The Stata Journal (2020)
20, Number 1, pp. 3–29 DOI: 10.1177/1536867X20909688

The random forest algorithm for statistical learning

Matthias Schonlau
University of Waterloo
Waterloo, Canada
schonlau@uwaterloo.ca

Rosie Yuyan Zou
University of Waterloo
Waterloo, Canada
y53zou@uwaterloo.ca

Abstract. Random forests (Breiman, 2001, *Machine Learning* 45: 5–32) is a statistical- or machine-learning algorithm for prediction. In this article, we introduce a corresponding new command, `rforest`. We overview the random forest algorithm and illustrate its use with two examples: The first example is a classification problem that predicts whether a credit card holder will default on his or her debt. The second example is a regression problem that predicts the log-scaled number of shares of online news articles. We conclude with a discussion that summarizes key points demonstrated in the examples.

Keywords: st0587, rforest, random decision forest algorithm

1 Introduction

In recent years, the use of statistical- or machine-learning algorithms has increased in the social sciences.[1] For instance, to predict economic recession, Liu et al. (2017) compared ordinary least-squares regression results with random forest regression results and obtained a considerably higher adjusted R-squared value with random forest regression compared with ordinary least-squares regression (Nyman and Ormerod 2017). In economics, a recent book overviews various statistical-learning algorithms for predicting economic growth and recession (Basuchoudhary, Bang, and Sen 2017). In environmental science, a recent article used learning algorithms, including least absolute shrinkage and selection operator regression, random forest, and neural networks, to predict ragweed pollen concentration based on 27 years of historical data and 85 predictor variables, with the best predictive performance obtained using random forest.

Why does random forest do better than linear regression for prediction tasks? Linear regression makes the assumption of linearity. This assumption makes the model easy to interpret but is often not flexible enough for prediction. Random decision forests easily adapt to nonlinearities found in the data and therefore tend to predict better than linear regression. More specifically, ensemble learning algorithms like random forests are well suited for medium to large datasets. When the number of independent variables is larger than the number of observations, linear regression and logistic regression algorithms will not run, because the number of parameters to be estimated exceeds the number of observations. Random forest works because not all predictor variables are used at once.

1. "Statistical learning" and "machine learning" are synonymous. We use "statistical learning" for the remainder of the article.

Random forest is one of the best-performing learning algorithms. For social scientists, such developments in algorithms are useful only to the extent that they can access an implementation of the algorithm. In this article, we introduce `rforest`, a command for random forests developed by the authors that is built on the Weka library (Witten et al. 2016; Hall et al. 2009).

The outline of this article is as follows: In section 2, we briefly discuss the random forest algorithm. In section 3, we give the syntax of the `rforest` command. In section 4, we give an example for predicting whether a given credit card user will default on his or her debt. In section 5, we give an example for estimating the log-scaled number of shares of online news articles. In section 6, we conclude with a discussion.

2 The random forest algorithm

We first discuss tree-based models because they form the building blocks of the random forest algorithm. A tree-based model involves recursively partitioning the given dataset into two groups based on a certain criterion until a predetermined stopping condition is met. At the bottom of decision trees are so-called leaf nodes or leaves.

Figure 1 illustrates a recursive partitioning of a two-dimensional input space with axis-aligned boundaries—that is, each time the input space is partitioned in a direction parallel to one of the axes. Here the first split occurred on $x_2 \geq a_2$. Then, the two subspaces were again partitioned: The left branch was split on $x_1 \geq a_4$. The right branch was first split on $x_1 \geq a_1$, and one of its subbranches was split on $x_2 > a_3$. Figure 2 is a graphical representation of the subspaces partitioned in figure 1.

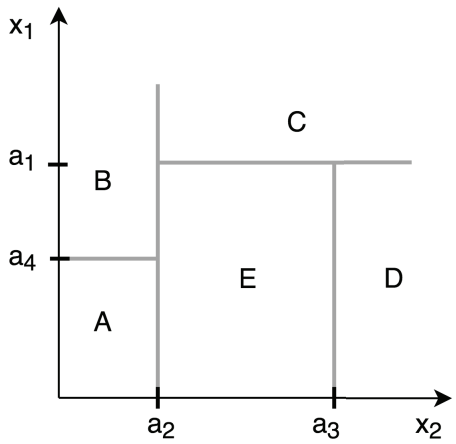

Figure 1. Recursive binary partition of a two-dimensional subspaces

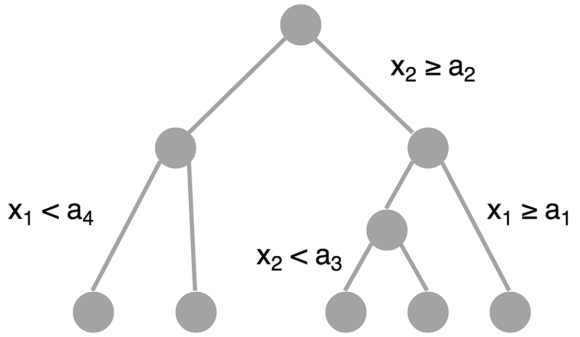

Figure 2. A graphical representation of the decision tree in figure 1

Depending on how the partition and stopping criteria are set, decision trees can be designed for both classification tasks (categorical outcome, for example, logistic regression) and regression tasks (continuous outcome).

For both classification and regression problems, the subset of predictor variables selected to split an internal node depends on predetermined splitting criteria that are formulated as an optimization problem. A common splitting criterion in classification problems is entropy, which is the practical application of Shannon's (2001) source coding theorem that specifies the lower bound on the length of a random variable's bit representation. At each internal node of the decision tree, entropy is given by the formula

$$E = -\sum_{i=1}^{c} p_i \times \log(p_i)$$

where c is the number of unique classes and p_i is the prior probability of each given class. This value is maximized to gain the most information at every split of the decision tree. For regression problems, a commonly used splitting criterion is the mean squared error at each internal node.

A drawback of decision trees is that they are prone to overfitting, which means that the model follows the idiosyncrasies of the test dataset too closely and performs poorly on a new dataset—that is, the test data. Overfitting decision trees will lead to low general predictive accuracy, also referred to as generalization accuracy.

One way to increase generalization accuracy is to consider only a subset of the observations and build many individual trees. First introduced by Ho (1995), this idea of the random-subspace method was later extended and formally presented as the random forest by Breiman (2001). The random forest model is an ensemble tree-based learning algorithm; that is, the algorithm averages predictions over many individual trees. The individual trees are built on bootstrap samples rather than on the original sample. This is called bootstrap aggregating or simply bagging, and it reduces overfitting. The algorithm is as follows:

```
for i ← 1 to B do
    Draw a bootstrap sample of size N from the training data;
    while node size != minimum node size do
        randomly select a subset of m predictor variables from total p;
        for j ← 1 to m do
            if jth predictor optimizes splitting criterion then
                split internal node into two child nodes;
                break;
            end
        end
    end
end
return the ensemble tree of all B subtrees generated in the outer for loop;
```

Algorithm 1. Random forest algorithm

Individual decision trees are easily interpretable, but this interpretability is lost in random forests because many decision trees are aggregated. However, in exchange, random forests often perform much better on prediction tasks.

The random forest algorithm more accurately estimates the error rate compared with decision trees. More specifically, the error rate has been mathematically proven to always converge as the number of trees increases (Breiman 2001).

The error of the random forest is approximated by the out-of-bag (OOB) error during the training process. Each tree is built on a different bootstrap sample. Each bootstrap sample randomly leaves out about one-third of the observations. These left-out observations for a given tree are referred to as the OOB sample. Finding parameters that would produce a low OOB error is often a key consideration in model selection and parameter tuning. Note that in the random forest algorithm, the size of the subset of predictor variables, m, is crucial to controlling the final depth of the trees. Hence, it is a parameter that needs to be tuned during model selection, which will be discussed in the examples later.

