arulesCBA: Classification for Factor and Transactional Data Sets Using Association Rules

Ian Johnson
Southern Methodist University

Abstract

This paper presents an R package, **arulesCBA**, which uses association rules mined with the apriori algorithm from **arules** to build a classifier for discrete or transactional data sets. The package also provides an interface to use an association-rule classifier to predict classes for new data entries. The classification algorithm implemented in **arulesCBA** performs competitively when compared to existing discrete classification algorithms.

Keywords: data mining, classification, association rules, R.

1. Introduction

Association rule mining is a well-established strategy for discovering relationships among attributes of discrete, factor, and transactional data sets. Existing R packages such as **arules** provide interfaces to C implementations of fast association rule mining algorithms such as apriori. For the purpose of this paper, association rule mining will be treated as a black box, as association rules for the classification algorithm in **arulesCBA** will be mined using the apriori interface in **arules**. The rules mined from the algorithm will have five relevant fields. The first three: lift, support, and confidence, are statistical measures of the strength of an association rule which will be discussed in later sections. The final attributes of an association rule are the predicate, or left-hand-side of the rule, and the class, or the right-hand-side of the rule. The predicate of a rule is the set of elements in a row of data which are associated with the resulting class of the rule. The classification algorithm in **arulesCBA** uses a special type of association rule where the right-hand-side is a set of size one (called a class). These rules are called CARs (Class Association Rules). The remainder of the paper will be organized as follows: first, the R interface of the **arulesCBA** will be explored, then the CBA algorithm will be detailed, and then an example of using the **arulesCBA** interface to classify the `Iris` data set from the R package **datasets** will be provided.

2. The arulesCBA Interface

The **arulesCBA** package provides four user-facing functions: `CBA`, `predict`, `rules`, and `print`. The functions `rules` and `print` functions simply return the rules of a CBA classifier object and print the important information about a CBA object, respectively. The `CBA` and `predict` functions are used to build a classifier from an existing data set, and to predict classes for an incoming data set, respectively.

2.1. The CBA Function

The function `CBA` provides an R interface to a C implementation of the CBA algorithm used to generate a classifier. The function accepts two required arguments, and three option arguments. The required arguments, `data` and `class`, must contain the data set for which a classifier will be built, and a character vector with the name of the column of the data set which will be used as the class variable. The argument `data` must be a data.frame or a transaction matrix from the **arules** package, and it must include a column whose name is equal to the argument `class`.

The optional arguments, `support`, `confidence`, and `verbose`, are used to provide parameters to the `apriori` interface from the **arules** package. The argument `support`, set to 0.2 by default, is used to set a minimum support value for the association rules mined from `apriori`. Likewise, the argument `confidence`, set to 0.8 by default, is used to set a minimum confidence value for the association rules mined from `apriori`. Finally, the `verbose` argument, set to `FALSE` by default, can be used to print out diagnostic run-time information from within the `CBA` and `apriori` functions.

The following are an examples of valid calls to the *CBA* function:

```
> classifier <- CBA(formula, data)
> classifier <- CBA(formula data, confidence = 0.95, verbose = TRUE)
```

These are only valid calls to the *CBA* function if the following conditions are met:

- The object `formula` contains a symbolic description of the model to be fitted. The only model type currently supported is of the from `class ~ .`, where `class` is the name of the class variable.
- The object `data` is a data.frame whose columns are all factors, or an **arules** transaction object.
- The column in `date` representing the class is not allowed to have missing values.

The object returned by the CBA function is a CBA object, which is a list with three elements: `rules`, `default`, and `levels`. The object `rules` is an ordered vector of **arules** association rules which are used for classification. The object `default` is a character vector of size 1 which holds the default class for the classifier, which will be discussed in later sections. Finally, `levels` is a vector of all of the possible classes for an object being sent to the classifier.

The `rules` object can be extracted from the `CBA` object using the function `rules(CBA)`. The `default` and `levels` objects are only used internally in the `predict` function.

2.2. The Predict Function

The function `predict` is used to apply a CBA classifier to a new set of data to be classified. The function accepts only two arguments, `object` and `newdata`, which are the CBA object and the new data set to be classified, respectively. The CBA classifier `object` can be sent to `predict` directly from the `CBA` function. The `newdata` object must be a data.frame or **arules** transaction object whose columns match those used to build the original classifier.

