# Adaptive Fuzzy String Matching

**Aja Klevs**
Center for Data Science
New York University
ak7288@nyu.edu

**Laureano Nisenbaum**
Center for Data Science
New York University
lvn218@nyu.edu

**Adrian Pearl**
Center for Data Science
New York University
ap3465@nyu.edu

**Orion Taylor**
Center for Data Science
New York University
ojt212@nyu.edu

## Abstract

Approximate string matching persists as a machine learning problem with no universal "solution". Despite the pervasiveness of string mismatch as an impediment to text data processing activities, the field of available string matching tools remains somewhat disparate, often tailored to be application-specific and difficult to evaluate at scale. We present three promising potential generic solutions to the string matching problem in this paper: (i) a supervised learning approach, with a combination of relevant string distance metrics as features and iterative Human-in-the-Loop (HitL) training set augmentation; (ii) a batched hierarchical clustering method with text vectorization; (iii) a naive ranking and clustering approach. We demonstrate their efficacy by comparing them on a large, difficult many-to-many string matching task involving unprocessed organization names pulled from two sources, and discuss combining them into a multistage modeling method as a potential future course for refinement.

# 1 Introduction

The approximate string matching problem boils down to a simple question: presented with two strings representing the same object, can a machine correctly identify them as matches (and conversely, presented with two strings representing different objects, can the machine correctly discern)? A basic example, in the vein of the data sets we explore in our analysis, might be:

"j.p. morgan chase & co." $\rightarrow$ "JP Morgan"

At first glance this seems to be a simple task, but it quickly grows difficult as additional considerations are introduced. How clean is the input data? How many strings are being matched? Is the matching task directional (i.e. are we trying to match strings from one set to another)? Are there multiple strings representing each object in one pool? Both pools? On top of these challenges, there may be domain particularities in the data that affect the nature of the strings; in the case of organization names, the data may be rife with misleading acronyms, commonalities (organization type words that pack the strings, e.g., "corporation", "inc.", etc.), and context-specific complications like hierarchical considerations (e.g., should we label "Planned Parenthood" and "Planned Parenthood of New York" as a matching pair? "Planned Parenthood of New York" and "Planned Parenthood of California"?).

A more comprehensive expression of the requisite task might look something like the matrix below, which could ideally encompass simpler cases like the one-to-one match case expressed above.

| | JP Morgan | j.p. morgan chase & co. | Jpm & associates | JPM |
|---|---|---|---|---|
| JP Morgan | 1 | 1 | 0 | 1 |
| j.p. morgan chase & co. | 1 | 1 | 0 | 1 |
| Jpm & associates | 0 | 0 | 1 | 0 |
| JPM | 1 | 1 | 0 | 1 |

Table 1: Sample two-dimensional binary label matrix for variations of "JP Morgan"

The current landscape of prepackaged tools to assist in fuzzy string matching is a heterogeneous reflection of the contextual variety expressed above. The most commonly used out-of-the-box tool is likely SeatGeek's FuzzyWuzzy [1] package, which is not a machine learning model at all, rather a simple and highly scalable implementation of one-to-one string comparison based on the Levenshtein string distance metric (discussed in greater detail below).

In general, common methods demonstrate variety at least in terms of (i) string similarity metric choice (e.g., Levenshtein vs. Jaro-Winkler distance, etc.), (ii) modeling methods (e.g., whether to utilize typical machine learning methods, and if so, algorithm choice considerations) and (iii) implementation goals (e.g., whether to consider two distinct lists, binary match classification vs. top-x similarity rankings, etc.). We begin by providing background on string similarity metrics.

## 1.1 Background: String Distance Metrics

String similarity underpins all attempts to classify a relation of two strings, and it has been explored since the advent of programmatic text processing. As such, string distance metrics are abundant, and each is worth considering. In general, string distance metrics can be divided into (i) edit distance-based metrics (which generally involve single-character operational quantification), (ii) token-based metrics (that require deconstruction of strings as part of pre-processing into n-grams in the manner typically associated with language processing tasks), and (iii) sequence-based metrics (which seek to quantify based on matching sequence length and volume).

Levenshtein distance, for instance, quantifies string similarity as the smallest number of adjustments (insertions, replacements, etc.) required to reduce two strings into a single one.

This metric might logically address discrepancies such as typos (e.g., "JP Morgan" $\rightarrow$ "JP Morgxn" merely requires one substitution), but might fall short, for example, in the context of acronym-ization ("JPM" $\rightarrow$ "JP Morgan" would require a substantially greater number of steps in this case).

[1] https://chairnerd.seatgeek.com/fuzzywuzzy-fuzzy-string-matching-in-python/

$$\operatorname{lev}_{a,b}(i,j) = \begin{cases} \max(i,j) & \text{if } \min(i,j) = 0, \\ \min \begin{cases} \operatorname{lev}_{a,b}(i-1,j) + 1 \\ \operatorname{lev}_{a,b}(i,j-1) + 1 \\ \operatorname{lev}_{a,b}(i-1,j-1) + 1_{(a_i \neq b_j)} \end{cases} & \text{otherwise.} \end{cases}$$

Equation 1: Levenshtein distance formula

Another common metric for string similarity is Jaccard distance, which measures n-gram set overlap ratio between two strings. This metric naturally requires tokenization as a pre-processing step, and may be particularly effective in handling cases where similar strings are ordered differently (e.g., "JP Morgan" → "Morgan, JP").

$$JaccardIndex = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|}$$
$$JaccardDistance = 1 - JaccardIndex$$

Equation 2: Jaccard distance formula

A third metric we highlight from background research is the Jaro-Winkler distance, which examines character index proximity, weighting matches earlier in the sequence more heavily. This order-dependent case may be better-suited to capture partial matches, where one string is an incompletely-recorded version of another (e.g., "JP Morg" → "JP Morgan").

