An Evaluation of Transliterated Arabic Name Matching Methods

Jessica Moore[1,2][0000−0003−2076−3796], Sohaib Hamid[1,3][0000−0002−6965−5588], and Seth Bromberger[1,4][0000−0003−1411−0385]

[1] Global Analytic Platform, US Special Operations Command
[2] LMI Consulting, Tysons, VA 22102, USA
jmoore@lmi.org
[3] World Wide Language Resources, Fayetteville, NC 28301, USA
sohaib.hamid.ctr@socom.mil
[4] Lawrence Livermore National Laboratory, Livermore, CA 94550, USA
seth@llnl.gov

Abstract. Name matching algorithms play a key role in information retrieval systems and are especially critical when names may be represented in a wide variety of valid ways. This is often the case for names conventionally written with non-Latin characters where there is no de facto transliteration standard; for example, the same sequence of Arabic characters can be written in a multitude of ways using Latin ones. We propose a method for encoding transliterated Arabic names that is highly effective compared to common existing methods. We use our encoding to assess whether pairs of transliterated Arabic names match, i.e., whether they are the same in the original Arabic. We demonstrate that our method achieves state-of-the-art results, outperforming previous algorithms for this task. We validate the utility of our encoding on a collection of hand-curated, transliterated Arabic names. We make this data set and all of our code available to other authors for reproduction or expansion.

Keywords: Arabic · Entity resolution · Name similarity · Transliteration

1 INTRODUCTION

Conventionally, an individual's name has a single spelling in its original language. However, when a name is transliterated to another language (especially one that does not have the same character set as the language of origin), the name may be spelled in a variety of ways [1]. As a result, a single name may be identified by a multitude of character sequences. For example, the same sequence of Arabic characters can be transliterated as *Muhammed*, *Mohammed*, *Mohamad*, or eleven other variants, potentially leading to significant confusion [2].

The authors wish to thank COL Brendan Dunne and MAJ Gabriel Samudio of the United States Special Operations Command (USSOCOM) Global Analytic Platform (GAP) for their support of this research.

Rising international travel and migration ensures that more and more individuals with names traditionally spelled using Arabic characters will have their names represented with Latin ones, as they travel to countries that use the Latin alphabet. The high level of ambiguity in written Arabic and the lack of consistently applied standards for transliteration frequently result in the same individual's name being spelled in a wide variety of ways across the various systems in which their information is recorded [3]. As a result, organizations must accurately identify matches among transliterated names in order to maintain data free from duplication and ambiguity.

We propose a method for matching pairs of Arabic names that have been transliterated using Latin characters. Our algorithm enables rapid retrieval of names that approximately match a given query string, while requiring no change to an organization's underlying data storage processes (e.g., altering databases to include non-Latin characters). This paper contributes to the existing literature in several ways:

- It presents a new method for encoding transliterated Arabic names and shows that method achieves state-of-the-art results on a single-name matching task.
- It demonstrates that the proposed method maintains impressive recall, even at high values of required precision (e.g., 0.75 or 0.9) and vice versa, especially compared to other encoding methods.
- It is accompanied by open-source data and code, so that other researchers may easily compare their work and, hopefully, develop even more effective name matching algorithms in the future.[5]

2 PRIOR WORK

Name matching has a long history as a key element of information retrieval systems. More than a quarter-century ago, Thompson and Dozier observed that names occur frequently enough in text documents to serve as a useful feature in document search [4]. Even more commonly, name matches are used to solve record-linkage (and the broader class of entity resolution) problems, allowing a system to combine all records that relate to the same individual [5]. Name matching has been applied to a wide variety of data sets; various name matching algorithms have seen applications in electronic health records [6], legal information systems [7], and voter files [8]. All of these fields, and many more, could therefore see benefit from a name matching algorithm geared towards matching transliterated Arabic names.

In order for information retrieval systems to effectively leverage names as features of a document or record, the names must be comparable, i.e., it must be possible to quantitatively assess how similar two names are. At the most basic level, one might determine similarity by comparing the characters in the

[5] GitHub link removed to maintain anonymity; it will be added to the final version of the paper.

names. Approaches that do so are known as "string-similarity" methods and have historically been applied to many information retrieval and entity resolution problems, including name matching [9–11]. One well-known example of a string-similarity method is Levenshtein edit distance, which calculates the number of character changes to transform one word into another. (Several other string-similarity algorithms are discussed in Section 4.1.)

