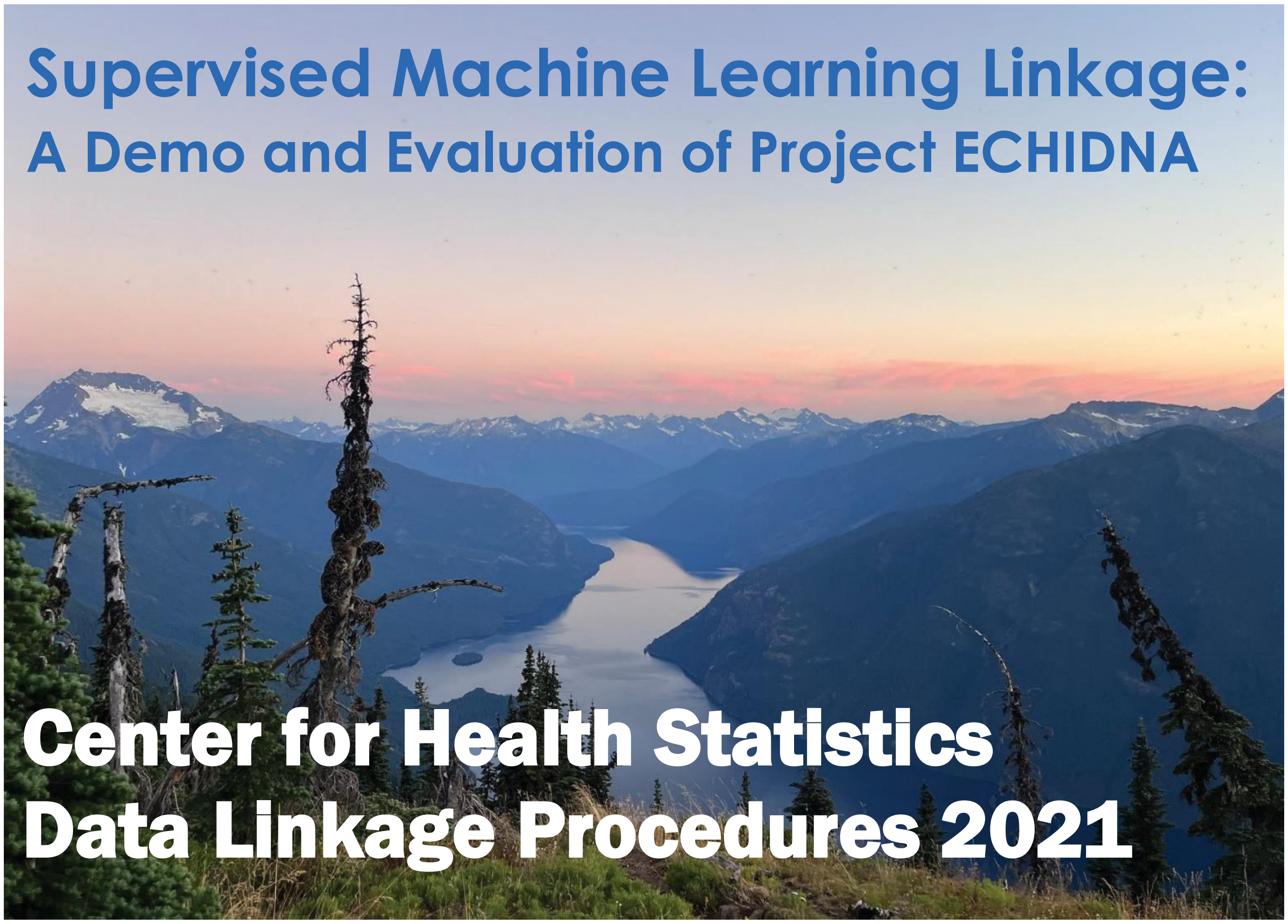
WASHINGTON STATE DEPARTMENT OF HEALTH

Supervised Machine Learning Linkage: A Demo and Evaluation of Project ECHIDNA



Center for Health Statistics
Sean Coffinger (Author)

Office of Public Affairs & Equity
Nikki Lanka (Contributor)

Washington State Department of HEALTH

DOH 346-147 January 2024

Table of contents

ACKNOWLEDGEMENTS: This work was possible due to the contribution and review of multiple people on the Washington State Department of Health (DOH) Center for Health Statistics (CHS) staff and leadership team. Maya Bhat and Julian Kapoor provided review and support for early iterations of this strategy. They currently lead the linkage team at CHS and continue to push the boundaries of public health entity resolution. This process is an evolution, and they have taken it to the next level. Heather Cohen provided review and crucial feedback. Seth Rothbard and Andrew Sikes provided additional reviews and guidance. Katie Hutchinson and Kristin Reichl have provided consistent leadership support for improving data linkage practices at DOH.

Executive summary

The Center for Health Statistics (CHS) has produced many customized data-linkage products and provided them to various stakeholders since 2021. This report provides a detailed example of one such linkage project. Our goal is to help data linkage scientists of all levels gain greater familiarity with the supervised machine learning entity resolution techniques used at CHS. Methods utilized by CHS are constantly evolving, and the recommendations presented in this report reflect practices implemented in 2022. While some practices described are now antiquated, they provide context about the state of the art and its evolution. A separate comprehensive report will update these methodologies for 2023 and beyond.

This report provides a tutorial using the project ECHIDNA (Equating CHARS Identities to Death-records through Networked Attributes), which links hospital discharge records in the Comprehensive Hospital Abstract Reporting System (CHARS) and death records in the Washington Health and Live Events System (WHALES).

This demonstration is written for scientists familiar with basic data linkage and entity resolution concepts who are curious about the machine learning methodologies that we implemented. This tutorial is in R, but it was coded exhaustively with minimal loops and functions to prioritize stepwise understanding rather than copy and paste coding. These methods can easily be implemented in Python, Julia, or other languages.

This report is written more casually than a piece of academic writing. We find demos and tutorials reach our audience most successfully when first person perspective, slang, short acronyms, and R specific 'verbs' are permissible. Though we define many terms throughout the report, we welcome any readers to reach out to the author directly for clarification. Additionally, for readers who are completely new to data-linking, our recorded presentation, found on the Analytical Methods and Reports webpage, reviews the fundamentals of the practice.

Note: Notes are provided throughout the document in these grey boxes.
Note: R package names, 'verbs' and function parameters are indicated by italics (e.g. *Tidyverse, ifelse()* and *toupper*) and variable names are given double quotation marks (e.g. "FIRST_NAME")

Data sources and descriptions

Project ECHIDNA (**E**quating **CH**ARS **I**dentities to **D**eath-records through **N**etworked **A**ttributes) links hospital discharge records with death certificates.

WHALES (Washington Health and Life Events System)
The death certificate data is queried from the death table.

- The purpose of the Washington Health and Life Events System (WHALES) system is to support the registration of Washington vital events for the Washington Department of Health and other users, such as funeral directors, attending physicians, medical examiners, and birthing facilities.
- The data are queried at the individual birth certificate level.
- Washington born individuals who receive a birth certificate are included in the dataset.

CHARS (The Comprehensive Hospital Abstract Reporting System)
Hospital discharge data is derived from the discharge record table.

- The Comprehensive Hospital Abstract Reporting System (CHARS) is a Department of Health system that collects record level information on inpatient and observational patient community hospital stays.
- The data are queried at the discharge event level.

ECHIDNA machine learning linkage demo

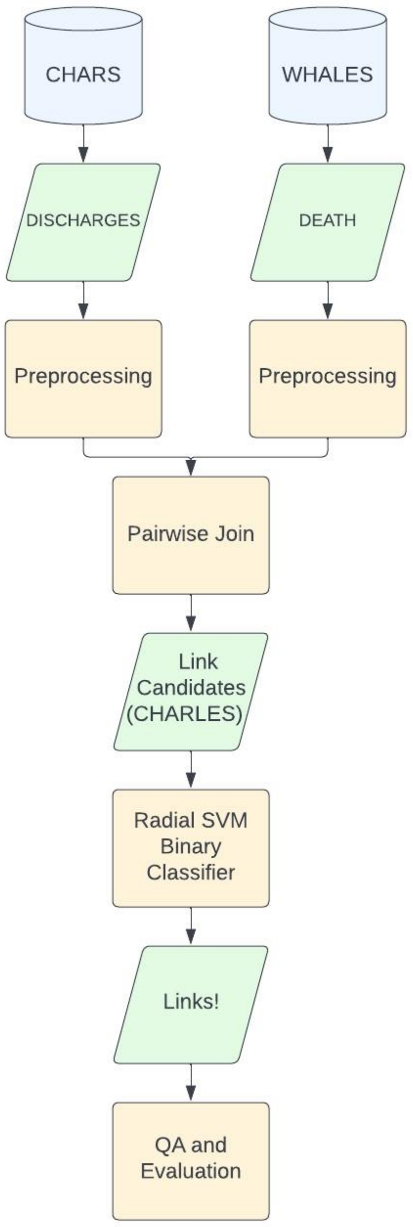
The nGram + SVM (support vector machines) supervised machine learning methodology used to link two data sources consistently captures more links—with fewer errors—than probabilistic, fuzzy and deterministic models. It is well suited for large-scale linkage projects with lots of records. It handles erroneous and missing data well and requires few comparison variables to derive a viable list of links. Machine learning strategies excel in their ability to learn from iterative training and thrive in automated setups. Speed, accuracy, consistency, and the removal of biases created by inadequate algorithms are benefits we often observe. In this report, I go step-by-step through a completed project (ECHIDNA) with code examples, notes and recommendations.

This document is intended for individuals who want to develop their own Machine Learning (ML) linkage project and have some familiarity with R. If you are a seasoned pro at probabilistic matching and are still 'on the fence' about machine learning methods, or you're curious about ML model performance, feel free to begin this document by reading the evaluation prior to the demonstration. If you are completely new to data-linking and are not entirely sure what a machine learning model is or does, please view my recorded presentation that introduces the concept on the Analytical Methods and Reports webpage.

This training document uses the ECHIDNA project as an example. ECHIDNA aimed to link WHALES death records with CHARS discharge data.

Note: there is more than one way to do this, so there is room to be creative! This is not an out-of-the-box solution. It will require tinkering for implementation.

Step 0 – What are we doing?



- The goal is to link data from two separate databases.
- We query the data that we want to link.
- We manipulate, clean and standardize the data that we want to link.
- We block and join these data to create an exhaustive pairwise comparison dataframe that includes all potential links.
- We calculate distance metrics and inputs for the Machine Learning classifier.
- We train and implement a SVM classifier.
- We derive all identified links.
- We perform quality assurance and evaluate the linkage performance.

Step 1 – Do some homework

Review these common terms and resources:
1. SVM
2. nGram
3. Type I Error
4. Cosine, Hamming and other Distances
5. False Discovery Rate
6. Good overview of Sensitivity and Specificity
7. Cohen's Kappa
8. Parallel Processing Basics

Step 2 – Familiarize yourself with your data

By familiarizing yourself with your day, you can identify which variables you will be comparing to determine if there is a match or not.

1. Identify variables common to both datasets

Note: In more advanced versions of the machine learning linkage methodology, this step is often more convoluted than "which identifiable fields are shared between my two, or more, datasets?". For example, geographic indicators can be transformed to produce proximity values, or combining multiple fields can produce logical values that you can compare. These advanced but very helpful values will not be utilized in this demo, but it should be noted that the search for "helpful" variables (e.g. flags like plural births) can be intensive and critical for performance.

2. Use summary statistics to investigate if there is enough populated data to include in the model. Sparse data is acceptable if the field is a strong identifier (think SSN). However, if both datasets have a weak identifier (e.g. ZIPCODE) and only 5 percent of the records have ZIPCODES, you may opt to omit this field from your model. The tradeoff here is the strength of the identifier versus how sparse the data is. The more variable you have the longer the model will take to run due to increased volume of comparisons, but it may be worth it for a sparsely-coded strong identifier.
3. For this training example, we have identified the following variables:
   1. First Name
   2. Last Name
   3. Middle Initial
   4. Sex
   5. SSN (Last 4 digits)
   6. Date of Birth

Step 3 – Hop into R and load your packages

You may or may not use the following packages in your project, but this is the standard set I used during this linkage project:

```r
### PROJECT ECHIDNA ###
## Equating CHARS Identities to Death-records through Networked Attributes ##

# Script: 8.1.9
# Created: 8/31/2021 SC
# Last Edit: 8/31/2021 SC

# Short Description: Dummy Script to show process

rm(list = ls())

# Load Packages
library(tidyverse)
library(DBI)
library(odbc)
library(janitor)
library(stringdist)
library(data.table)
library(umap)
library(e1071)
library(Rtsne)
library(stats)
library(mltools)
library(doParallel)
library(snow)
library(fuzzyjoin)
library(stringr)
library(pbapply)

# Prep the kitchen
gc()
mlimit = memory.limit() * 9
memory.limit(mlimit)

# Helpful Functions
'%ni%' = Negate('%in%')
```

Note: If you are primarily a *tidyverse* user, we will be heavily relying on *data.table* – so you may want to brush up on your *DT*. Here is a useful cheat sheet.

Step 4 – Query your data

First, we query from WHALES. Here, we deliberately start with an unfiltered and unmanipulated query because we will prep our data in R post-query. We make sure to capture any alternative fields like "Also Known As" names and ensure we capture any variants of the fields we will be comparing.

Note: It is important to note that a strength of the machine learning methodology is its ability to compare numerous versions of the same variables rather rapidly. For instance, **we want to derive every possible version or variation of a name (or other variable) for downstream distance calculations.** If the same individual had a death certificate name 'RICHARD JOHNSON' and a discharge name 'DICK JOHNSON' it would be valuable it the death certificate had 'DICK' as an alternative name. We also want to split names: If 'JAY-BOB SMITH' and 'ROBERT SMITH' are being compared, I want to compare 'JAY' to 'ROBERT', 'BOB' to 'ROBERT' and 'JAYBOB' to 'ROBERT'. The model can handle them all!

```r
# Connection to DQSS
con1 <- odbc::dbConnect(
  odbc(),
  Driver = "SQL Server",
  Server = "REDACTED",
  Database = "REDACTED",
  Trusted_Connection = "True"
)

# Death Table Query
waDeathQuery <- function() {
  deaths = paste(
    "SELECT ISNULL(DEATH_REC_ID, ' ') as 'DEATH_REC_ID',
                 ISNULL(SFN_NUM, ' ') as 'DEATH_SFN_NUM',
                 ISNULL(DOB, ' ') as 'DOB',
                 ISNULL(DOD, ' ') as 'DOD',
                 ISNULL(SSN, ' ') as 'SSN',
                 ISNULL(GNAME, ' ') as 'FIRST_NAME',
                 ISNULL(MNAME, ' ') as 'MIDDLE_NAME',
                 ISNULL(LNAME, ' ') as 'LAST_NAME',
                 ISNULL(AKA1_FNAME, ' ') as 'AKA1_FNAME',
                 ISNULL(AKA1_LNAME, ' ') as 'AKA1_LNAME',
                 ISNULL(AKA2_FNAME, ' ') as 'AKA2_FNAME',
                 ISNULL(AKA2_LNAME, ' ') as 'AKA2_LNAME',
                 ISNULL(SEX, ' ') as 'SEX',
                 BIRTH_MATCH_CODE,
                 ISNULL(BIRTH_INTERNAL_CASE_NUMBER, ' ') as 'BIRTH_SFN_NUM'
            FROM [REDACTED].[REDACTED]
            WHERE FL_CURRENT = '1'
                AND FL_VOIDED = '0'" ,
    sep = ''
  )

  death_tbl = dbGetQuery(con1, deaths)
  return(death_tbl)
}
waDeath = waDeathQuery()
```

CHARS was a little more straight forward because there were no multiple related fields:

```r
# Connection to CHARS
con2 <- odbc::dbConnect(
  odbc(),
  Driver = "SQL Server",
  Server = " REDACTED ",
  Database = " REDACTED ",
  Trusted_Connection = "True"
)

# CHARS DischargeRecord
CHARSdist <- function() {
  dist = paste(
    "SELECT ISNULL(DischargeRecordID, ' ') as 'DischargeRecordID',
                 ISNULL(PatientBirthDate, ' ') as 'DOB',
                 ISNULL(PatientLastFourSSN, ' ') as 'SSN',
                 ISNULL(PatientFirstName, ' ') as 'FIRST_NAME',
                 ISNULL(PatientMiddleName, ' ') as 'MIDDLE_NAME',
                 ISNULL(PatientLastName, ' ') as 'LAST_NAME',
                 ISNULL(PatientSexCode, ' ') as 'SEX',
                 ISNULL(AdmissionDate, ' ') as 'DOA'
            FROM [REDACTED].[ REDACTED]",
    sep = ''
  )

  death_tbl = dbGetQuery(con2, dist)
  return(death_tbl)
}
CHARS.1 = CHARSdist()
```

Step 5 – Standardize, standardize, standardize, clean and standardize

The best way to preprocess your data is extremely dependent on your data. Here we will look at two datasets that have different standardization requirements.

