Lecture 13

Improving Model Performance

Parameter tuning

ML models might have different parameters that can affect their performance. For example, the performance of kNN models depends on *k* and performance optimization requires that we find the best value of *k*. Similarly, neural networks must be tuned by adjusting the number of nodes or hidden layers. Another example is the *trial* parameter in C5.0, which specifies the maximum number of decision trees to be built and used for voting. The idea is to build many decision trees and to take a vote on the best class for each example. In Lecture 7, we increased the model's accuracy by using 10 trials, i.e., a maximum of 10 decision trees.

The process of adjusting the model parameters to identify the best fit is called **parameter tuning.**

Manual parameter tuning could be extremely laborious as it requires fitting models for different parameter values and comparing their performance. The tuning could be further exacerbated if a model involves several parameters. Then, we need to search for a large number of different combinations of parameter values.

Automated parameter tuning

Fortunately, the *caret* library provides exceptional tools for automatic parameter tuning and training of classifiers. The core functionality is provided by *train()* and *predict()* functions that serve as a standardized interface for over 175 different machine learning models for both classification and regression tasks. By using this function, it is possible to automate the search for optimal models using a choice of evaluation methods and metrics. The table below shows some models that are supported by *caret* and their corresponding parameters that can be automatically tuned.

Model	Learning Task	Method name	Parameters
k-Nearest Neighbors	Classification	`knn`	`k`
Naive Bayes	Classification	`nb`	`fL, usekernel`
Decision Trees	Classification	`C5.0`	`model, trials, winnow`
OneR Rule Learner	Classification	`OneR`	None
RIPPER Rule Learner	Classification	`JRip`	`NumOpt`
Linear Regression	Regression	`lm`	None
Regression Trees	Regression	`rpart`	`cp`
Model Trees	Regression	`M5`	`pruned, smoothed, rules`
Neural Networks	Dual use	`nnet`	`size, decay`
Support Vector Machines (Linear Kernel)	Dual use	`svmLinear`	`C`
Support Vector Machines (Radial Basis Kernel)	Dual use	`svmRadial`	`C, sigma`
Random Forests	Dual use	`rf`	`mtry`

Automated parameter tuning

The modelLookup() function provides the tuning parameters for a particular model:

```
> library(caret)
> modelLookup("C5.0")
  model parameter                   label forReg forClass probModel
1  C5.0    trials # Boosting Iterations  FALSE     TRUE       TRUE
2  C5.0     model            Model Type  FALSE     TRUE       TRUE
3  C5.0    winnow                Winnow  FALSE     TRUE       TRUE
```

The output shows that there are three C5.0 parameters that can be automatically tuned: trials, model, and winnow. The model parameter has two values, tree and rules, that specify the type of the model. The whinnow parameter can be either TRUE or FALSE. Winnowing is the process of removing unimportant attributes in C5.0. When the number of attributes is large not all of them may provide important predictive information. C5.0 has an option (or a parameter) to winnow or remove predictors: the algorithm uncovers which predictors have a relationship with the outcome, and the final model is created from only the important predictors.

Automated parameter tuning

The kNN method has only parameter to be automatically tuned:

```
> modelLookup("knn")
  model parameter      label forReg forClass probModel
1   knn         k #Neighbors   TRUE     TRUE      TRUE
```

Automatic tuning is based on a matrix of all possible parameter combinations, called **grid**. Because of the large number of combinations, it is impractical to search every possible combination. For this reason, only a subset of combinations is used to construct the grid. By default, caret searches at most three values for each of the *p* parameters. This means that if the numbers of the parameters is p, at most 3^p candidate models will be tested.

Caret supports all of the resampling strategies and many of the performance statistics we've learned, including accuracy and kappa, R-squared, sensitivity, specificity, and the area under the ROC curve (AUC).

By default, caret will select the candidate model with the largest value of the desired performance measure. Because this sometimes results in the selection of models that achieve marginal performance improvements via large increases in model complexity, alternative model selection functions are provided.

Creating a simple tuned model

Let's tune the credit scoring model from Lecture 7 using caret.

```
> credit <- read.csv("credit.csv")
> set.seed(1)
> m <- train(default ~ ., data = credit, method = "C5.0")
Loading required package: C50
Loading required package: plyr
```

As you can see, the C5.0 package is automatically loaded. In the train() function, we use the formula default ~. in order to predict the loan default status (yes or no) using all of the other features in the dataset. The parameter method = "C5.0" tells caret to use the C5.0 decision tree algorithm. The execution of the train() function might take some time because it needs to generate random samples of data, build decision trees, compute performance statistics, and evaluate the result.

The result is saved in an object named m, which can be examined by typing its name:

```
> m
```

The results are shown on the next slide.

