# Record Deduplication: Reference Pipeline and Common Pitfalls

Jon Downs

5/15/2025

## Introduction

[TODO] We end up referencing the ECHIDNA paper a lot. I need a standard way to refer to it as well as an explicit call-out that we will be talking about it a lot.

Record linkage and record deduplication are a common tasks in vital records and public health. This is due to the nature of how data are received. The Center for Health Statistics (CHS) receives data from hospitals, primary care offices, medical examiners, funeral homes, and birthing centers. To facilitate this work, CHS has a dedicated Linkage and Data Analysis team, or LIDA. LIDA's primary focus is to identify links with high accuracy, but demands for better script runtimes and modularity have grown alongsize the catalog of work. This is exacerbated by an anticipated loss of cloud computing power due to changes in the funding landscape. In 2025, LIDA partnered with the Data Science and Engineering team (DSE) at CHS to improve the performance and efficiency of some pipelines. This collaboration is ongoing, but many improvements have already been found.

LIDA previously created a tutorial version of a real production linkage pipeline [here](https://doh.wa.gov/sites/default/files/2024-01/346147-MachineLearningLinkageDemo.pdf). Anecdotally, others have since adapted this work for their own linkage processes. However, this pipeline is not optimized for performance, and it is not a reproducible example. A reproducible example is a combination of documentation, source code, and input data that allows someone to rerun a code example from start to finish and get exactly the same result. Reproducible examples are both a great resource for learning and proof that the code being provided works as described.

This document provides a performance-minded, reproducible guide to implementing a data deduplication pipeline. To start, we will walk through a reference data deduplication pipeline for a publicly available dataset. We use the term "reference pipeline" similarly to the concept of a [golden test](https://blog.thecodewhisperer.com/permalink/surviving-legacy-code-with-golden-master-and-sampling) in computer science. The reference pipeline is a constant set of inputs where the truth is known or well-approximated. It serves two key purposes. First, testing a script against a golden test confirms consistent behavior. This is useful when migrating or upgrading software used to run a process. Second, it creates a controlled environment to test out new modeling strategies. A reference pipeline can be converted into production with minimal changes. Reference pipelines and other code testing strategies are highly encouraged for these reasons. After this, we highlight performance pitfalls discovered during a review of our own linkage pipelines.

This document is designed for intermediate R users with a background in public health or the social sciences. The goal is not to achieve state of the art accuracy or optimal performance. Rather, it represents a base case framework that can be customized and optimized according to the needs of a project. We will highlight places where customization would likely improve the end product.

## Example Pipeline: Deduplicating a List of Records

To begin, we will describe the basic steps of our pipeline. Given a table of records, the goal is to create a common person identifier to identify which records correspond to the same individual. There are many decisions to be made when designing a a deduplication pipeline, such as blocking methods and model choice. An overview of such decisions is beyond the scope of this article. There are many great overviews readily available, such as this [white paper](https://www2.stat.duke.edu/~rcs46/linkage_readings/2014-SteortsBlockingComparisons.pdf) on blocking methods. CHS LIDA uses traditional blocking and supervised learning in most linkage pipelines. The primary models are support vector machines (SVM) and random forest (RF) models. Traditional blocking is a combination of exact and fuzzy matches on fields such as first name, last name, date of birth, and social security number. An exact match is one where the two fields are identical, and fuzzy matches allow some amount of pre-defined error between the two fields. For example, if you chose to block on exact DOB and a fuzzy match on last name, a John Doe born on 2/1/1980 would be in the same block as John Dove born 2/1/1980.

As previously mentioned, our final product is a reference implementation. In a proper golden test, exact replication is expected. Here, we have a stochastic process with a model. Proving the whole system catches enough true links with few false positives is sufficient for now. In real projects, revisit this testing strategy once the process is mature.

### Data Source

Data come from the RLdata10000 data source of the [RecordLinkage](https://cran.r-project.org/web/packages/RecordLinkage/RecordLinkage.pdf) R package. The dataset is artificially-generated records based on the most common names in Germany, and 10% of the records are duplicates that should link to another record in the dataset.