The Jaro distance $d_j$ of two given strings $s_1$ and $s_2$ is

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

Equation 3: Jaro-Winkler distance formula

One last commonly-used metric worth mentioning is cosine similarity, which can be applied to any vector pairing: in our case, given two vector representations of two words (A & B below), similarity is evaluated as an equation of their dot product and magnitudes, where $A_i$ and $B_i$ are components of vectors $A$ and $B$ respectively.

$$\cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2}\sqrt{\sum_{i=1}^{n} B_i^2}},$$

Equation 4: Cosine similarity formula

The resulting similarity ranges from -1 (diametrically-opposite strings) to 1 (identical strings), with 0 implying orthogonality or lack of correlation, in-between values indicating intermediate degrees of similarity or dissimilarity.

Additional metrics not specifically defined above but noteworthy in our exploration include: Sorensen-Dice distance and the Tversky index. As with the above, each has weighting strength particular to some kind of matching case, with weakness in others.

### 1.2 Background: Human-in-the-Loop

The Human-in-the-Loop (HitL) methodology crystallizes the utility of a human operator when developing and improving a traditional machine learning model. This begins with human labelling of data to create or supplement training data, human involvement in hyper-parameter tuning and providing "feedback" to the model, ideally to curb over-fitting or help the model reallocate feature weights to deal with edge cases.

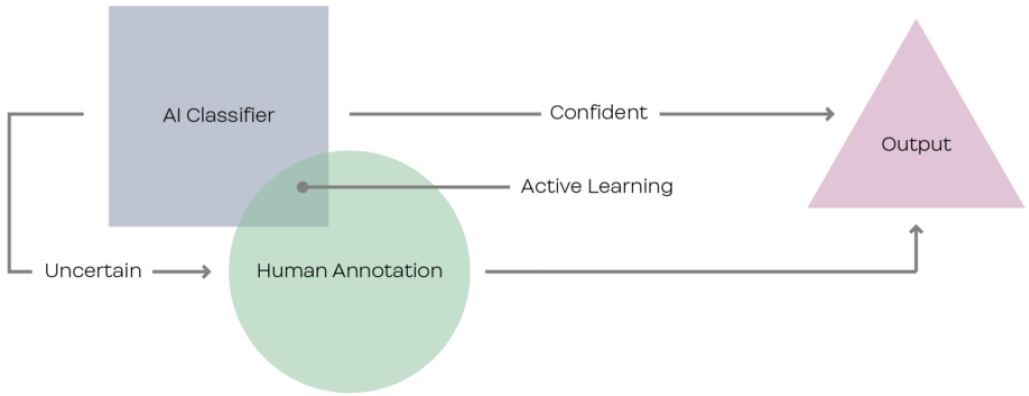


Figure 1: Human in the Loop methodology diagram

Theoretically, an iterative HitL approach (where the human selects model outputs methodically for evaluation, feeds adjusted results back into the training set, the model is refined, etc.) can lead to performance improvement with each progressive round (although this requires a sufficient pool of *un*labeled data), and enables learning in a batched, controlled manner superior to the typical single-batch train/val/test approach.

A simple cyclic flow here might be: (i) model trains on "obvious" cases (i.e. those that are measured to be very similar or drastically different); (ii) human reviews partial validation set results; (iii) human selects edge incorrect labels (e.g., in a Levenshtein-derived model cases where edit distances are large but strings represent the same concept) and relabels, feeding relabels into augmented training set; (iv) model is retrained; (v) the process repeats.

### 1.3 Background: Existing Tools

There is no single tool with substantial "market share" in the domain of adaptive string matching; the closest thing is likely FuzzyWuzzy which, as mentioned above, provides a relatively bare-bones, one-to-one Levenshtein ratio measurement package available in Python, with a few optional arguments (e.g., single-character vs. space-delimited word tokenization, "full" vs. "partial" ratio choices to addresses punctuation, etc.). We discuss our specific implementation in greater detail later.

The bulk of additional tools tends to be either similarly basic, or else customized for real-world (primarily business) applications. Other methods that did not actually employ loss function-based learning algorithms were generally tokenization methods (e.g., a TF-IDF plus cosine similarity combination that we wound up using as our standard for comparison) or slightly-varied manifestations of some of the string distance metrics described above. The more complex methods tended to involve layers of data collection, manipulation and ensembled modeling that were likely too narrowly scoped to be reasonably replicated or even to perform well in new domains.

## 2 Data Sets

We approached this problem with a single test case consisting of two messy sets of strings requiring many-to-many mappings. The data sets represented overlapping organization names as obtained from two sources: co-signers of *amicus curiae* briefs submitted to the United States Supreme Court (referred to as the "Amicus" data set going forward), and Adam Bonica's Database on Ideology,

Money in Politics, and Elections (DIME)[2], which tracks organizations that have donated to the campaigns of candidates for federal office (referred to as the "Bonica" data set going forward).

We received the data as CSV files with a manifest stating that the Amicus data set ($n$ = 13,940) contained unique organization names and the larger Bonica data set ($n$ = 1,332,470) had been pre-cleaned and could be mapped to the Amicus set, but upon further inspection we determined that neither source was sufficiently prepped (i.e. within each and between each, multiple strings might refer to the same organization). We also received a small "match-dense" file of 453 positive-match tuples derived manually from many-to-one mappings between Bonica to Amicus, created via Amazon Mechanical Turk (MTurk).

