Journal of Statistical Software

MMMMMM YYYY, Volume VV, Issue II. *doi: 10.18637/jss.v000.i00*

JOUSBoost: An R Package for Improving Machine Learning Classifier Probability Estimates

Matthew Olson
University of Pennsylvania

Abstract

Machine learning classifiers, such as AdaBoost, are often able to achieve superior misclassification error on a vast range of binary classification problems. However, these same algorithms often produce poor conditional class probability estimates. It is shown in Mease *et al.* (2007) that any classifier which achieves the Bayes error rate can be turned into a good probability estimator by fitting the classifier to different re-weightings of the original data set. The authors call this procedure *JOUS*, whose name derives from the steps of **J**ittering the training data, along with **O**ver/**U**nder-**S**ampling. The original paper applied this algorithm successfully to the AdaBoost classifier.

This paper describes an implementation of the JOUS algorithm in the R package **JOUSBoost** (Olson 2017). The package allows the user to apply JOUS to a generic classifier in order to produce probability estimates. This is useful not only for improving a classifier's existing probability estimates, but also for producing probability estimates for classifiers that have no natural probability output. The **JOUSBoost** package also contains an efficient implementation of AdaBoost. **JOUSBoost** is available from the Comprehensive R Archive Network (CRAN) at `https://CRAN.R-project.org/package=JOUSBoost`.

Keywords: probability estimation, machine learning, AdaBoost, R.

1. Introduction

It is common practice to evaluate a machine learning classifier by the fraction of cases it misclassifies on a test set, that is, its test error. In binary classification, the test error implicitly weights false negatives the same as false positives. Therefore, minimizing test error is equivalent to estimating the conditional class median (Elkan 2001). However, in many applications, a user is interested in classification that involves an asymmetric cost for different types of errors. For example, in a diagnostic test, it may be much more costly for the clinician to classify a sick patient as healthy than the more conservative mistake of classifying a healthy

patient as sick. More generally, one is often interested in estimating the probability of class membership for a new observation. This is a natural goal in a variety of contexts, including propensity score estimation, ranking, classification with unequal costs, and expected utility calculations, to name a few.

The manner in which one produces probability estimates differs greatly from classifier to classifier. Some methods, such as logistic regression or naive Bayes, can be cast in terms of generative probability models, from which it is natural to consider notions of probability. For example, logistic regression posits that the the log-odds of class membership is a linear function of the input variables. Once this linear function is estimated, one can estimate probabilities in a straightforward way. At the other end of the spectrum, there is a broad set of classifiers for which there is no natural notion of probability. Support vector machines, for example, are motivated from a purely optimization-based point of view: construct a hyperplane which maximally separates the classes in the training set. A priori, there is no model-based notion of a class probability.

While many of these classifiers can achieve superior generalization error in terms of test error, these same classifiers can sometimes produce very poor probability estimates (Niculescu-Mizil and Caruana 2005). One can find methodology for improving probability estimates from specific classifiers in the literature. As an example, Wang *et al.* (2008) suggest improvements for support vector machines based on bracketing. Their procedure is implemented in the R package **probsvm** (Zhang *et al.* 2013). Other methodology is more generally applicable to a broader class of machine learning classifiers. Niculescu-Mizil and Caruana (2005) explore Platt scaling (Platt 1999) and isotonic regression as a means to producing calibrated probability estimates. These methods are implemented in the R package **CORElearn** (Robnik-Sikonja and Savicky 2017) as well as the Python package **sklearn** (Pedregosa *et al.* 2011). These implementations are both limited to the extent that they require a classifier to produce a continuous valued "score" (instead of merely a class label), and they are restricted to classifiers implemented in each respective package.

In this article, we present the R package **JOUSBoost**, which implements methodology to improve the conditional class probability estimates returned by machine learning classifiers such as AdaBoost. In particular, we implement the Jittering with Under/Over Sampling algorithm of Mease *et al.* (2007), which allows one to form principled probability estimates from a classifier (we will refer to this algorithm as JOUS throughout). Unlike Platt scaling or isotonic regression, JOUS is generically applicable to any classifier, even those that are capable of only producing a class label, and not a continuous valued score. Our implementation was designed to be as efficient as possible, leveraging C++ and parallelization wherever possible. We also include a lightweight implementation of AdaBoost in our package, the classifier that was considered in the original JOUS paper (Mease *et al.* 2007). Section 4.1 provides implementation details as well as a performance comparison to other implementations of AdaBoost.

In Section 2 we provide background on the AdaBoost algorithm as well as its use and shortcoming in conditional class probability estimation. With this background, we then present the motivation for the JOUS algorithm in Section 3 and discuss some implementation details. After giving an outline of JOUS, we describe the **JOUSBoost** package in Section 4 as well as a number of use cases. In particular, we describe JOUS as applied to AdaBoost, gradient boosting, a support vector machine, and discuss options for use with a parallel back end. Finally, Appendix A includes some further details on the theoretical background underlying JOUS.

2. Background

In this section we will introduce the original AdaBoost.M1 algorithm proposed in Freund and Schapire (1996), as well as discuss a traditional approach for creating probability estimates from the classifier's output. After introducing AdaBoost, we will present a simulated example to demonstrate that traditional probability estimates returned by AdaBoost often diverge to 0 and 1, motivating the need for the JOUS algorithm.

2.1. AdaBoost

We begin with customary notation for a classification problem. We are given a set of N pairs of training data pairs $\{(x_i, y_i)\}_{i=1}^N$, where each *predictor variable* $x_i \in \mathcal{X}$ and each *response variable* $y_i \in \{-1, 1\}$. For statistical tractability, we assume that these pairs are independent and identically distributed, with the response variable linked to the predictor variable through the conditional class probability $\mathbb{P}(y = 1|x)$. In practice, this probability is always unknown. In many cases, the analyst is concerned with simply predicting a class for a new example x, but as we discussed, it is also of interest to form an estimate for the conditional class probability.

The AdaBoost algorithm introduced in Freund and Schapire (1996) has been found to be one of the most successful off-the-shelf classifiers in existence, and has led to the development of a large class of related machine learning classifiers known generically as boosting algorithms. AdaBoost is designed to take a simple base classification rule, known as a *weak learner*, and output a more powerful classifier by repeatedly fitting this rule on different weighted versions of the data set. In particular, one fits a collection of M base classifiers $f_1, \ldots, f_M$ to the data in sequence, each classifier working to correct the mistakes of the previous one. One can then form a prediction from this sequence of classifiers by aggregating their output into a score function F_M. The most common base classifier used in practice is a decision tree, and this is the base classifier we consider in our package for our implementation of AdaBoost. The precise steps of AdaBoost are laid out in Algorithm 1.

