RTextTools: A Supervised Learning Package for Text Classification

by Timothy P. Jurka, Loren Collingwood, Amber E. Boydstun, Emiliano Grossman, and Wouter van Atteveldt

Abstract Social scientists have long hand-labeled texts to create datasets useful for studying topics from congressional policymaking to media reporting. Many social scientists have begun to incorporate machine learning into their toolkits. **RTextTools** was designed to make machine learning accessible by providing a start-to-finish product in less than 10 steps. After installing **RTextTools**, the initial step is to generate a document term matrix. Second, a container object is created, which holds all the objects needed for further analysis. Third, users can use up to nine algorithms to train their data. Fourth, the data are classified. Fifth, the classification is summarized. Sixth, functions are available for performance evaluation. Seventh, ensemble agreement is conducted. Eighth, users can cross-validate their data. Finally, users write their data to a spreadsheet, allowing for further manual coding if required.

Introduction

The process of hand-labeling texts according to topic, tone, and other quantifiable variables has yielded datasets offering insight into questions that span social science, such as the representation of citizen priorities in national policymaking (Jones et al., 2009) and the effects of media framing on public opinion (Baumgartner et al., 2008). This process of hand-labeling, known in the field as "manual coding", is highly time consuming. To address this challenge, many social scientists have begun to incorporate machine learning into their toolkits. As computer scientists have long known, machine learning has the potential to vastly reduce the amount of time researchers spend manually coding data. Additionally, although machine learning cannot replicate the ability of a human coder to interpret the nuances of a text in context, it does allow researchers to examine systematically both texts and coding scheme performance in a way that humans cannot. Thus, the potential for heightened efficiency and perspective make machine learning an attractive approach for social scientists both with and without programming experience.

Yet despite the rising interest in machine learning and the existence of several packages in R, no package incorporates a start-to-finish product that would be appealing to those social scientists and other researchers without technical expertise in this area. We offer to fill this gap through **RTextTools**. Using a variety of existing R packages, **RTextTools** is designed as a one-stop-shop for conducting supervised learning with textual data. In this paper, we outline a ten-step process, discuss the core functions of the program, and demonstrate the use of **RTextTools** with a working example.

The core philosophy driving **RTextTools**' development is to create a package that is easy to use for individuals with no prior R experience, yet flexible enough for power users to utilize advanced techniques. Overall, **RTextTools** offers a comprehensive approach to text classification, by interfacing with existing text pre-processing routines and machine learning algorithms and by providing new analytics functions. While existing packages can be used to perform text preprocessing and machine learning, respectively, no package combines these processes with analytical functions for evaluating machine learning accuracy. In short, **RTextTools** expedites the text classification process: everything from the installation process to training and classifying has been streamlined to make machine learning with text data more accessible.

General workflow

A user starts by loading his or her data from an Access, CSV, Excel file, or series of text files using the `read_data()` function. We use a small and randomized subset of the Congressional bills database constructed by Wilkerson and Adler for the running example offered here.[1] We choose this truncated dataset in order to minimize the amount of memory used, which can rapidly overwhelm a computer when using large text corpora. Most users therefore should be able to reproduce the example. However, the resulting predictions tend to improve (nonlinearly) with the size of the reference dataset, meaning that our results here are not as good as they would be using the full congressional bills dataset. Computers with at least 4GB of memory should be able to run **RTextTools** on medium to large datasets (i.e., up to 30,000 texts) by using the three low-memory algorithms included: general linearized models (Friedman et al., 2010) from the **glmnet** package, maximum entropy (Jurka, 2012) from **maxent**, and support vector machines (Meyer et al., 2012) from **e1071**. For users with larger datasets, we recommend a cloud computing service such as Amazon EC2.

1. Creating a matrix

First, we load our data with `data(USCongress)`, and use the **tm** package (Feinerer et al., 2008) to create a document-term matrix. The `USCongress` dataset comes with **RTextTools** and hence the function `data` is relevant only to data emanating from R packages. Researchers with data in Access, CSV, Excel, or text files will want to load data via the `read_data()` function. Several pre-processing options from the **tm** package are available at this stage, including stripping whitespace, removing sparse terms, word stemming, and stopword removal for several languages.[2] We want readers to be able to reproduce our example, so we set `removeSparseTerms` to .998, which vastly reduces the size of the document-term matrix, although it may also reduce accuracy in real-world applications. Users should consult Feinerer et al. (2008) to take full advantage of preprocessing possibilities. Finally, note that the text column can be encapsulated in a `cbind()` data frame, which allows the user to perform supervised learning on multiple columns if necessary.

