UNIVERSITÉ DE MONTRÉAL

LEARNING ALGORITHM TO AUTOMATE FAST AUTHOR NAME DISAMBIGUATION

BANAFSHEH MEHRI
DÉPARTEMENT DE MATHÉMATIQUES ET DE GÉNIE INDUSTRIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE INDUSTRIEL)
OCTOBRE 2017

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé :

LEARNING ALGORITHM TO AUTOMATE FAST AUTHOR NAME DISAMBIGUATION

présenté par : MEHRI Banafsheh
en vue de l'obtention du diplôme de : Maîtrise ès sciences appliquées
a été dûment accepté par le jury d'examen constitué de :

ACKNOWLEDGMENTS

I would like to thank my adviser, Professor Yves Goussard for guiding and supporting me over the course of my Master's degree. He has set an example of excellence as a mentor and instructor. A very special gratitude goes out to Dr. Matthieu Voorons for all of his guidance through out my academic pathway; his discussions, ideas, and feedbacks have been absolutely invaluable. It was a great opportunity to work with him. I also thank my co-adviser Professor Martin Trepanier for his help and advice. I'd like to thank my fellow graduate students, technicians and those who contributed to this research. I am very grateful to all of you.

I would like to thank Science-Metrix company for support and insight.

RÉSUMÉ

La production scientifique mondiale représente une quantité massive d'enregistrements auxquels on peut accéder via de nombreuses bases de données. En raison de la présence d'enregistrements ambigus, un processus de désambiguïsation efficace dans un délai raisonnable est nécessaire comme étape essentielle pour extraire l'information correcte et générer des statistiques de publication. Cependant, la tâche de désambiguïsation est exhaustive et complexe en raison des bases de données volumineuses et des données manquantes. Actuellement, il n'existe pas de méthode automatique complète capable de produire des résultats satisfaisants pour le processus de désambiguïsation.

Auparavant, une application efficace de désambiguïsation d'entité a été développée, qui est un algorithme en cascade supervisé donnant des résultats prometteurs sur de grandes bases de données bibliographiques. Bien que le travail existant produise des résultats de haute qualité dans un délai de traitement raisonnable, il manque un choix efficace de métriques et la structure des classificateurs est déterminée d'une manière heuristique par l'analyse des erreurs de précision et de rappel. De toute évidence, une approche automatisée qui rend l'application flexible et réglable améliorerait directement la convivialité de l'application. Une telle approche permettrait de comprendre l'importance de chaque classification d'attributs dans le processus de désambiguïsation et de sélectionner celles qui sont les plus performantes. Dans cette recherche, nous proposons un algorithme d'apprentissage pour automatiser le processus de désambiguïsation de cette application.

Pour atteindre nos objectifs, nous menons trois étapes majeures: premièrement, nous abordons le problème d'évaluation des algorithmes de codage phonétique qui peuvent être utilisés dans le *blocking*. Six algorithmes de codage phonétique couramment utilisés ont été sélectionnés et des mesures d'évaluation quantitative spécifiques ont été développées afin d'évaluer leurs limites et leurs avantages et de recruter le meilleur. Deuxièmement, nous testons différentes mesures de similarité de chaîne de caractères et nous analysons les avantages et les inconvénients de chaque technique. En d'autres termes, notre deuxième objectif est de construire une méthode de désambiguïsation efficace en comparant plusieurs algorithmes basés sur les *edits* et les *tokens* pour améliorer la méthode du *blocking*. Enfin, en utilisant les méthodes d'agrégation *bootstrap (Bagging)* et *AdaBoost*, un algorithme a été développé qui utilise des techniques d'optimisation de particle swarm et d'optimisation de set covers pour concevoir un cadre d'apprentissage qui permet l'ordre automatique des weak classifiers et la détermination de leurs seuils. Des comparaisons de performance ont été effectuées sur des données réelles extraites du *Web of Science (WoS)* et des bases de données bibliographiques *SCOPUS*.

En résumé, ce travail nous permet de tirer des conclusions sur les qualités et les faiblesses de chaque algorithme phonétique et mesure de similarité dans la perspective de notre application. Nous avons montré que l'algorithme phonétique NYSIIS est un meilleur choix à utiliser dans l'étape de *blocking* de l'application de désambiguïsation. De plus, l'algorithme de Weighting Table-based surpassait certains des algorithmes de similarité couramment utilisés en terme de efficacité de temps, tout en produisant des résultats satisfaisants. En outre, nous avons proposé une méthode d'apprentissage pour déterminer automatiquement la structure de l'algorithme de désambiguïsation.

ABSTRACT

The worldwide scientific production represents a massive amount of records which can be accessed via numerous databases. Because of the presence of ambiguous records, a time-efficient disambiguation process is required as an essential step of extracting correct information and generating publication statistics. However, the disambiguation task is exhaustive and complex due to the large volume databases and existing missing data. Currently there is no complete automatic method that is able to produce satisfactory results for the disambiguation process.

Previously, an efficient entity disambiguation application was developed that is a supervised cascade algorithm which gives promising results on large bibliographic databases. Although the existing work produces high-quality results within a reasonable processing time, it lacks an efficient choice of metrics and the structure of the classifiers is determined in a heuristic manner by the analysis of precision and recall errors. Clearly, an automated approach that makes the application flexible and adjustable would directly enhance the usability of the application. Such approach would help to understand the importance of each feature classification in the disambiguation process and select the most efficient ones. In this research, we propose a learning algorithm for automating the disambiguation process of this application. In fact, the aim of this work is to help to employ the most appropriate phonetic algorithm and similarity measures as well as introduce a desirable automatic approach instead of a heuristic approach.

To achieve our goals, we conduct three major steps: First, we address the problem of evaluating phonetic encoding algorithms that can be used in blocking. Six commonly used phonetic encoding algorithm were selected and specific quantitative evaluation metrics were developed in order to assess their limitations and advantages and recruit the best one. Second, we test different string similarity measures and we analyze the advantages and disadvantages of each technique. In other words, our second goal is to build an efficient disambiguation method by comparing several edit- and token-based algorithms to improve the blocking method. Finally, using bootstrap aggregating (Bagging) and AdaBoost methods, an algorithm has been developed that employs particle swarm and set cover optimization techniques to design a learning framework that enables automatic ordering of the weak classifiers and determining their thresholds. Performance comparisons were carried out on real data extracted from the web of science (WoS) and the SCOPUS bibliographic databases.

In summary, this work allows us to draw conclusions about the qualities and weaknesses of each phonetic algorithm and similarity measure in the perspective of our application. We have shown that the NYSIIS phonetic algorithm is a better choice to use in blocking step of the disambiguation application. In addition, the Weighting Table-based algorithm outperforms some of the commonly used similarity algorithms in terms of time-efficiency, while producing satisfactory results. Moreover, we proposed a learning method to determine the structure of the disambiguation algorithm automatically.

TABLE OF CONTENTS

ACKNOWLEDGMENTS iii

RÉSUMÉ iv

ABSTRACT vi

TABLE OF CONTENTS viii

LIST OF TABLES x

LIST OF FIGURES xi

LIST OF ABBREVIATIONS xii

LIST OF APPENDICES xiii

CHAPTER 1 INTRODUCTION 1
1.1 Concepts and Definitions 2
1.2 Research Objectives 7
1.3 Thesis Plan 7

CHAPTER 2 LITERATURE REVIEW 8
2.1 Name-based Disambiguation 8
2.1.1 Blocking 10
2.1.2 Phonetic Encoding Algorithms 12
2.1.3 String Similarity Metrics 12
2.2 Ensemble Learning 14
2.2.1 Bootstrapping Aggregation (Bagging) 14
2.2.2 Boosting 15
2.3 Particle Swarm Optimization 17
2.4 Set Cover Optimization 19

CHAPTER 3 FAST AUTHOR NAME DISAMBIGUATION ALGORITHM 20
3.1 Overview 20
3.2 Pre-processing 21
3.3 Blocking 22

3.4 Classification Approach 22
3.5 Cascade Structure Configuration 27
3.6 Evaluation 29
3.6.1 Datasets 30
3.6.2 Results 30
3.6.3 Summary of the Results 31

CHAPTER 4 METHODOLOGY AND DESIGN 33
4.1 Experimental Setup 33
4.1.1 Step 1: Comparing Phonetic Encoding Algorithms 33
4.1.2 Step 2: Comparing String Similarity Metrics 38
4.1.3 Step 3: Training Methodology for the Cascade Classifier 48

CHAPTER 5 RESULTS 57
5.1 Results of Comparative Study of Phonetic Distance Metrics 57
5.2 Results of Comparative Study of String Similarity Algorithms 58
5.3 Results of Automating Cascade Structure 65

CHAPTER 6 CONCLUSION 70
6.1 Summary of the Results 70
6.2 Future Work 71

REFERENCES 73

APPENDICES 82

LIST OF TABLES

Table 3.1	Characteristics of different types of weak classifiers	24
Table 3.2	Structure of the first disambiguation block - WoS subset (large size)	24
Table 3.3	Structure of the second disambiguation block - WoS subset (large size)	25
Table 3.4	Structure of the first disambiguation block - SCOPUS subset (small size)	26
Table 3.5	Structure of the second disambiguation block - SCOPUS subset (small size)	26
Table 3.6	Results of the manual cascade approach (WoS) (Voorons et al., 2017)	30
Table 3.7	Results of the manual cascade approach (SCOPUS) (Voorons et al., 2017)	31
Table 4.1	Weighting Table-Based algorithm parameters	39
Table 4.2	Parameters used in the learning process	50
Table 5.1	Results of evaluating phonetic algorithms using different metrics	57
Table 5.2	Structure of the first disambiguation block - WoS subset (large size)	65
Table 5.3	Structure of the second disambiguation block - WoS subset (large size)	66
Table 5.4	Results of the automatic cascade approach (WoS)	66
Table 5.5	Structure of the first disambiguation block - SCOPUS subset (small size)	67
Table 5.6	Structure of the second disambiguation block - SCOPUS subset (small size)	68
Table 5.7	Results of the automatic cascade approach (SCOPUS)	69

LIST OF FIGURES

Figure 2.1 Demonstration of the Bagging algorithm 15
Figure 2.2 Demonstration of the Boosting algorithm 16
Figure 2.3 Demonstration of the AdaBoost algorithm 16
Figure 2.4 Particle Swarm Optimization algorithm flow chart 18
Figure 3.1 Cascade approach (Voorons et al., 2017) 23
Figure 3.2 Process of entity aggregators and recursive aggregators (Voorons et al., 2017) 23
Figure 3.3 Detailed design of an aggregator (Voorons et al., 2017) 28
Figure 4.1 Methodology steps 44
Figure 4.2 Previewing first few entries of a similarity matrix 45
Figure 4.3 Hierarchical agglomerative clustering tree plot 46
Figure 4.4 Overall Demonstration of the Proposed Framework 49
Figure 4.5 Training Methodology 50
Figure 5.1 Intra-buckets average distances 59
Figure 5.2 Intra-buckets average distances 60
Figure 5.3 Results of computing the rand index for different similarity algorithms . . 61
Figure 5.4 Number of buckets per cutree height for the HAC algorithm 62
Figure 5.5 Number of buckets per cutree height for the HAC algorithm 63
Figure 5.6 Total execution time 64

LIST OF ABBREVIATIONS

AI	Artificial Intelligence
ANN	Artificial Neural Networks
BOW	Bags of Words
CC	Cascade/Compound Classifier
DBSCAN	Density-Based Spatial Clustering of Applications with Noise
EA	Entity Aggregator
FN	False Negative
FP	False Positive
GNU	GNU's Not Unix
GPL	General Public License
HAC	Hierarchical Agglomerative Clustering
HMM	Hidden Markov Model
ILP	Integer Linear Programming
IR	Information Retrieval
ISI	Institute for Scientific Information
MIP	Mixed Integer Programming
MSCP	Minimum Set Cover Problem
NED	Named Entity Disambiguation
NEL	Named Entity Linking
NLP	Natural Language Processing
NSERC	Natural Sciences and Engineering Research Council of Canada
PSO	Particle Swarm Optimizer
RA	Recursive Aggregator
RI	Rand Index
SCO	Set Cover Optimizer
SSB	Sum of Squares Between clusters
SSE	Sum of Squared Errors
SVM	Support Vector Machines
TN	True Negative
TP	True Positive
WC	Weak Classifier
WoS	Web of Science

LIST OF APPENDICES

APPENDIX A CO-AUTHORSHIP 82
APPENDIX B CASCADE CONFIGURATION XML FILE FOR WEB OF SCIENCE DATASET (LARGE SIZE) 83

CHAPTER 1 INTRODUCTION

Authors of scholarly documents often share names which makes it difficult to identify each author's work. Hence, a major problem for any scholarly document in scientific citations is to recognize the person(s) who wrote it and to discover all the documents that belong to a given person. In the context of bibliographic citations this problem is known as ”Author Name Ambiguity" and it is made difficult by the several characteristics of bibliographic data such as 1) imperfect collection and transcription of information; 2) occurrence of duplicate records and swapped fields 3) presence of redundant information or data ambiguities (e.g., homonyms, author with multiple affiliations, etc.), cultural specificities (e.g., change of name through marriage) and non-discriminating data.

Most of the time disambiguation process is done by hand, and such a process is often challenging in terms of human capabilities. Computerized solutions, on the other hand, often produce accurate results; however, they do not have the ability to overcome the problem in a time-efficient way, and they are not appropriate in the domain of very large datasets.

In spite of many efforts from the research community to address the disambiguation problem, most of the proposed disambiguation methods require many improvements, especially in terms of time-efficiency. In fact, the time-consuming aspect of such methods is due to the computational load they require to process the available information, especially in the bibliographic domain where we are faced with large-scale data sets.

Beforehand, there have been many attempts that tried to facilitate the manual disambiguation task. For instance, DBLife (DeRose et al., 2007) is a platform that extracts author information from several resources and displays it in a standardized layout that is used for manual correction. Even though it can extract lists of database researchers together with structured, related information including publications that they have authored, it does not contain large-scale data integration due to its heavy computational cost. Another project that addresses the author ambiguity problem is the *ResearcherID*[1] which is a website that asks authors to link their articles to their researcher ID. This helps avoiding author misidentification, however, the web site is able to identify only those who claim and showcase their publications by registering an online account.

Formerly, a supervised disambiguation method (Voorons et al., 2017) with the ability to process real-size bibliographic databases and generate satisfactory results has been developed. This method is tuned distinctly to the data to determine the relative importance and mutual effects of diverse features such as *co-author*, *journal name*, *title* and *affiliation*. Despite having an efficient processing time, hence outperforming any existing methods of this kind, it still lacks an approach to find the

[1]www.researcherid.com

best metrics and the learning method to stand upon an automated approach to determine the best configuration for its classifiers; hence, it is ill-suited for complex databases with many features, and unable to provide insights about each classifier's impact and importance.

Thus, this research aims to conduct a comparative evaluation and analysis of metrics being used in the aforementioned algorithm to help to choose the right ones in disambiguation process, and afterwards, it also aims to develop a learning algorithm that constructs the ordering of the classifiers and their respective thresholds. Consequently, it would make possible to adapt to any type of the input file, and to cope with large and complex databases.

This research was conducted in collaboration with Science-Metrix, a Canadian company specialized in the measurement and evaluation of scientific and technological activities, as part of the Natural Sciences and Engineering Research Council of Canada (NSERC) Grant #EGP 433987-12.

1.1 Concepts and Definitions

In this section, short summaries of essential concepts related to the research topic are presented:

Bibliographic Database

It is a database containing bibliographic records about scientific journals, books and conference proceedings and it holds information such as authors information, keywords, titles, affiliation, and references. Generally, bibliographic databases provide a comprehensive overview of the world's research output.

Well-known examples include PubMed [2] that is the world largest medical bibliographic database on life sciences and biomedical information developed and maintained by the United States National Library of Medicine (NLM). It became a free resource in 1997, covers all biomedical fields and contains more than 20 million citations in 40+ languages. Another one is SCOPUS [3] that is the largest abstract citation and database of peer-reviewed literature. It provides precise entry points to literature in the fields of science, technology, medicine, social sciences and art and humanities. It has information regarding abstracts and citations for 50+ million items and over 20 thousand peer-reviewed journals. It supports creating alerts on getting informed about certain publications or citations in other articles through its user interface, making it unique with a broad usage. Another one is the Web of Science (WoS) [4] core collection that is another famous interconnected resource information that enables searching science related scholarly articles. It indexes citations

[2]https://www.ncbi.nlm.nih.gov/pubmed/

[3]https://www.scopus.com/home.uri

[4]https://www.webofknowledge.com

from thousands journals in the areas of science, social sciences and arts and humanities. The indexing includes 12,000+ scholarly journals, including open-access, and over 160,000+ conference proceedings.

Named Entity Disambiguation

In information extraction, a named entity is a real object such as person, location, organization, etc that can be denoted with a proper name. Example of named entities could be authors in bibliographic databases. With vast and enormous amount of data being represented in such databases, which can be because of typographical errors and imperfect information, it is difficult to recognize named entities in a manner that each unique item refers to its citations only. This raises the problem of named entity disambiguation. Author name disambiguation is a type of named entity disambiguation that applies to scholarly articles and the aim is to find all the publication records of the same author and distinguish them from the others. In other words, it can be defined as partitioning of a collection of data records into clusters where all records belong to a unique entity. Typical approaches of author name disambiguation rely on information about the authors such as email addresses, their affiliations, co-authors and citation graphs that helps identifying them among the all. Accordingly, this information can be used to implement a machine learning classifier that decides whether two entities refer to the same author or not.

Blocking

It refers to partitioning a set of records in a dataset into blocks and then only comparing records that belong to the same block. Blocking can reduce significantly the amount of time required to perform pair-wise comparisons, making it a suitable component of disambiguation techniques. It helps to create scalable frameworks of disambiguation by providing an effective way of comparing entities in a large volume of data. Blocking uses a metric to distinguish between the records. This metric is typically a similarity distance between string records or any type of encoder that generates relevant information necessary to discriminate.

Bag Of Words (BOW)

In learning from text, one of the fundamental problems is that the length of the entity we are learning from (the title of the document for example) can vary so much that it becomes impossible to use them as an input feature. In order to overcome this problem, we can just consider the significant words that make up the entity under study; regardless of order, grammar and number of occurrences. BOW is a simplified modeling used widely in information retrieval (IR), natural language processing (NLP) and image processing (by treating image features as words) and it is used to extract and quantify features, perform object categorization, search, document classification and topic modeling.

Supervised Learning Algorithms

Within the context of artificial intelligence (AI) and machine learning, a supervised learning method is defined as a learning algorithm in which the training data is a combination of a set of examples of input subjects and their desired outputs. For example, in image processing techniques, a supervised learning for an AI system takes various labeled pictures in different categories, then after a specific amount of observation and training, such system should be able to distinguish between them and categorize unlabeled images. It is called supervised training because the information is available to the learner beforehand, and it can use a direct measure of its results to improve its performance. Unsupervised learning, on the other hand, refers to an algorithm that can learn to perform the desired task from unlabeled data.

One of the goals of this research is to enhance the disambiguation method by implementing a supervised learning algorithm that learns how to generate the best configuration of classifiers.

Weak Classifier (WC)

WCs refers to simple predictors implemented by using elementary classification features that perform slightly better than a random classifier. WCs make decisions based on simple rules avoiding complex predictions. These rules can be implemented using thresholds on similarity measures of single classification features (e.g. surname or forename) in disambiguation problems. Consequently, such WCs are efficient in terms of computational cost. The classic example of a weak classifier is a *Decision Stump* that is a one-level decision tree.

Cascade Classifier (CC)

A cascade classifier is composed of several weak classifiers, assembled in a parallel structure that is followed by a decision operator. The structure of the CC and threshold values for the WCs are determined using a supervised training method.

Phonetic Encoding Algorithm

The phonetic algorithm is a method of hashing words and names based on the way that they are pronounced. There are many implementations based on several commonly used algorithms such as

Soundex, NYSIIS, Metaphone and Double Metaphone. An implementation of such algorithms is called a phonetic encoder, which is a function that takes a word as an input and produces the output of its pronunciation on codes. In fact, such algorithms allow comparisons of words by associating two different words with similar pronunciation with the same code, thereby providing an option for finding approximate rather than exact matches. In this research, phonetic encoding algorithms are used in blocking method to reduce the number of record-pair comparisons to calculate.

String Similarity Metric

String similarity algorithms are used to measure distances between two strings. They are used in many domains such as document clustering, consolidation of data, fraud detection, word-sense disambiguation, natural language processing and information retrieval. In the context of the application under study in this research, the decision of WCs is based on a threshold on the similarity between features such as surname, forename, etc. Therefore, choosing a suitable string similarity metric is critical for finding an appropriate compromise between speeding up the process, reducing sensitivity to thresholds and increasing the ability to cluster similar words (as well as avoiding being too discriminative to separate highly similar ones).

Clustering

Clustering refers to an unsupervised learning process of classifying objects into groups where the members of each group are as similar as possible to one another, and different groups are as dissimilar as possible from one another. The difference between clustering and classification is that classification is a supervised method for predicting the class of the data from pre-labeled classified instances, whereas clustering is an unsupervised learning that tends to find ”natural" grouping of data given un-labeled instances. Typically, a good clustering method should produce high quality clusters in which the intra-cluster (within a cluster) similarity is high and the inter-cluster (between clusters) similarity is low. The quality of a clustering result also depends on the similarity measure used by the method. However, it is often very hard to define ”similar enough" or ”good enough", and the answer is typically highly subjective. Therefore, the evaluation process for the similarity measure that is used in clustering requires specific metrics that are defined based on the application at hand.

There are many applications for clustering. For instance, in business it can be used to discover and characterize customer segments for marketing purposes (Montani and Leonardi, 2014) whereas in biology it can be used for classification of plants and animals given their features (Rhee and Mutwil, 2014). Clustering methods also applied to disambiguation problem (Nadimi and Mosakhani, 2015) (Liu et al., 2015) (Caron and van Eck, 2014) and revealed successful results of their applications in this domain.

Bootstrapping Aggregation (Bagging)

Bootstrapping is a meta-algorithm used to reduce variance and avoid over-fitting in a machine learning algorithm, and its principle is to approximate the sampling distribution by simulating it from a model of the data, and treating the simulated data just like the real data. In bootstrap algorithms (Efron, 1979), we draw many independent bootstrap samples from a set of data points X, then we evaluate the corresponding bootstrap replications $\overset{*}{X}_1, \overset{*}{X}_2, \ldots \overset{*}{X}_n$, and we estimate the standard error for the number of bootstrap samples used. Bootstrapping Aggregation (Bagging) (Breiman, 1996) relies on the bootstrap technique, and it is a way of building ensemble learners by using the same learning algorithm, but training each learner on a different set of data. We create random subsets (bags) of data, each one containing $N' \sqsubset N$ data points, where N is the number of training instances in the original data. We use each bag to train a different model, and we obtain an ensemble of different models that can be combined through model averaging.

Particle Swarm Optimization (PSO)

Particle swarm optimization (PSO) is a stochastic optimization technique that was introduced by Eberhart and Kennedy (Eberhart and Kennedy, 1995) and is inspired by the social behavior of bird flocking or fish schooling. This type of algorithm tries to find a solution in a search space by setting a collection of particles that move around this search space taking into the account their own best past location and the best past location of the assembled swarm. It explores a multi-dimensional search space in which all the particles locate the optima. Although it resembles an evolutionary approach, this method does not use selection operator and all population members survive to the end. The emergent behavior of particles will result in iterative amelioration of the quality of problem solutions over time. We used the PSO to find the best set of values for classifiers thresholds in disambiguation method.

Set Cover Optimization (SCO)

Given a universe of items U and a collection of a set of items S, minimum set cover problem (Gens and Levner, 1980) tries to find the minimum number of sets that covers the whole universe. This problem is a NP-hard problem (proven by (Karp, 1972)). An instance of such problem can be viewed as a graph, with sets represented by vertices on the left, the universe represented by vertices on the right, and the inclusion of elements in sets represented by edges. The objective, then is to come across a minimum cardinality subset of left-vertices which covers all of the right-vertices.

We used SCO to figure out the minimum set of classifiers that minimizes the number of incorrectly classified records and the processing time in order to find the best arrangement of the classifiers in the disambiguation method.

1.2 Research Objectives

The aim of this research is to optimize a cascade disambiguation application. For this purpose first, we conduct comparative evaluations of different metrics used in the disambiguation task such as string similarity measures and phonetic encoding algorithms. Such evaluations enable the use of most appropriate measures in the studied application. Second, we develop a learning algorithm that can establish the best order and threshold values for the cascade structure of classifiers. All in all, these optimization steps should increase adaptability, flexibility, and usability of such algorithm to extend its utilization to different domains and data sets.

1.3 Thesis Plan

In Chapter 2, we explain a background literature review on most of the concepts related to this research, and we provide information with regard to previous works. Chapter 3 describes the existing algorithm, which is the subject of our research. We briefly explain the elements of the algorithm in order to provide understanding of its limitations that are target topics of our research. First an overview of the method is introduced. Second, we discuss the pre-processing, classification approach and the algorithm structure, and finally, we discuss the evaluation process. Chapter 4 includes the design methodology and the steps of our research. Once the context and problem are explained, we take a look at the method used to conduct our experiments. We focus on three major elements (similarity metrics and phonetic encoding algorithms evaluation and development of a learning algorithm to generate the best configuration of the classifiers) in order to optimize the application at hand. Chapter 5 summarizes our results and findings, and finally, in Chapter 6 a general discussion and conclusion about the achievements of this research are presented as well as some directions for the possible future studies.

CHAPTER 2 LITERATURE REVIEW

This chapter reviews the literature and provides background information about the concepts and methodologies concerned with this research. First, in Section 2.1 we define the problem of name-based disambiguation in a bibliometric context and review related works that address this problem. Then we explain the ensemble learning method in Section 2.2 and we review techniques and methods that are used within its context. Since our methodology consists of three separate optimization steps, we provide background information about each of them in the following sections. Section 2.3 presents the definition of Particle Swarm Optimization algorithm and the variety of its applications and settings. Section 2.4 deals with the Set Cover problem and related works in literature that attempted to solve this problem.

2.1 Name-based Disambiguation

Bibliometric research[1] is an important methodology to evaluate the output and impact of scientific activities of researchers and institutions. The presence of ambiguous names corresponding to a large number of authors is a well-known major problem to bibliometric assessors and digital library users. Cultural aspects of naming conventions add more to the problem. For example, Chinese names represent the highest level of ambiguity since many Chinese share a few family names such as ”Wang" and ”Zhang" that can correspond to over a million of authors in many bibliographic databases. The disambiguation of bibliographic data seems inevitable to avoid reducing the accuracy of bibliometric evaluations. Name-based disambiguation is a fundamental step in bibliometric analyzes and it aims to associate each author to its publication records in a bibliographic database. Knowing individuals, in principle, is also crucial for establishing new resources such as citation/collaboration networks and author profiles.

Some previous studies were conducted with the goal of developing time-efficient algorithms for name-based disambiguation by using density-based classification algorithms (Huang et al., 2006), probabilistic approaches (Torvik and Smalheiser, 2009) (Torvik et al., 2005), boosted-tree method (Wang et al., 2012), hierarchical clustering (Cen et al., 2013) and decision trees (Treeratpituk and Giles, 2009). Although techniques used to solve small-scale disambiguation problems are effective, they mostly have a high computational cost, hence, they are inappropriate for processing large databases (Mitra et al., 2005). In fact, when dealing with large-scale problems, methods proposed in the literature are poorly efficient due to the size and variability of the data, or require auxiliary

[1] "A quantitative method to take into the account analysis of related publications in order to examine the knowledge structure and development of research fields."

information that may not be available at any time, hence forcing them to require a long processing time to produce the results.

Author name disambiguation is a particular case of name-based entity disambiguation, in which the entities being disambiguated are the authors of scientific publications in bibliographic databases. Such a problem can be defined as: partitioning of a collection of author information records into clusters where all records belong to a unique author. It is divided into multiple parts or N-class partitioning problem, in which a set of N_{R} bibliographical records $R_{\mathrm{i}}(0 < i < N_{\mathrm{R}} - 1)$ has to be associated to a set of N_{E} authors $E_{\mathrm{j}}(0 < j < N_{\mathrm{E}} - 1)$ based on predefined criteria.

Although author name disambiguation can be considered as a binary comparison problem, it cannot be accomplished by performing a pair-wise comparison of all records (since the calculation time is quadratic), especially when operating on very large datasets such as bibliographic databases. Additionally, performance of pairwise classification may be hindered by the so-called violation of transitivity phenomenon (Loomes et al., 1991) that can be explained as: $E1 \sim E2$ and $E1 \sim E3$ but $E2 \nsim E3$ where $\sim$ is a similarity operator and $E1$, $E2$ and $E3$ are three entities to be classified. At the beginning of the classification process, violations of transitivity are important because the information collected by each author entity is very limited, and may be of low quality. Two similar records can easily be seen as belonging to different authors, which results in over-segmentation because of this lack of quality information. Several approaches have been developed (Torvik and Smalheiser, 2009) (Culotta et al., 2007) to tackle this problem, but they appear not suited to disambiguation of large bibliographic databases due to the degree of the extra computations they require.

