different features is shown. For each feature, 20 permutations are performed, and the average is taken as the feature's importance. It can be noticed that Jaccard on split words (sw) of Name and Website are highly important features. They show to be exponentially more important compared to the other features. It can also be noticed that both Jaccard and lvsMax can be used as a feature, as neither is always more important than the other for all the features. For some features, the Jaccard is more important, while for others, the lvsMax is of more significance. The last that can be noticed is that some features are unimportant and should be removed. For all 20 runs, the accuracy improved after permutating

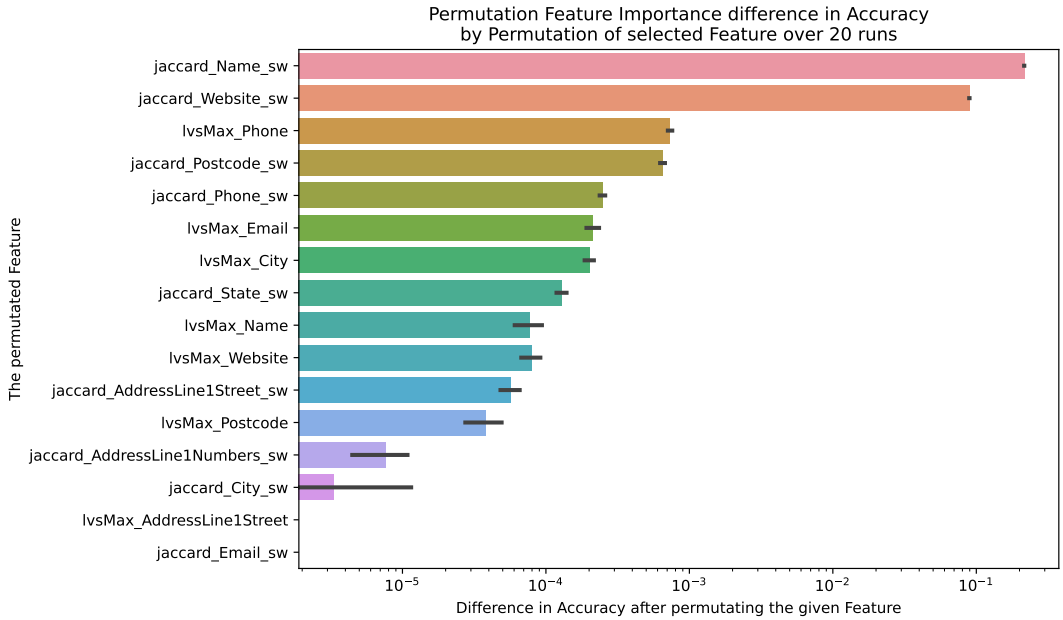

Figure 6.1: Feature Importance on separated Features

lvsMax of AddressLine1Street and Jaccard of Email, and for some of the runs, the accuracy improved after permutating Jaccard of City. It shows that if these features were to be removed, the accuracy would increase, which is what was done.

After removing these poor features, a combined feature importance is created, in which all features that correspond to the same property are merged. Instead of independently permutating the feature importance as shown in Fig. 6.1, they are permutated per property. The result of the feature property importance can be seen in Fig. 6.2. It can be seen that the Name and Website are by far the most important properties to classify if the Match is a Negative- or Positive-Match. City, State, and AL1 are the least important features; this can be explained by the fact that these features all express a physical address. If one of these features gets permutated, there is still information on the physical address in one of these other features. The classifier learned that a mix of these features results in the highest accuracy, so permutating a property corresponding to the physical address will not have high importance, as information will remain in these other features. The feature property importance order was selected as the order of PropertyMasks, as seen in Table 6.1.

6.2.4. Feature Correlation

In Section 6.2.3, it was explained that features expressing the physical location of companies are probably highly correlated. A Pearson correlation plot is shown in Fig. 6.3 to allow for further investigation of the correlation between the features. Pearson correlation is used when the features are on an interval scale, and this correlation is expressed in the range of -1 to 1, negatively to positively correlated. If the Pearson correlation is near 0, then there is no correlation. Unlike feature importance, which requires a model to be trained, no training is needed to calculate feature correlation. The correlation can be calculated across all available samples, not just the labeled samples from PropertyMask$_{255}$. The features in Fig. 6.3 are those without the bottom three features of Fig. 6.1.

The desired features should have a high correlation with the Match and a low correlation with other features. If there is a high correlation with Match, then it means if the feature increases, the chance of a True-Match, on average, will also increase. If the feature is not correlated to other features, that feature expresses a unique context. The correlation plot has a few characteristics that can be noticed. The first is that the lvsMax and Jaccard of the same property are highly correlated, which makes sense as they are both based on the same property; thus, they must be highly correlated. However, although they are highly correlated, they are not the same, meaning both features express the property differently. There is also a relatively high correlation between features expressing the physical location of the Accounts. The features City, Postcode, AddressLine1Number, AddressLine1Street, and State are relatively highly correlated since these features all express the physical location. Since the Accounts data set is incomplete, and most properties are unavail-

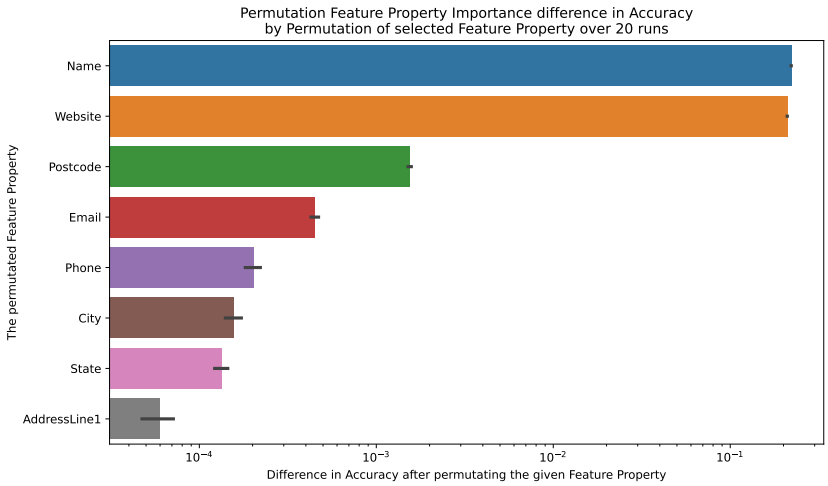

Figure 6.2: Feature Importance per Property

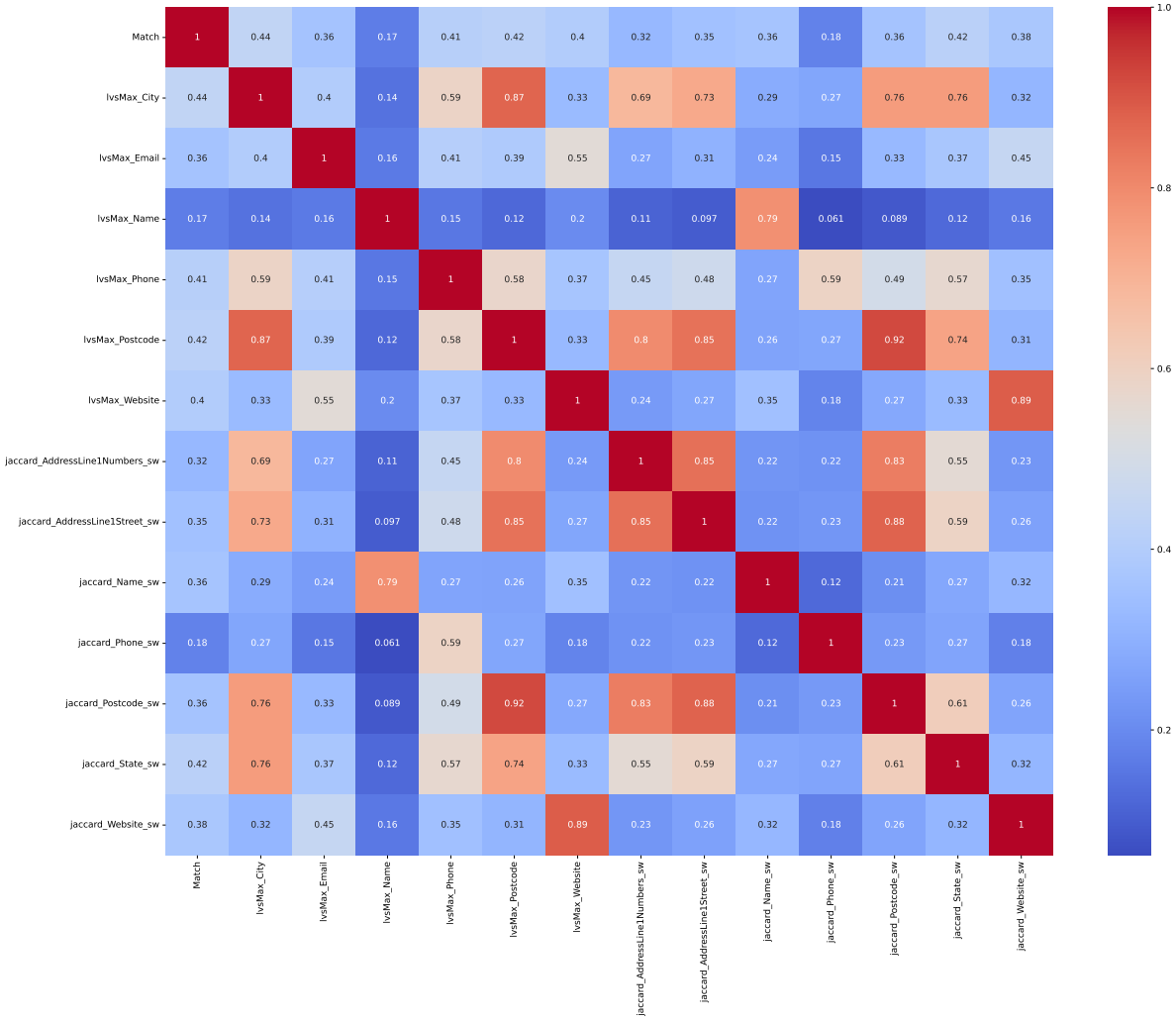

Figure 6.3: Pearson Correlation of all the Features

able per Match, a high correlation means that even though some properties are unavailable, they can still be used to express physical addresses independently. The lvsMax of Name is an outlier, as it only has a high

correlation to itself and the Jaccard of Name. The correlation to the other properties is relatively low. All the Names follow either the English or Dutch language. Even though these Names are different, they could still follow generic language patterns, meaning they can consist of the same groups of characters. Very common words, such as a city name, are part of most Names. This results in the feature value increasing while it is not necessarily a Match. The lowest correlation between a Match and a feature is 0.17, and all the features are not heavily correlated with each other; therefore, it can be concluded that the selected features are a fair mix of feature correlations and can be used as features for training classifiers.

6.3. Training and Testing sets

By using the knowledge of the previous sections, different models can be trained and compared with each other. Ultimately, one Model is selected as the best performing and used for all the benchmark runs.

6.3.1. Labeled Match Distribution

There are a limited number of labeled Accounts. If these labeled Accounts are cross-compared with each other, it results in mostly False-Matches, while there will only be a limited number of True-Matched, as seen in Table 3.2. There are roughly 4000 times more false than true labeled matches. Because of this imbalance of labeled matches, classifiers will, most likely, always predict a Match to be a Negative-Match as this will give a very high accuracy, which is erroneous behavior.

It is crucial to balance the data so that a fair decision can be made on a Match if it is Positive-Match or a Negative-Match. Therefore, the number of True-Matches gets sampled out of the False-Matches, creating an even split of true and false labeled matches. There is only one problem: the majority of False-Matches are two completely different Accounts, meaning that all the features of such a False-Match are likely to be near zero. If classifiers get trained on False-Matches that are almost always dissimilar, then if a single feature is above average, it will, most likely, be classified as a True-Match, generating a lot of FPs.

More complicated False-Match cases should be used to train the classifiers, and these problematic cases are already available, namely the results of RunAugLSH. In RunAugLSH, blocks are created of highly similar Accounts, False-Matches within these blocks are, therefore, complex cases to classify and ideal to use as the training set of the classifiers.

Now that a fair balance of True-Matches and False-Matches exists, they can be divided into a train and test set. For the train and test set, an 80/20 split is used. However, dividing the data set into a training and test set is more complex with a multi-model schema, as discussed in Table 6.1. Each classifier has its own ClassifierMask, representing the properties that must be available for using that classifier, as discussed in Section 6.1.

The labeled Accounts are not evenly distributed across the PropertyMasks. Some PropertyMasks have more labeled Matches than others. PropertyMasks with more available properties can also be used as labeled matches for a given PropertyMask. All the Matches of the highest PropertyMask, meaning all properties are available, can be used for every one of the classifiers. Simply dropping the unused properties in that classifier will create a valid comparison for the classifier. The idea behind this implementation is that an Account referring to a particular real-world company will still refer to that same company even if the user forgot to fill in a property, meaning that dropping a property will not change the real-world company it refers to. The equality of Eq. (6.2) needs to hold for a given MatchPropertyMask and a ClassifierMask so the classifier can use that Match for its knowledge base.

$$\text{MatchPropertyMask} \,\&\, \text{ClassifierMask} == \text{ClassifierMask} \tag{6.2}$$

Since there are 256 different PropertyMasks available, plotting them all inside a single plot will make it too complex to comprehend. Therefore, it is opted only to investigate the most interesting PropertyMasks. The most interesting PropertyMasks to investigate are in which no property is available (PropertyMask_0), the case in which only a single property is available ($\text{PropertyMask}_{2^n}$; $0 \leq n < 8$), and finally, the case in which all properties are available ($\text{PropertyMask}_{255}$).

