Using FastMap and KDTrees for Computationally Efficient Address Matching for Database Deduplication

Nathan Decker, Anthony Nguyen

Abstract—In fields such as real estate, economics, politics, and government, property addresses are commonly used as unique identifiers when merging data from disparate data sources. However, since many sources record addresses from user submitted data, typos are common. As a result, merging of these data sources can yield "duplicate" instances in the final data set. A duplicated instance is when two instances are identified differently by address, but represent the same physical property. While resolving instances of duplication is doable provided knowledge of the ground truth, identifying which instances are likely duplicates can be computationally challenging. One proposed approach is to use distance metrics for evaluating similarity between pairs of addresses to identify possible duplicate instances. However, the computational cost for evaluating an entire database pairwise is infeasible for large datasets. A tool for identifying duplicates utilizing FastMap and KDTrees to quickly screen for potential matches before iterating through pairs is proposed as a more computationally efficient solution to this problem. This method is shown to have significantly lower computational cost than assessing similarity via a distance function for all possible pairs while preserving accuracy.

1 Introduction

ACROSS a variety of disciplines, there is increasingly a desire to utilize the abundance of publicly available data to better inform decisions and forecast future trends. In this process, data is extracted from public sources and aggregated to created sufficiently large databases where methods in statistics and machine learning can be applied to make inferences about the data. While methods used and interpretations made are often domain specific, a shared problem exists with regard to merging large data sets. To merge data from disparate sources, a unique identifier is typically needed to link instances from one database to another. In fields such as real estate, economics, politics, and government, property addresses are commonly used for this purpose.

The issue with using addresses as a unique identifier is that they are typically user submitted and not validated against a ground truth list of addresses making them subject to typos. Additionally, addresses from disparate data sources can also result in a variety of different formats and utilize different abbreviations. Various packages exist to convert differently formatted addresses into standardized addresses, but these are imperfect and can result in incorrect address listings that are effectively similar to having an address with typos in it. When this occurs, merging of the data sets will result in undesired duplicate. We define a duplicate as when the data set has multiple different entries for a single physical property. Having duplicates results in redundancies in the data, and inaccuracies when analyzing the data and make inferences.

An example of this problem is in the real estate industry. A real estate analytics company may accumulate data for rent prices from various publicly available data sources. Different sources may have the same property listed as different entries because of typos. When aggregating data sources, this will result in an a duplicate property in the aggregated data set. Further, sometimes the same data source may contain typos for the same property between different points in time. This leads to substantial issues when generating longitudinal data sets for a temporal based analysis.

1.1 Problem Description

Given a set of addresses that is likely to contain typos, and no set of true addresses that can be considered the ground truth, the goal is to identify addresses that are likely to represent the same physical property. Doing so will enable deduplication of data sets formed when merging of data that utilize user submitted entries as the unique identifier. We will refer to this problem as an address matching problem. One approach to handling this problem is to perform pairwise comparisons between all addresses using a similarity metric or distance function. A threshold can be set for the the similarity score, and all addresses within said threshold could be considered a duplicate. However, this process is computationally inefficient, especially when considerations are not made to prevent repeating comparisons that have already been performed. For large data sets, this approach is not feasible. Our objective is to develop a more efficient method for address matching in large data sets to identify possible duplication in the set.

1.2 Contribution

We propose a screening methodology that can be used to reduce the set of addresses that need to be pairwise compared. By strategically reducing the number of comparisons needing to be performed by establishing a spatial and numerical relationship between addresses to apply clustering techniques, we are able to drastically reduce computation time while maintaining high accuracy in duplicate identification. Further, we will assess how changes in the data set size and hyper-parameters in our methodology impact accuracy and computation time.

2 Methods

In our methods, we discuss the high level theory that plays into our our screening algorithm for similar objects. We will also discuss how we implement a brute force method for pairwise comparisons applied after the screening algorithm. Finally, we will discuss the design of experiments performed to assess the effectiveness of incorporating our screening algorithm prior to performing a brute force implementation for pairwise comparisons.