Creating a simple tuned model

```
C5.0
1000 samples
  16 predictor
   2 classes: 'no', 'yes'
No pre-processing
Resampling: Bootstrapped (25 reps)
Summary of sample sizes: 1000, 1000, 1000, 1000, 1000, 1000, ...
Resampling results across tuning parameters:
  model  winnow  trials  Accuracy   Kappa
  rules  FALSE    1      0.7036502  0.2941714
  rules  FALSE   10      0.7215853  0.3267834
  rules  FALSE   20      0.7295410  0.3393856
  rules   TRUE    1      0.6960290  0.2788396
  rules   TRUE   10      0.7182906  0.3197240
  rules   TRUE   20      0.7297487  0.3388376
  tree   FALSE    1      0.7004524  0.2767404
  tree   FALSE   10      0.7286852  0.3077532
  tree   FALSE   20      0.7341156  0.3240079
  tree    TRUE    1      0.6960570  0.2651172
  tree    TRUE   10      0.7249896  0.2982891
  tree    TRUE   20      0.7263679  0.2996354

Accuracy was used to select the optimal model using  the largest value.
The final values used for the model were trials = 20, model = tree and winnow
= FALSE.
```

Creating a simple tuned model

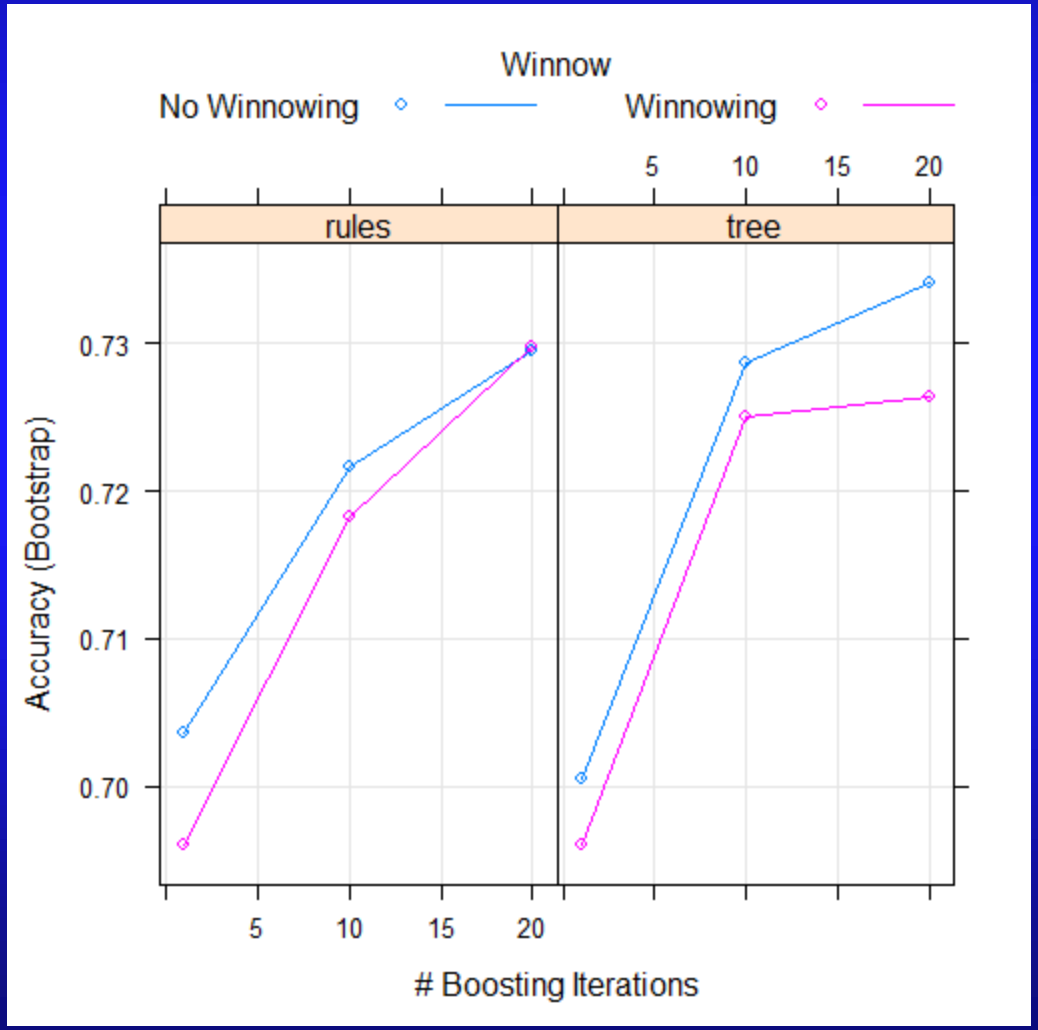
The output shows that that 25 bootstrap samples were used to train the model, each including 1,000 examples. The list of model shows that 12 different models were tested, based on different combinations of three C5.0 tuning parameters—model, trials, and winnow. The average and standard deviation of the accuracy and kappa statistics for each candidate model are also shown. The sentence at the bottom explains that the model with the largest accuracy was selected. This was the model that used a decision tree with 20 trials and the setting winnow = FALSE. After identifying the best model, the train() function uses its tuning parameters to build a model on the full input dataset, which is stored in the m list object as m$finalModel:

```
> m$finalModel
Call:
C5.0.default(x = structure(c(0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 "winnow", "noGlobalPruning", "CF", "minCases", "fuzzyThreshold",
 "sample", "earlyStopping", "label", "seed")))
Classification Tree
Number of samples: 1000
Number of predictors: 35
Number of boosting iterations: 20
Average tree size: 76.5
Non-standard options: attempt to group attributes
```