Broadly, in the Amicus and Bonica data sets we could describe sources of mismatch as:

- data recording problems (typographical errors, phonetic misrepresentation, truncation, problems with spacing, acronym-ization and nicknaming, word reordering, etc.);
- organization classification problems (circumstantial interpretation of denominational terms like "corporation" and "llc", associated differences in prefixing and suffixing, inconsistent use of regional clauses as mentioned in Section I, etc.).

As the data in the Bonica case appeared to be parsed from documents that perhaps do not adhere to a congruous formula, this set contained cases that were even less clear than those mentioned above; "organization names" consisting of single letters ("a"), names, addresses, digits, etc. These cases made data preparation difficult and certainly affected overall performance in the sections to come.

# 3 Methodology

## 3.1 Baseline(s): *stringmatch_amicus.R*, FuzzyWuzzy & TF-IDF

The bundle of materials we were supplied included a preliminary model ("stringmatch_amicus") for the Amicus-Bonica matching case written in R. This model took the MTurk hand-matched set described above, shuffled pairs a few times to generate mismatches (e.g., 3 shuffles on 453 matched pairs would generate 1,359 training pairs in roughly a 3:1 mismatch-to-match ratio), gathered a few arbitrary distance metrics between each pair, then fed the vectorized metrics into an un-tuned random forest. The HitL aspect was hard-coded a few times (e.g., skimming for a few mismatch indices to hard relabel), with an arbitrary stopping criterion.

Our first task, at the directive of the project requester, was to decipher and translate this process into Python. After some adjustments to the underlying assumptions and test runs on larger string subsets from Amicus-Bonica, we determined that before proceeding (or addressing the HitL component of the task) we would first explore more time-efficient and easily-implemented tools like FuzzyWuzzy.

We initially decided to create a wrapper around the FuzzyWuzzy ratio that would take each string in the unique set of Bonica and output a "fuzz" score with each string in the unique set of Amicus strings. We created a looping function with input specs for threshold (floor probability-of-match value below which to discard pairs) and size of Bonica subset to consider.

We used this method in a broad exploratory sense at first; an example finding is summarized below:

[2]https://dataverse.harvard.edu/dataset.xhtml?persistentId=doi:10.7910/DVN/O5PX0B

| FuzzyWuzzy (Levenshtein) baseline | |
|---|---|
| len(subset(Bonica)) | 10,000 |
| len(set(Amicus)) | 13,940 |
| Tot. string comparisons | 139,400,000 |
| Threshold | 90/100 |
| Num. matches > threshold | 59 |
| Num. correct matches | 40 |
| Precision | 0.677 |
| Time elapsed | 0:12:34.11 |

Table 2: FuzzyWuzzy baseline configuration and results

In this test we compared 10,000 strings from Bonica against the Amicus set (139,400,000 pairwise combinations), and set our discard threshold at 90 (note: this is a very strict threshold in the Levenshtein formulation, and should generally capture either exact matches, typos or other slight character mismatches like similar acronyms). As noted above, across more than 100 million comparisons, only 59 were passable at this threshold (both an indication of the sparsity of "obvious" matches in the data and an indication of the breadth of the "edge case" volume). A quick manual check of labels here confirmed the variety of problematic cases anticipated beforehand (acronym similarity cases like "rb international" $\rightarrow$ "dba international", ambiguous locality cases like "nd hospital association" $\rightarrow$ "nevada hospital association" and "ashland area chamber of commerce" $\rightarrow$ "allegan area chamber of commerce", etc.).

We also came across and wound up settling on an implementation that used cosine similarity on vectors derived by standard TF-IDF tokenization (a niche package from a public repository with a "StringMatch" object available, TF-IDF will be explained in more detail in section 3.4). The primary benefit of this package was the speed of calculation of cosine similarity using a specific sparse matrix multiplication algorithm published by ING Bank. Results are below.

| StringMatch (TF-IDF + Cosine) baseline | | |
|---|---|---|
| len(subset(Bonica)) | 10,000 | 100,000 |
| len(set(Amicus)) | 13,940 | 13,940 |
| Tot. string comparisons | 139,400,000 | 1,394,000,000 |
| Threshold | 90/100 | 90/100 |
| Num. matches > threshold | 32 | 293 |
| Num. correct matches | 31 | 286 |
| Precision | 0.969 | 0.976 |
| Time elapsed | 0:00:02.24 | 0:00:12.94 |

Table 3: StringMatch baseline configuration and results

This method showed a run-time improvement of 375x at 1.4 million pairs, and did not display linear or exponential increase with scale (as above, a 10x increase in input data volume only resulted in a 6x increase in run-time). In both cases we also discarded scores *above* 0.99, since these implied effectively perfect matches. The TF-IDF baseline also showed improvement in precision (a few tricky cases still snuck through, e.g., "noble & nobl e" $\rightarrow$ "noble co.", likely since the cosine computation was based on vectors of character sequence n-grams).

The baseline evaluation processes above gave us a heuristic sense of the compositions of our input data sets, and a valid comparator for precision across modeling methods, but due to the data volume and match sparsity, made it difficult to comment on or measure recall (ability to discern negative matches). This influenced our decision to create a "closed system" two-dimensional matrix structure for test performance evaluation (discussed more below).

### 3.2 Brief Summary: Pre-Processing

In the course of our baseline exploratory analysis it became clear that some somewhat domain-specific adjustment of inputs would be required to proceed. We reviewed similar ad-hoc pre-process jobs online, and created a reference mapping based on the following cleaning steps:

- uniform-izing casing and removing inconsistent spacing;
- removing most punctuation; conforming some symbols (e.g., "&" $\rightarrow$ "and") and removing parentheticals;
- trimming some obvious errors (e.g., numeric-only strings like "005" that were likely parsed out of phone numbers or addresses accidentally).

