Part 2: Supervised Methods for Entity Resolution

2024-10-20

Why Use a Supervised Method?

Supervised ER relies on a labeled dataset where each record pair is marked as a match or a non-match. This labeled data enables machine learning models to learn patterns and features that distinguish matching pairs from non-matching ones. The primary advantage of this method is its ability to generalize and accurately predict matches in new, unseen data. However, it requires a significant amount of labeled data and careful preparation to ensure the model's effectiveness.

Example Scenario:

Imagine you're part of an analytics team at a company tasked with organizing data before modeling. The data schema is complex and requires strong business and domain knowledge. The analytics engineering team has manually labeled some duplicate records for the 2023 dataset. You plan to use this labeled dataset as a training set to build a model that can deduplicate records for the 2024 data.

Therefore, we are going to show how to implement ER with several supervised ML methods.

```
library(RecordLinkage)
set.seed(111)
data(RLdata10000)   # Used to Generate Training Set
data(RLdata500)     # Used to Generate Evaluation Set
```

We will use two datasets provided by the RecordLinkage package: `RLdata500` and `RLdata10000`. Like previously, `RLdata500` is a smaller dataset used for evaluation, and `RLdata10000` is a larger dataset from which we will generate our training set. We will use `compare.dedup` to generate pairs of records for deduplication. By setting `n_match = 100` and `n_non_match = 400`, we can determine the number of matching pairs as well as the number of non-matching pairs in our training dataset. The `eval_pairs` dataset will be used to assess the performance of our trained ER models.

```
set.seed(111)
# Generate a training set with 100 matches and 400 non-matches from RLData10000
train_pairs = compare.dedup(RLdata10000, identity = identity.RLdata10000,
                            n_match = 100, n_non_match = 400)

# Generate an evaluation set using record pairs from RLData500
eval_pairs = compare.dedup(RLdata500, identity = identity.RLdata500)
```

```
levels(as.factor(train_pairs$pairs$is_match))
```

```
## [1] "0" "1"
```

```
print(table(train_pairs$pairs$is_match))
```

```
##
##   0   1
## 400 100
```

```
class_weights <- c("N" = 100, "L" = 400)
```

Class imbalance occurs when one class significantly outnumbers another in the dataset. As we can tell that in this case, non-matches ("N") vastly outnumber matches ("L"). Class imbalance is quite common in real world problems, and it can lead to several issues if not being treated properly, including biased learning and misleading metrics. Therefore, it is of great importance that we take this seriously. In this case, we will show how to implement class weighting to mitigate the issues.

Training Supervised Models

Tree-based Method: Bagging

Bagging involves training multiple decision trees on different subsets of the data and aggregating their predictions to improve accuracy and control overfitting.

```
model_bagging = trainSupv(train_pairs, method = "bagging",
                          omit.possible = TRUE, weights = c("N" = 100, "L" = 400))
# Or you can write this:
model_bagging = trainSupv(train_pairs, method = "bagging",
                          omit.possible = TRUE, parms = list(prior = c(0.2, 0.8)))
```

We can use `trainSupv` function to train a supervised model.

`train_pairs` is our training dataset.

`method = "bagging"` specifies the bagging algorithm.

`weights = class_weights` applies class weights to handle imbalance.

Alternatively, `parms = list(prior = c(0.2, 0.8))` sets prior probabilities for the classes.

`omit. possible = TRUE`: The omit.possible argument controls whether possible matches should be excluded from the training process. There are three categories of matches: Matches, Non-matches, and Possible matches. When `omit.possible = TRUE`, possible matches, which are pairs where it's uncertain whether they match or not, are excluded from the training data, meaning the model is trained only on clear matches and non-matches.

```
result_bagging = classifySupv(model_bagging, eval_pairs)
```

After training a classifier using `trainSupv()`, the `classifySupv()` function is used to classify new, unlabeled record pairs. It predicts whether the record pairs are matches or non-matches based on the model built with the `trainSupv()` function.