[2] Languages include Danish, Dutch, English, Finnish, French, German, Italian, Norwegian, Portuguese, Russian, Spanish, and Swedish

```
data(USCongress)

# CREATE THE DOCUMENT-TERM MATRIX
doc_matrix <- create_matrix(USCongress\$text,
                            language="english", removeNumbers=TRUE,
                            stemWords=TRUE, removeSparseTerms=.998)
```

2. Creating a container

The matrix is then partitioned into a container, which is essentially a list of objects that will be fed to the machine learning algorithms in the next step. The output is of class `matrix_container` and includes separate train and test sparse matrices, corresponding vectors of train and test codes, and a character vector of term label names. Looking at the example below, `doc_matrix` is the document term matrix created in the previous step, and `USCongress$major` is a vector of document labels taken from the `USCongress` dataset. `trainSize` and `testSize` indicate which documents to put into the training set and test set, respectively. The first 4,000 documents will be used to train the machine learning model, and the last 449

[1] http://www.congressionalbills.org/research.html

documents will be set aside to test the model. In principle, users do not have to store both documents and labels in a single dataset, although it greatly simplifies the process. So long as the documents correspond to the document labels via the `trainSize` and `testSize` parameters, `create_container()` will work properly. Finally, the `virgin` parameter is set to `FALSE` because we are still in the evaluation stage and not yet ready to classify virgin documents.

```
container <- create_container(doc_matrix, USCongress\$major, trainSize=1:4000,
                              testSize=4001:4449, virgin=FALSE)
```

From this point, users pass the container into every subsequent function. An ensemble of up to nine algorithms can be trained and classified.

3. Training models

The `train_model()` function takes each algorithm, one by one, to produce an object passable to `classify_model()`. A convenience `train_models()` function trains all models at once by passing in a vector of model requests.[3] The syntax below demonstrates model creation for all nine algorithms. For expediency, users replicating this analysis may want to use just the three low-memory algorithms: support vector machine (Meyer et al., 2012), glmnet (Friedman et al., 2010), and maximum entropy (Jurka, 2012).[4] The other six algorithms include: scaled linear discriminant analysis (slda) and bagging (Peters and Hothorn, 2012) from **ipred**; boosting (Tuszynski, 2012) from **caTools**; random forest (Liaw and Wiener, 2002) from **randomForest**; neural networks (Venables and Ripley, 2002) from **nnet**; and classification or regression tree (Ripley., 2012) from **tree**. Please see the aforementioned references to find out more about the specifics of these algorithms and the packages from which they originate. Finally, additional arguments can be passed to `train_model()` via the `...` argument.

[3] `classify_models()` is the corollary to `train_models()`.

[4] Training and classification time may take a few hours for all 9 algorithms, compared to a few minutes for the low memory algorithms

```
SVM <- train_model(container,"SVM")
GLMNET <- train_model(container,"GLMNET")
MAXENT <- train_model(container,"MAXENT")
SLDA <- train_model(container,"SLDA")
BOOSTING <- train_model(container,"BOOSTING")
BAGGING <- train_model(container,"BAGGING")
RF <- train_model(container,"RF")
NNET <- train_model(container,"NNET")
TREE <- train_model(container,"TREE")
```

4. Classifying data using trained models

The functions `classify_model()` and `classify_models()` use the same syntax as `train_model()`. Each model created in the previous step is passed on to `classify_model()`, which then returns the classified data.

```
SVM_CLASSIFY <- classify_model(container, SVM)
GLMNET_CLASSIFY <- classify_model(container, GLMNET)
MAXENT_CLASSIFY <- classify_model(container, MAXENT)
SLDA_CLASSIFY <- classify_model(container, SLDA)
BOOSTING_CLASSIFY <- classify_model(container, BOOSTING)
BAGGING_CLASSIFY <- classify_model(container, BAGGING)
RF_CLASSIFY <- classify_model(container, RF)
NNET_CLASSIFY <- classify_model(container, NNET)
TREE_CLASSIFY <- classify_model(container, TREE)
```

5. Analytics

The most crucial step during the machine learning process is interpreting the results, and **RTextTools** provides a function called `create_analytics()` to help users understand the classification of their test set data. The function returns a container with four different summaries: by label (e.g., topic), by algorithm, by document, and an ensemble summary.