### Dependencies

* R 4.3.2
* The following R packages as well as their dependencies:
  + data.table, version 1.17.2
  + stringdist, version 0.9.15
  + RecordLinkage, version 0.4-12.4, or any version that contains the RLdata10000 dataset.
  + e1071, version 1.7-16, for SVM models.
  + randomForest, version 4.7-1.2
    - [TODO] Finalize package list at some point

A proper renv lockfile can be provided to interested users on request. [TODO can we please just put this on GitHub?]

### Implementation

#### Key Steps

Our reference pipeline contains the following steps:

1. Setup and Load Data
2. Blocking
3. Feature engineering
4. Split data into training and testing datasets
5. Model fit on training set
6. Model evaluation on testing set

One notable difference from a real-world, production pipeline is that we have the answer beforehand. That is, we know which records should be linked and which ones should not. This is not a luxury available to any non-trivial linkage project. A real-world project might look more like this:

1. Setup and Load Data
2. Blocking
3. Feature engineering
4. Generate model predictions
   * If this is the first iteration, unsupervised methods or transfer learning will be required to generate the first set of links. This will likely require heuristics or a great deal of human review. The result of this work can then be used to train a custom supervised model.
5. If necessary, validate the model predictions to remove bad links, or catch links the model missed.
6. Assign a common person identifier to any successful links.
7. Retrain the linkage model as more links are made or the model stops improving.

#### Setup and Load Data

First, load all required packages and the target dataset.

# Load packages  
library(data.table)  
library(RecordLinkage)  
library(e1071)  
library(randomForest)  
library(stringdist)  
  
# If first run, need to install packages  
# install.packages(c('RecordLinkage', 'data.table', 'stringdist', 'e1071', 'randomForest'))  
  
# Load data, convert to data.table  
data(RLdata10000)  
recs <- data.table(RLdata10000)

The data.table package offers a variety of performance improvements over the base data.frame R class. For more information, you might check out response to "why didn't you just enhance data.frame in R?" in the [FAQ](https://cran.r-project.org/web/packages/data.table/vignettes/datatable-faq.html). Key data.table functions, such as joins and if-else statements, are parallelized under the hood (see [package documentation](https://rdatatable.gitlab.io/data.table/reference/openmp-utils.html) for details). As of writing, the default data.table behavior is to use half of the logical cores available for parallelized tasks. It is a reasonable default: this should roughly match the total physical cores on most machines. Consider toggling the number of threads using the setDTthreads function if necessary.

Next, we perform some light data manipulation to facilitate blocking. First, we convert birth day/month/year to string variables. Next, we assign a row ID to uniquely identify each record. Finally, we assign a "person ID", which uniquely identifies persons, some of whom have 2+ records in the full dataset. The person ID is the desired target of our model.

# Universal type conversions, missingness handling, and assigning persistent IDs  
recs[, `:=` (  
 # Convert to string  
 by = paste0(by),  
 bd = paste0(bd),  
 bm = paste0(bm),  
  
 # Record and person identifiers  
 row\_id = .I,  
 person\_id = identity.RLdata10000,  
  
 # Replace NA with blank strings  
 fname\_c1 = fcoalesce(fname\_c1, ''),  
 fname\_c2 = fcoalesce(fname\_c2, ''),  
 lname\_c1 = fcoalesce(lname\_c1, ''),  
 lname\_c2 = fcoalesce(lname\_c2, '')  
)]

Next, let's create a dataset of true links to evaluate the performance of the rest of the pipeline. Note that following a join, the default data.table behavior is to add an i. prefix to any columns on the right hand side if it shares a name with the left hand side.