To gain some insight on the complex model, we calculate the so-called variable importance of each variable. This is calculated by adding up the improvement in the objective function given in the splitting criterion over all internal nodes of a tree and across all trees in the forest, separately for each predictor variable. In the Stata implementation of random forest, the variable importance score is normalized by dividing all scores over the maximum score: the importance of the most important variable is always 100%.

3 Syntax

The syntax to fit a random forest model is

```
rforest depvar indepvars [if] [in] [, type(string) iterations(int)
   numvars(int) depth(int) lsize(int) variance(real) seed(int)
   numdecimalplaces(int)]
```

with the following postestimation command:

```
predict newvar|varlist|stub* [if] [in] [, pr]
```

4 Example: Credit card default

Yeh and Lien (2009) and Dheeru and Karra Taniskidou (2017) investigated the predictive accuracy of the probability of default of credit card clients. There are a total of 30,000 observations, 1 response variable, 22 explanatory variables, and no missing values. The response variable is a binary variable that encodes whether the card holder will default on his or her debt, with 0 encoded as "no default" and 1 encoded as "default". Of the 22 explanatory variables, 10 are categorical variables containing information such as gender, education, marital status, and whether past payments have been made on time or delayed. The remaining 12 continuous explanatory variables contain information on the monthly bill amount and payment amount over 6 months. For a complete list of variables, please refer to appendix A.

In this example, we will investigate the predominant factors that affect credit card default prediction accuracy, and we will contrast the prediction accuracies obtained using random forest and logistic regression.

4.1 Model training and parameter tuning

To start the model-training process, we arrange the data points in a randomly sorted order. When the data are split into training and test data, a random sort order ensures that the training data are random as well. To allow for reproducible results, we set a seed value. Then, we split the dataset into two subsets: 50% of the data are used for training, and 50% of the data are used for testing (validation). In small datasets, a 50-50 split may reduce the size of the training data too much; for this relatively large dataset, a 50-50 split is not problematic. The randomization process mentioned previously ensures that the training data contain observations belonging to all available classes as long as the class probabilities are not heavily imbalanced. Additionally, it removes the model's potential dependency on the ordering of observations relative to the test data. Finally, because the variable for marital status uses values 0, 1, 2, and 3 to encode unordered categorical information, we need to create four new binary indicator variables for each marital status using the command `tabulate marriage,`

`generate(marriage_enum)`. Creating the fourth indicator variable is redundant, but this does not matter to tree-based algorithms like `rforest`.

Next, we tune the hyperparameters to find the model with the highest testing accuracy. Specifically, we tune the number of iterations (that is, the number of subtrees) and number of variables to randomly investigate at each split, `numvars()`. The following code segment iteratively calculates the OOB prediction accuracy as a function of the number of iterations and `numvars()`. The number of iterations starts at 10 and is incremented by 5 every time until it reaches 500. We will use both a OOB error (tested against training data subsets that are not included in subtree construction) and a validation error (tested against the testing data) to determine the best possible model.

Usually, tuning parameters in statistical-learning models requires a grid search, that is, an exhaustive search on a user-specified subspace of hyperparameter values. In this case, however, because random forest OOB error rates converge after the number of iterations gets large enough, we simply need to set the iterations to a value large enough for convergence to have occurred prior to tuning the `numvars()` parameter.

To illustrate how the OOB error and validation error have similar trends as the number of iterations grow, we call the random forest function iteratively. The number of iterations variable is initialized to 10 and increments by 5 per function call until it reaches 500. Finally, the trends of OOB error and validation error can be visualized by plotting those values against the number of iterations, as shown in figure 3.

```
. import delimited using "default of credit card clients", varnames(2)
(25 vars, 30,000 obs)
. label define marriage_label 0 missing 1 married 2 single 3 other
. label values marriage marriage_label
. tabulate marriage, generate(marriage_enum)
```

MARRIAGE	Freq.	Percent	Cum.
missing	54	0.18	0.18
married	13,659	45.53	45.71
single	15,964	53.21	98.92
other	323	1.08	100.00
Total	30,000	100.00	

```
. set seed 201807
. generate u=uniform()
. sort u, stable
```

The `stable` option ensures that the result replicates even if there are ties on the sort variable. The number of variables is investigated below; for simplicity, we set `numvars(1)` here.

```
. // figure out how large the value of iterations() need to be
. generate out_of_bag_error1 = .
(30,000 missing values generated)

. generate validation_error = .
(30,000 missing values generated)

. generate iter1 = .
(30,000 missing values generated)

. local j = 0

. forvalues i = 10(5)500 {
  2. local j = `j´ + 1
  3. rforest defaultpaymentnextmonth limit_bal sex
> education marriage_enum* age pay* bill* in 1/15000,
> type(class) iterations(`i´) numvars(1)
  4. quietly replace iter1 = `i´ in `j´
  5. quietly replace out_of_bag_error1 = `e(OOB_Error)´ in `j´
  6. predict p in 15001/30000
  7. quietly replace validation_error = `e(error_rate)´ in `j´
  8. drop p
  9. }

. label variable out_of_bag_error1 "Out-of-bag error"

. label variable iter1 "Iterations"

. label variable validation_error "Validation error"

. scatter out_of_bag_error1 iter1, mcolor(blue) msize(tiny) ||
> scatter validation_error iter1, mcolor(red) msize(tiny)
```

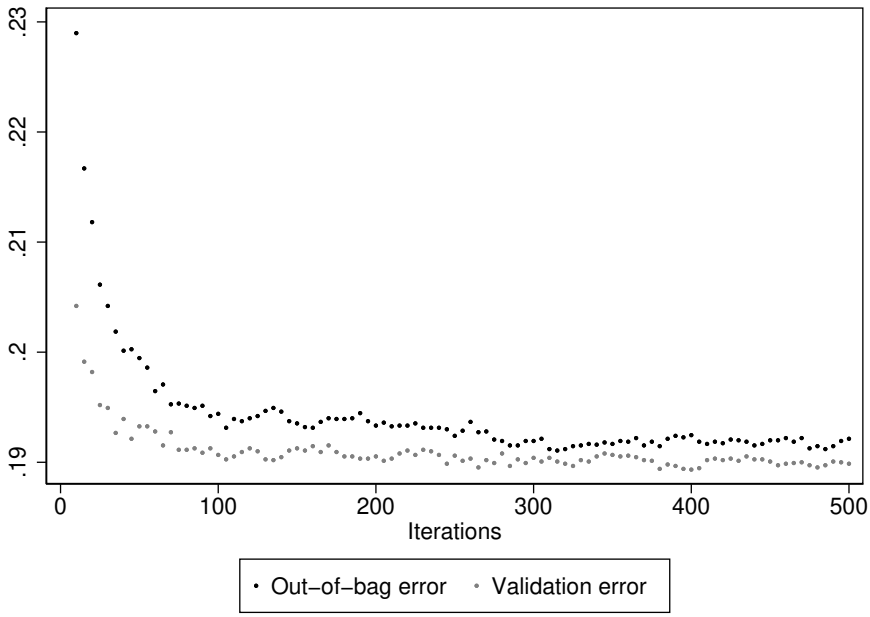

Figure 3. OOB error and validation error versus iterations plot

We can see from figure 3, generated by the above code block, that both the OOB error and the validation error stabilize at around 19%. Hence, fixing the number of iterations at 500 is a good choice.