Algorithm 1 AdaBoost

1. Initialize the observation weights $w_i = \dfrac{1}{N},\ i = 1, 2, \ldots, N$.
2. For $m = 1$ to M:
   (a) Fit classifier f_m to the training data using weights w_i.
   (b) Compute the weighted training error $\epsilon_m = \sum_{i=1}^n w_i \mathbb{I}\left[y_i \neq f_m\left(x_i\right)\right]$.
   (c) Compute $\alpha_m = \frac{1}{2} \log \frac{1-\epsilon_m}{\epsilon_m}$.
   (d) Set $F_m = F_{m-1} + \alpha_m f_m$.
   (e) Update weights to $w_i = w_i = e^{-\alpha_m f_t(x_i) y_i}$ and normalize by $w_i = \frac{w_i}{\sum_{i=1}^n w_i}$.
3. Output prediction $F(x) = \operatorname{sign}\left(F_M\left(x\right)\right)$.

Despite a tremendous track record of achieving low misclassification error on a number of data sets, it is still a largely unresolved question as to why AdaBoost works so well. There have been a number of diverse explanations in the literature, many of which are summarized in the

book Schapire and Freund (2012). One of the most celebrated interpretations of AdaBoost has been given in Friedman *et al.* (2000), in which it is viewed as a stagewise optimization approach to fitting a logistic regression. The implication of this perspective is that one can use the score function F_M produced by the iterations of AdaBoost to form probability estimates according to

$$\mathbb{P}(y = 1|x) = \frac{1}{1 + e^{-2F_M(x)}}. \tag{1}$$

It should be emphasized that class probabilities do not appear anywhere in Algorithm 1, and as such, we can only judge Equation 1 to the extent that is is useful (or is not) in practice. However, this remains one of the most common ways to extract probability estimates from boosting algorithms, and is the default in popular packages such as the R package **ada**.

2.2. Probability estimates for AdaBoost

Although commonly used, it has been shown that Equation 1 often fails to produce reasonable probability estimates (Mease *et al.* 2007). In particular, for some data sets, it is optimal to run AdaBoost for a large number of rounds M in order to achieve the lowest possible classification error, but in doing so, the probabilities returned by Equation 1 are driven to the extremes of 0 and 1. In this section, we will present a simulated example to illustrate this point in order to demonstrate the need for the **JOUSBoost** machinery.

The model we consider here is borrowed from Friedman *et al.* (2000) and reveals the pattern of diverging probability estimates just discussed. In this probability model, we first draw a predictor variable $x \in \mathbb{R}^{10}$ with each component normally distributed with mean 0 and variance 1. We then generate the class label for the associated response y according to the follow log-odds ratio:

$$\log \frac{\mathbb{P}(y = 1|x)}{\mathbb{P}(y = -1|x)} = \gamma \left(1 - x^{(1)} + x^{(2)} - x^{(3)} + x^{(4)} - x^{(5)} + x^{(6)}\right) \sum_{j=1}^{6} x^{(j)}$$

where we set $\gamma = 0.5$. The parameter γ controls the Bayes error rate. In our simulations, we draw a training set of size $n = 500$ and we produce probability estimates for a holdout test set using AdaBoost applied to depth three decision trees in two ways: the logistic regression approach from Equation 1 and the **JOUSBoost** approach discussed in Section 3. We consider a few different metrics to compare the quality of probability estimates produced by these two methods:

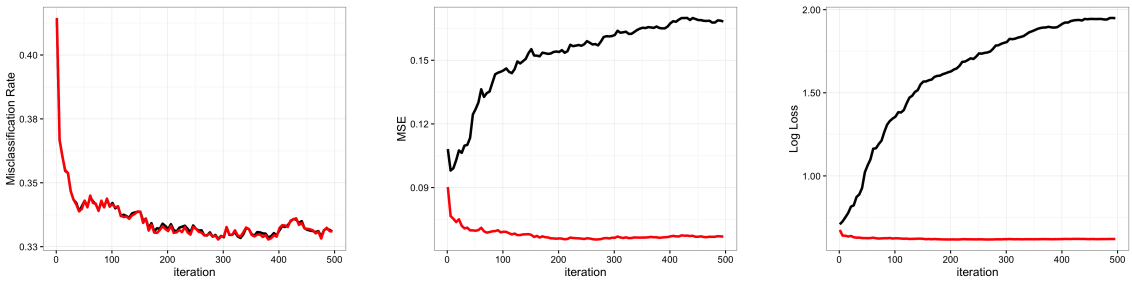
$$\text{Misclassification Error: : } \frac{1}{N} \sum_{i=1}^{N} \mathbb{I}(\widehat{y}_i \neq y_i)$$

$$\text{Mean Squared Error : } \frac{1}{N} \sum_{i=1}^{N} (p(x_i) - \widehat{p}(x_i))^2$$

$$\text{Log-Loss : } -\frac{1}{N} \sum_{i=1}^{N} p(x_i) \log \widehat{p}(x_i) + (1 - p(x_i)) \log(1 - \widehat{p}(x_i)))$$

where we use the notation $\mathbb{P}(y = 1|x) \equiv p(x)$, $\widehat{p}(x)$ is our estimate for $p(x)$, and $\widehat{y}$ is our class prediction.

In Figure 1, we plot the three different types of error as a function of the number of boosting iterations. The red line corresponds to the probabilities produced by JOUS, and the black line to the probabilities produced from the logistic link function. As demonstrated in Figure 1b, the misclassification rate is roughly monotonically decreasing with the number of boosting iterations for AdaBoost. However, the story is quite different for the error rates corresponding to mean squared error and log-loss. The quality of probability estimates produced by AdaBoost is almost strictly increasing with the number of boosting iterations: in other words, for the best probability estimates using the logistic link function, one should run AdaBoost for a very small number of iterations. However, this is clearly at odds with our desire to produce a small misclassification error rate. Conversely, AdaBoost run using the JOUS procedure for probability estimates shows much more reasonable behavior. In line with the behavior of the misclassification error, both mean squared error and log loss are monotonically decreasing with the number of boosting iterations, achieving values that are uniformly smaller than those produced by the logistic link function approach.