CHARS has fewer alternate ID variables, so we will start there.

The full code block is presented first, and the components of the block are described below:

```
# Preprocess and standardize
CHARS.2 = CHARS.1 %>%
  mutate(
    DOB = gsub("[[:punct:]]|[[:space:]]", "", DOB),
    DOB = substr(DOB, 1, 8),
    DOA = gsub("[[:punct:]]|[[:space:]]", "", DOA),
    DOA = substr(DOA, 1, 8),
    FIRST_NAME = gsub(
      "[[:punct:]]|[[:space:]]",
      "",
      iconv(FIRST_NAME, "latin1", "ASCII", sub = "")
    ),
    MIDDLE_NAME = gsub(
      "[[:punct:]]|[[:space:]]",
      "",
      iconv(MIDDLE_NAME, "latin1", "ASCII", sub = "")
    ),
    LAST_NAME = gsub(
      "[[:punct:]]|[[:space:]]",
      "",
      iconv(LAST_NAME, "latin1", "ASCII", sub = "")
    ),
    SSN = gsub(
      "[[:punct:]]|[[:space:]]",
      "",
      iconv(SSN, "latin1", "ASCII", sub = "")
    ),
    MIDDLE_INIT = substr(MIDDLE_NAME, 1, 1)
  ) %>%
  distinct() %>%
  dplyr::select(-MIDDLE_NAME) %>%
  mutate(SSN = ifelse(SSN == '0000', '', SSN)) %>%
  mutate_all(.funs = toupper)

CHARS.2[is.na(CHARS.2)] = " "
```

1. Remove punctuation and spaces

```
gsub("[[:punct:]]|[[:space:]]", "", ...)
```

2. Ensure only English characters without *Translit*

```
iconv(..., "latin1", "ASCII", sub = "")
```

3. Create middle initial field
4. Quick deduplication
5. Remove invalid 0000 SSNs and replace with a space
6. Ensure everything is capitalized

```
mutate_all(.funs = toupper)
```

7. Replace all NAs with a space. This helps with downstream distance calculations and ensures NAs are not accidentally translated into literally ‘NAS’. For example, if you aren’t careful, ‘JONAS’ could be compared to ‘NAS’ rather than a missing field.

WHALES is more convoluted because multiple name fields need to be split up:

```
waDeath.1 = waDeath %>%
  distinct() %>%
  mutate(LAST_NAME2 = ifelse(
    grepl('-', waDeath$LAST_NAME) |
      grepl(' ', waDeath$LAST_NAME) == TRUE,
    LAST_NAME,
    ' '
  )) %>%
  mutate(LAST_NAME2 = gsub('-', ' ', LAST_NAME2)) %>%
  mutate(LAST_NAME2 = gsub(' - ', ' ', LAST_NAME2)) %>%
  mutate(LAST_NAME2 = gsub('  ', ' ', LAST_NAME2)) %>%
  mutate(LAST_NAME2 = gsub('   ', ' ', LAST_NAME2)) %>%
  separate(LAST_NAME2, c('LAST_NAME2', 'LAST_NAME3'), ' ') %>%
  mutate(
    lubDOB = lubridate::mdy(DOB),
    lubDOD = lubridate::mdy(DOD),
    DOB = gsub("[[:punct:]]|[[:space:]]", "", lubDOB),
    DOD = gsub("[[:punct:]]|[[:space:]]", "", lubDOD),
    FIRST_NAME = gsub(
      "[[:punct:]]|[[:space:]]",
      "",
      iconv(FIRST_NAME, "latin1", "ASCII", sub = "")
    ),
    MIDDLE_NAME = gsub(
      "[[:punct:]]|[[:space:]]",
      "",
      iconv(MIDDLE_NAME, "latin1", "ASCII", sub = "")
    ),
    LAST_NAME = gsub(
      "[[:punct:]]|[[:space:]]",
      "",
      iconv(LAST_NAME, "latin1", "ASCII", sub = "")
    ),
    LAST_NAME2 = gsub(
      "[[:punct:]]|[[:space:]]",
      "",
      iconv(LAST_NAME2, "latin1", "ASCII", sub = "")
    ),
    LAST_NAME3 = gsub(
      "[[:punct:]]|[[:space:]]",
      "",
      iconv(LAST_NAME3, "latin1", "ASCII", sub = "")
    ),
    MIDDLE_INIT = substr(MIDDLE_NAME, 1, 1),
    AKA1_FNAME = gsub(
      "[[:punct:]]|[[:space:]]",
      "",
      iconv(AKA1_FNAME, "latin1", "ASCII", sub = "")
    ),
    AKA1_LNAME = gsub(
      "[[:punct:]]|[[:space:]]",
      "",
      iconv(AKA1_LNAME, "latin1", "ASCII", sub = "")
    ),
    AKA2_FNAME = gsub(
      "[[:punct:]]|[[:space:]]",
      "",
      iconv(AKA2_FNAME, "latin1", "ASCII", sub = "")
    ),
    AKA2_LNAME = gsub(
      "[[:punct:]]|[[:space:]]",
      "",
      iconv(AKA2_LNAME, "latin1", "ASCII", sub = "")
    )
  )
```

The first step in preprocessing a messier dataset is to do the exact same thing as we did in CHARS, in the exact same manner. However, we had to take some additional steps to get there:

1. utilize *lubridate* to standardize DOB and DOD

```
lubDOB = lubridate::mdy(DOB),
lubDOD = lubridate::mdy(DOD),
```

2. Splitting names with spaces or "-" into different name variables

```
mutate(LAST_NAME2 = ifelse(
  grepl('-', waDeath$LAST_NAME) |
    grepl(' ', waDeath$LAST_NAME) == TRUE,
  LAST_NAME,
  ' '
)) %>%
  mutate(LAST_NAME2 = gsub('-', ' ', LAST_NAME2)) %>%
  mutate(LAST_NAME2 = gsub(' - ', ' ', LAST_NAME2)) %>%
  mutate(LAST_NAME2 = gsub('  ', ' ', LAST_NAME2)) %>%
  mutate(LAST_NAME2 = gsub('   ', ' ', LAST_NAME2)) %>%
  separate(LAST_NAME2, c('LAST_NAME2', 'LAST_NAME3'), ' ')
```

Next, we can take advantage of previously linked data (in this case birth records to death records) to see if we can gather any additional information from the birth records that is missing in the death records (i.e. middle name, different last name, etc.)

Here we query and preprocess the birth table in the same way:

```
# Birth Table Query
waBirthQuery <- function() {
  births = paste(
    "SELECT BIRTH_REC_ID,
                 ISNULL(SFN_NUM, ' ') as 'BIRTH_SFN_NUM',
                 DEATH_SFN as 'DEATH_SFN_NUM',
                 ISNULL(IDOB, ' ') as 'BDOB',
                 ISNULL(CHILD_GNAME, ' ') as 'BIRTH_FIRST_NAME',
                 ISNULL(CHILD_MNAME, ' ') as 'BIRTH_MIDDLE_NAME',
                 ISNULL(CHILD_LNAME, ' ') as 'BIRTH_LAST_NAME',
                 ISNULL(INFANT_SEX, ' ') as 'BIRTH_SEX'
```

```
            FROM [REDACTED].[ REDACTED]
            WHERE FL_CURRENT = '1'
                AND FL_FILED <> 'N'
                AND FL_VOIDED = '0'" ,
    sep = ''
  )
  birth_tbl = dbGetQuery(con1, births)
  return(birth_tbl)
}
waBirth = waBirthQuery()

# Preprocess and standardize
waBirth.1 = waBirth %>%
  mutate(
    lubDOB = lubridate::mdy(BDOB),
    BDOB = gsub("[[:punct:]]|[[:space:]]", "", lubDOB),
    BIRTH_FIRST_NAME = gsub(
      "[[:punct:]]|[[:space:]]",
      "",
      iconv(BIRTH_FIRST_NAME, "latin1", "ASCII", sub = "")
    ),
    BIRTH_MIDDLE_NAME = gsub(
      "[[:punct:]]|[[:space:]]",
      "",
      iconv(BIRTH_MIDDLE_NAME, "latin1", "ASCII", sub = "")
    ),
    BIRTH_LAST_NAME = gsub(
      "[[:punct:]]|[[:space:]]",
      "",
      iconv(BIRTH_LAST_NAME, "latin1", "ASCII", sub = "")
    ),
    BIRTH_MIDDLE_INIT = substr(BIRTH_MIDDLE_NAME, 1, 1)
  ) %>%
  distinct()
```

Then, we join the birth table information into the WHALES death dataset to supplement any missing information or differing names:

Note: these steps can introduce very problematic errors if these secondary joins are not restricted to only very high-confidence links. If low quality links are allowed, it can lead to the propagation of error and a potential increase type I error (false detections).

```
# Join WHALES b/d into each other
waDeath.2 = waDeath.1 %>%
  mutate(DEATH_SFN_NUM = as.numeric(DEATH_SFN_NUM)) %>%
  left_join(waBirth.1, by = c('DEATH_SFN_NUM', 'BIRTH_SFN_NUM'))

# Clean this up and just do the last decade
WHALES.1 = waDeath.2 %>%
  filter(DOD > 20190000 & DOD < 20200000) %>%
  mutate(
    DOB = ifelse(is.na(DOB), BDOB, DOB),
    SSN = ifelse(SSN == '999-99-9999', ' ', SSN),
    SSN = substr(SSN, 8, 11),
    SEX = ifelse(SEX == ' ', BIRTH_SEX, SEX),
    MIDDLE_INIT = ifelse(MIDDLE_INIT == '', BIRTH_MIDDLE_INIT, MIDDLE_INIT)
  ) %>%
  dplyr::select(
    -BIRTH_REC_ID,
    -BDOB,
    -lubDOB.x,
    -lubDOD,
    -lubDOB.y,
    -BIRTH_MATCH_CODE,
    -BIRTH_SFN_NUM,-BIRTH_MIDDLE_INIT,
    -BIRTH_SEX,
    -BIRTH_MIDDLE_NAME,
    -MIDDLE_NAME,
    -DEATH_REC_ID
  )

WHALES.1[is.na(WHALES.1)] = " "
```

In the above code, we also focus on a single year of WHALES death records and run the full model for each year. Training does not need to be done for every year unless there are drastic changes in the input variables to your model change or their sparseness.

The last thing we do for the WHALES data is collapse all the unique names, for first and last name respectively, into more manageable fields. Specifically, rather than having a FIRST_NAME_1, AKA1_FNAME and BIRTH_FIRST_NAME in a record, we want to collapse this to FIRST_NAME_1, FIRST_NAME_2, FIRST_NAME_3, etc., and deduplicate the names.

First name example:

We want to go from this sparse and unduplicated dataframe…

ID	FIRST_NAME	AKA1_FNAME	AKA2_FNAME	BIRTH_FIRST_NAME	ALT_FNAME
54941	James			James	
58494	Ricardo	Ricky	Ricardo	Ricardo	Rick
98741	Steve			Steven	

To something like this…

ID	FIRST_NAME_1	FIRST_NAME_2	FIRST_NAME_3
54941	James		
58494	Ricardo	Ricky	Rick
98741	Steve	Steven	

```
no_cores =  (detectCores() / 2) - 1
cl = makeCluster(no_cores, type = "SOCK")
registerDoParallel(cl)
start.time = Sys.time()

c2 = foreach(i = 1:nrow(WHALES.1),
             .packages = c('tidyverse', 'data.table')) %dopar% {
               a = WHALES.1[i, ]
               b = a %>% melt(id.vars = c('DEATH_SFN_NUM', 'DOB', 'DOD', 'SSN', 'SEX', 'MIDDLE_INIT'))
%>% ungroup() %>%
                 mutate(variable = ifelse(
                   variable == 'AKA1_FNAME',
                   'FIRST_NAME',
                   ifelse(
                     variable == 'AKA1_LNAME',
                     'LAST_NAME',
                     ifelse(
                       variable == 'AKA2_FNAME',
                       'FIRST_NAME',
                       ifelse(
                         variable == 'AKA2_LNAME',
                         'LAST_NAME',
                         ifelse(
                           variable == 'LAST_NAME2',
                           'LAST_NAME',
                           ifelse(
                             variable == 'LAST_NAME3',
                             'LAST_NAME',
                             ifelse(
                               variable == 'BIRTH_FIRST_NAME',
                               'FIRST_NAME',
                               ifelse(
                                 variable == 'BIRTH_LAST_NAME',
                                 'LAST_NAME',
                                 ifelse(
                                   variable == 'FIRST_NAME',
                                   'FIRST_NAME',
                                   ifelse(variable == 'LAST_NAME', 'LAST_NAME', 'ERROR')
                                 )
                               )
                             )
                           )
                         )
                       )
                     )
                   )
                 )) %>%
                 distinct() %>%
                 filter(value != '') %>%
                 group_by(variable) %>%
                 mutate(n = row_number()) %>%
                 mutate(name = paste(variable, n, sep = '_')) %>%
                 ungroup() %>%
                 dplyr::select(-variable,-n) %>%
                 spread(name, value)
               return(b)
             }

end.time = Sys.time()
end.time - start.time
stopCluster(cl)

WHALES.2 = plyr::rbind.fill(c2)
```

This is an excellent way to introduce parallel processing. The code chunk above includes a structure that is not exactly computationally friendly. Obviously, it can be written and constructed better without a stack of 10 *ifelse* statements; however, I will use my inefficient coding here to discuss computational efficiency and the balance between time, RAM and what we are asking R to do for us.

The machine learning method requires R to essentially compare millions of rows individually through a very long algorithm that makes hundreds of calculations per comparison. If run sequentially, this could take days, weeks, or even months, which is unrealistic and unsustainable. This is where we rely on parallel processing.

Details on the parallel processing process in R

1. *no_cores* dictates how many 'workers' you want to enable in your parallel process. In R, each worker receives the data you explicitly state prior to *%dopar%.* This has a few implications:

    a. Each worker does not automatically receive a copy of your environment (think of each worker as a separate RStudio session)

    b. The more workers you utilize, the faster your process will go

    c. The more workers you utilize, the more RAM you are using. We are still using one computer, so if you are sending out massive files to each worker, you may run out of RAM and "kill" one of your workers.

    d. The balance between b and c is dependent on your computer and the data you are sending to each worker.