Next, we can tune the hyperparameter `numvars()`:

```
. generate oob_error = .
(30,000 missing values generated)
. generate nvars = .
(30,000 missing values generated)
. generate val_error = .
(30,000 missing values generated)
. local j = 0
. forvalues i = 1(1)26 {
  2. local j = `j´ + 1
  3. rforest defaultpaymentnextmonth limit_bal sex
>      education marriage_enum* age pay* bill* in 1/15000,
>      type(class) iter(500) numvars(`i´)
  4. quietly replace nvars = `i´ in `j´
  5. quietly replace oob_error = `e(OOB_Error)´ in `j´
  6. predict p in 15001/30000
  7. quietly replace val_error = `e(error_rate)´ in `j´
  8. drop p
  9. }
. label variable oob_error "Out-of-bag error"
. label variable val_error "Validation error"
. label variable nvars "Number of variables randomly selected at each split"
```

```
. scatter oob_error nvars, mcolor(blue) msize(tiny) ||
> scatter val_error nvars, mcolor(red) msize(tiny)
```

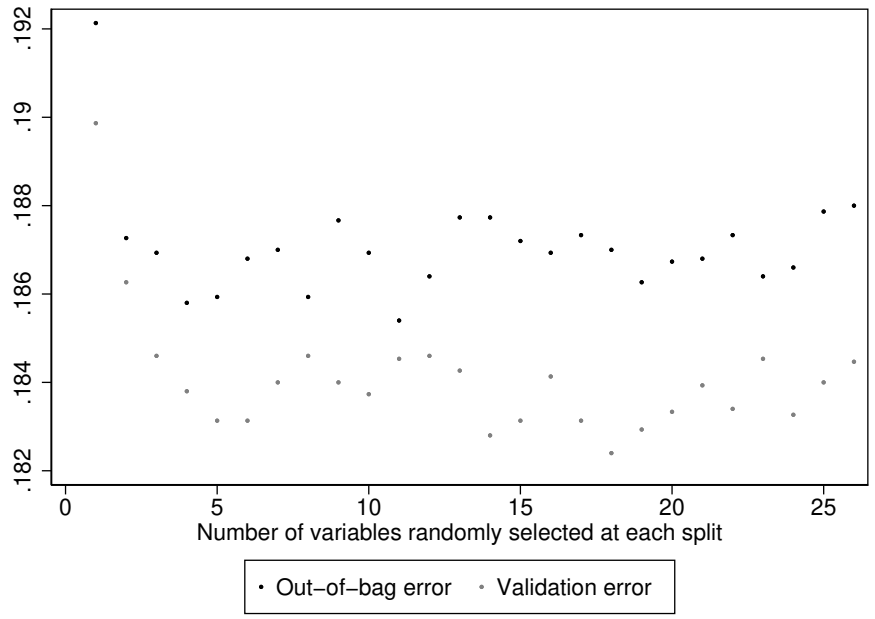

Figure 4. OOB error and validation error versus number of variables plot

In figure 4, we can see for how many variables the minimum error occurs. The following code automates finding the minimum error and the corresponding number of variables. (This code uses frames and requires Stata 16.)

```
. frame put val_error nvars, into(mydata)
. frame mydata {
.    sort val_error, stable
.    local min_val_err = val_error[1]
.    local min_nvars = nvars[1]
. }
. frame drop mydata
. display "Minimum Error: `min_val_err´; Corresponding number of
> variables `min_nvars´"
Minimum Error:    0.1824; Corresponding number of variables 18
```

We can see that at `numvars(18)`, we get the lowest validation error at 0.1824. Hence, we will use `numvars(18)` for our final model.

In principle, the random forest algorithm can output an OOB error at each iteration. However, the Weka implementation of random forest used for the Stata plugin does not output running calculations of OOB error as the algorithm runs and instead only

outputs one final OOB error for the total number of iterations. This means that tuning the iterations parameter requires running the random forest algorithm k times for every value of `iterations(`k`)`. To make this process efficient, we set minimum and maximum values and a reasonable increment to see the trend of the change of OOB error over increasing iterations.

4.2 Final model and interpretation of results

As shown in the previous section, we have set the values of the hyperparameters to be `iterations(500)` and `numvars(18)`. Having reached convergence after 500 iterations, we are free to set the number of iterations even higher. Out of an abundance of caution we set `iterations(1000)`. The following code block gives the final model and prediction error:

```
. // final model: numvars(18) and iterations(1000)
. rforest defaultpaymentnextmonth limit_bal sex education marriage_enum*
> age pay* bill* in 1/15000, type(class) iterations(1000) numvars(18)
. display e(OOB_Error)
.18666667
. predict prf in 15001/30000
. display e(error_rate)
.18253333
```

The final OOB error is 18.25%, which is larger than the actual prediction error, which is 18.24%, calculated over 15,000 test observations. We can see from both figure 3 and figure 4 that the OOB error and the validation error have the same pattern when plotted against the two hyperparameters, which are iterations and number of variables.

We also would like to ascertain which factors are the most important in the prediction process. Random forests are black boxes in that they do not offer insight on how the predictions are accomplished. The variable-importance scores of each predictor provide some limited insight. The following code segment plots the variable importance:

```
. // variable importance plot
. matrix importance = e(importance)
. svmat importance
. generate importid=""
(30,000 missing values generated)
. local mynames: rownames importance
. local k: word count `mynames´
. if `k´>_N {
.      set obs `k´
. }
. forvalues i = 1(1)`k´ {
  2. local aword: word `i´ of `mynames´
  3. local alabel: variable label `aword´
  4. if ("`alabel´"!="") qui replace importid= "`alabel´" in `i´
  5. else qui replace importid= "`aword´" in `i´
  6. }
```

```
. graph hbar (mean) importance, over(importid, sort(1) label(labsize(2)))
> ytitle(Importance)
```

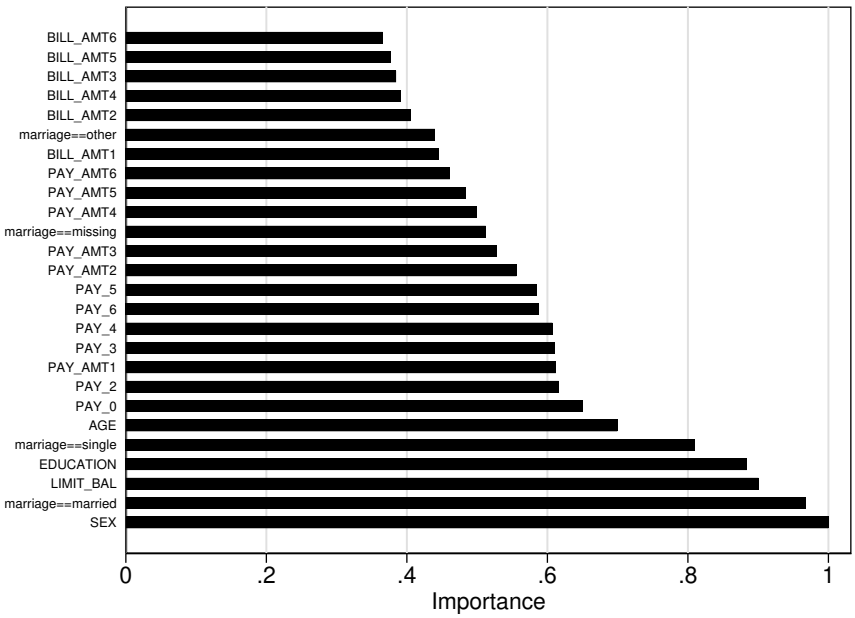

Figure 5. Importance scores of predictor variables

We can see from figure 5 that the five most important predictors are basic demographic and background information such as gender, education, and marital status ("married" and "single") as well as the monthly spending limit (`limit_bal`). We can also see that none of the variables encoding monthly bill amounts (`bill_amt`) is particularly important, compared with the rest of the predictors. Surprisingly, however, the amount of monthly spending limit (`limit_bal`) is the third most important predictor in the random forest model. We can overlay two histograms of the monthly spending limit to obtain more insight on how this variable affects the response variable:

```
. twoway (hist limit_bal if defaultpaymentnextmonth == 0)
> (hist limit_bal if defaultpaymentnextmonth == 1,
> fcolor(none) lcolor(black)), legend(order(1 "no default" 2 "default" ))
```