However, string-similarity methods fail when they encounter words that sound similar, but are spelled differently (e.g., "Steven" and "Stephen"). This is uniquely problematic in the case of names (especially transliterated ones), due to the diversity of spellings associated with the same pronunciation. As a result of this deficiency, phonetic similarity algorithms (i.e., algorithms that identify words as similar if they sound the same) are more common [12–14]. Phonetic similarity methods typically have an encoding scheme that maps the characters in the original string to an "encoded" series of characters (e.g., "Steven" and "Stephen" are both encoded by the Soundex algorithm as "S135"). These encoded series are then compared, either to find an exact match or using a string-similarity method [15].

Because of their ability to account for varying spellings of similar-sounding terms, phonetic similarity methods are commonly used to match names written in Arabic script, either to others in Arabic script or ones that have been transliterated with Latin characters. There exist a number of phonetic similarity algorithms (often termed "Arabic Soundex") used for matching names written in Arabic script [16–18]. These algorithms each use a distinct mapping strategy to encode Arabic names; the mappings differ slightly, some accounting for specific accents or other nuances of the language. There also exist several methods for matching Arabic script to its transliterated equivalent. Yousef encodes Arabic script and Latin transliterations with Arabic Soundex and English Soundex, respectively; he then compares the two encodings to each other [19]. Freeman et al. employ a cross-linguistic Levenshtein edit distance which accounts for sets of characters considered to "match" between Arabic and English. [20]. In contrast, Kay and Rineer elect to normalize the Arabic script and then transliterate it to compare it to its already-transliterated counterpart [21]. Approaches vary, but the philosophy of comparison using phonetic similarity is consistent throughout the literature.

While phonetic-similarity methods seem highly applicable to our use case (matching pairs of transliterated Arabic names), the common phonetic matching algorithms for words written with Latin characters are based on English pronunciations [22]. To our knowledge, there has been one paper in modern literature that addresses this problem [23]. Holmes, et al. use a highly-detailed rules-based process for encoding transliterated names. Their methodology works well, but fails to account for the variation in Arabic names that could potentially have resulted in the transliteration. For example, the English "a" may result from either a short-a (fatha) or long-a (alif) in the original Arabic. In addition, the methodology from Holmes can produce false positives by eliding distinguishing characteristics of names, in particular, the common prefix "Abdul-". Thirdly, the

Holmes methodology relies on a complex ordering of substitution rules to provide a surjective mapping between transliterated n-grams and encoded tokens. The order of rules is of outsize importance in the quality of the matching, and inadvertent modification of the rule order (for example, by adding, changing, or removing a rule) can drastically impact the method's effectiveness.

3 THE ARCODER ENCODING METHOD

ARCoder is an encoding strategy specifically designed for transliterated Arabic names. It provides a mapping between characters of the transliterated name and representative encoded symbols, where homophonous elements map to the same symbol.

ARCoder has three features not commonly found in other name matching strategies that provide advantages in matching Arabic names:

- The mapping can encode multiple variants, so that a single unencoded element can be represented by multiple encoded elements. The overall name therefore is represented by the Cartesian product of all the variant elements. For example, ARCoder encodes *Zeinah* as both "sena" and "seina."
- Characters within a name may be encoded as n-grams (in contrast with, for example, Soundex, which encodes character-by-character). For example, the digraph "iy" encodes to "e" in ARCoder, which represents a long "e" sound ($\overline{e}$).
- The position of a character within the name may result in alternate mappings: a final "h" is ignored, whereas "h"s are not ignored if they appears elsewhere in the name, and "t" can map to both "t" and "d" at the end of a word, but only encodes to "t" within a word.

A detailed description of the encoding methodology used by ARCoder is provided in Appendix A.

4 EVALUATION OF ENCODING STRATEGIES

In additional to Holmes' methodology, we compare ARCoder to three other common encoding strategies:

- Metaphone [24]
- New York State Information and Intelligence System (NYSIIS) [25]
- Soundex [26]

All of these algorithms were originally developed to compare English-language words. Their performance can be considered a reasonable baseline for the comparison of transliterated Arabic names, as these algorithms are designed to operate on Latin letters that result in English pronunciation (which is the nominal use of transliterations). We also consider the original transliteration to be its own encoding ("None").