Moreover, author name disambiguation is faced with two major problems: bibliographic databases that contain author records usually include a large volume of data, and the quality of the data is low. Thus far, manual or semi-automatic disambiguation techniques were used (Balsmeier et al., 2015) to overcome these problems; however, results of such methods are questionable for frequent surnames, e.g. ”Smith" or ”Wang" because the information collected by each entity is limited. In addition, the processing time of these methods for real-world, large databases is significantly high. To solve these problems, some automatic methods have been developed by researchers that can be divided into two categories according to whether or not they need supervised training in order to specify the values of the tuning parameters.

There have been several approaches regarding unsupervised disambiguation that avoided any training method without using privileged information. For instance, (Cota et al., 2010) designed a heuristic-based hierarchical method by combining similarity functions, without knowing a priori the correct number of clusters. This method was successfully applied to large-scale datasets, but its main drawback was a very high computational cost. In another research, (Sun et al., 2015) developed an entity disambiguation algorithm using the artificial neural networks (ANNs). They have shown that without using training data and by incorporating semantics of contexts, the quality of the disambiguation process can be significantly improved. Nonetheless, the method was computationally intensive and hard to interpret and understand.

Supervised disambiguation methods, on the other hand, were developed based upon using sequential Bayesian classification and Support Vector Machines (SVM) (Han et al., 2004), K-spectral clustering (Giles et al., 2005) and hidden Markov models (Tang et al., 2012). Although these techniques can produce accurate results, all of them suffer from high computational costs and their inability to generalize in the presence of small amounts of training data.

To cope with the problem of speed, there were some attempts to make the process of entity disambiguation more time-efficient; as an example, Blanco et al. built a probabilistic model that significantly increases the speed of entity linking (linking free text to entities) in web search queries (Blanco et al., 2015). One limitation of this model is the lack of scalability which makes it inappropriate in the domain of large datasets.

Another work (Nguyen et al., 2014) attempted to establish a scalable efficient method to perform name-based entity disambiguation of unstructured data in the context of the web of data in a reasonable time. The result of this research was a name entity disambiguation system that reconciles high output quality with high-throughput usage at the Web scale, but it has a limit of the disambiguation method being highly dependent on the domain-specific features. The method was developed to perform well on specific domains (such as annotated news articles), but it was not possible to extend its usage and generalize the disambiguation process to other types of data.

Another way of addressing the problem of speed is to employ a method called *blocking* that will be explained in the following section.

2.1.1 Blocking

Blocking is a method used to speed up the disambiguation process (Christen, 2012). To perform blocking, we partition a set of records of a dataset into groups (blocks) of similar surnames, and then we only compare those records that were put together in the same block instead of comparing all the records together. Basically, blocking tends to reduce the amount of time required by pairwise comparisons in applications such as disambiguation. This reduction of time comes with the slight price of reducing the efficiency because of not considering all the possible solutions in all records of the dataset. Nonetheless, this trade-off between speed and efficiency could be in favor of the disambiguation application because it avoids the exhaustive search and has a minimal impact on the results.

The blocking method has been successfully applied in the domain of name-based disambiguation.

For instance, Lee et al. (Lee et al., 2005) used a two-level blocking framework to solve the mixed citation problem in bibliographic digital libraries with the significant reduction in computation complexity to O(C |B|), with C being the number of blocks and |B| the average size of them. In another work, (Huang et al., 2006) designed a method to solve the name-based disambiguation problem with the help of a blocking module that can significantly reduce the cost of similarity calculation.

There are several approaches that have been used in literature to implement blocking in author name disambiguation method:

- **Spelling-based**: This approach suggests to group author name-based on their name spellings. String similarity measures can be used in this approach to measure the similarity distance between the names.

- **Token-based**: This approach is based on grouping authors that are, at least sharing one common token into the same block e.g., ”Jack C. Watson" and ”Watson, Karen" will be added to a similar block (Li et al., 2010). The main disadvantage of such approach is that it can result in small-sized blocks if the author names have very rare spelling (non-English words more probable) and if the authors' name spellings are common (Chinese words), or have several initials in the name, then the obtained blocks can have a large number of names.

- **N-gram**: It is similar to the previous token-based approach with the difference in granularity. In fact, instead of finding common tokens in data, it aims to check the existence of common N continuous characters from author names (N = 4 has proven to obtain good results (Gravano et al., 2003)). This approach tends to put many authors on the same block because of its finer granularity compared with previously mentioned approaches. For instance, suppose that we have ”Jane C. Johnson" and ”L. Jane-Donc" in our data set that have the common 4-gram ”Jane" will be put on the same block.

- **Sampling**: This approach suggests to randomly draw a number of samples that are most similar to a string (author name) and group them into the same block. One of the sampling techniques which is fast and accurate is the sampling-based join approximation method proposed by (Gravano et al., 2003). By choosing an appropriate randomization process, this approach can be time-efficient and as a result, it can serve as a good blocking method in disambiguation applications.

One key factor to the success of the application under study is the right choice of the encoder that acts as a measure to filter records and assign them to the blocks. If the filtering criterion is too aggressive, then similar words would end up being distinguished, and correct matches may be incorrectly pruned away. On the other hand, if the blocking method uses a measure that is too lenient, many numbers of inputs (including noisy data) will pass. As a consequence, the size of blocks and the calculation time would increase and the quality of the results will decrease significantly.

2.1.2 Phonetic Encoding Algorithms

A phonetic encoder is a function which indexes a word based on its pronunciation. In fact, such an algorithm allows similarity-based word set comparisons by associating two different words with similar pronunciation with the same code. Phonetic algorithms have been used in a variety of applications such as: normalizing short text messages in under-resourced language (Rafae et al., 2015), finding reasonable variants of names (Pinto et al., 2012), ranking normalization pairs in order to construct a dictionary of lexical variants (Han et al., 2012), and introducing protocols for achieving privacy preserving record linkage (Karakasidis and Verykios, 2009).

Phonetic encoding takes a word as an input, then generates an encoded key, which should be the same for all words that are pronounced similarly while preserving a reasonable amount of fuzziness. These phonetic matching schemes have underlying rule-based mechanisms by which they partition the consonants using phonetic similarity, and then they use a single key to encode each of these sets. Consequently, strings that sound similar come out identical in their encoded forms. For example, the *RefinedSoundex* algorithm groups a set of words: *{Hairs, Hark, Hars, Hayers, Heers, Hiers}* into a single code as: "*H093*".

As a particular optimization step in this research, we set out to evaluate several commonly used phonetic algorithms to understand whether they can reduce the number of comparisons and save time in the disambiguation process. The closest work to our research was done by Zobel and Dart (Zobel and Dart, 1996), in which they performed a comparison of such algorithms. However, the scope of their work was limited to a performance evaluation, whereas we have made a novel contribution by conducting a range of quantitative analyses of the quality of each phonetic encoding algorithm in terms of computation time, ability to assemble similar words under the same code, and ability to give different codes to phonetically very dissimilar words. In addition, our experiments were performed on a realistic dataset as opposed to a hand-edited dataset for (Zobel and Dart, 1996).

2.1.3 String Similarity Metrics

A string similarity metric is used to measure similarity/dissimilarity between two words for comparison or approximate string matching (also known as fuzzy string searching) (Baeza-Yates and

Navarro, 1996). Various applications utilize string similarity metrics to perform tasks such as clustering or matching entity names (Cohen and Richman, 2002) (Branting, 2003), spelling error detection and normalization of micro texts (Xue et al., 2011), extracting structured data from unstructured textual data in order to enrich product ads (Ristoski and Mika, 2016), ontology alignment (Cheatham and Hitzler, 2013), as well as adaptation of such algorithms to perform data cleaning operations such as duplicate detection (Bilenko and Mooney, 2003) (Martins, 2011). These algorithms are used in Information Retrieval (IR) systems to develop applications that enable searching for information in databases, documents or even Linked Data. Such applications are usually evaluated in terms of the ability to retrieve relevant and accurate information regarding a string query. Hence, using an efficient similarity metric is critical in this context.

Although most of the existing works rely on optimizing result accuracy of such metrics, the trade-off between speed of the measurement process (especially when operating on large datasets), the sensitivity of metric with respect to the threshold and discriminatory power is mostly ignored in the literature. With regards to performance, previous works such as (Mitra et al., 2002) attempted to introduce a new feature similarity algorithm that establishes high performance then dealing with real-life data sets. (Cohen et al., 2003) conducted a comparative study of the performance of several algorithms in name-matching tasks. They have shown that a modified version of the *Levenshtein* algorithm outperforms the others by obtaining better results, but they considered only the F1 score as the evaluation metric, and also the computational cost was not considered.

Another work has revealed the importance of similarity measures and their impact in the semantic web by developing a time-efficient approach for the discovery of links between knowledge bases (KBs) on the Linked Data (Ngomo and Auer, 2011). Papadimitriou et al. (Papadimitriou et al., 2010) discussed the possible effect of the sensitivity of similarity functions on web graphs that would associate with the quality of search results in search engines. In fact, they have shown that by having a similarity function that is more sensitive to changes in high-quality vertices in web graphs, results of a search query can be significantly improved. Applications of these metrics target a broader range. Another study showed that choosing an efficient similarity measure can help producing an ontology-based Question Answering (QA) system that supports query disambiguation, knowledge fusion, and ranking mechanisms, to identify the most accurate answers to queries (Lopez et al., 2012).

String similarity algorithms are a key component of disambiguation methods since they are used as the distance measure between features (words or tokens).

2.2 Ensemble Learning

Ensemble learning is the process of combining several different models such as classifiers to create a better model to solve a particular problem. It takes into account that many predictors can perform better than a single one. An ensemble combines many predictors (often a weighted combination of predictors) that might be either the same kind of learner or different types if we do not know what kind of predictor could be best for a given problem. Simple ensembles are unweighted averages of simple learners with the outcome defined as a majority vote. In a weighted situation, we might think that some of the predictors are generally better than the others and we might want to give them more weight than the others. Another possibility is to treat individual predictors as features and combine them using entirely new predictors that take the output of the rest of the predictors and produces better results.

Cascading is a particular case of ensemble learning based on the concatenation of several classifiers, using all information collected from the output of a given classifier as additional information for the next classifier in the cascade. Cascade classifiers are mostly employed in image processing for object detection (Viola and Jones, 2001) and tracking (Okuma et al., 2004).

The cascade structure results in computational efficiency since the average number of elementary features that have to be computed is limited by the early elimination of most records. Nevertheless, accurate classification is obtained because all relevant features are used when a decision is made. Cascade classifiers are trained with sample data, and once trained, they can be applied to new data. Such methods were tested in previous studies and they have shown very high classification rates in near real-time in domains such as: face detection (Degtyarev and Seredin, 2010), and semantic web (Berendt et al., 2002).

Previous works show that the ensemble learning can be used to solve the disambiguation problem. For instance, ensemble learning methods proved to improve the accuracy of word sense disambiguation (Pedersen, 2000) and entity linking (Chen and Ji, 2011). In another work, Speck et. al performed an extensive evaluation of 15 different ensemble methods used to solve the named entity recognition problem (Speck and Ngomo, 2014). Their study reveals that the *ANNs* and the *AdaBoost* methods perform better than the other techniques (such as regression analysis or Bayesian networks). However, only F1 score was used to carry out the evaluation process, and the time-efficiency of such methods was ignored.

2.2.1 Bootstrapping Aggregation (Bagging)

The Bootstrapping Aggregation (Bagging) method was first developed by Breiman (Breiman, 1996), and it is by definition: an ensemble technique that is capable of ”generating multiple versions of a predictive model in parallel and use them to get an aggregated predictor".

The method tries to improve the quality of classification by combining classifications of randomly generated training sets. Given a training set of size N, bagging generates new training sets, each of size N' by sampling from the original training set and with replacement (some observations may be repeated). Multiple M models on different samples (data splits) will be created until a certain iteration number or an acceptance criterion is reached. Then, bagging will average their predictions (in the case of regression) or uses a majority vote (in the case of classification) of these models to predict new data. Figure 2.1 depicts the general idea of the Bagging method.

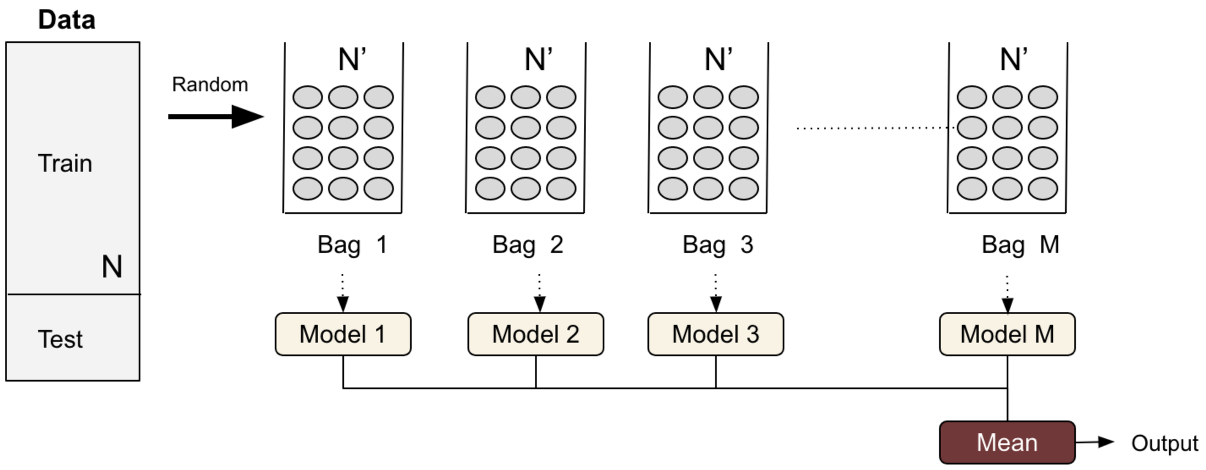


Figure 2.1 Demonstration of the Bagging algorithm

2.2.2 Boosting

Boosting is an ensemble learning technique in machine learning based on the idea of creating a highly accurate prediction rule by combining many relatively weak and inaccurate rules. Each individual predictor tends to be very simple and by combining many of these weak learners that are not able to learn complex functions, we can convert them all into an overall much more complex classifier. The training data is divided into several sample train sets and we use a learner algorithm for each to build a separate classifier and by combining all together we achieve one strong classifier (as shown in 2.2) at the final step that can be used to predict new incoming data with better accuracy than each individual classifiers.

The main idea of this algorithm is to focus new learners on examples that others get wrong (that we call them *hard samples*), and while training learners sequentially, focus later predictions on getting those examples right. In the end, the whole set is combined and many weak learners are converted into a complex predictor. Boosting methods were used for regression (including generalized re-

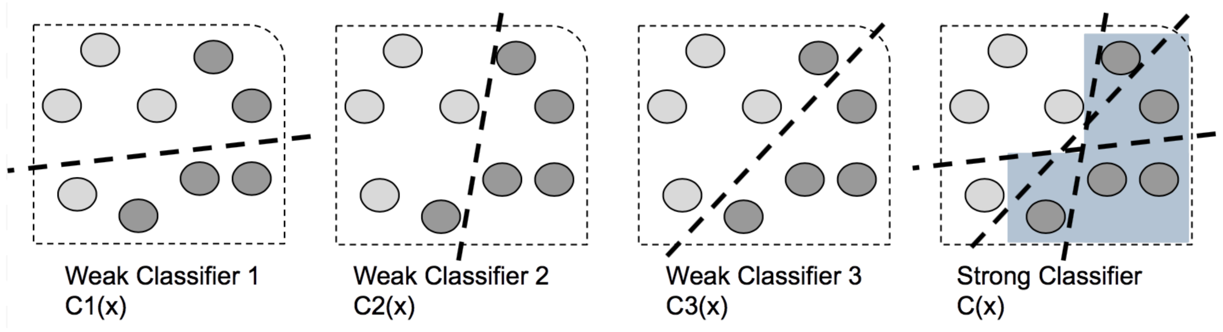


Figure 2.2 Demonstration of the Boosting algorithm

gression) (Bühlmann and Yu, 2003), for density estimation (Ridgeway, 2002) and for multivariate analysis (Friedman et al., 2000).

The first realization of boosting that saw great success in the application was Adaptive Boosting or AdaBoost (Freund et al., 1999) for short. AdaBoost is different than the Boosting in a sense that it takes the misclassified data and adds them to the next sample of the training set (see Figure 2.3). It also puts more weight on wrongly classified instances. Weighted error of each new classifier tends to increase as a function of boosting iterations.

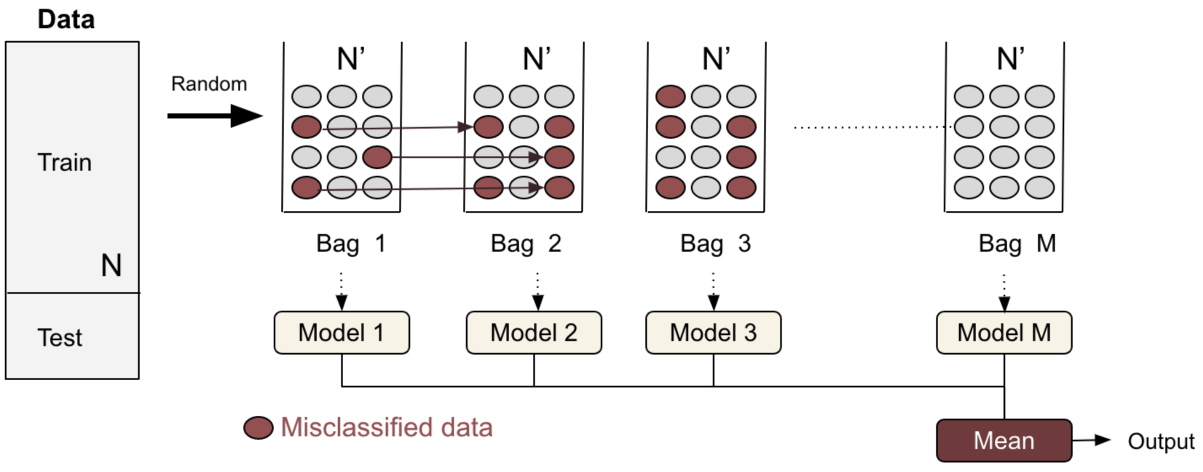


Figure 2.3 Demonstration of the AdaBoost algorithm

The AdaBoost method was used in many domains such as computer vision for text detection (Lee et al., 2011) (Liu, 2010), face detection (Yang et al., 2010) and cascade classification when combined with SVM (Cheng and Jhan, 2011).

Although AdaBoost provides good solutions to a variety of classification problems, it needs adaptations to tackle the over-fitting problem. Over-fitting is a problem in machine learning that happens when a model learns the random fluctuations and noise in the training data to the extent that it has a negative impact on the performance of the model on new data. A common approach to avoid the over-fitting when training a model is to use validation or cross-validation to compare the predictive accuracy of test data (Rätsch et al., 1998).

2.3 Particle Swarm Optimization

Particle Swarm Optimization is a stochastic optimization method of continuous nonlinear functions first introduced by Kennedy and Eberhart (Eberhart and Kennedy, 1995). Originally, this method was inspired by social behavior of bird flocking, and it was developed to work on the social adaptation of knowledge (Kennedy, 1997) with the aim of simulating the adaptive sharing of representations among social collaborators. The algorithm represents a systematic approach to explore a problem space using a population of individuals. The progress and success of these individuals influence their searches and those of their peers. The main concept is to initialize with having a population that we call *swarm* of candidate solutions to be called *particles*. The particles move in a search space based on the knowledge about their own best-known position as well as the entire swarm's best-known position. Each particle's movement is derived by two forces: one pulling it with the random magnitude to the fittest location so far reached by the particle (Pbest), and another one pulling it with the random magnitude to the best location obtained by any of the particle's neighbors in the swarm (Gbest). Figure 2.4 shows a flow chart of a typical standard PSO algorithm.

The velocity of each particle in the swarm is updated using the following standard equation (although there exists variety of variants proposed in literature):

$$v_i(t+1) = \omega v_i(t) + c_1 r_1 \left[\hat{x}_i(t) - x_i(t) \right] + c_2 r_2 [g(t) - x_i(t)]$$

In above equation *i* is the index of each particle, therefore, $v_i(t)$ is the velocity of particle *i* at time *t* and $x_i(t)$ is representing the position of particle at the same time. The parameters ω, c_1 and c_2 are user-supplied coefficients with conditions as such: $0 \leqslant \omega \leqslant 1.2$, $0 \leqslant c_1 \leqslant 2$ and $0 \leqslant c_2 \leqslant 2$. Values r_1 and r_2 are random values generated for each update of the velocity such that $0 \leqslant r_1 \leqslant 1$ and $0 \leqslant r_2 \leqslant 1$. In addition, $\hat{x}_i(t)$ is representing the best candidate solution for particle *i* at time *t*, and $g(t)$ is the swarm's global best candidate solution.

Basically, each three terms of this equation have a certain role in PSO algorithm. $\omega v_i(t)$ is called *inertia component* and it is used to keep the particle moving in the same direction originally headed. The value of ω is called *inertial coefficient* (usually between 0.8 and 1.2) and it can either dampen the particle's inertia or accelerate the particle in its original direction. The second term $c_1 r_1$ [$\hat{x}_i(t)$ - $x_i(t)$] is called *cognitive component* and it is presenting the particle's memory, making it

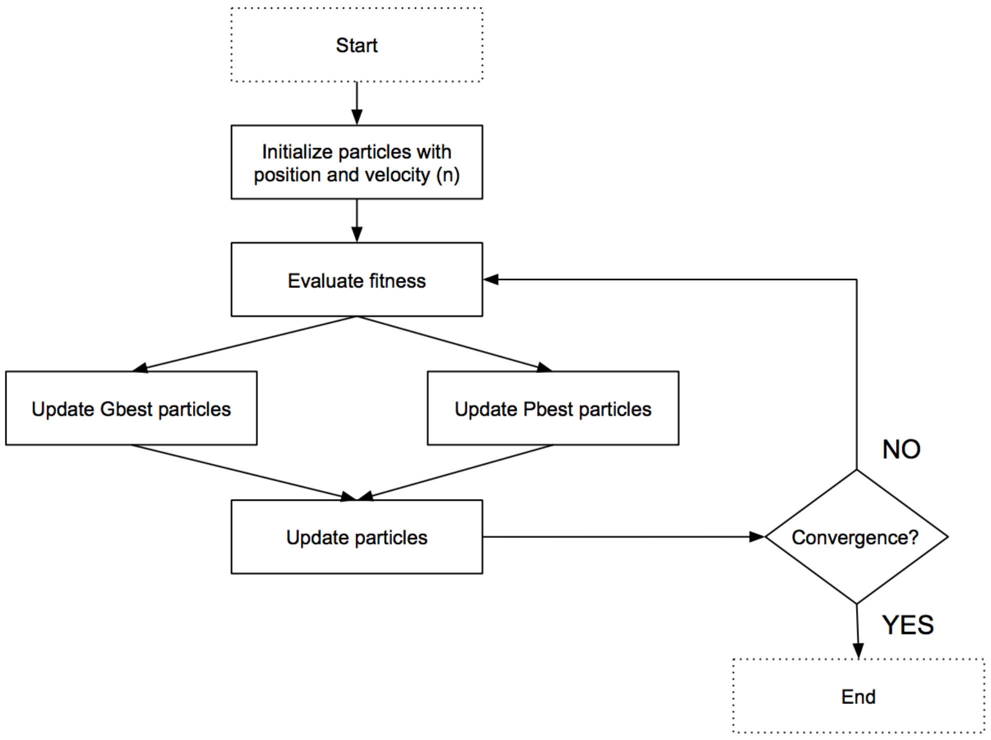


Figure 2.4 Particle Swarm Optimization algorithm flow chart

to return to the regions of the search space that high individual fitness was observed. Finally, the last term $c_2r_2[g(t)$ - $x_i(t)]$ is called *social component* that guides the particle to move to the best region the swarm has found so far. All the mentioned terms have a stochastic influence on the velocity update. To avoid particles from moving too far beyond the search space, another values of *Maximum Momentum* and *Minimum Momentum* are used.

When the velocity for each particle is computed, the position of each particle is updated by applying the new velocity to the particle's previous position:

$$x_i(t+1) = x_i(t) + v_i(t+1)$$

Eventually, particles tend to move toward the optimized solution. The algorithm is repeated through iteration numbers, and it will stop when reaching a stop condition, which can be either maximum number of iterations or a cost-effective solution achievement.

PSO has been used in several areas including human motion tracking and pose estimation (John et al., 2010), scheduling (Liu et al., 2010) Pandey et al. (2010), image processing (Broilo and De Natale, 2010) and feature selection in classification (Xue et al., 2013).

Generally, PSO can be applied to any optimization problem that deals with maximizing/minimizing an objective function. While all the gradient-based methods can do this, they require the problem space to be convex, smooth, and continuous which is a big barrier when solving real world problems. By contrast, PSO can operate on rough spaces and still produce reasonable solution. In addition, PSO has fewer parameters to adjust, when compared to its competitor genetic algorithms. Hence, it became a source of attraction for many problems in many research domains. However, the performance of the particle swarm optimizer depends on choosing the best settings for its parameters, hence, it requires guidelines for selecting the most efficient ones.

2.4 Set Cover Optimization

Suppose to have a set of n elements $U = [e_1, e_2, \dots e_n]$ and a collection $S = \{S_1, S_2, \dots S_m\}$ of m non-empty subsets of U, where $\cup_{i=1}^{m} S_i = U$ and each S_i is associated with a positive cost $c(S_i) \geq 0$. The minimum set covering optimization problem (Gens and Levner, 1980) is to find a subset $A \subseteq S$ such that $\sum_{S_i \in A} c(S_i)$ is minimized subject to $\cup_{S \in A} S = U$.

The set cover is a NP-complete problem (no fast solution is known), thus we are looking for an approximate solution rather than optimal one.

We consider the problem of ordering WCs in the disambiguation algorithm as a set cover problem in which by using a cost function, we try to find a minimum set of WCs with less classification error and more time efficiency.

To solve the minimum set cover problem, previous studies suggested several approximations and heuristics algorithms (Akhter, 2015). For instance, (Emek and Rosén, 2016) studied the set cover problem under the semi-streaming model and proposed an approximation algorithm that obtained with a computational complexity of $O(\sqrt{n})$. In other efforts, Guanghui Lan et al. introduced Meta-RaPS (meta-heuristic for randomized priority search) solution for this problem (Lan et al., 2007), and Fabrizio Grandoni et al. proposed an algorithm based on the interleaving of standard greedy algorithm with the ability to select the min-cost set which covers at least one uncovered element (Grandoni et al., 2008).

CHAPTER 3 FAST AUTHOR NAME DISAMBIGUATION ALGORITHM

In this section, we describe an algorithm that has been used to run the disambiguation task. This algorithm is provided as an application that generates results of disambiguated data given a bibliographic data set in a reasonable time-efficient manner (Voorons et al., 2017). Our research is conducted over this application, for this reason, we explain its steps to provide background information for our methodology.

3.1 Overview

The context of this application is to evaluate the scientific activity of individuals and institutions. This type of assessment is beneficial to funding agencies and governments by enabling considerations on the effectiveness of funding programs and planning research policies. For this purpose, bibliographic databases, which are very enriched source of data can be used. Generally, such databases hold very large amounts of information about researches and publications. However, they face certain difficulties: First of all, the size of the bibliographic databases are too large that slows down the process. In addition, there are many problems of ambiguities which have multiple reasons: natural reasons and reasons related to compilation errors that can be summarized as:

- A single individual may have published under multiple names, that includes: a) orthographic and spelling variants b) spelling errors, c) name changes over time as may occur with marriage, religious conversion or gender re-assignment, and d) the use of pen names.
- Common names may comprise several thousand individuals (as an example, we could have millions of Mr. ”Wang" since its a very common name in China).
- Meta-data are either not present or incomplete. As an example, some publishers and bibliographic databases did not record authors’ first names, their geographical locations, or identifying information such as their degrees or their positions.
- Duplicate records and swapped fields usually exist.
- A large percentage of scholarly articles are multi-authored, and in fact, represent multi-disciplinary and multi-institutional efforts, which results in ambiguous data.

The accuracy of the results obtained by performing assessments using bibliographic databases relies on a process in which each publication record must be associated with the correct individual and/or institution. This process is called entity disambiguation.

In this section, we explain a new methodology for entity disambiguation that was used in application under study. This proposed method was developed to address the problem of entity disambiguation while preserving the efficiency, versatility and handling the missing fields. This method was inspired from the AdaBoost cascade classifiers (Viola and Jones, 2001), which has been used successfully in image processing for object or face detection in large datasets. The internal structure of this cascade arrangement included the boolean "OR" operator instead of "AND", because it was a better choice in terms of computational efficiency. This choice also enabled to handle missing fields by either jump to the next classifier of the same step when the missing data was not essential, or to exit from the comparison process if it was (e.g., missing surname) and thus create a new author. The problem of transitivity violation was resolved by the use of an extra classification stage aiming at result consolidation.

Additionally, unlike other approaches (e.g. artificial neural networks (ANN)), the structure of the proposed method is easily understandable because it was inferred by the disambiguation techniques used by bibliography experts. Nonetheless, the cascade structure of the proposed method was selected by hand (using trials and errors) and it lacks an automated approach for determination.

In the following sections, we describe the elements of the aforementioned algorithm to provide a general understanding of each stage it contains to run the disambiguation task:

3.2 Pre-processing

This stage included a set of standard automatic information retrieval routines that are commonly applied in natural language processing (NLP) methods that helps to increase the effectiveness of data processing:

- Select only the most relevant information fields based on predefined use-cases depending on the entity to disambiguate.
- Remove duplicate records and redundant words.
- Filter “stop words” and non-discriminants words e.g., ”university" which is common to many affiliations.
- Create bags of words (BOW) for the reference, address, and keyword fields.
- Replace keywords by identification numbers as processing integers are more efficient than strings.