2. There are two ways to explicitly tell the process what data to send to each worker. Any data, function or object in your environment that is inside the *%dopar%* function must be sent or it will fail.

```
  i = 1:nrow(WHALES.1) # Including a variable in "i" sends that object

  # or

  .export = c("WHALES.1") # using .export also works
```

3. Keep these the same.

```
cl = makeCluster(no_cores, type = "SOCK")
registerDoParallel(cl)
```

4. Treat your *foreach* like a normal *for* function and specify which packages are required for your *%dopar%.* Just like data from the environment each worker needs to know what packages it needs to run the function:

```
foreach(
  i = 1:nrow(WHALES.1),
  .packages = c('tidyverse','data.table')
  #.export = c('df1','df2',...) if you have more data than WHALES.1
)
```

5. Always finish with *stopCluster(cl).*

Step 6 – 'CHARLES' file creation

Here we do something interesting: We *fuzzy_join* two very large datasets into a massive dataset full of pairwise comparisons based on fuzzy-DOB. This serves as the complete dataframe of possible links.

Why do we do this? It enables batch processing. If we don't create this large CHARLES file (CHARS + WHALES), during the parallel process we must send individual records to the workers, find a 'pool' of potential link candidates, then calculate and classify all those links in the pool. Instead, we can send batches of potential links to the workers and allow *data.table* to do its job and compute row-wise calculations quickly and efficiently. In this one-to-many (one death to many possible hospitalizations) linkage we treat the death event as the 'target' and the hospitalization data as the 'pool' of potential matches.

It also allows us to "trim the fat" off these comparisons. For example, we can require at least one field to match (inclusion can be as easy as a SEX match; however, if your data is huge, you can remove fields like SEX from the trim-fat filter). This fat trimming cuts down the total number of comparisons drastically and ultimately saves us time. If we were running sequentially (even in a parallel processing fashion), we could waste time by processing links that have nothing in common besides date of birth (DOB).

If your pairwise file is too large, you will often get a "large vectors are not supported" or an "allocated vector of size XX GB… etc.". If this is the case, split the target set (here WHALES because there is ONE whales death record -> multiple CHARS records). Here we split by year to make it easier. As a result, the CHARLES files were large but not unmanageable for a good computer.

```
# Fuzzy Join by DOB
start.time = Sys.time()
CHARLES = WHALES.3 %>% fuzzyjoin::stringdist_left_join(CHARS.2,
                                                       by = 'DOB',
                                                       method = 'hamming',
                                                       max_dist = 1)
end.time = Sys.time()
end.time - start.time

# Trim the fat
CHARLES2 = CHARLES %>%
  filter(DOD > DOA) %>%
  mutate(
    noSim = ifelse(
      DOB.x != DOB.y &
        SSN.x != SSN.y &
        SEX.x != SEX.y &
        MIDDLE_INIT.x != MIDDLE_INIT.y &
        FIRST_NAME != FIRST_NAME_1 &
        FIRST_NAME != FIRST_NAME_2 &
        FIRST_NAME != FIRST_NAME_3 &
        LAST_NAME != LAST_NAME_1 &
        LAST_NAME != LAST_NAME_2 &
        LAST_NAME != LAST_NAME_3 &
        LAST_NAME != LAST_NAME_4,
      1,
      0
    )
  ) %>%
  filter(noSim == 0) %>%
  dplyr::select(-noSim)
```

If you have to split your ‘target’ dataset, use the following (or equivalent code).

```
WHALES_split1 = split(WHALES.3, (seq(nrow(WHALES.3))-1) %/% 50000)

CHARLES_list = function(x){
nWHALES = WHALES_split[[x]]
CHAVID = cases.4 %>%
  fuzzyjoin::stringdist_left_join(CHARS.1, by = 'DOB',  method = 'hamming', max_dist = 1)
return(CHAVID)
}

CHARLES_listOut = pblapply(1:length(WHALES_split1), CHARLES_list)
```

Here we use *pblapply* to get a time estimate (highly recommended).

Note: this process may take an hour or two depending on your computer. Stick with it—this is the master file we will be working out of. Feel free to wrap this into a parallel processing strategy as indicated above.

Step 7 – Add name frequencies (optional, but encouraged)

Next, we sample from CHARLES (or CHARLES split). For this example, we will be using CHARLES.

Start by creating a dataframe from your target dataset, in this case WHALES, and find the frequency of each first and last name in your data. This can be as easy as grouping by each name and finding the proportion, or as convoluted as finding the highest proportion of “near-neighbor” names and using that. Sometimes we standardize these proportions between .01-.99 and subtract the proportions from 1 so it is in alignment with the *stringdist* calculations (0 being more prevalent, .99 being extremely rare). Whatever you decide to go with, join in those name frequency values as new columns for each target name field. You only need to do this on one side of the data, and the side with more variability provided by alternate fields is preferable.

In this example, we use the WHALES birth table and calculate standardized name frequency by decade, then join in the first and last name frequency scores by DOB (decade). We also used a pseudo-near neighbor approach using Levenshtein distance. For example, the name DAVV would receive the lower (higher frequency) score from DAVE if, in fact, DAVE was more frequent than DAVV in that decade. These files were produced separately and were imported in. I will leave you to decide what is best for your data, but this step is highly encouraged because the model will then be able to cluster off very common names from extremely rare names (something that probabilistic weights do accurately and a SVM without these values does not account for)

Here we create a unique identifier for each comparison (linkId) which we will use to tell *data.table* that we want row-wise calculations. We also doublecheck leading zeros because I exported this to Excel for a quick QA. There are better ways to do this, like creating a solid R object in addition to a CSV. But, hey, we all are stuck in our ways sometimes:

```
CHARLES.2 = CHARLES2 %>%
  mutate(
    linkId = row_number(),
    SSN.x = str_pad(SSN.x, 4, pad = '0'),
    SSN.y = str_pad(SSN.y, 4, pad = '0')
  )

CHARLES.3 = CHARLES.2 %>% mutate(Decade = as.numeric(substr(DOB.y, 1, 4)) -
                                  as.numeric(substr(DOB.y, 4, 4))) %>%
  left_join(fname1, by = c('Decade', 'FIRST_NAME')) %>%
  mutate(nameScore = ifelse(is.na(nameScore), .99, nameScore)) %>%
  rename('freqFirst' = nameScore) %>%
  left_join(lname1, by = c('Decade', 'LAST_NAME')) %>%
  mutate(nameScore = ifelse(is.na(nameScore), .99, nameScore)) %>%
  rename('freqLast' = nameScore)
```

Step 8 – Get ready to train sample #1

The fact of the matter is that there are relatively few true-links in CHARLES, compared to the ocean of non-links that have nothing in common except SEX and a DOB within 1 hamming distance (or share just a birthday). That is fine for now because our job is to identify where the true-links are and *begin* to teach the SVM how to identify them. However, because our CHARLES set is so unbalanced in terms of true-links and non-links, our initial sample will be very unbalanced as well. This requires a larger sample, but don't worry. You won't be reviewing 1 million records by hand because there is a trick that works very well. Also, this number is highly dependent on how much data you are working with and the size of your combined files (here CHARLES). In short, play with this number until you identify a few links via the strategy outlined in step 9. It could be far fewer than a million!

Here we will also demonstrate a splitting strategy. For powerful machines, you may get away with not splitting the dataset, but it is nonetheless a very useful tool in managing RAM and optimizing speed. This is a good method of introducing you to the concept we will heavily rely on once we run our model on the entire dataset:

```
CHUCK.1 = CHARLES.3 %>% sample_n(1000000)
CHUCK_SPLIT.1 = split(CHUCK.1, (seq(nrow(CHUCK.1))-1) %/% 10000)
```

Step 9 – Calculate SVM inputs

This step typically has a big learning curve, so I pasted the whole function in Appendix A to go into further detail. Once you learn the process, it should be easy enough to adapt it to your own data. This is also the meat of the algorithm and is 90 percent of the process, so it is worth knowing in and out. If you need to brush up on your *data.table,* please refer to the cheat sheet I provided above.

First, we set up our parallel processing as we did before. Nothing new here.

Note: Since we are working with lists, we use *length()* because our 'i' variable is essentially a DF from a list. If we were doing this sequentially from the sample, make sure to use *nrow()* so it sends individual records to the nodes. Again, feel free to mess with the number of cores depending on how your RAM is performing on your machine (if it is near 100 percent, maybe drop down the number of cores. However the sample isn't 'HUGE' so you may get away with lots of cores depending on your machine)

```
no_cores = (detectCores() / 2) - 1
cl = makeCluster(no_cores, type = "SOCK")
registerDoParallel(cl)
```

```
x9 = foreach(
  i = 1:length(CHUCK_SPLIT.1),
  .combine = rbind,
  .packages = c('tidyverse',
                'stringdist',
                'data.table',
                'e1071')
)
```

Next, we dive into our *%dopar%* function. This is what each worker-node will perform for each 'i'. In this example, for each 10,000 row dataframe in our 1 million row "link" sample. Let's quickly go over what variables we will be computing, which will eventually go into our example SVM model:

1. DOB Hamming distance - either 1 or 0 because of the *fuzzy_join* we performed earlier
2. First Name Bigram Cosine Distance - between 0 and 1, 0 being a perfect match in bigrams
3. First Name Trigram Cosine Distance - between 0 and 1, 0 being a perfect match in trigrams
4. Middle Initial Disagree? - either 1 or 0, a disagreement is a value of 1 if both fields are populated
5. Middle Initial Missing? - either 1 or 0, flag if middle initial is missing in one or both fields
6. Last Name Bigram Cosine Distance - between 0 and 1, 0 being a perfect match in bigrams
7. Last Name Trigram Cosine Distance - between 0 and 1, 0 being a perfect match in trigrams
8. SSN Hamming distance - between 0 and 1, 0 being a perfect match. Hamming distance divided by 4.
9. SSN Missing - either 1 or 0, flag if SSN is missing in one or both fields
10. SEX disagree? - either 1 or 0, a disagreement is value of 1 if both fields are populated
11. Is female flag - either 1 or 0, if target individual (WHALES side) is female, value is 1 not calculated here but included in model:
12. First name frequency score - between .01 and .99 with .01 being the most common name
13. Last name frequency score - between .01 and .99 with .01 being the most common name

The first thing we do is convert our dataframe of interest into a *data.table*:

```
pool = data.table(CHUCK_SPLIT.1[[i]])
```

Now we can start with our calculations. In this example, we begin with DOB hamming distance. Here, data table (DT) is creating a new variable 'DOB_HAM' and populating that field, for each row, with the *stringdist* calculation. This is equivalent to *mutate()* in *dplyr*, but much faster with large data.

Note: the 'linkId'is DT's grouping variable and essentially tells DT to work in a row-wise fashion. In simple calculations like this, it is unnecessary but doesn't slow down DT at all. Later we perform *min()* or *max()* calculations in a row-wise fashion and these parameters are critical. We include them to be safe—in case a duplicate makes its way in or something funky!

```
pool[, 'DOB_HAM' := stringdist(DOB.x,
                               DOB.y,
                               method = c('hamming')), linkId]
```

Here, we get fancy. Since WHALES had multiple first name fields, we are going to calculate the bigram distances for each combination of first names then take the *min()* of these values to be our true bigram cosine distance (ultimately selecting the closest one: i.e., if 'Richard' is being compared to the first name 'Rich' and an alternative first name of 'Dick' then 'Richard' will be compared to 'Rich'). We further convoluted this example by the fact that CHARS has some data depreciation around 2009, and only one or two characters of a name field are populated. To handle these short names when calculating bigram cosine distance, let alone trigram distance, we add some custom logic that I will describe a bit more in depth below:

The first name trigram example is below:

```
pool[, 'FIRSTNAME_COS_1a1' := stringdist(
  FIRST_NAME_1,
  FIRST_NAME,
  method = c('cosine'),
  q = ifelse(
    nchar(FIRST_NAME) < 3 |
      nchar(FIRST_NAME_1) < 3,
    min(nchar(FIRST_NAME), nchar(FIRST_NAME_1)),
    3
  )
), linkId]

pool[, 'FIRSTNAME_COS_1b1' := ifelse(
  nchar(FIRST_NAME) < 3 &
    substr(FIRST_NAME, 1, nchar(FIRST_NAME)) != substr(FIRST_NAME_1, 1, nchar(FIRST_NAME)) |
    nchar(FIRST_NAME_1) < 3 &
    substr(FIRST_NAME, 1, nchar(FIRST_NAME)) != substr(FIRST_NAME_1, 1, nchar(FIRST_NAME)),
  1,
  FIRSTNAME_COS_1a1
), linkId]

pool[, 'FIRSTNAME_COS_2a1' := stringdist(
  FIRST_NAME_2,
  FIRST_NAME,
  method = c('cosine'),
  q = ifelse(
    nchar(FIRST_NAME) < 3 |
      nchar(FIRST_NAME_2) < 3,
    min(nchar(FIRST_NAME), nchar(FIRST_NAME_2)),
    3
  )
), linkId]

pool[, 'FIRSTNAME_COS_2b1' := ifelse(
  nchar(FIRST_NAME) < 3 &
    substr(FIRST_NAME, 1, nchar(FIRST_NAME)) != substr(FIRST_NAME_2, 1, nchar(FIRST_NAME)) |
    nchar(FIRST_NAME_2) < 3 &
    substr(FIRST_NAME, 1, nchar(FIRST_NAME)) != substr(FIRST_NAME_2, 1, nchar(FIRST_NAME)),
  1,
  FIRSTNAME_COS_2a1
), linkId]

pool[, 'FIRSTNAME_COS_3a1' := stringdist(
  FIRST_NAME_3,
  FIRST_NAME,
  method = c('cosine'),
  q = ifelse(
    nchar(FIRST_NAME) < 3 |
      nchar(FIRST_NAME_3) < 3,
    min(nchar(FIRST_NAME), nchar(FIRST_NAME_3)),
    3
  )
), linkId]
```

```
pool[, 'FIRSTNAME_COS_3b1' := ifelse(
  nchar(FIRST_NAME) < 3 &
    substr(FIRST_NAME, 1, nchar(FIRST_NAME)) != substr(FIRST_NAME_3, 1, nchar(FIRST_NAME)) |
    nchar(FIRST_NAME_3) < 3 &
    substr(FIRST_NAME, 1, nchar(FIRST_NAME)) != substr(FIRST_NAME_3, 1, nchar(FIRST_NAME)),
  1,
  FIRSTNAME_COS_3a1
), linkId]

pool[, 'FIRSTNAME_COSn3' := min(FIRSTNAME_COS_1b1,
                                FIRSTNAME_COS_2b1,
                                FIRSTNAME_COS_3b1,
                                na.rm = T), linkId]

pool[, 'FIRSTNAME_COSn3' := ifelse(FIRST_NAME == '' |
                                     FIRST_NAME == ' ', 1, FIRSTNAME_COSn3)]
```

Here, we compute three cosine distance (trigram) because there are three first name fields in our WHALES-side dataset. We want the *min()* of those, which results in our final value “FIRSTNAME_COSn3”.

For each name comparison, there are two calculations:

1. Calculate the cosine distance with a trigram. If one of the names has fewer than 3 characters, drop the nGram size to be that of the minimum nGram size. For example, comparing ‘SE’ and ‘SEAN’, this calculation would effectively mirror the bigram calculation because the nGram length would be 2.
2. Additional logic: If the name is less than three characters, require the first letter of each name to match. For example, ‘SE’ and ‘SEAN’ would receive the calculated ‘bigram’ score from number the first calculation, even though it is a trigram calculation. However, if the names were ‘EA’ and ‘SEAN’, the calculated similarity would be replaced by a 1, which indicates no calculated similarity.