Creating a simple tuned model

The m object can be depicted using the plot() function:

```
> plot(m)
```



Creating a simple tuned model

The predict() function can be applied directly on the m object to build a vector of predictions:

```
> p <- predict(m, credit)
```

As usual, we can plot a confusion matrix that compares the predicted and actual values:

```
> table(p, credit$default)
p       no yes
  no   700   2
  yes    0 298
```

The accuracy of 99.8% is misleading because the model was built on the whole dataset, i.e., on both the training and test data. The bootstrap estimate of 73 percent (shown in the summary output) is a more realistic estimate of future performance:

```
 tree   FALSE   20       0.7341156  0.3240079
```

Creating a simple tuned model

One advantage of the caret package is that the data preparation steps applied by the train() function will also be applied by the predict() function. That is, we can skip some standard data preparation operations, such as scaling, imputation of missing values, etc.

In addition, the predict() function provides both the predicted class values and class probabilities:

```
> head(predict(m, credit, type = "prob"))
         no        yes
1 0.9606970 0.03930299
2 0.1388444 0.86115561
3 1.0000000 0.00000000
4 0.7720279 0.22797208
5 0.2948062 0.70519385
6 0.8583715 0.14162851
```

Customizing the tuning process

Each step in the parameter tuning and the model selection process can be customized. The trainControl() function is used to create a set of configuration options known as a **control object**, which guides the train() function. The trainControl() has several configuration parameters, the most important of which are *method* and *selectionFunction*. The *method* parameter is used to set the resampling method, such as holdout sampling or k-fold cross-validation. The table below lists the possible method types as well as any additional parameters for adjusting the sample size and number of iterations.

Resampling method	Method name	Additional options and default values
Holdout sampling	`LGOCV`	`p = 0.75` (training data proportion)
k-fold cross-validation	`cv`	`number = 10` (number of folds)
Repeated k-fold cross-validation	`repeatedcv`	`number = 10` (number of folds) `repeats = 10` (number of iterations)
Bootstrap sampling	`boot`	`number = 25` (resampling iterations)
0.632 bootstrap	`boot632`	`number = 25` (resampling iterations)
Leave-one-out cross-validation	`LOOCV`	None

Customizing the tuning process

The selectionFunction parameter is used to specify the function that will choose the optimal model among the various candidates. Three such functions are included: best, oneSE, and tolerance. For example:

```
best(x, metric, maximize)
oneSE(x, metric, num, maximize)
tolerance(x, metric, tol = 1.5, maximize)
```

x is a data frame of tuning parameters and model results. *metric* is a string that specifies what summary metric will be used to select the optimal model. By default, possible values are "RMSE" and "Rsquared" for regression and "Accuracy" and "Kappa" for classification. *maximize* is logical and it specifies whether the metric must be maximized or minimized. num is the number of resamples for oneSE. tol is the acceptable percent tolerance for tolerance only.

The best function simply chooses the candidate with the best value on the specified performance measure. This is used by default. The oneSE function chooses the simplest candidate within one standard error of the best performance, and tolerance uses the simplest candidate within a user-specified percentage.

Customizing the tuning process

We will create a control object named ctrl that uses 10-fold cross-validation and the oneSE selection function:

```
> ctrl <- trainControl(method = "cv", number = 10, selectionFunction
+ = "oneSE")
```

The method "cv“ is a k-fold cross-validation.

The next step is to create the grid of parameters to optimize. The grid must include a column named for each parameter in the desired model, prefixed by a period. It must also include a row for each desired combination of parameter values. Since we are using a C5.0 decision tree, this means we'll need columns named .model, .trials, and .winnow. We would like to hold constant model = "tree" and winnow= "FALSE" while searching eight different values of trials. This can be created using the expand.grid() function:

```
> grid <- expand.grid(.model = "tree", .trials = c(1, 5, 10, 15, 20,
+  25, 30, 35), .winnow = "FALSE")
```

The resulting grid data frame contains *8* rows:

Customizing the tuning process

```
> grid
  .model .trials .winnow
1   tree       1   FALSE
2   tree       5   FALSE
3   tree      10   FALSE
4   tree      15   FALSE
5   tree      20   FALSE
6   tree      25   FALSE
7   tree      30   FALSE
8   tree      35   FALSE
```

The train() function will build a candidate model for evaluation using each row's combination of model parameters.

This completes the customization process and we are ready to call the train() function.

```
> set.seed(1)
```

Customizing the tuning process

```
> m <- train(default ~ ., data = credit, method = "C5.0", metric =
"Kappa", trControl = ctrl, tuneGrid = grid)
```

Here, we are using all features to predict the default status (default ~ .), the method used is C5.0, the performance metric is kappa, the control object, ctrl, specifies that we are using 10-fold cross-validation with oneSE selection function, and the grid specifies the selected combinations of the three parameters, .model, .trials, and .winnow

The resulting object m is shown on the next slide.