4.1 Similarity Measures

In order to compare two encoded names, one needs to establish a metric by which to assess their similarity. In the interest of providing a fair comparison, we use several common text similarity measures as outlined below. Each computes p, the similarity between strings s_1 and s_2. For each measure, $0 \leq p \leq 1$.

4.1.1 Jaro Jaro similarity accounts for both the number of exact matches between two strings, as well as the number of character transpositions that would yield a match.

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

where m is the number of matching characters, t is the number of transpositions (i.e., pairs of transposed characters), and $|s_i|$ is the number of characters in s_i.

4.1.2 Jaro-Winkler (JW) Jaro-Winkler similarity extends Jaro similarity by adding a bonus for overlap between the first several characters in a string.

$$p = p_j + cd(1 - p_j) \tag{2}$$

where p_j is the Jaro similarity between s_1 and s_2, c is the number of overlapping characters (capped at 4) at the beginning of each string, and d is a scaling factor (here, set to 0.1, as is convention). So long as d does not exceed 0.25, p has an upper bound of 1.

4.1.3 Levenshtein Levenshtein edit distance is the number of single-character changes (additions, deletions, substitutions) required to change one word to another. Since the Levenshtein algorithm computes an unnormalized distance, we use it to compute a similarity between s_1 and s_2, as follows:

$$p = 1 - \frac{l}{max(|s_1|, |s_2|)} \tag{3}$$

where l is the Levenshtein edit distance between the strings.

4.1.4 Damerau-Levenshtein(DL) Damerau-Levenshtein edit distance is identical to Levenshtein edit distance except in that it accounts for transpositions of adjacent letters, treating these as one change, rather than two changes. We obtain a similarity based on Damerau-Levenshtein edit distance in the same way we do for Levenshtein, above.

4.2 Data

We employ a set of 1,519 transliterated Arabic names compiled by a native Arabic linguist. These names are divided into 501 name groups, i.e., collections of names transliterated from the same sequence of Arabic characters. For example, a single name group may contain *Ahmed*, *Ahmad*, and *Ahmet*, because these are all transliterations of the Arabic أحمد.

We create pairwise combinations of these transliterated names and mark them as matches or non-matches, depending on whether they are from the same name group. Naturally, our data are heavily skewed in the direction of non-matches, per Table 1. We believe this distribution approximately reflects the real-world data onto which our algorithm might be applied, since in any given pool of names, there are far more non-matching pairs than matching ones.

Regardless of the underlying distribution, our key metrics – precision and recall – do not tend to be heavily skewed by the data distribution. We prefer precision / recall curves to standard receiver operating characteristics (ROC) curves to avoid obscuring the algorithms' performance on the negative (non-matching) class. Specifically, the false-positive rate used by the ROC curve tends to remain artificially low, when there is a large number of negatives (non-matches) in the data set, as is the case here [27–29]. Precision (the fraction of true matches out of those assessed to be matches) does not encounter the same problem and so it remains an informative metric in spite of the skewed distribution.

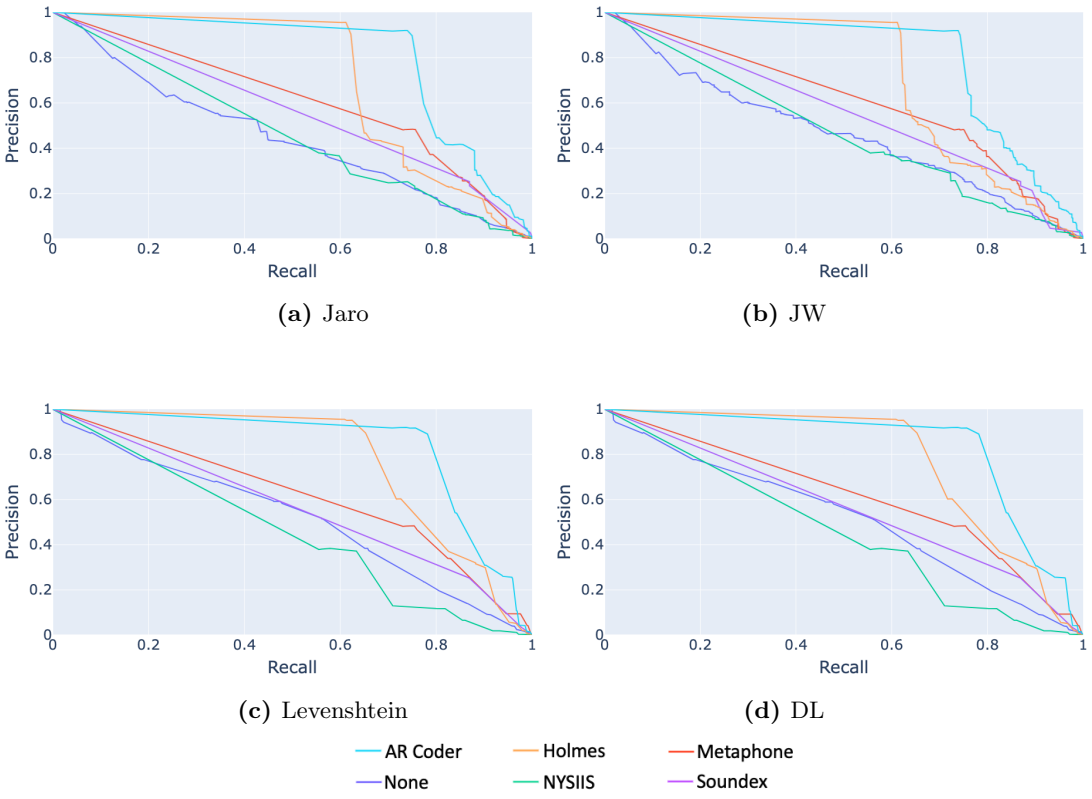
Table 1. Number of matches and non-matches in evaluation data.