This may be entirely unnecessary for your data, but additional logic is typically required to fit data’s intricacies. This is one such example.

We then repeat this calculation for all name comparison fields, followed by a final *min()* calculation for bigram first name, trigram last name and bigram last name. This is the bulk of the function.

Next, we calculate SSN Hamming and add a field that flags if one or more SSNs are missing. This teaches the SVM to differentiate between fields that are populated but disagree completely, and a value that is compared to a missing value. (Both would receive a Hamming distance of 1.) This is desirable because a true disagreement in SSN fields that are populated is grounds for exclusion, but a disagreement to a missing value shouldn’t add or subtract any information to our categorization task. You may be wondering how the SVM knows that flag is referring to the SSN Hamming distance. The answer is it doesn’t. The model is blind to what is and what is not a flag or calculation. But in hyperspace, these flags split clusters that would normally be chunked together, allowing the model to treat them ‘separately’ with different hyperplane criteria.

Note: We use ‘0000’ as a missing SSN because our leading zero fix, also 0000, is an invalid SSN. That is great news for us!

```
pool[, 'SSN_HAM' := stringdist(SSN.x,
                               SSN.y,
                               method = c('hamming')) / 4, linkId]

pool[, 'SSN_Missing' := ifelse(SSN.x == '0000' |
                                 SSN.y == '0000', 1, 0), linkId]

pool[, 'SSN_HAM' := ifelse(SSN_Missing == 1, 1, SSN_HAM)]
```

You may have noticed I skipped middle initial, but I wanted to introduce it with SEX because they use the same "Do these disagree?" logic. "Do these disagree?" basically asks just that. If both values are present do they disagree? If so, the value is 1. If one or more values are missing or they agree, the calculated value is 0. Then, to differentiate those that truly agree with those with missing values, we also include a missing flag.

Note: We used a missing flag for SEX, but every comparison we had in our CHARLES flag had a SEX value. This causes an error in the generation of the SVM model, and if there is no variance in a variable, that variable should be removed prior to training the SVM. Also note the overkill of possible missing values. This is unnecessary, but it gives me peace of mind.

```
pool[, 'MINAME_Disagree' := ifelse(
  MIDDLE_INIT.x == MIDDLE_INIT.y |
    is.na(MIDDLE_INIT.x) |
    is.na(MIDDLE_INIT.y) |
    MIDDLE_INIT.x == '' |
    MIDDLE_INIT.y == '' |
    MIDDLE_INIT.x == ' ' |
    MIDDLE_INIT.y == ' ',
  0,
  1
), linkId]

pool[, 'MINAME_Missing' := ifelse(MIDDLE_INIT.x == '' |
                                    MIDDLE_INIT.y == '' |
                                    MIDDLE_INIT.x == ' ' |
                                    MIDDLE_INIT.y == ' ',
                                  1,
                                  0), linkId]
...

pool[, 'SEX_Disagree' := ifelse(
  SEX.x == SEX.y |
    is.na(SEX.x) |
    is.na(SEX.y) |
    SEX.x == '' |
    SEX.y == '' |
    SEX.x == ' ' |
    SEX.y == ' ',
  0,
  1
), linkId]

pool[, 'SEX_Missing' := ifelse(SEX.x == '' | SEX.y == '' |
                                 SEX.x == ' ' |
                                 SEX.y == ' ', 1 , 0), linkId]
```

Next, we create another flag field (isFemale) to tell the machine that there are fundamental differences between sex values and how their calculated fields should be weighted in its classification of links and non-links. This helps account for, and correct, sex gaps commonly found in cultures where women change last names upon marriage.

Here we also create a calculated field called 'SORT_HELP'. This will be your best friend in the manual review process. What it does is provide a cumulative score for the distance calculated fields, where lower scores are more likely to be true-links and higher scores are more likely to be non-links. This is similar to a non-weighted cumulative probabilistic score, but it is important to note that the SVM does not use this variable. This is purely for our sanity in reviewing thousands of lines of names.

Finally, we wrap up by removing any NA or INF that could have snuck in (which is unlikely, but not impossible) and making those values a solid 1. Then, we select the fields to stare at for a few hours while we sift through potential links to find those elusive true-links!

```
pool[, 'isFemale' := ifelse(SEX_Disagree == 0 &
                              SEX.x != 'M', 1, 0), linkId]


pool = pool[, lapply(.SD, function(x) {
  ifelse(is.na(x), 1, x)
})]

pool = pool[, lapply(.SD, function(x) {
  ifelse(is.infinite(x), 1, x)
})]

pool[, 'SORT_HELP' := DOB_HAM + SSN_HAM + FIRSTNAME_COSn3 + FIRSTNAME_COSn2 +
       LASTNAME_COSn3 + LASTNAME_COSn2 + SEX_Disagree + MINAME_Disagree, linkId]

pool2 = pool[, .(
  linkId,
  DischargeRecordID,
  DEATH_SFN_NUM,
  DOD,
  DOB.x,
  DOB.y,
  DOB_HAM,
  SSN.x,
  SSN.y,
  SSN_HAM,
  SSN_Missing,
  SEX.x,
  SEX.y,
  SEX_Disagree,
  SEX_Missing,
  FIRST_NAME,
  FIRST_NAME_1,
  FIRST_NAME_2,
  FIRST_NAME_3,
  FIRSTNAME_COSn3,
  FIRSTNAME_COSn2,
  LAST_NAME,
  LAST_NAME_1,
  LAST_NAME_2,
  LAST_NAME_3,
  LAST_NAME_4,
  LASTNAME_COSn3,
  LASTNAME_COSn2,
  MIDDLE_INIT.x,
  MIDDLE_INIT.y,
  MINAME_Disagree,
  MINAME_Missing,
  isFemale,
  freqFirst,
  freqLast,
  SORT_HELP
)]

return(pool2)
```

Don't forget to *stopCluster*!

At this point, you now have 1 million record pairs that are overwhelmingly garbage non-links, but that is ok! Export everything to Excel, or however you want to label these. Sort by SORT_HELP, and begin to hand label links from non-links in a column named 'isLink'.

Some notes from this first sample:

1. I cannot stress this enough: You may find very few links. That is ok for now!
2. Links you do find may be extremely obvious (i.e., a handful of deterministic matches and a bunch of garbage). Again, this is ok for now.
3. *Do not read all 1 million rows!* Go until SORT_HELP falls apart and you hit a very large chunk of non-links. Feel free to filter by name similarity fields to seek out possible interesting inexact links, but at this point all we are trying to do is get a feel for our data and hopefully capture a few obvious links.

Step 10 – Train the SVM(s)

Congratulations! We are ready to train our first SVM. But don't get too excited, as chances are, it won't be very good… yet!

Read back in your labeled data (here named 'tset1'). This is your first training set. Train two SVMs:

1. *as.factor(isLink)* ~ will predict whether something is a link or not a link (1, 0). This is a true binary classifier.
2. *isLink* ~ will produce a Platt scaled score, where higher numbers are more likely to be a link in the eyes of the SVM (between 0 and 1). Alongside SORT_HELP, this will be your other best friend.

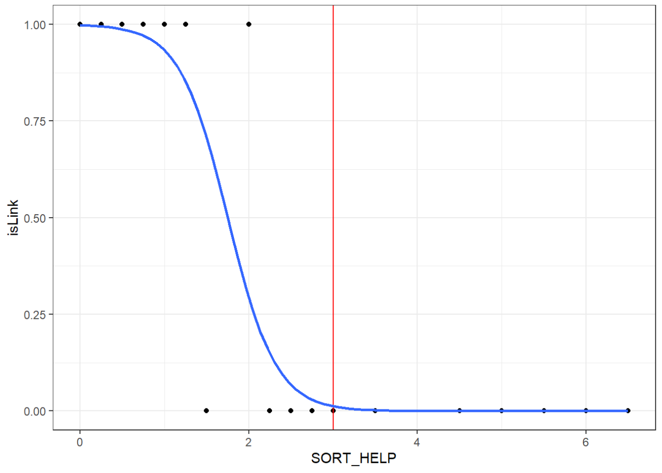
```
t1_SVM = svm(
  data = tset1,
  as.factor(isLink) ~ DOB_HAM + SSN_HAM + SEX_Disagree + FIRSTNAME_COSn3 +
    FIRSTNAME_COSn2 + LASTNAME_COSn3 + LASTNAME_COSn2 + MINAME_Disagree +
    SSN_Missing + MINAME_Missing + freqFirst + freqLast + isFemale
)

t1_SVM_score = svm(
  data = tset2,
  isLink ~ DOB_HAM + SSN_HAM + SEX_Disagree + FIRSTNAME_COSn3 +
    FIRSTNAME_COSn2 + LASTNAME_COSn3 + LASTNAME_COSn2 + MINAME_Disagree +
    SSN_Missing + MINAME_Missing + freqFirst + freqLast + isFemale
)
```

Feel free to save these models out and keep them organized in a folder. This is an iterative process, and it can get very messy very quickly.

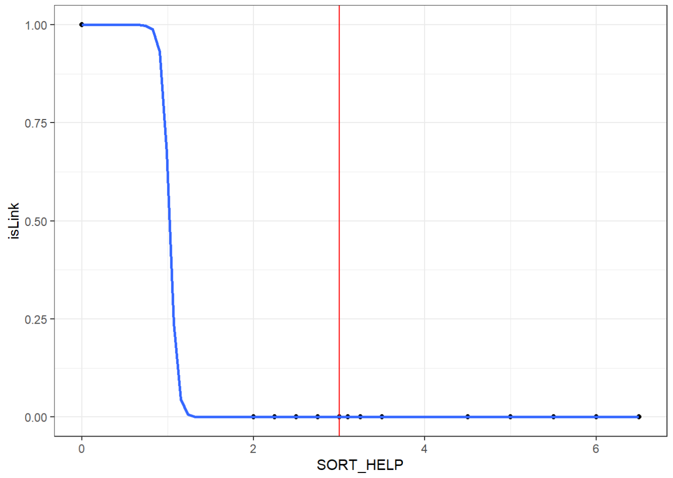
Step 11 – Determine where to 'zoom' in for the next sample

In short, we are going repeat what we just did with a few minor adjustments. First, we will include a SORT_HELP filter. This will drastically mitigate the amount of manual review needed in each subsequent iteration. To do this, plot your training set and draw a logistic curve to observe where you can make a conservative filter for obviously non-links only (at this time). Some examples are below:

```
##Example 1 - Good sample with some overlap (i.e. links and non-links) by SORT_HELP
```



```
##Example 2 - Less optimal sample with no overlap (i.e. links and non-links) by SORT_HELP
```



The first example is more ideal, but if you are only getting a few deterministic links and a bunch of garbage, it will look more like the second one. I am *not* advocating for a cut at exactly 3, and I wish I could give more advice or a hard-and-fast rule to determine how to make this cut, but it is completely determined by your data. Do your best to make a conservative cut that will most likely not include any links. As uncomfortable as this part of the technique may be to some people, as you perfect the 'art' of this, it will save you some training headaches.

The rationale behind this is twofold. First, it will save you time and cut the size of our sampled files for review down by cutting out the garbage. It will also allow you to start zooming in on the interesting area of this curve that holds the non-obvious and inexact true-links. We want to capture as many of those non-obvious, inexact links as possible to ingest the type of variances we observe in inexact links into the model's training set. There are more advanced techniques if you struggle to get any interesting links, but give it a few iterations. Then, try fuzzy joins or your own flavor of finding inexact links (see note below). A non-obvious link, or 'interesting' link, is an inexact match that would likely be missed using deterministic or fuzzy methodology. For a review of deterministic and fuzzy methods, please review the introduction video.

Note: If your link incidence is very low for your project and you are working with a ton of data, you will likely have a very unbalanced set. This takes some tinkering beyond SORT_HELP cuts. Try to filter your samples by inexact match combinations, especially if you are working with a pairwise file (like CHARLES here). For example, you can do an iteration where the first and last name have a similarity score higher than 0 but lower than one. Then, do one with inexact SSN and inexact last names, then first names, then so on and so on. This can get intensively manual quickly, but remember if your incidence of a link is low, capturing these links is very critical. Take pride that you are likely finding sub-populations that other algorithms often miss.

Step 12 – Run second sample set with determined cuts and predict

Once again, we are zooming in on the area of this curve where the probability of a link is not 0, and we are accepting those at a 1 probability now as well.

If your training file is extremely unbalanced with very few links (probably deterministic), and your curve looks like the second example, up your sample. If you are capturing a good number of interesting links, keep it the same or even lower it. Play with this value.

Besides the SORT_HELP filter, we will also be including the predictions from our SVM built by the previous iteration:

```
CHUCK.2 = CHARLES.3 %>% sample_n(500000)
CHUCK_SPLIT.2 = split(CHUCK.2, (seq(nrow(CHUCK.1))-1) %/% 10000)

no_cores = (detectCores() / 2) - 2
cl = makeCluster(no_cores, type = "SOCK")
registerDoParallel(cl)

x9 = foreach(
  i = 1:length(CHUCK_SPLIT.2),
  .combine = rbind,
  .packages = c('tidyverse',
                'stringdist',
                'data.table',
                'e1071')
) %dopar% {
  pool = data.table(CHUCK_SPLIT.2[[i]])

  ...# All those calculations from above #

  pool[, 'SORT_HELP' := DOB_HAM + SSN_HAM + FIRSTNAME_COSn3 + FIRSTNAME_COSn2 +
         LASTNAME_COSn3 + LASTNAME_COSn2 + SEX_Disagree + MINAME_Disagree, linkId]

  pool['SORT_HELP' < 4] # HERE IS THE SORT_HELP FILTER



  pool2 = pool[, .(
    linkId,
    DischargeRecordID,
    DEATH_SFN_NUM,
    DOD,
    DOB.x,
    DOB.y,
    DOB_HAM,
    SSN.x,
    SSN.y,
    SSN_HAM,
    SSN_Missing,
    SEX.x,
    SEX.y,
    SEX_Disagree,
    SEX_Missing,
    FIRST_NAME,
    FIRST_NAME_1,
    FIRST_NAME_2,
    FIRST_NAME_3,
    FIRSTNAME_COSn3,
    FIRSTNAME_COSn2,
    LAST_NAME,
    LAST_NAME_1,
    LAST_NAME_2,
    LAST_NAME_3,
    LAST_NAME_4,
    LASTNAME_COSn3,
    LASTNAME_COSn2,
    MIDDLE_INIT.x,
    MIDDLE_INIT.y,
    MINAME_Disagree,
    MINAME_Missing,
    isFemale,
    freqFirst,
    freqLast,
    SORT_HELP
  )]

  # HERE ARE THE PREDICTIONS FROM THE SVM
  pool2$predLink = predict(t1_SVM, pool2)
  pool2$predLinkScore = predict(t1_SVM_score, pool2)

  return(pool2)
}
stopCluster(cl)
```

Step 13 – Evaluate predictions

This part is as fun as it is easy. Export your file in the preferred way just as you would for any manual review.

Note: HIDE THE PREDICT COLUMN. NO PEEKING. NO EXCEPTIONS!

Proceed to sort by SORT_HELP and repeat the process that you did in the first iteration. Save the labeled file and import it back into R. Now we can evaluate how our model did with the 'hopefully' zoomed in sample. Do not panic if it did not perform well.