- Normalize fields by transforming information to the English alphabet, removing non-character symbols and unnecessary numbers, standardizing regional or cultural specificities e.g., removal of ”née" before woman maiden names.
- Store double Metaphone encoding of the surname in a new field to speed up the classification algorithm by the use of a blocking method.
- Calculate statistics on the occurrences of each word of each field for later use during the classification process.

This stage, in summary, significantly reduced the size of the dataset and increased the relevance of the retained information.

3.3 Blocking

First, in order to reduce the number of pairwise comparisons, a blocking method based on the surnames of the authors was used. Using a phonetic encoding algorithm, each author was compared only to authors sharing the same phonetic code.

3.4 Classification Approach

The pre-processing stage helped to reduce the amount of memory and execution time by generating smaller amounts of data. However, the data still contained noisy data and ambiguities, therefore, the classification stage took place.

The global architecture of such classification approach was made of N cascade classifiers (CC) that is depicted in Figure 3.1. In order to achieve high precision rate, the first disambiguation block contained very strict rules and discriminative classification features such as *Email Address* or *Employee Identification Codes* when available. Despite resulting in a set of very reliable authors, this step produced over-segmentation. That is the reason why the second disambiguation block was used to take care of unclassified records with more relaxed rules. Basically, this second block was responsible to perform the remaining task of disambiguation on the rest of records that were not classified with strict rules.

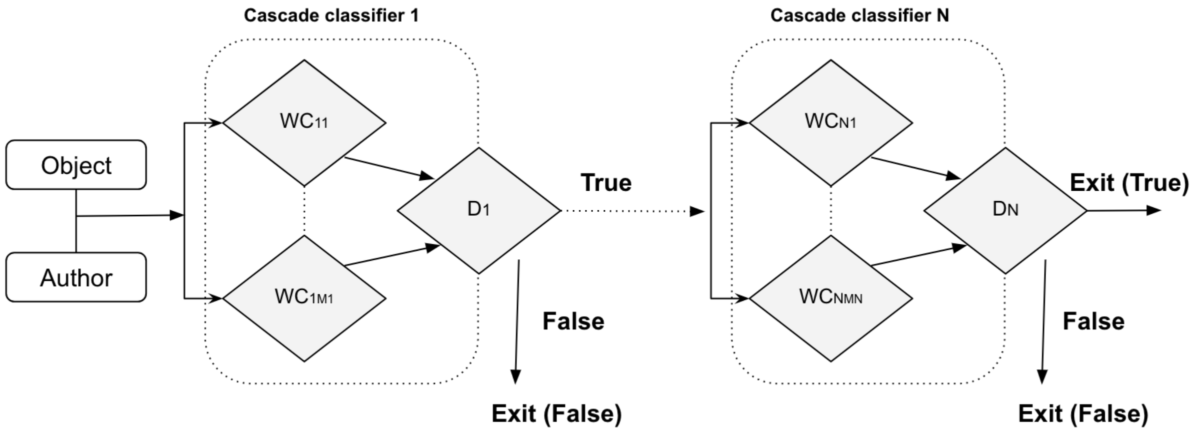


Figure 3.1 Cascade approach (Voorons et al., 2017)

As it is schematically shown in Figure 3.2, the disambiguation method included four aggregators. In processing chain, an entity aggregator (EA) was put in place for aggregation of records into author entities, followed by a recursive aggregator (RA), which addressed transitivity violation by recursive aggregation of authors until no merge occurs. The cascade structure has a high impact on the performance efficiency as the average number of elementary features to be computed is limited due to the fact that most objects are eliminated early.

Moreover, the selection of the data and the formation of the EA and RA depend on the training.

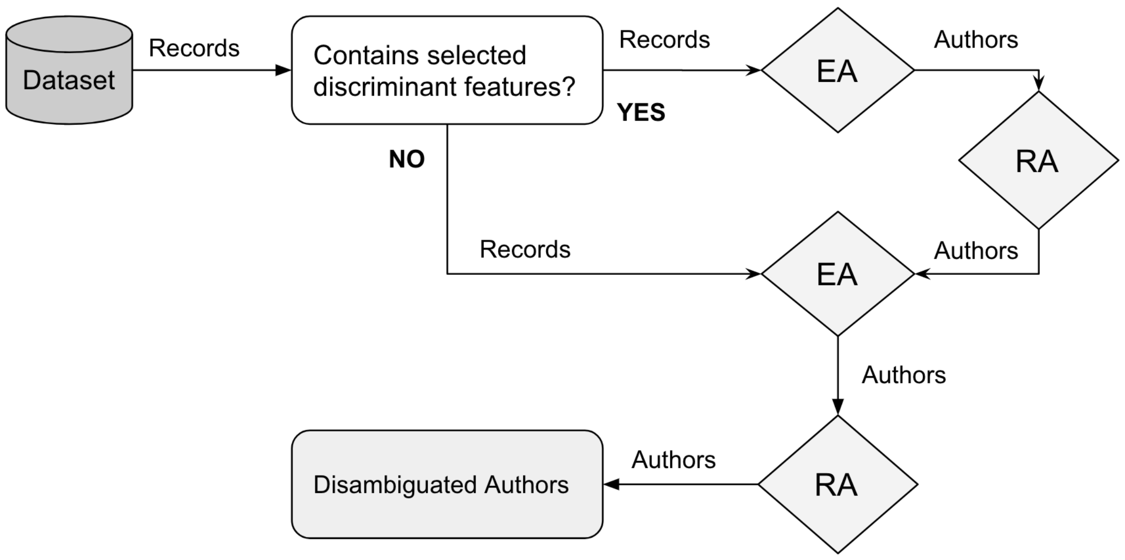


Figure 3.2 Process of entity aggregators and recursive aggregators (Voorons et al., 2017)

Table 3.1 Characteristics of different types of weak classifiers

Type	Operations	Decision	Advantages	Disadvantages	Usage
Set Theory	Equality, intersection or cardinality of the intersection over two BOWs	Boolean or Threshold-based	Simple and fast to compute	Sensitive to non-discriminant data and typographical errors	Keyword, Co-author and reference BOWs
Similarity distances	Measuring similarity using string similarity algorithms	Threshold-based	Robust to typographical errors	Time consuming when applied to BOWs, and sensitive to the metric	Address BOWs
Specialized	Based on specific characteristics of the data	Threshold-based	Time efficient	Problem of over-fitting regional or uncommon particularities	Surname and Forename

Each cascade classifier was made of multiple weak classifiers (WC). These WCs were designed to make decisions based on simple rules (similarity measures of single classification features). Three different types of weak classifiers have been used in this classification approach which is summarized in Table 3.1.

Table 3.2 Structure of the first disambiguation block - WoS subset (large size)

Entity Aggregator (EA)

Classifier	Classifier Type	WC Field	Blocking	Hit	Threshold
1	Set (contains any)	E-mail	True	False	N/A
2	Similarity distance	Surname	–	–	0.85
3	Specialized (to forenames)	Forename	–	–	0.85
4	Set (contains no)	Article Ids	True	False	–

Recursive Aggregator (RA)

Compound Classifier	Classifier Type	WC Field	Blocking	Hit	Threshold
1	Similarity distance	Surname	–	–	0.98
2	Specialized (to forenames)	Forename	–	–	0.98
3	Set (contains no)	Article Ids	True	False	–
4	Set (% of intersection)	Subfields	False	False	0.3
	Set (contains any)	BOW reference Ids	False	False	–
	Set (contains any)	BOW co-authors	False	False	–
5	Set (% of intersection)	BOW co-authors	False	False	0.2
	Set (% of intersection)	BOW reference Ids	False	False	0.5
6	Set (contains any)	BOW reference Ids & Article Id	True	False	–
	Set (contains any)	Article Id & BOW reference Ids	True	False	–

Table 3.3 Structure of the second disambiguation block - WoS subset (large size)

Entity Aggregator (EA)

Classifier	Classifier Type	WC Field	Blocking	Hit	Threshold
1	Similarity distance	Surname	–	–	0.95
2	Specialized (to forenames)	Forename	–	–	0.95
3	Set (contains no)	Articles Ids	True	False	–
4	Set (contains any)	BOW reference Ids	False	False	–
	Set (contains any)	BOW keywords	False	False	–
	Set (contains any)	BOW co-authors	False	False	–
	Set (% of intersection)	BOW addresses	False	False	0.5
5	Set (% of intersection)	Fields	False	False	0.7
	Set (% of intersection)	Journal Id	False	False	0.1
	Set (% of intersection)	Subfields	False	False	0.3
	Set (% of intersection)	BOW keywords	False	False	0.7
6	Set (% of intersection)	BOW reference Ids	True	False	0.3
	Specialized (naive Bayesian)	BOW co-authors	True	False	0.3
	Set (contains any)	BOW reference Ids & Article Id	True	False	–
	Set (contains any)	Article Id & BOW reference Ids	True	False	–

Recursive Aggregator (RA)

Classifier	Classifier Type	WC Field	Blocking	Hit	Threshold
1	Similarity distance	Surname	–	–	0.95
2	Specialized (to forenames)	Forename	True	False	0.95
3	Set (contains no)	Article Ids	True	False	–
4	Set (contains any)	Fields	False	False	–
	Set (% of intersection)	BOW address	False	False	0.8
	Set (contains any)	BOW co-authors	False	False	–
	Set (contains any)	BOW reference Ids	False	False	–
5	Set (% of intersection)	BOW keywords	False	False	0.6
	Set (% of intersection)	Subfields	False	False	0.6
	Set (% of intersection)	Journal Id	False	False	0.4
6	Set (contains any)	Article Id & BOW reference Ids	True	False	–
	Set (contains any)	BOW reference Ids & Article Id	True	False	–
	Set (% of intersection)	BOW reference Ids	True	False	0.4
	Set (% of intersection)	BOW co-authors	True	False	0.3

Table 3.2 and Table 3.3 provide information with regards to each cascade classifier that contains multiple WCs when operating on the WoS dataset. Each WC operates on a certain field, and if applicable, it has a threshold value (in a range between 0 and 1), by which it computes a decision. For example, the surname classifier (similarity distance) addresses name inversions if one has several, and the forename classifier (specialized to forenames) handles initials, diminutives and middle names. In other cases, each WC is a simple classifier based on the thresholding of the overlap cardinality of two sets of numbers (e.g. set contains no/any classifier or set percentage of intersection). *Set Contains Any* is a simple classifier built on a boolean operator, which returns true if the two feature sets overlap, and *Set Contains No* is another simple classifier built on a boolean operator, which returns true if none of the features of each set is included in the other.

Table 3.4 and Table 3.5 show the cascade structure for the small-size SCOPUS dataset that has been used in the evaluation process. The structure is very similar to the one used for WoS, however, modifications were applied to thresholds and some WCs were rearranged in certain steps.

Table 3.4 Structure of the first disambiguation block - SCOPUS subset (small size)

Entity Aggregator (EA)

Classifier	Classifier Type	WC Field	Blocking	Hit	Threshold
1	Set (contains any)	E-mail	True	False	N/A
2	Similarity distance	Surname	–	–	0.85
3	Specialized (to forenames)	Forename	–	–	0.85
4	Set (contains no)	Article Ids	True	False	–

Recursive Aggregator (RA)

Compound Classifier	Classifier Type	WC Field	Blocking	Hit	Threshold
1	Similarity distance	Surname	–	–	0.98
2	Specialized (to forenames)	Forename	–	–	0.98
3	Set (contains no)	Article Ids	True	False	–
4	Set (% of intersection)	Subfields	False	False	0.3
	Set (contains any)	BOW reference Ids	False	False	–
	Set (contains any)	BOW co-authors	False	False	–
5	Set (% of intersection)	BOW co-authors	False	False	0.2
	Set (% of intersection)	BOW reference Ids	False	False	0.5
6	Set (contains any)	BOW reference Ids & Article Id	True	False	–
	Set (contains any)	Article Id & BOW reference Ids	True	False	–

Table 3.5 Structure of the second disambiguation block - SCOPUS subset (small size)

Entity Aggregator (EA)

Classifier	Classifier Type	WC Field	Blocking	Hit	Threshold
1	Similarity distance	Surname	–	–	0.95
2	Specialized (to forenames)	Forename	–	–	0.90
3	Set (contains no)	Articles Ids	True	False	–
4	Set (contains any)	BOW reference Ids	False	False	–
	Set (contains any)	BOW keywords	False	False	–
	Set (contains any)	BOW co-authors	False	False	–
	Set (% of intersection)	BOW addresses	False	False	0.55
5	Set (% of intersection)	Fields	False	False	0.3
	Set (% of intersection)	Journal Id	False	False	0.1
	Set (% of intersection)	Subfields	False	False	0.2
6	Set (% of intersection)	BOW keywords	False	False	0.33
	Set (% of intersection)	BOW reference Ids	True	False	0.33
	Specialized (naive Bayesian)	BOW co-authors	True	False	0.1
	Set (contains any)	BOW reference Ids & Article Id	True	False	–
	Set (% of intersection)	BOW addresses	False	False	0.33

Recursive Aggregator (RA)

Classifier	Classifier Type	WC Field	Blocking	Hit	Threshold
1	Similarity distance	Surname	–	–	0.88
2	Specialized (to forenames)	Forename	True	False	0.95
3	Set (contains no)	Article Ids	True	False	–
4	Set (contains any)	Fields	False	False	–
	Set (contains any)	BOW co-authors	False	False	–
	Set (contains any)	BOW reference Ids	False	False	–
5	Set (% of intersection)	BOW keywords	False	False	0.2
	Set (% of intersection)	Subfields	False	False	0.1
	Set (% of intersection)	Journal Id	False	False	0.1
	Set (% of intersection)	BOW addresses	False	False	0.2
6	Set (contains any)	Article Id & BOW reference Ids	True	False	–
	Set (contains any)	BOW reference Ids & Article Id	True	False	–
	Set (% of intersection)	BOW reference Ids	True	False	0.1
	Set (% of intersection)	BOW co-authors	True	False	0.1

As illustrated in Figure 3.1, each cascade classifier was followed by a decision operator D_i. The different types of decisions can be itemized as indicated below:

- **PASS**: The decision is *true*, meaning that the simple classifier judges that the distance between pairwise Entities or records is small enough to consider that the two entities or records belong to one unique entity.
- **HIT**: The decision is *true* and leads to a perfect match, meaning that the comparison of pairwise Entities or records is performed on a key feature, which decides with high confidence that the two Entities or records must belong to the same entity.
- **BLOCK**: The decision is *false*, meaning that the simple classifier judges that the distance between pairwise Entities or records is big enough to consider that the two entities or records belong to two separate entities.
- **MISSING_BLOCK**: The decision is *false* due to missing data, meaning that some data are missing and so the comparison of pairwise Entities or records can not be performed, and it is not possible to judge whether the two entities or records belong to one unique entity or not.
- **MISSING_PASS**: The decision is *false* due to missing data but this is not crippling, meaning that we can not achieve the comparison of pairwise Entities or records and leave further analysis to other simple classifiers.

Furthermore, as depicted in Figure 3.3, pairwise classification is performed in an agglomerative manner: when a record is found to belong to an author, the record features are merged into the corresponding author features, thereby increasing the knowledge available about the author will increase the quality of results. In addition, objects that are not associated with any of the authors of the existing database are added as new authors in the database.

3.5 Cascade Structure Configuration

In the application under study, a heuristic approach was used to find out the structure and the thresholds. The objective was to achieve effectiveness, insensitivity to missing data, and to filter out irrelevant records to increase processing efficiency. This structure was determined with trials and errors, but this was a very difficult task since the structure and thresholds interact with each other and cannot be specified separately or sequentially. Therefore, the structure was determined mostly based on what disambiguation experts do manually and through extensive consultation that was summarized under the form of guidelines:

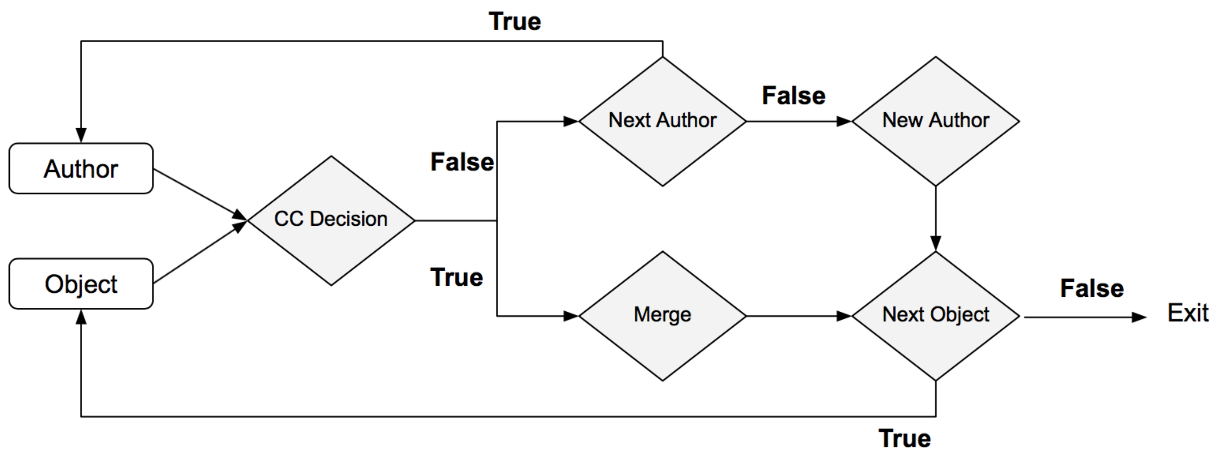


Figure 3.3 Detailed design of an aggregator (Voorons et al., 2017)

1. Assess the discriminating power of the various fields
2. Emphasize on research domain information since authors rarely change their research domains
3. Tend to use low computational complexity WCs in early steps
4. Pay attention to the frequency of missing data in some key fields by grouping the WCs assigned to evaluate them in CC
5. Associate appropriate thresholds in accordance with the variability of some fields (such as address that can vary in typography)

According to these guidelines, the configuration of the cascade structure is based on:

- Associating the first CC to highly discriminative fields such as surnames, forenames and email addresses. However, it is followed by a control WC which prevents merging the two authors if they share an article ID, therefore avoiding merging two homonyms authors.
- Keeping the structure of WCs simple but efficient in the early stages (in each aggregator, the next stage aims to efficiently filter out homonyms that are not related based on their research fields).
- Ensuring the overall efficiency of each aggregator by pushing intensive computations to the last stages, since the last stages mostly include WCs based on the citation analysis measures

such as self-citing and bibliographic coupling[1].

- Grouping the WCs that operate on frequently missing data (e.g, keywords, address, references and co-authors) together to take the advantage of OR boolean decision operator.
- Setting thresholds heuristically depending on strictly enforced rules (corresponding to high-value thresholds) and discriminating power and variability of the various fields. For example, thresholds for *Surname* and Forename were set to 0.98 while it has been a lower value of 0.5 and 0.2 for *BOW reference Ids* and *BOW co-authors* respectively.

3.6 Evaluation

Beforehand, an experiment was performed on two datasets extracted from Web of Science[2] and SCOPUS[3], in order to assess the quality of the results obtained by the discussed method in the application under study. The indicators used to perform such evaluation included:

- Standard quantitative quality indicators:
  - **Precision rate:** The ratio of the records assigned to an author that actually belong to him.
  - **Recall rate:** The ratio of the total number of database records associated with one author that has been correctly assigned to him.
  - **F1 score:** Product of the precision by the recall divided by their sum.
- Manual disambiguation results provided by experts (as the gold standard).
- A standard supervised reference method (Huang et al., 2006) combining Support Vector Machine (SVM) and Density-Based Spatial Clustering of Applications with Noise (DBSCAN). The reference method (SVM-DBSCAN) operates a blocking method based on author names; then, using the SVM method enables to constructs clusters based on multiple pair-wise distances, and then using the DBSCAN algorithm addresses the transitivity violation problem.

[1]"Bibliographic coupling provides a similarity relationship between scientific artifacts. It happens when two documents reference a common third document in their bibliographies, therefore indicating that a probability exists that the two documents are concerned with a related subject matter." (Weinberg, 1974)

[2]www.webofknowledge.com

[3]www.elsevier.com/solutions/scopus

3.6.1 Datasets

The dataset used for evaluation consists of large number of records of the WoS database which were automatically disambiguated using author information from the Orcid [4] dataset. It has to be mentioned that Orcid has a web site that provides a unique digital identifier to every researcher registered on it. Author manually enter their bibliographic information on the Orcid web site, therefore, it ensures a high confidence in the quality of this dataset. Pairwise record comparisons were performed based on the title, author names, page numbers, journal name, DOI and publication year in order to extract records of the WoS using the Orcid dataset. This process resulted in generating a fully disambiguated dataset of more than 2.89 million records corresponding to 132899 distinct authors. This datasets dataset contained 111120 distinct surnames and forename initials, meaning that 21779 authors had at least one homonym, which resulted in some very ambiguous cases (this figure is conservative as it does not account for misspelled names, name variations and multi part names).

Another dataset that has been used in the evaluation process was composed of 185,253 records condensed from the SCOPUS database, holding the information that belonged to 2398 distinct authors working in a variety of Canadian universities and companies.

In addition, in all experiments, a gold standard was created manually by bibliometric experts from a private company and was used as a base-line to assess the quality of the results and to perform the quantitative analysis.

3.6.2 Results

The results of the cascade approach and the SVM-DBSCAN methods on WoS subset (large size) dataset are displayed in Table 3.6.

Table 3.6 Results of the manual cascade approach (WoS) (Voorons et al., 2017)

	SVM-DBSCAN	**Manually configured cascade approach**
Precision (%)	99.47	99.23
Recall (%)	73.88	83.21
F1 Score	0.848	0.905
Processing time (s)	1,817,819	2,724

The results show that despite the size of the dataset, the general quality of the disambiguation can be regarded as satisfactory with an advantage to the proposed approach that has a better recall rate.

[4]http://www.orcid.org

The better recall is undoubtedly linked to the recurring steps occurring in the recursive aggregators (RA) of the method.

With regards to the processing time, it took about 45 minutes for the disambiguation of the whole dataset with the proposed cascade approach, while it took about 21 days with SVM-DBSCAN. The large difference in processing time can be explained by the fact that, in the cascade approach, we have the pre-processing stage and the blocking stage which reduced the number of pair-wise comparisons, thereby improving the time-efficiency of the application.

Furthermore, the cascade approach was tested on the SCOPUS database (small size) and the results are shown in Table 3.7. The performance was assessed under three disambiguation strategies: strategy C_1 which favors precision over recall, strategy C_2 which balances precision and recall while strategy C_3 favors recall over precision. The cascade structure was kept similar for all strategies, whereas the thresholds were adjusted for each case.

It can be seen that C_1 obtained a prefect precision as WC parameters were chosen as strict decision rules. However, a low recall rate was obtained due to an increase in transitivity violations because when classification rules are strict, records are more likely to be classified as unique authors. C_3 produced the best recall rate and C_2, as expected, appeared as a trade-off between C_1 and C_3.

Finally, regarding time efficiency, it can be underlined that the proposed method was more time-efficient than the SVM-DBSCAN method since it performed the disambiguation of the dataset in less than 30 seconds for the three strategies, as opposed to 30 minutes for SVM-DBSCAN. This advantage is clearly of critical importance for disambiguation of large, real-world datasets.

Table 3.7 Results of the manual cascade approach (SCOPUS) (Voorons et al., 2017)

		Manually configured cascade approach		
	SVM-DBSCAN	**C1**	**C2**	**C3**
Precision (%)	99.260	100.000	99.991	99.967
Recall (%)	96.893	88.578	95.883	97.436
F1 Score	0.980	0.939	0.978	0.986
Processing time (s)	1782	26	23	28

3.6.3 Summary of the Results

To conclude, the cascade structure of this algorithm resulted in a high computational efficiency and helps to scale up to large datasets, whereas the accuracy of the results still depends on the performance of WCs. Considering the fact that the blocking method is a crucial component of this application, its influence was not addressed in the method of disambiguation. Hence, it must be optimized by the choice of appropriate phonetic encoding algorithm metric. The other important part of the application is the choice of string similarity metric as it takes place in many weak classifiers. In particular, we are interested in choosing a similarity measure that compromises between accuracy and speed. In addition, another drawback of the application under study is the manual arrangement of WCs and their thresholds.

To tackle the problems mentioned above, we decided to dedicate our study to examine the appropriateness of the phonetic encoding algorithm and similarity distance measure in terms of their impacts on the results, then develop a learning process to automatically adjust the arrangement of WCs and their thresholds.

CHAPTER 4 METHODOLOGY AND DESIGN

4.1 Experimental Setup

The application under study is not yet optimized regarding the choice of appropriate phonetic encoding algorithms for the blocking method and string similarity measures for the classification operators. Additionally, it lacks adaptability because of not being automated with regards to the selection of the best ordering and thresholds of its classifiers.

To work on the problems mentioned above, three major steps have been taken:

- Comparing different phonetic encoding algorithms and choose the suitable one. The purpose of this step is to optimize the blocking method, and the outcome is a set of recommendations and guidelines to help choose the best one for our problem.
- Comparing different string similarity metrics and figure out which one fits best to our problem. The purpose of this step is to optimize the decision-making process inside the classifiers, and the outcome is recommendations and guidelines to choose the best one that fits our problem.
- Develop a training methodology to automatically construct the configuration of classifiers. The final product of this step is an automated framework with the ability to generate a cascade structure along with the corresponding thresholds of the classifiers adaptable to every type and format of input data that can also help identify the importance of each feature of the classifiers in overall disambiguation process.

The following sections explain in details our methodology for each step explained above.

4.1.1 Step 1: Comparing Phonetic Encoding Algorithms

Six commonly used phonetic algorithms were selected; since, to the best of our knowledge, quantitative performance evaluation and comparison of such algorithms has not been reported in the literature, specific quantitative metrics were developed in order to rate different aspects. Comparisons between these phonetic encoding algorithms were performed on real data extracted from SCOPUS, and we were able to draw conclusions about the qualities and weaknesses of each encoder in the perspective of an application as a blocking method for author names. This allows us to point out trade-offs between these phonetic algorithms.

Selection of Phonetic Encoding Algorithms

We selected six commonly used phonetic encoding algorithms, which, for the sake of generality, are suitable for working with English and foreign names. Source code and implementation of these algorithms were taken from the Apache Commons Codec [1].

RefinedSoundex: This phonetic algorithm is a variant of *Soundex* (NAT, 2007), which was proposed in the 1910s by Robert Russell. Soundex matches words to numerical indexes by partitioning consonants in groups with ordinal numbers and compiling them to the resulting value. Soundex reserves the first letter, then it matches the subsequent letters to digits by certain rules. Vowels and some consonants are ignored, and adjacent letters or letters separated by "H" or "W" that are in the same group are treated as one letter. The result, however, is truncated to four characters. Refinded-Soundex is an improved version of Soundex, optimized for spell checking. In this modified version of Soundex, the letters are divided into more groups, and there are no special cases with "H" and "W" since they are simply ignored by the algorithm. Another major difference is that the length of the result is not truncated, so the code does not present a fixed length.

Caverphone2: It is the second generation of the *Caverphone* algorithm that was developed by David Hood (Hood, 2002) as part a project to match the data in the old and new electoral lists in New Zealand. It was mainly designed (by the Caversham Project at the University of Otago) to be implemented on New Zealand pronunciations, but it works well with foreign names too. Generally, this algorithm applies a series of replacement rules to strings (e.g. convert to lowercase, remove anything not A-Z, replace all vowel at the word beginning with "A", etc.) until getting the first six characters as the code.

Metaphone: This algorithm was developed (Philips, 1990) to index words by their English pronunciation. The intention to implement this algorithm was to provide more accurate encoding, therefore, it preserves more information since the letters are not divided into groups. The final code is a set of characters. Metaphone allows specifying the maximum length of the code (up to 12 characters) so that it can be focused on the first few syllables or words of complex data rather than the entire raw data. Thus, it maintains the control over the sensitivity of the phonetic similarity. *Metaphone* codes are particularly useful in a situation where spelling discrepancies might occur in words that sound the same.

[1]http://commons.apache.org

DoubleMetaphone: It is a phonetic encoder algorithm for indexing strings by their pronunciation. *DoubleMetaphone* is the second generation of *Metaphone* (Philips, 1990) with several design improvements over the original *Metaphone* algorithm. This algorithm codes English words phonetically and reduces them to a combination of twelve consonant sounds. It has a large number of different rules that consider the origin of words, focusing on Italian, Eastern European, Chinese and others. This encoder was used as a measure in the blocking stage of the application under study.

ColognePhonetics: This algorithm assigns a sequence of digits to words, *i.e.* the phonetic code. Similar-sounding words can be identified by having the same phonetic code. Generally, this algorithm implements the "Kölner Phonetic" (Cologne Phonetic) algorithm proposed by Hans Joachim Postel (Postel, 1969) and it was used as a search function based on phonetics. This algorithm is related to *Soundex* but is optimized to match the German language. An important difference is that unlike the Soundex code, the length of the codes generated by the *ColognePhonetics* is not limited, and unlike the RefinedSoundex, it can adapt to the German alphabet special characters.