Customizing the tuning process

```
> m
C5.0
1000 samples
  16 predictor
   2 classes: 'no', 'yes'
No pre-processing
Resampling: Cross-Validated (10 fold)
Summary of sample sizes: 900, 900, 900, 900, 900, 900, ...
Resampling results across tuning parameters:
  trials  Accuracy  Kappa
   1      0.718     0.2922926
   5      0.710     0.2825184
  10      0.722     0.2925821
  15      0.734     0.3325025
  20      0.741     0.3383736
  25      0.748     0.3675356
  30      0.741     0.3465481
  35      0.742     0.3469012
Tuning parameter 'model' was held constant at a value of tree
Tuning  parameter 'winnow' was held constant at a value of FALSE
Kappa was used to select the optimal model using  the one SE rule.
The final values used for the model were trials = 20, model = tree and winnow
 = FALSE.
```

Customizing the tuning process

As 10-fold cross-validation was used, the sample size to build each candidate model was reduced to 900 rather than the 1,000 used in the bootstrap.

The best model here uses trials = 20. Even though the 25-trial model offers the best raw performance according to kappa, the 20-trial model offers nearly the same performance with a simpler form. Not only are simple models more computationally efficient, but they also reduce the chance of overfitting the training data.

Improving model performance with meta-learning

Meta-learning is a form of machine learning that exploits meta data about learning algorithms. A meta-learning algorithm learns from experience how to change certain aspects of itself, in order to improve its experience. There are different types of meta-learning:

Inductive transfer: it partitions the parameters into a task-specific and general. The general parameters can be transferred from one task to another, whereas the task-specific parameters are not transferred. For example, when training artificial neural networks, one can either use a shared architecture, or transfer the learned network weights directly from another network.

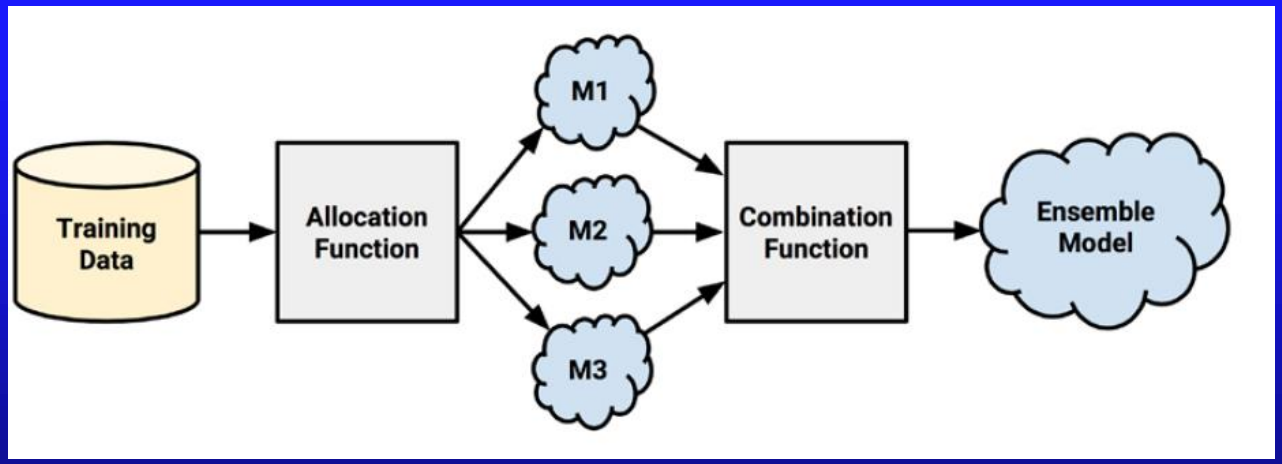
Ensemble methods: combine the outputs of multiple classifiers and are typically used for classification. Stacked generalization, for example, either uses various algorithms to produce its component classifiers, or the same algorithm trained on different subsets of the data. Bagging builds data subsets by sampling. Boosting iteratively builds new classifiers by reweighing samples to put higher emphasis on misclassified samples.

Self-modification: the learning algorithm learns how to modify some of its parameters in order to improve its performance. For example, recurrent neural networks can learn to run their own weight change algorithm.

Understanding ensembles

All the ensemble methods are based on the idea that by combining multiple weaker learners, a stronger learner is created. Ensemble methods differ based on how weak learners are chosen, based on how data is allocated between the weak learners and based on how the weak learners' predictions are combined to make a single final prediction.

Ensemble learning consists of two main steps: allocating data among learners and combining learners’ outputs:



Understanding ensembles

The **allocation function** dictates how much of the training data each model receives. For example, bootstrap sampling can be used to construct a unique training dataset for each learner. Another alternative is to give each learner a different subset of features. If the ensemble already includes a diverse set of learners, the allocation function might pass the same data on to each learner.