Each summary's contents will differ depending on whether the virgin flag was set to `TRUE` or `FALSE` in the `create_container()` function in Step 3. For example, when the virgin flag is set to `FALSE`, indicating that all data in the training and testing sets have corresponding labels, `create_analytics()` will check the results of the learning algorithms against the true value to determine the accuracy of the process. However, if the virgin flag is set to `TRUE`, indicating that the testing set is unclassified data with no known true value, `create_analytics()` will return as much information as possible without comparing each predicted value to its true label.

The label summary provides statistics for each unique label in the classified data (e.g., each topic category). This includes the number of documents that were manually coded with that unique label (`NUM_MANUALLY_CODED`), the number that were coded using the ensemble method (`NUM_CONSENSUS_CODED`), the number that were coded using the probability method (`NUM_PROBABILITY_CODED`), the rate of over- or under-coding with each method (`PCT_CONSENSUS_CODED` and `PCT_PROBABILITY_CODED`), and the percentage that were correctly coded using either the ensemble method or the probability method (`PCT_CORRECTLY_CODED_CONSENSUS` or `PCT_CORRECTLY_CODED_PROBABILITY`, respectively).

The algorithm summary provides a breakdown of each algorithm's performance for each unique label in the classified data. This includes metrics such as precision, recall, F-scores, and the accuracy of each algorithm's results as compared to the true data.[5]

The document summary provides all the raw data available for each document. By document, it displays each algorithm's prediction (`ALGORITHM_LABEL`), the algorithm's probability score (`ALGORITHM_PROB`), the number of algorithms that agreed on the same label (`CONSENSUS_AGREE`), which algorithm had the highest probability score for its prediction (`PROBABILITY_CODE`), and the original label of the document (`MANUAL_CODE`). Finally, the `create_analytics()` function outputs information pertaining to ensemble analysis, which is discussed in its own section.

Summary and print methods are available for `create_analytics()`, but users can also store each summary in separate data frames for further analysis or writing to disk. Parameters include the container object and a matrix or `cbind()` object of classified results.[6]

[5] If a dataset is small (such as the present example), there is a chance that some f-score calculations will report `NaN` for some labels. This happens because not all labels were classified by a given algorithm (e.g., no cases were given a 5). In most real-world datasets, this will not happen.

[6] Users who conduct the analysis with the three low-memory algorithms will want to slightly modify the code below.

```
analytics <- create_analytics(container,
                              cbind(SVM_CLASSIFY, SLDA_CLASSIFY,
                                    BOOSTING_CLASSIFY, BAGGING_CLASSIFY,
                                    RF_CLASSIFY, GLMNET_CLASSIFY,
                                    NNET_CLASSIFY, TREE_CLASSIFY,
                                    MAXENT_CLASSIFY))
summary(analytics)

# CREATE THE data.frame SUMMARIES
topic_summary <- analytics@label_summary
alg_summary <- analytics@algorithm_summary
ens_summary <-analytics@ensemble_summary
doc_summary <- analytics@document_summary
```

6. Testing algorithm accuracy

While there are many techniques used to evaluate algorithmic performance (McLaughlin and Herlocker, 2004), the summary call to `create_analytics()` produces precision, recall and f-scores for analyzing algorithmic performance at the aggregate level. Precision refers to how often a case the algorithm predicts as belonging to a class actually belongs to that class. For example, in the context of the `USCongress` data, precision tells us what proportion of bills an algorithm deems to be about defense are actually about defense (based on the gold standard of human-assigned labels). In contrast, recall refers to the proportion of bills in a class the algorithm correctly assigns to that class. In other words, what percentage of actual defense bills did the algorithm correctly classify? F-scores produce a weighted average of both precision and recall, where the highest level of performance is equal to 1 and the lowest 0 (Sokolova et al., 2006).