NYSIIS: It was first introduced in 1970 as part of the "New York State Identification and Intelligence System" (Rajkovic and Jankovic, 2007). This algorithm works by transcoding the characters of a string based on predefined rules, removing trailing vowels, collapsing all strings of repeated characters, and keeping only the first 6 characters in final code. *NYSIIS* tends to produce better results than *Soundex* because NYSIIS deals with some multi-character n-grams and supports relative vowel positioning, whereas *Soundex* does not. It obtains an accuracy increase of 2.7% over the *Soundex* algorithm (Rajkovic and Jankovic, 2007). This algorithm was designed to be used with American names as well as with phonemes that occur in European and Hispanic surnames.

Data Preparation

The data used for this comparative analysis was extracted from the SCOPUS [2] bibliographic database. SCOPUS is a very large citation database of peer-reviewed literature, enabling the development of smart tools to track, assess, analyze and visualize citations. The dataset holds information about citations for different types of publications of many authors. In order to perform evaluations, we collected 185,253 records corresponding to 2398 distinct Canadian authors disambiguated manually, with a wide variety of origins (European, Asian, etc.). Author surnames were used as input data in our experiments.

[2] www.scopus.com

Evaluation Metrics

We implemented and developed special metrics to assess each of the algorithms based on different characteristics. Each metric addressed specific assessment criteria, and this enabled us to compare these algorithms in the perspective of blocking in disambiguation algorithm.

We now present the proposed metrics:

- *Number of buckets*: This metric is used to determine which algorithm obtains an appropriate number of buckets since we are interested in finding a compromise between quality of the results and time efficiency. In particular, we would like to obtain large enough number of buckets to contain all the variations to improve the accuracy of the results, and also we would like to obtain buckets containing small enough number of names to avoid doing a lot of comparisons.

- *Cross-bucket distance*: This metric evaluates by which proportion dissimilar words are placed in different buckets. This metric should return large values of between-buckets distances if the phonetic algorithm performs well. In other words, when applying the blocking method, dissimilar words end up in separate buckets far away from each other. Since we do not have any gold standard to compare our result, we used the Levenshtein distance (Levenshtein, 1966) as a reference indicator. We conducted several tests with other metrics as reference indicator and because the results were similar, only experiments performed with the Levenshtein distance are reported here. This metric computes the distances between centroids of the buckets created by each phonetic algorithm, then we report which proportion of distances is greater than a threshold. The threshold was determined because our results showed that all the distances below this thresholds were relatively similar for all algorithms, hence removing them from the comparisons could reduce the calculation efforts.

- *Intra-bucket distance*: This metric is defined to evaluate to what proportion similar words placed in the same bucket using each algorithm. To assess the performance of the algorithms in this respect, we used the Levenshtein distance as a reference indicator (for the same reason explained in the previous metric) and computed the normalized distances between all the elements of each bucket. Finally, we report the proportion of those only with a value less than a threshold. According to our results, we decided to remove the comparisons of distances higher than this threshold, since the numbers were relatively similar for all types of algorithms, therefore it could reduce the calculation efforts.

- *Percentage of errors*: This metric measures number of names inside a bucket that must be replaced into another bucket (wrongly grouped ones). The lower error indicates better quality of the phonetic algorithm.

Algorithm 1 Computing the number of errors per bucket

```
1: procedure COMPUTE ERROR($bucketA$)
2:     $error = 0$
3:     for each name $i$ in $A$ do
4:         for each name $j$ in $A$ do
5:             $dist1$ = AVG(Levenshtein($i$, $j$)) , $i \neq j$
6:         end for
7:         for each bucket $B$ in $AllBuckets$ do
8:             for each name $k$ in $B$ do
9:                 $dist2$ = AVG(Levenshtein($i$, $k$))
10:                if $dist2$ < $dist1$ then
11:                    $error$++;
12:                end if
13:            end for
14:        end for
15:    end for
16:    Return $error$
17: end procedure
```

Algorithm 1 shows the procedure used to compute the number of errors for each bucket. First, for each and every bucket $A_{1...N}$, we calculate the average distance of each name $i \in A_i$ with all other names $j \in A_i$ in the same bucket, and this value is stored in variable $dist1$. Then, for all other buckets $B_{1...N-1}$, we calculate the average distance between the name i and all other names $k \in B$ and save it in $dist2$. If $dist2$ is less than $dist1$ it means that the name is misplaced because it is more similar to other names in another bucket. Hence, it should be replaced, and the number of errors should be increased. Once we computed the error for all the buckets, we report the percentage of the total number of errors divided by data points.

- *Number of comparisons*: This metric is directly related to the application of phonetic encoding for blocking, whose numerical efficiency depends on the total number of comparisons between words. The number of comparisons is an arithmetic progression of the number of elements in a bucket. This metric computes the total number of pairwise comparisons that a disambiguation algorithm would have to perform if it was based on the blocking partitioning generated by a phonetic algorithm. This provides an indication of its suitability for blocking.

- *Execution time*: Computation time is an important element when dealing with large datasets, but clearly such a metric is machine- and implementation-dependent, hence the importance of the other quantitative indicators. In order to make this metric more precise, the reported values represent the average execution time of 100 runs, all performed in similar conditions. The blocking procedure based on the various phonetic encoding algorithms was implemented as a Java application. Since the different phonetic algorithms obtained a different number of buckets, we had to normalize the scores with respect to the number of buckets.

4.1.2 Step 2: Comparing String Similarity Metrics

While the literature proposes compelling disclosure of comparative studies on the precision of such algorithms, other characteristics such as speed, sensitivity to thresholds and desired discriminatory power are rather neglected in most cases. The goal of this step is to compare several algorithms to find one that adapts best to the situations where these attributes of the algorithm are unavoidably valuable to improve the application under study. We discuss a new string similarity algorithm: *Weighting Table-Based* that has been already developed specifically for the application under study as a string similarity measure in classifiers. We evaluate the advantages and limitations of this method by comparing it with several prominent well-known algorithms.

Selection of String Similarity Metrics

Gomaa and Fahmy (Gomaa and Fahmy, 2013) grouped string-based similarity measures in two categories: *Character-based similarity measures*, which consider distance as the difference between characters of strings (thus useful in the case of typographical errors) and *Term-based similarity measures*, which take into the account the distance between the two terms. That said, these types of categorization are mostly addressing the problem of document clustering, however, our analysis is related to the string-to-string comparison.

For this reason, we chose several well-known algorithms and grouped them in different categories that will be explained in this section:

Weighting Table-Based (WTB) Algorithm

This algorithm is a string similarity measure that provides a quick comparison of letters of two words which eventually helps to increase the speed of the disambiguation process. It computes the similarity of two words converted to integer arrays corresponding to the indexes of the letters. WTB is sensitive to detect changes in letters of strings and penalizes transformations (changing and swapping letters) by assigning weights to them. Another type of weight is added considering only the changes in first letters of the two strings. For instance, it adds up a weight for a change from the letter "K" in "Katerine" when compared with the letter "C" of "Catherine", meaning that instead of putting zero for the similarity of such two letters, it detects the change and put a non-zero value. It also considers that the shift might have occurred in just more than *N* letters, equal to the defined radius.

It also makes it possible to weight transformations between letters and digits such as "O" and "0". The weighting of transformations allows a considerable flexibility and makes the algorithm adaptable to different languages. Algorithm 2 presents the pseudo-code of this algorithm.

Algorithm 2 Weighting Table-Based string similarity algorithm

```
1: procedure SIMILARITY($string1$, $string2$)
2:     $SETradius$
3:     $SETcostShift$
4:     $SETcostDiff$
5:     $SETcostFirst$
6:     $difference\_Sizes = |string1.length - string2.length|$
7:     if ($difference_Sizes > this.maxDiff$) then return $0$
8:     end if
9:     $ind1 \leftarrow getIndices(string1)$
10:    $ind2 \leftarrow getIndices(string2)$
11:    $WTB \leftarrow loadthetableofweights$
12:    $penalty = \frac{costDiff \times max(string1.length, string2.length)}{min(string1.length, string2.length)}$
13:    $dist = 0$
14:    for $i = 0 \rightarrow min(string1.length, string2.length)$ do
15:        $distRadius = vector[4 \times radius + 1]$
16:        if ($i == 0$) then
17:            $distRadius.add(WTB[ind2[i]][ind1[i]] \times costFirst)$
18:            $ElsedistRadius.add(WTB[ind2[i]][ind1[i]])$
19:        end if
20:        for $j = 0 \rightarrow radius$ do
21:            $penaltyShift = costShift \times |j - i|$
22:            $distRadius.add(WTB[ind2[i]][ind1[i - j]] + penaltyShift)$
23:            $distRadius.add(WTB[ind2[i]][ind1[i + j]] + penaltyShift)$
24:            $distRadius.add(WTB[ind1[i]][ind2[i - j]] + penaltyShift)$
25:            $distRadius.add(WTB[ind1[i]][ind2[i + j]] + penaltyShift)$
26:        end for
27:        $dist += min(distRadius)$
28:    end forreturn $dist$
29: end procedure
```

Table 4.1 Weighting Table-Based algorithm parameters

Parameter name	**Default value**	**Description**
radius	2	Radius of the filter for the comparison of pairwise letters
costShift	0.4	Cost of swapping two letters
costDiff	0.4	Cost of the difference of size of two letters
costFirst	1.7	Cost of the difference of two first letters

The steps of the algorithm can be summarized as:

1. Initializing parameters, depicted in Table 4.1. The default values of these parameters were determined manually by trials and errors.

2. Compute the difference between the size of the two strings and return zero if it is greater than the radius.

3. Convert the two strings to indices of letters.

4. Construct similarity matrix of letters, which contains the cost of changing one letter to another. For certain letters, we initialize the matrix values by specific weight values.

5. Compute a penalty related to the size difference of the two strings, replacing a letter by

another, the penalty related to the shift between letters, and an additional penalty on the first letter.

6. Iterate through the letters of the shortest string and generate a vector storing all the distances between pair letters within a radius.
7. Compute the distance between letters of the first index.
8. Iterate over all the indices in a radius and add up a penalty based on the number of shifts.
9. Perform the comparison of two letters within the radius in both strings.
10. Find the minimum distance within the radius and return it.

Parameters shown in Table 4.1 are determined manually based on simple assumptions:

- Mistakes on the first letter of a word rarely happens, therefore, the cost of such mistake is determined to be high.
- Swapping letters in a radius greater than 2 characters is not likely to happen, that is the reason why the radius size is set to 2.
- The inversion of letters is quite common as a mistake, therefore the cost is determined to be low.

One future improvement of this algorithm would be automatic determination of these parameters.

Edit-based Similarity Measures

These measures are used to calculate the difference between strings as a weighted aggregate of the number of additions, eliminations, substitutions and additionally transpositions needed to obtain the second string from the first one. Weighting table-based and the following three algorithms belong to this category:

Levenshtein: It is a commonly used similarity measure that describes the distance between two strings by checking the base number of operations expected to change one string into the other, where an operation is defined as an addition, cancellation, or substitution of a character, or a transposition of two nearby characters (Levenshtein, 1966). In other words, The Levenshtein distance between two strings is the minimum number of edits needed to transform one string into the other, with the permissible operations being: insertion, deletion, or substitution of a single character.

Therefore, the similarity between two words is $(1 - distance)$. Levenshtein is good for measuring the similarity between words, and it has been used widely for spell checking since it detect the variations in spelling.

Smith-Waterman: This algorithm aligns two strings (sequences of characters) by matches or mismatches (substitutions), insertions, and deletions to find the segments that have similarities (determining similar regions between two strings) (Smith and Waterman, 1981). The Smith-Waterman algorithm compares segments of all possible lengths and optimizes similarity degree. More specifically, this algorithm determines the sequence of operations needed to transform one string to another, but attributes lower weights to transformations among similar-sounding characters and employs specialized logic for handling alignment gaps such as a "gap start" penalty corresponding to the beginning of a string of unmatched characters, and a separate "gap continuation" penalty for its continuation.

Jaro–Winkler: It is, basically, an extension of Jaro distance (Jaro, 1989). In theory, the Jaro distance is identifiable as the minimum number of single-character transpositions required to change one string into the other, whereas the Jaro-Winkler distance utilizes a prefix that establishes more favorable weights to strings that match from the beginning for a set prefix length. In other words, Winkler modified this algorithm such that differences close to the start of the string have larger impacts on the measure than differences observed at the end of the string.

Token-based Similarity Measures

They are a type of similarity functions that first try to tokenize strings as token[3] sets, and then compute the similarity based on the token sets. Usually, two similar strings are identifiable by having a large overlap in the sets. Nonetheless, Token-based similarity measures suffer from a limitation that they only consider the exact match of two tokens in a bag of words, hence ignoring string fuzzy matches. In general, such measures are not quite efficient to calculate similarity when typos and misspelling words are introduced. Three following algorithms are from this category:

Jaccard: This measure is mostly used in document similarity applications. The Jaccard index refers to the ratio of the size of the intersection of two strings to the size of their union (Jaccard, 1912). In order to use this algorithm, a document, typically, must be presented as a bag of words which is the list of unique words in it, then we can compute Jaccard index between them. The

[3]In natural language processing (NLP), a token is identified as a unit of processing. In most cases, it is just a string of contiguous characters between two spaces, or between a space and punctuation marks. (Baeza-Yates et al., 1999)

Jaccard similarity measure between two strings A and B is calculated by first, converting them to sets of n-grams (sequences of n characters, also called k-shingles), then it computes the similarity as $\frac{|A|+|B|-|A\cup B|}{|A\cup B|}$ where $|A|$ and $|B|$ represent the size of the two strings and $|A \cup B|$ is the size of the union of the two strings.

Sorensen-Dice: This algorithm works by comparing the number of identical character pairs between the two strings (Sørensen, 1948). It is often called *Sørensen index* or *Dice's Coefficient*. It measures the similarity between two strings A and B by first, converting them to boolean sets of k-shingles (sequences of k characters), then the similarity is computed as $\frac{2\times C}{(|A|+|B|)}$ where $|A|$ and $|B|$ represent the size of the two strings, and C is the number of identical character pairs (common terms) between the two strings.

Cosine: This algorithm is a very famous similarity measure, extensively used in document similarity in information retrieval domain (Baeza-Yates et al., 1999) and clustering (Larsen and Aone, 1999). Cosine similarity measure determines the cosine of the angle between two vectors. Once the strings are transformed in vectors of occurrences of sequences of *K* characters, the similarity between them will be the cosine of their respective vectors.

Hybrid Similarity Measures This type of similarity measures combines the benefits of edit-based and token-based methods. When more control is needed over the similarity measure, hybrid algorithms can be effective. Unlike edit-based measures, hybrid measures can be used for matching an attribute value to its abbreviation or acronym. The following algorithm belongs to this category:

Monge-Elkan: This similarity measure computes the average of the similarity values between the more similar token pairs in both strings. This algorithm was introduced by Monge and Elkan (Monge et al., 1996) and it has been used in many name-matching and record linkage comparative studies (Bilenko and Mooney, 2003) (Branting, 2003). This hybrid method maintains the properties of the internal character-based measure, the ability to deal with misspellings, typos, OCR errors, and deals successfully with missing or disordered tokens, and it can combine any token comparison measure. Given two texts *S1*, *S2* and their number of tokens $|S1|$ and $|S2|$ respectively, the Monge-Elkan algorithm measures the average of the similarity values between pairs of more similar tokens within texts *S1* and *S2*. The main advantage of this algorithm is being recursive, which gives an ability to handle sub-fields or sub-sub-fields, meaning that the algorithm is more likely to find a match between a string and its corresponding incomplete string in several formats.

Data Preparation

In this step of our research, we used the WikEd Error Corpus (Grundkiewicz and Junczys-Dowmunt, 2014), which is a large and diversified data set extracted from the Wikipedia revision history. It is a freely available corpus including 12 million sentences and a total of 14 million edits of various types. Although there are many types of edits in this corpus, the scope of our research is limited to correction of spelling errors. Because the data is too noisy and the variability of texts is high, we had to perform an error selection process. This process included the following two steps of extraction and cleansing. First, during the extraction, we implemented a program that crawls the raw data and looks into the spelling errors that are presented inside the logs with the format: *[-donload-] [+download+]*. Basically, this format indicates that the word between the two 'minuses' was misspelled and replaced with the word between the two 'pluses'.

Then, in the cleansing step, we filtered out texts consisting of stop words such as "and", "then", "when" etc, and those including both numbers, digits and special characters. Even though this refinement process reduced the size of our dataset, the remainder contained a very large number of records. Since the average length of a word in most English documents[4] is over 5, we eliminated non-English words and those with less than five characters. Finally, a subset of 10,000 misspelled words was assembled to perform the study. A gold standard was brought together by merging the results of forming 777 groups of similar words by performing several manual adjustment and verification processes. Each group was carefully edited by hand and repeatedly verified to ensure having a reliable solution file. The biggest group contained 40 words and smallest one only 4 words.

Methodology Design

Since it is difficult to perform a direct comparison between selected similarity algorithms, we had to proceed with an auxiliary step of clustering to conduct indirect comparison tests. The implication of our evaluation is that a similarity algorithm outperforms the others if the cluster analysis shows better results while using this particular algorithm. In this section, we discuss our methodology regarding the implementation of different parts of the evaluation process.

We set out to perform a comparative analysis of selected string similarity algorithms by implementing a program that generates clusters, which we call *Buckets*, and estimating the correctness of the obtained results by comparing them to the gold standard. The operational methods are demonstrated in Figure 4.1, and can be itemized as follows:

- Creating the similarity matrix: This corresponds to a similarity graph with data points for

[4]http://www.wolframalpha.com/input/?i=average+english+word+length

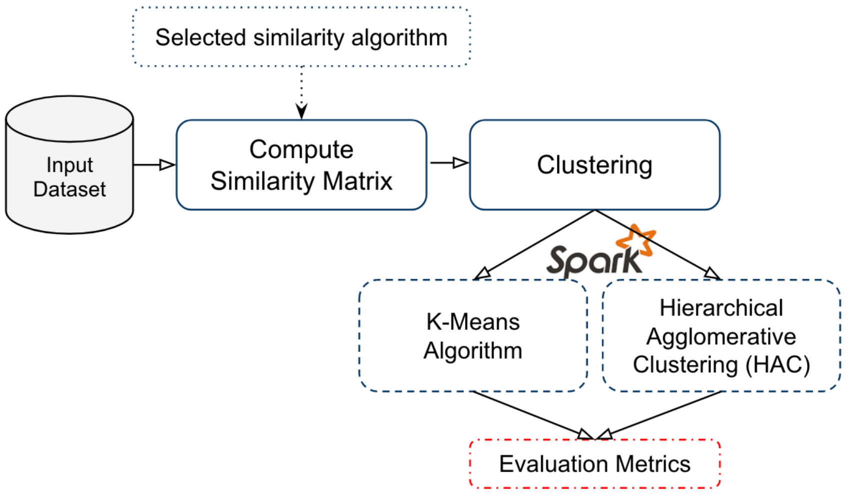

Selected similarity algorithm
Input Dataset
Compute Similarity Matrix
Clustering
Spark
K-Means Algorithm
Hierarchical Agglomerative Clustering (HAC)
Evaluation Metrics

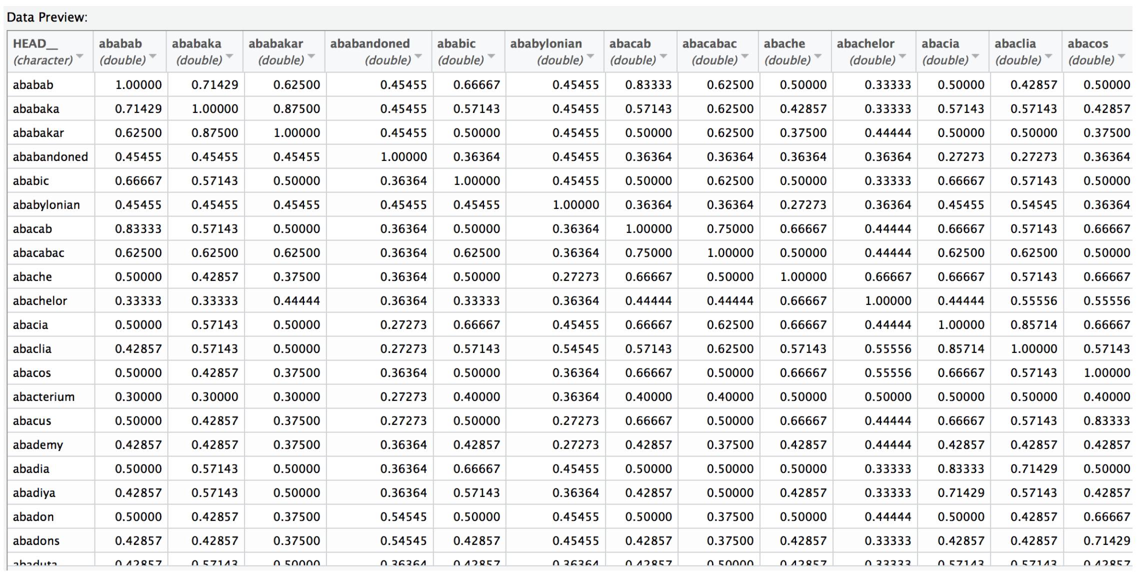
Figure 4.1 Methodology steps

nodes and edges whose weights are the similarity between data points represented by a value between 0 and 1 computed using specific similarity algorithm. Figure 4.2 shows a screen shot of a sample similarity matrix created using one of the similarity algorithms. The dimension of all similarity matrices is $10^4 \times 10^4$. Each point represents the similarity value between the words in corresponding row and column. We used a Java library called SimMetrics (Chapman, 2009) that contains the implementations of string similarity algorithms in order to develop an application that performs pair-wise comparisons to compute this matrix for all of the algorithms. However, the Weighting Table-based method is not part of this library and it was developed and implemented beforehand.

- Implementing K-Means clustering algorithm (MacQueen et al., 1967) to build clusters (that we refer them as *buckets*) of similar words by using each similarity matrix for different similarity algorithms. K-Means requires setting the parameter for the number of buckets, and in our experiment, we set this number to the number of buckets in our gold standard. We used Apache Spark machine learning library (Meng et al., 2016) to implement the K-Means algorithm.

- Implementing Hierarchical Agglomerative Clustering (HAC) (Cios et al., 2012) to create buckets with no a priori information about the number of clusters required.

  HAC is a bottom-up clustering method which creates clusters that have sub-clusters repetitively. It starts with every single object (sample data) in a single cluster. Then, in each successive iteration, it agglomerates (merges) the closest pair of clusters by satisfying some similarity criteria, until all of the data is in one cluster. Use of different distance metrics



Data Preview:

HEAD__ (character)	ababab (double)	ababaka (double)	ababakar (double)	ababandoned (double)	ababic (double)	ababylonian (double)	abacab (double)	abacabac (double)	abache (double)	abachelor (double)	abacia (double)	abaclia (double)	abacos (double)
ababab	1.00000	0.71429	0.62500	0.45455	0.66667	0.45455	0.83333	0.62500	0.50000	0.33333	0.50000	0.42857	0.50000
ababaka	0.71429	1.00000	0.87500	0.45455	0.57143	0.45455	0.57143	0.62500	0.42857	0.33333	0.57143	0.57143	0.42857
ababakar	0.62500	0.87500	1.00000	0.45455	0.50000	0.45455	0.50000	0.62500	0.37500	0.44444	0.50000	0.50000	0.37500
ababandoned	0.45455	0.45455	0.45455	1.00000	0.36364	0.45455	0.36364	0.36364	0.36364	0.36364	0.27273	0.27273	0.36364
ababic	0.66667	0.57143	0.50000	0.36364	1.00000	0.45455	0.50000	0.62500	0.50000	0.33333	0.66667	0.57143	0.50000
ababylonian	0.45455	0.45455	0.45455	0.45455	0.45455	1.00000	0.36364	0.36364	0.27273	0.36364	0.45455	0.54545	0.36364
abacab	0.83333	0.57143	0.50000	0.36364	0.50000	0.36364	1.00000	0.75000	0.66667	0.44444	0.66667	0.57143	0.66667
abacabac	0.62500	0.62500	0.62500	0.36364	0.62500	0.36364	0.75000	1.00000	0.50000	0.44444	0.62500	0.62500	0.50000
abache	0.50000	0.42857	0.37500	0.36364	0.50000	0.27273	0.66667	0.50000	1.00000	0.66667	0.66667	0.57143	0.66667
abachelor	0.33333	0.33333	0.44444	0.36364	0.33333	0.36364	0.44444	0.44444	0.66667	1.00000	0.44444	0.55556	0.55556
abacia	0.50000	0.57143	0.50000	0.27273	0.66667	0.45455	0.66667	0.62500	0.66667	0.44444	1.00000	0.85714	0.66667
abaclia	0.42857	0.57143	0.50000	0.27273	0.57143	0.54545	0.57143	0.62500	0.57143	0.55556	0.85714	1.00000	0.57143
abacos	0.50000	0.42857	0.37500	0.36364	0.50000	0.36364	0.66667	0.50000	0.66667	0.55556	0.66667	0.57143	1.00000
abacterium	0.30000	0.30000	0.30000	0.27273	0.40000	0.36364	0.40000	0.40000	0.50000	0.50000	0.50000	0.50000	0.40000
abacus	0.50000	0.42857	0.37500	0.27273	0.50000	0.27273	0.66667	0.50000	0.66667	0.44444	0.66667	0.57143	0.83333
abademy	0.42857	0.42857	0.37500	0.36364	0.42857	0.27273	0.42857	0.37500	0.42857	0.44444	0.42857	0.42857	0.42857
abadia	0.50000	0.57143	0.50000	0.36364	0.66667	0.45455	0.50000	0.50000	0.50000	0.33333	0.83333	0.71429	0.50000
abadiya	0.42857	0.57143	0.50000	0.36364	0.57143	0.36364	0.42857	0.50000	0.42857	0.33333	0.71429	0.57143	0.42857
abadon	0.50000	0.42857	0.37500	0.54545	0.50000	0.45455	0.50000	0.37500	0.50000	0.44444	0.50000	0.42857	0.66667
abadons	0.42857	0.42857	0.37500	0.54545	0.42857	0.45455	0.42857	0.37500	0.42857	0.33333	0.42857	0.42857	0.71429
abaduta	0.42857	0.57143	0.50000	0.36364	0.42857	0.36364	0.42857	0.50000	0.42857	0.33333	0.57143	0.57143	0.42857

Figure 4.2 Previewing first few entries of a similarity matrix

(similarity algorithms in our problem) for measuring distances between clusters may generate different results. An HAC tree plot visually demonstrates the hierarchy within the final cluster, where each merger is represented by a binary tree. Figure 4.3 shows a sample HAC tree plot. It can be viewed that using the HAC, two very similar words are joined at a "node", representing a "cluster". That cluster is joined to the next nearest word or sub-cluster, and so on. At the end, very similar words tend to appear beside each other in the diagram. It is possible to define the clusters by cutting branches off the dendrogram using a constant height cutoff value (cutree height values) shown as vertical dotted lines in Figure 4.3. These cutree height values are similar to the threshold values for the WCs in our training framework.

The overall process of HAC can be summarized as:

- Assign each object to a separate cluster.
- Evaluate all pair-wise distances between clusters.
- Construct a distance matrix using the distance values.
- Look for the pair of clusters with the shortest distance.
- Remove the pair from the matrix and merge them.
- Evaluate all distances from this new cluster to all other clusters, and update the matrix.
- Repeat until the distance matrix is reduced to a single element.

We used HAC algorithm to measure the sensitivity of the algorithms to thresholds. For training purposes, an algorithm performs better if it is not too sensitive to the change in

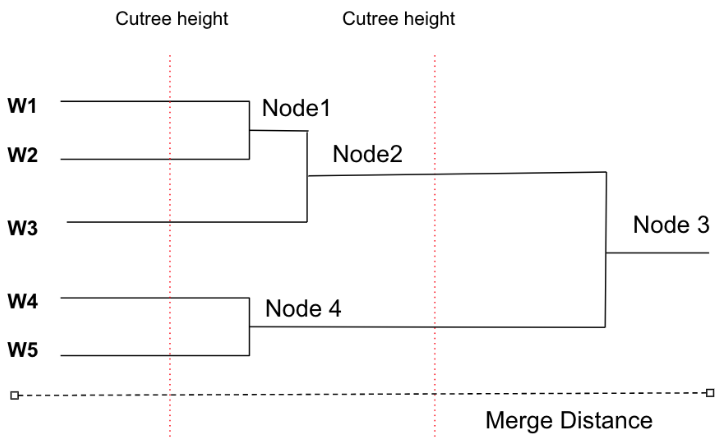


Figure 4.3 Hierarchical agglomerative clustering tree plot

threshold. In fact, by observing the number of buckets created at each height level in HAC, we can compare different algorithms and understand which one results in a dramatic change to the number of clusters with regard to a change in the threshold. We used Apache Spark machine learning library (Meng et al., 2016) to implement HAC algorithm.

Some metrics were used to evaluate the results of clustering methods. In the following section, we explain such metrics and their purpose in details.

Evaluation Metrics

We developed several quantitative metrics to assess clustering results, hence realizing indirect comparison tests between the similarity algorithms:

- *Average intra-bucket distances:* This metric computes the total distances between words (corresponding to each similarity metric) in each bucket divided by the number of words in the same bucket in a range between 0 and 1 (lowest and highest similarity between words, respectively). If a string similarity algorithm performs well, majority of the buckets it obtains should contain words with a very low distance between them. By plotting the frequency of the average distances for all the buckets, we can visualize the distribution of the distances inside buckets, hence it gives an insight about the peculiarity of the results obtained by each string similarity algorithm. We expect to see high frequency of the low average distance values for the algorithms that outperform the others.