The **combination function** combines the learners' individual outputs into one final output. For example, the ensemble might use a majority vote to determine the final prediction. **Stacking** is the process of training a separate learning algorithm to combine the predictions of several other learning algorithms. First, all learners are trained using the available data, then a combiner algorithm is trained to make a final prediction using all the predictions of the other algorithms.

Advantages of ensemble learning

- **Better generalizability to future problems**: The risk of overfitting can be reduced by using a diversified set of learners.
- **Improved performance on massive or miniscule datasets**: Massive datasets can be split into several pieces that are allocated to different learners. Resampling methods such as bootstrapping can be used for smaller datasets. Performance gains can be obtained by training an ensemble in parallel using distributed computing methods.
- **The ability to synthesize data from distinct domains**: ensemble learning can incorporate knowledge from multiple types of learners.
- **A more nuanced understanding of difficult learning tasks**: by dividing the learning task and data into smaller portions, ensemble learning can discover subtle data patterns.

Bagging

Bagging (Bootstrap aggregating) was proposed by Leo Breiman in 1994 to improve the classification by combining classifications of randomly generated training sets using bootstrap sampling the original training data.

Bagging tends to improve the performance of unstable learners, such as artificial neural networks, classification and regression trees, and subset selection in linear regression. Unstable learners are those that generate models that tend to change substantially when the input data changes only slightly. Using unstable models in bagging increases the ensemble's diversity despite minor variations between the bootstrap training datasets.

As an example, we will the ipred package to implement bagged decision trees:

```
> install.packages("ipred")
> library(ipred)
```

The function that impelements the bagging functionality is called bagging().

Bagging

```
> set.seed(1)
> mybag <- bagging(default ~ ., data = credit, nbagg = 25)
```

The nbagg parameter controls the number of bootstrap replications, i.e., the number of decision trees voting in the ensemble. By default is 25.

The resulting mybag model can be used to make predictions:

```
> credit_pred <- predict(mybag, credit)
> table(credit_pred, credit$default)
credit_pred  no yes
        no  700   2
        yes   0 298
```

The table shows an accuracy rate of 99.7%. We will evaluate the model performance using the train() function from the caret package with 10-fold CV.

```
> library(caret)
> set.seed(1)
```

We will first create a control object using the trainControl() function:

```
> ctrl <- trainControl(method = "cv", number = 10)
```

Bagging

Then, we will call the train() function with “treebag” as the method name for the ipred bagged tree:

```
> train(default ~ ., data = credit, method = "treebag", trControl = ctrl)
Loading required package: plyr
Loading required package: e1071
Bagged CART
1000 samples
  16 predictor
   2 classes: 'no', 'yes'
No pre-processing
Resampling: Cross-Validated (10 fold)
Summary of sample sizes: 900, 900, 900, 900, 900, 900, ...
Resampling results:
  Accuracy  Kappa
  0.753     0.3757035
```

The kappa statistic is on a par with the best-tuned C5.0 decision tree

Bagging

The caret package also provides a more general bag() function that supports many more models. The function uses a control object to configure the bagging process. The control object is used to specify three bagging functions: one function for fitting the model, another for making predictions, and a third function for aggregating the votes. The caret package comes with several built-in control objects for different models: ldaBag plsBag nbBag ctreeBag svmBag nnetBag.

As an example, we will predict the credit default risk using a SVM model..

```
> library(kernlab)
```

The svmBag list object provides the three functions for the control object:

```
>  str(svmBag)
List of 3
 $ fit      :function (x, y, ...)
 $ pred     :function (object, x)
 $ aggregate:function (x, type = "class")
```

We can look at each function separately. The fitting function calls the ksvm() function from the kernlab package and returns the result:

Bagging

```
> svmBag$fit
function (x, y, ...)
{
    loadNamespace("kernlab")
    out <- kernlab::ksvm(as.matrix(x), y, prob.model = is.factor(y),
        ...)
    out
}
<environment: namespace:caret>
```

The prediction function calls the predict() function to create a data frame of prediction and sets the row and column names:

```
> svmBag$pred
function (object, x)
{
    if (is.character(lev(object))) {
        out <- predict(object, as.matrix(x), type = "probabilities")
        colnames(out) <- lev(object)
        rownames(out) <- NULL
    }
    else out <- predict(object, as.matrix(x))[, 1]
    out
}
<environment: namespace:caret>
```

Bagging

The svmBag$aggregate function aggregates the votes by taking the median value.