# Row pairs can come in a jumbled order  
# This function subsets/deduplicates the dataset so only unique pairs remain  
# The resulting df will include only row ID's, where the lowest ID in each  
# pair is listed first.  
row\_id\_dedup <- function(row\_id, i.row\_id) {  
 df <- data.table('row\_id' = row\_id, 'i.row\_id' = i.row\_id)  
 df[, `:=` (  
 row\_id = pmin(row\_id, i.row\_id),  
 i.row\_id = pmax(row\_id, i.row\_id)  
 )]  
 df <- unique(df)  
 return(df)  
}  
  
# This is the source of truth: will come in useful in evaluation  
truth <- recs[recs, on='person\_id', nomatch=0][row\_id != i.row\_id]  
truth <- row\_id\_dedup(truth$row\_id, truth$i.row\_id)  
truth <- truth[recs, on='row\_id', nomatch=0]  
truth <- truth[recs, on=c('i.row\_id' = 'row\_id', 'person\_id'), nomatch=0]

#### Blocking

Next we perform blocking. Without blocking, we would have to compare all records to one another. In our example dataset of 10,000 records, 49,995,000 unique comparisons are possible. With 1,000 true links, this means there are 49,994 bad candidates for each true link. For every true link there are enough bad candidates to sell out the next Seattle Mariners game, with 2,065 people left over. This is clearly a waste of computing resources. Blocking uses relatively cheap comparisons early in the linkage process to filter out comparisons that are unlikely to be a true match. CHS pipelines tend to use traditional blocking based on deterministic rules. For our reference implementation, we chose the following rules for blocking:

1. Exact match on year of birth, first name differs by no more than two characters, last name differs by no more than two characters.
2. Exact match on first name, last name differs by no more than two characters
3. Exact match on last name, first name differs by no more than two characters

Any pairs not meeting one or more of these criteria will be excluded from further analysis.

Blocking rules should be a key focal point of your design. Consider all available fields as well as the completeness/quality of those fields when determining blocking criteria. In our example, we have few fields and high data quality. If we had access to gender or social security numbers, those would be great things to block on. The hallmark of a good blocking algorithm is excluding as many true negatives as possible from further testing while keeping almost all of the true positives.

Here is the blocking code in the reference implementation:

# Exact on birth year  
join\_cols = c('by')  
dob = recs[recs, on=join\_cols, allow.cartesian=TRUE][row\_id != i.row\_id]  
dob[, `:=` (  
 fname\_diff = stringdist(fname\_c1, i.fname\_c1, method='lv'),  
 lname\_diff = stringdist(fname\_c1, i.fname\_c1, method='lv')  
)]  
dob <- dob[fname\_diff <= 2 | lname\_diff <= 2]  
  
# Exact on first name  
join\_cols = c('fname\_c1', 'fname\_c2')  
first = recs[recs, on=join\_cols, allow.cartesian=TRUE][row\_id != i.row\_id]  
first <- first[stringdist(lname\_c1, i.lname\_c1, method='lv') <= 2]  
  
# Exact on last name  
join\_cols = c('lname\_c1', 'lname\_c2')  
last = recs[recs, on=join\_cols, allow.cartesian=TRUE][row\_id != i.row\_id]  
last <- last[stringdist(fname\_c1, i.fname\_c1, method='lv') <= 2]  
  
# Join all candidates to a single list and deduplicate  
keep\_cols <- c('row\_id', 'i.row\_id')  
candidates <- rbindlist(list(  
 dob[, ..keep\_cols], last[, ..keep\_cols], first[, ..keep\_cols]  
))  
candidates <- row\_id\_dedup(candidates$row\_id, candidates$i.row\_id)  
  
# Add the other columns for feature engineering  
candidates <- candidates[recs, on='row\_id', nomatch=0]  
candidates <- candidates[recs, on=c('i.row\_id' = 'row\_id'), nomatch=0]  
  
# We have labeled data- use it to apply a yes/no indicator.  
# This is our model target  
candidates[, is\_match := as.factor(person\_id == i.person\_id)]

Since we have a source of truth, let's see how the blocking algorithm did.

print\_blocking\_performance <- function(df, candidates, truth) {  
 # Identify cases where truth and candidates agree  
 agree <- candidates[truth, on = c('row\_id', 'i.row\_id'), nomatch=0]  
 pct\_agree <- formatC(100 \* nrow(agree) / nrow(truth), format="f", digits = 2)  
 msg <- paste0(  
 "Number of true matches found: ",  
 nrow(agree),  
 " (", pct\_agree, "%)\n"  
 )  
 cat(msg)  
  