  This metric gives an indication of coherence and the internal functioning of each similarity metric. However a good value for this metric does not guarantee the quality of the buckets, therefore, we also calculate the Rand Index measure.

- *Rand Index (RI):* As an external evaluation criterion, the Rand Index (Rand, 1971) measures the similarity between two clusterings by considering all pairs of samples and counting pairs that are placed in the same or separate clusters in the predicted and true clusterings. In other words, this metric makes a comparison between the assignments of each pair of words in buckets in the gold standard and the computed clustering of each string similarity algorithm.

  The mathematical definition of rand index is as follows:

  $RI = \frac{TP+TN}{TP+FP+FN+TN}$

  where:

  - TP is the two words of a same class that are in the same bucket
  - TN is two words of two different classes are in different buckets
  - FP is two words of two different classes are in the same bucket
  - FN is two words of a same class are in two different buckets

  The rand index has a value between 0 and 1, where 0 indicates that the computed clustering and the gold standard do not agree on any pair of points and 1 means that the two clusterings are exactly the same.

- *Number of buckets per threshold:* Using this metric helps understand the sensitivity of each algorithm with respect to changes in thresholds. This metric was calculated by measuring the total number of buckets obtained when cutting the dendrogram of the HAC algorithm using different cutree height (threshold) values. An efficient metric is expected to avoid being too sensitive to the threshold, meaning that it should not generate a considerable different number of buckets when the threshold is changed insignificantly. When applied to a learning method, this characteristic of the similarity algorithm has a direct impact because it makes the learning method heavily dependent on the threshold.

- *Total execution time*: The overall time that it takes to generate buckets is calculated in milliseconds. This metric shows the efficiency of string similarity algorithm in terms of computational cost. Lower execution time makes the algorithm more adopting and applicable in a variety of domains with high interest in speed. Our experiments were conducted 30 times and the average amount was reported.

4.1.3 Step 3: Training Methodology for the Cascade Classifier

One of the problems of the existing method is that there is no automated approach for determining the structure and the parameter values of the classifiers. Although some guidelines were taken into account (refer to Section 3.5) to design the most efficient ordering and threshold values such as assigning a high threshold value to fields with low variability, a need for a proper learning algorithm is essential to automate the disambiguation process.

Therefore, within the context of this project, we have developed an automated cascade method for the disambiguation task which includes four aggregators. The classifiers of the first aggregator was determined in advance. These classifiers were operating on very high discriminating and reliable features (e.g. Email addresses that are unique for authors) in order to filter out a large number of records early. The rest of the aggregators were automatically adjusted and configured by the learning algorithm.

The benefits of automating the whole disambiguation process are to make the classification flexible and adjustable when applied to different databases with various data fields, hence being practically efficient regardless of the database types.

In the following sections, we give a general description of the framework, then we explain practical steps toward its implementation.

Training Framework

To determine WCs and the threshold values assigned to each WC, we introduced a training framework. Figure 4.4 illustrates the supervised training framework used to automate the process of configuration embodied in the overall disambiguation process. The whole process contains following steps:

- Read data from the input data set
- Pre-process the input and generate the pre-processed data
- Carry out the training methodology
  - Initialize the training process by creating a solution matrix and initialize the parameters used in the learning process
  - Run the learning algorithm by using the particle swarm and the set cover optimizers.
  - Construct the configuration file (that holds the arrangement and thresholds of the WCs) and output in an XML format.

- Perform the disambiguation process
- Create clusters of entities (authors) based on the results generated by the disambiguator
- Compare the clustering results with the gold standard

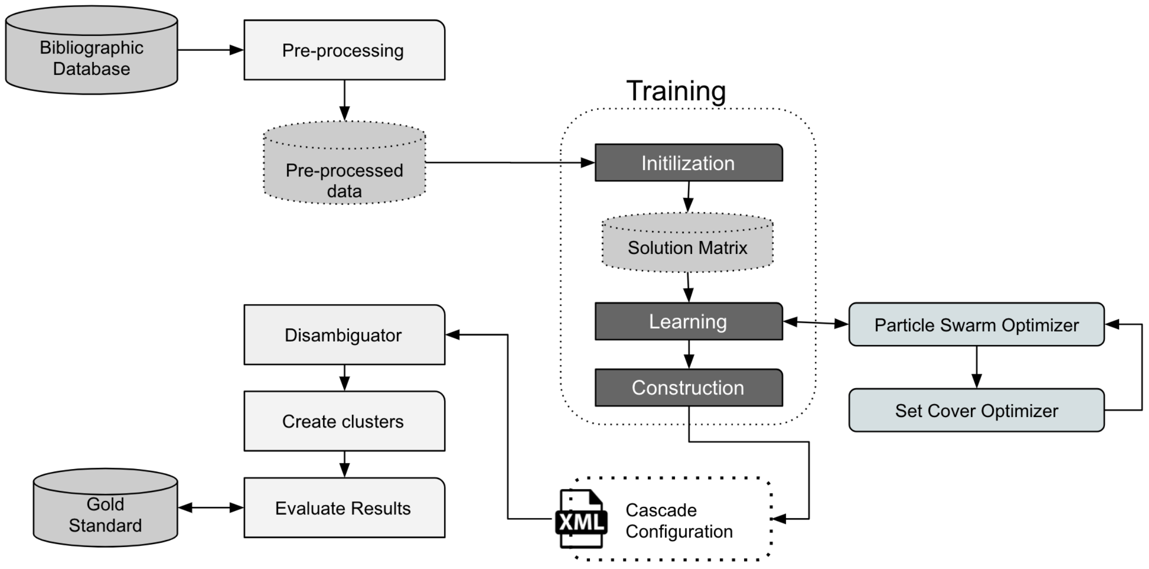


Figure 4.4 Overall Demonstration of the Proposed Framework

As mentioned above, the training methodology contains three procedures: *initialization*, *learning* and *construction*. In this section we describe each of these procedures in details (as depicted in Figure 4.5):

1. **Initialization**: In this procedure, a matrix of distances between records for each weak classifier is created. The rows correspond to each pair of the data and the columns are for the different classifiers. The outcome of this step is a solution matrix containing each pair-wise comparison, the indexes of the false and true solution values of the comparisons and the processing time spent by each classifier while making a decision. The solution created in this step will be used to train our learner models by comparing the results of the classifiers and calculating the rate of correctly classified and wrongly classified records for each classifier.

   This step also includes the initialization of parameters used in the training process. Table 4.2 shows these parameters.

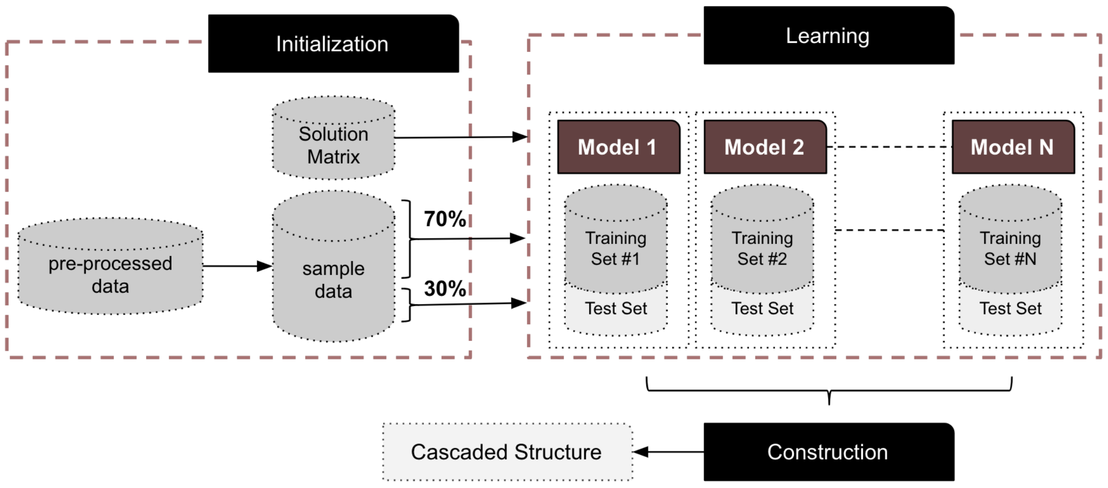


Figure 4.5 Training Methodology

Table 4.2 Parameters used in the learning process

Name	Type	Default Value	Description
AUTOMATE	boolean	true	Run the training method
AUTOMATE_VERBOSE	boolean	true	Generated structure will be displayed in details
SAMPLE_BLOCK	int	1000	Block of scientists to put in train set
NB_CC_MAX	int	6	Maximum number of classifiers to consider
ALPHA	double	10	Penalization weight
BETA	double	2	Penalization scaling factor
PSO_PARTICLES	int	250	Number of particles
PSO_ITERATIONS	int	5000	Number of iterations
PSO_COST	double	0.001	Stop criteria for the PSO objective function
PSO_V1	float	1.0f	Control the weight of the best position achieved by a particle
PSO_V2	float	1.0f	Control the weight of the best position achieved by all particles
PSO_mom_MIN	float	0.0001f	Minimum value allowed for the momentum
PSO_mom_MAX	float	0.0001f	Maximum value allowed for the momentum
PSO_FUN	Function	FUNC2	Objective function

These values are related to the datasets we used in this research (WoS and SCOPUS), which were determined by trials and errors.

2. **Learning**: The second procedure is an algorithm depicted in Algorithm 3 with the purpose of building many learning models of classifiers (illustrated in Figure 4.5) and use an ensemble learning technique to generate the best classifier structure and its thresholds among all the possibilities.

   Two important learning steps of this algorithm are: 1) find the best set of thresholds for the weak classifiers and 2) select the most efficient classifier arrangement. Overall, using an error-driven optimization technique, the learning method performs the two learning steps iteratively until an acceptance criterion is reached [5].

The algorithm contains two optimization methods with their own associated cost functions. It starts with iteratively taking random samples of the input data and split the set of sample records in such a way as to include 70% for the training set and 30% for the test set. The training method initializes WCs with random values of thresholds, then it optimizes those values by applying the PSO method. Next step is to use the SCO method to determine the best arrangement of such classifiers that are speedy while obtaining satisfactory result.

At each optimization process, the learning algorithm puts more weights (α) to wrongly classified records (False Negative rate) of each step to penalize the classifiers with higher error rate. The penalization weight is scaled by the factor of β to increase the pressure of minimizing the number of errors through the end of the process.

Inspired by the AdaBoost model, the poorly classified records are added to the next step at the end of each iteration. The process of the optimization continues in a loop until the algorithm obtains the best-anticipated results (95% confidence level, with only 5% error on the test set) or a certain number of iterations is reached.

The results of each learning model is a classification step that contains an efficient number of weak classifiers with optimized threshold values. For the purpose of aggregating learning models' results, an evaluation function was used in the *construction* procedure in order to compare the classification steps and choose the best one in terms of time efficiency at each and all learning iterations.

The overall process represents a training method with the focus on minimizing wrong outputs i.e. FP and FN, and continuously updating the structure of the classifiers to an optimized version. In the following sections, we explain in details the optimization methods used in the training process:

- **Determining the Thresholds of the Classifiers:** Using the particle swarm optimization (PSO) technique, our learner model creates a swarm of particles that correspond to threshold values (scaled between zero and one). Then, at each step, the *N* most different and best particles were selected based on an objective function. The optimization is repeated until an objective function is satisfied or a maximum number of iterations is reached.

  To determine the best evaluation function that achieves the most efficient classification results, different objective functions were put in place and tested in our PSO method:

[5] This will stop the algorithm before it falls into the trap of over-fitting.

Algorithm 3 Learning Algorithm

```
procedure TRAIN($train_set, classifiers, block\_size$)
    $data \leftarrow pickBlocks(block\_size)$;    ▷ Read input data
    $SETNB\_SAMPLES$;    ▷ Number of samples
    $SET \quad structure.Agg1 \ldots 4$;    ▷ Initialize empty aggregators
    $SET \quad structure.Agg1 \leftarrow StrictClassifiers()$    ▷ Perform strict classification with first aggregator
    $SET \quad optimizerThresholds$;    ▷ Particle Swarm Optimizer (PSO)
    $SET \quad optimizerStructure$;    ▷ Set Cover Optimizer (SCO)
    $SET \quad Error$;    ▷ Boosting error rate
    $SET \quad NB\_CC\_MAX$;    ▷ Maximum number of desired classifiers
    for $i = 1 \rightarrow structure.Agg.size$ do
        for $j = 1 \rightarrow NB\_CC\_MAX$ do
            $SET \quad step$;    ▷ Initialize a step
            $SET \quad samplesIndicesErrors$;    ▷ Store error indexes
            $SET \quad \alpha$;    ▷ Penalization weight
            $SET \quad \beta$;    ▷ Scaling factor for penalization weights
            do
                $samples \leftarrow 70\% \ of \ sample$    ▷ Create the train set
                $samples.addAll(samplesIndicesErrors)$;    ▷ Add errors to train set
                $ds \leftarrow extractDistances(samples)$;    ▷ Extract distances for train data
                $ss \leftarrow extractSolutions(samples)$;    ▷ Extract solutions for train data
                $sts \leftarrow extractTrueSolutions(samples)$;    ▷ Extract true solution indexes for train data
                $thresholds \leftarrow optimizerThresholds.optimize(\alpha)$;    ▷ Optimize thresholds with PSO
                $updateThresholds(thresholds)$;    ▷ Update thresholds
                $crs \leftarrow evaluateClassifier()$;    ▷ Initialize base step evaluations
                $crs2 \leftarrow evaluateClassifier()$;    ▷ Initialize next step evaluations
                $structureIndices \leftarrow optimizerStructure.optimize(\alpha)$    ▷ Optimize orderings with SCO
                $SET \quad totalTime$;    ▷ Total time of classifiers in step
                $SET \quad stepTmp$;    ▷ Temporary step
                for $indStructure \ \in \ structureIndices$ do    ▷ Build temporary step
                    $stepTmp.addClassifier(classifiers.get(indStructure), crs.get(indStructure))$;    ▷ Save the weak classifier
                    $totalTime+ = this.time.get(indStructure)$;    ▷ Compute total time
                    $samplesIndicesErrors.addAll(crs.get(indStructure).getFalseNegativeSet())$;    ▷ Save hard sample indexes
                    $samplesIndicesErrors.addAll(crs.get(indStructure).getFalsePositiveSet())$;    ▷ Save hard sample indexes
                end for
                $stepTmp.buildStepResult()$;    ▷ Build results of temporary steps
                $stepTmp.stepResult.setProcessingTime(totalTime)$;    ▷ Store processing time
                $step.buildStepResult()$;    ▷ Build step results
                $step \leftarrow compareSteps([first], [second], step)$;    ▷ Compare newly created step with previous one
                for $AbstractClassifier \ CL \ \in \ step.stepClassifiers$ do    ▷ Store classifiers for the best step
                    if $CL.Classifier \notin duplicates$ then    ▷ Avoid duplicate classifiers
                        $CL.get(i).addWeakClassifier(j, CL)$;    ▷ Store classifier in final structure
                        $duplicates.add(CL.getClassifierType())$;    ▷ Store the history to avoid duplicates
                    end if
                end for
                $test\_set \leftarrow 30\% \ of \ sample$;    ▷ Create the test set
                $ERROR\_TH = 5\% \ of \ test\_set.size()$    ▷ Boosting error threshold equals to 5% of test samples size
                $ds\_test = extractDistances(test\_set)$;    ▷ Extract distances for test data
                $ss\_test = extractSolutions(test\_set)$;    ▷ Extract distances for test data
                $crs3 \leftarrow evaluateClassifier()$;    ▷ Test new structure on test set
                $SET \quad FN$    ▷ Number of False Negatives
                $SET \quad FP$    ▷ Number of False Positives
                for $index \ \in crs3.keySet()$ do    ▷ Compute results accuracy of test
                    $FN+ = crs3.get(index).getFalseNegativesCount()$;    ▷ Compute False Negatives
                    $FP+ = crs3.get(index).getFalsePositivesCount()$;    ▷ Compute False Positives
                end for
                $Error = (FN + FP)$;    ▷ Compute error on test samples
                $\alpha \leftarrow \alpha \times \beta$    ▷ Scaling penalization for hard samples
            while $(Error > ERROR\_TH)$    ▷ Error-driven optimization
        end for
    end for
end procedure
```

- **FUNC 1**: The first function is based on the sigmoid function. Using this method of evaluation, we penalize classifiers with a higher weight if their results show an increase in FP, therefore, the overall process steers the method toward obtaining lower FP. Below equation shows the calculation method for this objective function:

  $Cost = (\frac{FN}{FN+TP}) + (\frac{FP}{FP+TP} + \frac{1}{L}) \times \alpha \times \beta$

  where FP and FN are number of misclassified errors, TP is number of correctly classified records, α is penalization weight, β is penalization scaling factor and L is size of data.
  The aim of this objective function is to minimize the cost such that the associated classifier generates lower FP rates.

- **FUNC 2**: It is a customized function very similar to the accuracy calculated from the precision and recall, with the focus on minimizing False Negative (FN) rates as the iterations proceed:

  $Cost = (\frac{FP+FN\times\alpha\times\beta}{TP+TN+FP+FN\times\alpha\times\beta})$

- **FUNC 3**: We have also tested the classical accuracy statistical measure in our objective function which represents the proportion of true results. In this case also a penalty was applied to the FN:

  $Cost = (\frac{TP+TN}{TP+TN+FP+FN\times\alpha\times\beta})$

The value of the parameter α (used to add weights to classifiers with wrong results) and β (the scaling factor to increase α) were determined manually by conducting a certain number of tests using above objective functions. However, results obtained by using *FUNC 2* are interesting despite an optimal search for the right value of the parameters.

- **Determining the Orderings of the Classifiers:** In order to determine the most efficient arrangement of the weak classifiers in each step, we used the set cover optimizer (SCO) based on the integer linear programming (ILP). In fact, the problem was to find a minimum number of weak classifiers, with efficient processing time while having maximum number of correctly classified records (TP) and minimum number of errors (FN) in their results.

  We formulate the linear programming to solve as such:

  Minimize: $\sum_{C_{i=1}}^{N} t_{C_i} \times (1 + \frac{\alpha\times\beta\times FN_{C_i}}{TP_{C_i}})$

  Where C_i is the classifier i, and α is the penalization weight, β is the scaling factor for

α, FN_{C_i} and TP_{C_i} are the number of wrongly classified records and correctly classified records of the classifier i, respectively (computed by comparing the result of each classifier with the solution discussed in the *initialization* step), t_{C_i} is the processing time of the classifier i, and N is the total number of classifiers,

Subject to: $\forall r \in S_{TP}: \quad \sum_{C_{i=1}}^{N} C_{i_r} \geq 1$

Where S_{TP} is the set of TPs captured for each step, r is a record in S, and C_{i_r} indicates the index of the classifier that classified record r correctly and $C_{i_r} = 1$ shows the presence of the classifier, whereas $C_{i_r} = 0$ shows the absence of the classifier.

The aim is to find the minimal number of weak classifiers that covers the set of true positive. In fact, among all the WCs, we seek to choose those that will allow to have only true positive results with more efficient processing time.

At first, we have all the WCs that cover the TPs space including a few number of FNs (note that we want to discover all possibilities). Then, the optimizer tries to reduce the number of FNs by putting more weights on classifiers that produced more FNs in their results. Thereby, the final result of this optimization process is a minimal set of selected time-efficient weak classifiers with a maximum number of correctly classified records.

To figure out the impact of choosing the different solver implementations on the efficiency of the SCO, we decided to try out three different solvers and find the most appropriate one to our problem:

- **LPSOLVE**: It is a free (GNU[6] licensed software) linear integer programming solver based on the revised simplex and the Branch-and-bound methods for the integers [7].
- **GLPK**: It is a free (GNU) package that is intended for solving large-scale linear programming (LP), Mixed Integer Programming (MIP), and other problems [8].
- **CPLEX**: It is a program introduced by the IBM company, that builds a specific small LP model and then solves it [9].

Through several experiments, we concluded that the most efficient results (in terms of both speed and accuracy) were obtained with "LPSOLVE" implementation package.

[6]"Free software license, which guarantees end users the freedom to run, study, share and modify the software"
[7]http://scpsolver.org
[8]http://www.gnu.org/software/glpk/
[9]http://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/

Therefore, it was chosen for our further experiments.

3. **Construction**: The result of each learner model is a classification step, holding the best set of weak classifiers. A comparison between this step and the previous step (the result of the previous learner model) was done to evaluate at each learning iteration the most efficient step with respect to the quality of the results and time efficiency. The scoring function below is used to rank the steps and choose the one with the higher score:

   $Score(S_{i=1...K}) = \frac{1}{t_{S_i} \times (FP+FN)}$

   Where S_i is the step number i, t_{S_i} is the total processing time of the classifiers in this step, and FP and FN represent the total number of wrongly classified records of all the weak classifiers in this step, and K is the number of steps.

   It has to be mentioned that in an ideal case, when FP and FN are both equal to zero, we compare the two steps with regards to the processing time and we choose the one that is faster.

   Once the proper step is chosen, the construction process adds this step to the corresponding aggregator. When the algorithm reaches to a maximum number of iterations, an XML file is generated that holds the information about the cascade classification structure. The result of this process can be used to perform the disambiguation task.

Data Preparation

The first bibliographic database that was used in this step of disambiguation method is Web of Science (WoS) which covers a large amount of citation indexes. This database is indexing over 33,000+ scholarly journals and over 160,000+ conference proceedings (as of August 2017), and it has been published and managed by Clarivate Analytics, formerly Thomson Reuters; ISI (Institute for Scientific Information). We used an extraction of this databases containing 2.89 million records corresponding to 132899 distinct authors. The second bibliographic database was a subset of SCOPUS which is a citation database of peer-reviewed literature: scientific journals, books and conference proceedings. Worldwide, SCOPUS is used by more than 3,000 academic, government and corporate institutions (as in August 2017). The data we used in our training method is a subset of this dataset and it holds 185,253 records including the information belonged to 2398 unique authors.

Evaluation Metrics

The training methodology was put in a test by operating on two datasets including a subset of WoS (large size) and a subset of SCOPUS (small size) under the different set of parameters. For the SCOPUS- and WoS-based datasets, comparison with the disambiguation results provided by (Voorons et al., 2017) was done. Final results, then compared to the gold standards, and to understand the quality of such results, quantitative metrics such as machine learning performance measures were used. These metrics include: Precision, Recall and F1 Score (typically used in document retrieval). These quality measures first defined by Perry, Kent, and Berry in 1955 and within the context of disambiguation application, they can be summarized as:

- *Precision:* The ratio of the records assigned to an author that actually belong to him.
- *Recall:* The ratio of the total number of database records associated with one author that has been correctly assigned to him.
- *F1 Score:* It is a product of the precision by the recall divided by their sum. Basically, it can represent a more realistic measure of the classifier's performance since it takes into the account both precision and recall rates.

Once the training method generated the arrangement and thresholds of the classifiers, they have been used to perform the disambiguation task. Then, results (clusters of disambiguated data) were compared with a ground truth solution and in a form of a confusion matrix, values were reported for each of our metrics. The purpose of this evaluation was to state the quality of the disambiguation process and determine the overall efficiency of the automatically generated cascade structure.

CHAPTER 5 RESULTS

This chapter summarizes our findings and presents our results.

5.1 Results of Comparative Study of Phonetic Distance Metrics

This section presents and discusses the results of our experiments regarding the comparison of phonetic encoding metrics in the disambiguation application. The goal of this step is to evaluate several phonetic algorithms to find the most suitable one for a blocking method used in the disambiguation process. Table 5.1 summarizes the results of each metric used for testing the phonetic algorithms.

Table 5.1 Results of evaluating phonetic algorithms using different metrics

Algorithm	**Number of buckets**	**Intra-bucket normalized distance <0.2**	**Cross-bucket distance >0.8**	**Percentage of errors**	**Number of comparisons**	**Average execution time (ms)**
RefinedSoundex	1845	41%	25%	0.43%	1840	3901
Cavephone2	1434	25%	28%	0.34%	5476	68715
Metaphone	1602	24%	31%	0.24%	3564	3992
DoubleMetaphone	1412	19%	19%	0.27%	4836	5143
ColognePhonetics	1388	24%	27%	0.28%	5260	3458
NYSIIS	1762	35%	29%	0.22%	2818	6221

We first examined the performance of the algorithms with respect to the total number of buckets (Table 5.1, second column). A large number of buckets would indicate that the phonetic algorithm is not successful at detecting similar names, whereas a small number of buckets will result in low accuracy by putting many dissimilar words together in a bucket. A phonetic encoding algorithm should ideally generate a lot of low populated buckets rather than few highly populated buckets, while also grouping very similar names together. In a fair and sensible way, all algorithms cannot be compared using only the number of buckets metric, for this reason we needed to use other indicators, such as the inter- and cross-bucket similarity distances.

Ideally, a metric should yield small intra-bucket distances and large cross-bucket distances. Therefore, the quality of a given metric can be assessed through the proportion of records whose intra-bucket distance (to the centroid) lies below a small threshold, and through the proportion of records whose average cross-bucket distance is greater that a large threshold. Here, the small and large threshold values were set to 0.2 and 0.8 on the scale of zero to one, and the two indicators (proportions) were considered together through their sum.

In this respect, Table 5.1 shows the results of these two metrics. Phonetic encoding algorithms can be put into three groups: *RefineSoundex* and *NYSIIS* have a total around 65%, whereas, *Caver-*

phone2, *Metaphone* and *ColognePhonetics* can be grouped together by having a total between 50 and 55% and, finally, *DoubleMetaphone* has a total below 40%. In the first group, *RefineSoundex* has a low computation volume very likely due to a low number of comparisons, but a high percentage of errors, that will translate into poor *recall* (because of the large number of buckets) when used for blocking in disambiguation tasks. Conversely, *NYSIIS* seems to have a balanced performance. In the second group, *Caverphone2* stands out by an excessive volume of computation, and the other two methods appear quite similar. Finally, *DoubleMetaphone* seems to have a lower performance with only 38% total average distances for between and inside buckets.

Additional information can be drawn from the total number of comparisons performed by the algorithm. Recall that the number of comparisons within a bucket is a quadratic function of the number of elements. With an appropriate encoder, the number of large buckets should be small. In this respect, algorithms yielding large buckets, (*Metaphone* and to a lesser extent *Caverphone2* and *DoubleMetaphone*) may not be well suited for blocking. In addition, a very large number of comparisons can be attributed to a few very large buckets, which is clearly inappropriate for blocking.

With these elements in mind, the two algorithms that stand out are *RefinedSoundex* and *NYSIIS*, as they both provide a low number of comparisons, a limited number of large buckets and appropriate performance in terms of intra- and cross-bucket distances. However, *NYSIIS* exhibits better performance, at the expense of a higher computational cost. The *RefinedSoundex* algorithm is less interesting than the *NYSIIS* because it alway generates a lower Recall rate, and the calculation cost is not a major problem since the phonetic codes for blocking are generated only once in the whole process of the disambiguation.

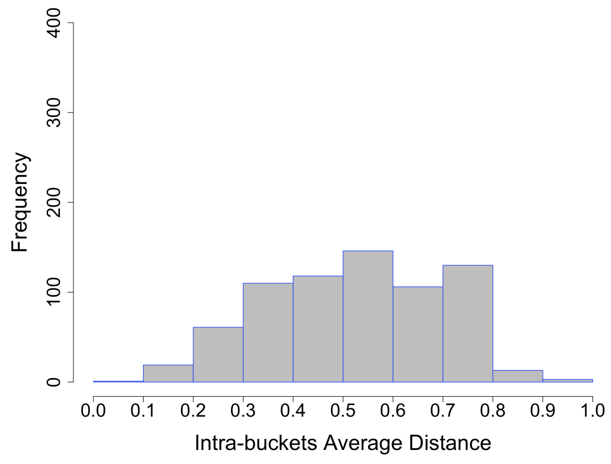
Currently, the application under study uses *DoubleMetaphone*, and our results suggest that it would be advantageous to change for either *NYSIIS* or *RefinedSoundex*.

5.2 Results of Comparative Study of String Similarity Algorithms

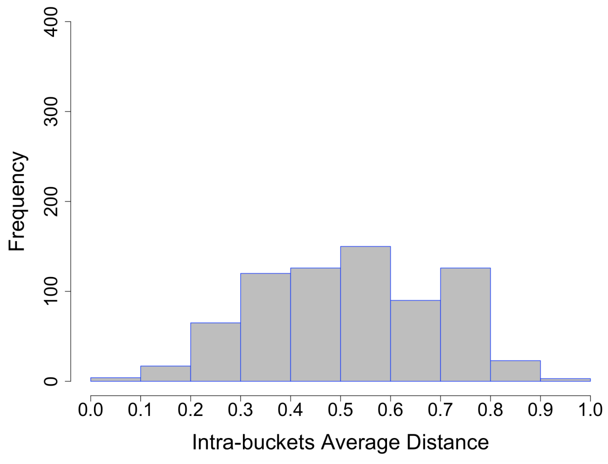
In this section, we discuss the study on the efficiency of the Weighting Table-Based (WTB) string similarity algorithm in comparison with broadly used ones. We report the outcome of experimental analyzes in the direction of specified characteristics that make an algorithm effective while operating on expansive datasets such as bibliographic databases. Our experiments were done using a virtual machine set up on an Intel Xeon 2.6 GHz computer with 28 Gig of RAM.