We also found and employed a Python package called *cleanco* that used an external dependency dictionary of common organizational legal entity terms (groups like "l.l.c.", "llc", "ltd.", "limited liability company", etc.) and offered a string stripping tool to standardize along these lines. Finally, after getting green-lighted by the sponsoring professor, we experimented with cleaning some geographical distinctions; in the professor's particular use case these were often linkages that it would be beneficial to treat as manifestations of the same underlying entity. So, for example we stripped all U.S. state-inclusive strings of the form " of New York" or " of NY", etc. The end result of this pre-processing was a theoretical mapping to reduce the size of the larger Bonica set by roughly 5% (1,332,470 to 1,266,742), helping us combat run-time issues.

### 3.3 Random Forest for Supervised Learning

Expanding on the initial model presented to us, our first course of action was to develop a supervised modeling case. Our only hand-labeled data set consisted of only pairs of confirmed matches to begin with, so to create a full training set of both positive and negative examples and to control for class skew, we shuffled the Amicus strings and repaired them to create incorrect matches, and appended these to the data set. The training procedure for the random forest was as follows: begin by converting each pair of strings into a feature vector consisting of multiple distance metrics and train a random forest classifier on these vectors, tuning hyper-parameters to achieve the optimal validation performance. Next, predict the match probability for a batch of random pairs pulled from our unmatched pool of data, and output the 100 most likely matches according to the classifier. The user would then validate these matches, labeling them either correct or incorrect, and rerun the training procedure. The subsequent iteration would then train a new classifier using both the initial training set and any previous batches of human-validated results (thus the "human in the loop").

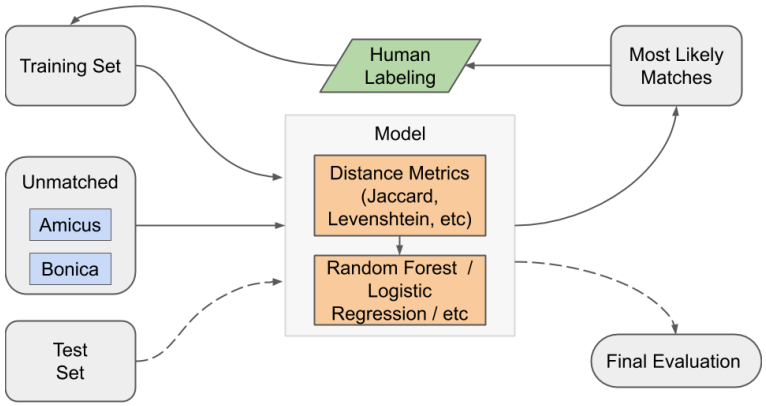


Figure 2: HitL random forest training workflow; solid arrows represent iterative processes (training, predictions on unmatched data, validation), dashed arrows represent one-time evaluation

While our initial experimentation used several different distance metrics, after some basic analysis we altered the feature distances to ensure that we did not include any two distance metrics that were too algorithmic-ally redundant (for example, the Jaccard index is equivalent to the Sorensen coefficient). A summary of respective feature importances is as follows:

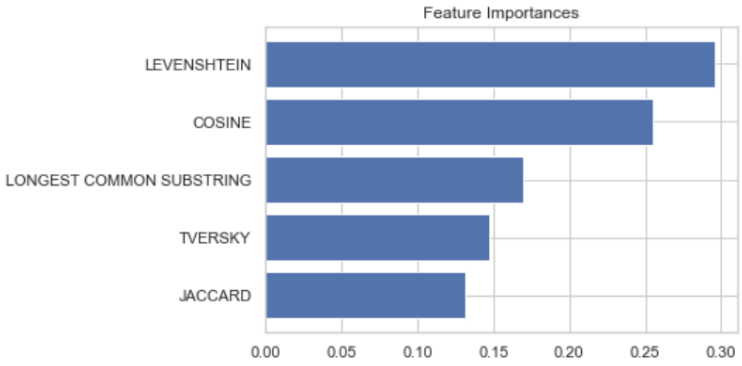


Figure 3: Random forest feature importances

To simplify our training procedure, we elected to use only the Levenshtein, cosine and longest common sub-string distance metrics. As our experimentation progressed to predict match probabilities for larger and larger sets of pairs, we realized that the Levenshtein distance was the most computationally expensive of these and had created a bottleneck in our processing pipeline. We then replaced the Levenshtein distance with the Jaro-Winkler distance and noticed no significant effect on performance.

### 3.4 Clustering for Unsupervised Learning

While embarking on our random forest HitL iterations, we could see that this approach would generate accurate matches on the whole. What was less clear was how many of the total number of matches in our data set we would be able to uncover. In other words, we could see that our model would have high specificity, but we were unsure of its sensitivity. In addition, we wanted another model with a completely different structure to which we could compare our results.

Given our concerns about the viability of an approach requiring substantial labeled training data, we explored unsupervised clustering options in an attempt to produce a more sensitive model. We used two distinct clustering methods. For the first we referenced Ven Den Blog and attempted to cluster using k-means on a term frequency-inverse document frequency (TF-IDF) vectorization of the data set. For the second clustering technique, we used a more naive approach which involved finding and ranking the count of each word that appeared in the data set and clustering terms by their rarest word.

### 3.5 Clustering with K-Means and TF-IDF

In this case, TF-IDF is a technique which represents each term in the data set as a sparse vector, where each component represents a word in the vocabulary. To compute the TF-IDF representation for word $w$ in term $t'$, we use:

$$TFIDF(w)_{t'} = \frac{|\{w \in t'\}|}{len(t')} log(\frac{|\{t\}|}{|\{t : w \in t\}|})$$

Equation 5: TF-IDF representation formula

The second term ensures that words that occur less frequently in the data set are weighted higher than more common words. Once the data set was converted to the TF-IDF representation of each term, we used k-means to cluster. However, we had to overcome several challenges to do this effectively.