2.1 Screening Algorithm for Similar Objects

Our screening method utilizes the FastMap algorithm to represent each address as numeric values in a 2 dimensional vector. This vector effectively maps each address object to a location in a 2 dimensional space. We then build a 2D tree ("KDTree") that we can use to quickly identify all points within a specific distance of our point. Objects in the identified radius are the only points that will be pairwise compared with the point of interest. Because the number of potential matches that must be evaluated is reduced, the computational cost of the search decreases.

2.1.1 FastMap of Distances

FastMap is an algorithm that can be utilized to convert a set of arbitrary objects into n-dimensional vectors of numeric values by utilizing a domain appropriate distance function for the objects of interest. Abstract objects are created in a euclidean embedding that preserves pairwise distances between them [3]. This conversion enables the use of traditional methods in machine learning that require numeric type features in geometric space to understand patterns and identify spatial relationships. More details about the original FastMap can be found in the following paper by Faloutsos and Lin, 1995: "FastMap: A Fast Algorithm for Indexing, Data-Mining and Visualization of Traditional and Multimedia Datasets" [4].

2.1.2 Levenshtein Distance Function

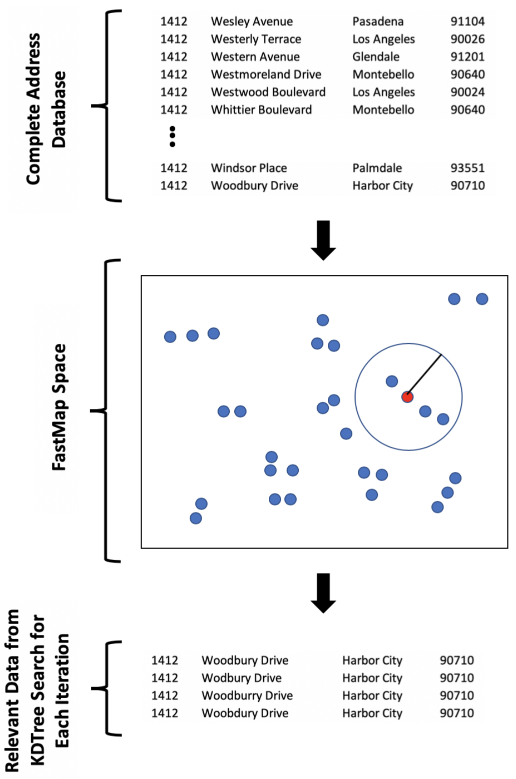
The Levenshtein distance function is a standard string metric used to measure the difference between 2 string sequences. In our example, each address is a string sequence. The Levenshtein distance is an integer the reflects the number of edits that must be made to one sequence to replicate the second sequence. An edit is considered an insertion, deletion, or substitution of character. Levenshtein distance is commonly used in applications such as spellcheckers. This method can be quite useful for shorter string sequences, but can quickly become computationally demanding for longer sequences [5]

2.1.3 Our Modified Distance Function

Because a pure Levenshtein distance function is infeasible for use with address data types, we implement a hybrid distance function approach. In this function, if address or apartment unit numbers are different, a set distance penalty is added to the sums of the Levenshtein distances between the street and city names. This approach incorporates domain knowledge regarding the structure of addresses, and accounts for the fact that two addresses can differ in only numbers, but should not be considered close to one another. One downside of this approach is that if a typo occurs in the address number of an instance, it will likely not be matched with its intended true address. We use this distance function in our FastMap implementation and to quantify similarity between addresses.