There are a few ways to do this:

- For unbalanced sets, you can use Cohen's kappa. Use this metric if you have multiple reviewers as well.
- For balanced sets, analyze the sensitivity and specificity of the model
- You could also focus on where the model predicts true-links (more useful for later iterations) and calculate false discovery rate. We use FDR because we do not know how many true-links exist, and FDR helps us focus on type 1 error evaluation.

In either case, you will most likely be creating an "agree" column and comparing the "isLink" column that you hand labeled to your "predLink" column. Calculate and document your desired performance metric.

Next, we remove the "predLink" and "predLinkScore" columns (as well as the agree column, and any others you may have added) and *rbind* our hand labeled training set together. Once both the first and second samples training sets are in one dataframe, retrain the SVMs and save those models.

Step 14 – Repeat, repeat, repeat

After the second iteration, you should have a feel for your data. Keep zooming in, adjusting sample sizes, and adding to your full training set. Keep predicting based off the latest trained SVM and keep track of the model's performance.

Ideally, you will begin getting samples rich with non-obvious links and your performance metrics will begin to plateau.

When do you stop this process? That is a difficult question, and the answer is entirely up to you. I like to shoot for above 99 percent agreement. When you see performance starting to plateau and you feel satisfied with the diversity of non-obvious link examples you have in your training set, call it quits. If you have gaps somewhere, it's likely you will discover them down the road in QA.

One of the biggest benefits of implementing a machine learning linkage strategy is just this: it allows maximum flexibility. If you miss a sub-population weeks down the line, all you have to do is supplement your master training dataset with examples from that population and retrain the model.

Be patient, and stay vigilant. This is the most painful part of the process.

Step 15 – Run Your finalized model on the entire dataset

This is going to look familiar, but since we are likely working with massive data, I want to show some examples of optimization techniques that go beyond parallel processing.

We'll begin with a little background. R is not the greatest for parallel processing, and it relies on a few packages to achieve this time saving process. Here we use *parallel* and *snow*, though there are other options to choose from. Regardless, when you initialize a parallel processing *%foreach%* loop, you are essentially calling on multiple processors to tackle a "single" task. Remember how I mentioned visualizing this as multiple RStudio sessions? This is similar. (Open your task manager and see for yourself). To achieve this, R sends the data you specify out to these nodes (as we went over earlier), which increases your RAM burden by a factor of how many nodes (or cores) you decide to utilize. For example, if I send a billion-line dataframe to 20 workers, my computer won't be able to handle it. R has a hard time handling a single file with long vectors in a single environment, let alone multiple. As a result, we need to be cognizant of a few things as we apply all these techniques to the whole dataset. That includes:

1. Our computer's RAM and what are we asking it to do (I use task manager to monitor the process)
2. Using more cores quickens the process, but the computer experiences more memory burden
3. The length of time this will take and if it's acceptable

The goal is to balance the process. Go fast without 'killing' workers. Unfortunately, there is no universal answer to the most optimal parameters for your machine or data. You will have to play around with this. But being familiar with all the variables discussed in this training module will help you identify where you can optimize your own project.

One lesson I learned, splitting is your friend, also cleaning up your master environment and using garbage collect, *gc()*, seems to prep your memory for a large job. Post-optimization, if your process occasionally fails, think about saving out intermediate files. We also wrap the entire thing in a *pblapply* to get a time estimate:

```
# split these into batches
CHARLES_SPLIT = split(CHARLES.3, (seq(nrow(CHARLES.3))-1) %/% 1500000)

# Clean Environment
rm(CHARLES, CHARLES.2) # remove all large data variables from your env that you are not using
gc()

# Wrap dopar for batches OUTSIDE of dopar fun
doubleSplit = function(x){
  CHUCK = CHARLES_SPLIT[[x]]
  CHUCK_SPLIT = split(CHUCK, (seq(nrow(CHUCK))-1) %/% 100000)

no_cores = (detectCores()/2) - 2
cl = makeCluster(no_cores, type = "SOCK")
registerDoParallel(cl)
start.time = Sys.time()
x9 = foreach(
  i = 1:length(CHUCK_SPLIT),
  .combine = rbind,
  .packages = c('tidyverse',
                'stringdist',
                'data.table',
                'e1071')
) %dopar% {
  SAME CALCULATIONS IN ALGORITHM PRESENTED IN APPENDIX A

  pool2$predLink = predict(t5_SVM, pool2)
  pool2$predLinkScore = predict(t5_SVM_score, pool2)

  pool3 = pool2[predLink == 1]

  return(pool3)
}
stopCluster(cl)
return(x9)
}

x10 = pblapply(1:length(CHARLES_SPLIT), doubleSplit)
x11 = do.call('rbind', x10)
```

We start with the CHARLES file, which is split by year, resulting in ~500 million rows. Then, we split the data into chunks of 1.5 million rows before popping into the *pblapply* function level. At this point, the function selects one of those 1.5 million line dataframes from the list and splits it again into chunks of 100,000 lines before initializing the *%dopar%* function. Why do we do this? Well, the first split enables us to send each of the workers a list of 100,000 dataframes with a total size of 1.5 million, which my machine can handle. Then, each worker chooses a dataframe from that split list with a *nrow* of 100,000 to work on calculations and predictions. Once that list of 1.5 million is exhausted, the parallel process stops and the nodes report the compiled links for that 1.5 million dataframe. Then the *pblapply* function says restarts the process, initializing a brand new parallel process for the next 1.5 million potential links.

All in all, this process takes a lot of tinkering and your optimal numbers will almost certainly be different. But I am able to process about 500 million rows in just over 6 hours, for example (enough time to catch up on documentation and to begin training your next model!).

Step 16 – QA

By this time, you are very familiar with your model and your data. There are many ways you can QA your final output, I encourage people to follow these steps at a minimum:

1. Run a deterministic and fuzzy filter through your CHARLES file and compare the outputs. Sometimes, you will find subpopulations that your model missed (gender changes, for example are very commonly missed). Supplement your model with the links it missed. If the population is large, supplement your training set, retrain your model, and run it again.
2. Sample and hand check. This is painful, but it is rewarding to quantify your model's performance! Calculate those metrics and look for oddities.
3. Speaking of oddities, a visual check of your complete link file is also very handy. Filter the rows, look for cases where names don't match but scores say they do, etc. I hope I have squashed many bugs for you in the initial months of development, but starting with a different dataset can always result in the unexpected.

At this point, you will have the saved the final model and you can very quickly implement it for non-historical new data in an automated fashion that takes significantly less time!

Final demo thoughts:

1. Know your data.
2. Know your goals and the requirements of your linkage project.
3. Try to have fun with the process. You are building a robot army!
4. Expect the process to be bumpy, and expect a learning curve.
5. Be flexible. This is a quickly developing strategy and recent updates may not be included in this script.
6. Make it your own. *This report is just one way.* There are tons of ways to do anything in R. Be creative, and try new things.

ECHIDNA machine learning evaluation

ECHIDNA aims to link WHALES death record data with CHARS discharge data. Here, we compare deterministic, fuzzy, Fellegi-Sunter and machine learning strategies in the linking of death record data with all-time discharge data. This evaluation found that the machine learning (ML) model found more total links, linked more WHALES records, and was more accurate in those link designations than any other model. Furthermore, the ML model found more minority race and ethnicity links compared to FS.

For this evaluation, we will focus on WHALES death records from 2019 to 2020 (n = 59,698) and finding matching patient records in all time CHARS (n = 13,160,882).

Evaluation introduction

This evaluation compares the machine learning (ML) methodology to the currently implemented probabilistic methods, specifically Fellegi-Sunter (FS), and deterministic methods, including exact match and 'fuzzy' logic. This comparison helps CHS evaluate the pros and cons of updating current linkage strategies.

Firstly, we will briefly introduce each model. Then, we will compare performance metrics. We will look at the sensitivity, specificity, and accuracy of each strategy with a particular focus on type 1 error rates and false discovery rates (FDR). Additionally, we will investigate proportional capture of links by race and ethnicity and discuss how models may handle name fields differently. The report ends with a note about computation time and resources, followed by a brief general discussion.

ECHIDNA 2019-2020 overview

The table below lists the fields used in this linkage.

Table 1: Fields used in this linkage

Field Desc	WHALES	CHARS
First Name (character)	GNAME	PatientFirstName
	AKA(1-2)_FNAME	
	CHILD_GNAME (from birth link)	
Last Name (character)	LNAME	PatientLastName
	AKA(1-2)_LNAME	
	CHILD_LNAME (from birth link)	
	LNAME (split)	
Middle Initial (character)	MNAME	PatientMiddleName
	CHILD_MNAME (from birth link)	
Date of Birth (numeric)	DOB	PatientBirthDate
Last 4 of SSN (numeric)	SSN	PatientLastFourSSN
SEX (character)	SEX	PatientSexCode

WHALES derived name fields were capped at up to three distinct first names and four distinct last names per individual. The birth table was used to supplement name field data by leveraging existing links between the birth and death tables (as previously discussed).

Deterministic and 'fuzzy' strategy overview

The deterministic model requires exact matches in four fields: DOB, SSN, first name, and last name. No fields can have missing values.

Fuzzy criteria are based on the same four fields (still omitting SEX) but allow inexact matches to DOB, names, and SSN fields. To be considered a match in the fuzzy model, both the first and last name must be below or equal to a Levenshtein distance of 1. Additionally, SSN and DOB comparisons must have a Hamming distance of 1 or below.

FS strategy overview

The FS strategy follows past protocols and uses the Fellegi-Sunter method (FS Overview). It deviates from past DOH scripts to expedite the process and conform our FS model closer to what is in the literature. FS matches are predicated on an initial field match decision, followed by a weighted score calculation. Current FS implementations block on exact DOB (i.e., requires exact matches in the DOB field), so we do the same here. For additional comparison to out-of-the-box solutions, please refer to our COVID case and vaccination linkage report available on the Analytical Methods and Reports webpage. For this evaluation, the initial match decision rules are below:

Table 2: Initial match decision rules

Field	Match Criteria
First Name	Jaro-Winkler similarity of >.85
Middle Initial	Exact Match
Last Name	Jaro-Winkler similarity of >.85
SSN	Exact Match
Sex	Exact Match

Weighted scores for FS are determined by two probabilities: m and u. Missing field comparisons are given a 0 score. The u probability is defined as the probability that an identifier in two non-matching records would agree purely by chance. The m probability is the probability that an identifier in matching pairs will agree.

Table 3: *M* and *u* probability

Field	*m* probability	*u* probability
First Name	.99	Proportion of name in WHALES by decade*
Middle Initial	.97	.01
Last Name	.99	Proportion of name in WHALES by decade*
SSN	.99	.0001

Sex	.97	.48

*Name frequency for the FS model uses a proportion calculated from the prevalence of each name's most common derivative. For example, the name "Jon" is within a Levenshtein distance of 1 of "John", and if "John" was more frequently present in that decade than "Jon", "Jon" would receive the same proportion as "John".

These probabilities could be even more refined, but we were able to get ample distribution separation with these numbers. This 'bimodel' distribution is displayed in Figure 1 and in log scale in Figure 2. If multiple names exist for a single data field (first and/or last name), the maximum weighted probabilistic score is selected (we utilize the closest name if multiple are available). Field scores are summed to get a total weighted score. M-probabilities were estimated from a previous WHALES linkage project and reflect the proportions of true-links that fell within a Jaro-Winkler similarity >.85.

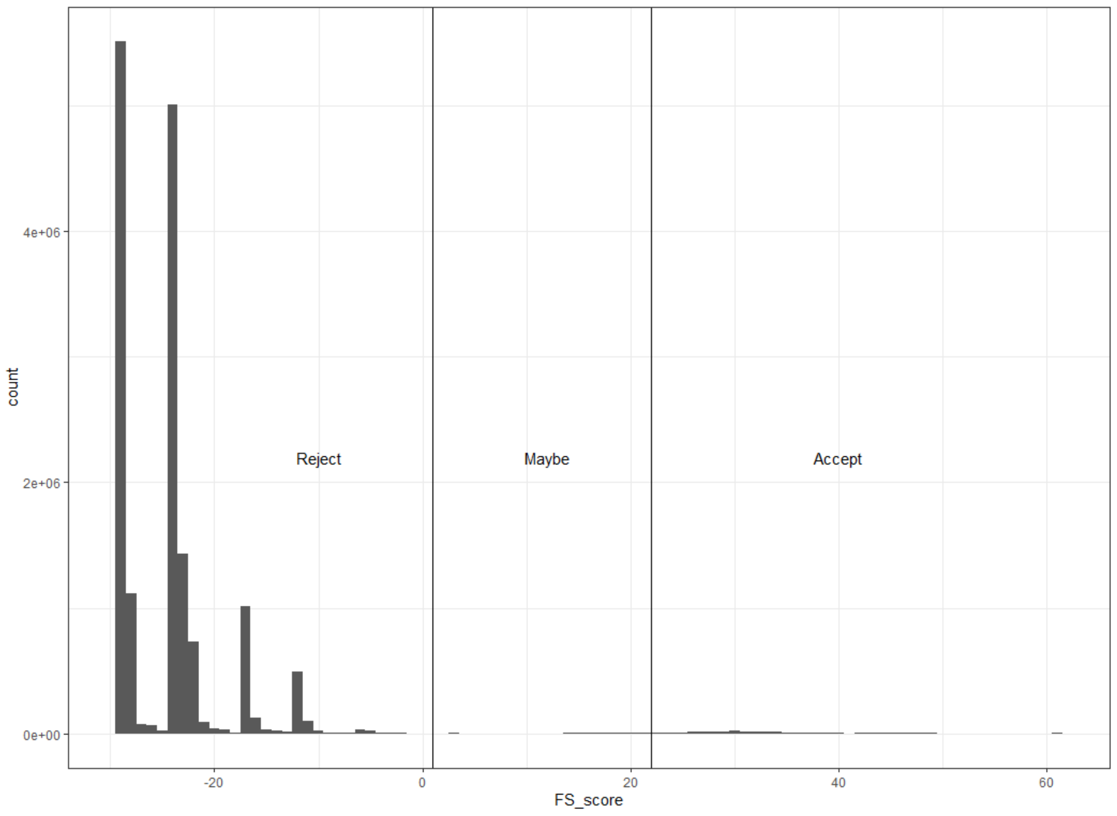


Figure 1 - Raw count of FS weighted score output of all potential links

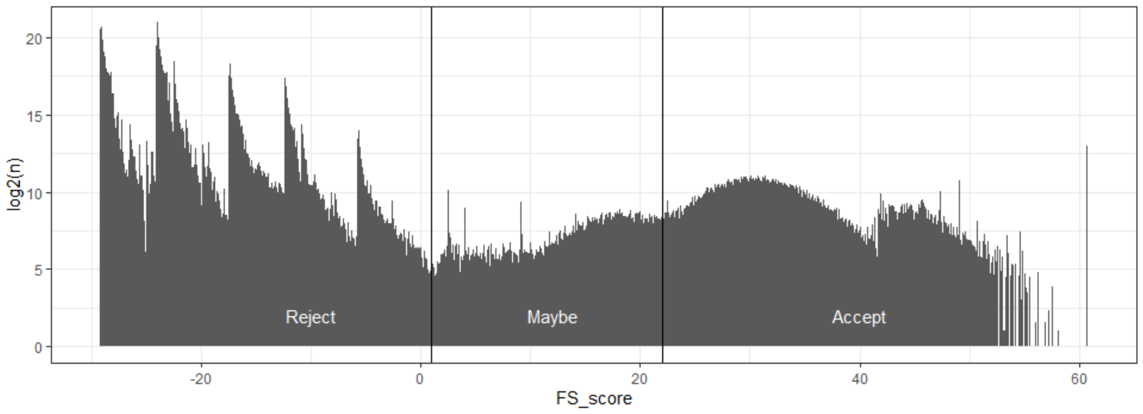


Figure 2 - Same as Figure 1 but in Log2 scale

Each category (reject, maybe and accept) were randomly sampled (n=250*3, 750 total) and manually labeled. Sensitivity and specificity were calculated at each whole number threshold in the “maybe” zone. Below we observe the tradeoff between sensitivity and specificity in this region and attempt to find a cutoff threshold that balances these parameters at a suitable level (here 95 percent). The results are below:

Table 4: Tradeoff between sensitivity and specificity in this region

Threshold FS_score value	Sensitivity (%)	Specificity (%)
2	99.5	84.8
3	99.1	87.8
4	99.1	88.9
5	98.8	90.2
6	98.0	91.2
7	97.1	92.2
8	96.9	92.5
9	96.4	93.2
10	95.8	95.3
11	94.9	95.9
12	94.4	95.9
13	92.2	97.0
14	90.5	98.3
15	88.3	98.6
16	85.2	98.6
17	78.8	98.6
18	74.0	98.9
19	67.4	98.9
20	62.7	99.6
21	59.2	100
22	55.0	100

An ideal estimated FS score threshold was determined to be greater than or equal to 10, where both the specificity and sensitivity calculated from the sample was greater than 95 percent. Performance of this model is estimated to be at the level of currently implemented FS models within DOH.