Category	Number
Match	2,100
Non-match	1,134,178

4.3 Comparison to Other Encoding Algorithms

Per Figure 1 and Table 2, ARCoder outperforms all other encoding strategies using every assessed measure of similarity. Unsurprisingly, the English-based encoding algorithms (including no-encoding / "None") perform significantly worse than the two Arabic-specific encoding strategies.

For both ARCoder and Holmes, there is a noticeable drop-off in precision above 0.6 recall, but ARCoder maintains high levels of precision at higher recall values than Holmes, resulting in a significant outperformance overall. We believe that the ability of ARCoder to encode transliterated n-grams to multiple variants increases the overall number of positive match results. When applied intelligently, these variants improve the true positive rate of the algorithm.



(a) Jaro **(b)** JW

(c) Levenshtein **(d)** DL

Fig. 1. Precision-recall curves for various encoder / measure combinations.

4.4 Performance at Extreme Precision / Recall Values

It is often the case that information retrieval systems must work at very high levels of precision or recall. For example, in the case of matching candidates for high-trust jobs in the banking sector to a roster of candidates convicted of financial crimes, high recall (minimizing false negatives) is absolutely necessary as it could potentially be very costly to bring on an ineligible employee. However, in the case of querying a database of existing customers to record a new sale, high precision might be preferred. It would be far less damaging to create a duplicate customer entry than to attribute a sale to the wrong customer.

To assess the performance of each algorithm at high levels of precision/recall, we find the maximum recall value for a given precision level. We then do the same with precision and recall reversed. For simplicity, we use Damerau-Levenshtein to make these comparisons; similar results hold for all other similarity measures. We report these values in Tables 3 and 4. The values may not comport precisely with chart (d) in Figure 1, because they are affected by the thresholds available in the data, whereas the precision / recall curves interpolate between them. As

Table 2. Area under precision-recall curves. ARCoder has higher area across all distance metrics tested.

Encoding	Jaro	JW	Levenshtein	DL
ARCoder	**0.801**	**0.798**	**0.833**	**0.833**
Holmes	0.707	0.707	0.773	0.773
Metaphone	0.606	0.605	0.612	0.612
None	0.446	0.465	0.514	0.513
NYSIIS	0.466	0.472	0.452	0.452
Soundex	0.561	0.557	0.560	0.560

a result, for a given minimum precision (recall) value, it may be the case that the true precision (recall) is *above* the required minimum. To add appropriate context, we report the true precision and recall values in parentheticals.

ARCoder outperforms the other encoding methods for almost every extreme value of required precision and recall. In particular, ARCoder has moderately higher recall, as compared to Holmes, at high required precision (Table 3) and significantly outperforms Holmes in precision at high required recall (Table 4). We do note that at the most extreme precision value assessed (0.95), Holmes outperforms ARCoder. However, this appears to occur largely because ARCoder's predictions on this data set do not yield any precision values between 0.92 (where ARCoder achieves recall of approximately 0.76) and 1.0 (where ARCoder achieves recall of 0).