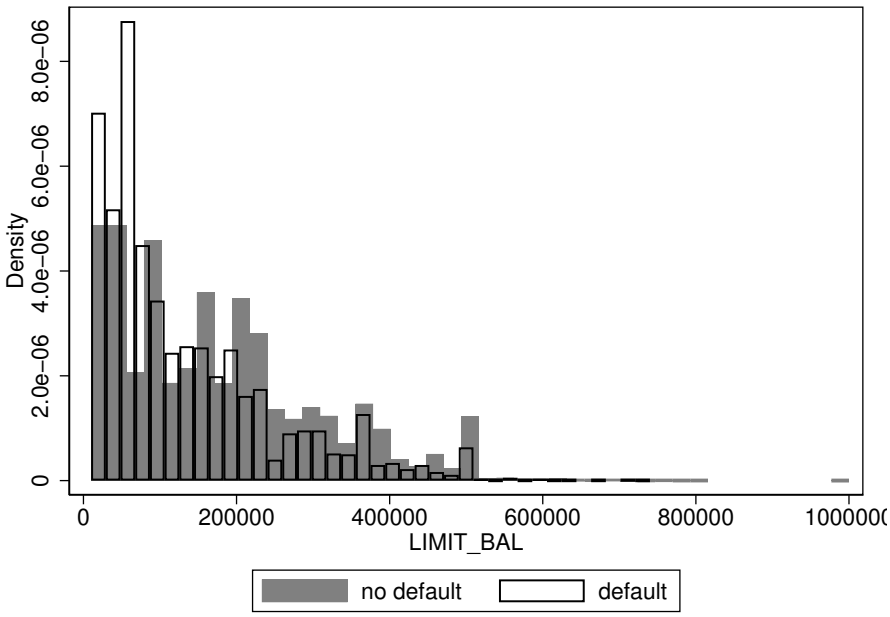

Figure 6. Histograms of monthly spending limit

We can see from the histograms in figure 6 that card holders who default on their debt generally have a lower monthly spending limit than those who do not default. Variable importance measures the contribution of an x variable to the model but depends on the set of x variables. Another x variable correlated with the first would rise in importance if the first x variable was excluded.

4.3 Comparison with logistic regression

Alternatively, credit card debt default can be modeled using logistic regression. The following code returns the prediction accuracy of logistic regression using the same set of predictor variables and the same train-and-test split:

```
. logistic defaultpaymentnextmonth limit_bal sex education
> marriage_enum* age pay* bill* in 1/15000
note: marriage_enum4 omitted because of collinearity
Logistic regression                             Number of obs     =     15,000
                                                LR chi2(25)       =    1910.25
                                                Prob > chi2       =     0.0000
Log likelihood = -6962.3913                     Pseudo R2         =     0.1206
```

defaultpaym~h	Odds Ratio	Std. Err.	z	P>|z|	[95% Conf.	Interval]
limit_bal	.9999994	2.20e-07	-2.79	0.005	.999999	.9999998
sex	.8492577	.0368609	-3.76	0.000	.7799994	.9246657
education	.9141707	.027126	-3.02	0.002	.8625212	.968913
marriage_en~1	.417391	.3057084	-1.19	0.233	.0993346	1.753824
marriage_en~2	1.003318	.1921736	0.02	0.986	.6892883	1.460414
marriage_en~3	.8158847	.1579004	-1.05	0.293	.5583331	1.192241
marriage_en~4	1	(omitted)				
age	1.008542	.0025965	3.30	0.001	1.003466	1.013644
pay_0	1.803168	.0448785	23.69	0.000	1.717319	1.893309
pay_2	1.065803	.030625	2.22	0.027	1.007438	1.127549
pay_3	1.034829	.0336037	1.05	0.292	.9710189	1.102832
pay_4	1.045964	.037159	1.26	0.206	.9756117	1.12139
pay_5	1.065612	.0404813	1.67	0.094	.9891523	1.147983
pay_6	.9714182	.0303388	-0.93	0.353	.9137387	1.032739
pay_amt1	.999986	3.23e-06	-4.35	0.000	.9999796	.9999923
pay_amt2	.9999883	2.90e-06	-4.06	0.000	.9999826	.9999939
pay_amt3	.9999947	2.59e-06	-2.03	0.042	.9999897	.9999998
pay_amt4	.999994	2.51e-06	-2.41	0.016	.999989	.9999989
pay_amt5	.9999973	2.61e-06	-1.03	0.304	.9999922	1.000002
pay_amt6	.9999966	1.94e-06	-1.76	0.079	.9999928	1
bill_amt1	.9999938	1.66e-06	-3.73	0.000	.9999906	.9999971
bill_amt2	1.000001	2.17e-06	0.60	0.549	.999997	1.000006
bill_amt3	1.000003	1.85e-06	1.36	0.173	.9999989	1.000006
bill_amt4	.9999999	1.93e-06	-0.05	0.959	.9999961	1.000004
bill_amt5	1.000005	2.09e-06	2.20	0.028	1.000001	1.000009
bill_amt6	.9999979	1.59e-06	-1.33	0.184	.9999948	1.000001
_cons	.4392684	.1048452	-3.45	0.001	.2751464	.7012873

```
Note: _cons estimates baseline odds.
Note: 3 failures and 0 successes completely determined.
. predict plogit in 15001/30000
(option pr assumed; Pr(defaultpaymentnextmonth))
(15,000 missing values generated)
. replace plogit = 0 if plogit <= 0.5 & plogit != .
(13,896 real changes made)
. replace plogit = 1 if plogit > 0.5 & plogit != .
(1,104 real changes made)
. generate error = plogit != defaultpaymentnextmonth
. summarize error in 15001/30000
```

Variable	Obs	Mean	Std. Dev.	Min	Max
error	15,000	.1886	.3912036	0	1

The prediction error obtained using logistic regression is 18.86%, compared with the best-so-far error rate that we have from random forest, which is 18.25%. The difference in error rate is small but might still be meaningful to prevent credit card defaults.

5 Example: Online news popularity

Fernandes et al. (2015) and Dheeru and Karra Taniskidou (2017) investigated the popularity of online news.[2] The data were originally presented at a Portuguese conference on artificial intelligence in 2015. There are a total of 39,644 observations, 1 response variable, and 58 explanatory variables. For this problem, we are interested in the log-scaled number of "shares" an online article obtains based on various nominal and continuous attributes such as whether the article was published on a weekend, whether certain keywords are present, number of images in the article, etc. For a full list of variable names and descriptions, please refer to appendix B.

5.1 Model training and parameter tuning

First, we need to randomize the data as we did for the previous classification example. Then, we generate a new variable for the log-scaled number of shares:

```
. import delimited OnlineNewsPopularity.csv
(61 vars, 39,644 obs)
. set seed 201807
. generate u = runiform()
. sort u, stable
. generate logShares = ln(shares)
```

We will use a 50-50 split to partition the data into training and testing sets as in the previous example. To tune the hyperparameters `numvars()` and `iterations()`, we use the same technique as the previous example, where we fix the value of one hyperparameter when tuning the other. This is a viable parameter-optimization method that results from the error rate for random forest converging when the number of iterations is large enough. Essentially, our goal is to set a reasonably large number of iterations where the OOB and validation errors converge so that when we tune the number of randomly selected variables, we can ascertain that the errors differ because of the value of `numvars()` and not because of `iterations()`. We will again start with `iterations(10)` and increase it by increments of 5 until `iterations(100)`, which is approximately the highest possible value with which one can run this dataset on a CPU because of constraints on runtime memory. At the end of the loop, we plot the OOB errors and the actual root mean squared error (RMSE) values validated using the test data against the number of iterations.