The following is an example of a valid call to the `predict` function, where a data.frame `dataset` is split into `training` and `testing` sets which are used to build a CBA classifier, and then use it, respectively:

```
> className <- colnames(data)[1]
> training <- data[1:750,]
> testing <- data[751, 1000]
> classifier <- CBA(training, className)
> classes <- predict(classifier, testing)
```

This is correct use of the *predict* function if all conditions are met for the `CBA` function, where in this example `dataset` is a data.frame. After executing this code, the objects `classes` and `data[,1]` should be similar, or exactly the same if the classifier worked with 100% accuracy. A cursory examination of the success of the prediction can be done using `table(classes)` and `table(data[,1])`. A more in-depth comparison can be done by computing a confusion matrix.

3. The CBA Algorithm

The CBA (Classification Based on Association rules) algorithm used in **arulesCBA** is adapted from *Liu, et al., 1998*. The algorithm is split up into three stages, each of which is implemented in C and interfaced from R through **arulesCBA**. The non-performance-critical and data formatting operations are completed in R, while performance-critical operations take place in C.

3.1. Stage 0

Prior to the three stages of the CBA algorithm, a number of preconditions must be met. Prior to stage 1 of the algorithm, therefore, a stage 0 occurs to establish those preconditions. First, a set of association rules must be generated. This is completed using a call to `apriori` from **arules**. These rules are then sorted primarily based on their *confidence*, and then by their *support* and *lift*. *Confidence* is a measure of how frequently the predicate of an association rule correctly predicts the class of a data entry. For a rule whose predicate predicts the class in every case, the *confidence* value is 1. It is therefore the primary tool for ranking association rules for use in a classifier. Generating association rules, and sorting them as described above, is achieved as follows:

```
> rules <- apriori(ds.mat, ...)
> rules.sorted <- sort(rules, by=c("confidence", "support", "lift"))
```

Note that the call to `apriori` includes a number of additional parameters to guarantee that the mined rules will be useful for the classifier, but those parameters have been redacted for simplicity.

The data input to the classifier must also be formatted as an **arules** transaction object, and two matrices are constructed, `rulesMatchLHS` and `rulesMatchRHS`, which identify which rules from the mined ruleset correspond to the predicate and class of which data entries in the input data set. The matrices are generated as follows:

```
> rulesMatchLHS <- is.subset(lhs(rules.sorted), ds.mat)
> rulesMatchRHS <- is.subset(rhs(rules.sorted), ds.mat)
```

In this version of **arulesCBA**, dense matrices are used for rulesMatchLHS and rulesMatchRHS. Future versions will use sparse matrices for memory efficiency.

A number of other data structures are instantiated and organized for later use, but their purpose, while critical to functionality, is not critical to understanding the algorithm, and they have therefore been omitted from this description of the algorithm. This description of the CBA algorithm is simplified considerably for the sake of clarity. It provides more than enough information to be able to use the **arulesCBA** interface confidently.

3.2. Stage 1

In stage 1 of the CBA algorithm, a linear pass is made through the entire input data set, and a set `A` is built of all falsely classified record. A falsely classified record is one which matches the left-hand-side of a rule in the classifier but whose class doesn't match the right-hand-side of that rule. Each falsely classified record in `A` is stored alongside a corresponding *crule* and *wrule* for the record. A *crule* (correct rule) is a rule which matches an entry on both the left and right-hand-sides of the rule, while a *wrule* (wrong rule) is a rule which matches an entry on the left-hand-side, but not the right-hand-side. Stage 1 of the algorithm also builds a list of *strong rules*, rules which correctly identify entries in the input data set and will therefore be used in the final classifier.

3.3. Stage 2

Stage 2 of the CBA processes the set `A` to find possible replacement rules for the *wrules* which falsely classified records. This stage performs a linear pass through `A`, a subset of the input data set. For each element in `A`, a list of possible replacement rules for the *crule* identified is generated and added to a new set, `replace`, which will be used in stage 3. A possible replacement rule is defined as any rule which correctly classifies the data entry in question.

3.4. Stage 3

In stage 3, the final stage of the CBA algorithm, the set `replace` is processed, and a final classifier is built. A linear pass is made through the set of association rules which have been labeled as *strong rules*. For each rule, all possible replacement rules identified in `replace` are evaluated for possible replacement. If the replacement rules correctly classify a record, they are prioritized over the rule to be replaced. As the set is processed, information is maintained about how many elements from the original data set are correctly and incorrectly classified by the set of already-processed rules. At each rule, the number of falsely classified records is stored in a set `totalErrors`. After every rule has been processed, the classifier is built as the subset of the original rule set up to the index of the minimum number of class errors in the `totalErrors` set. This classifier is then returned with a default class via the R interface in **arulesCBA**.

