String Distance Metrics

Caleb Woo

2024-10-22

Motivation and Goals

In this article, we will explore various string distance metrics and conduct an experiment comparing the efficacy of some popular metrics on the task of matching entity names. We will briefly summarize some metrics that Cohen, Ravikumar, and Fienberg discussed in their paper. Inspired by these authors, we will conduct a similar experiment comparing how well different string distance metrics perform on matching entity names in an example data set.

Cohen, Ravikumar, and Fienberg

Cohen, Ravikumar, and Fienberg published a paper in 2003 called *A Comparison of String Distance Metrics for Name-Matching Tasks*. At the time, they recently implemented an open-source, Java toolkit with a variety of string distance metrics for numerous name-matching methods. They compared how the different methods in their toolkit performed on the task of matching entity names across numerous different data sets. Academic communities such as statistics, databases, and artificial intelligence have proposed different methods for the common problem of matching entity names throughout the years. Cohen, Ravikumar, and Fienberg conducted experiments to provide direct and intuitive comparisons of some of the most popular string distance metrics among the numerous communities.

String Distance Metrics

String distance metrics refer to both distance functions and similarity functions. Although they encode analogous information, their interpretations are reversed. For a given pair of strings that are very similar, they would be assigned a small distance value and a large similarity value. Correspondingly, for very different strings, they would be assigned a large distance value and a small similarity value.

Below, we will explore three broad categories of metrics: edit-distance, token-based, and hybrid. First, we will summarize some popular examples of metrics in these categories that Cohen, Ravikumar, and Fienberg discussed in their paper. Afterwards, we will conduct our own experiment comparing how some of these metrics perform on the task of matching entity names in an example data set.

Edit-distance

Edit-distance metrics are calculated as the "cost" of the best order of edits to one string to make it identical to the other string. Edits can include character insertion, deletion, and substitution where each is assigned a particular cost. The **Levenstein distance** is a simple example which assigns a unit cost to all edits. The **Monge-Elkan** distance is a more complex example which assigns a particular cost $\in [0, 1]$ for each edit where insertions and deletions are assigned relatively lower costs.

Although not an edit-distance metric, the **Jaro** metric takes a broadly similar approach by calculating a score based on the count and sequence of the common characters for a pair of strings. The **Jaro-Winkler** metric is an extension which incorporates the size of the longest common prefix for a pair of strings.

Token-based

Token-based metrics treat each string as a multiset of characters or words called tokens. The **Jaccard similarity** is a simple example which is calculated as the size of the intersection of a pair of strings divided by the size of the union. The **cosine similarity** is another popular example.

Hybrid

Hybrid metrics calculate scores by combining multiple existing string distance metrics. Monge and Elkan's **level two** metric calculates a linear combination of a chosen metric calculated on the sub strings for a pair of strings. Cohen, Ravikumar, and Fienberg experimented with using the Monge-Elkan distance, Jaro metric, and Jaro-Winkler metric in their implementations. **"Soft" cosine similarity** includes similar tokens instead of only identical tokens in the intersection of a pair of strings and combines a secondary distance. Cohen, Ravikumar, and Fienberg used the Jaro-Winkler metric as a secondary distance in their implementation. They also developed their own hybrid metric by training a **binary SVM classifier** with input features being the scores from various existing metrics including the Monge-Elkan distance, the Jaro-Winkler metric, and the cosine similarity.

Experiment

In our experiment below, we will explore 4 of the metrics discussed above using implementations from the `stringdist` package in R. We will compare how these metrics perform on the task of matching entity names in an example data set from the `RecordLinkage` package in R.

The 2 edit-distance metrics we will explore are the Levenstein distance and the Jaro-Winkler metric. The 2 token-based metrics we will explore are the Jaccard similarity and the cosine similarity.