Users can quickly compare algorithm performance. For instance, our working example shows that the SVM, glmnet, maximum entropy, random forest, SLDA, and boosting vastly outperform the other algorithms in terms of f-scores, precision, and recall. For instance, SVM's f-score is 0.65 whereas SLDA's f-score is 0.63, essentially no difference. Thus, if analysts wish to only use one algorithm, any of these top algorithms will produce comparable results. Based on these results, the Bagging, Tree, and NNET should not be used singly.[7]

[7] Note that these results are specific to these data. The overall sample size is somewhat small. Bagging, Tree, and NNET perform well with more data.

Algorithm	Precision	Recall	F-score
SVM	0.67	0.65	0.65
Glmnet	0.68	0.64	0.64
SLDA	0.64	0.63	0.63
Random Forest	0.68	0.62	0.63
Maxent	0.60	0.62	0.60
Boosting	0.65	0.59	0.59
Bagging	0.52	0.39	0.39
Tree	0.20	0.22	0.18
NNET	0.07	0.12	0.08

Table 1: Overall algorithm precision, recall, and f-scores

7. Ensemble agreement

We recommend ensemble (consensus) agreement to enhance labeling accuracy. Ensemble agreement simply refers to whether multiple algorithms make the same prediction concerning the class of an event (i.e., did SVM and maximum entropy label the text the same?). Using a four-ensemble agreement approach, Collingwood and Wilkerson (2012) found that when four of their algorithms agree on the label of a textual document, the machine label matches the human label over 90% of the time. The rate is just 45% when only two algorithms agree on the text label.

RTextTools includes `create_ensembleSummary()`, which calculates both recall accuracy and coverage for **n** ensemble agreement. Coverage simply refers to the percentage of documents that meet the recall accuracy threshold. For instance, say we find that when seven algorithms agree on the label of a bill, our overall accuracy is 90% (when checked against our true values). Then, let's say, we find that only 20% of our bills meet that criterion. If we have 10 bills and only two bills meet the seven ensemble agreement threshold, then our coverage is 20%. Mathematically, if k represents the percent of cases that meet the ensemble threshold, and n represents

total cases, coverage is calculated in the following way:

$$\text{Coverage} = \frac{k}{n} \tag{1}$$

Users can test their accuracy using a variety of ensemble cut-points, which is demonstrated below. Table 2 reports the coverage and recall accuracy for different levels of ensemble agreement. The general trend is for coverage to decrease while recall increases. For example, just 11% of the congressional bills in our data have nine algorithms that agree. However, recall accuracy is 100% for those bills when the 9 algorithms do agree. Considering that 90% is often social scientists' inter-coder reliability standard, one may be comfortable using a 6 ensemble agreement with these data because we label 66% of the data with accuracy at 90%.[8]

[8] This result is excellent, given that the training data consist of just a few thousand observations.

```
create_ensembleSummary(analytics@document_summary)
```

	Coverage	Recall
n >= 2	1.00	0.77
n >= 3	0.98	0.78
n >= 4	0.90	0.82
n >= 5	0.78	0.87
n >= 6	0.66	0.90
n >= 7	0.45	0.94
n >= 8	0.29	0.96
n >= 9	0.11	1.00

Table 2: Ensemble Agreement Coverage and Recall

8. Cross validation

Users may use n-fold cross validation to calculate the accuracy of each algorithm on their dataset and determine which algorithms to use in their ensemble. **RTextTools** provides a convenient `cross_validate()` function to perform n-fold cross validation. Note that when deciding the appropriate n-fold it is important to consider the total sample size so that enough data are in both the train and test sets to produce useful results.

```
SVM <- cross_validate(container, 4, "SVM")
GLMNET <- cross_validate(container, 4, "GLMNET")
MAXENT <- cross_validate(container, 4, "MAXENT")
SLDA <- cross_validate(container, 4, "SLDA")
BAGGING <- cross_validate(container, 4, "BAGGING")
BOOSTING <- cross_validate(container, 4, "BOOSTING")
RF <- cross_validate(container, 4, "RF")
NNET <- cross_validate(container, 4, "NNET")
TREE <- cross_validate(container, 4, "TREE")
```