Due to our lack of labels, in general we had no systematic way to estimate how many matches exist in our data set. Therefore, we had no way to know how large our clusters should be on average or how many centers we should cluster. Since we were attempting a high-recall model, we erred on the side of larger clusters in the hope that we would have fewer false negatives (in exchange for more false positives). Ultimately we decided to use approximately 15 terms per cluster on average.

We were still faced with the volume of our data: there were around 1.3 million terms in the combined set of our Amicus and Bonica data sets, which would require clustering with about 87,000 centers on high-dimensional data points. This was computationally infeasible, so instead we devised a system to recursively cluster the data into workable batches, using the *scikit-learn* MiniBatchKMeans package to further reduce computation time.

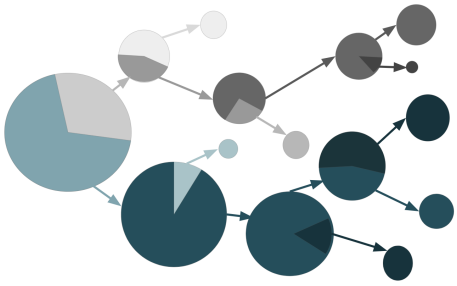

Figure 6: Visualization of Recursive K-means Clustering

We would first cluster the entire data set with two centers, thereby splitting it into two smaller sets. We would cluster with two centers on each of these, and so on until each leaf of the our data was less than 100,000 terms. We ended up with 258 batches, with a mean size of 5,212 terms and a median size of 981 terms. Some batches had only one term, and our largest had 99,938 terms.

For each of the above batches we clustered with multiple centers. If the batch had 15 or fewer terms, we left it as is and considered it one cluster. If there were more than 15 terms in the batch, we clustered again using a number of centers proportional to the size of the batch (see Figure 4). In general we clustered with $ceil(\frac{batch\ size}{15})$ centers, but for the largest batches we capped the number of centers at 1,500.

As was expected, the size of our clusters varied from one term to thousands. A few randomly selected clusters are visualised in Figure 5. Aside from some of the huge clusters, all of these had at least one word in common.

| | term | label |
|---|---|---|
| | amt interior design inc | 6 |
| | kobi karp architecture & interior design inc | 6 |
| | carolyn franklin interior design inc | 6 |
| | custom interior design inc | 6 |
| | william r eubanks interior design inc | 6 |
| | jan turner hering interior design inc | 6 |
| | barnett jonathan interior design inc | 6 |
| | forum architecture & interior design inc | 6 |
| | alene workman interior design inc | 6 |
| | nu image flooring & interior design center inc | 6 |
| | aircraft interior design inc | 6 |
| | denny interior design inc | 6 |
| | equilibrium interior design inc | 6 |
| | galleria interior design inc | 6 |
| | lisa gielincki interior design inc | 6 |
| | bridewell interior design inc | 6 |
| | k b f interior design inc | 6 |
| | bright ideas interior design inc | 6 |
| | interior design force inc | 6 |
| | nu-image flooring & interior design center inc | 6 |
| | susan doris interior design inc | 6 |
| | interior motives design &or inc | 6 |
| | leslie herpin interior design inc | 6 |
| | myriam hutchinson interior design inc | 6 |

| | term | label |
|---|---|---|
| | southern unreadable recovery inc | 6 |
| | southern recovery & towing inc | 6 |
| | southern paper recovery inc | 6 |
| | southern credit recovery inc | 6 |

| | term | label |
|---|---|---|
| 1 | gieln enterprises | 7 |
| 2 | r skelton enterprises | 7 |
| 5 | fordham enterprises | 7 |
| 6 | dlo enterprises | 7 |
| 7 | cooling enterprises | 7 |
| ... | ... | ... |
| 6608 | kta enterprises | 7 |
| 6609 | batchelor enterprises | 7 |
| 6612 | scoby enterprises | 7 |
| 6613 | how jan enterprises | 7 |
| 6614 | croweer enterprises ins | 7 |

4523 rows × 2 columns

| | term | label |
|---|---|---|
| | industrial petro-chemicals inc | 1 |
| | industrial chemicals inc | 1 |

| | term | label |
|---|---|---|
| | combined management inc | 0 |

| | term | label |
|---|---|---|
| 0 | district 21 democrats | 8 |
| 12 | 41st district democrats | 8 |
| 21 | virginia democrats - 05th cong district | 8 |
| 34 | house district 12 democrats | 8 |
| 42 | 3rd district democrats | 8 |
| ... | ... | ... |
| 1321 | district 25 democrats | 8 |
| 1379 | 36th district democrats | 8 |
| 1384 | district 22 democrats | 8 |
| 1391 | first district democrats | 8 |
| 1397 | second congressional district democrats | 8 |

64 rows × 2 columns

| | term | label |
|---|---|---|
| | castro enterprises | 8 |

Figure 7: A random sample of 7 clusters obtained via K-Means with TF-IDF vectors

One limitation that became immediately obvious was that this method would not handle misspelled words effectively. In an attempt to adjust for this, we tried a version of the TF-IDF approach with character-based representations of each term (instead of word/n-gram-based). While we did not evaluate this method on any test set, the observed clusters were much worse than the word-based clusters. This might be because the Amicus-Bonica data set has relatively few misspelled words (its errors tend to take forms other than obvious typos). When running the TF-IDF program, we added a user input argument at the top of the process to gather from the user whether they believed the data set might have proportionally many misspelled matches. If the answer was "no", our program used word-based TF-IDF, and if "yes" we used character-based vectorization.