2.1.4 KDTree

A KD Tree (K-Dimensional Tree) is a space partitioning data structure in the form of a binary search tree. Each node represents a K-Dimensional point that divides the space into two half-spaces. In our application, the KD tree allows us to quickly search the set of addresses for address clusters that are in proximity to each other using our FastMap transformed address objects. Using a radius hyper-parameter, we can define how large of a sphere around each given point we want to consider when doing pairwise comparisons. Defining a larger radius means that more addresses will be pairwise compared to the center address to assess for matches. This process is illustrated in Figure 1. A flowchart explaining the full methodology is given in Figure 2.


Complete Address Database
FastMap Space
Relevant Data from KDTree Search for Each Iteration

1412	Wesley Avenue	Pasadena	91104
1412	Westerly Terrace	Los Angeles	90026
1412	Western Avenue	Glendale	91201
1412	Westmoreland Drive	Montebello	90640
1412	Westwood Boulevard	Los Angeles	90024
1412	Whittier Boulevard	Montebello	90640
⋮			
1412	Windsor Place	Palmdale	93551
1412	Woodbury Drive	Harbor City	90710

1412	Woodbury Drive	Harbor City	90710
1412	Wodbury Drive	Harbor City	90710
1412	Woodburry Drive	Harbor City	90710
1412	Woobdury Drive	Harbor City	90710

Fig. 1. Diagram of Neighbor Selection Process

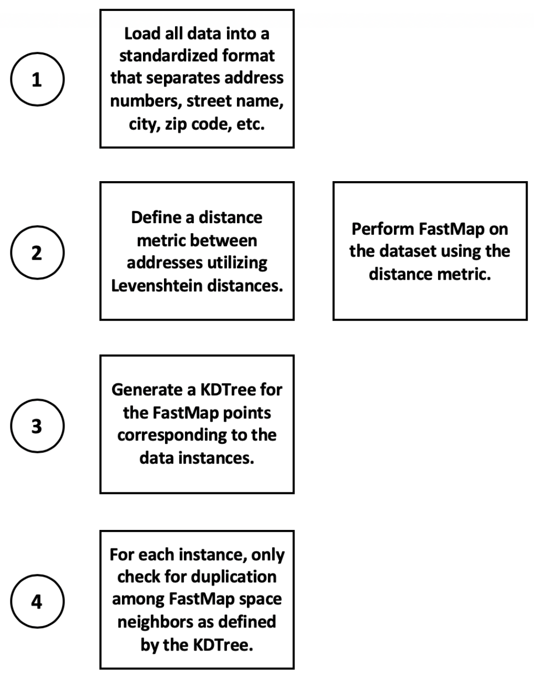

1
Load all data into a standardized format that separates address numbers, street name, city, zip code, etc.
2
Define a distance metric between addresses utilizing Levenshtein distances.
Perform FastMap on the dataset using the distance metric.
3
Generate a KDTree for the FastMap points corresponding to the data instances.
4
For each instance, only check for duplication among FastMap space neighbors as defined by the KDTree.

Fig. 2. Flowchart of Overall Methodology

2.2 Brute Force Pairwise Comparisons

A brute force approach to pairwise comparison involves nesting for loops that each iterate through all items. In this construction, each item is compared to all other items. However, this results in repeated computations as we would compare A with B, and then likewise B with A at a later step. In our implementation of the brute force method for efficiency comparison, we will add an extra step to check if a calculation has already been performed to ensure repeat calculations are not done.

2.3 Algorithm Testing

A variety of tests were performed to assess performance for the address matching problem. The base case is a strategically designed brute force approach that performs no redundant calculations. We compare this with our approach that utilizes a screening algorithm to select a subset of addresses in proximity to the address being assessed. In our method, pairwise comparisons are performed on this reduced subset rather than the whole set of addresses.

We will first outline the data utilized in our tests. We will then discuss the structure of our accuracy and efficiency test. Last, we will present a sensitivity analysis on one of the hyper-parameters necessary for our algorithm that impacts the trade-off between computational time and accuracy.