9. Exporting data

Finally, some users may want to write out the newly labeled data for continued manual labeling on texts that do not achieve high enough accuracy requirements

during ensemble agreement. Researchers can then have human coders verify and label these data to improve accuracy. In this case, the document summary object generated from `create_analytics` can be easily exported to a CSV file.

```
write.csv(analytics@document_summary, "DocumentSummary.csv")
```

Conclusion

Although a user can get started with **RTextTools** in less than ten steps, many more options are available that help to remove noise, refine trained models, and ultimately improve accuracy. If you want more control over your data, please refer to the documentation bundled with the package. Moreover, we hope that the package will become increasingly useful and flexible over time as advanced users develop add-on functions. **RTextTools** is completely open-source, and a link to the source code repository as well as a host of other information can be found at the project's website – http://www.rtexttools.com.

Bibliography

F. Baumgartner, S. De Boef, and A. Boydstun. *The decline of the death penalty and the discovery of innocence*. Cambridge University Press, 2008. [p1]

L. Collingwood and J. Wilkerson. Tradeoffs in accuracy and efficiency in supervised learning methods. *Journal of Information Technology & Politics*, 9(3):298–318, 2012. [p5]

I. Feinerer, K. Hornik, and D. Meyer. Text mining infrastructure in R. *Journal of Statistical Software*, 25(5):1–54, 2008. [p2]

J. Friedman, T. Hastie, and R. Tibshirani. Regularization paths for generalized linear models via coordinate descent. *Journal of statistical software*, 33(1):1, 2010. [p2, 3]

B. Jones, H. Larsen-Price, and J. Wilkerson. Representation and American governing institutions. *The Journal of Politics*, 71(01):277–290, 2009. [p1]

T. P. Jurka. maxent: An R package for low-memory multinomial logistic regression with support for semi-automated text classification. *The R Journal*, 4(1):56–59, June 2012. URL http://journal.r-project.org/archive/2012-1/RJournal_2012-1_Jurka.pdf. [p2, 3]

A. Liaw and M. Wiener. Classification and regression by randomForest. *R news*, 2 (3):18–22, 2002. [p3]

M. McLaughlin and J. Herlocker. A collaborative filtering algorithm and evaluation metric that accurately model the user experience. In *Proceedings of the 27th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 329–336. ACM, 2004. [p5]

D. Meyer, E. Dimitriadou, K. Hornik, A. Weingessel, and F. Leisch. *e1071: Misc Functions of the Department of Statistics (e1071), TU Wien*, 2012. URL http://CRAN.R-project.org/package=e1071. R package version 1.6-1. [p2, 3]

A. Peters and T. Hothorn. *ipred: Improved Predictors*, 2012. URL http://CRAN.R-project.org/package=ipred. R package version 0.8-13. [p3]

B. Ripley. *tree: Classification and regression trees*, 2012. URL http://CRAN.R-project.org/package=tree. R package version 1.0-31. [p3]

M. Sokolova, N. Japkowicz, and S. Szpakowicz. Beyond accuracy, F-score and ROC: a family of discriminant measures for performance evaluation. *AI 2006: Advances in Artificial Intelligence*, pages 1015–1021, 2006. [p5]

J. Tuszynski. *caTools: Tools: moving window statistics, GIF, Base64, ROC AUC, etc.*, 2012. URL http://CRAN.R-project.org/package=caTools. R package version 1.13. [p3]

W. Venables and B. Ripley. *Modern applied statistics with S*. Springer, 2002. [p3]

Timothy P. Jurka
Department of Political Science
University of California, Davis
One Shields Avenue
Davis, CA 95616
USA
tpjurka@ucdavis.edu

Loren Collingwood
Department of Political Science
University of California, Riverside
900 University Avenue
Riverside, CA 92521
USA
lorenc@ucr.edu

Amber E. Boydstun
Department of Political Science
University of California, Davis
One Shields Avenue
Davis, CA 95616
USA
aboydstun@ucdavis.edu

Emiliano Grossman
Sciences Po /CEE
28, rue des Saints-Pères
75007 Paris
France
emiliano.grossman@sciences-po.fr

Wouter van Atteveldt
Communication Science Department
Vrije Universiteit
de Boelelaan 1081a
1081 HV Amsterdam
The Netherlands
wouter@vanatteveldt.com