Machine learning strategy refresher

If you skipped the tutorial above, the ML strategy uses a combination of string distance calculations, dummy variables, and matching logic to train a support vector machine (SVM) to classify a pair of records as either a link or a non-link. Below are the specific calculations serving as inputs to the model.

Table 5: Specific calculations serving as inputs to the model

Field	Calculation
Bigram First Name	Minimum bigram cosine distance between all first name fields
Trigram First Name	Minimum trigram cosine distance between all first name fields
Bigram Last Name	Minimum bigram cosine distance between all last name fields
Trigram Last Name	Minimum trigram cosine distance between all last name fields
Middle Initial Disagree	If both records have a middle initial present, do they disagree?
Middle Initial Missing	Does one or both the records not have a middle name value?
SSN Hamming	Hamming distance of the last 4 digits in SSN divided by 4
SSN Missing	Does one or both the records not have a SSN value?
Sex Disagree	If both records have a sex present, do they disagree?
Sex Missing	Does one or both the records not have a Sex value?
Female Flag	Is the death record in question Sex = Female?
First Name Frequency	Name proportion derived from WHALES birth table by decade*
Last Name Frequency	Name proportion derived from WHALES birth table by decade*

*Name frequency for the ML model used a modified frequency score that calculated the prevalence of each name's most common derivative. For example, the name "Jon" is within a Levenshtein distance of 1 of "John", and if "John" was more frequently present in that decade than "Jon", "Jon" would receive the same proportion score as "John". These raw proportions are subtracted from 1, then normalized to a 0-1 scale. This results in a scaled dictionary where common names are close to 0, rare names are close to 1.

Rather than blocking on exact DOB like many previous attempts, we leveraged the computational efficiency of ML to expand linkage candidates, now including DOBs with a Hamming distance less than or equal to 1. This 'fuzzy' blocking greatly increases the number of potential links; however, from previous linkage projects, we have observed value in allowing a single clerical error in the DOB field. In other words, inexact DOB fields are not exclusionary in the ML model.

Results

Performance:

The table below displays total WHALES to CHARS links found (multiple per individual WHALES records allowed). The machine learning method found more links than any other method.

Table 6: Total links found by each strategy

Strategy	Total Links Found
Deterministic	210,410
Fuzzy	214,432
Fellegi-Sunter	262,184
Machine Learning	267,560

Rather than looking at total links, which include multiple hospitalizations, we can look at the number of death records linked to at least one CHARS record. The table below displays WHALES death records (n = 59,698) linked to at least one CHARS record:

Table 7: WHALES death records linked to at least one CHARS record by strategy

Strategy	*n* WHALES Linked	% WHALES Linked
Deterministic	42,090	70.5
Fuzzy	42,694	71.5
Fellegi-Sunter	50,738	85.0
Machine Learning	51,497	86.3

Next, we can observe the number of links found using one method that another method did not identify. Below we display the unique number of links identified in method 'X' but not in method 'Y':
Table 8: Unique number of links identified in method 'X' but not in method 'Y'

		Y			
		Deterministic	Fuzzy	Fellegi-Sunter	Machine Learning
X	Deterministic	--	0	3 A	15* B
	Fuzzy	4,022	--	1,877 C	32 D
	Fellegi-Sunter	51,777	49,629	--	910 E
	Machine Learning	57,165	53,160	6,286 F	--

*It should be noted for the actual ECHIDNA project deliverable, the ML model was supplemented with Deterministic results. Here we keep them separate. All 15 missed by the ML model here were SEX + MIDDLE INIT disagreements.

Next, we derived samples (n = 500) of each model's links and manually reviewed them to compute the false discovery rate (FDR) of each model.

Table 9: False discovery rate (FDR) of each model

Strategy	Detected False Positives (Type I Errors)	Estimated FDR*
Deterministic	0	0
Fuzzy	0	0
Fellegi-Sunter	2	.004
Machine Learning	0	0

*Please note that higher type 1 error rates are likely non-zero, they were just undetectable at this sample size

Additionally, samples (up to n = 100) were taken from the cells above letters A-F to estimate the FDR of links within those cells. We then manually investigated both type I and type II errors in cells A-F and described them below:

Table 10: Type I errors and description

		Y			
		Deterministic	Fuzzy	Fellegi-Sunter	Machine learning
X	Deterministic	--	--	0 A	0 B
	Fuzzy	--	--	0 C	.38 D
	Fellegi-Sunter	--	--	--	.48 E
	nGram + SVM	--	--	0 F	--

Superscript	Sample Size	X Type I Errors	Y Description
A	3	0	FS missed a few sex disagreements with very common names
B	15	0	ML missed sex disagreements with both differing first names and middle initials
C	100	0	FS missed the following: • 74% different DOBs • 26% both misspelled names and differing SSNs
D	32	12	ML missed the following: • 44% (14) sex disagreements with both differing first names and middle initials (stemming from 3 WHALES records) • 16% (5) differing SSNs and middle initials with a common first name • 3% (1) differing DOB and middle initials with a common first and last name ML correctly rejected the following: • 38% (12) differing first name, last name, middle initials and SSNs. All common names and within Levenshtein distance requirement.
E	100	49	ML missed the following: • 29% (29) females with completely different last name, often with different middle initials • 14% (14) apparent first and middle name switches with no similarity • 5% (5) missing or disagreeing SSNs with combined or disagreeing name fields • 2% (2) first name changes from a female name to a common male name with no SEX flag change • 1% (1) Short name with two transposed letters ML correctly rejected the following: • 42% (42) Similar first and last names and SSN disagreements. Apparently different individuals. • 7% (7) SSN matches and a single very uncommon name field match. Apparently different individuals.
F	100	0	FS missed the following: • 46% (46) Different SSNs often missing middle initial and more common names • 28% (28) Differing DOBs • 26% (26) Non-obvious name combinations and misspellings

Table 11: Unique number of links (Links found in X but not Y) adjusted by estimated FDR:

		Y			
		Deterministic	Fuzzy	Fellegi-Sunter	nGram + SVM (ML)
X	Deterministic	▨	0	3	15
	Fuzzy	4,022	▨	1,877	20
	Fellegi-Sunter	51,777	49,629	▨	437
	nGram + SVM (ML)	57,165	53,160	6,286	▨

Common Linkage Gaps:

Next we evaluate performance through the lens of common 'linkage gaps'. For an overview of this premise, please watch the presentation on linkage provided on our website.

Table 12: WHALES 2019-2020 death record demographic breakdown excluding unknowns

Sex Code	n	Proportion
F	28,659	48%
M	31,018	52%
U	21	<1%

Race Summary Category	n	Proportion
White	52,546	88%
Black	1,838	3%
American Indian/Alaska Native (AIAN)	862	1%
Asian	2,456	4%
Native Hawaiian or Pacific Islander (NHPI)	355	<1%
Multiracial	1,641	3%

Ethnicity	n	Proportion
Non-Hispanic	56,107	94%
Hispanic	3,591	6%

Sex gap:

Table 13: Sex Gap

Strategy	Total Number of WHALES-CHARS Links			WHALES Records Linked		
	F	M	U	F	M	U
Deterministic	106,125	104,275	10	20,884	21,199	7
Fuzzy	108,347	106,075	10	21,212	21,475	7
Fellegi-Sunter	133,023	129,116	45	25,223	25,506	9
nGram + SVM (ML)	135,611	131,904	45	25,590	25,898	9

Proportion of WHALES records successfully linked to CHARS by Sex Code in FS and ML models:

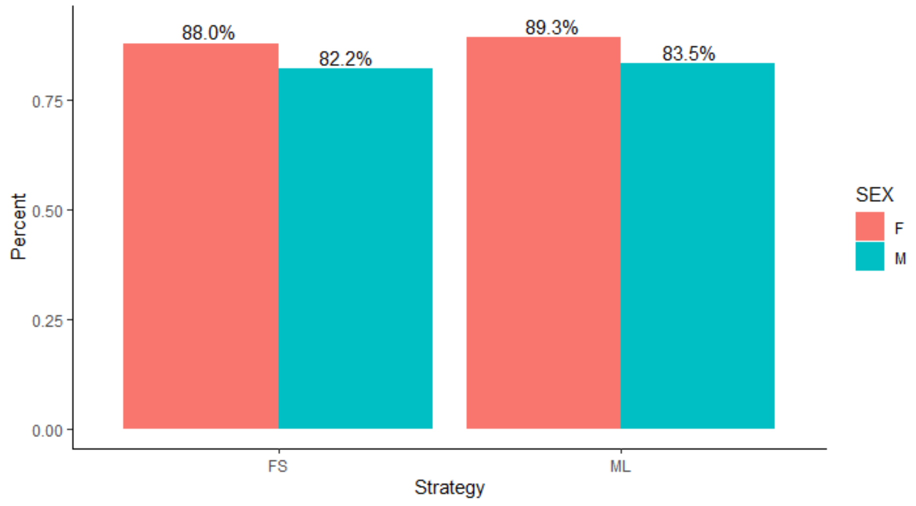


Figure 3 – Sex gap illustration, in 2019-2020 there was no apparent gap.

Typically, we see a sex gap, with women receiving fewer proportional links than male counterparts. In this evaluation, we do not observe such a gap (Figure 3) and will be primarily focused on gaps in race and ethnicity.

Race and ethnicity gap:

For the race and ethnicity evaluation, we will focus entirely on the two best models: FS and ML. First, we will evaluate it much like sex by looking at cumulative links and proportions of WHALES records successfully linked. Then we will zoom in on non-white and Hispanic links captured by the ML model and missed in the FS model, with a proposal on why we may see differences between these two models. Race was derived by joining RACE_SUMMARY and ethnicity used the PERSON_ETHNICITY field. Multiracial individuals and categories of non-Hispanic ethnicities were each combined into one racial and ethnicity group respectively.

Table 14: n Links captured by FS and ML strategies

Strategy	n Links						
	White	Black	AIAN	Asian	NHPI	Multiracial	TOTAL*
FS	230,135	9,919	5,274	9,353	1,842	5,661	262,184
ML	234,714	10,271	5,406	9,518	1,859	5,792	267,560
Δ (%)	+ 1.98%	+ 3.55%	+ 2.50%	+ 1.76%	+ 0.92%	+ 2.31%	+ 2.05%
	WHALES Records Linked						
FS	45,064	1,487	746	2,054	289	1,098	50,738
ML	45,709	1,522	756	2,090	295	1,125	51,497
Δ (%)	+ 1.43%	+ 2.35%	+ 1.34%	+ 1.75%	+ 2.08%	+ 2.46%	+ 1.50%

*Note: TOTAL column includes individuals without race data

Strategy	n Links		
	Non-Hispanic	Hispanic	TOTAL*
FS	249,792	12,392	262,184
ML	254,766	12,794	267,560
Δ (%)	+ 1.99%	+ 3.24%	+ 2.05%
	WHALES Records Linked		
FS	48,334	2,404	50,738
ML	49,015	2,482	51,497
Δ (%)	+ 1.41%	+ 3.24%	+ 1.50%

*Note: TOTAL column includes individuals without ethnicity data

Of the 6,286 links that ML captured, and FS missed, 1,522 (24.2%) were non-white or Hispanic.

ECHIDNA machine learning discussion

It is evident that there is a difference in how these models capture non-white and Hispanic individuals. But why?

To illustrate why, we sampled the first and last names of 200 non-white and Hispanic links:

- 100 individuals from records linked by ML, but not FS, with imperfect name comparisons (e.g., 'Juan Soto' and 'John Soto')
- 50 individuals with perfect name matches (e.g. 'Juan Soto' and 'Juan Soto')
- 50 belonging to different individuals with no similarity requirement (e.g. 'Juan Soto' and 'Julio Rodriguez')

Then, we computed and compared two metrics for a group of name pairs: Bigram Cosine Distance (ML) and Jaro-Winkler Distance (FS). We then plot this in Figure 4 with an example line depicting inclusion criteria for a probabilistic model.

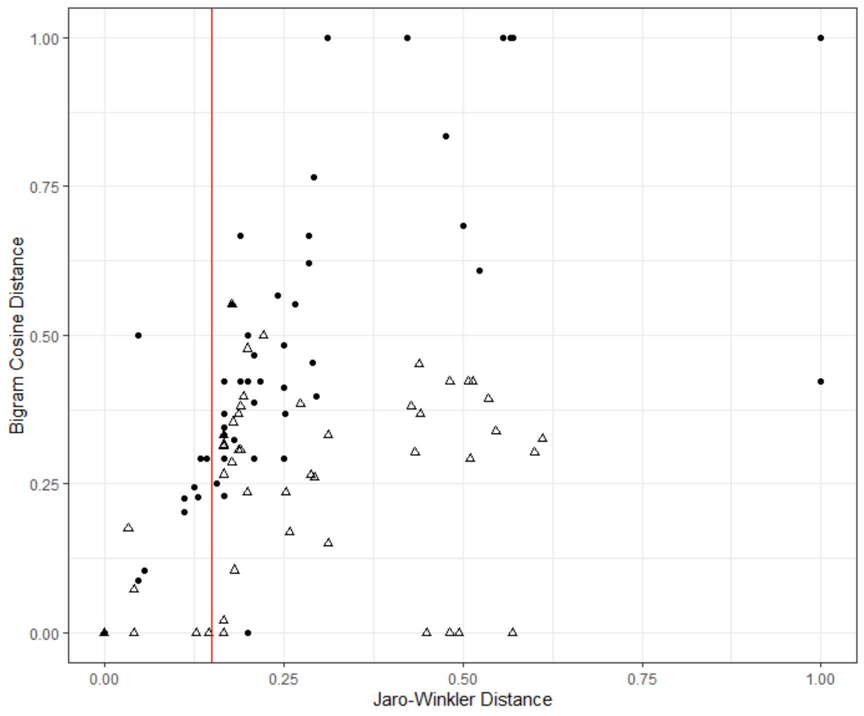


Figure 4 - Black dots are first names. Triangles are last names. The red vertical line shows FS criteria of a distance of .15. A perfect match would be at (0,0) and a completely dissimilar comparison would be at (1,1). In this graph there are 50 overlapping points at (0,0) and (1,1) that are not jittered.