### 3.6 Naive Rank Clustering

During our first attempt at clustering we realized that the TF-IDF K-means method did not always cluster on the most "important" word. For instance, 'out island properties llc' was clustered with 'cenla properties management llc' but not with 'out island properties inc', indicating that for this example our model placed more importance on the token 'properties' than on 'island'. This could be because TF-IDF is better-suited for longer documents, in which a word can occur multiple times throughout. This gave rise to a different approach that would eliminate the problem of picking the number of cluster centers, and might yield better results.

For our second clustering method, we would count the number of occurrences of each word in the total set. We would then eliminate all words which only occurred once, and rank the rest in order of count in data set. Starting with the rarest words, we would create clusters of terms containing that word, and then remove them from the data set so that there would be no repeated terms across clusters. If not all of the terms were clustered by the time we got through words with at least two occurrences in the data set, we might be able to conclude that the remaining terms had no words matching any other words in the data set.

For this method we made 42,096 clusters; these were on average smaller than those generated by the TF-IDF k-means approach. These clusters had a mean size of 32 terms but a median size of just 2 terms. Exactly 12,294 of the terms (just shy of 1% of our total strings) ended up clustered by themselves. Even more (31,562 terms, or 2.3%) were clustered with just one other term. Our largest cluster contained 18,262 terms.

| term | label |
|---|---|
| senate district47 | 11.0 |
| cat4 llc | 11.0 |
| senate district41 | 11.0 |
| t4 development llc | 11.0 |
| dot429 inc | 11.0 |
| print4less | 11.0 |
| mt40 llc | 11.0 |

| term | label |
|---|---|
| levymba sphr deborah | 15.0 |

| term | label |
|---|---|
| russo picciurro agency | 8.0 |
| russo picciurro maloy | 8.0 |

| term | label |
|---|---|
| rodolitz assc | 3.0 |
| rodolitz associates | 3.0 |

| term | label |
|---|---|
| trinty stemmons land corp | 5.0 |
| stemmons law firm pc | 5.0 |
| systemmatic inc | 5.0 |
| stemmans inc | 5.0 |

| term | label |
|---|---|
| silver garvett and henkel pa | 10.0 |
| silver and garvett pa | 10.0 |

| term | label |
|---|---|
| geosling chiropractic clinic pc | 7.0 |
| geosling chiropractic clinic | 7.0 |

| term | label |
|---|---|
| steve bissen house district 31b cmte | 7.0 |
| margaret hanson house district 31b cmte | 7.0 |

| term | label |
|---|---|
| gottschlich and portune | 15.0 |
| gottsch cattle co | 15.0 |
| gottschalk inc | 15.0 |
| gottschalk 483 william | 15.0 |
| gottsch feeding corp | 15.0 |
| candice m gottschalk candice m gottschalk | 15.0 |
| law office of robert gottschalk | 15.0 |

Figure 8: A random sample of 9 clusters obtained via naive rank method

# 4 Evaluation

## 4.1 Random Forest for Supervised Learning

We had two possible evaluation criteria for the supervised random forest approach, the first being through the iterative training process itself. While validating the predicted matches output by each iteration of the model, it was straightforward to calculate the percentage of predicted matches that were correct. In fact, the intended stopping criterion for the HitL process is that, after some number of iterations, this percentage becomes sufficiently high and stabilizes. It is tempting to think of this as a way to estimate the precision of the model, but this runs the risk of overestimating the model's performance on the real, larger full data set. A more realistic estimate of the model's performance would validate its predictions on a random subset of examples rather than on only the few examples it deems likeliest to be matches; however, this approach is largely infeasible, as the empirical percentage of matches for our data set is so low that it would require a subset of thousands of random pairs to include a significant number of matches (as described in Section 3.1, one might note that 1.4 *billion* string pairings resulted in only a few hundred high-confidence matches via Levenshtein and cosine similarity-supported tools).

Evaluating the random forest model with this method yielded largely mixed results. There were several exact or nearly exact matches between the two sets that the model easily identified. Beyond this, approximately 1-2% of each batch of predictions contained true matches. This number may seem underwhelming, but it is difficult to discern the reason for this performance. It is possible that the model failed to locate the true matches in the data set, but it may be that the number of random pairings was simply too large and the frequency of matches too low, so that every subset of pairings we tried that the model was able to process in a reasonable amount of time had very few matches.

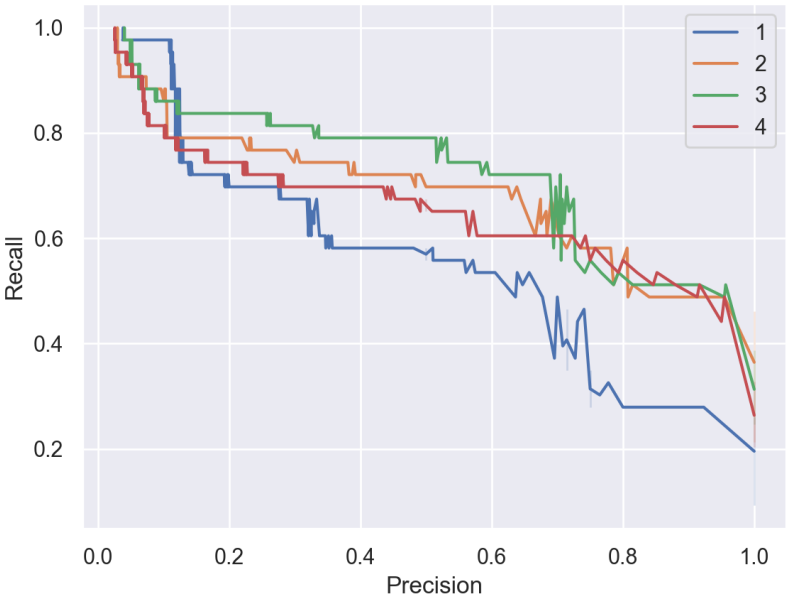


Figure 9: Precision vs. Recall for 4 HitL iterations (random forest retraining)