After creating the 80/20 train and test set for all the PropertyMasks with their valid Matches, the total set sizes per investigated PropertyMask are shown in Fig. 6.4. This plot shows that the training and test sets differ across the PropertyMasks, which makes sense since PropertyMask_0 includes all the other PropertyMask training and testing sets, and the training and testing set of $\text{PropertyMask}_{255}$ is included in all the other PropertyMasks. It can be noticed that, therefore, the set with the smallest training and testing set still has

two million samples in its training set, of which one million are True-Matches and the other one million are difficult False-Matches.

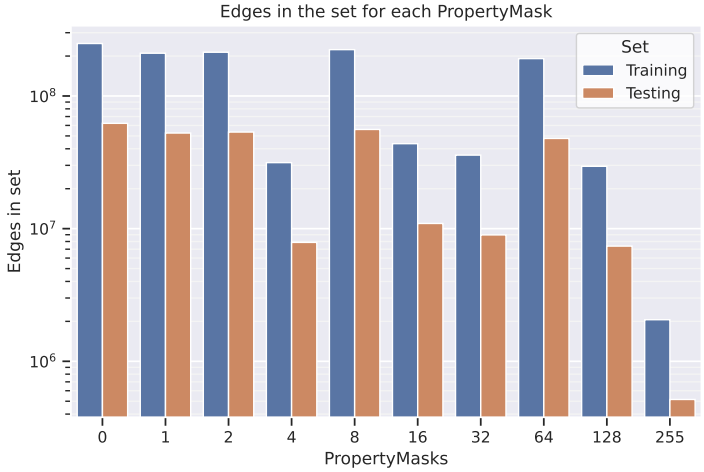

Figure 6.4: The Training and Test set sizes of the inspected PropertyMasks

6.4. Classification Models

Different classifiers can be used to classify a problem; some are simple, while others are complex. More flexible classification models can use certain features better than a simpler classifier. Investigating different models for a classification problem and selecting the most fitting one is essential. Four classification models were tested: LogisticRegression, DecisionTree, RandomForest, and GradientBoostedTree. These are listed from least complex to most complex. The classifiers are trained on the investigated PropertyMasks as described in Section 6.3. After finding the best-performing model, it is applied to all the different PropertyMasks.

When the recalls of the different models are compared in Fig. 6.5, it can be seen that the LogisticRegression performs the worst on average, the DecisionTree and the RandomForest perform roughly equally, and the GradientBoostedTree performs the best on all the inspected PropertyMasks. The precision of the models follows the same pattern as how the models perform for the recall, which can be seen in Fig. 6.6. The GradientBoostedTree has the highest precision of all the inspected PropertyMasks. Since a fair distribution of labels was used, the accuracy can also be measured to indicate the performances of the classifiers. The accuracy for the classifiers can be seen in Fig. 6.7. The LogisticRegression has the worst accuracy, followed by DecisionTree and RandomForest, which both have similar accuracy. The GradientBoostedTree has the highest accuracy across all the PropertyMasks. So far, the GradientBoostedTree performs the best in recall, precision, and accuracy. The only downside of using the GradientBoostedTree as the classifier is that the time taken to train the model is significant, as seen in Fig. 6.8.

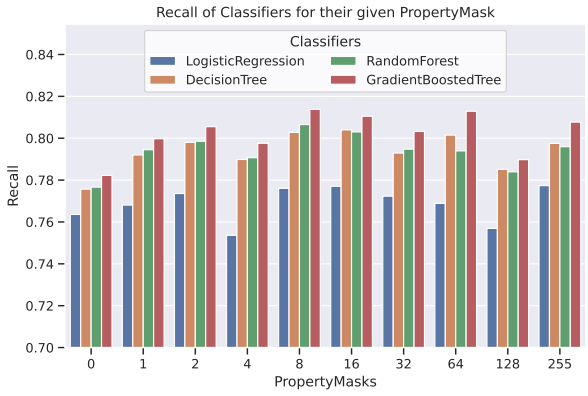

Figure 6.5: Recall of the different Models for the inspected PropertyMasks

Figure 6.6: Precision of the different Models for the inspected PropertyMasks

Figure 6.7: Accuracy of the different Models for the inspected PropertyMasks

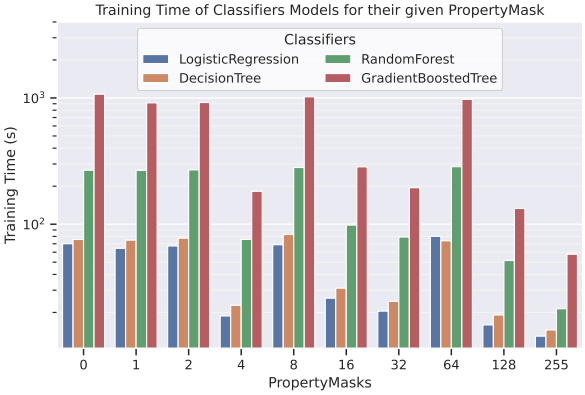

Figure 6.8: Training Times of the different Models for the inspected PropertyMasks

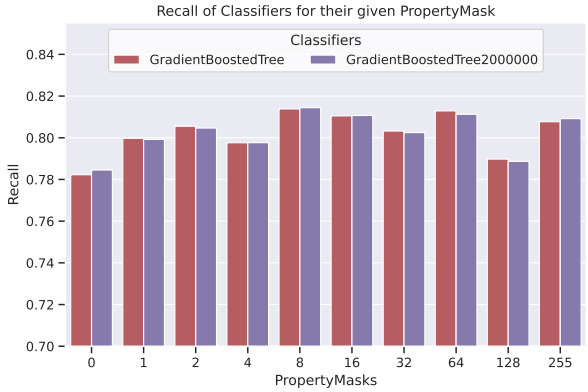

Figure 6.9: Recall of the different Models for the inspected PropertyMasks

Figure 6.10: Precision of the different Models for the inspected PropertyMasks

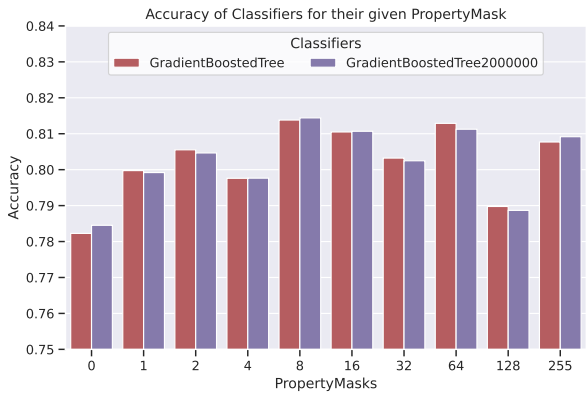

Figure 6.11: Accuracy of the different Models for the inspected PropertyMasks

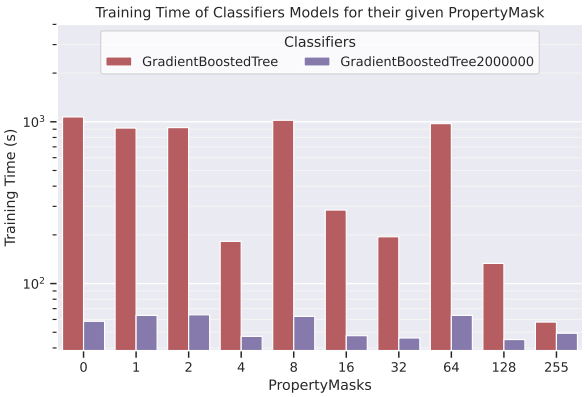

Figure 6.12: Training Times of the different Models for the inspected PropertyMasks

The models are trained on the entire data set, which has more than enough data samples to classify the problem. Therefore, it might be possible to sample each classifier's training set to a given size. The PropertyMask with the least number of training samples is $\text{PropertyMask}_{255}$, as seen in Fig. 6.4, since it requires all the properties to be available. All the training sets of the other PropertyMasks are a superset of $\text{PropertyMask}_{255}$. The training size of $\text{PropertyMask}_{255}$ is slightly over two million different matches with a 50/50 distribution of a True- and a False-Match. All the other classifiers relying on another PropertyMask are guaranteed to have a training set size of at least that of $\text{PropertyMask}_{255}$. Therefore, it is possible to use a fixed sample size across all the classifiers. All the classifiers have a train set sampled to one million True-Matches and one million False-Matches, resulting in a training set size of two million, and the model used is the best-performing model with the worst training time, the GradientBoostedTree. The $\text{GradientBoostedTree}_{2000000}$ has the same settings as

GradientBoostedTree but a training set of two million entries for all the different classifiers trained. It can be noticed that sampling the training set to a fixed size yields a near identical recall, precision, and accuracy, as seen in Figs. 6.9, 6.10 and 6.11, respectively, but with a training time that is significantly less than that of GradientBoostedTree, as seen in Fig. 6.12. With just a few features, the classification problem is not overly complex, meaning the models trained can still be effective with just a limited number of samples. Therefore, the $\text{GradientBoostedTree}_{2000000}$ is selected to be the classifier for all the PropertyMasks, as it shows to have a high recall, precision, and accuracy, in combination with a now manageable total training time.

6.5. Benchmark Runs

The benchmark runs are split into two parts. First, a benchmark is run on the raw data set, in which the data set is prepared for classification. Secondly, the classification of the runs is benchmarked.

6.5.1. Creating Raw Benchmark Runs

Raw Matches have yet to be classified as True-Matches or False-Matches. These Raw Matches are created and stored per MatchPropertyMask so that each classifier can access the correct MatchPropertyMasks without recalculating all the Raw Matches for each classifier. The Recall, Precision, and RR of Table 6.3 should be identical to the results of Table 5.5 for the runs that do not have a pruning phase applied. Since RunBaseline and RunAugBaseline do not have a pruning step applied, the measurements are the same.

In Table 6.3, the number of comparisons of the runs that include LSH are significantly higher than those of the baseline. Therefore, a pruning step can be applied, as discussed in Section 2.6. In this case, since the LSH causes an increase in comparisons, it is opted only to prune the blocks created by the LSH. The Pruning method applied is that blocks created by LSH that have more than 1000 InverseIDs in them will be ReBlocked by the Name. In essence, this will reduce the number of comparisons required, but it will lose the spelling correctness of the largest blocks. However, since the largest blocks are the most likely to be erroneous, as discussed in Section 5.9, doing so does not cause a significant loss. In Table 6.2, applying this ReBlocking on the LSH runs will roughly reduce the total number of comparisons by a factor of four while significantly increasing the precision and only negligibly impacting recall.

Run	Pruning	Recall	Precision	RR	TSU	Comparisons
RunLSH	×	0.73453	0.73436	0.999989479	95051	37,646,059,617
RunLSH	LSH-RB1000	0.72155	0.86790	0.999997401	384792	9,299,411,424
RunAugLSH	×	0.83425	0.69483	0.999987909	82709	43,264,011,035
RunAugLSH	LSH-RB1000	0.82110	0.78169	0.999996896	322153	11,107,549,706

Table 6.2: LSH Pruning Differences

The final Raw Benchmark Runs can be viewed in Table 6.3. With the applied pruning settings on the LSH it shows a good comparison between the runs. Implementing LSH over the baseline will provide higher recall but lower precision while creating more comparisons. Adding augmentations to the blocking scheme will increase the recall but lower the precision. The number of comparisons is the final number of comparisons that need to be classified by the classifier, meaning that the RR and TSU are final for the given runs. That the TSU is final means the slowest run is 322,153 times faster than cross-comparing all the Accounts, while the fastest run is 1,229,959 times faster. This significant speedup will allow the matches to be classified within hours instead of years.

Run	Pruning	Recall	Precision	RR	TSU	Comparisons
RunBaseline	×	0.70740	0.89753	0.999999187	1229959	2,909,319,298
RunLSH	LSH-RB1000	0.72155	0.86790	0.999997401	384792	9,299,411,424
RunAugBaseline	×	0.81401	0.80092	0.999998693	765217	4,676,244,916
RunAugLSH	LSH-RB1000	0.82110	0.78169	0.999996896	322153	11,107,549,706

Table 6.3: Matching Raw Benchmark Runs

6.5.2. Classifying the Raw Benchmark Runs

The Raw Benchmark Runs must be classified, requiring the classifiers described in Section 6.4. The Matches can be classified after applying each specific classifier with its unique ClassifierMask on the corresponding PropertyMatchMasks, as seen in Table 6.4. The precision is calculated with the GT, while the classifiers classify the Matches as Positive or Negative. After classifying, the general trend is that recall decreases slightly while precision increases. The most notable is the number of Negative-Matches. The Negative-Matches are the Matches that will be removed, as they are classified not to be a True-Match. For the RunBaseline, there are just a few Negative-Matches, which makes sense as blocks are created of Accounts having exactly the same Name. The classifiers are trained on all the Matches labeled as a True-Match, and when the Names are identical, it is most likely to be a True-Match. The Negative-Matches are most likely Matches with the same name, yet all their other properties are filled in and differ from one another. There are more Negative-Matches in the runs that rely on LSH, which makes sense as introducing spelling correctness can cause many FPs. The number of Positive-Matches is important, as in the clustering phase, more True-Matches means more knowledge about the graph, meaning more confidence in applying further methods. The RunLSH and RunAugBaseline are close in the number of True-Matches, even though RunLSH does not implement augmentation.