For all the buckets produced by each algorithm, we computed the normalized distances between the elements of each bucket i.e. the total distances between words in each bucket divided by the number of words in the same bucket. Figure 5.1 and 5.2 depict the distribution of these computed values. As an example, Figure 5.2(b) shows that most of the buckets (approximately 250 out of 777) generated by the WTB algorithm contained word with the average distances between 0.3 and 0.4 to each other.

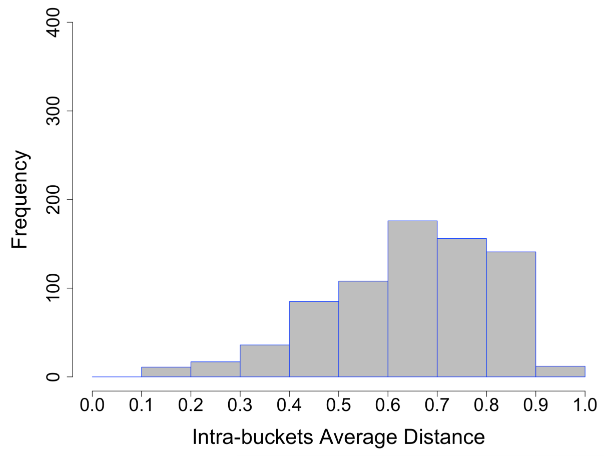
Overall, the figures indicate that the Jaro-Winkler, Weighting Table-Based, and Levenshtein have better results compared to others respectively, since the distribution of inside buckets average distance values are biased toward the minimum values (left side of the histograms). On the one hand, the rest of algorithms show tendency toward an average distance of 0.4 to 0.7 between words in buckets.



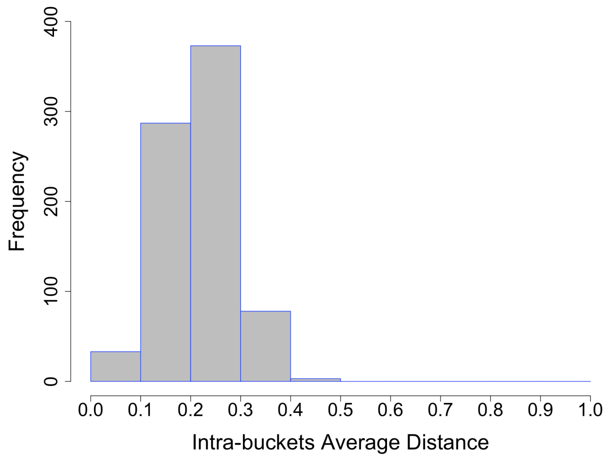
a) *Cosine*



b) *Sorensen-Dice*



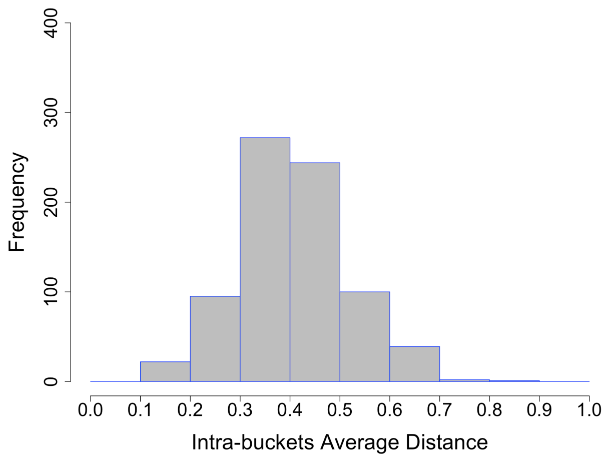
c) *Jaccard*



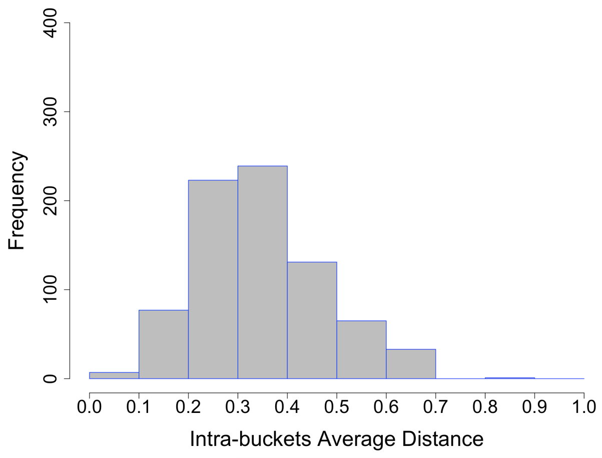
d) *Jaro–Winkler*

Figure 5.1 Intra-buckets average distances

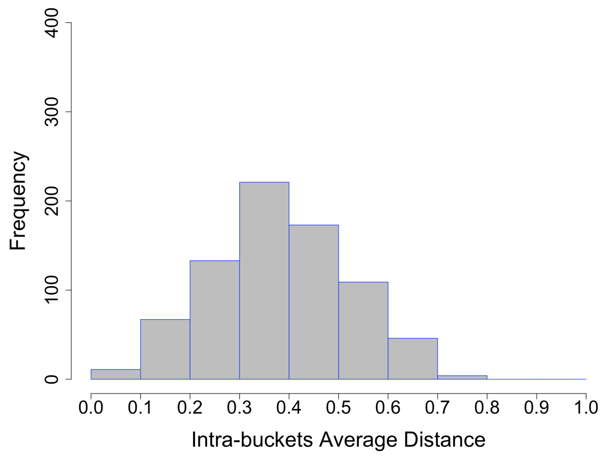
As an example, take the histogram of distances generated by the Sorensen-Dice algorithm in Figure 5.1; it can be seen that the number of distances is higher in the middle range which means that



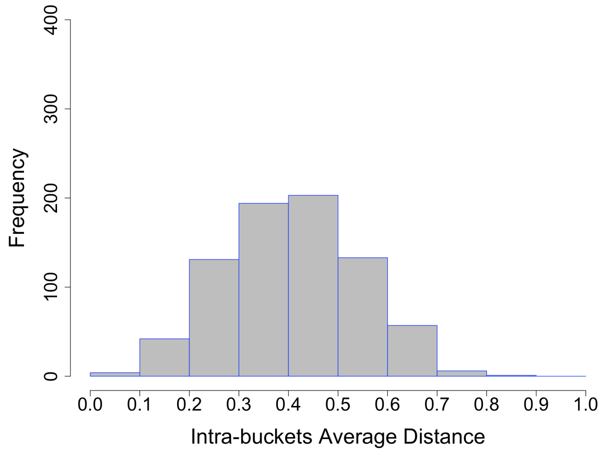
a) *Levenshtein*



b) *Weighting Table-Based*



c) *Monge-Elkan*



d) *Smith-Waterman*

Figure 5.2 Intra-buckets average distances

compared to WTB, it put together more dissimilar words. Undoubtedly, Jaro-Winkler was the best by creating buckets with distances almost less than 0.4 which shows that this algorithm obtained more buckets with less dissimilar words. In contrast, Jaccard has been poorly performed since this algorithm created buckets with higher average distances (more than 0.6) inside buckets (see Figure 5.1-c).

Results of the rand index metric (as discussed in section 4.1.2) are shown in Figure 5.3 in descending order. The most remarkable feature of this diagram is that the five algorithms of Jaro-Winkler, Levenshtein, Weighting Table-based, Monge-Elkan and Smith-Waterman depict a similar good performance.

On the contrary, Sorensen-Dice, Cosine and Jaccard have lower rand index values compared to the rest of the algorithms, which represents their low efficiency. The Jaccard algorithm has the lowest rand index among all others, which shows that this algorithm is ill-suited for our application.

Furthermore, we observed that the Weighting Table-based algorithm is amongst the best, and it slightly outperforms the hybrid algorithm of Monge-Elkan, and it displays relatively good performance when compared to the rest of the algorithms.

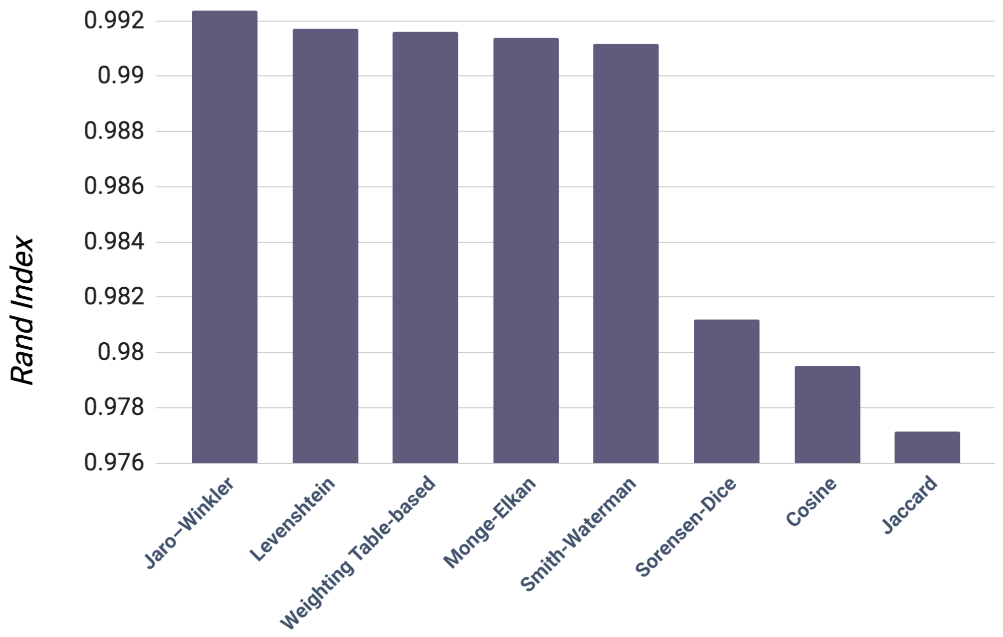


Figure 5.3 Results of computing the rand index for different similarity algorithms

In order to further investigate the efficiency of the algorithms, we inspected the buckets created by each algorithm and found out that for instance some very similar misspelled words like: "achievedto", "chievment" and ”unachieved" were put together in one bucket when we applied the WTB algorithm, whereas all other algorithms distinguished them by putting into separate buckets. Such performance of the WTB algorithm is useful for the disambiguation method as it undertakes the problem of misspelled or incomplete names.

Sorensen-Dice placed the two words "substration" and "altrations" in one bucket, despite being very different. Whilst WTB separated them and grouped "substration" with more similar words such as "subtraction", "sebtraction", etc. In the same case for Jaro-Winkler, "subtraction" and "sebtraction" were put together, nonetheless, words like "opulations" and "relutions" were also put in the same bucket despite their dissimilarities. Moreover, although the Levenshtein algorithm produced errors close to the WTB, it considered the mentioned words as not similar because of not having first and last similar characters, which makes the discriminatory power of this algorithm questionable in the course of the disambiguation problem where typos are frequent.

When the first letters of the two strings are completely different, Levenshtein algorithm distinguishes them by reporting a lower similarity value, whereas the WTB method detects the shift in letters and returns a higher value. As another example, WTB algorithm computes the similarity between the words "chmith" and "shmith" as: "0.9325", while the Levenshtein algorithm return the value of "0.833333313"; because of having two different letters of "c" and "s" at the beginning of each word. The WTB method also included some errors in its results. As an example, this method put the the word "waybright" with not much similar words such as "maybridge" and "unabridge" in the same bucket, whereas Levenshtein and Jaro-Winkler methods separated them and put "waybright" with more similar words like "albright" and "abright" in the same bucket.

K-means is extremely sensitive to the number of clusters and requires to set this parameter beforehand, whereas the hierarchical agglomerative clustering (HAC) creates sub-clusters by which we can compare similarity metrics at certain levels of determination.

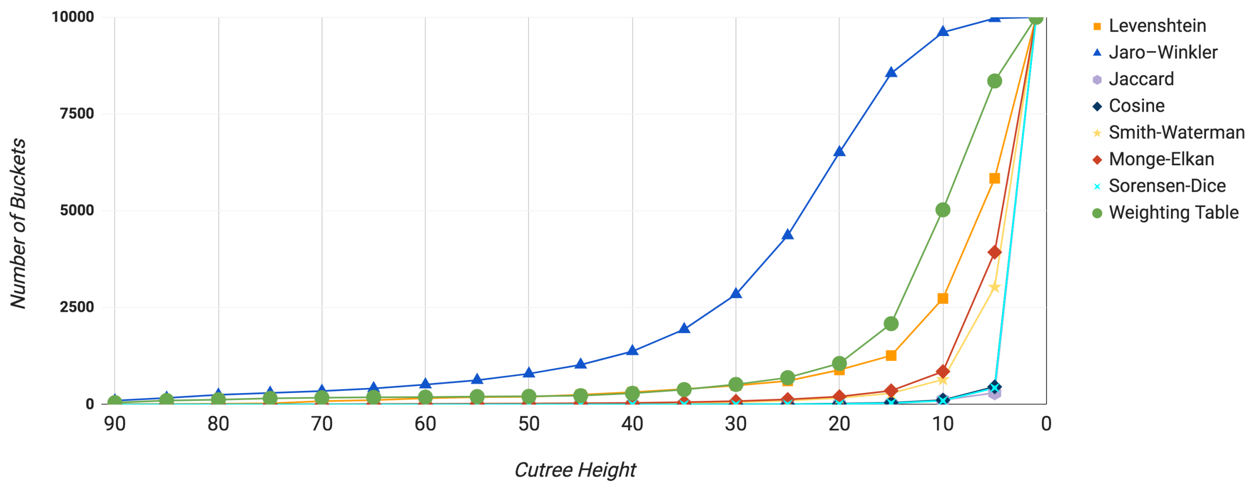


Figure 5.4 Number of buckets per cutree height for the HAC algorithm

HAC produces a hierarchical clustering tree in which clusters correspond to branches of the tree. In an effort to perform cluster identification (tree cutting or branch pruning), we pick a point (cutree height) and cut branches. This allows to measure the number of clusters produced at that level and perform analyses concerning the sensitivity of the similarity algorithm to a change in the threshold. Figure 5.4 shows the number of buckets created by each algorithm per height. It can be seen that the WTB is less sensitive because it produces continuous good results through the shift in threshold during the hierarchical clustering process.

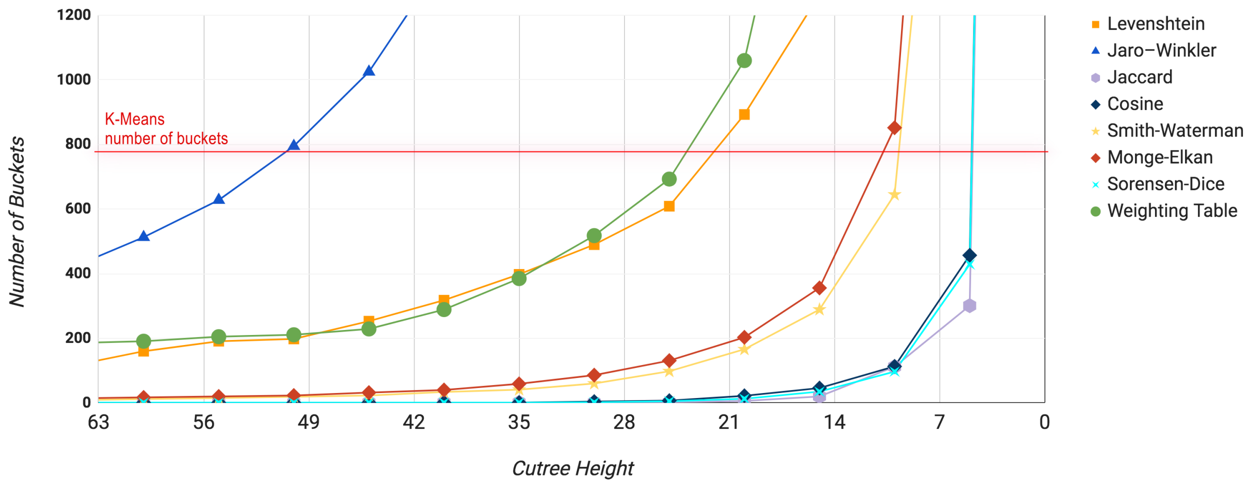


Figure 5.5 Number of buckets per cutree height for the HAC algorithm

We produced another HAC tree plot[1] in Figure 5.5 with a zoom into the previous one and put a horizontal line representing the number of clusters we used in the K-Means algorithm. The purpose of this second plot is to show the variation in number of buckets produced by each similarity algorithm with respect to the number of buckets determined in our previous analysis. In order to assess the quality of the algorithms, we must look at the shape of the curve close to the red line (number of buckets in K-Means algorithms as well as the gold standard). A good algorithm is the one with a moderate slope (not steep slope) when approaching the red line.

We can distinguish the algorithms in three groups with reference to their results depicted in Figure 5.5. In the first one Jaro-Winkler, WTB and Levenshtein have fairly similar results. These algorithms show a trend of gradual increase in number of buckets they generated with respect to reducing the height. This characteristic matches well to our problem since we are seeking an algorithm with lower sensitivity to the distance threshold.

The second group contained the Smith-Waterman and the Monge-Elkan with to some extent lower sensitivity to the changes in height than the remaining algorithms. Nonetheless, both algorithms are not good candidates for our problem since still they disclose high sensitivity to the threshold. The remaining algorithms of Cosine, Sorensend-Dice and Jaccard can be grouped together with significant difference from the rest of the algorithms. The most remarkable feature is that the Jaccard algorithm has a rapid change in number of buckets over the change in height which proves its eminent sensitivity concerning the threshold variation.

When using similarity measures in order to compare large volumes of data, many orders of magnitude faster algorithm can surpass quality limitations and improve application scalability. More

[1]Horizontal line is the number of clusters determined for the K-Means algorithm based on the gold standard

specifically, since the disambiguation task often deals with heavy computations, we can argue that this measure is as important as accuracy when performing the disambiguation. In cascade classification used to obtain disambiguation results on large amount of data, each classifier should produce a decision as fast as possible in order to speed up the overall process. The comparison results in terms of speed shown in Figure 5.6 indicate that our algorithm enables fast string similarity measure, thereby making it a better choice for very large data sets and real-time applications such as: spelling correction in search engines, genome data analysis, matching DNA sequences, browser fingerprint analysis and specially entity disambiguation.

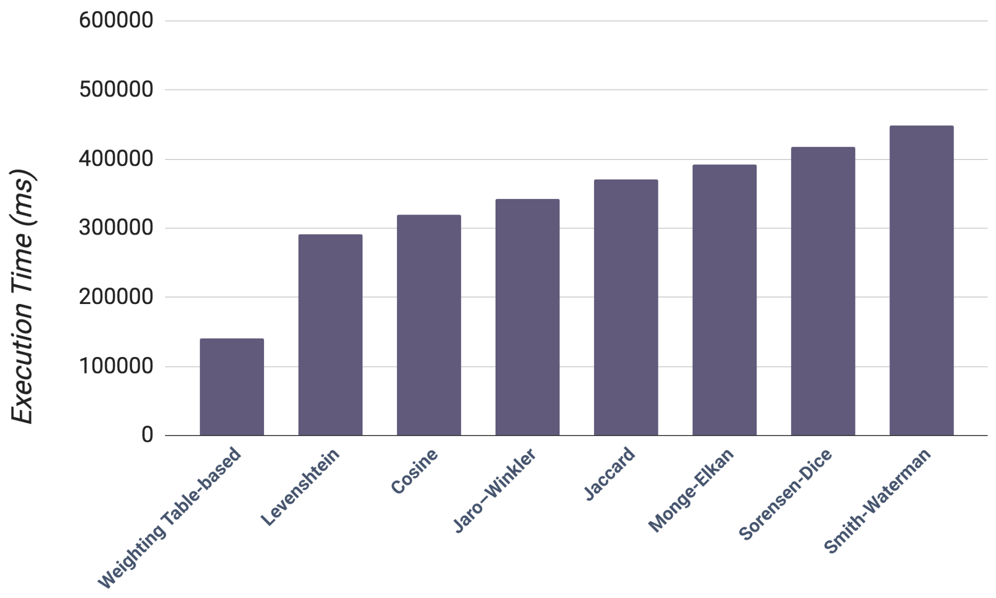


Figure 5.6 Total execution time

The closest competitor of WTB is Levenshtein, with the significant difference of practically twice the amount of total time to produce results. The slowest algorithm is Smith-Waterman with about three times the completion time compared to WTB. Since the application under study was developed in a collaboration with an industrial partner, there are limited resources available, therefore, any effort to reduce the computational cost and make the application speedy can have a large positive impact. In conclusion, our results depict that the Levenshtein, Jaro-Winkler, and WTB seem to be better choices among all, but the WTB is more suitable for the processing of large volumes of data due to its computational efficiency.

5.3 Results of Automating Cascade Structure

In this section, we present the results of the learning framework that is designed to automatically construct the cascade structure of our disambiguation application. The developed training methodology was tested on both WoS and SCOPUS datasets. For the WoS subset (large size), running with default parameters (explained in table 4.2 in the previous chapter), we were able to obtain the structure demonstrated in tables 5.2 and 5.3. These two tables show the configuration of the two blocks of the disambiguation method including entity aggregators and recursive aggregators. The training method used 160,000+ randomly selected records from the pre-processed data to perform the training process and create the configuration.

Table 5.2 Structure of the first disambiguation block - WoS subset (large size)

Entity Aggregator (EA)

Classifier	Classifier Type	WC Field	Blocking	Hit	Threshold
1	Set (contains any)	E-mail	True	False	–
2	Similarity distance	Surname	–	–	0.850000
3	Specialized (to forenames)	Forename	–	–	0.850000
4	Set (contains no)	Article Id	True	False	–

Recursive Aggregator (RA)

Compound Classifier	Classifier Type	WC Field	Blocking	Hit	Threshold
1	Similarity distance	Surname	–	–	0.979175
2	Specialized (to forenames)	Forename	–	–	0.969512
3	Set (contains no)	Article Id	True	False	–
	Set (contains any)	Fields	False	False	–
	Set (% of intersection)	BOW reference Ids	False	False	–
4	Set (contains any)	BOW keywords	False	False	–
	Set (% of intersection)	SubFields	False	False	0.446204
	Set (% of intersection)	Fields	True	False	0.300201
	Set (% of intersection)	BOW keywords	True	False	0.581463
	Set (% of intersection)	BOW reference Ids	True	False	0.251489
	Set (contains any)	Article Id & BOW reference Ids	True	False	–
5	Set (% of intersection)	BOW Affiliations	True	False	0.730993
	Specialized (naive Bayesian)	BOW addresses	True	False	0.300000
6	Set (contains any)	Article Id & BOW reference Ids	True	False	–
	Set (contains any)	BOW co-authors	False	False	–

Table 5.3 Structure of the second disambiguation block - WoS subset (large size)

Entity Aggregator (EA)

Classifier	Classifier Type	WC Field	Blocking	Hit	Threshold
1	Similarity distance	Surname	–	–	0.999999
2	Specialized (to forenames)	Forename	–	–	0.955555
3	Specialize (naive Bayesian)	BOW co-authors	False	False	–
	Set (% of intersection)	Fields	True	False	0.872208
	Set (contains no)	Article Id	True	False	–
	Set (% of intersection)	BOW keywords	True	False	0.781460
	Set (% of intersection)	BOW Address	True	False	0.730993
	Set (% of intersection)	BOW Affiliation	False	False	0.669699
4	Set (% of intersection)	BOW keywords	False	False	–
	Set (contains any)	BOW co-authors	False	False	–
5	Set (% of intersection)	Subfields	True	False	0.326542
	Set (% of intersection)	BOW keywords	False	False	0.654803
	Set (contains any)	BOW reference Ids & Article Id	True	False	–
	Set (contains any)	Article Id & BOW reference Ids	True	False	–

Recursive Aggregator (RA)

Classifier	Classifier Type	WC Field	Blocking	Hit	Threshold
1	Specialized (to forenames)	Forename	–	–	0.999999
2	Set (contains any)	Fields	True	False	0.884204
	Set (% of intersection)	BOW keywords	False	False	0.172180
	Set (% of intersection)	BOW Affiliations	True	False	0.741823
	Set (contains any)	BOW reference Ids	True	False	–
3	Set (contains no)	Article Id	True	False	–
	Set (contains any)	Fields	False	False	–
4	Set (% of intersection)	Subfields	False	False	–
	Set (% of intersection)	Journal Id	False	False	0.299081
	Set (contains any)	BOW reference Ids	False	False	–
5	Set (% of intersection)	Subfields	False	False	0.713096
	Specialized (naive Bayesian)	BOW co-authors	True	False	0.126721
	Set (contains any)	Article Id	False	False	–
6	Set (contains any)	BOW reference Ids	True	False	–
	Set (contains any)	Article Id & BOW reference Ids	True	False	–
	Set (contains any)	BOW co-authors	False	False	–

Table 5.4 Results of the automatic cascade approach (WoS)

			Automatically configured cascade approach		
	SVM-DBSCAN	**Manually configured cascade approach**	**FUNC 1**	**FUNC 2**	**FUNC 3**
Precision (%)	99.47	99.23	95.41	98.17	92.68
Recall (%)	73.88	83.21	80.65	81.33	74.23
F1 Score	0.848	0.905	0.874	0.889	0.824
Processing time (s)	1,817,819	2,724	4,601	4,982	4,214

Finally, we assessed the suggested training methodology by comparing the results of the disambiguation with the gold standard at hand and using standard quantitative metrics (explained in Chapter 4). Table 5.4 summarizes the results of disambiguation process on WoS dataset using the training method to configure the structure and thresholds of the classifiers when applying three different objective functions to the particle swarm optimizer. The first objective function considers the length of data while penalizing FP rates. The second one is mostly focused on strictly penalizing FN rates and the last one is a typical accuracy statistical measure.

It can be seen that the training method with an objective function (FUNC 2), which is focused on minimizing False Negative (FN) rates (as explained in chapter 4) obtains the best results compared to other objective functions, with the precision rate very similar to cascade approach with manually determined arrangements and thresholds.

To evaluate the proposed method using a different dataset, we ran another experiment on the second dataset SCOPUS which contained a smaller amount of records. The parameters of the learning algorithm (as shown in Table 4.2) were chosen the same as what we used for the WoS dataset, except the block size was reduced since the amount of data in SCOPUS was not as large as the WoS. From the SCOPUS, our training method used 50,000+ randomly selected records from the pre-processed data to perform the training process.

Results obtained for the second dataset SCOPUS (small size) are shown in Table 5.5 and 5.6. The results of the training methodology on this dataset (SCOPUS small size) are relatively close to the ones used in the previous experiment (WoS large size) as the method was able to put the computationally efficient classifiers such as similarity distance between Surname and Forename features in early steps of the aggregators. In addition, for the second block of the disambiguation, only five steps were determined which is shorter than the manual configuration and would translate into an increase in time-efficiency of the overall process.

Table 5.5 Structure of the first disambiguation block - SCOPUS subset (small size)

Entity Aggregator (EA)

Classifier	Classifier Type	WC Field	Blocking	Hit	Threshold
1	Set (contains any)	E-mail	True	False	–
2	Similarity distance	Surname	–	–	0.850000
3	Specialized (to forenames)	Forename	–	–	0.850000
4	Set (contains no)	Article Id	True	False	–

Recursive Aggregator (RA)

Compound Classifier	Classifier Type	WC Field	Blocking	Hit	Threshold
1	Similarity distance	Surname	–	–	0.834452
2	Specialized (to forenames)	Forename	–	–	0.999999
3	Set (contains no)	Article Id	True	False	–
	Set (% of intersection)	Fields	True	False	0.297863
4	Set (% of intersection)	BOW Address	True	False	0.747846
	Set (contains any)	Article Id & BOW reference Ids	True	False	–
	Specialized (naive Bayesian)	BOW addresses	True	False	0.300000
5	Set (contains any)	BOW keywords	False	False	–
	Set (% of intersection)	BOW Affiliations	True	False	0.601132
	Set (contains any)	BOW co-authors	False	False	–
	Set (% of intersection)	BOW keywords	True	False	0.410362
6	Set (contains any)	Article Id & BOW reference Ids	True	False	–

Table 5.6 Structure of the second disambiguation block - SCOPUS subset (small size)

Entity Aggregator (EA)

Classifier	Classifier Type	WC Field	Blocking	Hit	Threshold
1	Similarity distance	Surname	–	–	0.899999
2	Specialized (to forenames)	Forename	–	–	0.900000
3	Specialized (naive Bayesian)	BOW addresses	True	False	0.300000
	Set (% of intersection)	BOW keywords	True	False	0.423138
	Set (contains any)	Fields	True	False	0.936692
	Set (contains no)	Article Id	True	False	–
4	Set (% of intersection)	BOW Affiliation	False	False	0.747846
	Set (% of intersection)	Journal Id	False	False	0.244160
5	Set (contains any)	Fields	False	False	–
	Set (% of intersection)	Subfields	True	False	0.296438
	Set (contains any)	BOW reference Ids & Article Id	True	False	–
	Set (contains any)	Article Id & BOW reference Ids	True	False	–

Recursive Aggregator (RA)

Classifier	Classifier Type	WC Field	Blocking	Hit	Threshold
1	Specialized (to forenames)	Forename	–	–	0.988655
2	Set (contains no)	Article Id	True	False	–
	Set (contains any)	Fields	True	False	0.802608
	Set (% of intersection)	Subfields	False	False	–
	Set (contains any)	BOW reference Ids	True	False	–
	Set (% of intersection)	BOW keywords	False	False	0.209317
3	Set (contains no)	Article Id	True	False	–
	Set (contains any)	Fields	False	False	–
4	Set (% of intersection)	Journal Id	False	False	0.199687
	Set (contains any)	BOW reference Ids	False	False	–
5	Set (contains any)	Article Id	False	False	–
	Set (% of intersection)	Subfields	False	False	0.202357
	Set (contains any)	BOW reference Ids	False	False	–
	Specialized (naive Bayesian)	BOW co-authors	True	False	0.227499
	Set (contains any)	Article Id & BOW reference Ids	True	False	–
	Set (% of intersection)	BOW reference Ids	True	False	0.183102

Results of running the training method on the SCOPUS dataset are depicted in Table 5.7. We compared the global precison and recall obtained by our training method with the best strategy of manually configured disambiguation C_3 in (Voorons et al., 2017) (refer to the Table 3.7). As observed in the previous results, objective function (FUNC 2) outperforms the other ones by achieving better precision and recall rates.