Instead of using svmBag$pred, we will define our own predict function:

```
> predfunct<-function (object, x)
+ {
+  if (is.character(lev(object))) {
+     out <- predict(object, as.matrix(x), type = "probabilities")
+     colnames(out) <- lev(object)
+     rownames(out) <- NULL
+   }
+   else out <- predict(object, as.matrix(x))[, 1]
+   out
+ }
```

We can then combine predfunct, svmBag$fit, and svmBag$aggregate to create a bagging control object:

```
bagctrl <- bagControl(fit = svmBag$fit,predict =
predictfunct,aggregate = svmBag$aggregate)
```

Bagging

Now, we are ready to build a bagged object using the train() function：

```
> set.seed(1)
> svmbag <- train(default ~ ., data = credit, "bag",trControl =
ctrl, bagControl = bagctrl)
```

As usual, it can take awhile to compute the result. The bag object is:

```
> svmbag
Bagged Model
1000 samples
  16 predictor
   2 classes: 'no', 'yes'
No pre-processing
Resampling: Cross-Validated (10 fold)
Summary of sample sizes: 900, 900, 900, 900, 900, 900, ...
Resampling results:
  Accuracy  Kappa
  0.734     0.3147658
Tuning parameter 'vars' was held constant at a value of 35
```

Given that the kappa statistic is 31%, the svm bagged model is not as good as bagged decision tree model.

Boosting

The *bagging* approach suffers from a serious shortcoming: the voting classifiers have all been induced independently of each other from randomly selected data. One would surmise that a smarter approach should rely on a mechanism that makes the classifiers complement each other. For instance, this can be done by inducing each of them from training examples that are perceived as difficult by the other classifiers. *Schapire's boosting* was invented with this idea in mind. Boosting combines weak classifiers to obtain very strong classifier.

The basic idea behind boosting is the following: given a weak learner, run it multiple times on (reweighted) training data, then let the learned classifiers vote. On each iteration, weight each training example by how incorrectly it was classified. The final classifier is a linear combination of the votes of the different classifiers weighted by their strength.

The principal difference between boosting and bagging is that the base classifiers are trained in sequence, and each base classifier is trained using a weighted form of the data set in which the weighting coefficient associated with each data point depends on the performance of the previous classifiers. In particular, points that are misclassified by one of the base classifiers are given greater weight when used to train the next classifier in the sequence.

AdaBoost

One of the first boosting algorithm was AdaBoost or adaptive boosting proposed by Freund and Schapire in 1997. The algorithm won the prestigious Godel prize. In general the idea of boosting is considered one of the most powerful inventions in the area of ML.

The Adaboost algorithm maintains a set of weights over the original training set S and adjusts these weights after each classier is learned by the base learning algorithm. The adjustments increase the weight of examples that are misclassified by the base learning algorithm and decrease the weight of examples that are correctly classified. There are two ways that Adaboost can use these weights to construct a new training set S_1 to give to the base learning algorithm. In boosting by sampling, examples are drawn with replacement from S with probability proportional to their weights. The second method, boosting by weighting, can be used with base learning algorithms that can accept a weighted training set directly. With such algorithms, the entire training set S (with associated weights) is given to the base learning algorithm.

AdaBoost

1. Consider a two-class problem, output variable coded as Y $\in$ {-1,+1}. For a predictor variable X, a classifier G(X) produces predictions that are in {-1,+1}. The error rate on the training sample is:

$$\overline{err} = \frac{1}{N}\sum_{i=1}^{N} I(y_i \neq G(x_i))$$

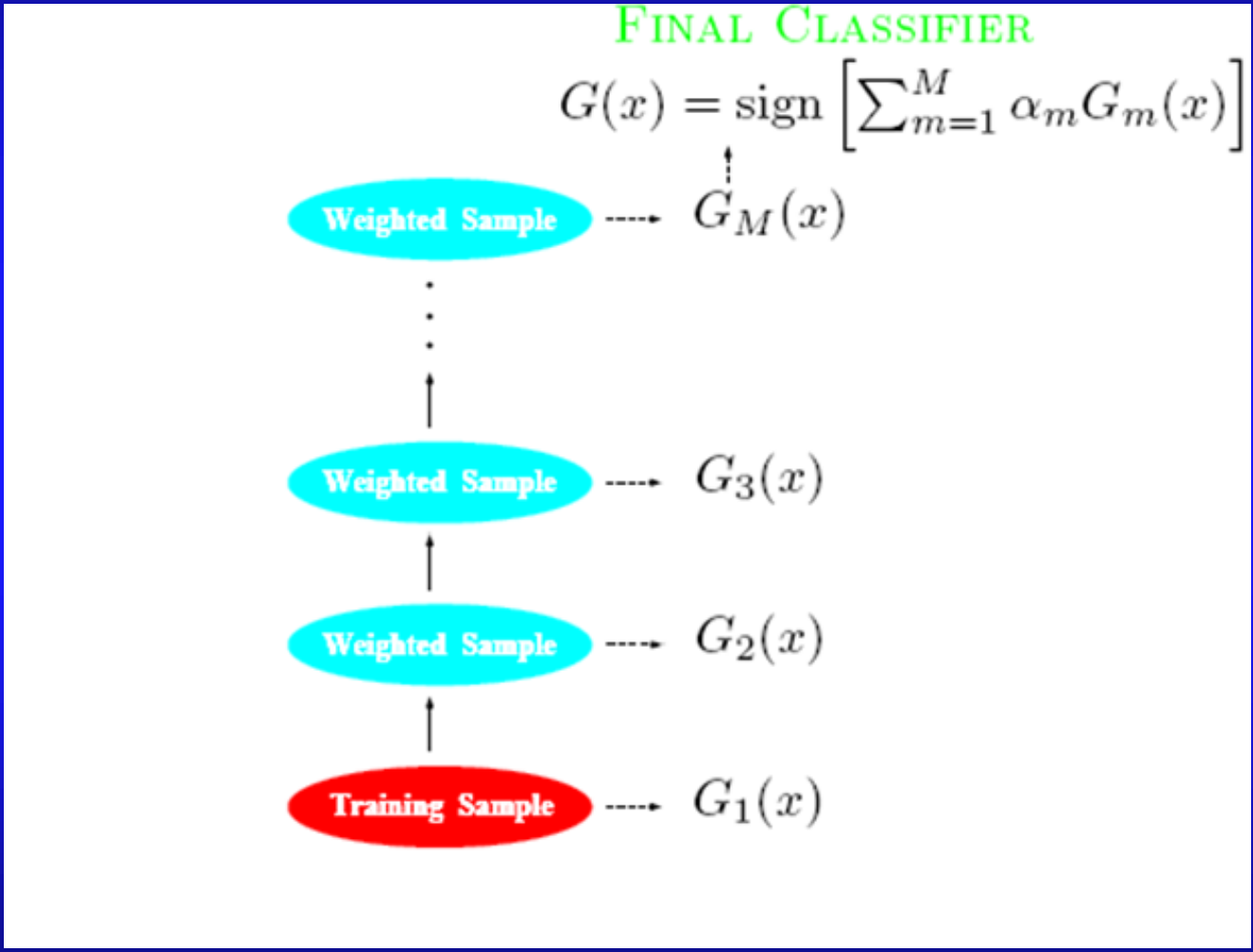
2. Sequentially apply the weak classification to repeatedly modified versions of data to produce a sequence of weak classifiers $G_m(x)$, m=1,2,..,M.

3. The predictions from all classifiers are combined via majority vote to produce the final prediction:

FINAL CLASSIFIER

$$G(x) = \text{sign}\left[\sum_{m=1}^{M} \alpha_m G_m(x)\right]$$

AdaBoost


FINAL CLASSIFIER
$G(x) = \text{sign}\left[\sum_{m=1}^{M} \alpha_m G_m(x)\right]$
Weighted Sample
$G_M(x)$
Weighted Sample
$G_3(x)$
Weighted Sample
$G_2(x)$
Training Sample
$G_1(x)$

AdaBoost

As an example, we will apply the AdaBoost.M1 algorithm on the credit dataset. The implementation of the algorithm is provided by the adabag package.

```
> install.packages("adabag")
> library(adabag)
> set.seed(1)
> m_adaboost <- boosting(default ~ ., data = credit)
```

We will apply the predict() function to the m_adaboost object to make predictions:

```
> p_adaboost <- predict(m_adaboost, credit)
```

We can look at the confusion matrix, which is stored in the confusion sub-object of p_adaboost:

```
> p_adaboost$confusion
               Observed Class
Predicted Class  no yes
            no  700   0
            yes   0 300
```

AdaBoost

It is impressive that the accuracy is 100%. Since boosting allows the error rate to be reduced to an arbitrarily low level, the learner simply continued until it made no more errors. This likely resulted in overfitting on the training dataset.

Since the error rate is measured on the training data, we need a more accurate assessment of performance on unseen data. As usual, we will perform a10-fold CV. The boosting.cv function from the adabag package provides the 10-fold CV:

```
> set.seed(1)
> adaboost_cv <- boosting.cv(default ~ ., data = credit)
```

It took about 15 minutes on my computer to produce the result. We can look at the confusion matrix once again:

```
> adaboost_cv$confusion
                Observed Class
Predicted Class  no yes
            no  600 159
            yes 100 141
```

AdaBoost

To view the kappa statistic, we will use the vcd package:

```
> library(vcd)
> Kappa(adaboost_cv$confusion)
             value    ASE     z  Pr(>|z|)
Unweighted 0.3466 0.0325 10.67 1.468e-26
Weighted   0.3466 0.0325 10.67 1.468e-26
```

The kappa statistic is little bit below the one of the best-tuned C5.0 decision tree and bagged decision trees.

The AdaBoost.M1 algorithm can be tuned in caret by specifying method = "AdaBoost.M1".

Random forests

Random forests were developed by Breiman in 2001 as an ensemble learning method that operates by constructing many decision trees and outputting the class that is the mode of the classes (classification) or mean prediction (regression) of the individual trees.

Random forests build a large collection of *de-correlated* trees. On many problems, the performance of random forests is very similar to boosting, and they are simpler to train and tune. As a consequence, random forests are popular, and are implemented in a variety of packages. The essential idea in bagging is to average many noisy but approximately unbiased models, and hence reduce the variance. Trees are ideal candidates for bagging, since they can capture complex interaction structures in the data, and if grown sufficiently deep, have relatively low bias. Since trees are notoriously noisy, they benefit greatly from the averaging.