Data Cleaning

We start by combining the corresponding ids with an example data set of 500 names

```
# package for metrics
library(stringdist)
# package for example data set
library(RecordLinkage)
# package for data wrangling
library(tidyverse)
# package for precision and recall
library(yardstick)
# package for HTML table
library(kableExtra)

# load example data set
data("RLdata500")
# combine data and ids
rl_data = cbind(RLdata500, id = identity.RLdata500)

# remove for memory
rm(RLdata500)
rm(identity.RLdata500)

head(rl_data)
```

```
##   fname_c1 fname_c2 lname_c1 lname_c2   by bm bd  id
## 1  CARSTEN     <NA>    MEIER     <NA> 1949  7 22  34
## 2     GERD     <NA>    BAUER     <NA> 1968  7 27  51
## 3   ROBERT     <NA> HARTMANN     <NA> 1930  4 30 115
## 4   STEFAN     <NA>    WOLFF     <NA> 1957  9  2 189
## 5     RALF     <NA>  KRUEGER     <NA> 1966  1 13  72
## 6  JUERGEN     <NA>   FRANKE     <NA> 1929  7  4 142
```

Since each row represents a particular person, we want to combine names and birthday information to compare all information for a given person within a single string. We start by adding a hyphen between both first names, a hyphen between both last names, and hyphens in the birthday information to make it month-day-year format.

```
rl_data = rl_data %>%
  mutate(
    # hyphenated first name
    fname = case_when(
      # if second first name exists, hyphenate
      !is.na(fname_c2) ~ paste0(fname_c1, "-", fname_c2),
      # no hyphen o/w
      .default = fname_c1
    ),
    # hyphenated last name
    lname = case_when(
      # if second last name exists, hyphenate
      !is.na(lname_c2) ~ paste0(lname_c1, "-", lname_c2),
      # no hyphen o/w
      .default = lname_c1
    ),
    # hyphenated birthday
    bday = paste0(bm, "-", bd, "-", by)
    )

# hyphenated examples
rl_data[56:60,] %>%
  select(fname, lname, bday)
```

```
##             fname      lname      bday
## 56 PETER-GUENTHER ZIMMERMANN 2-10-1996
## 57         MARTIN   HERRMANN 8-13-1944
## 58          FRANK    MUELLDR 5-20-1978
## 59          BERND JUNG-KLEIN 1-14-1935
## 60       THORSTEN      FUCHS 9-27-1952
```

The final step is to combine all information into a single string and add the corresponding ids. By doing so, we can compare each person's string with every other string, predict a name match based on a string distance metric, and confirm a true name match using the ids to evaluate predictions across different metrics.

```
df = as.data.frame(cbind(
  # add a space between first name, last name, and birthday
  ent = paste0(rl_data$fname, " ", rl_data$lname, " ", rl_data$bday),
  # corresponding ids
  id = rl_data$id
```

```
  ))

head(df)
```

```
##                          ent  id
## 1   CARSTEN MEIER 7-22-1949  34
## 2      GERD BAUER 7-27-1968  51
## 3 ROBERT HARTMANN 4-30-1930 115
## 4     STEFAN WOLFF 9-2-1957 189
## 5    RALF KRUEGER 1-13-1966  72
## 6   JUERGEN FRANKE 7-4-1929 142
```

Name Matching

With our reformatted data in hand, we can now proceed with our experiment. Our experiment will require comparing each of the names with every name to predict name matches. There are 500 total names so to avoid running such a computationally expensive task multiple times, we will run our experiment and save our results in RDS files. Below is the path for the directory that will store the results and the paths for the results files.

```
# results directory name
dir_path = file.path(getwd(), "experiment_results")
# responses (binary true name match) file name
responses_path = file.path(dir_path, "responses.rds")
# scores (metric predictions) file name
scores_path = file.path(dir_path, "scores.rds")
```

Now we will run our experiment below and save our results

```
# list storing predictions from each metric
scores = list(
  lv = c(),
  jw = c(),
  jaccard = c(),
  cosine = c()
)
# vector storing name match responses
responses = c()

# total number of names
n = nrow(df)

# for each name
for (i in 1:n) {
  # for each name
  for (j in 1:n) {
    # 1 = true match, 0 = not a match
    response = as.integer(df[i, "id"] == df[j, "id"])
    responses = c(responses, response)

    # maximum number of chars between names being compared
    max_char = max(nchar(df[i, "ent"]), nchar(df[j, "ent"]))
```

```
    # for each metric
    for (metric in names(scores)) {
      # calculate metric on names being compared
      score = stringdist(df[i, "ent"], df[j, "ent"], method = metric)

      # levenstein distance divided by max_char to convert to 0-1 range
      if (metric == "lv") {
        score = score / max_char
      }

      # low metric values suggest high similarity
      # 1 - metric to make high scores correspond to high similarity
      score = 1 - score

      scores[[metric]] = c(scores[[metric]], score)
    }
  }
}

# create results directory if it does not exist
dir.create(dir_path)

# save results files
saveRDS(responses, responses_path)
saveRDS(scores, scores_path)
```