Table 3. Maximum recall, at specified minimum precision levels. Corresponding precision values are provided in parentheses, for added context.

Encoding	Minimum Precision		
	0.95	**0.90**	**0.75**
ARCoder	0.00 (1.00)	**0.76** (0.92)	**0.78** (0.89)
Holmes	**0.62** (0.95)	0.63 (0.95)	0.65 (0.90)
Metaphone	0.00 (1.00)	0.00 (1.00)	0.00 (1.00)
None	0.02 (0.95)	0.03 (0.93)	0.19 (0.78)
NYSIIS	0.00 (1.00)	0.00 (1.00)	0.00 (1.00)
Soundex	0.00 (1.00)	0.00 (1.00)	0.00 (1.00)
	Recall (Precision)		

4.5 Computational Requirements

Given the scale of data on which these algorithms will need to be applied, the computational requirements associated with encoding need to be taken into ac-

Table 4. Maximum precision, at specified minimum recall levels. Corresponding recall values are provided in parentheses, for added context.

Encoding	**Minimum Recall**		
	0.95	**0.90**	**0.75**
ARCoder	**0.25** (0.96)	**0.31** (0.90)	**0.92** (0.76)
Holmes	0.06 (0.95)	0.30 (0.90)	0.37 (0.83)
Metaphone	0.09 (0.95)	0.09 (0.95)	0.48, (0.75)
None	0.04 (0.96)	0.09 (0.91)	0.24 (0.77)
NYSIIS	0.01 (0.96)	0.02 (0.92)	0.12 (0.80)
Soundex	0.03 (0.98)	0.03 (0.98)	0.25 (0.87)
	Precision (Recall)		

count. We encode every name in our evaluation data 1,000 times to obtain relative speed values for each algorithm [6] ARCoder's computational requirements, as we show in Figure 2, are on par with Holmes (if not an improvement from their algorithm).[7] Both ARCoder and Holmes, however, are significantly more computationally expensive than the other algorithms we have assessed. It is worthwhile to note that this problem is trivially parallel, insofar as each name can be encoded without information about the others.

5 CONCLUSION

In this paper, we introduce ARCoder, a new method of matching transliterated Arabic names. ARCoder provides unique features not typically found together in any other encoding algorithm: the ability to encode transliterated names into multiple variants; the ability to tokenize both single-characters and bigrams; and the ability to customize encoding rules based on the position of the substring within the name. Taken together, these features result in improved performance by ARCoder, relative to existing methods. We demonstrate as much on a hand-curated set of transliterated Arabic names, which we make available to other researchers. In addition to demonstrating ARCoder's overall performance improvement, we show its effectiveness at extreme values of precision and recall, which are often required in information retrieval applications. Finally, we show that ARCoder is no more computationally expensive than the next-best encoding algorithm, making it the optimal choice for transliterated Arabic name encoding.

[6] Speed assessment was conducted on a MacBook Air with a 1.8 GHz Dual-Core Intel Core i5.

[7] The authors have reimplemented Holmes' algorithm in Python and used this reimplementation for the assessment of computational complexity. Given that Holmes' algorithm was originally implemented in SQL, there may be optimizations on which we do not fully capitalize.

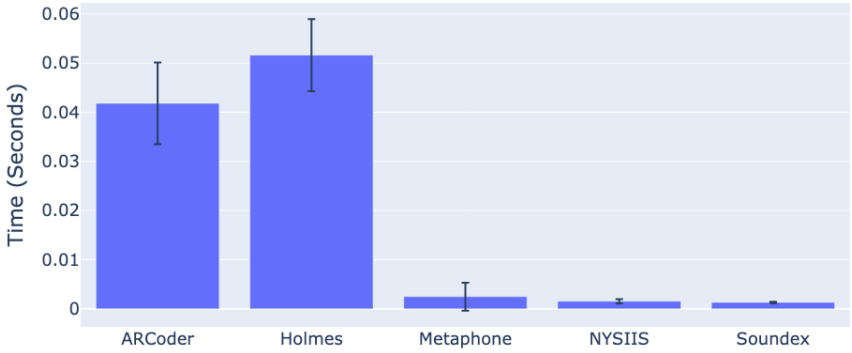


Fig. 2. Time to encode 1,519 names. ARCoder takes less time than Holmes across multiple runs.