(a) Misclassification Error (b) Mean Squared Error (c) Log Loss

Figure 1: Plots showing misclassification error, mean squared error in probability estimates, and logarithmic loss for the AdaBoost classifier as a function of the number of boosting iterations. The black line is associated with AdaBoost when probabilities are calculated according to a logistic link function, and the red line is associated with AdaBoost when probabilities are calculated according to the under-sampled JOUS procedure.

3. The JOUS algorithm

In this section we present motivation and implementation of the JOUS algorithm.

3.1. Motivation

The intuition behind the JOUS algorithm is that probability estimation can be reduced to a sequence of classification problems. In order to see this, let us first take a small diversion to understand the problem of classification with unequal costs.

The first point to make is that classification with unequal costs amounts to estimating a conditional quantile of the data. As before, consider a random variable y taking values in $\{-1, 1\}$, a random variable x, and a conditional distribution $p = p(y = 1|x)$. Let us further assign costs to incorrect predictions about y. We will attach a cost $c \in (0, 1)$ to predicting that $y = 1$ when $y = -1$, a false positive, and a cost $(1 - c)$ to predicting that $y = -1$ when $y = 1$, a false negative. Our goal in making a prediction about the value of y conditional on

x should be to minimize our expected loss. Our expected loss when predicting that $y = 1$ is $(1 - p)c$, and our expected loss when predicting that $y = -1$ is $p(1 - c)$. Thus we should predict that $y = 1$ if and only if

$$(1 - p)c < p(1 - c),$$

or equivalently, when $c < p$. In the case when the cost of a false negative and false positive are the same, $c = 1/2$, and our optimal prediction boils down to choosing $y = 1$ only if the $p > 1/2$. In other words, achieving a low misclassification rate is exactly the same as estimating the conditional median. We can estimate other quantiles analogously by changing the costs of false negatives and false positives when fitting our classifier.

One way to tilt these costs is to directly modify the objective function being fit by our classifier. Oftentimes, a surrogate to misclassification loss is estimated - such as deviance or exponential loss - and one modifies this criterion instead. For example, when fitting decision trees, one way to account for asymmetric costs is to modify the splitting function used at each node in the tree to reflect these different costs. There have also been attempts in the literature to modify the AdaBoost algorithm to directly account for asymmetric costs (Fan *et al.* 1999). However, this approach is not always possible, and may not have the intended consequence in certain kinds of decision trees (Elkan 2001).

A more general way of assigning asymmetric costs in the fitting procedure is tilt the data itself by over or undersampling observations from one class (Elkan 2001). Mease *et al.* (2007) work out the exact proportion in which one should sample from the different classes in order to achieve q quantile estimation. In particular, if N_+ and N_- denote the number of positively and negatively labeled points in the data set, then one should sample $k+$ positively labeled examples and k_- negative examples in such a way to satisfy

$$\frac{k_+}{k_-} = \frac{N_+}{N_-}\frac{1-q}{q}. \tag{2}$$

When one applies a classifier to this sampled data set, a prediction of $y = 1$ will correspond to the assertion that $\mathbb{P}(y = 1|x) > q$ on the original data set. One can leverage the previous discussion of quantile estimation and data sampling to assemble an algorithm for probability estimation Algorithm 2.

Algorithm 2 JOUSBoost

1. Fix an integer δ and a grid of quantiles $\mathcal{Q} = \{1/\delta, 2/\delta, \ldots, 1 - 1/\delta\}$.
2. For $q \in \mathcal{Q}$:
   (a) Find k_+ and k_- that satisfy $\frac{k_+}{k_-} = \frac{N_+}{N_-}\frac{1-q}{q}$.
   (b) Sample k_+ observations for which $y_i = 1$ and k_- observations for which $y_i = -1$.
   (c) Fit classifier $\widehat{f}_q$ to the tilted data set.
3. Estimate $\mathbb{P}(y = 1|x) > q$ if $\widehat{f}_q(x) = 1$.

3.2. Implementation details

There are a number of details underlying the JOUS algorithm which are implemented in our R package. One important issue regards the manner in which observations are sampled from each class in step 2(b) of Algorithm 2. The only constraint the algorithm places on k_+ and k_- is that their ratio must equal a fixed constant. Our package implements both *over-sampling* and *under-sampling*. In the first case we set $k_+ = \delta N_+ (1-q)$ and $k_- = \delta N_- q$, and sample with replacement from each class. In the *under-sampling* case we set $k_+ = N_+ (1-q)$ and $k_- = N_- q$ and sample without replacement.

Another important detail in the JOUS algorithm relates to step 2(c). While in theory re-weighting the data set allows one to classify at different quantiles, the AdaBoost algorithm tends to behave somewhat differently. It was pointed out in Mease *et al.* (2007) that the AdaBoost classifier effectively treats a tilted data set as a re-initialization of its starting weights, and this has little affect on its final fit. In order to get around this issue, one can apply "jittering" to the data set by applying a small amount of uniform random noise to the predictors. This issue is discussed more carefully in Mease *et al.* (2007). There is also a very thoughtful discussion of the costs and benefits of over/under sampling in Chawla *et al.* (2002), along with discussion about perturbing the predictors when over-sampling.

Finally, it is very important to note that in practice one needs to take care to enforce monotonicity in the quantile estimates output by JOUS in step 3. It is clear that at the population level that for $q_1 \leq q_2$, $\mathbb{I}[\mathbb{P}(y=1|x) > q_2] \leq \mathbb{I}[\mathbb{P}(y=1|x) > q_1]$. However, there is no guarantee that this reasonable property should hold for our quantile estimation functions, that is, $\widehat{f}_{q_2}(x) \leq \widehat{f}_{q_1}(x)$. This problem was also pointed out in Wang *et al.* (2008). In order to enforce monotonicity, Mease *et al.* (2007) suggests estimating empirical probabilities in the following way:

- If $\widehat{f}_{0.5}(x) = 1$ set $\widehat{p}(x) = \min\{q > 0.5 | \widehat{f}_q(x) = 0\}$, and $\widehat{p}(x) = 1 - 1/(2\delta)$ if there is no such q.
- If $\widehat{f}_{0.5}(x) = 0$ set $\widehat{p}(x) = \max\{q < 0.5 | \widehat{f}_q(x) = 1\}$, and $\widehat{p}(x) = 1/(2\delta)$ if there is no such q.

There are a number of other more subtle implementation details, and the reader is invited to consult Mease *et al.* (2007) for more information.