2. To access the exact dataset used in this example, please visit https://archive.ics.uci.edu/ml/datasets/Online+News+Popularity

```
. generate out_of_bag_error1 = .
(39,644 missing values generated)
. generate validation_error = .
(39,644 missing values generated)
. generate iter1 = .
(39,644 missing values generated)
. local j = 0
. forvalues i = 10(5)100 {
  2. local j = `j´ + 1
  3. rforest logShares n_* average_* num_*
>      data_* kw_* self_* weekday_* lda_* global_*
>      is_weekend rate_* min_* max_* avg_* title_* abs_* in 1/19822,
>      type(reg) iterations(`i´) numvars(1)
  4. quietly replace iter1 = `i´ in `j´
  5. quietly replace out_of_bag_error1 = `e(OOB_Error)´ in `j´
  6. predict p in 19823/39644
  7. quietly replace validation_error = `e(RMSE)´ in `j´
  8. drop p
  9. }
. label variable out_of_bag_error1 "Out-of-bag error"
. label variable iter1 "Iterations"
. label variable validation_error "Validation RMSE"
. scatter out_of_bag_error1 iter1, mcolor(blue) msize(tiny) ||
> scatter validation_error iter1, mcolor(red) msize(tiny)
```

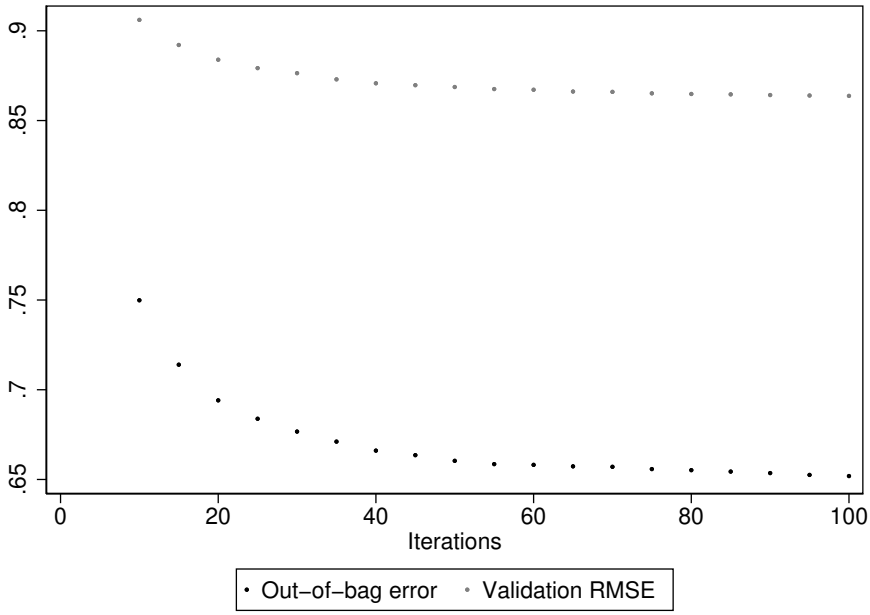

Figure 7. OOB error and validation RMSE versus iterations plot

We can see from the graph that the OOB error and validation RMSE start to converge around 80 iterations. We get the lowest value for both errors at 100 iterations, which will be used for the final model. Now we can tune the other hyperparameter, `numvars()`, to see which one gives the lowest validation RMSE.

```
. generate oob_error = .
(39,644 missing values generated)
. generate nvars = .
(39,644 missing values generated)
. generate val_error = .
(39,644 missing values generated)
. local j = 0
. forvalues i = 1(1)58 {
  2. local j = `j´ + 1
  3. rforest logShares n_* average_* num_*
>      data_* kw_* self_* weekday_* lda_* global_*
>      is_weekend rate_* min_* max_* avg_* title_* abs_* in 1/19822,
>      type(reg) iterations(100) numvars(`i´)
  4. quietly replace nvars = `i´ in `j´
  5. quietly replace oob_error = `e(OOB_Error)´ in `j´
  6. predict p in 19823/39644
  7. quietly replace val_error = `e(RMSE)´ in `j´
  8. drop p
  9. }
. label variable oob_error "Out-of-bag error"
. label variable val_error "Validation RMSE"
. label variable nvars "Number of variables randomly selected at each split"
```

```
. scatter oob_error nvars, mcolor(blue) msize(tiny) ||
> scatter val_error nvars, mcolor(red) msize(tiny)
```

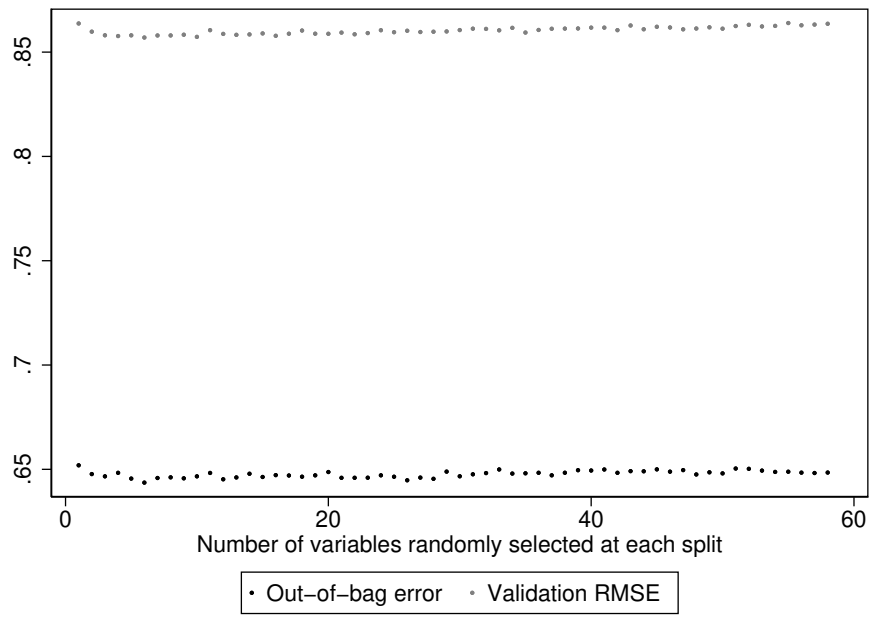

Figure 8. OOB error and validation error versus number of variables plot

Again, we automate finding the minimum error:

```
. capture frame drop mydata2
. // run only when tuning is run
. frame put val_error nvars, into(mydata2)
. frame mydata2 {
.    sort val_error, stable
.    local min_val_err = val_error[1]
.    local min_nvars = nvars[1]
. }
. frame drop mydata2
. display "Minimum Error: `min_val_err´; Corresponding number of
> variables `min_nvars´"
Minimum Error:    0.8570; Corresponding number of variables 6
```

For `numvars(6)`, we get the lowest validation error at 0.8570. Hence, we will use `numvars(6)` for our final model. For this dataset, the model is fairly robust to changes in the number of variables, `numvars()`, and `numvars(6)` has only a slight edge compared with other values. This might not always be the case.