Our results in this paper leave ample room for further improvements to transliterated Arabic name-matching. Potentially informative future work includes the development of hybrid / ensemble methods where the relative advantage of Holmes at very high precision values could be leveraged to improve the overall precision / recall trade-off. Using a traditional classification model on the raw similarity metrics may also yield improvements in name matching predictions. Changes to the encoding map itself may also result in improved performance.

References

1. Hallak, H.: Tools and libraries for matching Arabic names written in English, 29 Dec 2020, `https://ws-dl.blogspot.com/2020/12/2020-12-28-tools-and-libraries-for.html`. Last accessed 8 Aug 2022
2. Meierhans, J., & England, R.: Baby names: Is Muhammad the most popular? BBC News, 26 Sept 2018, `https://www.bbc.com/news/uk-england-45638806`
3. Halpern J.: The challenges and pitfalls of Arabic romanization and arabization. In: Proc. Workshop on Comp. Approaches to Arabic Scriptbased Lang. (2007).
4. Thompson, P., Dozier, C.: Name searching and information retrieval. In: Second Conference on Empirical Methods in Natural Language Processing. (1997).
5. Herzog, T.H., Scheuren, F., Winkler, W.E.: Record linkage. In: Wiley Interdisciplinary Reviews: Computational Statistics, 2(5), pp. 535–543. Springer, New York (2007). https://doi.org/10.1007/0-387-69505-2
6. McCoy, A.B., Wright, A., Kahn, M.G., Shapiro, J.S., Bernstam, E.V., Sittig, D.F.: Matching identifiers in electronic health records: implications for duplicate records and patient safety. In: BMJ Quality & Safety, 22(3), pp. 219–224. (2013). https://doi.org/10.1007/0387695052

7. Branting, L. K.: A comparative evaluation of name-matching algorithms. In: Proceedings of the 9th international conference on Artificial Intelligence and Law, pp. 224–232. (2003). https://doi.org/10.1145/1047788.1047837
8. Meredith, M., Morse, M.: Do voting rights notification laws increase ex-felon turnout? In: The ANNALS of the American Academy of Political and Social Science, 651(1), pp. 220–249. (2014). https://doi.org/10.1177/000271621350293
9. Bilenko, M., Mooney, R.J.: Adaptive duplicate detection using learnable string similarity measures. In: Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining, pp. 39–48. (2003). https://doi.org/10.1145/956750.956759
10. `https://journal.hep.com.cn/fcs/EN/article/downloadArticleFile.do?attachType=PDF&id=12803`
Yu, M., Li, G., Deng, D., Feng, J.: String similarity search and join: a survey. In: Frontiers of Computer Science, 10(3), pp. 399–417. (2016.) https://doi.org/10.1007/s11704-015-5900-5
11. https://www.cs.cmu.edu/ wcohen/postscript/ijcai-ws-2003.pdf
Cohen, W.W., Ravikumar, P., Fienberg, S.E.: A Comparison of String Distance Metrics for Name-Matching Tasks. In: IIWeb (3), pp. 73–78. (2003.)
12. Tissot, H., Dobson, R.: Combining string and phonetic similarity matching to identify misspelt names of drugs in medical records written in Portuguese. In: Journal of biomedical semantics, 10(1), pp. 1–7. (2019.) https://doi.org/s13326-019-0216-2
13. Petrik, S., Kubin, G.: Reconstructing medical dictations from automatically recognized and non-literal transcripts with phonetic similarity matching. In: 2007 IEEE International Conference on Acoustics, Speech and Signal Processing-ICASSP'07 (4), pp. IV–1125). (2007.) https://doi.org/10.1109/ICASSP.2007.367272
14. Lam, W., Huang, R., Cheung, P.S.: Learning phonetic similarity for matching named entity translations and mining new translations. In: Proceedings of the 27th annual international ACM SIGIR conference on Research and development in information retrieval, pp. 289–296. (2004.) https://doi.org/10.1145/1008992.1009043
15. Zobel, J., Dart, P.: Phonetic string matching: Lessons from information retrieval. In: Proceedings of the 19th annual international ACM SIGIR conference on Research and development in information retrieval, pp. 166–172. (1996.)
16. Aqeel, S. U., Beitzel, S., Jensen, E., Grossman, D., Frieder, O.: On the development of name search techniques for Arabic. In: Journal of the American Society for Information Science and Technology, 57(6), pp. 728–739. (2006). https://doi.org/10.1002/asi.20323
17. Yahia, M. E., Saeed, M. E., Salih, A. M.: An intelligent algorithm for Arabic soundex function using intuitionistic fuzzy logic. In: 3rd International IEEE Conference Intelligent Systems, pp. 711–715. (2006). https://doi.org/10.1109/IS.2006.348506
18. Ousidhoum, N. D., Bensaou, N.: Towards the refinement of the Arabic Soundex. In: International Conference on Application of Natural Language to Information Systems, pp. 309–314. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38824-8_30
19. Yousef, A. H.: Cross-language personal name mapping. arXiv preprint arXiv:1405.6293. (2014). https://doi.org/10.48550/arXiv.1405.6293
20. Freeman, A., Condon, S., & Ackerman, C.: Cross linguistic name matching in English and Arabic. In: Proceedings of the Human Language Technology Conference of the NAACL, Main Conference, pp. 471–478. (2006).