Run	Pruning	Recall	Precision	Positive-Matches	Negative-Matches
RunBaseline	×	0.70742	0.89755	2,909,195,858	123,440
RunLSH	LSH-RB1000	0.72019	0.89188	3,495,198,921	5,804,212,503
RunAugBaseline	×	0.79295	0.82460	3,679,956,373	996,288,543
RunAugLSH	LSH-RB1000	0.79828	0.82078	4,217,941,009	6,889,608,697

Table 6.4: Matching Cleaned Benchmark Runs

7

Clustering

In this chapter, we investigate the fourth and final phase of the ER pipeline, as detailed in Section 3.4. This phase builds on the benchmark result described in Section 6.5. First, in Section 7.1, we will be investigating how to transform the graph of Positive-Matches to an EntityMapper. In the next section, Section 7.2, NoiseRemoval is applied to the matches. NoiseRemoval relies on looking at the similarity of neighborhoods between vertices that have a Positive-Match. Afterward, missed True-Match edges can be discovered, as discussed in Section 7.3. In MissingLinks, the neighbors of the vertices are used once again, but this time to discover new edges. Finally, the results of applying these clustering methods on the runs will be discussed in Section 7.4.

7.1. Entity Mapping

The output of the matching phase is a graph in which the vertices are Accounts, and the edges the Positive-Matches. The ER pipeline should serve as a solution that can be used directly to deduplicate the Accounts data set. The input of the ER pipeline is a table of Accounts. The desired output of the ER pipeline is that Accounts that are duplicates of a real-world company will have the same identifier, called EntityID. Therefore, a mapping table that maps AccountIDs to EntityIDs in an n-to-1 relation should be created. Within this mapper, there is only a single instance per AccountID in the mapping table, while there can be many deduplicated EntityIDs since many AccountIDs should be able to be mapped to a single EntityID. Therefore, the graph should be transformed into a mapping.

Cliques can be used to create the mapping. Cliques are groups of vertices within the graph that share an edge with all the other members of their group. These cliques can serve as the basis for creating the Entities. Each clique receives a unique identifier, their EntityID. Afterward, connected cliques can be compared and merged into a larger Entity. However, there is a problem with using cliques: finding cliques within a graph is a notoriously hard problem [13]. Discovering cliques may be possible for smaller data sets, but it is not suitable for big data as it is unscalable.

Creating Entities can be achieved by locating all the disconnected sub-graphs from the graph. These disconnected sub-graphs then get their unique EntityID. Using this technique creates loosely defined cliques out of the disconnected sub-graphs. Since there are only edges between vertices that are a Match, in theory, the graph should consist of many disconnected sub-graphs that all refer to a unique real-world company. However, since the classifiers rely only on a few features, they are prone to creating noise. A single edge between two disconnected sub-graphs will connect these sub-graphs; such an edge is called a bridge edge. Removing the bridge edges will increase the total number of disconnected sub-graphs. In the Blocking phase, an elaborated schema is implemented to create as much overlap between the Accounts as possible on all their provided properties, and this means that there is a high chance of creating many bridge edges, which might create a single large clique out of the graph. Indeed, while the graph consisted of a majority of bride edges, without any additional transformation on the graph, the deduplicated data set consisted of a single EntityID containing almost all the AccountIDs, drastically impacting the performance of the generated EntityMapper, as can be seen in Table 7.1. The only run in which not applying noise removal is viable is the RunBaseline. In this run, the blocks are non-overlapping, disallowing the creation of a single large cluster with a single EntityID. However, in the runs with overlapping block generation, a single large Entity is created containing most of the AccountIDs. This results in an excellent recall as the majority of the AccountIDs are mapped

together via this massive EntityID, while the precision is, because of this large cluster, extremely inadequate. When using disconnected sub-graphs to generate EntityIDs, applying noise removal for schemes relying on overlapping blocks is a must, as can be concluded from this experiment.

Run	Clustering	Recall	Precision	F_1-Score	EntityIDs	Size
RunBaseline	NoClustering	0.70742	0.89754	0.79122	27,205,371	32.16%
RunLSH	NoClustering	0.80423	0.00037	0.00074	11,462,887	13.55%
RunAugBaseline	NoClustering	0.99604	0.00026	0.00052	7,906,152	9.35%
RunAugLSH	NoClustering	0.99865	0.00025	0.00050	4,950,982	5.85%

Table 7.1: EntityMapping Measurements of applying NoClustering

7.2. Removing Noise

The bridge edges should be found and removed to create a graph of many disjoint components that can be transformed into EntityIDs. The classifiers used in the Matching phase are sufficient in classifying the potential that two Accounts are a match; however, if companies share, for example, the physical address, then this can introduce a FP. Neighbors in common can be used to mitigate these FPs. If the two Accounts of an edge have little neighbors in common, then it is most likely a bridge edge.

In smaller data sets, it is possible to calculate precisely the number of neighbors the two Accounts of an edge have in common. However, in the case of big data, it is not feasible to calculate the neighbors in common for each Account since an Account will have an edge to many other Accounts. Therefore, a better idea is to approximate the neighbors in common. MinHash can be implemented again to approximate the common neighbors; this implementation differs from how it was applied in Chapter 5. Previously, N-grams were utilized to create the vocabulary on which MinHash is applied. To remove noise, the vocabulary will consist of all the InverseIDs. There is only a problem with the PySpark implementation of MinHash, and that is the vocabulary size of LSH has an undocumented maximum size of 5 million. Since there are over 63 million InverseIDs, the lesser connected InverseIDs will be discarded. There is a big tail of poorly connected InverseIDs; dropping these will likely not have a significant impact.

First, a Neighborhood needs to be constructed for each InverseID. The Neighborhood consists of all the other InverseIDs that the InverseID has an edge to, as denoted by Neighbors(InverseID), and the InverseID itself. Adding the InverseID itself to its Neighborhood is crucial. If InverseID1 and InverseID2 have the same set of neighbors, excluding InverseID1 and InverseID2 in common, then the set of Neighbors(InverseID1) is not equal to Neighbours(InverseID2). If the Neighborhood is used, which appends the InverseID, then Neighborhood(InverseID1) and Neighborhood(InverseID2) would be equal. Secondly, the vocabulary of each InverseID can be constructed, which is a one-hot encoding of its Neighborhood, giving a large sparse vector per InverseID. Thirdly, several hashes can be generated using MinHash in combination with the vectors provided by the vocabulary. The final step is to retrieve for each edge(InverseID1, InverseID2) the number of hashes InverseID1 and InverseID2 have in common. The edge should be discarded if it is below a certain threshold.

After removing the noise with LSH, disconnected sub-graphs can again be discovered. The bridge edges can now be found and removed from the graph using spatial information. Locating sub-graphs without these bridge edges will make it so that most InverseIDs do not end up in a single EntityID but spread across several EntityIDs.

7.2.1. NoiseRemoval Benchmark

To remove noise in the edges, 40 hashes are generated for each InverseID, which allows approximating the similarity of the Neighborhood of the vertices that are connected via an edge. The ApproxSim is calculated by the number of hashes the vertices have in common, divided by the total number of hashes generated per vertex. Even though the results of RunBaseline were already sufficient without any clustering applied, they can still be improved. Therefore, a graph is generated to look into the ApproxSim versus the frequency and the precision, as can be seen in Fig. 7.1. Each of the possible ApproxSim values are buckets that have their own precision and number of True-Matches inside of them labeled frequency. It can be noted that the ApproxSim of RunBaseline is relatively high, as it starts at 0.675. The distribution of edges per ApproxSim bucket grows exponentially to fully approximated equal Neighborhoods. The precision is calculated per bucket, which can

be used to investigate which threshold of ApproxSim should be used to remove the most noise. Selecting a threshold of the ApproxSim will cause all the buckets with an ApproxSim lower than the given threshold to be discarded. Since the number of edges grows exponentially per larger ApproxSims, dropping the lower ApproxSims should have a negligible negative impact on the recall. In contrast, it can have a more significant impact on improving the precision in comparison to NoClustering. For the RunBaseline, dropping edges with an ApproxSim below 0.725 is a good idea, as this will keep the higher precision buckets and discard the lower precision buckets.

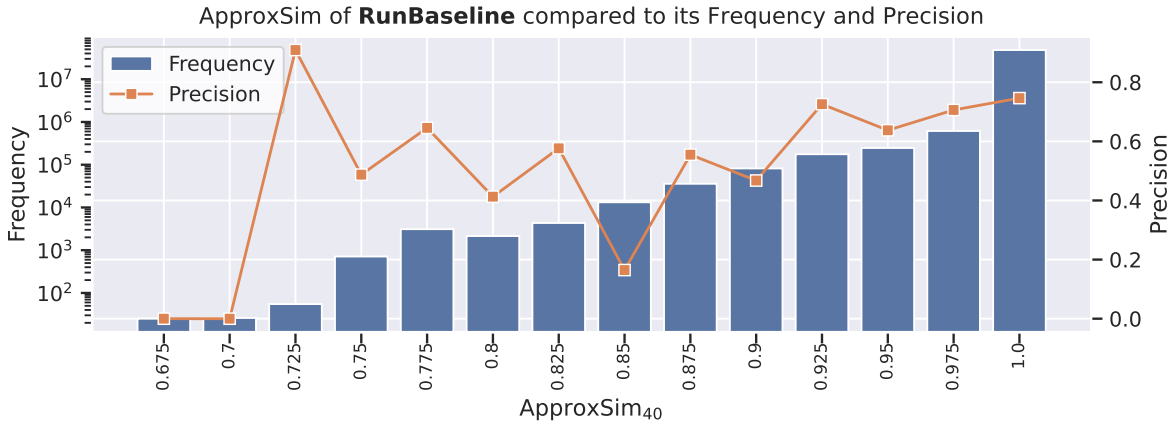

Figure 7.1: The NoiseRemoval characteristics of RunBaseline

In RunLSH, Fig. 7.2, it can be seen that the precision grows relatively linear, which can make it more challenging to decide which threshold to select. The general idea is to select a threshold in which there still is a high distribution of the total number of edges combined with all the buckets being high in precision. Therefore, a local peak is selected at 0.875 as the threshold to keep the edges. The buckets in the range from 0.875 to 1 are high in precision and have the most edges inside this range.

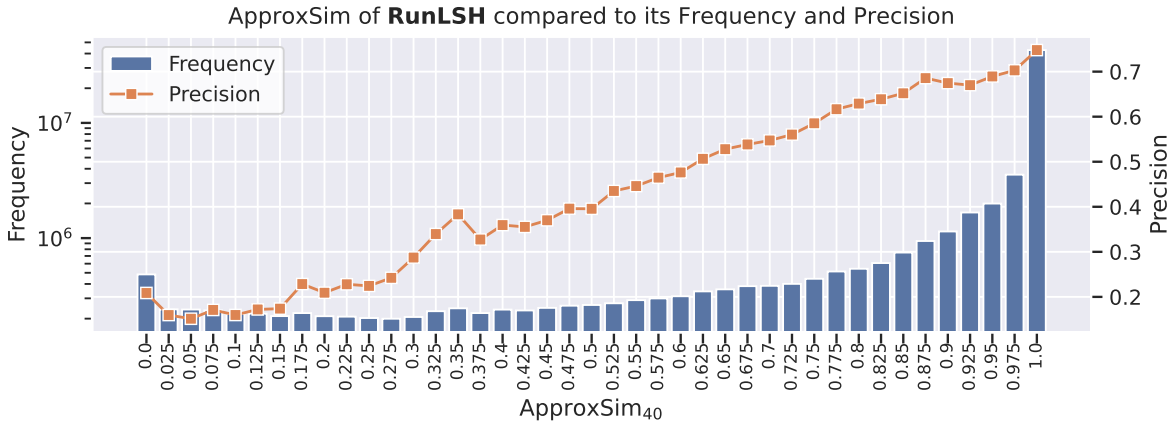

Figure 7.2: The NoiseRemoval characteristics of RunLSH

For the RunAugBaseline, the precision does not grow smoothly per ApproxSim, as seen in Fig. 7.3. Therefore, there is no absolute best threshold to select at first sight. Three local peaks have the most interesting characteristics: 0.4, 0.525, and 0.675. The peak of 0.4 has higher precision than the precision of the ApproxSim of 1 bucket, but the two following ApproxSims have an overall lower precision. The peak of 0.525 does not have this occurrence. The latter peak has high precision and still a high total frequency of the edges, making it a perfect fit to be selected as the threshold. Both of these peaks will be tested.