Results

With our results in hand, we can plot precision-recall curves and calculate areas under those curves to compare performance across the 4 metrics. First, we load our results from the files and combine them into a single dataframe.

```
# load results from files
responses = readRDS(responses_path)
scores = readRDS(scores_path)

# results dataframe
results = data.frame(
  # make responses a factor for pr_curve function
  y = factor(responses, levels = c(1, 0)),

  # predictions from each metric
  lv = scores$lv,
  jw = scores$jw,
  jaccard = scores$jaccard,
  cosine = scores$cosine
)

head(results)
```

```
##   y        lv        jw   jaccard    cosine
## 1 1 1.0000000 1.0000000 1.0000000 1.0000000
```

```
## 2 0 0.4782609 0.6999443 0.4090909 0.7601170
## 3 0 0.2400000 0.5919462 0.5238095 0.6683739
## 4 0 0.3913043 0.6870255 0.5238095 0.6411189
## 5 0 0.3913043 0.5533597 0.3043478 0.6424161
## 6 0 0.3913043 0.7672634 0.5238095 0.8461538
```

Next, we gather the PR curve dataframes for each metric into a list to plot all curves in a single figure. We also gather the PR AUCs for each metric into another list.

```
# list storing each PR curve df
curve_list = list()
# list storing each PR AUC
auc_list = list()

# for each metric
for (i in 1:length(names(scores))) {
  # metric name
  metric = names(scores)[i]
  # get metric PR curve df
  metric_curve = as.data.frame(pr_curve(results, "y", metric))

  # add Metric column for metric name
  metric_curve = metric_curve %>%
    mutate(
      "Metric" = rep(metric, nrow(metric_curve))
    )

  # store PR curve df
  curve_list[[i]] = metric_curve
  # store PR AUC
  auc_list[[metric]] = pr_auc(results, "y", metric) %>% pull(.estimate)
}
```

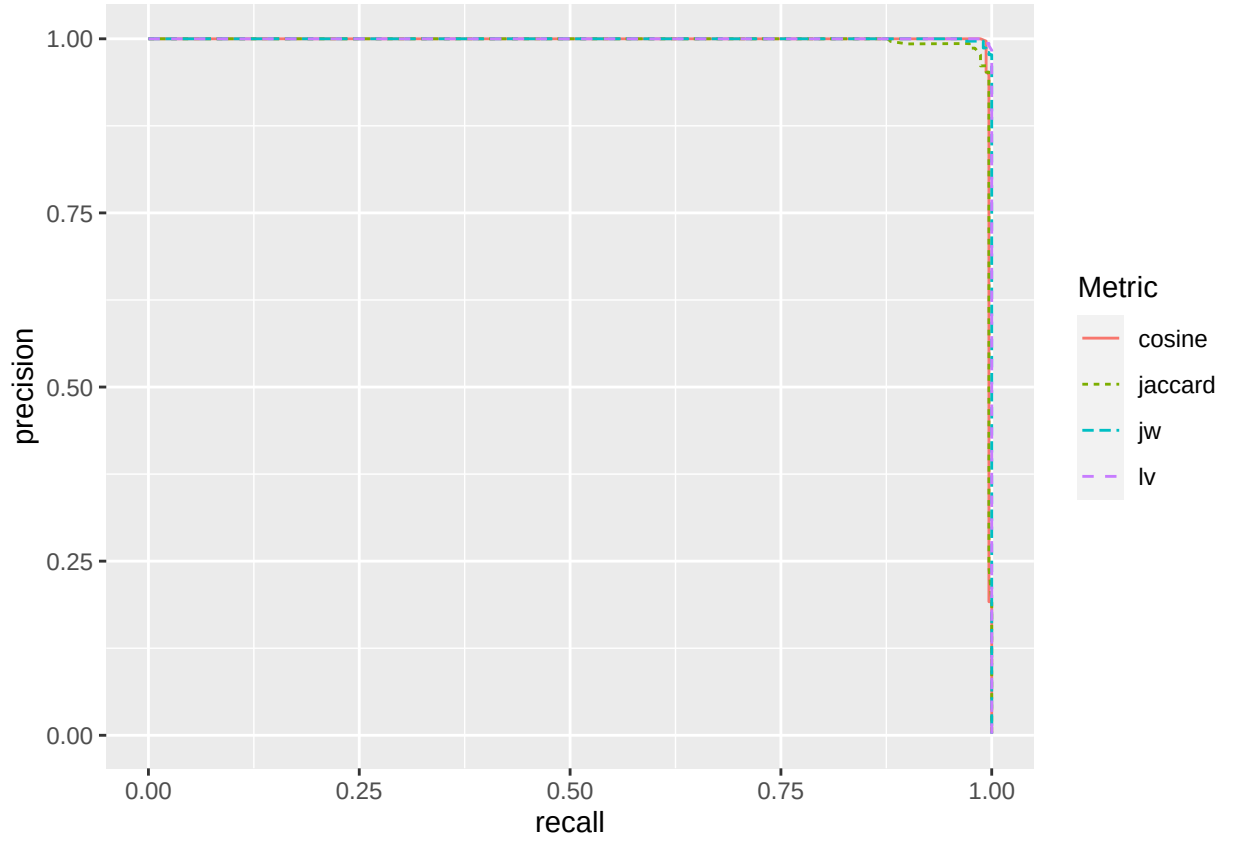
We can use `dplyr::bind_rows` to concatenate each dataframe in `curve_list` into a single dataframe `curve_df`. Now `curve_df` below contains the PR curve dataframes for each metric with the additional `Metric` column indicating which metric each row is associated with

```
curve_df = bind_rows(curve_list)
head(curve_df)
```

```
##   .threshold    recall precision Metric
## 1        Inf 0.0000000         1     lv
## 2  1.0000000 0.8333333         1     lv
## 3  0.9677419 0.8400000         1     lv
## 4  0.9655172 0.8433333         1     lv
## 5  0.9629630 0.8566667         1     lv
## 6  0.9615385 0.8766667         1     lv
```