On a final note, we would like to comment on the computational burden of the JOUS algorithm, as well as steps we have taken to make our implementation as efficient as possible. For one, step (2) of Algorithm 2 requires that we fit a classifier to δ different weighted versions of the data set, where the default value of $\delta = 10$. Moreover, when creating probability estimates on new data, we must create class predictions for each of the δ classifiers we fit during the training process. To alleviate some of this burden, we have included a parallel back end to both the fitting and predicting steps in JOUS using the R package **foreach** (Analytics and Weston 2014b). The user interface for this back end is described more in the following section. Furthermore, we discussed that step (3) of Algorithm 2 requires a enforcing a monotonicity constraint. We have implemented this step in C++ with the help of the R package **Rcpp** (Eddelbuettel and François 2011). In light of this discussion, we would like to remind the reader that the extra computational cost is the price one pays for using an algorithm with relatively relaxed assumptions.

4. The JOUSBoost package

In this section we will walk through the basic usage of the **JOUSBoost** package. In particular, we will discuss our implementation of AdaBoost, as well as demonstrate the application of the JOUS procedure to AdaBoost, a gradient boosting machine, and a support vector machine. Users can access the documentation of all of the functions discussed in this section in an R session through the use of the help commands, for instance, `?adaboost`.

The examples in this section rely on data generated from the *two-dimensional circle model* found in Mease *et al.* (2007) using the function `circle_data` in our package. The *two-dimensional circle model* consists of (x, y) pairs, where x is drawn uniformly over the square $[-28, 28]^2$ and the conditional probability that $y = 1$ given x is drawn according to

$$\mathbb{P}(y = 1|x) = \begin{cases} 1 & \text{if } r(x) < 8 \\ \frac{28 - r(x)}{20} & \text{if } 8 \leq r(x) \leq 28 \\ 0 & \text{if } r(x) > 8 \end{cases}$$

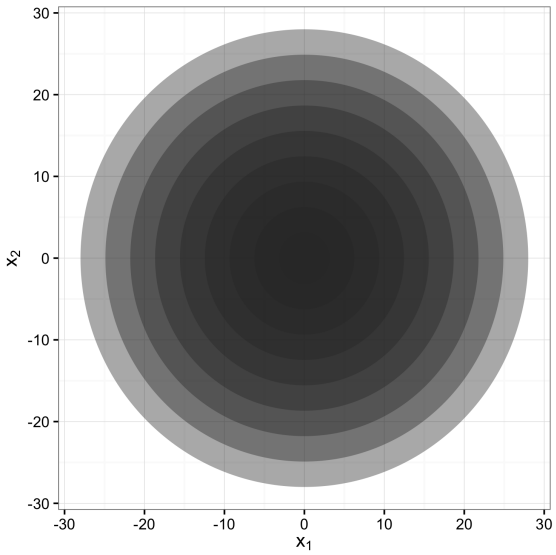


Figure 2: A plot of the conditional probability function for the *two-dimensional circle model*.

where $r(x)$ is the Euclidean distance of a point from the origin. It is easy to visualize the conditional density in this model as concentric circles, as in Figure 2. We will train our models on $n = 1,000$ examples generated from this distribution, and then we will evaluate our models on an independently drawn test set consisting of $n = 10,000$ examples. The user can replicate our work with the following code below:

```
R> library('JOUSBoost')
R> set.seed(123)
R> train_data = circle_data(1e3)
R> test_data = circle_data(1e4)
```

The lists `train_data` and `test_data` contains list members that consist of a matrix `X` of

predictor variables, a response vector `y` taking values in $\{-1, 1\}$, and a vector `prob` of the true underlying probabilities.

4.1. AdaBoost implementation

The **JOUSBoost** package contains a lightweight implementation of the AdaBoost algorithm applied to decision trees. The main inputs to the `adaboost` function are a matrix `X` of real values numbers, a vector `y` of integer class labels in $\{-1, 1\}$, the depth of decision tree for which to apply AdaBoost, and the number of boosting rounds. In addition, the user can further customize the type of decision tree fit at each iteration by passing in an optional `rpart.control` object. For instance one might specify the tree node-size or pruning options. The user can also specify whether to run the function in verbose mode by setting the flag `verbose = TRUE`. In the code below, we fit an AdaBoost classifier using depth 5 decision trees, boosted for 250 rounds.

```
R> boost = adaboost(train_data[['X']], train_data[['y']], tree_depth = 5,
+    n_rounds = 250)
R> boost

AdaBoost: tree_depth =  5  rounds =  250

 In-sample confusion matrix:
    yhat
y     -1   1
  -1 651   0
  1    0 349
```

The output of the fit produced by `adaboost` is an object of type `adaboost`. The object contains the decision trees fit by AdaBoost along with their weights, as well as a number of pieces of other information describing the parameters of the fit. There are corresponding `print` and `predict` generic functions that can be called on an object of type `adaboost`.

Each AdaBoost classifier has to store potentially hundreds of fitted decision trees (in this case 250), which can require a substantial amount of memory. We have taken measures to make our tree fits as lean as possible to minimize unnecessary overhead. A lot of the overhead comes from redundant formulas that are stored in the decision tree fits produced by the function `rpart` in the **rpart** package (Therneau *et al.* 2015). We have eliminated this redundancy, along with other of nonessential information stored in each tree.

In Table 1 we report the average training time in seconds and the relative memory usage for our implementation of `adaboost`, as well as that for the from three other R packages for the classification problem described in the beginning of this section. In particular, we consider **fastAdaboost**, **ada** (Culp *et al.* 2012), and **adabag**. The **JOUSBoost** version is slightly slower than that from **ada** and **fastAdaboost**. However, the memory usage is markedly smaller in our package. As reported in the second column of Table 1, our implementation requires about five times less memory that each of the other three packages (compare 6.5Mb to 35Mb). Each package constructs trees using **rpart**, but ours trims out redundant formulas.

Of course, each of these competing packages has its own strengths and weaknesses relative to ours. The Adaboost classifiers in **ada** and **adabag** contain other variations of boosting

	Training Time (s)	Relative Memory Usage
JOUSBoost	5.74	**1.00**
fastAdaboost	4.56	5.55
ada	**3.98**	4.72
adabag	15.99	5.57

Table 1: Performance comparison between different implementation of AdaBoost. The first column reports the average training time in seconds (over 10 replications), and the second column reports the fraction of memory required by each classifier relative to that of the **JOUSBoost** implementation. See Appendix B for more details.

algorithms, as well as support for variable importance, plotting, and other diagnostics. The **ada** package also supports other types of loss functions.