Finally, in the RunAugLSH, Fig. 7.4, there are two peaks again that are points of interest. These peaks are 0.55 and 0.725. The 0.55 peak shows a resemblance to the results of NoiseRemoval in RunAugBaseline. There is a higher frequency and a high precision at this peak, followed by some ApproxSim of lower precision but

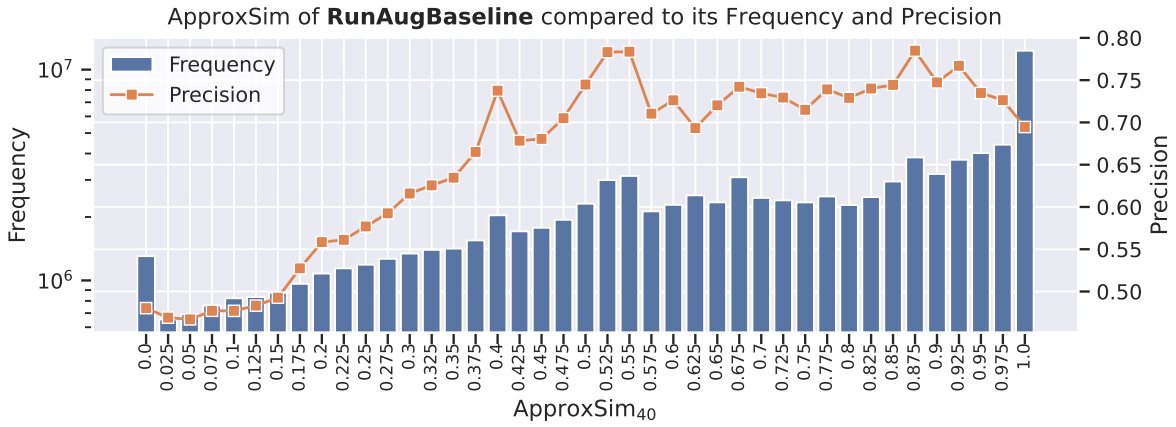

Figure 7.3: The NoiseRemoval characteristics of RunAugBaseline

also lower frequency. Onwards from the 0.725 peak, the precision of the subsequent buckets until the final bucket is larger than the precision of the final bucket.

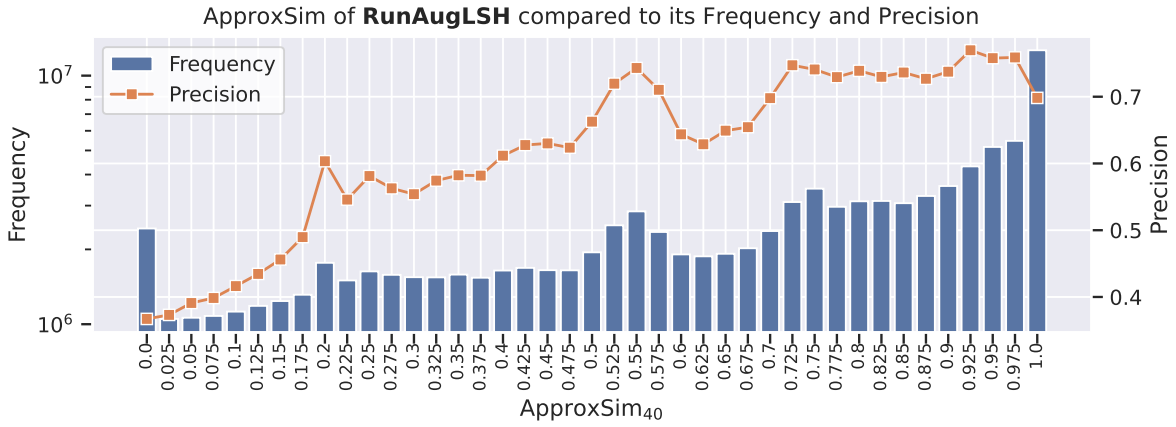

Figure 7.4: The NoiseRemoval characteristics of RunAugLSH

For each of the Runs, the previously selected ApproxSim thresholds are applied and visible in Table 7.2. Alongside these peaks, the strictest form of NoiseRemoval, which has an ApproxSim threshold of 1, is also applied to investigate if the number of hashes used, which was 40, is sufficient. If the measurements of threshold 1 outweigh that of the selected peaks, then the ApproxSim should be calculated with more hashes. The RunBaseline is a unique case in which applying NoiseRemoval will yield worse results than NoClustering, as seen in Table 7.1. This is most likely due to the discussed shortcomings of the MinHash within PySpark. Since there is no overlap between blocks, some blocks will not contain a Positive-Match that is part of the LSH vocabulary, causing the entire block to be disbanded into separate Entities. This also causes the size to increase when applying NoiseRemoval on RunBaseline, which will not happen with overlapping blocks, as with more connections, an InverseID will likely have at least a single Positive-Match that is part of the MinHash vocabulary. In the overlapping blocks, the F_1-Scores increase when applying NoiseRemoval. In RunLSH, the selected peak of 0.875, compared to the threshold of 1, has a significantly higher recall than the loss in precision. This result makes 0.875 an excellent choice to be used as the threshold of NoiseRemoval in RunLSH. In RunAugBaseline, multiple peaks were investigated, showing that a threshold of 0.675 has the highest F_1-Score. This threshold is selected and will be used in further methods of this run. Lastly, in RunAugLSH, it can be noticed that the threshold of 0.525 has poor precision yet high recall. The threshold of 0.725 has a lower recall but higher precision, giving a high F_1-Score. Finally, the threshold of 1 has a low recall and high precision. The threshold of 0.725 is more averaged compared to the other investigated thresholds of RunAugLSH; therefore, the EntityMapper generated with NoiseRemoval of threshold 0.725 will perform better than its counterparts for this run.

During these runs, it can be noticed that a lower threshold will increase the recall, as more True-Matches are kept, while it lowers the precision. A local maximum should be selected that fits the required function of the EntityMapper; if one wants to create an EntityMapper with little error, then a high number of hashes combined with a high threshold should be selected, granting a high precision at the cost of the recall. Otherwise, if the EntityMapper is used for exploratory use, then a lower number of hashes and a lower threshold can be used, giving a high recall at the cost of precision. Each NoiseRemoval run creates a different total number of unique EntityIDs, roughly half to two-thirds in size compared to the number of unique AccountIDs. A smaller size will help other down-the-line models handle fewer unique Accounts, making size an important overall aspect.

Run	ApproxSim_{40}	Recall	Precision	F_1-Score	EntityIDs	Size
RunBaseline	0.725	0.69877	0.89787	**0.78591**	58,194,435	68.79%
RunBaseline	1	0.69640	0.89808	0.78448	58,199,682	68.80%
RunLSH	0.875	0.70715	0.89451	**0.78987**	50,806,031	60.06%
RunLSH	1	0.67113	0.89929	0.76864	52,262,337	61.78%
RunAugBaseline	0.4	0.95093	0.00555	0.01104	44,230,722	52.28%
RunAugBaseline	0.525	0.88675	0.69985	0.78229	45,448,906	53.72%
RunAugBaseline	0.675	0.80875	0.81251	**0.81063**	46,742,748	55.25%
RunAugBaseline	1	0.28568	0.93063	0.43716	48,794,216	57.68%
RunAugLSH	0.525	0.89058	0.61016	0.72417	41,935,974	49.57%
RunAugLSH	0.725	0.77731	0.81855	**0.79740**	44,295,941	52.36%
RunAugLSH	1	0.28979	0.93860	0.44285	48,253,764	57.04%

Table 7.2: The Result of applied NoiseRemoval settings for the Benchmark Runs

7.3. Locating Missing Links

Due to the blocking stage, not all the accounts are compared. The accounts that are most likely similar are blocked into similar blocks, causing the creation of a Match. Therefore, edges can still be missing between different generated Entities. After removing the noise, missing links can be found using MinHash. In MissingLinks, for each vertex, the connected neighbor vertices are vectorized. Since the concept is to find the missing links, the use of the neighborhood is not required, as the vertices do not have a link with each other, which is a difference compared to NoiseRemoval. Then, with the vectorized neighborhood, hashes can be generated for each vertex equally, as explained in Chapter 5. Several hashes are generated, and these hashes are divided across the hash tables. Different vertices with the same table of hashes are connected with an edge, which is the missing link that is found. There are two parameters to adjust: the numHashesPerTable and the numHashTables, as discussed in Section 5.4. The former makes the found MissingLinks more likely to be similar, while the latter will increase the number of links that could be found per vertex. Both parameters will increase the required number of hashes generated by the LSH. The number of hashes required is the numHashesPerTable times numHashTables. The benefit of using MinHash to find missing links is that the problem becomes tractable, and it can be calculated how long it will take to generate the required hashes for all the vertices.

7.3.1. MissingLinks Benchmark

There are two scopes of implementing MissingLinks; the first is the InverseID scope, which is equal to the graphs used in the matching phase of the ER pipeline. The second scope is on EntityID scope. Since NoiseRemoval needs to be applied before the MissingLinks, it is possible to use the generated EntityMapper to transform the entire InverseID graph to an EntityID map simply by mapping these InverseIDs to their corresponding EntityIDs. Both scopes are applied on the best-performing NoiseRemoval setting, as seen in Table 7.2. The numHashesPerTable and the numHashTables are set to 20 and 10, respectively, meaning that for each InverseID or EntityID (dependent on the scope), 200 hashes will be generated. If at least a single hash table is equal to that of another InverseID or EntityID, then these different InverseIDs or EntityIDs are

considered to refer to the same real-world company, meaning that they can be merged. The result of this experiment can be found in Table 7.3.

Run	MissingLinks	Recall	Precision	F_1-Score	EntityIDs	Size
RunBaseline	×	0.70742	0.89754	**0.79122**	27,205,371	**32.16%**
RunBaseline	InverseID	0.70742	0.89754	0.79122	27,205,371	32.16%
RunBaseline	EntityID	0.70742	0.89754	0.79122	27,205,371	32.16%
RunLSH	×	0.70715	0.89451	0.78987	50,806,031	60.06%
RunLSH	InverseID	0.71538	0.89301	**0.79439**	41,040,759	48.51%
RunLSH	EntityID	0.71510	0.89343	0.79438	38,813,407	**45.88%**
RunAugBaseline	×	0.80875	0.81251	0.81063	46,742,748	55.25%
RunAugBaseline	InverseID	0.81112	0.81232	0.81172	37,282,869	44.07%
RunAugBaseline	EntityID	0.81097	0.81259	**0.81178**	33,029,113	**39.04%**
RunAugLSH	×	0.77731	0.81855	0.79740	44,295,941	52.36%
RunAugLSH	InverseID	0.78051	0.81838	0.79900	34,082,778	40.29%
RunAugLSH	EntityID	0.78042	0.81855	**0.79903**	31,289,149	**36.99%**

Table 7.3: Measurements of MissingLinks per Scope for the Benchmark Runs

In the column 'MissingLinks', the applied scope is noted, with × marking that no MissingLinks was applied to that run, which translates to the best NoiseRemoval setting of Table 7.2. One thing that is easily noticed is that applying NoiseRemoval on the RunBaseline will not have an effect. This phenomenon can be explained by the fact that the run is non-overlapping. In this run, each created block in the blocking phase will have its own unique EntityID, so no missing links can be found between different blocks, as there are no edges connecting the different blocks. The general trend that can be seen is that both scopes increase the F_1-scores and the deduplication factor, as can be seen in a decrease in size. Applying MissingLinks on both scopes will be roughly equal in terms of recall and precision, with the InverseID scope performing marginally better on recall while the EntityID scope performs marginally better in terms of precision. The main difference can be seen in size; EntityID will create fewer unique Entities compared to the InverseID scope. Fewer Entities generated with still a comparable F_1-score compared to InverseID means that the EntityID scope performs better. It can merge more Entities and still achieve an F_1-score as high as the InverseID scope. Therefore, the EntityID scope is used for further application.

Instead of applying MissingLinks with several hash tables, performing it with just a single hash table and applying multiple merge passes is also possible. Theoretically, this should cost fewer resources and allow one to find the local maximum without specifying the total number of hashes. However, since using LSH only requires limited resources, this experiment was not conducted, as MissingLinks already shows significant results in reducing the size of the generated EntityMapper.

7.4. Benchmark Runs

The best-performing EntityMapper per ER run is selected. These EntityMappers are listed in Table 7.4. These final mappers are created by applying NoiseRemoval with their given ApproxSim_{40} as the NoiseRemoval threshold. Afterward, MissingLinks is applied on the EntityID scope, which is the EntityMapper created by

Run	ApproxSim_{40}	NoiseRemoval	Recall	Precision	F_1-Score	EntityIDs	Size
RunBaseline	0.725	EntityID	0.70742	0.89754	0.79122	27,205,371	32.16%
RunLSH	0.875	EntityID	0.71510	0.89343	0.79438	38,813,407	45.88%
RunAugBaseline	0.675	EntityID	0.81097	0.81259	0.81178	33,029,113	39.04%
RunAugLSH	0.725	EntityID	0.78042	0.81855	0.79903	31,289,149	36.99%

Table 7.4: Measurements of the EntityMappers for the Benchmark Runs

the NoiseRemoval. In the runs that do not rely on augmentation, it can be seen that the RunLSH has a higher F_1-score than RunBaseline but a lower reduction in size. For the runs relying on augmentation, the RunAug-Baseline has a higher F_1-score, but a lower size reduction than RunAugLSH. Possible solutions to increase the results of the RunAugLSH are discussed in Chapter 10. The final noticeable result is that the augmentation has a relatively high recall but a relatively low precision.

8

Applied

In this chapter, we apply the ER pipeline to Late Payment Prediction (LPP), an existing model within Exact, and the results of this application are shown. First, in Section 8.1, we briefly introduce LPP combined with the problems applying ER could theoretically solve. Afterward, the created EntityMappers of the ER pipeline are applied to train LPP models, and we discuss their findings in Section 8.2.