The second, simpler evaluation criterion for the random forest approach was to chart its precision vs. recall curve on our small, labeled test set of pairs. This test set was made by collecting every possible pairing of a random selection of 108 strings from both sets, so it was a good way of evaluating both the model's ability to identify matches and its ability to ignore non-matches. That being said, it had a much higher percentage of matches than the overall data set, so it was difficult to tell how the performance would scale (this was somewhat by design; in order to include *any* matches in the test set we had to randomly pull alphabetical chunks of varying sizes from the data set, the rationale

being that two alphabetically adjacent terms were more likely to represent the same concept due to the types of phenomena like organizational suffixes relevant to this domain).

Using this evaluation method, we were able to measure two important characteristics of the model: its overall, "out of the box" performance, and the change in performance after each HitL iteration. The results of this experiment (after we had narrowed our features to the three most quantifiably important and computationally efficient distance metrics) are shown in Figure 9. Human-in-the-Loop training iterations demonstrably improved the testing performance, but this effect quickly plateaued and actually reversed after the fourth iteration for reasons we could not identify.

An important question is whether the result that we generally observed — slight improvements in performance after two or three iterations — was simply due to the addition of more training data, or because of how that data was added. To rephrase: would we have observed the same performance boost adding random examples to the training set, or was it that examples were added that specifically addressed the models weaknesses? From the perspective of a potential user of this system this may not be inconsequential; the user will have to label sufficient examples to train the model at some point, be it entirely beforehand or in between HitL retraining iterations. Regardless, this remains an interesting question and a topic worth further investigation.

### 4.2 Unsupervised Clustering Methods

To evaluate the clustering methods, we used two test sets. The first was the same test set used to compute results on the random forest method, and the other was a test set generated from the data used to train the random forest, as this was part of a rather small pool of human-confirmed labeled matches. We did not use the latter test set to evaluate the random forest, as it is not good practice to evaluate performance on training data. The former test set considered every combination of 108 terms, 43 of which were matches. The latter considered every combination of 125 terms, 66 of which were matches. Therefore the "match density" of the first test set was $\frac{43}{108^2-108} = 0.00372$, while the second test set had a "match density" of $\frac{66}{125^2-125} = 0.00426$. We incorporated this second "match-dense" test set since our original test set had many more terms within the same section of the alphabet, and therefore had many more similar terms and ambiguous cases. We used our best performing random forest in this table, at the threshold which maximized F1-score.

| | K-Means on TF-IDF | Naive Rank Clustering | Random Forest |
|---|---|---|---|
| Test Set Precision | 0.09 | 0.06 | 0.68 |
| "Match Dense" Precision | 0.03 | 0.36 | NA |
| Test Set Recall | 0.28 | 0.56 | 0.72 |
| "Match Dense" Recall | 0.10 | 0.80 | NA |

Table 4: Evaluation metrics for each method

We also generated some random matches from each of the clustering methods. To do this we picked a cluster at random, and if that cluster had more than two terms we picked two at random and considered them a match.

Clearly, the clustering models performed worse that the random forest in nearly all cases. By all metrics, the naive rank method performed better than the TF-IDF k-means method. Though we only show 6 randomly generated matches here, the rank method consistently output more sensible matches. As we suspected, both clustering methods had higher recall than precision by a significant margin. However, we underestimated the recall capabilities of the Random Forest which, it turns our could out-perform the clustering methods at most thresholds.

| Random Matches from Kmeans | | Random Matches from Naive Rank | |
|---|---|---|---|
| “mjs excavating” | “marfink excavating” | “vcwpc” | “vcw inc” |
| “bracewell & patterson l l p” | “bracewell & patterson l.l.p.” | “walfred associates” | “walfred associates memo” |
| “ng cfp chfc, jake s” | “gateway arms” | “fob” | No Match in Dataset |
| “bh tank works inc” | “b&t works inc” | “american aidspac” | “american aidspac” |
| “napal” | No Match in Dataset | “add” | “add inc” |
| “lower balarz, katherine m” | “garrett gear, katherine” | “amandas florest” | “amandas cove llc” |

Table 5: Random co-clustered matches from k-means (left) and naive rank (right)

Interestingly, the naive rank clustering model performed significantly better on the "match-dense" test set than it did on the generalized test set in both precision and recall. This could be because the ranking model was consistently "fooled" by the ambiguous cases in the alphabetically chosen test set. Or, the variation could have come from the different "match densities" of the respective test sets. Alternatively, it is possible that our test sets were simply too small and caused the models to be subject to large random variations in the evaluation. Our lack of labeled data consistently hindered our attempts at accurate evaluation.

# 5 Discussion

## 5.1 Evaluative Limitations

This investigation set out with an ambitious goal: to create a tool that could effectively solve the fuzzy string matching problem where relatively few labeled data points existed to begin with. The motivation behind this was that, while it would be feasible to label a large number of data points in a particular domain (such as Amicus-Bonica) to create a model to solve the task for that use case, it is reasonable to assume that this model’s usefulness might not be transferable to other domains. A general-purpose method for solving the problem without the upfront labor of labeling examples, however, would be immensely useful across nearly any domain.