For FS, Jaro-Winkler similarity typically requires a similarity of .85 or above (or a distance of .15 or below) to be considered a match. This is shown by the red line in Figure 4. However, when first and last names are regrouped and compared using this criterion, you get the following binary-squared distribution (1 = match based on JW > .85, 0 = not a match, (x-y) = (first name – last name). This is shown in Figure 5 below:

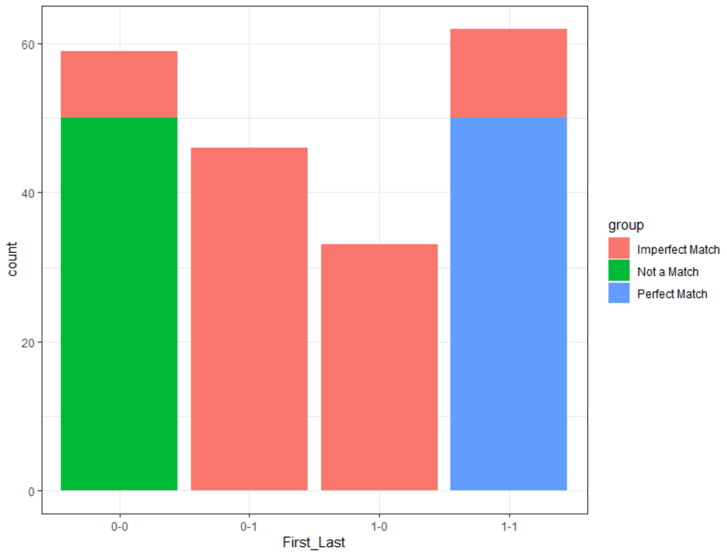


Figure 5 - The four possible categories from two binary decisions (FS)

Then you can calculate the weighted score displayed in Figure 6:

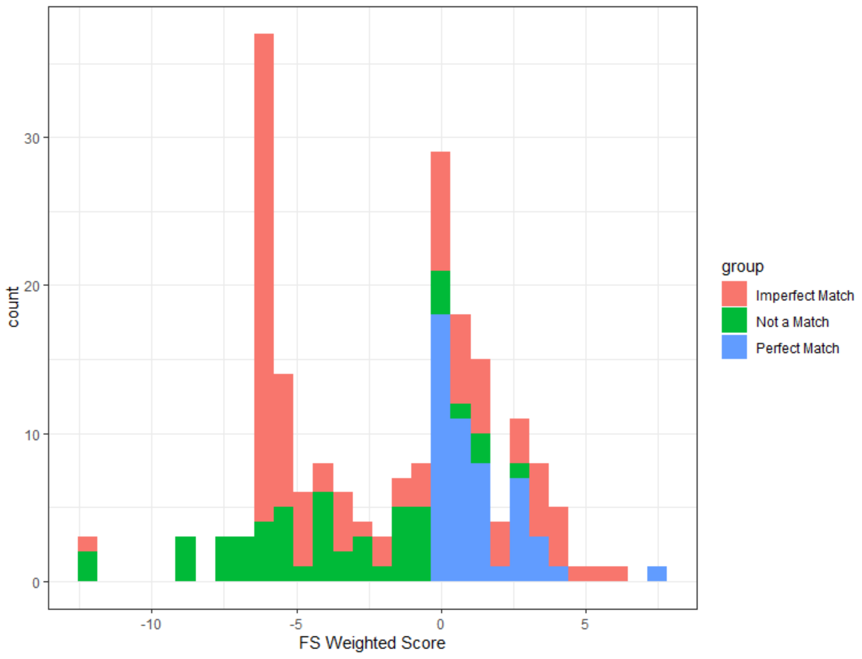


Figure 6 - The same as Figure 5 but with the FS calculated weights

The full FS model uses other data to further separate these distributions. But by using only first and last names, the "imperfect match" group creates a bimodal distribution that corresponds with both the "not a match" and the "perfect match" group. This significant overlap makes it difficult to categorize these names without other variables.

The ML strategy, on the other hand, provides ample separation in hyperspace. Below we use four input metrics: first name bigram cosine distance, first name trigram cosine distance, last name bigram cosine distance, and last name trigram cosine distance. Figure 7 reduced these distances to two dimensions (using t-stochastic neighbor embedding (TSNE)) to visually demonstrate how clear-cut categorization would be for the support vector machine in hyperspace.

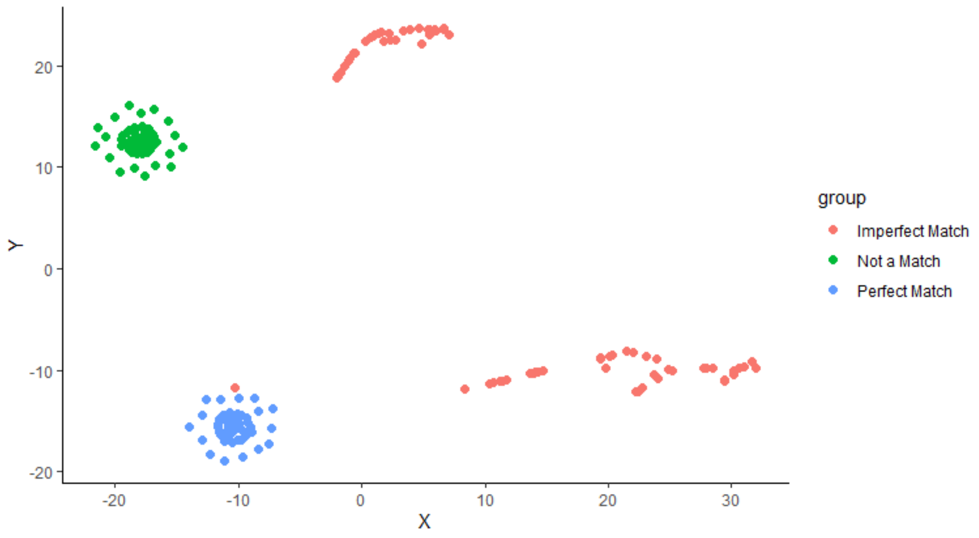


Figure 7 - TSNE representation of 4 cosine distance calculations (used in ML)

The ML model has a clear advantage when inputs are few, as demonstrated by these non-white or Hispanic names. It removes the decision threshold, enabling the model to factor in name similarity at the most granular level. For example, a common instance in missed Hispanic records are multiple surnames that are not split. Both algorithms use a space or a dash to separate names; however, if the record is inputted as SURNAME1SURNAME2, as it frequently is, these are not separated. FS would likely reject the comparison if only one surname was compared to the combined instance of two. In contrast, ML would pick up and incorporate the non-zero cosine distance. This illustrates one mechanism by which the ML strategy captures a higher proportion of minority race/ethnicity record pairs.

Inexact DOBs

More than a quarter of records found by the ML model and missed by FS methodology were due to inexact DOBs. Here we quickly investigate inexact DOBs and **urge anyone doing linkages to not block on exact DOBs.** Below is a summary and detailed breakdown (Figure 8) of inexact DOB linkages identified by the ML model:

Table 15: Proportion of inexact DOBs in all links and minority race and ethnicity links

Description	Proportion Inexact DOBs
All links	0.72 %
Minority Race/Ethnicity Links	1.05 %

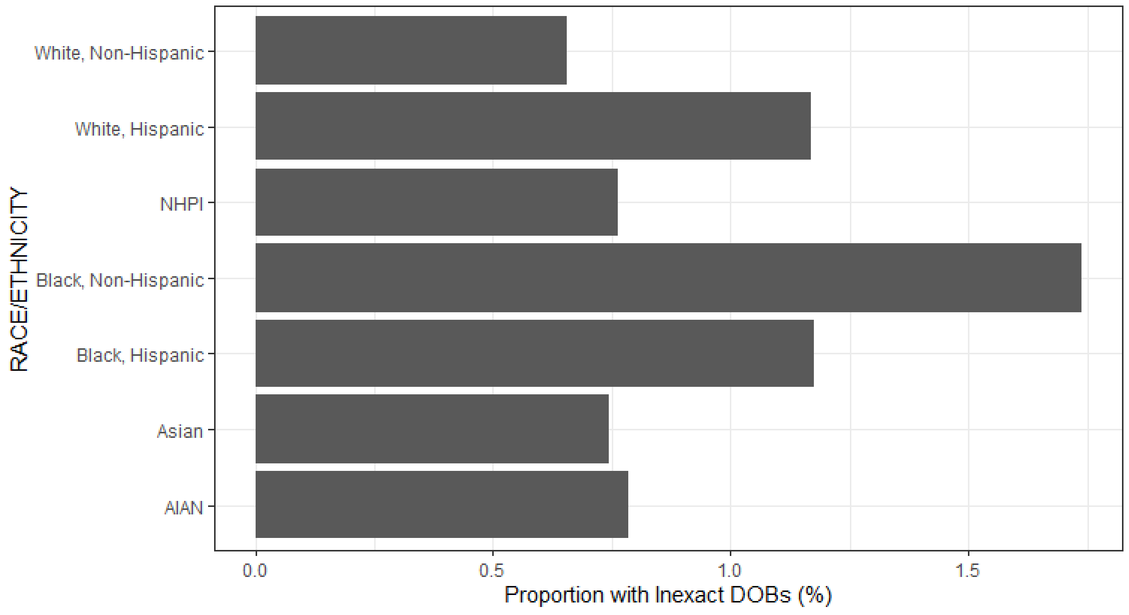


Figure 8- Proportion of links containing inexact DOB comparisons captured by the ML model, by race/eth.

FS and ML comparison summary

Table 16: Head-to-head summary

	n Links	WHALES Capture	Accuracy (type 1)	Gender Gap	Racial Gap
FS	Loser	Loser	Loser	Tie	Loser
ML	Winner	Winner	Winner	Tie	Winner

Additional benefits of ML over FS observed in and beyond ECHINDA:

- ML offers maximal flexibility. For example, from this evaluation we found a cohort of SEX disagreements that were being missed by the model. This will easily be supplemented for future iterations.
- ML is iterative and ever evolving. The model can always be improved, while FS algorithm can only refine weights.
- ML produces better results in more diverse datasets with higher incidence of errors. It's capable at handling old data with lots of errors.
- ML performs better with fewer data inputs.
- ML requires less post-development and computation QA.
- ML will work for any project, as long as you are willing to put in the time training.

There are clear benefits to ML over FS, but there are drawbacks too.

- ML requires heavy development time with complicated programming.
- ML is less generalizable with sometimes specific training sets (project dependent training sets may not be used for other linkage projects).
- ML requires high computational and processing time (hours rather than minutes).
- ML offers no out of the box package.
- Less literature support is available, and it is implemented far less frequently in practice.
- ML can be biased. The ML model is trained to be an extension of the trainer; therefore, it is vulnerable to the trainer's biases.

Table 17: When to use each model

Fellegi – Sunter	nGram + SVM
Time is short	The number of comparison fields is low
Type I errors are affordable	The population is diverse
Programming skills are developing	Experienced programmer
Data is of high quality	Data quality is variable
The number of comparison fields is high	Ample time
Uniform population demographics	Require little to no type I errors

Conclusion and brief discussion

We compared the performance of deterministic, fuzzy, Fellegi-Sunter (FS) and machine learning (ML) models, with particular attention to FS and ML models. The ML model found more total links, linked more unique WHALES death records, and was more accurate in those link designations than any other model. Furthermore, the ML model found more minority race and ethnicity links compared to FS, and captured links that were closely proportionate to the WHALES death records table's race and ethnicity makeup. This is critical to capturing elusive links in racial and ethnic minority groups that could drastically improve downstream public heath analyses.

We used 2019-2020 data because the data quality was high and many of the critical input fields (like the availability of SSN) are well kept and present. For this evaluation, we used the year with the highest quality data. In general, the FS model benefits from high data quality, many comparative data fields, and general lack of diversity, providing the most challenging comparison for ML model implementation. As we observed in the WHALES death to birth linkage, the further back in time the data stretches, the lower the data quality is and the less input fields you have available to work with. This is where the ML model truly shines. It is our aspiration that a similar report spanning multiple years will be developed in the future to quantify the entire impact of implementing this novel strategy.

The current ML model is not perfect and is in ongoing development. However, we believe that this report serves as the "tip of the iceberg" in the nGram + SVM ML model evaluation, and we believe that the potential impacts of utilization would go far beyond the estimated benefits quantified in this evaluation. Nonetheless, this evaluation has indicated that this method could be critical in obtaining more accurate and bountiful linkages that resemble the population of interest, even in very diverse communities with sparse and inaccurate data.

If you made it this far, pat yourself on the back!