8.1. Late Payment Prediction

To see the improvement in using the deduplicated Accounts data set over the original version, the results of the ER are used in the LPP module. This module predicts if the given Account will pay their next invoice on time. On the lowest level, the module looks at the Account data attached to the invoice. By using the deduplicated Accounts data set, it is possible to overcome some issues in comparison to the duplicated Accounts data set:

- **Cold start:** If the user just created the Account, then there is no historical data attached to this Account, making it difficult to predict if they will pay their invoice on time. However, suppose other Accounts that refer to the same real-world company are attached to the same Entity. In that case, it is possible to fill in the missing data with historical data of those Accounts.
- **Latest Information:** LPP looks at the 2000 latest invoices to determine if the Account will pay their invoice on time. These are the latest invoices attached to the Account but not necessarily the real-world company's latest ones in the data lake. If the Entity is used instead of the Account, it will utilize the real-world company's latest known information in the database.
- **Smaller size:** There are fewer unique Entities than unique Accounts. Therefore, the training problem will become more manageable, allowing more expensive operations to be applied to classify the problem more efficiently and effectively.

8.2. Benchmark Runs

The best-performing EntityMapper per ER run, as shown in Section 7.4, is used to train the LPP classification. The training sets are balanced. The provided labels are uniform for the classification; there is a 50/50 split between late invoices and those paid before expiration. Therefore, because of this uniformity, it is possible to measure the accuracy of the classifier. A classification model is trained on the entire data set on which no ER is applied. Models are also trained for the different EntityMappers runs. Mao's best LPP model settings are used to create the models are used to create the models [28]. The results of these different runs are shown in Table 8.1, which shows that applying ER has an adverse effect on the results of LPP. Applying no ER has the highest accuracy, recall, and precision. It is also noteworthy that the measurements are not significantly worse when ER is applied, even though the size of these data sets with ER applied is significantly smaller than the original data set. These results might indicate something inherently wrong with the EntityMapper used.

A few things can be noticed when taking the Accounts data set under the loop. The ER pipeline is implemented to find rows referring to the same real-world companies. However, not all the Account rows refer to a real-world company. In the Accounts data set, there are also end-users, i.e., persons, not companies.

EntityMapper	Data Scope	Accuracy	Recall	Precision	F_1-Score
No ER	Full	0.78344	0.62424	0.54835	0.58384
RunBaseline	Full	0.77402	0.59343	0.51216	0.54980
RunLSH	Full	0.77169	0.58832	0.50824	0.54536
RunAugBaseline	Full	0.77165	0.58824	0.50815	0.54527
RunAugLSH	Full	0.77014	0.58500	0.50544	0.54232

Table 8.1: Results of using the created EntityMappers of the Benchmark Runs on LPP

Most, if any, of these personal Account rows will not have a CoC, as having such a number is limited to companies. This results in personal Account rows not being part of the GT, as no label can be attached to these rows. This means that the ER pipeline never considers the existence of persons in the data set, only companies. These personal rows cause the application of ER to have undesired effects. After investigation, the ER pipeline will cluster rows with the same forename or surname together as if they were sub-divisions of a real-world company. Generating clusters of people who have parts of their name in common will give an undesired side-effect. The contact information of an employee of a company can be filled in for an Account row instead of the company's contact information. These personal fields are fields like Name, Email, and Phone. These persons should never be a Positive-Match with a company or vice versa. Even if the person works at said company, they are and should not be equal. It is possible for specific company clusters to incorrectly merge with these personal clusters because of matches within an employee-associated Account row.

The ER pipeline should perform differently for companies and persons. Ideally, the data set needs to be split into a data set of company and personal rows before the entire ER pipeline starts. Splitting the data set into these two counterparts makes the initial data set smaller and easier to process. Therefore, due to the clustering of end-user surnames, implementing the outcome of the ER pipeline directly into existing modules such as LPP might not improve the module's outcome. LPP Runs are performed on a data subset of Account rows associated with a real-world company. These additional runs are performed to see if the outcome increases if these personal Account rows are discarded.

8.2.1. Company Subset Benchmark

For the second benchmark experiment, a subset of AccountIDs that should refer to a company is selected instead of using the entire data set. For the preprocessing, as discussed in Chapter 4, the business structures were removed to clean up the Names. However, these business structures will now be used to create a set of Accounts that will likely represent a real-world company. A small subset of large Dutch companies is also added to filter upon, shown in Appendix A.5. The complete Accounts data set is filtered on rows that contain at least a single word of the filter list, creating the Company subset. This set of CompanyAccounts consisted of approximately 6 out of the 85 million Accounts in them, making the data set substantially smaller. The same LPP settings are used to create the runs performed on this Company data subset. The CompanyAccounts also has a uniform training and testing set, and the results are shown in Table 8.2.

EntityMapper	Data Scope	Accuracy	Recall	Precision	F_1-Score
No ER	Company	0.74788	0.66039	0.62090	0.64003
RunBaseline	Company	0.72740	0.63090	0.58999	0.60976
RunLSH	Company	0.72800	0.63190	0.59026	0.61037
RunAugBaseline	Company	0.72778	0.63120	0.59133	0.61061
RunAugLSH	Company	0.72517	0.62787	0.58581	0.60611

Table 8.2: Results of using the created EntityMappers of the Benchmark Runs with a Company Scope on LPP

The performed run has the same pattern as Table 8.1, which can be seen in Table 8.2. All the ER runs perform worse for the given model. There are a few different views on this phenomenon. The LPP model has features that Mao optimized for the dirty Accounts data set [28]. Using an entirely different input data set requires different parameters and features. However, implementing changes to these parameters and features will be

outside the scope of this feasibility study on the ER pipeline.

During the ER process, the main idea of the pipeline is to create entities that refer to the same real-world company. Causing, for example, different sub-divisions or physical stores of a real-world company to be mapped to the same given EntityID. However, this merged notion of entities is not always the correct form for all the models. There is no clear notion of what a company is and should represent within the Accounts data set. Some models require these entities to be split into different sub-divisions or physical stores. The same can be argued for LPP. If the invoice payment is performed not on a company level but per sub-division or physical store, then using this entity as a whole will make no sense; they should be split up into different compartments. Further methods that split the created entities into their division component should be investigated so they can be applied to different models.

Even though the results of applying the different EntityMappers do not improve the performance of LPP, they could still be used on the cold start cases. Account rows that previously had no invoices attached to them could not be used in LPP. However, invoices can now be found and used by merging these new Account rows with the more established rows. While still providing the measurements as seen in Tables 8.1 and 8.2.

9

Conclusion

In this chapter, we recap the findings discussed in each section of the thesis, which will help address the main Research Question (RQ). In order to address the main RQ, each of the Sub-Questions (SQs) will first be reviewed and addressed. These SQs can be found in Section 1.1. The thesis is performed on the data set offered by Exact but not solely created for Exact. The implemented methods are generalized and can be applied to other data sets with the same characteristics. These characteristics are big data sets with duplications, i.e., multiple rows refer to the same real-world object, and where the rows are provided by users, making them incoherent and incomplete. A knowledge gap exists in implementing D-ER pipelines for data sets following the characteristics above. The characteristics of the data set are investigated in a D-ER to make the findings applicable to other data sets. D-ER means that a single data set with duplicates is used for the ER. This thesis aims to determine if applying an ER pipeline starring LSH to the given data set is feasible. The desired outcome of such an ER pipeline is to create an EntityMapper that maps IDs to EntityIDs. These formed EntityIDs create new ventures of gathering statistics and implementing additional models.

The applied ER pipeline consists of four phases: preprocessing, blocking, matching, and clustering. An elaborated preprocessing phase is implemented to make the data as coherent as possible. In the blocking phase, blocks are created to reduce the required comparisons. In the matching phase, these blocks are cleaned by removing all the Negative-Matches. Finally, in the clustering phase, novel implementations of the LSH MinHash method are applied to remove noise and find missing edges to create the desired EntityMapper. Except for the clustering phase, the runs relying on LSH outperformed the other runs. This shows there is merit in applying LSH to the ER pipeline.

The main RQ is: "*Can Locality-Sensitive Hashing techniques help to improve the Dirty Entity Resolution efficiency and accuracy for incoherent and incomplete distributed big data scenarios?*" In order to address this question, multiple SQs need to be investigated and implemented. An investigation of applying LSH to the ER pipeline is required. Therefore, multiple distinctive runs are investigated throughout the different stages of the ER pipeline to address this question. Each run starts the same; the incoherent user-provided raw data is made coherent in the preprocessing phase. Rows that have, after preprocessing, exactly the same filled-in properties are merged. These cleaned fields can be used to predict if a Match is a Positive-Match.

Deduplication of small data sets could be possible with a cross-join operation; however, in the case of big data, this cross-join operation is infeasible and intractable as this operation grows quadratically in size. Almost all the Matches created by a cross-join operation are False-Matches, wasting significant computation power. Instead, the big data set should be split into smaller blocks with a high likelihood of being True-Matches, and only cross-join operations should be done in these smaller blocks. The blocking phase is responsible for creating these blocks within the data set. This significant phase makes the problem tractable, bringing us to SQ1: "*Can the blocking phase of Dirty Entity Resolution significantly reduce the number of required comparisons associated with big data, making it tractable, while still producing high-quality blocks?*"

SQ1's primary focus is on the total number of comparisons after blocking, as ER can not be feasibly applied without a high reduction of comparisons. Four distinct run configurations are introduced in the blocking phase and used throughout the ER pipeline: RunBaseline, RunLSH, RunAugBaseline, and RunAugLSH. These four runs rely on the same preprocessing, making it possible to compare them. RunBaseline are blocks

created with Standard Blocking on the always available property for each data set row, the Name row. Secondly, the RunLSH is created; in this run, the LSH method MinHash is used on the always available property to block the data. MinHash has the advantage of correcting spelling mistakes and finding matches with similar patterns within the property. In order to give a fair comparison, these two runs are joined with an additional blocking scheme, which is an augmentation and relies on the incomplete data of the data set. Incomplete data cannot be used as the base for blocking the data set, as that would mean rows will be left out of the ER pipeline. However, these incomplete data sets can augment the general blocking scheme. Multiple augmentative blocking schemes were investigated, and the best-performing augmentations were combined and then joined to RunBaseline and RunLSH to create RunAugBaseline and RunAugLSH, respectively.

An interesting outcome of the blocking phase is that applying blocking with the most simple RunBaseline would take 1.229.959 times fewer comparisons than the exhaustive cross-join. This run has a good PC and an excellent PQ. The most complex configuration, the RunAugLSH, still divided the total required comparisons by 82.709 times. The PC was better in this run, but the PQ was worse than RunBaseline. This result can be reasoned due to LSH applying error correction, which relies on creating more (required) comparisons. Error correction results from splitting the words into N-grams and creating blocks based on similar sets of tokens. The experiments also showed that applying augmentations significantly improves the PC at the cost of PQ. However, PQ will be improved in the next phase, making the outcome favorable.

The aforementioned allows us to address SQ1. It is possible to implement a blocking phase that can block similar rows together, reducing the total number of required comparisons by a significant number of times. Not only is this possible, but the results also show that this can be done with a high PC and PQ.

The second sub-question, SQ2, is as follows: "*Can the matching phase of Dirty Entity Resolution correctly classify comparisons between incoherent and incomplete rows?*". Due to the incomplete nature of the properties within the data set, classification is not straightforward. Only one property is always provided, but classification on only that property will not give the desired results. Including more properties in a match will generally result in a higher confidence in the results by a classifier. Multiple classifiers were trained to overcome the issue of training a classifier on incomplete data; each classifier corresponds to its unique subset of available properties. Splitting the comparisons into disjoint subsets and their uniquely trained classifiers has the extra benefit of a divide-and-conquer mentality while also giving more robust results. These different subsets can all be classified in parallel for big data, improving the required time for large sets; however, this comes at the cost of additional overhead computations.

The experiments showed that the recall will slightly decrease after applying the classifiers, but the precision significantly increases. With the aforementioned implementation and their results, creating a matching phase on a data set created with (user-provided) incoherent and incomplete rows of data is possible. These classifiers will improve the PQ while only slightly affecting the PC.

The next sub-question, SQ3, is the last question regarding the inner workings of the applied ER pipeline and goes as follows: "*Can the clustering phase of Dirty Entity Resolution be applied to cluster distributed big data with high correctness?*" The output of the matching phase is a graph, and the output of the ER pipeline should be an EntityMapper, which maps a unique ID to an EntityID. In order to create the mapping, clusters within the graph must be found and given an EntityID. One way to implement this is to split the graph into their disjoint sub-graphs. Using these sub-graphs as unique Entities of rows is an efficient method of creating such a mapper. However, disjoint sub-graphs are particularly susceptible to noise within the graph. The existence of bridge edges linking otherwise disjoint sub-graphs together will invalidly merge sub-graphs, which erroneously resulted in the creation of one large cluster that almost encapsulated all the different rows. In the matching phase, two rows are only compared by their available properties, but in the clustering phase, it is possible to compare rows with their spatial information in the graph. The edge is valid if the two rows share the majority of neighbors in the graph. If the two rows of an edge consist mainly of distinct neighbors, the edge is discarded. In this method, coined NoiseRemoval, the fraction of neighbors in common is found with the novel use of LSH MinHash, in which an edge is removed if the fraction found is below the specified threshold. NoiseRemoval was shown to be effective in removing noise, as separate clusters could now be generated that have good recall and high precision.