5.2 Final model and interpretation of results

The final model has hyperparameter values `numvars(6)` and `iterations(100)`.

```
. rforest logShares n_* average_* num_*
>      data_* kw_* self_* weekday_* lda_* global_*
>      is_weekend rate_* min_* max_* avg_* title_* abs_* in 1/19822,
>      type(reg) iterations(100) numvars(6)

. ereturn list OOB_Error
scalar e(OOB_Error)=  .6436290493772533

. predict prf in 19823/39644

. ereturn list RMSE
scalar e(RMSE)     =  .8570009318991625
```

The final OOB error is 0.6436. This is slightly lower than the RMSE calculated against the testing data, which is 0.8570. To learn which variables affect the prediction accuracy, we can generate a variable-importance plot using the same code segment as the previous classification example. For readability, only variables with an importance score of at least 40% as large as that of the most important variable are shown.

```
. // variable importance plot
. matrix importance2 = e(importance)

. svmat importance2

. generate importid2=""
(39,644 missing values generated)

. local mynames: rownames importance2

. local k: word count `mynames´

. if `k´>_N {
.      set obs `k´
. }

. forvalues i = 1(1)`k´ {
  2.      local aword: word `i´ of `mynames´
  3.      local alabel: variable label `aword´
  4.      if ("`alabel´"!="") qui replace importid2= "`alabel´" in `i´
  5.      else qui replace importid2= "`aword´" in `i´
  6. }
```

```
. graph hbar (mean) importance2 if importance2>.4, over(importid2, sort(1)
> label(labsize(2))) ytitle(Importance)
```

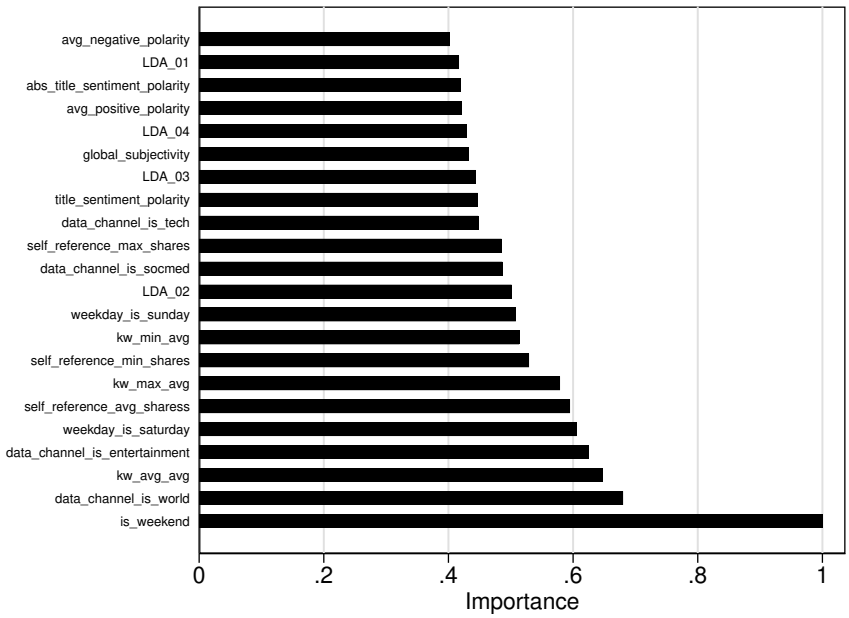

Figure 9. Importance score of predictor variables

Whether the article was published on a weekend is the most important predictor. Other important explanatory variables include news channel types and the number of keywords. To obtain more insight on how the log-scaled number of article shares is related to whether the article was published on a weekend, we use the following histogram to illustrate the relationship:

```
. twoway (hist logShares if is_weekend == 0)
>        (hist logShares if is_weekend == 1, fcolor(none) lcolor(black)),
>        legend(order(1 "weekday" 2 "weekend" ))
```

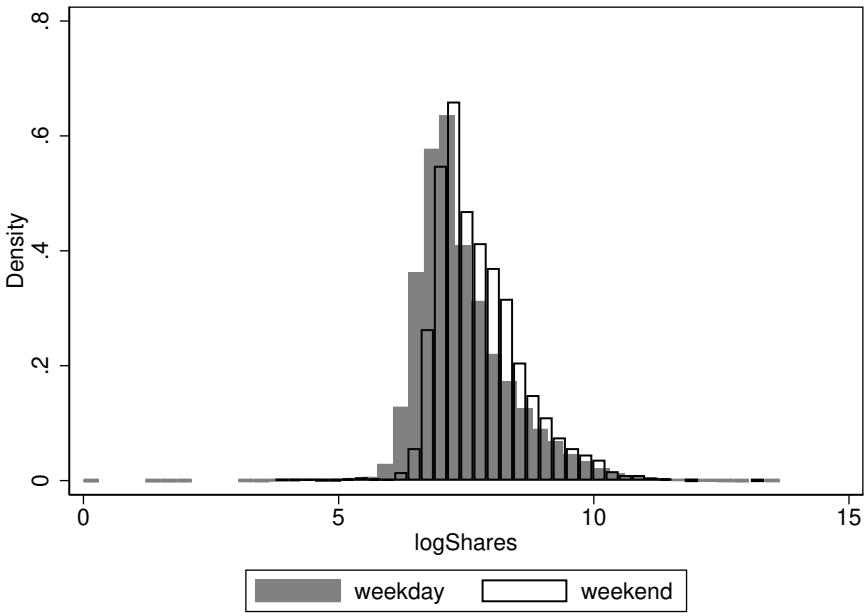

Figure 10. Histograms of log-scaled number of shares

The empirical distributions of log number of shares differ for weekdays versus weekends. This clear shift in empirical distribution helps to explain why the `is_weekend` explanatory variable was the most important in the model.

5.3 Comparison with linear regression

The following code block fits a linear regression model over the same set of dependent and independent variables using the same train-and-test split as shown in the random forest model:

```
. regress logShares n_* average_* num_*
> data_* kw_* self_* weekday_* lda_* global_*
> is_weekend rate_* min_* max_* avg_* title_* abs_* in 1/19822
note: weekday_is_friday omitted because of collinearity
note: weekday_is_saturday omitted because of collinearity
note: lda_01 omitted because of collinearity
```

Source	SS	df	MS		
Model	2257.55675	55	41.0464864	Number of obs	= 19,822
Residual	14963.6848	19,766	.757041628	F(55, 19766)	= 54.22
				Prob > F	= 0.0000
				R-squared	= 0.1311
				Adj R-squared	= 0.1287
Total	17221.2416	19,821	.86883818	Root MSE	= .87008