It has to be mentioned that the difference between the results of manually configured structure and the automated one is due to the determination of the parameters of the training method that have been selected by trials and errors, and it can be optimized as a future work.

Table 5.7 Results of the automatic cascade approach (SCOPUS)

			Automatically configured cascade approach		
	SVM-DBSCAN	**Manually configured cascade approach (C_3)**	**FUNC 1**	**FUNC 2**	**FUNC 3**
Precision (%)	99.260	99.967	89.41	92.54	88.26
Recall (%)	96.893	97.436	85.51	88.71	73.26
F1 Score	0.980	0.986	0.874	0.905	0.800
Processing time (s)	1782	28	32	35	33

In both aggregators in each block of the results obtained with the two datasets, classifiers such as Surname and Forename appeared in early stages. This indicates that the training method is able to choose computationally efficient classifiers and make them appear sooner than the others in each step of the disambiguation. This makes the results of our training methodology analogous to the guidelines that were used to manually design the cascade structure of the application under study. This also shows an advantage of this methodology to make the application adjustable to many types of fields and run the disambiguation process on various types of record entities represented in various types of bibliographic databases.

The disambiguation configuration generated by the learning algorithm created a smaller structure with adjusted thresholds when operating on both selected datasets. The algorithm was able to produce less steps compared to the one used in (Voorons et al., 2017) with close results. This major difference may help to significantly reduce the processing time, which contributes to the overall objective of the cascade approach.

In summary, the training method was successfully applied with satisfactory results obtained on a very large dataset. The training process itself took about three days for the WoS dataset and 4 hours for the SCOPUS dataset to complete. The structure and thresholds determined by the algorithm are efficient enough to produce satisfactory results. The method was able to automatically adapt to different datasets since it uses the pre-processing stage results to train itself and it does not require changes in code, nor does it require the manipulation in the configuration of the disambiguation application. Final results of this step are put into an XML file that is included in *Appendix B* in the case of obtaining results for the WoS dataset.

It has to be mentioned that the objective of our research was not to find the best set of parameters for the optimization method but rather to automate the process of creating an efficient cascade structure. The impact of the algorithm parameters on the final results can be the subject of a future study.

CHAPTER 6 CONCLUSION

6.1 Summary of the Results

Analysis of the of scientific publications is a difficult task due to the presence of ambiguous information, the large size of the bibliographic databases and the noise. To achieve high quality results in a reasonable time, an efficient disambiguation application was developed. This application contains a method that does not rely on specific input information and its cascade structure yields a high computational efficiency even if some important features are missing in the input. The algorithm steps used in this method enables to scale sufficiently to large-scale bibliographic datasets. This research aimed to optimize such application in regards to several characteristics of its method.

One important element of the application under study is the use of a blocking method that reduced the burden of pair-wise comparisons significantly. This blocking method was applied by choosing a phonetic encoding algorithm, however, the impact of this choice on the quality of the results and speed efficiency of the algorithm was not studied. One of the goals of our research was to provide quantitative analysis and comparison between several commonly used phonetic encoding algorithms in order to provide guidelines for using them. As indicated by the experimental results, there is no single best phonetic algorithm available, and each one has advantages and disadvantages. The level of accuracy, efficiency, speed and computation cost must be considered when selecting an encoding algorithm. However, the most noticeable result is that the *Caverphone2* is slower than the others making it not a suitable choice for blocking in disambiguation algorithm.

Additionally, *NYSIIS* seems to be a very good compromise by having acceptable between/inside buckets distances, not many errors, not many comparisons and not having buckets with too many elements. The *NYSIIS* phonetic algorithm, nevertheless, suffers from relatively high amount of calculation time, but it is negligible compared to the computation time of the comparisons between records. Following this work, the disambiguation process uses the *NYSIIS* for the blocking method rather than the *DoubleMetaphone*.

On the other hand, the application utilizes many weak classifiers assembled in a cascade structure that function based on a string similarity metric to make a decision whether to link the entities or not. On that premise, the influence of the performance of such metric on the quality of the results and the speed efficiency of the algorithm was not taken into consideration. Clearly, results accuracy of the decisions made by weak classifiers operators can improve the overall effectiveness of the method itself. Therefore, first we introduced a new string similarity measure with the focus on improving the speed of calculations required to make a decision, then we performed a series of comparisons between this newly introduced metric and several commonly used metrics in order to evaluate their performances and choose the best one suitable for the application at hand.

Our results suggest that the Weighting Table-Based has less computational cost with appropriate rate of accuracy. When operating on large datasets, this algorithm has special character of time-efficient, which can be of an interest in applications with inevitable need of high processing speed such as disambiguation.

Additionally, although the application under study was successfully applied to different datasets with promising results, the capability of adapting it to specific characteristics and information content that may be found in different datasets introduced another challenge for our research. Therefore, the final step of our research was to increase the usability and adaptability of the application and make it flexible to various types of input datasets.

For this purpose, we developed a training methodology that can automatically generate a structure for WCs along with their value of thresholds irrespective of features presented to the application as the input. Our experiments based on a subset of the large-size Web of Science (WoS) and small-size SCOPUS bibliographic databases showed that the proposed method is able to identify the most important fields and their impact on the disambiguation process when operating on the existing disambiguation method.

Our training methodology used pre-processed data to train a model that generates the orderings and thresholds of the WCs, and obtains satisfactory results when cross-validated with the reference dataset and it is adjustable to any kind of bibliometric database regardless of the size and type of fields (no manipulation or specific information is required to run disambiguation when using our methodology). The proposed solution can make the application easily adaptable to other scenarios such as disambiguation of other entities.

It has to be mentioned that one limitation of our work is that the training process is time-consuming due to the fact that it performs an exhaustive pair-wise comparison of records in order to obtain results.

6.2 Future Work

Future work could include improving the application in regards to different aspects such as:

- To improve the *Weighting Table-based* metric and make it adjustable by determining its tuning parameters with the help of an automatic learning process.

- To test our training algorithm on different author name databases (e.g. Asian name databases).

- To improve the performance of the PSO optimizer using parameter optimization methods such as population-based adaptive optimization technique (Meissner et al., 2006).
- Choosing a set of optimal hyper-parameters for the training algorithm.

REFERENCES

F. Akhter, “A heuristic approach for minimum set cover problem,” *International Journal of Advanced Research in Artificial Intelligence (IJARAI)*, vol. 1, no. 4, pp. 40–45, 2015.

R. Baeza-Yates and G. Navarro, “A faster algorithm for approximate string matching,” in *Combinatorial Pattern Matching*. Springer, 1996, pp. 1–23.

R. Baeza-Yates, B. Ribeiro-Neto *et al.*, *Modern information retrieval*. ACM press New York, 1999, vol. 463.

B. Balsmeier, A. Chavosh, G.-C. Li, G. Fierro, K. Johnson, A. Kaulagi, D. O’Reagan, B. Yeh, and L. Fleming, “Automated disambiguation of us patent grants and applications,” 2015.

B. Berendt, A. Hotho, and G. Stumme, “Towards semantic web mining,” in *International Semantic Web Conference*. Springer, 2002, pp. 264–278.

M. Bilenko and R. J. Mooney, “Adaptive duplicate detection using learnable string similarity measures,” in *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2003, pp. 39–48.

R. Blanco, G. Ottaviano, and E. Meij, “Fast and space-efficient entity linking for queries,” in *Proceedings of the Eighth ACM International Conference on Web Search and Data Mining*. ACM, 2015, pp. 179–188.

L. K. Branting, “A comparative evaluation of name-matching algorithms,” in *Proceedings of the 9th international conference on Artificial intelligence and law*. ACM, 2003, pp. 224–232.

L. Breiman, “Bagging predictors,” *Machine learning*, vol. 24, no. 2, pp. 123–140, 1996.

M. Broilo and F. G. De Natale, “A stochastic approach to image retrieval using relevance feedback and particle swarm optimization,” *IEEE Transactions on Multimedia*, vol. 12, no. 4, pp. 267–277, 2010.

P. Bühlmann and B. Yu, “Boosting with the l 2 loss: regression and classification,” *Journal of the American Statistical Association*, vol. 98, no. 462, pp. 324–339, 2003.

E. Caron and N. J. van Eck, “Large scale author name disambiguation using rule-based scoring and clustering,” in *19th International Conference on Science and Technology Indicators. Context counts: Pathways to master big data and little data. CWTS-Leiden University Leiden*, 2014, pp. 79–86.

L. Cen, E. C. Dragut, L. Si, and M. Ouzzani, "Author disambiguation by hierarchical agglomerative clustering with adaptive stopping criterion," in *Proceedings of the 36th International ACM SIGIR conference on Research and development in information retrieval*. ACM, 2013, pp. 741–744.

S. Chapman, "Simmetrics," *URL http://sourceforge.net/projects/simmetrics/ SimMetrics is a Similarity Metric Library, eg from edit distance's (Levenshtein, Gotoh, Jaro etc) to other metrics,(eg Soundex, Chapman). Work provided by UK Sheffield University funded by (AKT) an IRC sponsored by EPSRC, grant number GR N*, vol. 15764, 2009.

M. Cheatham and P. Hitzler, "String similarity metrics for ontology alignment," in *International Semantic Web Conference*. Springer, 2013, pp. 294–309.

Z. Chen and H. Ji, "Collaborative ranking: A case study on entity linking," in *Proceedings of the Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 2011, pp. 771–781.

W.-C. Cheng and D.-M. Jhan, "A cascade classifier using adaboost algorithm and support vector machine for pedestrian detection," in *Systems, Man, and Cybernetics (SMC), 2011 IEEE International Conference on*. IEEE, 2011, pp. 1430–1435.

P. Christen, "A survey of indexing techniques for scalable record linkage and deduplication," *IEEE transactions on knowledge and data engineering*, vol. 24, no. 9, pp. 1537–1555, 2012.

K. J. Cios, W. Pedrycz, and R. W. Swiniarski, *Data mining methods for knowledge discovery*. Springer Science & Business Media, 2012, vol. 458.

W. Cohen, P. Ravikumar, and S. Fienberg, "A comparison of string metrics for matching names and records," in *Kdd workshop on data cleaning and object consolidation*, vol. 3, 2003, pp. 73–78.

W. W. Cohen and J. Richman, "Learning to match and cluster large high-dimensional data sets for data integration," in *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2002, pp. 475–480.

R. G. Cota, A. A. Ferreira, C. Nascimento, M. A. Gonçalves, and A. H. Laender, "An unsupervised heuristic-based hierarchical method for name disambiguation in bibliographic citations," *Journal of the American Society for Information Science and Technology*, vol. 61, no. 9, pp. 1853–1870, 2010.

A. Culotta, P. Kanani, R. Hall, M. Wick, and A. McCallum, "Author disambiguation using error-driven machine learning with a ranking loss function," in *Sixth International Workshop on Information Integration on the Web (IIWeb-07), Vancouver, Canada*, 2007.

N. Degtyarev and O. Seredin, “Comparative testing of face detection algorithms,” in *International Conference on Image and Signal Processing*. Springer, 2010, pp. 200–209.

P. DeRose, W. Shen, F. Chen, Y. Lee, D. Burdick, A. Doan, and R. Ramakrishnan, “Dblife: A community information management platform for the database research community,” in *CIDR*, 2007, pp. 169–172.

R. Eberhart and J. Kennedy, “Particle swarm optimization, proceeding of ieee international conference on neural network,” *Perth, Australia*, pp. 1942–1948, 1995.

B. Efron, “Bootstrap methods: another look at the jackknife annals of statistics 7: 1–26,” *View Article PubMed/NCBI Google Scholar*, 1979.

Y. Emek and A. Rosén, “Semi-streaming set cover,” *ACM Transactions on Algorithms (TALG)*, vol. 13, no. 1, p. 6, 2016.

Y. Freund, R. Schapire, and N. Abe, “A short introduction to boosting,” *Journal-Japanese Society For Artificial Intelligence*, vol. 14, no. 771-780, p. 1612, 1999.

J. Friedman, T. Hastie, R. Tibshirani *et al.*, “Additive logistic regression: a statistical view of boosting (with discussion and a rejoinder by the authors),” *The annals of statistics*, vol. 28, no. 2, pp. 337–407, 2000.

G. Gens and E. Levner, “Complexity of approximation algorithms for combinatorial problems: a survey,” *ACM SIGACT News*, vol. 12, no. 3, pp. 52–65, 1980.

C. L. Giles, H. Zha, and H. Han, “Name disambiguation in author citations using a k-way spectral clustering method,” in *Digital Libraries, 2005. JCDL’05. Proceedings of the 5th ACM/IEEE-CS Joint Conference on*. IEEE, 2005, pp. 334–343.

W. H. Gomaa and A. A. Fahmy, “A survey of text similarity approaches,” *International Journal of Computer Applications*, vol. 68, no. 13, 2013.

F. Grandoni, A. Gupta, S. Leonardi, P. Miettinen, P. Sankowski, and M. Singh, “Set covering with our eyes closed,” in *Foundations of Computer Science, 2008. FOCS’08. IEEE 49th Annual IEEE Symposium on*. IEEE, 2008, pp. 347–356.

L. Gravano, P. G. Ipeirotis, N. Koudas, and D. Srivastava, “Text joins for data cleansing and integration in an rdbms,” in *Data Engineering, 2003. Proceedings. 19th International Conference on*. IEEE, 2003, pp. 729–731.

R. Grundkiewicz and M. Junczys-Dowmunt, “The wiked error corpus: A corpus of corrective wikipedia edits and its application to grammatical error correction,” in *International Conference on Natural Language Processing*. Springer, 2014, pp. 478–490.

B. Han, P. Cook, and T. Baldwin, “Automatically constructing a normalisation dictionary for microblogs,” in *Proceedings of the 2012 joint conference on empirical methods in natural language processing and computational natural language learning*. Association for Computational Linguistics, 2012, pp. 421–432.

H. Han, L. Giles, H. Zha, C. Li, and K. Tsioutsiouliklis, “Two supervised learning approaches for name disambiguation in author citations,” in *Digital Libraries, 2004. Proceedings of the 2004 joint ACM/IEEE conference on*. IEEE, 2004, pp. 296–305.

D. Hood, “Caverphone: Phonetic matching algorithm,” *Technical Paper CTP060902, University of Otago, New Zealand*, 2002.

J. Huang, S. Ertekin, and C. L. Giles, “Efficient name disambiguation for large-scale databases,” in *European Conference on Principles of Data Mining and Knowledge Discovery*. Springer, 2006, pp. 536–544.

P. Jaccard, “The distribution of the flora in the alpine zone.” *New phytologist*, vol. 11, no. 2, pp. 37–50, 1912.

M. A. Jaro, “Advances in record-linkage methodology as applied to matching the 1985 census of tampa, florida,” *Journal of the American Statistical Association*, vol. 84, no. 406, pp. 414–420, 1989.

V. John, E. Trucco, and S. Ivekovic, “Markerless human articulated tracking using hierarchical particle swarm optimisation,” *Image and Vision Computing*, vol. 28, no. 11, pp. 1530–1547, 2010.

A. Karakasidis and V. S. Verykios, “Privacy preserving record linkage using phonetic codes,” in *Informatics, 2009. BCI’09. Fourth Balkan Conference in*. IEEE, 2009, pp. 101–106.

R. M. Karp, “Reducibility among combinatorial problems,” in *Complexity of computer computations*. Springer, 1972, pp. 85–103.

J. Kennedy, “The particle swarm: social adaptation of knowledge,” in *Evolutionary Computation, 1997., IEEE International Conference on*. IEEE, 1997, pp. 303–308.

G. Lan, G. W. DePuy, and G. E. Whitehouse, “An effective and simple heuristic for the set covering problem,” *European journal of operational research*, vol. 176, no. 3, pp. 1387–1403, 2007.

B. Larsen and C. Aone, “Fast and effective text mining using linear-time document clustering,” in *Proceedings of the fifth ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 1999, pp. 16–22.

D. Lee, B.-W. On, J. Kang, and S. Park, “Effective and scalable solutions for mixed and split citation problems in digital libraries,” in *Proceedings of the 2nd international workshop on Information quality in information systems*. ACM, 2005, pp. 69–76.

J.-J. Lee, P.-H. Lee, S.-W. Lee, A. Yuille, and C. Koch, “Adaboost for text detection in natural scene,” in *Document Analysis and Recognition (ICDAR), 2011 International Conference on*. IEEE, 2011, pp. 429–434.

V. I. Levenshtein, “Binary codes capable of correcting deletions, insertions, and reversals,” in *Soviet physics doklady*, vol. 10, no. 8, 1966, pp. 707–710.

L. Li, B. Roth, and C. Sporleder, “Topic models for word sense disambiguation and token-based idiom detection,” in *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, 2010, pp. 1138–1147.

H. Liu, A. Abraham, and A. E. Hassanien, “Scheduling jobs on computational grids using a fuzzy particle swarm optimization algorithm,” *Future Generation Computer Systems*, vol. 26, no. 8, pp. 1336–1343, 2010.

M. Liu, “Fingerprint classification based on adaboost learning from singularity features,” *Pattern Recognition*, vol. 43, no. 3, pp. 1062–1070, 2010.

Y. Liu, W. Li, Z. Huang, and Q. Fang, “A fast method based on multiple clustering for name disambiguation in bibliographic citations,” *Journal of the Association for Information Science and Technology*, vol. 66, no. 3, pp. 634–644, 2015.

G. Loomes, C. Starmer, and R. Sugden, “Observing violations of transitivity by experimental methods,” *Econometrica: Journal of the Econometric Society*, pp. 425–439, 1991.

V. Lopez, M. Fernández, E. Motta, and N. Stieler, “Poweraqua: Supporting users in querying and exploring the semantic web,” *Semantic Web*, vol. 3, no. 3, pp. 249–265, 2012.

J. MacQueen *et al.*, “Some methods for classification and analysis of multivariate observations,” in *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, vol. 1. Oakland, CA, USA., 1967, pp. 281–297.

B. Martins, “A supervised machine learning approach for duplicate detection over gazetteer records,” in *International Conference on GeoSpatial Sematics*. Springer, 2011, pp. 34–51.

M. Meissner, M. Schmuker, and G. Schneider, “Optimized particle swarm optimization (opso) and its application to artificial neural network training,” *BMC bioinformatics*, vol. 7, no. 1, p. 125, 2006.

X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen *et al.*, “Mllib: Machine learning in apache spark,” *Journal of Machine Learning Research*, vol. 17, no. 34, pp. 1–7, 2016.

P. Mitra, C. Murthy, and S. K. Pal, “Unsupervised feature selection using feature similarity,” *IEEE transactions on pattern analysis and machine intelligence*, vol. 24, no. 3, pp. 301–312, 2002.

P. Mitra, J. Kang, D. Lee, and B.-w. On, “Comparative study of name disambiguation problem using a scalable blocking-based framework,” in *Digital Libraries, 2005. JCDL’05. Proceedings of the 5th ACM/IEEE-CS Joint Conference on*. IEEE, 2005, pp. 344–353.

A. E. Monge, C. Elkan *et al.*, “The field matching problem: Algorithms and applications.” in *KDD*, 1996, pp. 267–270.

S. Montani and G. Leonardi, “Retrieval and clustering for supporting business process adjustment and analysis,” *Information Systems*, vol. 40, pp. 128–141, 2014.

M. H. Nadimi and M. Mosakhani, “A more accurate clustering method by using co-author social networks for author name disambiguation,” *Journal of Computing and Security*, vol. 1, no. 4, 2015.

“The soundex indexing system,” NAT, National Archives and Records Administration, 2007, may 2007.

A.-C. N. Ngomo and S. Auer, “Limes-a time-efficient approach for large-scale link discovery on the web of data,” *integration*, vol. 15, no. 3, 2011.

D. B. Nguyen, J. Hoffart, M. Theobald, and G. Weikum, “Aida-light: High-throughput named-entity disambiguation.” in *LDOW*. Citeseer, 2014.

K. Okuma, A. Taleghani, N. d. Freitas, J. J. Little, and D. G. Lowe, “A boosted particle filter: Multitarget detection and tracking,” *Computer Vision-ECCV 2004*, pp. 28–39, 2004.

S. Pandey, L. Wu, S. M. Guru, and R. Buyya, “A particle swarm optimization-based heuristic for scheduling workflow applications in cloud computing environments,” in *Advanced information networking and applications (AINA), 2010 24th IEEE international conference on*. IEEE, 2010, pp. 400–407.

P. Papadimitriou, A. Dasdan, and H. Garcia-Molina, “Web graph similarity for anomaly detection,” *Journal of Internet Services and Applications*, vol. 1, no. 1, pp. 19–30, 2010.

T. Pedersen, “A simple approach to building ensembles of naive bayesian classifiers for word sense disambiguation,” in *Proceedings of the 1st North American chapter of the Association for Computational Linguistics conference*. Association for Computational Linguistics, 2000, pp. 63–69.

L. Philips, “Hanging on the metaphone,” *Computer Language*, vol. 7, no. 12 (December), 1990.

D. Pinto, D. Vilariño, Y. Alemán, H. Gómez, N. Loya, and H. Jiménez-Salazar, “The soundex phonetic algorithm revisited for sms text representation,” in *International Conference on Text, Speech and Dialogue*. Springer, 2012, pp. 47–55.

H. J. Postel, “Die kölner phonetik. ein verfahren zur identifizierung von personennamen auf der grundlage der gestaltanalyse,” *IBM-Nachrichten*, vol. 19, pp. 925–931, 1969.

A. Rafae, A. Qayyum, M. M. Uddin, A. Karim, H. Sajjad, and F. Kamiran, “An unsupervised method for discovering lexical variations in roman urdu informal text.” in *EMNLP*, 2015, pp. 823–828.

P. Rajkovic and D. Jankovic, “Adaptation and application of daitch-mokotoff soundex algorithm on serbian names,” in *XVII Conference on Applied Mathematics*, vol. 12, 2007.

W. M. Rand, “Objective criteria for the evaluation of clustering methods,” *Journal of the American Statistical association*, vol. 66, no. 336, pp. 846–850, 1971.

G. Rätsch, T. Onoda, and K. R. Müller, “An improvement of adaboost to avoid overfitting,” in *Proc. of the Int. Conf. on Neural Information Processing*. Citeseer, 1998.

S. Y. Rhee and M. Mutwil, “Towards revealing the functions of all genes in plants,” *Trends in plant science*, vol. 19, no. 4, pp. 212–221, 2014.

G. Ridgeway, “Looking for lumps: Boosting and bagging for density estimation,” *Computational Statistics & Data Analysis*, vol. 38, no. 4, pp. 379–392, 2002.

P. Ristoski and P. Mika, “Enriching product ads with metadata from html annotations,” in *International Semantic Web Conference*. Springer, 2016, pp. 151–167.

T. F. Smith and M. S. Waterman, “Identification of common molecular subsequences,” *Journal of molecular biology*, vol. 147, no. 1, pp. 195–197, 1981.

T. Sørensen, “A method of establishing groups of equal amplitude in plant sociology based on similarity of species and its application to analyses of the vegetation on danish commons,” *Biol. Skr.*, vol. 5, pp. 1–34, 1948.

R. Speck and A.-C. N. Ngomo, “Ensemble learning for named entity recognition,” in *International Semantic Web Conference*. Springer, 2014, pp. 519–534.

Y. Sun, L. Lin, D. Tang, N. Yang, Z. Ji, and X. Wang, “Modeling mention, context and entity with neural networks for entity disambiguation.” in *IJCAI*, 2015, pp. 1333–1339.

J. Tang, A. C. Fong, B. Wang, and J. Zhang, “A unified probabilistic framework for name disambiguation in digital library,” *IEEE Transactions on Knowledge and Data Engineering*, vol. 24, no. 6, pp. 975–987, 2012.

V. I. Torvik and N. R. Smalheiser, “Author name disambiguation in medline,” *ACM Transactions on Knowledge Discovery from Data (TKDD)*, vol. 3, no. 3, p. 11, 2009.

V. I. Torvik, M. Weeber, D. R. Swanson, and N. R. Smalheiser, “A probabilistic similarity metric for medline records: A model for author name disambiguation,” *Journal of the American Society for information science and technology*, vol. 56, no. 2, pp. 140–158, 2005.

P. Treeratpituk and C. L. Giles, “Disambiguating authors in academic publications using random forests,” in *Proceedings of the 9th ACM/IEEE-CS joint conference on Digital libraries*. ACM, 2009, pp. 39–48.

P. Viola and M. Jones, “Robust real-time object detection,” *International Journal of Computer Vision*, vol. 4, no. 34–47, 2001.

M. Voorons, Y. Goussard, and E. Archambault, “A new cascaded approach for fast author name disambiguation,” *Manuscript in preparation*, 2017.

J. Wang, K. Berzins, D. Hicks, J. Melkers, F. Xiao, and D. Pinheiro, “A boosted-trees method for name disambiguation,” *Scientometrics*, vol. 93, no. 2, pp. 391–411, 2012.

B. H. Weinberg, “Bibliographic coupling: A review,” *Information Storage and Retrieval*, vol. 10, no. 5-6, pp. 189–196, 1974.

B. Xue, M. Zhang, and W. N. Browne, “Particle swarm optimization for feature selection in classification: A multi-objective approach,” *IEEE transactions on cybernetics*, vol. 43, no. 6, pp. 1656–1671, 2013.

Z. Xue, D. Yin, B. D. Davison, and B. Davison, "Normalizing microtext." *Analyzing Microtext*, vol. 11, p. 05, 2011.

M. Yang, J. Crenshaw, B. Augustine, R. Mareachen, and Y. Wu, "Adaboost-based face detection for embedded systems," *Computer Vision and Image Understanding*, vol. 114, no. 11, pp. 1116–1125, 2010.

J. Zobel and P. Dart, "Phonetic string matching: Lessons from information retrieval," in *Proceedings of the 19th annual international ACM SIGIR conference on Research and development in information retrieval*. ACM, 1996, pp. 166–172.

APPENDIX A CO-AUTHORSHIP

The results of this study have been submitted as follows:

- Banafsheh Mehri, Matthieu Voorons, Yves Goussard "A Comparison of Phonetic Encodings for Authors' Names Blocking", *Conference on Empirical Methods in Natural Language Processing, 2017*

  My contribution: Methodology, analysis, and paper writing.

- Banafsheh Mehri, Matthieu Voorons, Yves Goussard, Martin Trépanier "Weighting Table Based: A Fast String Similarity Measure", *The 16th International Semantic Web Conference, 2017*

  My contribution: Analysis, evaluation and paper writing.