Returning to our example, it is also easy to make predictions with our fitted model. The default setting on the `predict` method is to return class label predictions, but one can also obtain probability estimates in the usual way by passing the score through a logistic link function (Friedman *et al.* 2000). As an additional feature, one can optionally pass an `n_tree` argument to `predict` to make predictions using only the first `n_tree` trees in the ensemble. If one has access to a holdout set, this feature can be useful when deciding how many rounds of boosting to use.

```
R> yhat = predict(boost, test_data[['X']])
R> phat = predict(boost, test_data[['X']], type='prob')
```

Figure 3 shows a histogram of the estimated probabilities returned by `adaboost`. It is clear that the classifier is pushing the probabilities towards 0 and 1, a behavior that we discussed in Section 2.2. The log-loss for this example is 0.96, which we will use as a reference in the next section.

4.2. JOUS for probability estimates

The main function in this package is `jous`, which allows one to calculate probability estimates using the sampling procedure described in Section 3. We implemented this function to work generically with classification methods besides AdaBoost. The price of this flexibility is that the user must write wrapper functions around the desired classifier to conform with the expected input to `jous`. In the following piece of code, we fit `jous` to the `adaboost` classifier considered in the last section:

```
R> class_func = function(X, y) adaboost(X, y, tree_depth = 5, n_rounds=250)
R> pred_func = function(fit_obj, X) predict(fit_obj, X)
R> jous_fit = jous(train_data[['X']], train_data[['y']], class_func = class_func,
+    pred_func = pred_func, keep_models = TRUE)
R> jous_fit

JOUS fit:
    type:  under
```

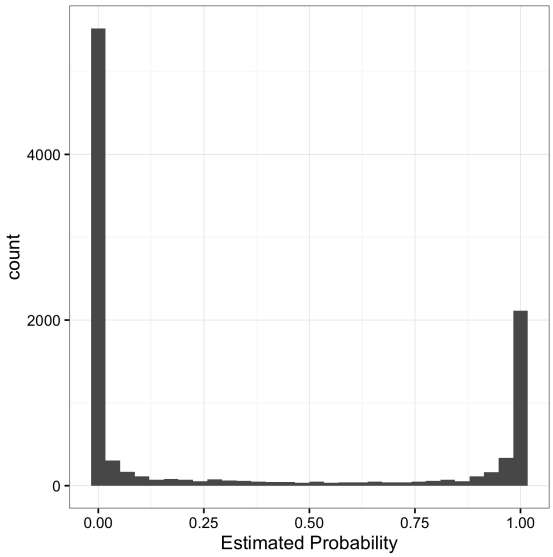


Figure 3: A histogram of probability estimates produced by AdaBoost on the *two-dimensional circle example*. The probability estimates are pushed to the extremes of 0 and 1.

```
    delta:  10

 In-sample confusion matrix:
    yhat
y     -1   1
  -1 651   0
  1    0 349
```

The `jous` function expects to receive two functions as arguments, namely `class_func` and `pred_func`. The `class_func` function takes as arguments a matrix `X` and a vector `y` of integer class labels in $\{-1, 1\}$, and should return an object for which one can create predictions on new data, usually through an interface to the `predict` generic function. The `pred_func` function takes as arguments the object returned by `class_func` and a matrix of predictors for test data, and returns integer class labels in $\{-1, 1\}$. In order to illustrate how these wrapper functions get used, we have included code below that replicates the analysis in the previous section.

```
R> ada_obj = class_func(train_data[['X']], train_data[['y']])
R> yhat = pred_func(ada_obj, test_data[['X']])
```

Unfortunately, such wrappers are necessary since there is no uniform interface in R for fitting different types of classifiers to data: some classifier functions take matrices as inputs, while others take data frames and accept formula notation for inputs. The advantage to this approach is that `jous` can be applied even to user defined classifiers. It is also worth mentioning that since `delta` models will be fit in total, the user should strive to make the `class_func` function as lean as possible in terms of memory usage.

Of course, we can also use our fit from the `jous` function to produce probability estimates.

```
R> yhat = predict(jous_fit, test_data[['X']])
R> phat = predict(jous_fit, test_data[['X']], type='prob')
```

The log-loss in probability estimates from using the JOUS procedure is 0.46, compared to the log-loss from the direct AdaBoost probability estimate of 0.96. Note that both methods achieve exactly the same misclassification rate.

We will close this section with a brief summary of additional important arguments to `jous`:

```
jous(X, y, class_func, pred_func, type="under", delta = 10, nu=1, X_pred=NULL,
  keep_models=FALSE, verbose=FALSE, parallel=FALSE, packages=NULL)
```

- `type` - whether to consider "over" or "under-sampling" when creating artificial data sets.
- `delta` - the number of quantiles to consider in grid: the default value of 10 works well.
- `nu` - the amount of "jittering" to apply to predictors when using oversampling.
- `X_pred` - matrix of predictors at which to create probability estimates.
- `keep_models` - whether or not to store each model: needs to be set to `TRUE` if the user wants to use `predict`.
- `verbose` - whether or not to print progress to the screen.
- `parallel` - whether or not to fit models using a parallel back-end.
- `packages` - a character vector of package names that get used in the `class_func` and `pred_func` functions. Only necessary when `parallel=TRUE`.

4.3. JOUS for gradient boosting

In this section we will apply `jous` to gradient boosting machines from the R package **gbm** (Ridgeway 2015). Gradient boosting is similar in nature to AdaBoost, but explicitly minimizes binomial deviance using a gradient descent procedure (Friedman 2001). The role of this example is twofold. First, the probabilities produced through gradient boosting should be more principled since they arise from an explicit probability model. It is worthwhile to see how the JOUS methodology performs in this scenario. Second, this example demonstrates a more involved construction of wrapper functions.