 # How much have we reduced the search space?  
 num\_possible\_pairs <- choose(nrow(df), 2)  
 reduction\_ratio <- 100 \* (  
 num\_possible\_pairs - nrow(candidates)  
 ) / num\_possible\_pairs  
 reduction\_ratio <- formatC(reduction\_ratio, format="f", digits = 2)  
 msg <- paste0("Reduction ratio: (", reduction\_ratio, "%)\n")  
 cat(msg)  
}  
  
print\_blocking\_performance(recs, candidates, truth)

Number of true matches found: 997 (99.70%)  
Reduction ratio: (99.93%)

Blocking reduced the number of records to compare by 99.93%, but 0.3% of true matches were also filtered out. Now, we only have around 34.1 candidates for every true match. There is certainly room for improvement: we leave that exercise to the reader.

#### Feature Engineering

Next up, feature engineering. Any model that compares words must numerically represent the difference between the two items being compared. In traditional models, we use string distance metrics. Here, we stick to a limited set of string comparisons. In a real pipeline, this is another place where customizations are encouraged.

column\_pairs <- list(  
 # LHS first name, first part  
 c("fname\_c1", "i.fname\_c1"),  
 c("fname\_c1", "i.fname\_c2"),  
 c("fname\_c1", "i.lname\_c1"),  
 c("fname\_c1", "i.lname\_c2"),  
  
 # LHS first name, second part  
 c("fname\_c2", "i.fname\_c1"),  
 c("fname\_c2", "i.fname\_c2"),  
 c("fname\_c2", "i.lname\_c1"),  
 c("fname\_c2", "i.lname\_c2"),  
  
 # LHS last name, first part  
 c("lname\_c1", "i.fname\_c1"),  
 c("lname\_c1", "i.fname\_c2"),  
 c("lname\_c1", "i.lname\_c1"),  
 c("lname\_c1", "i.lname\_c2"),  
  
 # LHS last name, second part  
 c("lname\_c2", "i.fname\_c1"),  
 c("lname\_c2", "i.fname\_c2"),  
 c("lname\_c2", "i.lname\_c1"),  
 c("lname\_c2", "i.lname\_c2"),  
  
 # Birth date (3 parts)  
 c("by", "i.by"),  
 c("bm", "i.bm"),  
 c("bd", "i.bd")  
)  
  
# Apply Jaro-Winkler similarity to each pair  
jw\_names <- paste0(  
 "JW\_", sapply(column\_pairs, function(x) paste0(x[1], "\_", x[2]))  
)  
candidates[, (jw\_names) := lapply(  
 column\_pairs,  
 function(cols) stringdist::stringdist(get(cols[1]), get(cols[2]), method='jw')  
)]

Here are a couple of potential features we might consider were we to improve upon this:

* Add another type of string distance, such as cosine edit distances
* Add indicators for when a string was empty or had very few characters
* Add handling for string distances when one string is very short
* Take the minimum of multiple string distances. This is useful if, say, it is common for a last name to wind up in the first name field.

Just remember: parsimony is a virtue. It makes your pipeline more stable and reduces compute time. Start with a relatively simple base, and use your reference dataset to prove that adding complexity improves the overall model fit.

#### Split into Training and Testing Datasets

Before training a model, we hold out 20% of all candidates as a test set. Models perform best on the data used to train them. The 20% of records held out will test how the model performs on data not used in training. This is a better approximation of how our model would behave in production.

# Set seed for reproducibility  
set.seed(123)  
  
# Controls percentage of records in training set vs. testing  
train\_ratio <- 0.80  
  
# Create training and testing indices  
train\_indices <- sample(  
 seq\_len(nrow(candidates)), size = train\_ratio \* nrow(candidates)  
)  
train <- candidates[train\_indices]  
test <- candidates[-train\_indices]  
  
# Extract model features and outputs from each set  
X\_train <- train[, ..jw\_names]  
Y\_train <- train$is\_match  
X\_test <- test[, ..jw\_names]  
Y\_test <- test$is\_match

#### Model Fit

We will use a ensemble modeling approach in this implementation. The structure is as follows:

* Component models (feeds into meta model)
* - A random forest model  
   - An support vector machine (SVM)
* Meta model: logistic regression

The component models each take all the string distances we calculated in the 'feature engineering' section. The meta model takes the probabilities from the component models and uses them to produce a final determination. Especially for larger pipelines, you will not necessarily want to train a new model at each run of your pipeline. Thus, we save our model fit to file and load it for subsequent runs. If we ever renamed or deleted the model weights we saved, that would trigger a new round of model training.