After NoiseRemoval, missing edges can be found. These missing edges were missed due to not being blocked together or marked as Negative-Match by the classification. A novel clustering technique, MissingLinks, is introduced to find these missing links. MissingLinks uses a technique similar to NoiseRemoval, but the technique is slightly adjusted to accommodate finding the missing links. LSH MinHash is applied on

the neighbors of each row, and then hashes are generated for each row. If the hashes overlap between two rows, they share neighbors; an edge will be created between them if sufficient hashes overlap. It showed that MissingLinks marginally affected the recall and precision while significantly reducing the number of Entities created. Merging more Entities and still having the same good measurements is an excellent outcome.

With the aforementioned, the SQ3 can be addressed. The recall and precision are marginally lower after applying the clustering phase. However, transforming an interconnected graph into clusters of disjoint Entities will always come with a natural loss. A marginal loss is an excellent result; thus, clustering can be applied to D-ER while maintaining the correctness of the matching outcome.

The final Sub-Question, SQ4, applies the created EntityMapper to an existing model and goes as follows: "*Can the created deduplicated data set be used as a solution in existing models to improve their results?*" The different EntityMappers were used to train a LPP model. However, the outcome was worse when applying the ER pipelines, which can be caused by the inconsistency of what the data represents within the data set and, more importantly, the notion of what the Entities should represent. The ER pipeline merges all companies' sub-divisions into a single Entity. However, each sub-division may handle its invoices differently than one another in the company. Therefore, for the case of LPP, an additional step of splitting the created Entities into their sub-divisions should be implemented. Even with a worse result, it is now possible to give a prediction for newly created rows that have no historical information attached to them.

The address to SQ4 is inconclusive. For the current model, it does not necessarily improve the outcome, although, with the use of ER, it is now possible to predict new Account rows without historical data. However, the average result is worse than not applying ER. The created EntityMappers are not a ready solution for the LPP model to improve its outcome, but that might be different for other models, as other models might require a different notion of what an Entity resembles.

The main RQ can be addressed with all the conclusions drawn on the SQs. In the implemented D-ER pipeline, LSH MinHash is used in the blocking stage to create blocks of similarly spelled rows. Although using LSH does require more comparisons, it will improve the PC at the cost of PQ. Applying an efficient matching phase on the incomplete big data is possible. In the clustering phase, LSH enables, efficiently and effectively, the removal of noise within the Positive-Matches based on spatial information. Links that are missed can also be found for distributed big data scenarios. The implementation of MinHash allows these two clustering methods to be applied. Without using LSH, the comparison would require a large space requirement. For the final created EntityMappers by the four different runs, the recall will improve faster than precision decreases when applying LSH. A significant and cheap improvement was the addition of augmentations. Finally, the applied scenario showed that it performs slightly worse when applying the created Entities, but by using these Entities it is possible to classify rows without historical data. A slightly different Entity notion could increase the outcome for the specific scenario, which requires an additional step. For other scenarios, however, merging sub-divisions from Entities into a single Entity will likely be correct.

Using LSH MinHash as a technique in an ER pipeline will improve the efficiency and accuracy of the resolved Entities within incoherent and incomplete big data of Exact's customers. LSH helps to find matches between incoherent rows that refer to the same real-world object. However, the traditional ER pipeline must be altered to fully utilize such an approach's strength. First, a more straightforward pipeline should be applied, and afterward, a more extensive error-corrective pipeline relying on LSH should be implemented.

10

Discussion & Future Work

In this final chapter, we discuss the limitations of this research work. We will also present an overview of the future works. The thesis is a feasibility study; after conducting such a study, it opens new ventures within the data that can be explored. The directions for future work will be elaborately provided. We start by explaining the limitations of using PySpark in Section 10.1. Afterward, we discuss the labels and other possible ways to measure effectiveness in Section 10.2. Next, in Section 10.3, we introduce a new double-pass pipeline that combines the strengths of the investigated implementations. The created Entities can be transformed into multiple sub-Entities, as discussed in Section 10.4. This section is followed by possible future work on applying the ER pipeline on different timesteps and investigating how the rows change over time to gather more information on the data, as explained in Section 10.5. We discussed in Section 8.2 that the Accounts data set has imperfections; we dive into them and their possible solutions in Section 10.6. In Section 5.4.3, we discussed the investigated manipulations, but future research is warranted, as they can be very effective. These hash manipulations are briefly described in Section 10.7. The Matching phase relies on multiple classifiers, as shown in Section 6.4. However, data sets with many varying properties could result in too many required classifiers. Additional setups should be investigated, as discussed in Section 10.8. In Section 10.9, we discuss additional use cases for applying the unique words, as explained in Section 5.3.1. Lastly, we present a novel addition to MultiGram in Section 10.10, which could improve MultiGram's results when implemented.

10.1. PySpark Limitations

The big data was handled in a distributed manner with PySpark in Python. PySpark is selected by Exact as the environment for working with big data. The environment enables the processing of big data in a distributed manner but has some inherent limitations. PySpark is being mapped to the Apache Spark engine in Scala. Due to the difference in environments, using UDFs in PySpark is inherently slow. It is so slow that processing a simple UDF for a thousand rows already adds so much overhead that computation is not feasible. The added overhead makes it impossible to use UDFs to create highly sophisticated functions that can be used throughout the ER pipeline. It is only possible to use complex methods from PySpark or create methods that do not require the inherently slow UDFs. The only problem of not being able to use these complex function constructors results in the methods being rather rudimentary by being bound to what is possible within the PySpark environment itself. This is a significant limitation and disallows complex methods such as Jaro-Winkler. The addition of methods written in the Scala Apache Spark environment and imported within the PySpark environment should be investigated. Having more methods at one's disposal could result in better results.

Another problem with using PySpark is that an undocumented limit was reached in the size of the created models. If the CountVectorizer[1], which creates the vocabulary for the LSH MinHash implementation, receives a VocabSize larger than five million, the execution would be terminated. Because of this issue, using the entire wanted vocabulary as the knowledge base of other methods is only sometimes possible. Due to this limitation, additional decisions must be made on which settings will generate the best executable model. This means the current implementation and results shown for all methods based on LSH could be improved

[1] https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.ml.feature.CountVectorizer.html

if this inherent limitation is gone. Investigation and removing this inherent limitation will, therefore, directly improve the results.

10.2. Label Generation

The project is based on big data. By definition, the data set will be too big to be labeled correctly by hand; therefore, the labels are generated from the information provided by the users. However, these labels are not always correct and can be split into two distinct groups:

- **True-Matching:** If the label is a match, then both the VATNumber and CoC are precisely the same value. It is improbable that, by chance, these overlap; therefore, the matching labels are almost entirely correct.
- **False-Matching:** The labels do not match by either their VATNumber, CoC, or both. A single uncorrected spelling mistake within one of these four fields (two per row) will already result in the incorrect label for the comparison.

With this clear division and the inclusion of spelling mistakes by the user, it is possible to claim that numerous comparisons classified as FP should be TP. The number of TPs will be greater, while the number of FPs will decrease. Since fewer Negative-Matches exist, the True Negative (TN) will be decreased. While the number of False Negative (FN) will marginally increase, allowing for spelling mistakes, it means that both the VATNumber and CoC of the companies need to be almost identical to be considered a spelling mistake, which is extremely unlikely.

$$recall = \frac{TP}{TP + FN} \tag{10.1}$$

$$precision = \frac{TP}{TP + FP} \tag{10.2}$$

Applying this claim, the recall, as shown in Eq. (10.1), would be increased due to TP and an almost unaffected FN. This can be explained by the fact that incorrectly labeled rows are now found. The precision, expressed in Eq. (10.2), would be increased more significantly than the recall since TP increases and FP decreases. This adjustment in label generation can find the incorrectly labeled rows that should have been a match. However, applying a different implementation for label generation that encapsulates spelling mistakes will require more resources.

10.2.1. Entity Notion

The classifier deems some Matches as a correct Match and are a Positive-Match, but the label marked it as a False-Match, making the Positive-Match a FP. However, after investigating the FPs, there are explainable cases in which a Matches should be viewed as TP. These are the following cases are:

1. The rows match in all their fields except their physical address; this company could be a franchise with multiple addresses. Each physical store has its own VATNumber.
2. The company has a single headquarters, which is used as their contact information. The Accounts can be equal in all their properties, but there is a structure hidden in this company, in which each subdivision has its own CoC or VATNumber.
3. Outdated information: even though the Accounts are filtered on having at least a single transaction within the last two years, it is possible that the account still uses an outdated VATNumber or CoC.

The first two cases boil down to what the created Entity should represent: do all the sub-divisions need to be merged into a single Entity or not? This notion of an Entity also changes depending on the company; some companies use different CoCs or VATNumbers to address their sub-division, while others only use a single CoC or VATNumber for all their departments. In the last case, outdated information resolves in creating an FP. In the future, the changes in the row's CoC and VATNumber can be tracked to see how these change over time. Instead of only looking at the current CoC or VATNumber, the historical versions can be used to check if these rows are a True-Match.

10.2.2. Label Generation Scopes

There are some disadvantages to using the labels based on the VATNumber and CoC. Companies with multiple sub-divisions or physical stores sometimes have different VATNumbers or CoCs for each department. So even if Accounts are part of the same unique real-world company, they can have different labels attached.

The question that comes forward is what a real-world company is. Is a franchise a single company or multiple smaller companies? In Exact's case, the data set will make the distinction with the information the user has provided. The trained classifiers find more TPs if these sub-divisions are considered Positive-Matches with each other. That means that the classifiers think an entire franchise is a single company. This notion of an Entity is neither right nor wrong; it depends on the representation that further applications require.

The other problem with using the labeled data for classification is that the users provide the data, meaning spelling errors can be made while typing. In the case of Exact, different settings can be used on the labels to determine whether the Match is a False or a True-Match.

- **Strict**: When using strict labels to create the metrics, both the VATNumber and CoC must be equal, ensuring a high probability that these two Accounts refer to the same real-world company.
- **Lenient**: When using lenient labels, only one VATNumber or CoC must be equal to be classified as a True-Match.
- **Error-Corrected**(n): It is also possible to correct the user error with edit-distance if either the VATNumber or CoC have a n edit-distance from each other, then it is still classified as a True-Match.

The uses of these different ways of handling the labels serve their own purpose. Classifiers should be trained on a data set that is labeled correct, meaning the strict case is the best to train the data on. Error-correctness can be used for both the strict and lenient options. However, when applied to the lenient case, many correctly classified False-Matches will be transformed into True-Matches, which is erroneous. This change also means that the False-Matches are more likely to be correctly labeled False-Matches. Creating strict and lenient labels can be done with simple grouping. Applying error correction, however, means that all the labels need to be cross-compared, which still requires at least 29 trillion comparisons, as seen in Table 3.2. Another novel implementation is to use LSH again to create buckets of highly similar labels and only compare those for possible spelling mistakes. Although the result is an approximation, every improvement on the created labels can be used to produce more correct classifiers.

10.3. Double-Pass Pipeline

Throughout the pipeline, the basic augmented run, RunAugBaseline, performs well with a relatively small number of required comparisons and has the highest performance, as seen in Table 7.4. The intermediate steps show that the error correction application of LSH blocks increases the PC. Combining these two ER runs into a single combined ER double-pass run is possible and allows the combination of the strengths of both RunAugBaseline and RunLSH. After the first pass, another ER pass is possible since the output of the ER process is equal to what the ER process requires as its input. The only difference is that the IDs are mapped to a new EntityID. So, another ER pipeline can be immediately applied in a double-pass fashion afterward.

First, all the Accounts are merged with more straightforward methods, similar to methods used in RunAugBaseline. Afterward, a more extensive pipeline can be applied since many rows are merged, reducing the input size. This pipeline can link Entities created by the first pass while taking error corrective measures. The applied LSH second pass can essentially be seen as a way to cluster the created disjoint Entities EntityMapper of the first pass. Instead of applying error correction directly, it is a way to create new Matches. These newly created Matches are created to overcome the problem of differently spelled rows not being placed in a joint block within the blocking phase of the first pass. This implementation enables error correction while consuming fewer resources than the RunLSH requires. This double-pass solution can be applied to all existing ER pipelines for link discovery and should result in a higher PC with the possibility of lower PQ.

10.4. Cluster Splitting

As stated in Chapter 8, further research should be conducted on splitting the final created clusters into their components. This cluster splitting could be applied for this cause and filter out rows not belonging to the Entity, creating more precise Entities. One idea for implementing the splitting of clusters is applying MinHash.

First, all the properties in the row can be merged into a single string, and that string can be used to create hashes with MinHash. The fraction of overlapping hashes can be used as a weight of the edges between different rows. Afterward, the rows can be split into sub-components by these weighted edges using a clustering method. Transforming the Entities into sub-components could improve the results of existing models. The PQ will be improved while the PC could slightly decrease.

10.5. Adaptive Entity Resolution

Currently, the data set of a specific point in time is used to apply the runs. Nevertheless, it is possible to rerun the ER pipeline on different time frames; seeing how companies in the Accounts data set change over time can give rise to an entirely new world of data within this data set. By doing so, the delta Accounts, which are Accounts that are new or have a field updated, can be placed into additional blocks, finding more connections with the other rows in Accounts. More information about a company itself can be discovered. Creating a new type of data within the provided data set, i.e., companies acquiring new locations, rebranding with a name change, or the merging of companies. Currently, if a company merges with another, it can cause a weird effect within ER because afterward, there might be three different companies, the original two and the merged one. Losing information about the company but in the sense of LPP, merging these two disjoint companies with different payment styles will result in incorrect behavior.