Using this dataframe, we plot all 4 PR curves in a single plot below where each metric is indicated by a different `color` and `linetype`

```
curve_df %>%
  ggplot(., aes(x=recall, y=precision, color=Metric, linetype=Metric)) +
  geom_line()
```



We can use `auc_list` to view the PR AUCs for each metric

```
# convert list to dataframe
as.data.frame(auc_list) %>%
  # metric and auc columns long format
  pivot_longer(
    cols = names(auc_list),
    names_to = "Metric",
    values_to = "PR AUC"
      ) %>%
  # HTML table
  kbl(digits = 5) %>%
  kable_styling(latex_options = "HOLD_position", full_width = F)
```

Metric	PR AUC
lv	0.99993
jw	0.99973
jaccard	0.99612
cosine	0.99713

The 2 edit-distance metrics (Levenstein distance and Jaro-Winkler metric) have slightly higher AUCs than

the 2 token-based metrics (Jaccard similarity and cosine similarity). However, the differences are minuscule and all AUCs are very close to 1 suggesting that all 4 metrics perform similarly well. The PR curves for each metric are also very similar to each other further suggesting that all 4 metrics perform similarly well.

The primary reason for this is likely due to the extremely low true match rate due to only a small percentage of names in the example data set being duplicates with random errors. As seen below, the true match rate is only 0.24%. Although the PR curves and AUCs suggest good performance for all 4 metrics, the low true match rate suggests that the good performance is rather trivial where each metric mostly predicts non matches.

```
paste0("True match rate = ", mean(responses) * 100, "%")
```

```
## [1] "True match rate = 0.24%"
```

For a more comprehensive experiment in the future, it may help to use the 10,000 name example data set `RLdata10000` instead of the 500 name example data set `RLdata500` that we used above. More data will provide more true match examples to evaluate performance across metrics. We considered experimenting with the 10,000 name example data set but found it too computationally expensive to run locally.

However, only 10% of names in both example data sets are duplicates with random errors so the issue of an overall low true match rate persists. More data will not fix the issue of a low true match rate so the metrics may still appear to perform similarly well. It may help to manually generate more duplicate names with random errors to supplement the example data. By doing so, the true match rate will increase and we may be able to observe some more noticeable differences in performance across the metrics.

Since the example data primarily contains English names, it is also likely that the 4 metrics we explored will perform similarly regardless of modifications to this data. It may be revealing to explore a different metric such as the Monge-Elkan distance which is known to perform empirically well on Spanish names. Although our experiment results here were rather trivial, we hope that our project can serve as a launching point for further explorations and experiments with string distance metrics.