We rely on the `gbm` function from the **gbm** package to fit our gradient boosting machine. Here, `gbm` requires a data frame as an input argument, along with a formula that specifies the target model. Our `class_func` wrapper, on the other hand, expects a matrix and a vector as inputs, so we need to make the transformation explicitly in our wrapper.

```
R> class_func = function(X, y) {
+    df = data.frame(X=X, y=ifelse(y < 0, 0, 1))
+    gbm(y ~ ., data=df, shrinkage=0.1, n.tree = 1000,
+      interaction.depth = 1, distribution='bernoulli', bag.fraction = 0.5)}
```

The prediction function for `gbm` also requires a data frame as an input, so our `pred_func` wrapper also needs to make a similar transformation. Here, there is the added complication that the data frame needs to have the same names as the training data, but luckily we can extract those from the fitted `gbm` object that gets passed into our wrapper.

```
R> pred_func = function(obj, X){
+     df = as.data.frame(X)
+     names(df) = obj$var.names
+     pred = predict(obj, df, n.tree=length(obj$trees))
+     ifelse(pred < 0, -1, 1)}
```

In Figure 4, we plot the test error rate, mean square error, and log loss as a function of boosting iterations for two different gradient boosting machines. The top set of figures correspond to boosting with depth eight trees, while the bottom set correspond to boosting with depth one trees. The red line shows error rates for JOUS, while the black line shows error rates for the probabilities output by `gbm`. When the depth of tree is set to eight, mean square error (Figure 4b) and log-loss (Figure 4c) increase quickly with the number of iterations as compared to JOUS. When the depth of tree is set to one, this phenomenon disappears and the two methods of probability estimates are very comparable (Figures 4e and 4f). Thus, in this instance, we can view `jous` as a hedge: `jous` returns reasonable probability estimates in both settings.

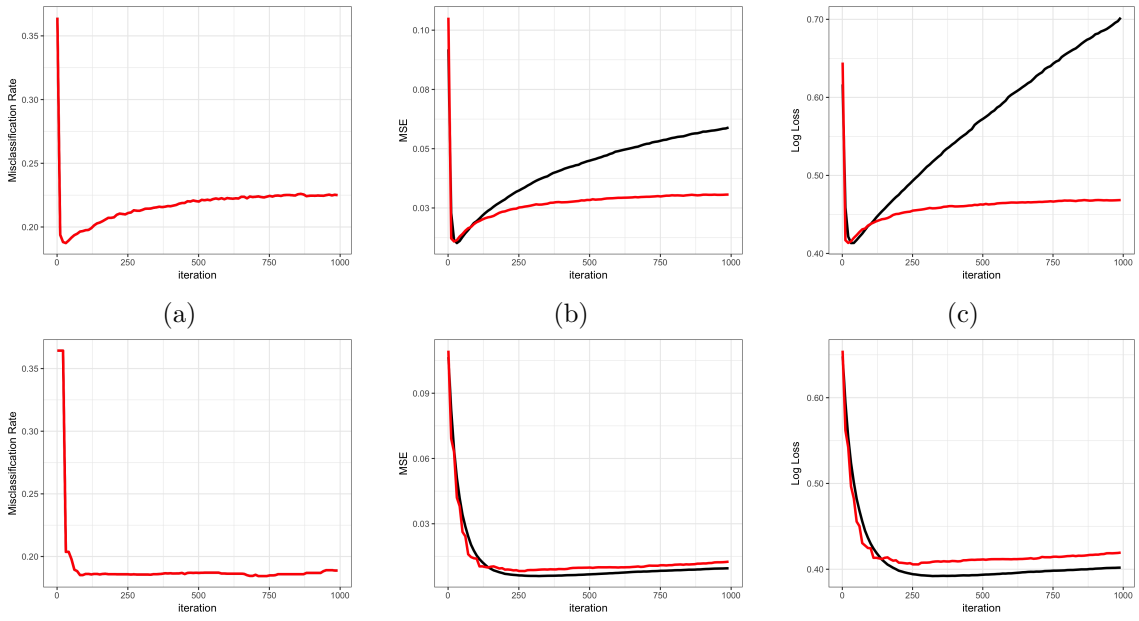


Figure 4: Plots of the quality of probability estimates as a function of the number of boosting rounds for a gradient boosting machine. The top three figures correspond to boosting with depth 8 trees, and the bottom three figures correspond to boosting with depth 1 trees. Probabilities produced by `jous` appear in red, while those from `gbm` appear in black.

4.4. JOUS for support vector machines

In this example, we will consider applying `jous` to a support vector machine (SVM) with radial basis kernel as found in the package **kernlab** (Karatzoglou *et al.* 2004). We will begin by fitting an SVM to the data, and the using its output to create a probability estimate.

```
R> library('kernlab')
R> svp = ksvm(train_data[['X']], as.factor(train_data[['y']]), kernel='rbfdot')
```

We form probability estimates on the test set by normalizing the output of the support vector machine. This is not the only way to do this, but it is commonly done in practice and will serve for illustration purposes.

```
R> score = predict(svp, test_data[['X']], type='decision')
R> phat_svm = (score - min(score))/(max(score) - min(score))
```

Alternatively, we can form probability estimates in a more principled way by appealing to the quantile estimation performed by `jous`.

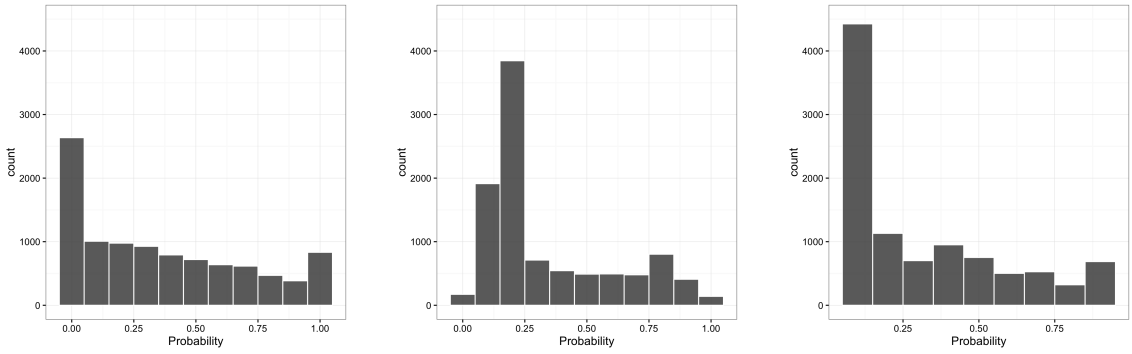
```
R> class_func = function(X, y) ksvm(X, as.factor(y), kernel='rbfdot')
R> pred_func = function(obj, X) as.numeric(as.character(predict(obj, X)))
R> jous_obj = jous(train_data[['X'], train_data[['y']], class_func = class_func,
+    pred_func = pred_func, keep_models = T)
R> jous_pred = predict(jous_obj, test_data[['X']], type='prob')
```

We need to do a little extra work here when specifying our wrapper functions. The function `class_func` takes a vector of integers while `ksvm` expects a factor, and so we must include code in our wrapper to make the conversion. Similarly, `pred_func` needs to return an integer, but the `predict` function returns a factor, and so conversions are needed here as well.