2.3.1 Data Generation

We start with a data set of property addresses for Los Angeles, California. We do not consider data sets of different formats because a variety of packages already exist to standardize address formats. Further, we only introduce typos into street address names and not numbers. This is because deviations in address numbers are highly likely to still be valid addresses and should not be assumed to be a typo when there is no ground truth for comparison. For larger data sets, the methodology can be expanded to incorporate cities, states, etc.

To expand our data set to include possible mistakes and typos, we create a typo generator. For each address in our data set, we first randomly determine how many instances of the property should exist in our final data set (1-3 additional instances chosen randomly). In each of these instances, we either do not make any edits or we implement a single sequence edit that is either an insertion, deletion, substitution, or translation (letter swap). Not making edits reflects a properly typed address while the remaining reflect types of typos a user may make. We assume that at most only typo may exists. We believe this is a reasonable assumption for user inputted addresses. Our final data set contains anywhere from 2-4 addresses for a true physical property that may have different types of typos. The order of the addresses will be shuffled prior to performing any of our tests.

2.3.2 Experiments for Assessment

First, we assess the differences in accuracy between the base case and when our screening algorithm is used. We perform this analysis for data sets of different sizes to assess if accuracy diminishes with larger data sets. This is a crucial consideration as the FastMap algorithm mapping process is dependent on the other objects being mapped. Next, we assess the computational efficiency of the base case in comparison to utilizing our screening algorithm. We again perform this analysis for varying data set sizes. Finally, we do a sensitivity analysis on the radius hyper-parameter that dictates how large of a subset should be used for our pairwise comparisons. We will assess sensitivity of both the computational time and accuracy with respect the the radius of the KDTree search hyper-parameter.

3 Results

We graphically present the findings from our algorithm assessments. In Figure 3, we compare computational time for the proposed method that involves a screening step and a pure brute force method. We perform tests for with number of instances ranging from 5,000 to 30,000. It can be seen that the proposed method is significantly more efficient as the size of the dataset increases. This is because less points need to be checked on each iteration as a result of the screening procedure.

In Figure 4, we compare accuracy between the same two approaches over the same range of instance in the data set. It can be seen that there is no significant decrease in the accuracy of the matching after implementing the FastMap screening proceedure.

Figures 5 and 6 showcase our sensitivity analysis on the KDTree ball query hyper-parameters (radius). We observe how changes in the query radius impact efficiency and accuracy of our method when run with a data set of 5,000 instances. This sensitivity analysis is only relevant in the method that involves a screening step. It can be seen that increasing the query radius beyond a certain point yields decreasing marginal gains in accuracy while significantly increasing the computational cost. As a result of this, it is advisable to tune this hyper-parameter for an optimal trade-off between efficiency and accuracy.