logShares	Coef.	Std. Err.	t	P>|t|	[95% Conf.	Interval]
n_tokens_ti~e	.0075709	.0030727	2.46	0.014	.0015482	.0135937
n_tokens_co~t	.0000896	.0000241	3.71	0.000	.0000422	.0001369
n_unique_to~s	.3790955	.2060237	1.84	0.066	-.0247283	.7829193
n_non_stop~ds	.8239396	.8831114	0.93	0.351	-.9070329	2.554912
n_non_stop~ns	-.3543805	.1750817	-2.02	0.043	-.6975553	-.0112057
average_tok~h	-.093957	.0258385	-3.64	0.000	-.1446027	-.0433113
num_hrefs	.0036305	.000706	5.14	0.000	.0022467	.0050144
num_self_hr~s	-.0073054	.0018167	-4.02	0.000	-.0108662	-.0037445
num_imgs	.0015477	.0009738	1.59	0.112	-.0003609	.0034564
num_videos	.0017068	.0017468	0.98	0.329	-.001717	.0051307
num_keywords	.005025	.00398	1.26	0.207	-.0027762	.0128261
data_channe~e	-.1193581	.0422913	-2.82	0.005	-.2022525	-.0364636
data_channe~t	-.2102381	.0273279	-7.69	0.000	-.263803	-.1566732
data_channe~s	-.1828533	.0412715	-4.43	0.000	-.2637489	-.1019577
data_chann~ed	.1076191	.039703	2.71	0.007	.0297978	.1854404
data_channe~h	.0696772	.0399362	1.74	0.081	-.0086011	.1479554
data_chann~ld	-.0547657	.0402811	-1.36	0.174	-.13372	.0241886
kw_min_min	.0008308	.0001722	4.82	0.000	.0004933	.0011684
kw_max_min	1.56e-06	6.26e-06	0.25	0.803	-.0000107	.0000138
kw_avg_min	9.84e-06	.0000391	0.25	0.802	-.0000669	.0000866
kw_min_max	-2.24e-07	1.26e-07	-1.78	0.076	-4.72e-07	2.33e-08
kw_max_max	3.50e-08	6.16e-08	0.57	0.570	-8.57e-08	1.56e-07
kw_avg_max	-2.92e-07	8.87e-08	-3.30	0.001	-4.66e-07	-1.18e-07
kw_min_avg	-.000054	8.12e-06	-6.65	0.000	-.0000699	-.0000381
kw_max_avg	-.0000451	2.75e-06	-16.40	0.000	-.0000505	-.0000397
kw_avg_avg	.0003413	.0000155	22.04	0.000	.000311	.0003717
self~n_shares	2.31e-07	7.78e-07	0.30	0.767	-1.30e-06	1.76e-06
self~x_shares	-3.93e-07	4.39e-07	-0.90	0.370	-1.25e-06	4.67e-07
self_refer~ss	2.48e-06	1.10e-06	2.26	0.024	3.28e-07	4.64e-06
weekday~onday	-.0140188	.0222015	-0.63	0.528	-.0575357	.029498
weekda~uesday	-.0813934	.0217944	-3.73	0.000	-.1241122	-.0386746
weekda~nesday	-.074833	.021556	-3.47	0.001	-.1170846	-.0325814
weekday~rsday	-.0582327	.0218933	-2.66	0.008	-.1011453	-.01532
weekday_~iday	0	(omitted)				
weekday_~rday	0	(omitted)				
weekday~unday	.0162751	.0340215	0.48	0.632	-.0504099	.0829602
lda_00	.3737897	.0569696	6.56	0.000	.2621246	.4854548
lda_01	0	(omitted)				
lda_02	-.1065375	.0557763	-1.91	0.056	-.2158638	.0027888
lda_03	.0406036	.0395897	1.03	0.305	-.0369955	.1182027
lda_04	.1717159	.0542922	3.16	0.002	.0652987	.2781332
global_subj~y	.3763543	.0916398	4.11	0.000	.1967326	.5559761
global_sent~y	.0136587	.1804347	0.08	0.940	-.3400085	.367326
g~itive_words	-.8139646	.7785847	-1.05	0.296	-2.340056	.7121268

```
g~ative_words |   .1631068   1.528481     0.11   0.915    -2.832844    3.159058
   is_weekend |   .2033014   .0296855     6.85   0.000     .1451154    .2614874
rate_positi~s |  -.4930114    .872509    -0.57   0.572    -2.203202     1.21718
rate_negati~s |  -.6051804   .8763014    -0.69   0.490    -2.322805    1.112444
min_positiv~y |  -.4281182   .1223362    -3.50   0.000    -.6679075    -.188329
min_negativ~y |  -.0100648    .049342    -0.20   0.838    -.1067792    .0866497
max_positiv~y |  -.0575846   .0461163    -1.25   0.212    -.1479764    .0328072
max_negativ~y |   .0240158    .111635     0.22   0.830    -.1947981    .2428298
avg_positiv~y |   .0174634    .147251     0.12   0.906    -.2711609    .3060877
avg_negativ~y |   -.076345   .1351286    -0.56   0.572    -.3412084    .1885185
title_subje~y |   .0581885   .0293427     1.98   0.047     .0006744    .1157026
title_senti~y |   .0597084   .0265768     2.25   0.025     .0076156    .1118011
abs_titl~vity |   .1600177   .0391203     4.09   0.000     .0833386    .2366968
abs_titl~rity |   .0400174   .0419938     0.95   0.341    -.0422941    .1223288
        _cons |   6.552971    .088802    73.79   0.000     6.378912     6.72703
------------------------------------------------------------------------------
```

```
. predict pregress in 19823/39644
(option xb assumed; fitted values)
(19,822 missing values generated)

. ereturn list rmse
scalar e(rmse)     =  .8700813917009586
```

The value of `e(rmse)` displayed is the RMSE calculated over the training data. To compare the linear model with the random forest model, we need to calculate the RMSE over the testing data using the following commands:

```
. generate diff_sqr= (logShares - pregress)^2
(19,822 missing values generated)

. summarize diff_sqr
    Variable |        Obs        Mean    Std. Dev.       Min        Max
-------------+---------------------------------------------------------
    diff_sqr |     19,822    40.90379    5651.692   1.02e-09     795706
```

We can see from the output that the mean squared error is 40.90379, which means the RMSE is equal to $\sqrt{40.90379} \approx 6.3956$, which is much higher than the RMSE fitted over the training data. Comparing with the testing RMSE obtained from the random forest model, the testing RMSE for the linear model is much higher. This is a strong indication that random forest outperforms linear regression for this example.

6 Discussion

The classification and regression examples have illustrated that random forest models usually have higher prediction accuracy than corresponding parametric models such as logistic regression and linear regression. Typically, greater gains in model performance are available for multiclass (multinomial) outcomes and regression than binary outcomes. Misclassification is a fairly insensitive performance criterion. When an improved algorithm changes the estimated classification probabilities for two classes from $p_1 = 0.10$ and $p_2 = 0.90$ to $p_1 = 0.40$ and $p_2 = 0.60$ for an observation, the resulting classification remains the same. An improvement over logistic regression with its linearity assumption can come either from nonlinearities or from interactions. Additionally, the scope of improvement is reduced when many of the variables are indicator

variables; nonlinearities do not exist for indicator variables. In our experience, many of the variables in social sciences are indicator variables. For example, Ing et al. (2019) found that support-vector machines did not improve over logistic regression. Similarly, in our classification example, the improvement of random forest over logistic regression was minor.

In the examples, the values of hyperparameters were determined based on which value gave the lowest testing error. In practice, when there are not enough observations to allow for a train-and-test split, the OOB error can be used instead. As previously demonstrated, the OOB error is a close estimation of the actual testing error and can be used on its own as a criterion for parameter tuning.

While the two examples primarily focused on the typical case of tuning the options `iterations()` and `numvars()`, depending on the dataset and software constraints, other hyperparameters such as max tree depth and minimum size of leaf nodes could be taken into consideration during parameter tuning. For instance, setting the max tree depth to a fixed value may become necessary on a machine with limited RAM.

7 Acknowledgments

The software development in Stata was built on top of the Weka Java implementation, which was developed by the University of Waikato. We are grateful to Eibe Frank for allowing us to use the Weka implementation for the plugin.

This research was supported by the Social Sciences and Humanities Research Council of Canada (# 435-2013-0128).

8 Programs and supplemental materials

To install a snapshot of the corresponding software files as they existed at the time of publication of this article, type

```
. net sj 20-1
. net install st0587     (to install program files, if available)
. net get st0587         (to install ancillary files, if available)
```

9 References

Basuchoudhary, A., J. T. Bang, and T. Sen. 2017. *Machine-Learning Techniques in Economics: New Tools for Predicting Economic Growth*. New York: Springer.

Breiman, L. 2001. Random forests. *Machine Learning* 45: 5–32.

Dheeru, D., and E. Karra Taniskidou. 2017. Default of credit card clients dataset. https://www.kaggle.com/uciml/default-of-credit-card-clients-dataset.