APPENDIX B CASCADE CONFIGURATION XML FILE FOR WEB OF SCIENCE DATASET (LARGE SIZE)

Listing B.1 XML configuration file generated by training algorithm for WoS subset (large size)

```
< list >
  <desambiguation. classification . ScientistAggregator >
    <aggregatorType> Scientists aggregator </aggregatorType>
    <cc class ="desambiguation. classification . CascadeClassification ">
      < classifiers serialization ="custom">
        < unserializable parents />
        <com.google.common.collect.TreeMultimap>
          <default/>
          <com.google.common.collect.NaturalOrdering />
          <com.google.common.collect.Ordering_ ArbitraryOrdering >
            <uids class ="com.google.common.collect.MapMaker$ComputingMapAdapter"
                ↪ resolvesto="com.google.common.collect.ComputingConcurrentHashMap$ComputingSerializationProxy"
                ↪ serialization="custom">
              < unserializable parents />
              <com.google.common.collect.MapMakerInternalMap_ AbstractSerializationProxy >
                <default>
                  <concurrencyLevel>4</concurrencyLevel>
                  <expireAfterAccessNanos>0</expireAfterAccessNanos>
                  <expireAfterWriteNanos>0</expireAfterWriteNanos>
                  <maximumSize>1</maximumSize>
                  <keyEquivalence class ="com.google.common.base.Equivalence$Identity" />
                  <keyStrength class ="com.google.common.collect.MapMakerInternalMap$Strength">WEAK</keyStrength>
                  <removalListener
                      ↪ class ="com.google.common.collect.GenericMapMaker$NullListener">INSTANCE</removalListener>
                  <valueEquivalence class ="com.google.common.base.Equivalence$Equals"/>
                  <valueStrength class ="com.google.common.collect.MapMakerInternalMap$Strength">STRONG</valueStrength>
                </default>
              </com.google.common.collect.MapMakerInternalMap_ AbstractSerializationProxy >
              <com.google.common.collect.ComputingConcurrentHashMap_ComputingSerializationProxy>
                <default>
                  <computingFunction class ="com.google.common.collect.Ordering$ ArbitraryOrdering $1">
                    <counter>
                      <value>0</value>
                    </counter>
                    <outer class reference ="    ../../../../..    "/>
                  </computingFunction>
                </default>
                <int>0</int>
                <null />
              </com.google.common.collect.ComputingConcurrentHashMap_ComputingSerializationProxy>
            </uids>
          </com.google.common.collect.Ordering_ ArbitraryOrdering >
          <int>4</int>
          <int>1</int>
          <int>1</int>
          <desambiguation. classification . SetContainsAnyClassifier >
            < classifierType >Bag contains any classifier </ classifierType >
            <classifierName>AG   111    Email  classifier              </ classifierName>
            <blocking>true</blocking>
            <hit>false</ hit>
            < field1  class=" utils .SMField">EMAIL</field1>
            < field2  class=" utils .SMField">EMAIL</field2>
            <mix>false</mix>
            <converter  class=" utils . converter . StringConverterFactory $1"/>
          </desambiguation. classification . SetContainsAnyClassifier >
          <int>2</int>
          <int>1</int>
          <desambiguation. classification . SurnameClassifier>
            < classifierType > Classificateur sur les noms</ classifierType >
            <classifierName>AG   121    Noms                           </ classifierName>
            <blocking>true</blocking>
            <hit>false</ hit>
            <threshold>0.85</ threshold>
            <metric class="desambiguation. classification . metric .WeightingTableMetric">
              <maxDiff>2</maxDiff>
              < costShift >0.4</ costShift >
              <costDiff>0.4</ costDiff>
              < costFirst >1.7</ costFirst >
            </ metric>
            < field  class=" utils .SMField">SURNAME</field>
          </desambiguation. classification . SurnameClassifier>
          <int>3</int>
          <int>1</int>
          <desambiguation. classification . ForenameClassifier>
            < classifierType >Forename classifier </ classifierType >
            <classifierName>AG   131    forename                       </ classifierName>
            <blocking>false</blocking>
            <hit>false</ hit>
            <threshold>0.85</ threshold>
            <metric class="desambiguation. classification . metric .WeightingTableMetric"
                ↪ reference=" ../../ desambiguation. classification . SurnameClassifier/metric"/>
            < field  class=" utils .SMField">FORNAME</field>
          </desambiguation. classification . ForenameClassifier>
          <int>4</int>
          <int>1</int>
          <desambiguation. classification . SetContainsNoClassifier >
            < classifierType >Bag contains no classifier </ classifierType >
            <classifierName>AG   141    ID ID anti  classifier         </ classifierName>
            <blocking>true</blocking>
            <hit>false</ hit>
            < field1  class=" utils .SMField">ID</field1>
            < field2  class=" utils .SMField">ID</field2>
            <mix>false</mix>
            <converter  class=" utils . converter . StringConverterFactory $1"/>
          </desambiguation. classification . SetContainsNoClassifier >
        </com.google.common.collect.TreeMultimap>
      </ classifiers >
    </cc>
    <comparators>
      <desambiguation. classification .comparator.RecordComparator>
```

```
        < fieldToTest   class =" utils .SMField">EMAIL</fieldToTest>
        <operators>
          < string >==</string>
        </ operators >
        <valuesToTest>
          < string ></ string >
        </ valuesToTest>
      </desambiguation. classification .comparator.RecordComparator>
    </comparators>
  </desambiguation. classification . ScientistAggregator >
  <desambiguation. classification .RecurrentAggregator2>
    <aggregatorType>Recurrent aggregator</aggregatorType>
    <cc class ="desambiguation. classification . CascadeClassification ">
      < classifiers   serialization ="custom">
        < unserializable  parents />
        <com.google.common.collect.TreeMultimap>
          <default/>
          <com.google.common.collect.NaturalOrdering
                ↪ reference ="   ../../../../../   desambiguation. classification . ScientistAggregator "/>
          <com.google.common.collect.Ordering_ ArbitraryOrdering
                ↪ reference ="   ../../../../../   desambiguation. classification . ScientistAggregator "/>
          <int>6</int>
          <int>1</int>
          <int>1</int>
          <desambiguation. classification . SurnameClassifier >
            < classifierType > Classificateur  sur  les  noms</ classifierType >
            <classifierName>AG   211     Noms                            </ classifierName >
            <blocking>true</blocking>
            <hit> false </ hit >
            <threshold>0.979175</threshold>
            <metric class ="desambiguation. classification . metric . WeightingTableMetric"
                ↪ reference ="   ../../../../../../   desambiguation. classification . ScientistAggregator "/>
            < field  class =" utils .SMField">SURNAME</field>
          </desambiguation. classification . SurnameClassifier >
          <int>2</int>
          <int>1</int>
          <desambiguation. classification . ForenameClassifier >
            < classifierType >Forename classifier </ classifierType >
            <classifierName>AG   221     forename                        </ classifierName >
            <blocking> false </blocking>
            <hit> false </ hit >
            <threshold>0.969512</threshold>
            <metric class ="desambiguation. classification . metric . WeightingTableMetric"
                ↪ reference ="   ../../../../../../   desambiguation. classification . ScientistAggregator "/>
            < field  class =" utils .SMField">FORNAME</field>
          </desambiguation. classification . ForenameClassifier >
          <int>3</int>
          <int>3</int>
          <desambiguation. classification . SetContainsNoClassifier >
            < classifierType >Bag contains  no  classifier </ classifierType >
            <classifierName>AG   231     ID ID  anti   classifier        </ classifierName >
            <blocking>true</blocking>
            <hit> false </ hit >
            < field1  class =" utils .SMField">ID</field1>
            < field2  class =" utils .SMField">ID</field2>
  <mix>false</mix>
  <converter  class =" utils . converter . StringConverterFactory $1"/>
</desambiguation.  classification  . SetContainsNoClassifier >
<desambiguation.  classification  . BagContainsAnyClassifier>
  < classifierType >Any ref</ classifierType >
  <classifierName>Any ref</ classifierName >
  <blocking> false </blocking>
  <hit> false </ hit >
  < field1  class =" utils .SMField">BOW_REFERENCES</field1>
  < field2  class =" utils .SMField">BOW_REFERENCES</field2>
  <mix>false</mix>
  <converter  class =" utils . converter . StringConverterFactory $1"/>
</desambiguation.  classification  . BagContainsAnyClassifier>
<desambiguation.  classification  . BagContainsAnyClassifier>
  < classifierType >Any  field</ classifierType >
  <classifierName>Any  field</ classifierName >
  <blocking> false </blocking>
  <hit> false </ hit >
  < field1  class =" utils .  ScientistField  ">FIELDS</field1>
  < field2  class =" utils .  ScientistField  ">FIELDS</field2>
  <mix>false</mix>
</desambiguation.  classification  . BagContainsAnyClassifier>
<int>4</int>
<int>6</int>
<desambiguation.  classification  . BagCountClassifier>
  < classifierType >Bag counter   classifier  </ classifierType >
  <classifierName>AG   243     Fields   classifier                 </ classifierName >
  <blocking> false </blocking>
  <hit> false </ hit >
  <threshold>0.300201</threshold>
  < field1  class =" utils .  ScientistField  ">FIELDS</field1>
  < field2  class =" utils .  ScientistField  ">FIELDS</field2>
  <mix>false</mix>
</desambiguation.  classification  . BagCountClassifier>
<desambiguation.  classification  . SetContainsAnyClassifier >
  < classifierType >Bag contains   classifier  </ classifierType >
  <classifierName>AG   246     ID References   classifier         </ classifierName >
  <blocking>true</blocking>
  <hit> false </ hit >
  < field1  class =" utils .SMField">ID</field1>
  < field2  class =" utils .SMField">BOW_REFERENCES</field2>
  <mix>false</mix>
  <converter  class =" utils . converter . StringConverterFactory $1"/>
</desambiguation.  classification  . SetContainsAnyClassifier >
<desambiguation.  classification  . BagCountClassifier>
  < classifierType >Bag counter   classifier  </ classifierType >
  <classifierName>AG   244     Keywords  classifier              </ classifierName >
  <blocking> false </blocking>
  <hit> false </ hit >
  <threshold>0.581463</threshold>
  < field1  class =" utils .SMField">BOW_KEYWORD</field1>
  < field2  class =" utils .SMField">BOW_KEYWORD</field2>
  <mix>false</mix>
  <converter  class =" utils . converter . StringConverterFactory $3"/>
</desambiguation.  classification  . BagCountClassifier>
```

```
<desambiguation. classification . BagCountClassifier>
  < classifierType >Bag contains  classifier </ classifierType >
  <classifierName>AG  245    References  classifier          </ classifierName>
  <blocking>false</blocking>
  <hit>false</hit>
  <threshold>0.251489</threshold>
  <field1  class=" utils .SMField">BOW_REFERENCES</field1>
  <field2  class=" utils .SMField">BOW_REFERENCES</field2>
  <mix>false</mix>
  <converter  class=" utils . converter . StringConverterFactory $1"/>
</desambiguation. classification . BagCountClassifier>
<desambiguation. classification . BagContainsAnyClassifier>
  < classifierType >Bag contains  classifier </ classifierType >
  <classifierName>AG  241    Keywords  classifier             </ classifierName>
  <blocking>false</blocking>
  <hit>false</hit>
  <field1  class=" utils .SMField">BOW_KEYWORD</field1>
  <field2  class=" utils .SMField">BOW_KEYWORD</field2>
  <mix>false</mix>
  <converter  class=" utils . converter . StringConverterFactory $3"/>
</desambiguation. classification . BagContainsAnyClassifier>
<desambiguation. classification . BagCountClassifier>
  < classifierType >Bag counter  classifier </ classifierType >
  <classifierName>AG  242    Subfields  classifier           </ classifierName>
  <blocking>false</blocking>
  <hit>false</hit>
  <threshold>0.446204</threshold>
  <field1  class=" utils . ScientistField ">SUBFIELDS</field1>
  <field2  class=" utils . ScientistField ">SUBFIELDS</field2>
  <mix>false</mix>
</desambiguation. classification . BagCountClassifier>
<int>5</int>
<int>3</int>
<desambiguation. classification . BagCountClassifier>
  < classifierType >Bag count  classifier </ classifierType >
  <classifierName>AG  251    Affiliation  classifier         </ classifierName>
  <blocking>true</blocking>
  <hit>false</hit>
  <threshold>0.730993</threshold>
  <field1  class=" utils . ScientistField ">AFFILIATION_BOW</field1>
  <field2  class=" utils . ScientistField ">AFFILIATION_BOW</field2>
  <mix>false</mix>
</desambiguation. classification . BagCountClassifier>
<desambiguation. classification . NaiveBayesianClassifier>
  < classifierType >Naive Bayesian  classifier </ classifierType >
  <classifierName>Addresses  classifier </ classifierName>
  <blocking>true</blocking>
  <hit>false</hit>
  <threshold>0.3</ threshold>
  < field >BOW_ADDRESS</field>
  <statsFilename>/code/ java / dis / data /run2/ processed_data /stats_BOW_KEYWORD.csv</statsFilename>
  <converter  class=" utils . converter . StringConverterFactory $1"/>
</desambiguation. classification . NaiveBayesianClassifier>
<desambiguation. classification . NaiveBayesianClassifier>
  < classifierType >Naive Bayesian  classifier </ classifierType >
```

```
          <classifierName>Addresses  classifier  </ classifierName >
          <blocking>true</blocking>
          <hit> false </ hit >
          <threshold>0.126721</threshold>
          < field >COAUTHORS</field>
          <statsFilename>/code/ java / dis / data /run2/ processed_data / stats_FULL_NAME.csv</statsFilename>
          <converter  class =" utils . converter . StringConverterFactory $1"/>
        </desambiguation. classification  . NaiveBayesianClassifier >
        <int>6</int>
        <int>2</int>
        <desambiguation. classification  . SetContainsAnyClassifier >
          < classifierType >Bag contains  classifier </ classifierType >
          <classifierName>AG   261    ID References  classifier        </ classifierName >
          <blocking>true</blocking>
          <hit> false </ hit >
          < field1  class =" utils .SMField">ID</field1>
          < field2  class =" utils .SMField">BOW_REFERENCES</field2>
          <mix>false</mix>
          <converter  class =" utils . converter . StringConverterFactory $1"/>
        </desambiguation. classification  . SetContainsAnyClassifier >
        <desambiguation. classification  . BagContainsAnyClassifier>
          < classifierType >Bag contains  classifier </ classifierType >
          <classifierName>AG   262    Co authors  classifier           </ classifierName >
          <blocking> false </blocking>
          <hit> false </ hit >
          < field1  class =" utils .SMField">COAUTHORS</field1>
          < field2  class =" utils .SMField">COAUTHORS</field2>
          <mix>false</mix>
          <converter  class =" utils . converter . StringConverterFactory $1"/>
        </desambiguation. classification  . BagContainsAnyClassifier>
      </com.google.common.collect.TreeMultimap>
    </ classifiers >
  </cc>
</desambiguation. classification  .RecurrentAggregator2>
<desambiguation. classification  . ScientistAggregator >
  <aggregatorType> Scientists  aggregator</aggregatorType>
  <cc class ="desambiguation. classification  . CascadeClassification ">
    < classifiers   serialization ="custom">
      < unserializable  parents />
      <com.google.common.collect.TreeMultimap>
        <default/>
        <com.google.common.collect.NaturalOrdering
             ↪ reference ="   ../../../../../   desambiguation. classification  . ScientistAggregator "/>
        <com.google.common.collect.Ordering_ ArbitraryOrdering
             ↪ reference ="   ../../../../../   desambiguation. classification  . ScientistAggregator "/>
        <int>5</int>
        <int>1</int>
        <int>1</int>
        <desambiguation. classification  . SurnameClassifier >
          < classifierType > Classificateur  sur  les  noms</ classifierType >
          <classifierName>AG   311    Noms                   </ classifierName >
          <blocking>true</blocking>
          <hit> false </ hit >
          <threshold>0.999999</threshold>
  <metric class="desambiguation. classification .metric.WeightingTableMetric"
      ↪ reference="   ../../../../../../   desambiguation. classification . ScientistAggregator "/>
  <field class=" utils .SMField">SURNAME</field>
</desambiguation. classification . SurnameClassifier>
<int>2</int>
<int>1</int>
<desambiguation. classification . FornameClassifier>
  < classifierType >Forename classifier </ classifierType >
  <classifierName>AG  321   forename                </ classifierName>
  <blocking> false </blocking>
  <hit> false </ hit>
  <threshold>0.955555</threshold>
  <metric class="desambiguation. classification .metric.WeightingTableMetric"
      ↪ reference="   ../../../../../../   desambiguation. classification . ScientistAggregator "/>
  <field class=" utils .SMField">FORNAME</field>
</desambiguation. classification . FornameClassifier>
<int>3</int>
<int>5</int>
<desambiguation. classification . BagCountClassifier>
  < classifierType >Bag counter  classifier </ classifierType >
  <classifierName>AG  334   Keywords classifier      </ classifierName>
  <blocking>true</blocking>
  <hit> false </ hit>
  <threshold>0.78146</threshold>
  <field1 class=" utils .SMField">BOW_KEYWORD</field1>
  <field2 class=" utils .SMField">BOW_KEYWORD</field2>
  <mix>false</mix>
  <converter class=" utils . converter . StringConverterFactory $3"/>
</desambiguation. classification . BagCountClassifier>
<desambiguation. classification . SetContainsAnyClassifier>
  < classifierType >Bag contains  classifier </ classifierType >
  <classifierName>AG  333   ID References  classifier    </ classifierName>
  <blocking>true</blocking>
  <hit> false </ hit>
  <field1 class=" utils .SMField">ID</field1>
  <field2 class=" utils .SMField">BOW_REFERENCES</field2>
  <mix>false</mix>
  <converter class=" utils . converter . StringConverterFactory $1"/>
</desambiguation. classification . SetContainsAnyClassifier>
<desambiguation. classification . BagCountClassifier>
  < classifierType >Bag counter  classifier </ classifierType >
  <classifierName>AG  332   Fields  classifier       </ classifierName>
  <blocking> false </blocking>
  <hit> false </ hit>
  <threshold>0.872208</threshold>
  <field1 class=" utils . ScientistField ">FIELDS</field1>
  <field2 class=" utils . ScientistField ">FIELDS</field2>
  <mix>false</mix>
</desambiguation. classification . BagCountClassifier>
<desambiguation. classification . BagContainsAnyClassifier>
  < classifierType >Bag contains  classifier </ classifierType >
  <classifierName>AG  331   Co authors  classifier     </ classifierName>
  <blocking> false </blocking>
  <hit> false </ hit>
  <field1 class=" utils .SMField">COAUTHORS</field1>
  <field2  class=" utils .SMField">COAUTHORS</field2>
  <mix>false</mix>
  <converter  class=" utils . converter . StringConverterFactory $1"/>
</desambiguation. classification  . BagContainsAnyClassifier>
<desambiguation. classification  . BagCountClassifier>
  < classifierType >Bag count  classifier </ classifierType >
  <classifierName>AG   335     Affiliation    classifier          </ classifierName>
  <blocking>true</blocking>
  <hit>false</hit>
  <threshold>0.669699</threshold>
  <field1  class=" utils . ScientistField ">AFFILIATION_BOW</field1>
  <field2  class=" utils . ScientistField ">AFFILIATION_BOW</field2>
  <mix>false</mix>
</desambiguation. classification  . BagCountClassifier>
<int>4</int>
<int>2</int>
<desambiguation. classification  . BagContainsAnyClassifier>
  < classifierType >Bag contains  classifier </ classifierType >
  <classifierName>AG   341     Keywords classifier              </ classifierName>
  <blocking>false</blocking>
  <hit>false</hit>
  <field1  class=" utils .SMField">BOW_KEYWORD</field1>
  <field2  class=" utils .SMField">BOW_KEYWORD</field2>
  <mix>false</mix>
  <converter  class=" utils . converter . StringConverterFactory $3"/>
</desambiguation. classification  . BagContainsAnyClassifier>
<desambiguation. classification  . BagContainsAnyClassifier>
  < classifierType >Bag contains  classifier </ classifierType >
  <classifierName>AG   342     Co authors   classifier             </ classifierName>
  <blocking>false</blocking>
  <hit>false</hit>
  <field1  class=" utils .SMField">COAUTHORS</field1>
  <field2  class=" utils .SMField">COAUTHORS</field2>
  <mix>false</mix>
  <converter  class=" utils . converter . StringConverterFactory $1"/>
</desambiguation. classification  . BagContainsAnyClassifier>
<int>5</int>
<int>4</int>
<desambiguation. classification  . BagCountClassifier>
  < classifierType >Bag counter  classifier </ classifierType >
  <classifierName>AG   351     Subfields   classifier             </ classifierName>
  <blocking>false</blocking>
  <hit>false</hit>
  <threshold>0.326542</threshold>
  <field1  class=" utils . ScientistField ">SUBFIELDS</field1>
  <field2  class=" utils . ScientistField ">SUBFIELDS</field2>
  <mix>false</mix>
</desambiguation. classification  . BagCountClassifier>
<desambiguation. classification  . BagCountClassifier>
  < classifierType >Bag counter  classifier </ classifierType >
  <classifierName>AG   352     Keywords classifier              </ classifierName>
  <blocking>false</blocking>
  <hit>false</hit>
  <threshold>0.654803</threshold>
  <field1  class=" utils .SMField">BOW_KEYWORD</field1>
```

```
          <field2  class=" utils .SMField">BOW_KEYWORD</field2>
          <mix>false</mix>
          <converter  class=" utils . converter . StringConverterFactory $3"/>
        </desambiguation. classification  . BagCountClassifier>
        <desambiguation. classification  . SetContainsAnyClassifier >
          < classifierType >Bag contains  classifier  </ classifierType >
          <classifierName>AG  353    References ID  classifier        </ classifierName>
          <blocking>true</blocking>
          <hit> false </ hit>
          <field1  class=" utils .SMField">BOW_REFERENCES</field1>
          <field2  class=" utils .SMField">ID</field2>
          <mix>false</mix>
          <converter  class=" utils . converter . StringConverterFactory $1"/>
        </desambiguation. classification  . SetContainsAnyClassifier >
        <desambiguation. classification  . SetContainsAnyClassifier >
          < classifierType >Bag contains  classifier  </ classifierType >
          <classifierName>AG  354    ID References  classifier        </ classifierName>
          <blocking>true</blocking>
          <hit> false </ hit>
          <field1  class=" utils .SMField">ID</field1>
          <field2  class=" utils .SMField">BOW_REFERENCES</field2>
          <mix>false</mix>
          <converter  class=" utils . converter . StringConverterFactory $1"/>
        </desambiguation. classification  . SetContainsAnyClassifier >
      </com.google.common.collect.TreeMultimap>
    </  classifiers  >
  </cc>
  <comparators>
    <desambiguation. classification  .comparator.RecordComparator>
      < fieldToTest  class=" utils .SMField">EMAIL</fieldToTest>
      <operators>
        < string >!=</ string >
      </ operators>
      <valuesToTest>
        < string ></ string >
      </ valuesToTest>
    </desambiguation. classification  .comparator.RecordComparator>
  </comparators>
</desambiguation. classification  . ScientistAggregator >
<desambiguation. classification  .RecurrentAggregator2>
  <aggregatorType>Recurrent  aggregator</aggregatorType>
  <cc  class="desambiguation. classification  . CascadeClassification  ">
    <  classifiers    serialization  ="custom">
      < unserializable   parents />
      <com.google.common.collect.TreeMultimap>
        <default/>
        <com.google.common.collect.NaturalOrdering
            ↪  reference="   ../../../../../   desambiguation. classification  . ScientistAggregator "/>
        <com.google.common.collect.Ordering_  ArbitraryOrdering
            ↪  reference="   ../../../../../   desambiguation. classification  . ScientistAggregator "/>
        <int>6</int>
        <int>1</int>
        <int>1</int>
        <desambiguation. classification  . SurnameClassifier >
          < classifierType > Classificateur  sur  les  noms</ classifierType >
  <classifierName>AG  411   Noms                    </classifierName>
  <blocking>true</blocking>
  <hit>false</hit>
  <threshold>0.999999</threshold>
  <metric class="desambiguation. classification .metric.WeightingTableMetric"
      ↪ reference="   ../../../../../../    desambiguation. classification . ScientistAggregator "/>
  <field class=" utils .SMField">SURNAME</field>
</desambiguation. classification . SurnameClassifier>
<int>2</int>
<int>4</int>
<desambiguation. classification . SetContainsAnyClassifier>
  < classifierType >Bag contains  classifier </ classifierType >
  <classifierName>AG  424   ID References  classifier     </classifierName>
  <blocking>true</blocking>
  <hit>false</hit>
  <field1 class=" utils .SMField">ID</field1>
  <field2 class=" utils .SMField">BOW_REFERENCES</field2>
  <mix>false</mix>
  <converter class=" utils . converter . StringConverterFactory $1"/>
</desambiguation. classification . SetContainsAnyClassifier>
<desambiguation. classification . BagCountClassifier>
  < classifierType >Bag counter  classifier </ classifierType >
  <classifierName>AG  422   Keywords classifier          </classifierName>
  <blocking>false</blocking>
  <hit>false</hit>
  <threshold>0.17218</threshold>
  <field1 class=" utils .SMField">BOW_KEYWORD</field1>
  <field2 class=" utils .SMField">BOW_KEYWORD</field2>
  <mix>false</mix>
  <converter class=" utils . converter . StringConverterFactory $3"/>
</desambiguation. classification . BagCountClassifier>
<desambiguation. classification . BagCountClassifier>
  < classifierType >Bag counter  classifier </ classifierType >
  <classifierName>AG  421   Fields  classifier           </classifierName>
  <blocking>false</blocking>
  <hit>false</hit>
  <threshold>0.884204</threshold>
  <field1 class=" utils . ScientistField ">FIELDS</field1>
  <field2 class=" utils . ScientistField ">FIELDS</field2>
  <mix>false</mix>
</desambiguation. classification . BagCountClassifier>
<desambiguation. classification . BagCountClassifier>
  < classifierType >Bag count  classifier </ classifierType >
  <classifierName>AG  423   Affiliation  classifier      </classifierName>
  <blocking>true</blocking>
  <hit>false</hit>
  <threshold>0.741823</threshold>
  <field1 class=" utils . ScientistField ">AFFILIATION_BOW</field1>
  <field2 class=" utils . ScientistField ">AFFILIATION_BOW</field2>
  <mix>false</mix>
</desambiguation. classification . BagCountClassifier>
<int>3</int>
<int>2</int>
<desambiguation. classification . BagContainsAnyClassifier>
  < classifierType >Bag contains  classifier </ classifierType >
  <classifierName>AG 432 References classifier </classifierName>
  <blocking>false</blocking>
  <hit>false</hit>
  <field1 class="utils. ScientistField ">FIELDS</field1>
  <field2 class="utils. ScientistField ">FIELDS</field2>
  <mix>false</mix>
</desambiguation. classification .BagContainsAnyClassifier>
<desambiguation. classification .SetContainsNoClassifier>
  <classifierType>Bag contains no classifier </classifierType>
  <classifierName>AG 431 ID ID anti classifier </classifierName>
  <blocking>true</blocking>
  <hit>false</hit>
  <field1 class="utils.SMField">ID</field1>
  <field2 class="utils.SMField">ID</field2>
  <mix>false</mix>
  <converter class="utils.converter.StringConverterFactory $1"/>
</desambiguation. classification .SetContainsNoClassifier>
<int>4</int>
<int>3</int>
<desambiguation. classification .BagContainsAnyClassifier>
  <classifierType>Bag contains classifier </classifierType>
  <classifierName>AG 443 References classifier </classifierName>
  <blocking>false</blocking>
  <hit>false</hit>
  <field1 class="utils.SMField">BOW_REFERENCES</field1>
  <field2 class="utils.SMField">BOW_REFERENCES</field2>
  <mix>false</mix>
  <converter class="utils.converter.StringConverterFactory $1"/>
</desambiguation. classification .BagContainsAnyClassifier>
<desambiguation. classification .BagContainsAnyClassifier>
  <classifierType>Bag contains classifier </classifierType>
  <classifierName>AG 441 References classifier </classifierName>
  <blocking>false</blocking>
  <hit>false</hit>
  <field1 class="utils. ScientistField ">SUBFIELDS</field1>
  <field2 class="utils. ScientistField ">SUBFIELDS</field2>
  <mix>false</mix>
</desambiguation. classification .BagContainsAnyClassifier>
<desambiguation. classification .BagCountClassifier>
  <classifierType>Bag counter classifier </classifierType>
  <classifierName>AG 442 Source ID classifier </classifierName>
  <blocking>false</blocking>
  <hit>false</hit>
  <threshold>0.299081</threshold>
  <field1 class="utils. ScientistField ">SOURCE_ID</field1>
  <field2 class="utils. ScientistField ">SOURCE_ID</field2>
  <mix>false</mix>
</desambiguation. classification .BagCountClassifier>
<int>5</int>
<int>2</int>
<desambiguation. classification .BagContainsAnyClassifier>
  <classifierType>Bag counter classifier </classifierType>
  <classifierName>AG 452 Source ID classifier </classifierName>
  <blocking>false</blocking>
  <hit>false</hit>
```

```
                <field1 class="utils.SMField">ID</field1>
                <field2 class="utils.SMField">ID</field2>
                <mix>false</mix>
                <converter class="utils.converter.StringConverterFactory$1"/>
              </desambiguation.classification.BagContainsAnyClassifier>
              <desambiguation.classification.BagCountClassifier>
                <classifierType>Bag counter classifier</classifierType>
                <classifierName>AG 451 Subfields classifier          </classifierName>
                <blocking>false</blocking>
                <hit>false</hit>
                <threshold>0.713096</threshold>
                <field1 class="utils.ScientistField">SUBFIELDS</field1>
                <field2 class="utils.ScientistField">SUBFIELDS</field2>
                <mix>false</mix>
              </desambiguation.classification.BagCountClassifier>
              <int>6</int>
              <int>3</int>
              <desambiguation.classification.BagContainsAnyClassifier>
                <classifierType>Bag contains classifier</classifierType>
                <classifierName>AG 463 Co authors classifier          </classifierName>
                <blocking>false</blocking>
                <hit>false</hit>
                <field1 class="utils.SMField">COAUTHORS</field1>
                <field2 class="utils.SMField">COAUTHORS</field2>
                <mix>false</mix>
                <converter class="utils.converter.StringConverterFactory$1"/>
              </desambiguation.classification.BagContainsAnyClassifier>
              <desambiguation.classification.SetContainsAnyClassifier>
                <classifierType>Bag count classifier</classifierType>
                <classifierName>AG 461 References classifier          </classifierName>
                <blocking>true</blocking>
                <hit>false</hit>
                <field1 class="utils.SMField">BOW_REFERENCES</field1>
                <field2 class="utils.SMField">BOW_REFERENCES</field2>
                <mix>false</mix>
                <converter class="utils.converter.StringConverterFactory$1"/>
              </desambiguation.classification.SetContainsAnyClassifier>
              <desambiguation.classification.SetContainsAnyClassifier>
                <classifierType>Bag average classifier</classifierType>
                <classifierName>AG 462 ID References classifier       </classifierName>
                <blocking>true</blocking>
                <hit>false</hit>
                <field1 class="utils.SMField">ID</field1>
                <field2 class="utils.SMField">BOW_REFERENCES</field2>
                <mix>false</mix>
                <converter class="utils.converter.StringConverterFactory$1"/>
              </desambiguation.classification.SetContainsAnyClassifier>
            </com.google.common.collect.TreeMultimap>
          </classifiers>
        </cc>
      </desambiguation.classification.RecurrentAggregator2>
    </list>
```