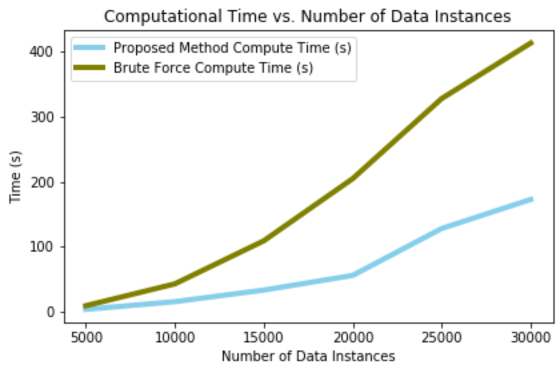


Fig. 3. Computational Time vs. Number of Data Instances

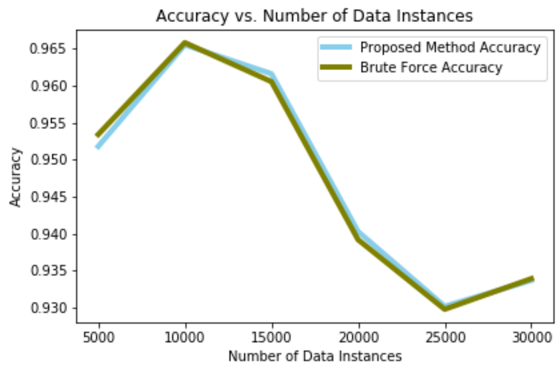


Fig. 4. Accuracy vs. Number of Data Instances

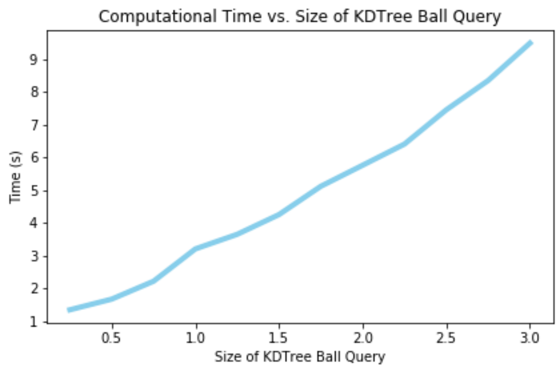


Fig. 5. Computational Time vs. Size of the KDTree Ball Query

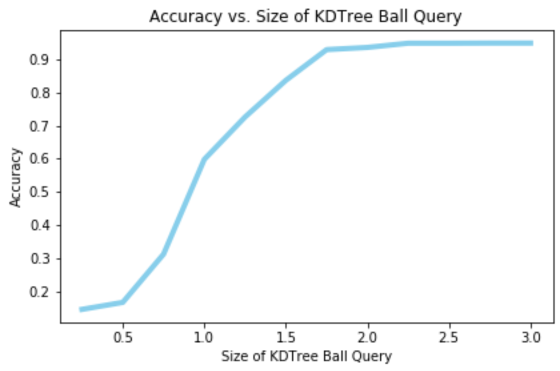


Fig. 6. Accuracy vs. Size of the KDTree Ball Query

4 Discussion

The goal of our screening algorithm is to assist speed up the time it takes to solve the address matching problem while minimizing loss in accuracy. By using a brute force method, N^2 operations must be performed if we do not account for repeating pairwise comparisons, and $\frac{N(N-1)}{2}$ operations if we ensure that no pairwise comparisons are repeated. When our screening method is applied, we are able to reduce this number because our screening algorithm subsets the number of pairwise comparisons that need to be performed for any given item. For small data sets, the computation time does not differ significantly, but as the number of instances increases, the difference in computation time becomes significant. This is highlighted in Figure 1. Both curves appear to be exponential, but the rates of time change relative to number of instances are different.

Improving computation time only has value if we are able to maintain high levels of accuracy. In Figure 2, we show that the accuracy in our implementation does not noticeably deviate form the base case brute force method. Regardless of method, we see that for more instances, the accuracy decreases, but our data does not show monotonicity in terms of the trade-off between accuracy and number of instances in the data. We suspect that this could simply be a matter of uncertainty as the max range in accuracy we assess, regardless of number of instances is only approximately 3.5% $(93\% - 96.5\%)$. If accuracy is too low, we believe improvements can be made by using a distance function that is more robust than our modified levenshtein distance.

In our screening algorithm, we utilize KD-Trees to generate our subsets of data to compare. The objects found within a specified radius around a point in the KD Tree is the subset of data points that should be pairwise compared to the address acting as the center point. This radius hyper-parameter can thus impact our accuracy and computation time as it determines what objects we perform our similarity comparisons on and how many comparisons should be made for a given object. Figure 4 shows that the computation time increases as our ball radius increases. The figure shows a generally linear scale up in computation time for larger hyper-parameter value. Accuracy however follows more of a sigmoid shape that plateaus after increasing the hyper-parameter to 1.75. These figures help us better understand the trade-off made between accuracy and computation time when we change the hyper-parameter.