Figure 5 shows histograms of the actual and estimated probabilities. The histogram on the far left shows what the distribution of the true probabilities should look like, and the two histograms to the right show the distribution of probability estimates using a naive approach and the JOUS approach. The naive approach concentrates probability in the wrong place - near 0.2 - while the estimated produced by `jous` come closer to the truth.

4.5. Using a parallel back-end

In this section we will provide a use case for running the `jous` function in parallel in the example from the previous section. Before the user can run `jous` in parallel, she must first instantiate a parallel back-end, such as that from the R package **doParallel** (Analytics and Weston 2014a). There are two remaining arguments that need to be set in order to fit models in parallel: `parallel` and `packages`. The second argument is a vector of strings that list the libraries that get used in the `class_func` and `pred_func` functions. This extra step is necessary since the nodes to which `foreach` allocates jobs do not share the same environment as the code that calls it, and so we need to load whatever libraries are necessary on the worker nodes. After registering the parallel back-end, one can then fit a JOUS model and create predictions as follows:



(a) True probability (b) SVM probability (c) JOUS probability

Figure 5: Histograms of probability estimates using a support vector machine.

```
R> library('doParallel')
R> cl <- makeCluster(4)
R> registerDoParallel(cl)

R> set.seed(123)
R> train_data = circle_data(1e4)
R> test_data = circle_data(1e4)

R> start_time = Sys.time()
R> jous_obj = jous(train_data[['X']], train_data[['y']], class_func = class_func,
    pred_func = pred_func, parallel = TRUE, packages='kernlab',
    keep_models = T)
R> phat_jous = predict(jous_obj, test_data[['X']], type='prob')
R> end_time = Sys.time()
R> end_time-start_time

R> stopCluster(cl)

Time difference of 11.97582 secs
```

We also include the serial version of the code for a speed comparison. With four cores, one can just about cut the model fitting time in half.

```
R> start_time = Sys.time()
R> jous_obj = jous(train_data[['X']], train_data[['y']], class_func = class_func,
     pred_func = pred_func, keep_models = T)
R> phat_svm = predict(jous_obj, test_data[['X']], type='prob')
R> end_time = Sys.time()
R> end_time-start_time

Time difference of 22.68876 secs
```

5. Conclusion

This article introduced the R package **JOUSBoost** for probability estimation with machine learning classifiers, especially AdaBoost. Our intention was to motivate the necessity for the JOUS methodology described in Mease *et al.* (2007) through the lens of its original application, correcting the diverging probability estimates produced by AdaBoost. The aim of the package was to extend this methodology to more general machine learning classifiers. This was accomplished through modular code and efforts to make the implementation as efficient as possible through C++ and parallelization. This article outlines several practical use-cases of the `jous` function, specifically, correcting probability estimates from AdaBoost and a support vector machine on simulated data. Our package also allows one to easily replicate the results from Mease *et al.* (2007).

There is room for further improvement in both the JOUS algorithm and the package itself. Recall that in Section 3.2 we discussed that one needs to take special care to ensure that the estimated probabilities obey a natural monotonicity condition. The implemented solution is admittedly ad hoc, and lacks theoretical justification. We also plan to introduce additional methodology for probability estimation in later iterations of the package. Notably, we intend to include support for Platt Scaling and Isotonic Regression which are popular methods for producing calibrated probabilities from machine learning classifiers (Niculescu-Mizil and Caruana 2005). While support for these methods is available in the Python package **scikit-learn**, there is not yet a generic implementation in R (Pedregosa *et al.* 2011).

Finally, we would like to note that JOUS can produce good probability estimates only under the assumption that the underlying classifier achieves (near) the Bayes error rate. Therefore, we would not expect this procedure to be useful in all cases, especially when classification performance is poor.

Acknowledgments

.

The author would like to acknowledge Justin Bleich and Josh Magarick for helpful comments about the package's design.

A. Derivation of re-weighting scheme

In this appendix we will spell out in a more detail the derivation of Equation 2, which gives the ratio in which one must sample positive and negative examples in the training data in order to achieve classification at conditional probability q. Our exposition will follow Mease *et al.* (2007). Let us first define the following set of quantities:

$$
\begin{aligned}
p(x) &= \mathbb{P}(y = 1|x) \\
\pi &= \mathbb{P}(y = 1) \\
f_+(x) &= \mathbb{P}(x|y = 1) \\
f_-(x) &= \mathbb{P}(x|y = -1)
\end{aligned}
$$

which give the conditional class probability that $y = 1$ given x, the unconditional probability that $y = 1$, and the conditional distributions of the predictor x given the class labels, respectively. Using Bayes rule, we may write

$$p(x) = \frac{f_+(x)\,\pi}{f_+(x)\,\pi + f_-(x)\,\pi}.$$

We can the write the ratio of conditional class probabilities as

$$\frac{p(x)}{1 - p(x)} = \frac{f_+(x)\,\pi}{f_-(x)\,(1 - \pi)}$$

which is the same as

$$\frac{p(x)\,(1 - \pi)}{(1 - p(x))\,\pi} = \frac{f_+(x)}{f_-(x)}.$$

Now, assume that $f_+(x)$ and $f_+(x)$ do not change if we alter the marginal distribution of y. Let us suppose that the marginal distribution of y changes from π to π^*, and denote the new corresponding conditional class probability as $p^*(x)$. Using our assumption, we may now write

$$\frac{p(x)\,(1 - \pi)}{(1 - p(x))\,\pi} = \frac{p^*(x)\,(1 - \pi^*)}{(1 - p^*(x))\,\pi^*}.$$

It is clear that the thresholds q on $p(x)$ and q^* on $p^*(x)$ transform in the same way so that

$$\frac{q\,(1 - \pi)}{(1 - q)\,\pi} = \frac{q^*\,(1 - \pi^*)}{(1 - p^*(x))\,\pi^*}.$$

This leaves us with the following question: what value of π^* allows us to map q to $q^* = 1/2$ (given π). Simple algebra shows the value of π^* satisfies

$$\frac{\pi^*}{1 - \pi^*} = \frac{(1 - q)\pi}{q(1 - \pi)}. \tag{3}$$

All of this occurs at the "population" level. If we observe N_+ positively labeled examples and N_- negatively labeled examples in our training data, we estimate π with $\frac{N_+}{N_+ + N_-}$. Our target data set with π^* is generated by resampling the observed data to arrive at k_+ positive examples and k_- examples such that $\pi^* = \frac{k_+}{k_+ + k_-}$ and π^* satisfies Equation 3.