The primary difficulty in the fuzzy string match scenario is that a model’s ability to perform adequately for a particular domain cannot be effectively measured without a sufficiently large set of labeled examples from that domain. Let us assume that a fuzzy string matching model can be fully evaluated by estimating the probability that a predicted match is an actual match (otherwise known as the *precision*) and the probability that an actual match will be labeled as such by the model (*recall*). It is quite straightforward to estimate the precision by predicting labels for a large number of randomly-selected pairs, isolating the relatively small number of predicted matches, and measuring the empirical precision on that subset. In a case like Amicus-Bonica, however, where the probability that a random pair is a match is well under 1%, the problem of estimating recall becomes virtually unsolvable without a sufficiently large labeled test set. This is because obtaining a reasonable estimate would require manually labelling thousands of predicted negatives to count the number that are actually matches (false negatives). This task is not only prohibitively labor-intensive for the general use case, but defeats the larger purpose of finding an automated solution to the string matching problem.

This challenge left us with a few different options on how to proceed. The first was to essentially abandon the task of measuring recall and focus on creating a tool that could identify a not-insignificant number of matches with high precision. The use case of such a tool would be a situation where the user wishes to find matches from a data set but is not concerned with doing so "exhaustively" — that is, where finding only the most obvious matches and leaving others un-found is acceptable. Such an intended application is very different from that of a tool like FuzzyWuzzy, where an event name

cannot simply be ignored because it is difficult to match to other events in the data set. Another, more feasible option was to create a small test set that would attempt to give a rough estimate of a model's recall performance. In either case, our specific task was inhibited by the quality of the data input; one cannot conceive of a machine learning solution that can identify the true object of a string such as "100", and in fact this touches the precedent task of removing strings that *should not* be matched in the first place (such a string is likely errant and would be just as well discarded). The combination of these confounding factors created an experimental path with significant (and based on our models' performances, perhaps somewhat ineluctable) obstacles.

### 5.2 Future Work

A potential future path for this tool could be a multi-phased ensemble-type modeling approach.

First we address the idea of generalized pre-processing: in theory, we could create a global library for external dependencies (like the cleanco tool we mention above), so that for each new string set domain the probability of an existing pre-processing reference increases (i.e. now that we've added the organization name processing reference, it exists for future implementations requiring access to this domain). We could then initialize a mapping dictionary to which to feed high-confidence matches (its first population would be the subset of initial strings that had different processed counterparts).

The second phase of this approach could piggyback existing high-speed high-precision tools like the StringMatch cosine similarity method we used as a baseline. Iteratively, and scaling the threshold for match classification down towards the boundary of match "ambiguity", we could run small batches for human label confirmation (i.e. set a threshold of 99/100 and feed all perfect matches to the mapping dictionary, scale the threshold to 95/100 and have a human confirm accuracy of pairs above this value, then feed these to the dictionary, etc.). At a certain point we'd approach the "turning point" below which we would begin to see failure of the basic cosine method. We would facilitate stopping here, and shift to a clustering approach.

After reducing our corpus by domain-specific pre-processing and picking off matches identified easily by cosine similarity, we could pass the remaining strings into the naive rank clustering model outlined above and, instead of using it as our ultimate classification model, featurize cluster memberships into each string vector. These could in turn be passed into a supervised random forest-type model. This entire process would, of course, require at least a baseline degree of data quality constraint, so as to minimize the occurrence of errant string input, and a sufficiently large matrix of pairwise labels for testing performance.

## 6 References

## References


R. Mozer, L. Miratrix, A. R. Kaufman, and L. J. Anastasopoulos, "Matching with text data: Anexperimental evaluation of methods for matching documents and of measuring match quality,"2018.

W. W. Cohen, P. Ravikumar, and S. E. Fienberg, "A comparison of string distance metrics forname-matching tasks," inProceedings of the 2003 International Conference on InformationIntegration on the Web, IIWEB'03, pp. 73–78, AAAI Press, 2003.

C. Wei, A. Sprague, and G. Warner, "Clustering malware-generated spam emails with a novelfuzzy string matching algorithm," inProceedings of the 2009 ACM Symposium on AppliedComputing, SAC '09, (New York, NY, USA), pp. 889–890, ACM, 2009.14

Mishra, Piyush Sagar. "An Ensemble Approach to Large-Scale Fuzzy Name Matching." *Medium*, BCG GAMMA, 28 Mar. 2019, https://medium.com/bcggamma/an-ensemble-approach-to-large-scale-fuzzy-name-matching-b3e3fa124e3c_References

"Super Fast String Matching in Python." *Bergvca*, Ven Den Blog, 14 Oct. 2017, https://bergvca.github.io/2017/10/14/super-fast-string-matching.html.

Bonica, Adam. "Database on Ideology, Money in Politics, and Elections (DIME)." *Harvard Dataverse*, Harvard Dataverse,26 Mar. 2019, https://dataverse.harvard.edu/dataset.xhtml?persistentId=doi:10.7910/DVN/O5PX0B.

Atlas, Aviad. "Hybrid Fuzzy Name Matching." *Medium*, Towards Data Science, 2 July 2019, https://towardsdatascience.com/hybrid-fuzzy-name-matching-52a4ec8b749c.

Ing-Bank. "Ing-Bank/sparse_dot_topn." *GitHub*, Ing-Bank, 29 Nov. 2019, https://github.com/ing-bank/sparse_dot_topn.

"Psolin/Cleanco." *GitHub*, Psolin, 26 Jan. 2019, https://github.com/psolin/cleanco.

Isma3il. "Supplier Names Normalization (Part1)." *Medium*, Medium, 14 Feb. 2018,https://medium.com/@isma3il/supplier-names-normalization-part1-66c91bb29fc3.

"An Overview of Fuzzy Name Matching Techniques." *Rosette Text Analytics*, Rosette Text Analytics, 9 Jan. 2018, https://www.rosette.com/blog/overview-fuzzy-name-matching-techniques/.