SVM Binary Classifier

The SVM model tries to find a hyperplane that best separates the matches from the non-matches in a high-dimensional space. (Fig 5 from https://dl.acm.org/doi/pdf/10.1145/956750.956759)

```
train_pairs_clean <- train_pairs$pairs[, !(names(train_pairs$pairs) %in% "fname_c2")]
model_svm = trainSupv(train_pairs, method = "svm",
                      omit.possible = TRUE, class.weights = class_weights)
```

```
## Warning in svm.default(x, y, scale = scale, ..., na.action = na.action):
## Variable(s) 'lname_c2' constant. Cannot scale data.
```

Similarily, `method = "svm"` specifies the use of the SVM algorithm. `class.weights = class_weights` applies class weights similarly to the SVM method.

```
result_svm = classifySupv(model_svm, eval_pairs)
```

Other Supervised Classification Methods:

We can also implement decision trees or Adaboost in `trainSupv()`.

```
# Decision trees
model_rpart = trainSupv(train_pairs, method="rpart",
                        omit.possible = TRUE, parms = list(prior = c(0.2, 0.8)))
result_rpart=classifySupv(model_rpart, eval_pairs)

# Adaboost
set.seed(111)
model_ada = trainSupv(train_pairs, method="ada",
                      omit.possible = TRUE, parms = list(prior = c(0.2, 0.8)))
result_ada=classifySupv(model_ada, eval_pairs)
```

Metrics Interpretation:

```
# summary(model_bagging)
# summary(result_svm)
# summary(result_bagging)
summary(result_ada)
```

```
##
## Deduplication Data Set
##
## 500 records
## 124750 record pairs
##
## 50 matches
## 124700 non-matches
## 0 pairs with unknown status
##
```

```
##
## 61 links detected
## 0 possible links detected
## 124689 non-links detected
##
## alpha error: 0.040000
## beta error: 0.000104
## accuracy: 0.999880
##
##
## Classification table:
##
##            classification
## true status      N      P      L
##       FALSE 124687      0     13
##       TRUE       2      0     48
```

We can use `summary()` to check the outcome of our ER model.

1. Deduplication Data Set Information:

- 500 records: The total number of records involved in the deduplication process remains the same.
- 124,750 record pairs: The total number of record pairs generated for comparison.
- 50 matches: Out of the 124,750 record pairs, 50 are true matches.
- 124,700 non-matches: The remaining record pairs are non-matches.
- 0 pairs with unknown status: No pairs have an unknown match status.

2. Linkage Results:

- 61 links detected: The classifier identified 61 record pairs as matches ("links"), even though there are only 50 true matches.
- 0 possible links detected: No pairs are classified as "possible matches". This is expected since we set `omit.possible = TRUE` in the `trainSupv()` function, which excluded possible matches from the training and classification process.
- 124,689 non-links detected: 124,689 record pairs were classified as non-matches, which is slightly fewer than the total number of true non-matches (124, 700), indicating that some nonmatches were misclassified as matches.
- Alpha error: the proportion of false positives (non-matches that were incorrectly classified as matches) relative to the total number of true non-matches.

$$\text{Alpha error } = \frac{\text{False Positives}}{\text{False Positives + True Negatives}}$$

0.040000: The alpha error (false positive rate) is 4%, meaning 4% of nonmatches were incorrectly classified as matches.

- Beta error: the proportion of false negatives (true matches that were incorrectly classified as non-matches) relative to the total number of true matches.

$$\text{Beta error} = \frac{\text{False Negatives}}{\text{False Negatives} + \text{True Positives}}$$

0.000104: The beta error (false negative rate) is extremely low at 0.0104%, meaning only a very small percentage of true matches were incorrectly classified as nonmatches.

- Accuracy: the proportion of correctly classified pairs (both matches and non-matches) relative to the total number of pairs.

$$\text{Accuracy} = \frac{\text{True Positives} + \text{True Negatives}}{\text{Total Pairs}}$$

0.999880: The overall accuracy of the classifier is about 99.99%, indicating that the model correctly classified the vast majority of record pairs.

3. Classification Table:

True Status	**N** (Non-links)	**P** (Possible links)	L (Links)
FALSE	124,687	0	13
TRUE	2	0	48

N (Non-links): The number of pairs classified as non-matches.

P (Possible links): The number of pairs classified as possible matches (which in this case is 0 since no pairs are classified as "possible").

L (Links): The number of pairs classified as matches.

- True matches (TRUE):
- 48 true matches were correctly classified as links (matches).
- 2 true matches were incorrectly classified as non-links (false negatives).
- True non-matches (FALSE):
- 124,687 non-matches were correctly classified as non-links.
- 13 non-matches were incorrectly classified as links (false positives).