4. Using arulesCBA

The following is an example of how **arulesCBA** can be used to classify flowers from the `Iris` data set in the **datasets** package. The `Iris` data set is a set of 150 observations of 5 variables. The first 4 variables are continuous measures of petal and sepal length of flowers. The 5th variable is the species of the flower. This will be used as the class for the classification process.

4.1. Installing arulesCBA

Prior to executing the following example, install **arulesCBA** using: `install.packages("arulesCBA")`

To install the most recent development version, use: `> install_github("ianjjohnson/arulesCBA")`.

`install_github` is available through the **devtools** package.

4.2. Loading Required Packages

To load the **arulesCBA** package, as well as the **caret** package, which is used for assessing the results of the classifier, use:

```
> library(arulesCBA)
```

Note that the package **caret** can be installed using `install.packages(caret)`.

4.3. Discretizing the Data

arulesCBA now performs discretization within the CBA and bCBA functions. Custom discretization can be used, but most state-of-the art class-based discretization strategies are already supported. Discretization method can be specified using the disc.method parameter to CBA or bCBA.

```
> data(iris)
```

4.4. Building the Classifier

Building the classifier can be done with a simple call to the `CBA` function:

```
> classifier <- CBA(Species ~ ., iris, supp = 0.05, conf=0.9)
```

Note that the second parameter to the `CBA` function, `class`, is the name of the column of `iris.disc` which contains the class of each entry. Also recall that the `iris.disc` data must be convertible to **arules** transaction data at this point in execution.

Basic information about the classifier can be found using the `print` function:

```
> classifier
```

```
CBA Classifier Object
Class: Species=setosa, Species=versicolor, Species=virginica
Default Class: Species=versicolor
Number of rules: 7
Classification method: first
Description: CBA algorithm by Liu, et al. 1998 with support=0.05 and
     confidence=0.9
```

4.5. Accessing the Rules of the Classifier

The rules as an arules rule list can be retreived using:

```
> rules(classifier)
```

```
set of 7 rules
```

The association rules of the classifier can be read found seen using:

```
> inspect(rules(classifier))
```

```
    lhs                             rhs                    support confidence lift count size
[1] {Petal.Length=[-Inf,2.45)}   => {Species=setosa}        0.33       1.00  3.0    50    2
[2] {Sepal.Length=[6.15, Inf],
     Petal.Width=[1.75, Inf]}    => {Species=virginica}     0.25       1.00  3.0    37    3
[3] {Sepal.Length=[5.55,6.15),
     Petal.Length=[2.45,4.75)}   => {Species=versicolor}    0.14       1.00  3.0    21    3
[4] {Sepal.Width=[-Inf,2.95),
     Petal.Width=[1.75, Inf]}    => {Species=virginica}     0.11       1.00  3.0    17    3
[5] {Sepal.Length=[6.15, Inf],
     Petal.Length=[2.45,4.75)}   => {Species=versicolor}    0.08       1.00  3.0    12    3
[6] {Sepal.Width=[2.95,3.35),
     Petal.Length=[2.45,4.75)}   => {Species=versicolor}    0.08       1.00  3.0    12    3
[7] {Petal.Width=[1.75, Inf]}    => {Species=virginica}     0.30       0.98  2.9    45    2
```

4.6. Using the Classifier

Once the classifier has been built, it can be used to classify the training data set as a cursory test of its accuracy using a simple call to `predict`:

```
> pred <- predict(classifier, iris)
```

```
> head(pred)
```

```
[1] setosa setosa setosa setosa setosa setosa
Levels: setosa versicolor virginica
```

```
> table(pred)
```

```
pred
    setosa versicolor  virginica
        50         54         46
```

This table shows that the **arulesCBA** classifier predicted that there is a 50-48-52 split of the 3 species in the test data.

4.7. Checking the Results

In order to ascertain the quality of the predicted classes, we can create a simple confusion matrix:

```
> table(pred, truth = iris$Species)
```

```
            truth
pred         setosa versicolor virginica
  setosa         50          0         0
  versicolor      0         49         5
  virginica       0          1        45
```

More sophisticated confusion matrices can be created using packages like **caret** or **gmodels**.

Affiliation:

Ian Johnson
Computer Science and Engineering
Southern Methodist University
Dallas, Texas
E-mail: ianjjohnson@icloud.com
URL: http://www.ianjjohnson.com