4.1 Code Level Optimization

A variety of steps were taking to enable optimizations in our algorithm. Most notably, we ensured that no redundant calculations were performed when using the brute force approach or our approach. This significantly sped up the matching process. Further, we utilized a package for calculating Levenshtein distances (the most computationally costly element of our code) that was compiled in C. This significantly improved the speed of the implementation compared to code written natively in Python.

4.2 Limitations

There are several limitations to this methodology as it stands. First, there is a significant limitation in the severity of typos that can render an instance unmatchable. One example of this might be a street name with multiple typos. At a certain point, it is impossible to determine whether a large Levenshtein distance between street names indicates a severe typo, or a completely different street name. Another example is a typo in an address number. Many of these typos could match to legitimate addresses on the street, making it impossible to detect. Further, even if it didn't match to a legitimate address, it would be immensely difficult to differentiate between possible true addresses it could match to. A second limitation of this approach is that even though it is a significant improvement over brute force, its complexity isn't O(N), which can limit applicability for very large datasets. Last, the use of Levenshtein distance as our distance and similarity metric has large computational costs for longer strings. Because of this, for longer street names and inclusion of other address attributes like city or state, computational time may increase substantially.

4.3 Future Work

There are a number of potential improvements to this methodology that might be examined. First, the use of FastMap to produce vectors of greater than 2 dimensions might be examined. The trade-off between accuracy and computational cost of this dimensional increase might be examined. Second, parallel processing might be implemented in order to speed up the task of producing comparisons. Third, work might be done to develop a methodology that is capable of running O(N). And finally, using a different distance function may yield results that are more accurate and more computationally efficient. A distance function that is more aligned that maps typo errors based on keyboard layout, over just Levenshtein distance for strings could. prove to be more useful.

References

[1] SciPy documentation and package for KD Trees: https://docs.scipy.org/

[2] Levenshtein Distance Python Package: https://pypi.org/project/python-Levenshtein/

[3] Liron Cohen and Tansel Uras and Shiva Jahangiri and Aliyah Arunasalam and Sven Koenig and T. K. Satish Kumar, *The FastMap Algorithm for Shortest Path Computations*

[4] Christos Faloutsos and King-Ip Lin, *FastMap: A Fast Algorithm for Indexing, Data-Mining and Visualization of Traditional and Multimedia Datasets*

[5] Babar Nikhil, *The Levenshtein Distance Algorithm* https://dzone.com/articles/the-levenshtein-algorithm-1

Appendix
Instructions for the Included Code

There are two shell scripts that will perform the analysis given in this paper. First, CompCostEval.py will generate information regarding the computational cost of the proposed method and the brute force approach as the number of data instances is increased. Second, BallSizeEval.py will generate information regarding the computational cost and accuracy of the proposed method as the size of the KDTree query radius is changed.

The dataset that is used for this work is called los_angeles_addresses.csv and contains addresses for LA County. This is then segmented down to just those for La Mirada, and a subset of those is used in each shell script.

There are a number of files containing functions that are called from within the shell scripts. First, FastMap.py contains our implementation of the FastMap algorithm, as well as the distance function that utilizes Levenshtein distances to determine distances between addresses. Second, FMKDT.py contains the function for performing our search methodology. Third, typo.py contains the function for loading the los_angeles_addresses.csv dataset and introducing duplicate values with errors.

It should also be noted that Levenshtein distances are implemented using the python-Levenshtein (0.12.0) package, which must be separately installed.

Appendix
Contributions

Anthony Nguyen developed the problem statement, reviewed literature, and ensured implementation reflected contextually relevant assumptions and constraints. Further, Anthony handled overall system architecture, wrote code for FastMap, assisted in development of test sets, assisted in developing strategies for search functions, reviewed final code, and led the final write-up.

Nathan Decker led code development and was the primary writer of all code including, but not limited to, the search functions, testing data set construction, distance functions, shell scripts, and algorithm testing. Further, Nathan assisted with FastMap functionality, developed/coded the tests of the methodology's functionality and hyper-parameter tuning, assisted and reviewed the write-up, and generated figures.