B. Code for Section 2.2

```
set.seed(123)

library('rbenchmark')
library('adabag')
library('ada')
library('fastAdaboost')

train_data = JOUSBoost::circle_data(1e3)
test_data = JOUSBoost::circle_data(1e4)

train_df = data.frame(X=train_data$X, y=as.factor(train_data$y))
test_df = data.frame(X=test_data$X, y=as.factor(test_data$y))

n_rounds = 250
n_reps = 10

t_fast = benchmark({
  fast_boost = adaboost(y ~ ., train_df, nIter=n_rounds)
  yhat_fast = predict(fast_boost, newdata=test_df)
}, replications = n_reps)

control_fast = get_tree(fast_boost, 1)[['0']]$control
t_jous = benchmark({
  jous_boost = JOUSBoost::adaboost(train_data[['X']], train_data[['y']],
                                   n_rounds=n_rounds, control=control_fast)
  yhat_jous = JOUSBoost::predict.adaboost(jous_boost, test_data[['X']])
}, replications=n_reps)

t_ada = benchmark({
  ada_boost = ada(train_data[['X']], train_data[['y']], loss='exponential',
                  type='discrete', iter=n_rounds, bag.frac=1, control_fast)
  yhat_ada = predict(ada_boost, as.data.frame(test_data[['X']]))
}, replications=n_reps)

t_adabag = benchmark({
  adabag_boost = boosting(y ~ ., data = train_df, mfinal=n_rounds, boos=FALSE,
                          coeflearn='Freund', control=control_fast)
  yhat_adabag = predict(adabag_boost, test_df)
}, replications=n_reps)

t_fast$elapsed
t_jous$elapsed
t_ada$elapsed
```

```
t_adabag$elapsed

object.size(fast_boost)
object.size(jous_boost)
object.size(ada_boost)
object.size(adabag_boost)
```

References

Analytics R, Weston S (2014a). ***doParallel: foreach*** *Parallel Adaptor for the Parallel Package*. R package version 1.0.8, URL http://CRAN.R-project.org/package=doParallel.

Analytics R, Weston S (2014b). ***foreach:*** *Foreach Looping Construct for R*. R package version 1.4.2, URL http://CRAN.R-project.org/package=foreach.

Chawla NV, Bowyer KW, Hall LO, Kegelmeyer WP (2002). "SMOTE: Synthetic Minority Over-Sampling Technique." *Journal of artificial intelligence research*, **16**, 321–357.

Culp M, Johnson K, Michailidis G (2012). ***ada:*** *an R Package for Stochastic Boosting*. R package version 2.0-3, URL http://CRAN.R-project.org/package=ada.

Eddelbuettel D, François R (2011). "**Rcpp**: Seamless R and C++ Integration." *Journal of Statistical Software*, **40**(8), 1–18. URL http://www.jstatsoft.org/v40/i08/.

Elkan C (2001). "The Foundations of Cost-Sensitive Learning." In *International joint conference on artificial intelligence*, volume 17, pp. 973–978. LAWRENCE ERLBAUM ASSOCIATES LTD.

Fan W, Stolfo SJ, Zhang J, Chan PK (1999). "AdaCost: Misclassification Cost-Sensitive Boosting." In *Icml*, pp. 97–105.

Freund Y, Schapire RE (1996). "Experiments with a New Boosting Algorithm." In *ICML*, volume 96, pp. 148–156.

Friedman J, Hastie T, Tibshirani R, *et al.* (2000). "Additive Logistic Regression: a Statistical View of Boosting (with discussion and a rejoinder by the authors)." *The Annals of Statistics*, **28**(2), 337–407.

Friedman JH (2001). "Greedy function approximation: a gradient boosting machine." *Annals of statistics*, pp. 1189–1232.

Karatzoglou A, Smola A, Hornik K, Zeileis A (2004). "**kernlab** – An S4 Package for Kernel Methods in R." *Journal of Statistical Software*, **11**(9), 1–20. URL http://www.jstatsoft.org/v11/i09/.

Mease D, Wyner A, Buja A (2007). "Cost-Weighted Boosting with Jittering and Over/Under-Sampling: JOUS-boost." *J. Machine Learning Research*, **8**, 409–439.

Niculescu-Mizil A, Caruana R (2005). "Predicting Good Probabilities with Supervised Learning." In *Proceedings of the 22nd international conference on Machine learning*, pp. 625–632. ACM.

Olson M (2017). ***JOUSBoost**: Implements Under/Oversampling for Probability Estimation*. R package version 2.1.0, URL http://CRAN.R-project.org/package=JOUSBoost.

Pedregosa F, Varoquaux G, Gramfort A, Michel V, Thirion B, Grisel O, Blondel M, Prettenhofer P, Weiss R, Dubourg V, *et al.* (2011). "**Scikit-learn**: Machine learning in Python." *Journal of Machine Learning Research*, **12**(Oct), 2825–2830.

Platt JC (1999). "Probabilistic Outputs for Support Vector Machines and Comparisons to Regularized Likelihood Methods." In *ADVANCES IN LARGE MARGIN CLASSIFIERS*, pp. 61–74. MIT Press.

Ridgeway G (2015). ***gbm**: Generalized Boosted Regression Models*. R package version 2.1.1, URL http://CRAN.R-project.org/package=gbm.

Robnik-Sikonja M, Savicky P (2017). ***CORElearn**: Classification, Regression and Feature Evaluation*. R package version 1.50.3, URL https://CRAN.R-project.org/package=CORElearn.

Schapire RE, Freund Y (2012). *Boosting: Foundations and algorithms*. MIT press.

Therneau T, Atkinson B, Ripley B (2015). ***rpart**: Recursive Partitioning and Regression Trees*. R package version 4.1-9, URL http://CRAN.R-project.org/package=rpart.

Wang J, Shen X, Liu Y (2008). "Probability Estimation for Large-Margin Classifiers." *Biometrika*, **95**(1), 149–167.

Zhang C, Shin SJ, Wang J, Wu Y, Zhang HH, Liu Y (2013). ***probsvm**: Class Probability Estimation for Support Vector Machines*. R package version 1.00, URL https://CRAN.R-project.org/package=probsvm.

Affiliation:

Matthew Olson
The Wharton School of the University of Pennsylvania
E-mail: maolson@wharton.upenn.edu

Journal of Statistical Software http://www.jstatsoft.org/
published by the Foundation for Open Access Statistics http://www.foastat.org/
MMMMMM YYYY, Volume VV, Issue II *Submitted:* yyyy-mm-dd
doi:10.18637/jss.v000.i00 *Accepted:* yyyy-mm-dd