Supervised ER using RecordLinkage v2

2024-10-17

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)

data(RLdata500)
data(RLdata10000)
```

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.

```
# 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 = class_weights)
## 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.

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

```
##
## Deduplication Data Set
##
## 500 records
## 124750 record pairs
##
## 50 matches
## 124700 non-matches
## 0 pairs with unknown status
##
##
## 365 links detected
## 0 possible links detected
## 124385 non-links detected
##
## alpha error: 0.020000
## beta error: 0.002534
## accuracy: 0.997459
##
##
## Classification table:
##
##             classification
## true status      N      P      L
##       FALSE 124384      0    316
##       TRUE       1      0     49
```

Then, we can use `classifySupv` function to apply the trained model to new data.

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)
```

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)
```

```
summary(result_svm)
```

```
##
## Deduplication Data Set
##
## 500 records
## 124750 record pairs
##
## 50 matches
## 124700 non-matches
## 0 pairs with unknown status
##
##
## 423 links detected
## 0 possible links detected
## 124327 non-links detected
##
## alpha error: 0.000000
## beta error: 0.002991
## accuracy: 0.997010
##
##
## Classification table:
##
##            classification
## true status      N      P      L
##       FALSE 124327      0    373
##       TRUE       0      0     50
```

Metrics Interpretation:

For Bagging:

- **Alpha Error** (α):

$$\alpha = \frac{1}{49+1} = \frac{1}{50} = 0.02 \quad (2\%)$$

- **Beta Error** (β):

$$\beta = \frac{154}{124,546 + 154} = \frac{154}{124,700} \approx 0.001235 \quad (0.1235\%)$$

- **Accuracy**:

$$\text{Accuracy} = \frac{124,546 + 49}{124,750} = \frac{124,595}{124,750} \approx 0.998758 \quad (99.88\%)$$

For SVM:

- **Alpha Error** (α):

$$\alpha = \frac{1}{49 + 1} = \frac{1}{50} = 0.02 \quad (2\%)$$

- **Beta Error** (β):

$$\beta = \frac{401}{124,299 + 401} = \frac{401}{124,700} \approx 0.003216 \quad (0.3216\%)$$

- **Accuracy**:

$$\text{Accuracy} = \frac{124,299 + 49}{124,750} = \frac{124,348}{124,750} \approx 0.996778 \quad (99.68\%)$$

Both Bagging and SVM miss 2% of the true matches, indicating high sensitivity. Bagging has a lower false positive rate compared to SVM, meaning it introduces fewer false duplicates.