# SVM model - train if needed, otherwise load pre-existing  
svm\_mod\_fn <- file.path('./models/svm\_mod.RDS')  
if(!file.exists(svm\_mod\_fn)) {  
 svm\_mod <- svm(y = Y\_train, x = X\_train, probability=TRUE)  
 saveRDS(svm\_mod, svm\_mod\_fn)  
} else {  
 svm\_mod <- readRDS(svm\_mod\_fn)  
}  
  
# RF model - train if needed, load if not  
rf\_mod\_fn <- file.path('./models/rf\_mod.RDS')  
if(!file.exists(rf\_mod\_fn)) {  
 rf\_mod <- randomForest(y = Y\_train, x = X\_train)  
 saveRDS(rf\_mod, rf\_mod\_fn)  
} else {  
 rf\_mod <- readRDS(rf\_mod\_fn)  
}  
  
# Meta model - train or load  
meta\_mod\_fn <- file.path('./models/meta\_model.RDS')  
if(!file.exists(meta\_mod\_fn)) {  
 # Get training set predictions to fit the meta model  
 svm\_train\_preds <- predict(svm\_mod, X\_train, probability=TRUE)  
 svm\_train\_probs <- attr(svm\_train\_preds, "probabilities")  
 rf\_train\_preds <- predict(rf\_mod, X\_train, type="prob")  
  
 # Input data for meta model  
 metadata <- data.table(  
 svm\_t = svm\_train\_probs[, 2],  
 rf\_t = rf\_train\_preds[, 2],  
 is\_match = Y\_train  
 )  
  
 # Fit model and save  
 meta\_model <- glm(is\_match ~ ., data = metadata, family = "binomial")  
 saveRDS(meta\_model, meta\_mod\_fn)  
} else {  
 meta\_model <- readRDS(meta\_mod\_fn)  
}

#### Model Evaluation

Now we evaluate the model against the testing set.

# Custom function to run the whole inference pipeline  
inference <- function(X, svm\_mod, rf\_mod, meta\_mod) {  
 svm\_train\_preds <- predict(svm\_mod, X, probability=TRUE)  
 svm\_train\_probs <- attr(svm\_train\_preds, "probabilities")  
 rf\_train\_preds <- predict(rf\_mod, X, type="prob")  
 metadata <- data.table(  
 svm\_t = svm\_train\_probs[, 2],  
 rf\_t = rf\_train\_preds[, 2]  
 )  
 return(predict(meta\_model, metadata, type="response"))  
}  
  
# Gets confusion matrix and evaluation stats  
compute\_metrics <- function(predicted, actual, positive\_class) {  
 # Convert predictions and actual labels to factors for consistency  
 predicted <- factor(predicted, levels = levels(actual))  
 actual <- factor(actual, levels = levels(actual))  
  
 # Confusion matrix  
 confmat <- table(Predicted = predicted, Actual = actual)  
  
 # Extract true/false positives and negatives  
 TP <- confmat[positive\_class, positive\_class]  
 FP <- sum(confmat[positive\_class, ]) - TP  
 FN <- sum(confmat[, positive\_class]) - TP  
 TN <- sum(confmat) - TP - FP - FN  
  
 # Output metrics  
 PPV <- TP / (TP + FP)  
 NPV <- TN / (TN + FN)  
 Sensitivity <- TP / (TP + FN)   
 Specificity <- TN / (TN + FP)  
 F1 <- 2 \* TP / (2 \* TP + FP + FN)  
  