Appendix A – Complete SVM input calculation long-form algorithm

```
no_cores = (detectCores() / 2) - 2
cl = makeCluster(no_cores, type = "SOCK")
registerDoParallel(cl)

x9 = foreach(
  i = 1:length(CHUCK_SPLIT.1),
  .combine = rbind,
  .packages = c('tidyverse',
                'stringdist',
                'data.table',
                'e1071')
) %dopar% {
  pool = data.table(CHUCK_SPLIT.1[[i]])
  pool[, 'DOB_HAM' := stringdist(DOB.x,
                                 DOB.y,
                                 method = c('hamming')), linkId]

  pool[, 'FIRSTNAME_COS_1a1' := stringdist(
    FIRST_NAME_1,
    FIRST_NAME,
    method = c('cosine'),
    q = ifelse(
      nchar(FIRST_NAME) < 3 |
        nchar(FIRST_NAME_1) < 3,
      min(nchar(FIRST_NAME), nchar(FIRST_NAME_1)),
      3
    )
  ), linkId]

  pool[, 'FIRSTNAME_COS_1b1' := ifelse(
    nchar(FIRST_NAME) < 3 &
      substr(FIRST_NAME, 1, nchar(FIRST_NAME)) != substr(FIRST_NAME_1, 1, nchar(FIRST_NAME)) |
      nchar(FIRST_NAME_1) < 3 &
      substr(FIRST_NAME, 1, nchar(FIRST_NAME)) != substr(FIRST_NAME_1, 1, nchar(FIRST_NAME)),
    1,
    FIRSTNAME_COS_1a1
  ), linkId]

  pool[, 'FIRSTNAME_COS_2a1' := stringdist(
    FIRST_NAME_2,
    FIRST_NAME,
    method = c('cosine'),
    q = ifelse(
      nchar(FIRST_NAME) < 3 |
        nchar(FIRST_NAME_2) < 3,
      min(nchar(FIRST_NAME), nchar(FIRST_NAME_2)),
      3
    )
  ), linkId]

  pool[, 'FIRSTNAME_COS_2b1' := ifelse(
    nchar(FIRST_NAME) < 3 &
      substr(FIRST_NAME, 1, nchar(FIRST_NAME)) != substr(FIRST_NAME_2, 1, nchar(FIRST_NAME)) |
      nchar(FIRST_NAME_2) < 3 &
      substr(FIRST_NAME, 1, nchar(FIRST_NAME)) != substr(FIRST_NAME_2, 1, nchar(FIRST_NAME)),
    1,
    FIRSTNAME_COS_2a1
  ), linkId]
  pool[, 'FIRSTNAME_COS_3a1' := stringdist(
    FIRST_NAME_3,
    FIRST_NAME,
    method = c('cosine'),
    q = ifelse(
      nchar(FIRST_NAME) < 3 |
        nchar(FIRST_NAME_3) < 3,
      min(nchar(FIRST_NAME), nchar(FIRST_NAME_3)),
      3
    )
  ), linkId]

  pool[, 'FIRSTNAME_COS_3b1' := ifelse(
    nchar(FIRST_NAME) < 3 &
      substr(FIRST_NAME, 1, nchar(FIRST_NAME)) != substr(FIRST_NAME_3, 1, nchar(FIRST_NAME)) |
      nchar(FIRST_NAME_3) < 3 &
      substr(FIRST_NAME, 1, nchar(FIRST_NAME)) != substr(FIRST_NAME_3, 1, nchar(FIRST_NAME)),
    1,
    FIRSTNAME_COS_3a1
  ), linkId]

  pool[, 'FIRSTNAME_COSn3' := min(FIRSTNAME_COS_1b1,
                                  FIRSTNAME_COS_2b1,
                                  FIRSTNAME_COS_3b1,
                                  na.rm = T), linkId]

  pool[, 'FIRSTNAME_COSn3' := ifelse(FIRST_NAME == '' |
                                       FIRST_NAME == ' ', 1, FIRSTNAME_COSn3)]

  pool[, 'FIRSTNAME_COS_1a2' := stringdist(
    FIRST_NAME_1,
    FIRST_NAME,
    method = c('cosine'),
    q = ifelse(
      nchar(FIRST_NAME) < 2 |
        nchar(FIRST_NAME_1) < 2,
      min(nchar(FIRST_NAME), nchar(FIRST_NAME_1)),
      2
    )
  ), linkId]

  pool[, 'FIRSTNAME_COS_1b2' := ifelse(
    nchar(FIRST_NAME) < 2 &
      substr(FIRST_NAME, 1, nchar(FIRST_NAME)) != substr(FIRST_NAME_1, 1, nchar(FIRST_NAME)) |
      nchar(FIRST_NAME_1) < 2 &
      substr(FIRST_NAME, 1, nchar(FIRST_NAME)) != substr(FIRST_NAME_1, 1, nchar(FIRST_NAME)),
    1,
    FIRSTNAME_COS_1a2
  ), linkId]

  pool[, 'FIRSTNAME_COS_2a2' := stringdist(
    FIRST_NAME_2,
    FIRST_NAME,
    method = c('cosine'),
    q = ifelse(
      nchar(FIRST_NAME) < 2 |
        nchar(FIRST_NAME_2) < 2,
      min(nchar(FIRST_NAME), nchar(FIRST_NAME_2)),
      2
    )
  ), linkId]

  pool[, 'FIRSTNAME_COS_2b2' := ifelse(
    nchar(FIRST_NAME) < 2 &
      substr(FIRST_NAME, 1, nchar(FIRST_NAME)) != substr(FIRST_NAME_2, 1, nchar(FIRST_NAME)) |
      nchar(FIRST_NAME_2) < 2 &
      substr(FIRST_NAME, 1, nchar(FIRST_NAME)) != substr(FIRST_NAME_2, 1, nchar(FIRST_NAME)),
    1,
    FIRSTNAME_COS_2a2
  ), linkId]

  pool[, 'FIRSTNAME_COS_3a2' := stringdist(
    FIRST_NAME_3,
    FIRST_NAME,
    method = c('cosine'),
    q = ifelse(
      nchar(FIRST_NAME) < 2 |
        nchar(FIRST_NAME_3) < 2,
      min(nchar(FIRST_NAME), nchar(FIRST_NAME_3)),
      2
    )
  ), linkId]

  pool[, 'FIRSTNAME_COS_3b2' := ifelse(
    nchar(FIRST_NAME) < 2 &
      substr(FIRST_NAME, 1, nchar(FIRST_NAME)) != substr(FIRST_NAME_3, 1, nchar(FIRST_NAME)) |
      nchar(FIRST_NAME_3) < 2 &
      substr(FIRST_NAME, 1, nchar(FIRST_NAME)) != substr(FIRST_NAME_3, 1, nchar(FIRST_NAME)),
    1,
    FIRSTNAME_COS_3a2
  ), linkId]

  pool[, 'FIRSTNAME_COSn2' := min(FIRSTNAME_COS_1b2,
                                  FIRSTNAME_COS_2b2,
                                  FIRSTNAME_COS_3b2,
                                  na.rm = T), linkId]

  pool[, 'FIRSTNAME_COSn2' := ifelse(FIRST_NAME == '' |
                                       FIRST_NAME == ' ', 1, FIRSTNAME_COSn2)]

  pool[, 'MINAME_Disagree' := ifelse(
    MIDDLE_INIT.x == MIDDLE_INIT.y |
      is.na(MIDDLE_INIT.x) |
      is.na(MIDDLE_INIT.y) |
      MIDDLE_INIT.x == '' |
      MIDDLE_INIT.y == '' |
      MIDDLE_INIT.x == ' ' |
      MIDDLE_INIT.y == ' ',
    0,
    1
  ), linkId]

  pool[, 'MINAME_Missing' := ifelse(MIDDLE_INIT.x == '' |
                                      MIDDLE_INIT.y == '' |
                                      MIDDLE_INIT.x == ' ' |
                                      MIDDLE_INIT.y == ' ',
                                    1,
                                    0), linkId]

  pool[, 'LASTNAME_COS_1a1' := stringdist(
    LAST_NAME_1,
    LAST_NAME,
    method = c('cosine'),
    q = ifelse(
      nchar(LAST_NAME) < 3 |
        nchar(LAST_NAME_1) < 3,
      min(nchar(LAST_NAME), nchar(LAST_NAME_1)),
      3
    )
  ), linkId]

  pool[, 'LASTNAME_COS_1b1' := ifelse(
    nchar(LAST_NAME) < 3 &
      substr(LAST_NAME, 1, nchar(LAST_NAME)) != substr(LAST_NAME_1, 1, nchar(LAST_NAME)) |
      nchar(LAST_NAME_1) < 3 &
      substr(LAST_NAME, 1, nchar(LAST_NAME)) != substr(LAST_NAME_1, 1, nchar(LAST_NAME)),
    1,
    LASTNAME_COS_1a1
  ), linkId]

  pool[, 'LASTNAME_COS_2a1' := stringdist(
    LAST_NAME_2,
    LAST_NAME,
    method = c('cosine'),
    q = ifelse(
      nchar(LAST_NAME) < 3 |
        nchar(LAST_NAME_2) < 3,
      min(nchar(LAST_NAME), nchar(LAST_NAME_2)),
      3
    )
  ), linkId]

  pool[, 'LASTNAME_COS_2b1' := ifelse(
    nchar(LAST_NAME) < 3 &
      substr(LAST_NAME, 1, nchar(LAST_NAME)) != substr(LAST_NAME_2, 1, nchar(LAST_NAME)) |
      nchar(LAST_NAME_2) < 3 &
      substr(LAST_NAME, 1, nchar(LAST_NAME)) != substr(LAST_NAME_2, 1, nchar(LAST_NAME)),
    1,
    LASTNAME_COS_2a1
  ), linkId]

  pool[, 'LASTNAME_COS_3a1' := stringdist(
    LAST_NAME_3,
    LAST_NAME,
    method = c('cosine'),
    q = ifelse(
      nchar(LAST_NAME) < 3 |
        nchar(LAST_NAME_3) < 3,
      min(nchar(LAST_NAME), nchar(LAST_NAME_3)),
      3
    )
  ), linkId]

  pool[, 'LASTNAME_COS_3b1' := ifelse(
    nchar(LAST_NAME) < 3 &
      substr(LAST_NAME, 1, nchar(LAST_NAME)) != substr(LAST_NAME_3, 1, nchar(LAST_NAME)) |
      nchar(LAST_NAME_3) < 3 &
      substr(LAST_NAME, 1, nchar(LAST_NAME)) != substr(LAST_NAME_3, 1, nchar(LAST_NAME)),
    1,
    LASTNAME_COS_3a1
  ), linkId]

  pool[, 'LASTNAME_COS_4a1' := stringdist(
    LAST_NAME_4,
    LAST_NAME,
    method = c('cosine'),
    q = ifelse(
      nchar(LAST_NAME) < 3 |
        nchar(LAST_NAME_4) < 3,
      min(nchar(LAST_NAME), nchar(LAST_NAME_4)),
      3
    )
  ), linkId]

  pool[, 'LASTNAME_COS_4b1' := ifelse(
    nchar(LAST_NAME) < 3 &
      substr(LAST_NAME, 1, nchar(LAST_NAME)) != substr(LAST_NAME_4, 1, nchar(LAST_NAME)) |
      nchar(LAST_NAME_4) < 3 &
      substr(LAST_NAME, 1, nchar(LAST_NAME)) != substr(LAST_NAME_4, 1, nchar(LAST_NAME)),
    1,
    LASTNAME_COS_4a1
  ), linkId]

  pool[, 'LASTNAME_COSn3' := min(LASTNAME_COS_1b1,
                                 LASTNAME_COS_2b1,
                                 LASTNAME_COS_3b1,
                                 LASTNAME_COS_4b1,
                                 na.rm = T), linkId]

  pool[, 'LASTNAME_COSn3' := ifelse(LAST_NAME == '' |
                                      LAST_NAME == ' ', 1, LASTNAME_COSn3)]

  pool[, 'LASTNAME_COS_1a2' := stringdist(
    LAST_NAME_1,
    LAST_NAME,
    method = c('cosine'),
    q = ifelse(
      nchar(LAST_NAME) < 2 |
        nchar(LAST_NAME_1) < 2,
      min(nchar(LAST_NAME), nchar(LAST_NAME_1)),
      2
    )
  ), linkId]

  pool[, 'LASTNAME_COS_1b2' := ifelse(
    nchar(LAST_NAME) < 2 &
      substr(LAST_NAME, 1, nchar(LAST_NAME)) != substr(LAST_NAME_1, 1, nchar(LAST_NAME)) |
      nchar(LAST_NAME_1) < 2 &
      substr(LAST_NAME, 1, nchar(LAST_NAME)) != substr(LAST_NAME_1, 1, nchar(LAST_NAME)),
    1,
    LASTNAME_COS_1a2
  ), linkId]

  pool[, 'LASTNAME_COS_2a2' := stringdist(
    LAST_NAME_2,
    LAST_NAME,
    method = c('cosine'),
    q = ifelse(
      nchar(LAST_NAME) < 2 |
        nchar(LAST_NAME_2) < 2,
      min(nchar(LAST_NAME), nchar(LAST_NAME_2)),
      2
    )
  ), linkId]

  pool[, 'LASTNAME_COS_2b2' := ifelse(
    nchar(LAST_NAME) < 2 &
      substr(LAST_NAME, 1, nchar(LAST_NAME)) != substr(LAST_NAME_2, 1, nchar(LAST_NAME)) |
      nchar(LAST_NAME_2) < 2 &
      substr(LAST_NAME, 1, nchar(LAST_NAME)) != substr(LAST_NAME_2, 1, nchar(LAST_NAME)),
    1,
    LASTNAME_COS_2a2
  ), linkId]

  pool[, 'LASTNAME_COS_3a2' := stringdist(
    LAST_NAME_3,
    LAST_NAME,
    method = c('cosine'),
    q = ifelse(
      nchar(LAST_NAME) < 2 |
        nchar(LAST_NAME_3) < 2,
      min(nchar(LAST_NAME), nchar(LAST_NAME_3)),
      2
    )
  ), linkId]

  pool[, 'LASTNAME_COS_3b2' := ifelse(
    nchar(LAST_NAME) < 2 &
      substr(LAST_NAME, 1, nchar(LAST_NAME)) != substr(LAST_NAME_3, 1, nchar(LAST_NAME)) |
      nchar(LAST_NAME_3) < 2 &
      substr(LAST_NAME, 1, nchar(LAST_NAME)) != substr(LAST_NAME_3, 1, nchar(LAST_NAME)),
    1,
    LASTNAME_COS_3a2
  ), linkId]

  pool[, 'LASTNAME_COS_4a2' := stringdist(
    LAST_NAME_4,
    LAST_NAME,
    method = c('cosine'),
    q = ifelse(
      nchar(LAST_NAME) < 2 |
        nchar(LAST_NAME_4) < 2,
      min(nchar(LAST_NAME), nchar(LAST_NAME_4)),
      2
    )
  ), linkId]

  pool[, 'LASTNAME_COS_4b2' := ifelse(
    nchar(LAST_NAME) < 2 &
      substr(LAST_NAME, 1, nchar(LAST_NAME)) != substr(LAST_NAME_4, 1, nchar(LAST_NAME)) |
      nchar(LAST_NAME_4) < 2 &
      substr(LAST_NAME, 1, nchar(LAST_NAME)) != substr(LAST_NAME_4, 1, nchar(LAST_NAME)),
    1,
    LASTNAME_COS_4a2
  ), linkId]

  pool[, 'LASTNAME_COSn2' := min(LASTNAME_COS_1b2,
                                 LASTNAME_COS_2b2,
                                 LASTNAME_COS_3b2,
                                 LASTNAME_COS_4b2,
                                 na.rm = T), linkId]

  pool[, 'LASTNAME_COSn2' := ifelse(LAST_NAME == '' |
                                      LAST_NAME == ' ', 1, LASTNAME_COSn2)]

  pool[, 'SSN_HAM' := stringdist(SSN.x,
                                 SSN.y,
                                 method = c('hamming')) / 4, linkId]

  pool[, 'SSN_Missing' := ifelse(SSN.x == '0000' |
                                   SSN.y == '0000', 1, 0), linkId]

  pool[, 'SSN_HAM' := ifelse(SSN_Missing == 1, 1, SSN_HAM)]

  pool[, 'SEX_Disagree' := ifelse(
    SEX.x == SEX.y |
      is.na(SEX.x) |
      is.na(SEX.y) |
      SEX.x == '' |
      SEX.y == '' |
      SEX.x == ' ' |
      SEX.y == ' ',
    0,
    1
  ), linkId]

  pool[, 'SEX_Missing' := ifelse(SEX.x == '' | SEX.y == '' |
                                   SEX.x == ' ' |
                                   SEX.y == ' ' , 1 , 0), linkId]

  pool[, 'isFemale' := ifelse(SEX_Disagree == 0 &
                                SEX.x != 'M', 1, 0), linkId]


  pool = pool[, lapply(.SD, function(x) {
    ifelse(is.na(x), 1, x)
  })]

  pool = pool[, lapply(.SD, function(x) {
    ifelse(is.infinite(x), 1, x)
  })]

  pool[, 'SORT_HELP' := DOB_HAM + SSN_HAM + FIRSTNAME_COSn3 + FIRSTNAME_COSn2 +
         LASTNAME_COSn3 + LASTNAME_COSn2 + SEX_Disagree + MINAME_Disagree, linkId]
  pool2 = pool[, .(
    linkId,
    DischargeRecordID,
    DEATH_SFN_NUM,
    DOD,
    DOB.x,
    DOB.y,
    DOB_HAM,
    SSN.x,
    SSN.y,
    SSN_HAM,
    SSN_Missing,
    SEX.x,
    SEX.y,
    SEX_Disagree,
    SEX_Missing,
    FIRST_NAME,
    FIRST_NAME_1,
    FIRST_NAME_2,
    FIRST_NAME_3,
    FIRSTNAME_COSn3,
    FIRSTNAME_COSn2,
    LAST_NAME,
    LAST_NAME_1,
    LAST_NAME_2,
    LAST_NAME_3,
    LAST_NAME_4,
    LASTNAME_COSn3,
    LASTNAME_COSn2,
    MIDDLE_INIT.x,
    MIDDLE_INIT.y,
    MINAME_Disagree,
    MINAME_Missing,
    isFemale,
    freqFirst,
    freqLast,
    SORT_HELP
  )]

  return(pool2)
}
stopCluster(cl)
```