The current implementation allows for multiple adaptive runs by ensuring the algorithms that use any sense of randomness are always run with the same seed, removing the randomness. The delta Accounts can be rerun through the pipeline. If done per interval, the number of delta Accounts is small, making it possible to keep applying ER on a specific interval, i.e., weekly, without requiring significant resources. The same blocks are mainly created when using adaptive ER, so the PQ will be mostly the same. The PC will be improved as more matches can be found using historical data. RR will be increased per time step, but since the ER pipeline will now only work with delta Accounts, there are fewer required comparisons per time frame.

10.6. Pre-Splitting

The active rows of the Accounts data set are used for the applied ER pipelines. However, this data set contains different types of what rows represent within the data set, as seen in Section 8.2. In the current implementation, the idea is only to create labels for the rows of interest; this will automatically give the unwanted rows no impact in the classification phase. However, since these rows still go through the entire ER pipeline, they can introduce noise within the generated Matches. In the blocking phase, Chapter 5, different blocks are created on family names, for example. These poorly created blocks are not expressed in terms of PC and PQ, as discussed in Section 8.2.

The Accounts data set should be split into personal and company rows, which is a unique case for the given data set used, but other data sets could also face a similar problem. Future ER pipelines should start with an in-depth investigation of the GT. Detecting which rows have a label attached and, more importantly, which rows do not have a label attached. If a subset of similar rows lacks labels, it should be looked into what will happen if that subset is removed from the data set before applying the ER pipeline. So, these rows will not be mapped to an Entity, as they are not included in the ER pipeline.

Another problem with the data set and what could be split into separate problems is that the entire ER pipeline uses data cleaned on the Dutch formats, i.e., Postcode. Different countries have different schemes for their Postcodes, invalidating other countries' Postcodes by default. It might be as simple as splitting the data set beforehand and applying multiple ER pipelines on the disjoint data sets; each split data set should have its own implementation.

10.7. Hash Manipulation

After applying the LSH algorithm, an array of hashes is generated that can be manipulated. Some ideas are tried and shared in Section 5.4.3. However, these or new ideas should be investigated further to manipulate the hashes into creating many different blocks without much computational cost. A solid performing hash manipulation can increase the blocking algorithm's efficiency and effectiveness for a little computational cost. It could thus serve as a significant improvement factor in creating many blocks.

10.8. Many Feature Classifiers

If many features rely on not always available properties, then the currently implemented classification scheme will become too large as 2^n classifiers are required to classify the problem. The exponential nature will make a large n require too many classifiers. Instead of classifiers with all the possible available properties, it can also be possible to train classifiers of size m, in which ($m > n$), with subsets of size n, essentially losing out on some (less significant) property values while still being able to classify the problem more precisely than only using the always available properties.

10.9. Unique Words Concatenation MinHash

Only the Name property is currently used for LSH MinHash in the blocking phase. Adding all the properties into a single document, separated by whitespace, might be feasible. Afterward, the common words can be retrieved from this single line, as discussed in Section 5.3.1. The common words are now known, which can be removed before applying the MinHash, making it possible to create longer strings with uniquely identifiable words for real-world entities, which can serve as the basis for applying MinHash in the blocking phase. This new blocking technique circumvents the problem of the incomplete dataset by appending all the rows to a single line, but being doing so, information is lost about what the words actually represent. If only the words that address unique cases for a real-world entity are kept, then the merged information can be used in an LSH MinHash blocking scheme. This blocking scheme should have a high PQ and a high RR. However, it depends on how much the words are shared between different entities; if rows do not contain any unique words, these rows cannot be used in such a blocking technique, impacting the PC. Such a blocking technique can also be applied as an augmentation scheme, improving the overall PQ and PC of the applied blocking phase.

10.10. Skip-MultiGrams

If N-gram is applied to two words where one has a single erroneous character, then at most N tokens will be different between the two. The size of N largely influences the Jaccard similarity used within error correction. A larger N will give a lower Jaccard similarity. A larger N will contain more information on the structure of the word; thus, a larger N does help decrease the number of FPs. Nevertheless, it also comes with the problem of making the tokens more dissimilar per spelling error between the rows.

Therefore, the following novel concept coined the Skip-MultiGrams, can be introduced to negate this problem while still containing more information about the structure of the words. Skip-MultiGrams is an alteration to the MultiGrams discussed in Section 5.4.1. The way the N-grams are created is different in this alteration. MinN should be set to two, making all the tokens contain at least two characters. For each token created, only keep the two outer characters and replace all the inner characters with a skip character, such as the underscore character. In essence, tokens will always contain just two characters (excluding the skip character). If the alphabet consists of a characters, then for each N, the maximum number of unique tokens would be a^2. This results in the number of tokens created with SkipGrams being independent of N, solving the overshadowing caused by a larger N within MultiGram. In MultiGram, a spelling mistake would make the generated tokens at most N tokens different per N, but with the implementation of Skip-MultiGrams, a spelling mistake would at most only make 2 tokens different per N. This alteration can overcome the inherent problems with applying MultiGrams for a MinHash implementation and should be investigated. This updated version should find more connections of better quality between rows. If used as the base for generating MinHash, it could improve the PC, PQ, and RR.

Bibliography

[1] Divyakant Agrawal, Philip Bernstein, Elisa Bertino, Susan Davidson, Umeshwas Dayal, Michael Franklin, Johannes Gehrke, Laura Haas, Alon Halevy, Jiawei Han, et al. Challenges and opportunities with big data 2011-1. 2011.

[2] Otmane Azeroual, Meena Jha, Anastasija Nikiforova, Kewei Sha, Mohammad Alsmirat, and Sanjay Jha. A record linkage-based data deduplication framework with datacleaner extension. *Multimodal Technologies and Interaction*, 6(4):27, 2022.

[3] Mikhail Bilenko, Raymond Mooney, William Cohen, Pradeep Ravikumar, and Stephen Fienberg. Adaptive name matching in information integration. *IEEE Intelligent Systems*, 18(5):16–23, 2003.

[4] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. Enriching word vectors with subword information. *Transactions of the Association for Computational Linguistics*, 5:135–146, 2017. ISSN 2307-387X.

[5] Surajit Chaudhuri, Kris Ganjam, Venkatesh Ganti, and Rajeev Motwani. Robust and efficient fuzzy match for online data cleaning. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 313–324, 2003.

[6] Peter Christen. A survey of indexing techniques for scalable record linkage and deduplication. *IEEE transactions on knowledge and data engineering*, 24(9):1537–1555, 2011.

[7] Peter Christen et al. Towards parameter-free blocking for scalable record linkage. 2007.

[8] Vassilis Christophides, Vasilis Efthymiou, Themis Palpanas, George Papadakis, and Kostas Stefanidis. End-to-end entity resolution for big data: A survey. *arXiv preprint arXiv:1905.06397*, 2019.

[9] Aaron Clauset, Mark EJ Newman, and Cristopher Moore. Finding community structure in very large networks. *Physical review E*, 70(6):066111, 2004.

[10] Mary Jordan Coe. Mechanization of library procedures in the medium-sized medical library: X. uniqueness of compression codes for bibliographic retrieval. *Bulletin of the Medical Library Association*, 58(4): 587, 1970.

[11] Uwe Draisbach and Felix Naumann. A generalization of blocking and windowing algorithms for duplicate detection. In *2011 International Conference on Data and Knowledge Engineering (ICDKE)*, pages 18–24. IEEE, 2011.

[12] Amr Ebaid, Saravanan Thirumuruganathan, Walid G Aref, Ahmed Elmagarmid, and Mourad Ouzzani. Explainer: entity resolution explanations. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 2000–2003. IEEE, 2019.

[13] Yixiang Fang, Xin Huang, Lu Qin, Ying Zhang, Wenjie Zhang, Reynold Cheng, and Xuemin Lin. A survey of community search over big graphs. *The VLDB Journal*, 29:353–392, 2020.

[14] Ivan P Fellegi and Alan B Sunter. A theory for record linkage. *Journal of the American Statistical Association*, 64(328):1183–1210, 1969.

[15] Jeffrey Fisher, Peter Christen, Qing Wang, and Erhard Rahm. A clustering-based framework to control block sizes for entity resolution. *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2015. doi: 10.1145/2783258.2783396.

[16] Luis Gravano, Panagiotis G Ipeirotis, Hosagrahar Visvesvaraya Jagadish, Nick Koudas, Shanmugauelayut Muthukrishnan, Divesh Srivastava, et al. Approximate string joins in a database (almost) for free. In *VLDB*, volume 1, pages 491–500, 2001.

[17] Ibrahim Abaker Targio Hashem, Ibrar Yaqoob, Nor Badrul Anuar, Salimah Mokhtar, Abdullah Gani, and Samee Ullah Khan. The rise of "big data" on cloud computing: Review and open research issues. *Information systems*, 47:98–115, 2015.

[18] Mauricio A Hernández and Salvatore J Stolfo. The merge/purge problem for large databases. *ACM Sigmod Record*, 24(2):127–138, 1995.

[19] Arthur Hovanesyan. *Late payment prediction of invoices through graph features.* MSc Thesis, Delft University of Technology, 2019.

[20] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pages 604–613, 1998.

[21] Omid Jafari, Preeti Maurya, Parth Nagarkar, Khandker Mushfiqul Islam, and Chidambaram Crushev. A survey on locality sensitive hashing algorithms and their applications. *arXiv preprint arXiv:2102.08942*, 2021.

[22] Armand Joulin, Edouard Grave, Piotr Bojanowski, and Tomas Mikolov. Bag of tricks for efficient text classification. In *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Volume 2, Short Papers*, pages 427–431. Association for Computational Linguistics, April 2017.

[23] Hung-sik Kim and Dongwon Lee. Harra: fast iterative hashed record linkage for large-scale data collections. In *Proceedings of the 13th International Conference on Extending Database Technology*, pages 525–536, 2010.

[24] Hisashi Koga, Tetsuo Ishibashi, and Toshinori Watanabe. Fast hierarchical clustering algorithm using locality-sensitive hashing. In *Discovery Science: 7th International Conference, DS 2004, Padova, Italy, October 2-5, 2004. Proceedings 7*, pages 114–128. Springer, 2004.

[25] Bastijn Kostense. *Assessing COVID-19 impact on Dutch SMEs using dynamic network analysis.* MSc Thesis, Delft University of Technology, 2021.

[26] Vladimir I Levenshtein et al. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, volume 10, pages 707–710. Soviet Union, 1966.

[27] Bo-Han Li, Yi Liu, An-Man Zhang, Wen-Huan Wang, and Shuo Wan. A survey on blocking technology of entity resolution. *Journal of Computer Science and Technology*, 35:769–793, 2020.

[28] Tianrui Mao. *Graph-based entity resolution and its application in debtor payment prediction.* MSc Thesis, Delft University of Technology, 2022.

[29] Andrew McCallum, Kamal Nigam, and Lyle H Ungar. Efficient clustering of high-dimensional data sets with application to reference matching. In *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 169–178, 2000.

[30] Alvaro E Monge, Charles Elkan, et al. The field matching problem: algorithms and applications. In *Kdd*, volume 2, pages 267–270, 1996.

[31] Hao Nie, Xianpei Han, Ben He, Le Sun, Bo Chen, Wei Zhang, Suhui Wu, and Hao Kong. Deep sequence-to-sequence entity matching for heterogeneous entity resolution. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*, pages 629–638, 2019.

[32] Mahdi Niknam, Behrouz Minaei-Bidgoli, and Rouhollah Dianat. The role of transitive closure in evaluating blocking methods for dirty entity resolution. *Journal of Intelligent Information Systems*, 58(3): 561–590, 2021. doi: 10.1007/s10844-021-00676-3.

[33] George Papadakis, Ekaterini Ioannou, Claudia Niederée, Themis Palpanas, and Wolfgang Nejdl. Eliminating the redundancy in blocking-based entity resolution methods. In *Proceedings of the 11th annual international ACM/IEEE joint conference on Digital libraries*, pages 85–94, 2011.

[34] George Papadakis, Ekaterini Ioannou, Themis Palpanas, Claudia Niederée, and Wolfgang Nejdl. A blocking framework for entity resolution in highly heterogeneous information spaces. *IEEE Transactions on Knowledge and Data Engineering*, 25(12):2665–2682, 2012.

[35] George Papadakis, George Alexiou, George Papastefanatos, and Georgia Koutrika. Schema-agnostic vs schema-based configurations for blocking methods on homogeneous data. *Proceedings of the VLDB Endowment*, 9(4):312–323, 2015.

[36] George Papadakis, George Mandilaras, Luca Gagliardelli, Giovanni Simonini, Emmanouil Thanos, George Giannakopoulos, Sonia Bergamaschi, Themis Palpanas, and Manolis Koubarakis. Three-dimensional entity resolution with jedai. *Information Systems*, 93:101565, 2020.

[37] George Papadakis, Dimitrios Skoutas, Emmanouil Thanos, and Themis Palpanas. Blocking and filtering techniques for entity resolution: A survey. *ACM Computing Surveys (CSUR)*, 53(2):1–42, 2020.

[38] George Papadakis, Vasilis Efthymiou, Emmanouil Thanos, and Oktie Hassanzadeh. Bipartite graph matching algorithms for clean-clean entity resolution: an empirical evaluation. *Learning*, 3(28):34, 2022.