 # Return results as a list  
 return(list(  
 Confusion = confmat,  
 PPV = PPV,  
 Sensitivity = Sensitivity,  
 Specificity = Specificity,  
 F1 = F1  
 ))  
}  
  
# Get predictions on test set and evaluate  
meta\_preds <- inference(X\_test, svm\_mod, rf\_mod, meta\_model)  
meta\_metrics <- compute\_metrics(as.factor(meta\_preds > 0.5), Y\_test, "TRUE")  
print(meta\_metrics)

$Confusion  
 Actual  
Predicted FALSE TRUE  
 FALSE 6630 5  
 TRUE 7 180  
  
$PPV  
[1] 0.9625668  
  
$Sensitivity  
[1] 0.972973  
  
$Specificity  
[1] 0.9989453  
  
$F1  
[1] 0.9677419

This pipeline captures 97.2% of true links. After considering the 0.3% of true links we lost in blocking, our overall process should successfully identify 96.9% of true links. This is a decent starting point, but could be improved further. One of the motivating principles for the LIDA team's work is that the 2-3% of links that standard tools systematically miss come from underserved groups. For a more complete accounting of the techniques LIDA uses to do this, please see their explainer of the ECHIDNA project.

## Performance Considerations

Next, we discuss performance considerations. The first thing to understand is that languages like R and Python are slow. This is not an insult: the designers have deliberately made choices that increase ease of coding at the cost of slower runtime performance. It is often a good tradeoff to make. However, understanding the limitations of a programming language is necessary to mitigate those weaknesses.

The second rule of performance is to do less stuff. This is the best type of performance win, because it typically increases the readability of your code, as well. Of course, this largely depends on the code you are optimizing. After a point, further code optimization will make your scripts more complex, not less.

Before proceeding, a reminder: this paper is designed for those with a public health or social science background with intermediate R experience. Computer science has terminology and theory that are beyond the expertise of this article's authors. Curious readers should read about time complexity in computer science on their own. A good place to start is [big O notation](https://web.mit.edu/16.070/www/lecture/big_o.pdf).

With that, let's talk about a few performance wins we have found in our own pipelines. Some of the examples below require the creation of new variables in our candidates dataset from above:

candidates[, `:=` (  
 DOB = as.Date(paste(by, bm, bd, sep="-"), format="%Y-%m-%d"),  
 i.DOB = as.Date(paste(i.by, i.bm, i.bd, sep="-"), format="%Y-%m-%d")  
)]

We will also create a "big" version of the dataset to see how our code snippets might perform on a larger set of data:

big\_candidates <- rbindlist(rep(list(candidates), 10))

### Avoid Rowwise Operations for Vectorized Functions

Many performance-optimized R functions are vectorized, meaning they take a vector as arguments. This is most effective when the underlying function is written in a faster language such as C. This way, C can do as much work as possible before incurring the overhead of sending the object back to your R session. This is why R users are always told that for loops are slow. A corollary to this is that rowwise operations are slow. Take this example of a code snippet adapted from the ECHIDNA demo:

candidates[, 'DOB\_HAM' := stringdist(DOB, i.DOB, method="hamming"), by=.I]

The by=.I turns this into a rowwise operation. The ECHIDNA demo includes the following note above this block of code (edited for clarity):

"the ‘[by=.I]’ grouping variable...tells DT to work in a row-wise fashion. In simple calculations...it is unnecessary but doesn’t slow down DT at all."

This is not entirely correct. A fully vectorized version of this function produces an identical result in a fraction of the time:

timing <- microbenchmark(  
 bygroup = candidates[, 'DOB\_HAM' := stringdist(DOB, i.DOB, method="hamming"), by=.I],  
 noby = candidates[, 'DOB\_HAM2' := stringdist(DOB, i.DOB, method="hamming")],  
 times = 5  
 )  
print(timing)

Unit: milliseconds  
 expr min lq mean median uq max neval  
 bygroup 1782.3194 1797.5544 1816.89428 1806.0168 1806.6876 1891.8932 5  
 noby 45.2718 46.0695 47.76328 46.4201 49.9911 51.0639 5

print(identical(candidates$DOB\_HAM, candidates$DOB\_HAM2))