Fernandes, K., P. Vinagre, and P. Cortez. 2015. A proactive intelligent decision support system for predicting the popularity of online news. In *Progress in Artificial Intelli-*

gence: 17th Portuguese Conference on Artificial Intelligence, EPIA 2015, Coimbra, Portugal, September 8–11, 2015. Proceedings, 535–546. New York: Springer.

Hall, M., E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. 2009. The WEKA data mining software: An update. *ACM SIGKDD Explorations Newsletter* 11(1): 10–18.

Ho, T. K. 1995. Random decision forests. In *Proceedings of 3rd International Conference on Document Analysis and Recognition*, 278–282. Piscataway, NJ: IEEE.

Ing, E., W. Su, M. Schonlau, and N. Torun. 2019. Support vector machines and logistic regression to predict temporal artery biopsy outcomes. *Canadian Journal of Ophthalmology* 54: 116–118.

Liu, X., D. Wu, G. K. Zewdie, L. Wijerante, C. I. Timms, A. Riley, E. Levetin, and D. J. Lary. 2017. Using machine learning to estimate atmospheric Ambrosia pollen concentrations in Tulsa, OK. *Environmental Health Insights* 11: 1–10.

Nyman, R., and P. Ormerod. 2017. Predicting economic recessions using machine learning algorithms. ArXiv Working Paper No. arXiv:1701.01428. https://arxiv.org/abs/1701.01428.

Shannon, C. E. 2001. A mathematical theory of communication. *ACM SIGMOBILE Mobile Computing and Communications Review* 5: 3–55.

Witten, I. H., E. Frank, M. A. Hall, and C. J. Pal. 2016. The WEKA workbench online appendix. In *Data Mining: Practical Machine Learning Tools and Techniques*, 4th ed. Burlington, MA: Morgan Kaufmann.

Yeh, I.-C., and C.-H. Lien. 2009. The comparisons of data mining techniques for the predictive accuracy of probability of default of credit card clients. *Expert Systems with Applications* 36: 2473–2480.

About the authors

Matthias Schonlau is a professor of statistics at the University of Waterloo, Canada. His interests include survey methodology and learning from text data in the context of open-ended questions.

Rosie Yuyan Zou is a recent B.CS graduate from the University of Waterloo. She will be joining Apple, Inc. as a software design engineer. Her academic passions include applied machine learning and software system design for digital hardware.

A Variable names for classification example

The column names from the variables `limit_bal` through `defaultpaymentnextmonth` appear as they do in the original documentation on UCI Machine Learning Repository's website.

Variable names	Column names
`id`	Row number
`limit_bal`	Amount of the given credit (NT dollar): it includes both the individual consumer credit and his/her family (supplementary) credit
`sex`	Gender (1 = male, 2 = female)
`education`	Education (1 = graduate school; 2 = university; 3 = high school; 4 = others)
`marriage`	Marital status (1 = married; 2 = single; 3 = others)
`age`	Age (year)
`pay_0`[3]	the repayment status in September, 2005
`pay_2`	the repayment status in August, 2005
`pay_3`	the repayment status in July, 2005
`pay_4`	the repayment status in June, 2005
`pay_5`	the repayment status in May, 2005
`pay_6`	the repayment status in April, 2005
`bill_amt1`	amount of bill statement in September, 2005
`bill_amt2`	amount of bill statement in August, 2005
`bill_amt3`	amount of bill statement in July, 2005
`bill_amt4`	amount of bill statement in June, 2005
`bill_amt5`	amount of bill statement in May, 2005
`bill_amt6`	amount of bill statement in April, 2005
`pay_amt1`	amount of previous payment in September, 2005
`pay_amt2`	amount of previous payment in August, 2005
`pay_amt3`	amount of previous payment in July, 2005
`pay_amt4`	amount of previous payment in June, 2005
`pay_amt5`	amount of previous payment in May, 2005
`pay_amt6`	amount of previous payment in April, 2005
`defaultpaymentnextmonth`	default payment (Yes = 1, No = 0), as the response variable
`marriage_enum1`	Marital status is not "married", "single", or "other"; generated during data preprocessing
`marriage_enum2`	Marital status = "married"; generated during data preprocessing
`marriage_enum3`	Marital status = "single"; generated during data preprocessing
`marriage_enum4`	Marital status = "other"; generated during data preprocessing

3. The values of the variables `pay_0`–`pay_6` correspond to number of months delayed.

B Variable names for regression example

The column names in this table are reproduced based on the original documentation on UCI Machine Learning Repository's website.

Variable names	Column names
`url`	URL of the article (non-predictive)
`timedelta`	Days between the article publication and the dataset acquisition (non-predictive)
`n_tokens_title`	Number of words in the title
`n_tokens_content`	Number of words in the content
`n_unique_tokens`	Rate of unique words in the content
`n_non_stop_words`	Rate of non-stop words in the content
`n_non_stop_unique_tokens`	Rate of unique non-stop words in the content
`num_hrefs`	Number of links
`num_self_hrefs`	Number of links to other articles published by Mashable
`num_imgs`	Number of images
`num_videos`	Number of videos
`average_token_length`	Average length of the words in the content
`num_keywords`	Number of keywords in the metadata
`data_channel_is_lifestyle`	Is data channel 'Lifestyle'?
`data_channel_is_entertainment`	Is data channel 'Entertainment'?
`data_channel_is_bus`	Is data channel 'Business'?
`data_channel_is_socmed`	Is data channel 'Social Media'?
`data_channel_is_tech`	Is data channel 'Tech'?
`data_channel_is_world`	Is data channel 'World'?
`kw_min_min`	Worst keyword (min. shares)
`kw_max_min`	Worst keyword (max. shares)
`kw_avg_min`	Worst keyword (avg. shares)
`kw_min_max`	Best keyword (min. shares)
`kw_max_max`	Best keyword (max. shares)
`kw_avg_max`	Best keyword (avg. shares)
`kw_min_avg`	Avg. keyword (min. shares)
`kw_max_avg`	Avg. keyword (max. shares)
`kw_avg_avg`	Avg. keyword (avg. shares)
`self_reference_min_shares`	Min. shares of referenced articles in Mashable
`self_reference_max_shares`	Max. shares of referenced articles in Mashable
`self_reference_avg_sharess`	Avg. shares of referenced articles in Mashable
`weekday_is_monday`	Was the article published on a Monday?
`weekday_is_tuesday`	Was the article published on a Tuesday?
`weekday_is_wednesday`	Was the article published on a Wednesday?
`weekday_is_thursday`	Was the article published on a Thursday?
`weekday_is_friday`	Was the article published on a Friday?

Continued on next page

Variable names	Column names
weekday_is_saturday	Was the article published on a Saturday?
weekday_is_sunday	Was the article published on a Sunday?
is_weekend	Was the article published on the weekend?
LDA_00	Closeness to LDA topic 0
LDA_01	Closeness to LDA topic 1
LDA_02	Closeness to LDA topic 2
LDA_03	Closeness to LDA topic 3
LDA_04	Closeness to LDA topic 4
global_subjectivity	Text subjectivity
global_sentiment_polarity	Text sentiment polarity
global_rate_positive_words	Rate of positive words in the content
global_rate_negative_words	Rate of negative words in the content
rate_positive_words	Rate of positive words among non-neutral tokens
rate_negative_words	Rate of negative words among non-neutral tokens
avg_positive_polarity	Avg. polarity of positive words
min_positive_polarity	Min. polarity of positive words
max_positive_polarity	Max. polarity of positive words
avg_negative_polarity	Avg. polarity of negative words
min_negative_polarity	Min. polarity of negative words
max_negative_polarity	Max. polarity of negative words
title_subjectivity	Title subjectivity
title_sentiment_polarity	Title polarity
abs_title_subjectivity	Absolute subjectivity level
abs_title_sentiment_polarity	Absolute polarity level
shares	Number of shares (target)