[39] Yaoshu Wang, Jianbin Qin, and Wei Wang. Efficient approximate entity matching using jaro-winkler distance. In *International conference on web information systems engineering*, pages 231–239. Springer, 2017.

[40] Wei Wu and Bin Li. Locality sensitive hashing for structured data: A survey. *arXiv preprint arXiv:2204.11209*, 2022.

[41] Xuyun Zhang, Christopher Leckie, Wanchun Dou, Jinjun Chen, Ramamohanarao Kotagiri, and Zoran Salcic. Scalable local-recoding anonymization using locality sensitive hashing for big data privacy preservation. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*, pages 1793–1802, 2016.

Glossary

Acronyms

AL1 AddressLine1. 19–21, 24, 35, 42

AL1N AddressLine1Number. 4, 5, 19, 21, 35, 40–42

AL1S AddressLine1Street. 4, 20, 21, 23, 40, 42

ANN Approximated Nearest Neighbor. 9, 26, 34

BKV Blocking Key Value. 5, 7–11, 16, 25, 26, 34, 35, 78

BRP Bucketed Random Projection. 9, 26

CC-ER Clean-Clean Entity Resolution. 1, 11, 16, 22, 23

CEP Cardinality Edge Pruning. 12

CNP Cardinality Node Pruning. 12

CoC Chamber of Commerce. 5, 17, 19, 21, 22, 38, 58, 66, 67, *Glossary:* Chamber of Commerce

D-ER Dirty Entity Resolution. vii, 1–3, 11, 16, 19, 23–25, 61–63

DF Document Frequency. 28, 31, 32

ER Entity Resolution. vii, xiii, xv, 1–6, 9, 10, 12, 13, 15, 16, 19, 21–23, 25, 31, 33, 35, 37, 39, 49, 53, 54, 57–59, 61–63, 65, 67, 68, 76

FN False Negative. 66

FP False Positive. 7, 36, 44, 48, 50, 66, 69

GT Ground Truth. xiii, xv, 15, 17, 19, 21–23, 25, 28, 31, 32, 48, 58, 68, 77

GT_1 Ground sub-Truth. xiii, xv, 32, 34, 36, 37, *Glossary:* Ground sub-Truth

IDF Inverse Document Frequency. 13, 23

LECP Low Entity Co-occurrence Pruning. 12

LPP Late Payment Prediction. xv, 4, 57–59, 63, 68

LSH Locality-Sensitive Hashing. vii, xv, 2–4, 9, 10, 25, 26, 29, 37, 38, 47, 48, 50, 52–54, 61–63, 65, 67–69, 76–78

MSE Mean Squared Error. 27

NLP Natural Language Processing. 26

PC Pairs Completeness. vii, xiii, 3, 9–11, 25, 26, 31–38, 62, 63, 67–69

PQ Pairs Quality. vii, 3, 9–11, 25, 26, 31, 32, 34–38, 62, 63, 67–69

RQ Research Question. 3, 61, 63

RR Reduction Ratio. vii, xiii, 9–11, 25, 31–38, 47, 68, 69

SQ Sub-Question. 3, 4, 61, 63

SQ1 Sub-Question 1. 3, 4, 61, 62

SQ2 Sub-Question 2. 3, 4, 62

SQ3 Sub-Question 3. 3, 4, 62, 63

SQ4 Sub-Question 4. 3, 4, 63

TF Term Frequency. 13, 23, 28, 29, 31

TF-IDF Term Frequency - Inverse Document Frequency. 13, 22, 23

TN True Negative. 66

TP True Positive. 36, 66, 67

TSU Theoretical SpeedUp. 25, 38, 47

UDF User Defined Function. 40, 65, *Glossary:* User Defined Function

WEP Weighted Edge Pruning. 12

WNP Weighted Node Pruning. 12

Glossary

Account A single row of the Account data set from Exact, containing divisions and their Accounts. This data set is used for applying the ER pipeline on. 5, 16, 19, 20, 22, 26, 41, 44, 50, 57–59, 63, 66–68, 76–78

AccountID The ID of a row in Accounts. 49, 53, 58

Accounts The data set used to apply ER on, the dataset contains rows referring to real-world persons and companies, this data set is dirty, meaning there exist duplicated references in to the same real-world entity. xiii, xv, 4, 5, 17, 19–23, 25–27, 35, 38–42, 44, 47–49, 57–59, 65, 67, 68, 76–78

ActiveAccounts A subset of Accounts in which each row has at least a single transaction assigned to them in the last two years. The data set has a size of roughly 85 million rows. 17, 19

AND-OR amplification The way the generated hashes of LSH are handled. Rows get multiple hash tables, consisting of hash AND-clauses. All the hashes of a hash table must be equal to be part of the same block. The different hash tables are the OR-clauses, causing the creation of multiple blocks per row. 30

Apache Spark Apache Spark is an engine allowing the processing of big data in a distributed fashion. 65

ApproxSim Approximated similarity of neighborhoods between two vertices. The approximation is done with MinHash and used as a threshold to consider the edge as noise. 50–54

BitMask Each property represents a specific bit, the specific bit is the BitMask of that property. xv, 40

Chamber of Commerce The Chamber of Commerce property from the Account data set. Each Dutch company must have a Chamber of Commerce number for tax and legal applications. 5, 20

City The City property from the Account data set. 4, 20, 21, 24, 40, 42

ClassifierMask The PropertyMask that is used for this classifier. The classifier must have all the selected properties of this mask available for each row that needs to be classified. 40, 44, 48

Company Generated dataset in which the Accounts are filtered by Name if they contain company related strings. 58, 77

CompanyAccount An Account that is in the Company subset of Accounts. 58

Country The Country property from the Account data set. 20

Email The Email property from the Account data set. 4, 5, 20–24, 35, 40, 42, 58, 79

Entity An group of merged Accounts created by the selected EntityMapper. vii, 16, 49, 52–54, 57, 62, 63, 65–68

EntityID A new ID given to rows of Accounts, mapping multiple rows to a single entity that all should refer to the same real-world company. The EntityMapper transforms the ID to the EntityID. 4, 49, 50, 53, 54, 59, 61, 62, 67, 77, 78

EntityMapper A data set that has two columns, an initial ID and an EntityID, it maps the unique ID to an EntityID shared by other IDs. xv, 16, 49, 52–54, 57–59, 61–63, 67, 77

EntityMapping The mappings of an EntityMapper that maps the unique ID to an EntityID shared by other IDs. xv, 16, 50

Exact Cloud business software for Small and medium-sized enterprises and their accountants, the thesis is conducted on the Account dataset from Exact. i, iii, 4, 5, 16, 17, 19, 22, 23, 26, 38, 57, 61, 63, 65, 67, 76

FastText Library developed by Facebook that is an extension of Word2Vec. Instead of words, a range of N-grams is used to overcome spelling errors and find more structure within the data. 29, 31, 77

Fax The Fax property from the Account data set. 20, 21, 23, 79

Ground sub-Truth A subset of GT, a smaller set is required to perform experiments in feasible time and space. The measurements created from this GT subset are not the same as measurements that are created from the GT but are sufficient to compare the results of the experiments. xiii, 32

initcap String transformation, changing the first letter of each word to a capital letter and the other letters to lowercase. 19–21

InverseID Instead of using AccountIDs, the InverseID can be used. The InverseIDs are groups of accounts that have the same properties. The lowest InverseID of the group will represent the InverseID. xv, 24, 26, 40, 47, 50, 52–54, 77

LSH-RB1000 A pruning method applied to runs containing LSH, in which blocks larger than 1000 InverseIDs are regrouped by their Name. 48

MatchPropertyMask A mask given to a Match, which is a combination bitwise-AND operation between the PropertyMasks of the two rows of the Match. 44, 47

MinHash An implementation of LSH, that relies on sparse vectors representing each row. vii, xiii, 4, 9, 25, 26, 28–33, 37, 50, 52, 53, 61–63, 65, 67–69, 76–78

MissingLinks Clustering method that uses MinHash to discover missed edges. If the vertices share a majority of neighboring vertices, then they can be linked again via MinHash by creating an edge between them. Afterward, the disconnected sub-graphs and transforming them into unique EntityIDs. xv, 49, 53, 54, 62, 63

MultiGram N-gram creation in the manner of FastText, in which the tokens are constructed with a range of N. 34, 65, 69

N-gram Splitting a sentence in words of size N, can also be applied on character level. vii, 8–10, 26, 29–31, 41, 50, 62, 69, 77

Name The Name property from the Account data set. xiii, 4, 5, 19–28, 32, 35, 38–44, 47, 48, 58, 62, 69, 77, 79

NoClustering Clustering method of applying no clustering method but only finding the disconnected sub-graphs and transforming them into unique EntityIDs. xv, 50–52

NoiseRemoval Clustering method that uses MinHash to see if the vertices of an edge share sufficient neighbors, without much overlap in neighbors, the edge is deemed as noise and is discarded. Afterward, the disconnected sub-graphs and transforming them into unique EntityIDs. xiii, xv, 49, 51–55, 62

Phone The Phone property from the Account data set. 4, 20–24, 35, 40, 58, 79

Postcode The Postcode property from the Account data set. 4, 5, 19–21, 23, 24, 35, 36, 40, 42, 68

PropertyMask A mask given to rows of Accounts, this mask resembles which properties are available for a given row. xiii, 40–42, 44–47, 76, 77

PySpark The Apache Spark implementation in Python code language, it essentially maps the majority of available functions from Python to the Spark implementation in Scala. 30, 40, 41, 50, 52, 65

RunAugBaseline Run in which blocks are created by grouping on Name in combination with augmentation of other properties. xiii, 38, 47, 48, 50–55, 58, 61, 62, 67

RunAugLSH Run in which blocks are created by applying LSH on Name in combination with augmentation of other properties. xiii, 38, 44, 48, 50–55, 58, 61, 62

RunBaseline Run in which blocks are created by grouping on Name. xiii, 38, 47–55, 58, 61, 62

RunLSH Run in which blocks are created by applying LSH on Name. xiii, 38, 48, 50–55, 58, 61, 62, 67

Sliding Comparison Window A blocking technique which simply partitions the data on the created BKVs, afterward a window is slid over the alphabetically sorted rows. Their BKVs are compared with Levenshtein; if they are below the selected threshold, they are discarded. 34–37

Sliding Window A blocking technique which simply partitions the data on the created BKVs and afterward slides a window over the sorted BKVs, creating closely related blocks in terms of BKV. 7, 10, 25, 34, 35

Soundex Method that transforms a word to its phonetic resemblance. 10

Standard Blocking A blocking technique which simply partitions the data on the created BKVs. 7, 9, 10, 32, 34–37, 62

State The State property from the Account data set. 4, 20, 21, 24, 40, 42

User Defined Function Functions defined by the user in the PySpark implementation, however, require too much overhead that they are unusable within a big data application. 40

VATNumber The VATNumber property from the Account data set. 5, 17, 19–22, 38, 66, 67

Website The Website property from the Account data set. 4, 20, 21, 23, 24, 35, 40–42, 79

Word2Vec Representing a word as a dense vector. This representation can be learned via machine-learning. 9, 26–28, 77

A

Appendix

Here, additional information is provided. This information is not important to know for this thesis but is important for reproducing the results of this thesis.

A.1. Stopwords

```
(^|\s)((((b\.?v|n\.?v\.?t|t\.?v\.?n|t\.?n\.?v|v\.?o\.?f|v\.?v\.?e|n\.?v)|[\.\-]*|
test|aan|vennootschap|maatschap|eenmanszaak|stichting|zzp|van|vd|de|der|den|het|en|
dhr|hr|mevr|mevrouw|heer|nederland|netherlands|fam|fa|gmbh|bewoners|deze|mw|holding|
ltd|the|bvba|group|in|co|familie|voor|and|you|for|mr|ter|bij|miss|mrs)\.?)(\s|$))+
```

Listing A.1: The regex of stopwords used to clean the Name field

A.2. Website regex

The pre-processing phase uses multiple regexes, which are shown below.

A.2.1. Website suffix regex

```
((h+t+p+s+:*[\/\\\]+)|(www[\/\\\]*[\.,]))+
```

Listing A.2: The regex used to leniently remove Website prefixes

A.2.2. Website extract regex

```
[\-a-z0-9]+(\.[\-a-z0-9]{2,})+
```

Listing A.3: The regex used to extract valid Website addresses

A.3. Email regex

```
([a-z0-9_\-][a-z0-9_\-\.\+]*)?[a-z0-9_\-]+@[a-z0-9\-\._]*\.[a-z0-9\-]{2,}
```

Listing A.4: The regex used to extract valid Email addresses

A.4. Phone/Fax regex

```
(\+[0-9]{2}|\+[0-9]{2}\(0\)|\(\+[0-9]{2}\)\(0\)|00[0-9]{2}|06?)
([0-9]{9}$|[0-9\-\s]{10})
```

Listing A.5: The regex used to extract valid Phone/Fax numbers

A.5. Applied Business Filter

```
'bv', 'vof', 'nv', 'cv', 'stichting', 'maatschap', 'vereniging', 'coöperatie',
'cooperatie', 'eenmanszaak', 'zzp', 'vennootschap', 'asml', 'prosus', 'unilever',
'shell', 'ing', 'philips', 'heineken', 'ahold', 'delhaize', 'akzonobel', 'dsm',
'exact'
```

Listing A.6: The words to filter the accounts on to have a high likelihood that the filtered account represent a company