[1] TRUE

The intended message of the note in the ECHIDNA tutorial was that the performance difference was ignorably small. At large enough scale, the difference becomes unignorable, however:

timing <- microbenchmark(  
 bygroup = big\_candidates[, 'DOB\_HAM' := stringdist(DOB, i.DOB, method="hamming"), by=.I],  
 noby = big\_candidates[, 'DOB\_HAM2' := stringdist(DOB, i.DOB, method="hamming")],  
 times = 5  
 )  
print(timing)

Unit: milliseconds  
 expr min lq mean median uq max neval  
 bygroup 18265.4656 18447.5255 18638.6676 18629.585 18825.2686 19020.9519 3  
 noby 445.3976 446.5203 451.6126 447.643 454.7201 461.7971 3

With 10x the rows to process, each method took 10x longer to run. Assuming the data continue to scale linearly, at some point the fully vectorized version of the code will run in one minute versus 35-40 minutes in the rowwise version. There are two reasons why the rowwise function performs so slowly. First, rather than calling a C function once with many input pairs, rowwise operations force a C call for each row in the dataset. The overhead of converting from C to R is incurred many, many times. Second, stringdist is both vectorized and multithreaded. If given two vectors, it will spread the work across multiple processors to speed things up. But the rowwise operation forces stringdist to consider each row individually.

### Use the Right Function for the Job

Later on in the ECHIDNA demo, the max and min are combined with a rowwise operation to get the row-wise maximum/minimum values. But there is built-in R function for this, [pmax](https://www.rdocumentation.org/packages/parttime/versions/0.1.2/topics/pmax):

print(microbenchmark(  
 rowwise = candidates[, MAX\_DOB := max(DOB, i.DOB), by=.I],  
 vectorized = candidates[, MAX\_DOB2 := pmax(DOB, i.DOB)]  
 ))

Unit: milliseconds  
 expr min lq mean median uq max neval  
 rowwise 138.1928 138.6273 139.83654 138.6489 140.8553 142.8584 5  
 vectorized 1.3857 1.6274 1.71172 1.6818 1.9126 1.9511 5

print(identical(candidates$MAX\_DOB, candidates$MAX\_DOB2))

[1] TRUE

If you find these sorts of mistakes in your own R code, I encourage you to spend a few days reading the documentation for the packages you use most frequently. You will be surprised how much you will learn!

### Use Functions as Intended

Another issue we found in our pipelines was in the way that branching logic was combined with string distance calculations. As mentioned earlier, we often want to modify q-gram based metrics when one string has fewer characters than q. However, the way this was implemented was buggy and a performance killer. Here is an adapted example from the ECHIDNA paper:

candidates[, 'FIRSTNAME\_COS' := stringdist(  
 fname\_c1,  
 i.fname\_c1,  
 method = c('cosine'),  
 q = ifelse(  
 nchar(fname\_c1) < 3 | nchar(i.fname\_c1) < 3,   
 min(nchar(fname\_c1), nchar(i.fname\_c1)),  
 3  
 )  
), by=.I]

There are three problems with this code. First, we are doing an avoidable rowwise operation. Second, data.table has an [fifelse function](https://www.rdocumentation.org/packages/data.table/versions/1.16.4/topics/fifelse) that is multithreaded and much faster than the base R implementation (read the docs!). Third, q does not take vectorized arguments. The third problem does not affect performance, but it does create an incorrect result. Let's show why with a simpler example:

a <- c('apple', 'banana', 'cavendish')  
b <- c('aple', 'bananas', 'Kaepernick')  
  
# Note both objects have q = 1 for the first-listed q argument  
out1 <- stringdist::stringdist(a, b, q=1, method='cosine')  
out2 <- stringdist::stringdist(a, b, q=c(1, 1056, 10e7), method='cosine')  
  
print(out1 == out2)  
# TRUE TRUE TRUE

When multiple arguments are passed to q, it ignores all but the first argument. Running this code within a data table seems to have caused a lot of unexpected behavior that I cannot fully explain. The end result is that 158 string distance calculations were incorrect in this example. Let's compare this to a new method that produces the correct result plus the added benefit of being over 100x faster:

# Cosine edit distance with flexible handling of small strings  
cosine\_qflex <- function(left, right, max\_q=3) {  
 # Get minimum length between each pair  
 min\_length <- pmin(nchar(left), nchar(right))  
  
 # Start output vector  
 out <- vector("numeric", length = length(left))  
  
 # Populate each case  
 q\_seq <- seq(max\_q)  
 for(q in seq\_along(q\_seq)) {  
 # For all integers up to the last, we only check for equality.  
 # In the last iteration, we include everything else  
 if (q == length(q\_seq)) {  
 idx <- which(min\_length >= q)  
 } else {  
 idx <- which(min\_length == q)  
 }  
  
 # Run the string distance function  
 out[idx] <- stringdist::stringdist(  
 left[idx], right[idx], method = 'cosine', q = q  
 )  
 }  
 return(out)  
}  
  
candidates[, FIRSTNAME\_COS\_C2\_ALT := cosine\_qflex(fname\_c2, i.fname\_c2)]

### Avoid multithreaded foreach on Windows

Many of our pipelines were using the foreach and doParallel packages in R. After setting up parallelization, we split data into equal chunks, one for each process:

library(snow)  
library(foreach)  
library(doParallel)  
library(parallel)  
  
# Set up parallel processes  
no\_cores <- 6  
cl <- makeCluster(no\_cores, type="PSOCK")  
registerDoParallel(cl)  
  
# Make a sufficiently large dataset for demonstration  
example <- rbindlist(rep(list(candidates), 20))  
  
# Split data into one group for each process  
example[, grp := cut(seq\_len(nrow(example)), breaks=no\_cores, labels=FALSE)]  
example <- split(example, by='grp')

Then, each chunk is processed on a worker thread using foreach and doParallel. The code snippet is wrapped in system.time() to track total time. Note that microbenchmark is not used as it can be misleading for user-defined parallelism.

foreach\_time <- system.time({  
 # For-each, works best forking, which is not available on Windows  
 foreach\_result <- foreach(  
 i = 1:no\_cores,  
 .packages = c('data.table', 'stringdist'),  
 .export="example",  
 .combine=c  
 ) %dopar% {  
 x <- example[[i]]  
 output <- stringdist::stringdist(x$by, x$i.by, method='jw', nthread=1)  
 return(output)  
 }  
})  
print(foreach\_time)

user system elapsed  
 2.12 6.04 9.70

How about we compare this to something using parLapply?

parlapply\_time <- system.time({  
 # Performs better on Windows  
 parlap\_result <- do.call(c, parLapply(  
 cl, example,   
 function(x) stringdist::stringdist(x$by, x$i.by, method='jw', nthread=1)  
 ))  
})  
print(parlapply\_time)

user system elapsed  
 0.36 1.11 1.70

In this case, parLapply is about 5.7x faster. This is because Windows does not allow process forking, a concept from Unix-based systems such as MacOS and Linux. Instead, parallelism must be implemented through sockets, which requires each worker to have its own memory space. The foreach and doParallel packages are designed for forking. In socket clusters, they copy memory over to each worker. As implemented, much of the memory copied over is unnecessary. Using parLapply addresses this by sending each worker process only the data it is responsible for. Each process is depicted graphically below. The timing outputs show that the foreach loop spent nearly 5 seconds more time in system calls than parLapply. Note that memory allocation is a system call. Curious readers can learn more about the different kinds of parallelism in R [here](https://stat.ethz.ch/R-manual/R-devel/library/parallel/doc/parallel.pdf) and [here](https://cran.r-project.org/web//packages//abn/vignettes/multiprocessing.html).

## Conclusion

In this paper, we have implemented a fully reproducible record deduplication pipeline and provided some general tips for improving performance. This effort was informed by our own efforts to standardize and optimize our own existing processes. We hope the reader finds this useful in their own processes. Future papers will build off of this work to implement more advanced functionality, such as writing custom Rust code to improve the memory efficiency of fuzzy joins.