21. https://aclanthology.org/2012.amta-caas14.5.pdf
Kay, B.N., Rineer, B.C.: Approaches to Arabic Name Transliteration and Matching in the DataFlux Quality Knowledge Base. In: Fourth Workshop on Computational Approaches to Arabic-Script-based Languages, pp. 32–37. (2012.)
22. Beider, A., Morse, S.: Phonetic Matching: A Better Soundex, Mar 2010, https://stevemorse.org/phonetics/bmpm2.htm. Last accessed 27 Sept 2022
23. Holmes, D., Kashfi, S., Aqeel, S. U.: Transliterated arabic name search. In: Communications, Internet, and Information Technology, pp. 267–273. (2004).
24. Lawrence Philips' Metaphone Algorithm, http://aspell.net/metaphone/. Last accessed 19 Sept 2022
25. IBM Documentation, NYSIIS coding, https://www.ibm.com/docs/en/iis/11.5?topic=statements-nysiis-coding. Last accessed 19 Sept 2022.
26. National Archives, Soundex System, https://www.archives.gov/research/census/soundex. Last accessed 19 Sept 2022.
27. Saito, T., Rehmsmeier, M. The precision-recall plot is more informative than the ROC plot when evaluating binary classifiers on imbalanced datasets. In: PloS one 10, no. 3 (2015). https://doi.org/10.1371/journal.pone.0118432
28. Jeni, L., Cohn, J., De La Torre, F.: Facing Imbalanced Data Recommendations for the Use of Performance Metrics. In: Humaine Association Conference on Affective Computing and Intelligent Interaction. (2013). https://doi.org/10.1109/ACII.2013.47
29. Chou, S.: Precision - Recall Curve, a Different View of Imbalanced Classifiers 20 Apr 2022, https://sinyi-chou.github.io/classification-pr-curve/

Appendix A. ARCODER METHODOLOGY

The process for encoding a transliterated Arabic name (with arbitrary capitalization) into a set of ARCoder-encoded strings is as follows:

1. Name Preprocessing
   1.1 Convert all upper-case to lower-case Latin ("Address-Book!" becomes "address-book!").
   1.2 Remove all invalid characters. The set of valid characters includes lower-case alphabetics as well as the forward and reverse single-quotation marks, spaces, and dashes. ("address-book!" becomes "address-book")
   1.3 Replace dashes with spaces ("address-book" becomes "address book")
   1.4 Remove all consecutive duplicate characters ("address book" becomes "adres bok")
   1.5 Capitalize the first character in each word and concatenate the words without spaces ("adres bok" becomes "AdresBok")
2. Tokenization:
   2.1 Create an empty ordered list of sets of tokens and initialize a variable representing the string index to 0.
   2.2 Starting at the first character in the processed name, determine whether the next two characters match any of the mapping rules, paying attention to the position of the characters within the word. For example, the presence of a capitalized letter indicates the start of a word. Perform the tokenization in the following order:
      – position-dependent bigram mapping
      – position-independent bigram mapping
      – position-dependent single-character mapping
      – position-independent single-character mapping
   2.3 Append the set of tokens (there can be more than one), and increment the string index variable by either 1 or 2, depending on whether a bigram or single-character mapping was used.
   2.4 When the string index variable exceeds the length of the string, create a set of variants by computing the Cartesian product of all items in the token list, and concatenate each variant into a separate single string. These strings represent the encoded name variants.
3. Comparison:
   Two transliterated names are equivalent if the intersection of their tokenized variant sets is non-empty. Furthermore, the maximum distance between all pairs of tokenized variants can be used as a representation of approximate equivalence.