Random forests vs. tree bagging

In standard tree bagging, the training algorithm repeatedly selects a random sample with replacement of the training set and fits trees to these samples. Predictions are made by averaging the predictions from all the individual trees on unseen example.

The main difference between tree bagging and random forests is that random forests use a modified algorithm that selects a random subset of features to fit a new tree. The reason for doing this is the correlation of the trees in an ordinary bagging: if some features are very strong predictors for the response variable, then they will be selected in many of the trees, causing them to become correlated.

Since random forests use a random portion of the full feature set, they can handle extremely large datasets, thereby avoiding the "curse of dimensionality“.

Random forests can handle noisy or missing data as well as categorical or continuous features. One drawback of random forests is that the results might not be easily interpretable.

Training random forests

We will use the randomForest package, which is also supported by caret for automated tuning.

```
> install.packages("randomForest")
> library(randomForest)
```

Here is the syntax of the randomForest() function:

Random forest syntax

using the `randomForest()` function in the `randomForest` package

Building the classifier:

```
m <- randomForest(train, class, ntree = 500, mtry = sqrt(p))
```

- `train` is a data frame containing training data
- `class` is a factor vector with the class for each row in the training data
- `ntree` is an integer specifying the number of trees to grow
- `mtry` is an optional integer specifying the number of features to randomly select at each split (uses `sqrt(p)` by default, where `p` is the number of features in the data)

The function will return a random forest object that can be used to make predictions.

Making predictions:

```
p <- predict(m, test, type = "response")
```

- `m` is a model trained by the `randomForest()` function
- `test` is a data frame containing test data with the same features as the training data used to build the classifier
- `type` is either `"response"`, `"prob"`, or `"votes"` and is used to indicate whether the predictions vector should contain the predicted class, the predicted probabilities, or a matrix of vote counts, respectively.

The function will return predictions according to the value of the `type` parameter.

Example:

```
credit_model <- randomForest(credit_train, loan_default)
credit_prediction <- predict(credit_model, credit_test)
```

Training random forests

By default, the randomForest() function creates an ensemble of 500 trees that consider $\sqrt{p}$ random features at each split, where p is the number of features in the training data set. For example, since the credit data has 16 features, each tree would be limited to splitting on four features at any time. The goal of using a large number of trees is to train enough so that each feature has a chance to appear in several models. This is the rationale behind the default value of $\sqrt{p}$. Using $\sqrt{p}$ features for a tree limits the features sufficiently so that substantial random variation occurs from tree-to-tree.

```
> set.seed(1)
> rf <- randomForest(default ~ ., data = credit)
> rf
Call:
 randomForest(formula = default ~ ., data = credit)
               Type of random forest: classification
                     Number of trees: 500
No. of variables tried at each split: 4
        OOB estimate of  error rate: 23.8%
Confusion matrix:
     no yes class.error
no  635  65  0.09285714
yes 173 127  0.57666667
```

Training random forests

The confusion matrix does not show the resubstitution error, i.e., the error on the training set. Instead, it shows the so called **out-of-bag error rate,** an unbiased estimate of future performance.

The out-of-bag estimate is computed during the construction of the random forest. When an example is not selected for a single tree's bootstrap sample, it can be used to test the model's performance on unseen data. At the end of the forest construction, the predictions for each example each time it was held out are tallied, and a vote is taken to determine the final prediction for the example. The total error rate of such predictions becomes the out-of-bag error rate.

Evaluating random forest performance

Since the randomForest() function is supported by the caret package, the model can be auto-tuned. Let's compare an auto-tuned random forest to the best auto-tuned boosted C5.0 model. We will use repeated 10-fold CV repeated 10 times in order to get better results.

```
> library(caret)
> ctrl <- trainControl(method = "repeatedcv", number = 10, repeats = 10)
```

The next step is to create the grid for the random forest. The only tuning parameter for this model is mtry, which defines how many features are randomly selected at each split. By default, it is 4, i.e., 4 features per tree will be used. We will tune the model based on four different values of mtry: 2, 4, 8, and 16.

```
> grid_rf <- expand.grid(.mtry = c(2, 4, 8, 16))
```

When the full set of features is used (mtry=16), the random forest will be equivalent to a bagged decision tree model.

Evaluating random forest performance

```
> set.seed(1)
> m_rf <- train(default ~ ., data = credit, method = "rf", metric =
+ "Kappa", trControl = ctrl, tuneGrid = grid_rf)
```

Again, it takes some time to compute the result.

```
> m_rf
Random Forest
1000 samples
  16 predictor
   2 classes: 'no', 'yes'
No pre-processing
Resampling: Cross-Validated (10 fold, repeated 10 times)
Summary of sample sizes: 900, 900, 900, 900, 900, 900, ...
Resampling results across tuning parameters:
  mtry  Accuracy  Kappa
   2    0.7307    0.1619802
   4    0.7500    0.3098880
   8    0.7535    0.3427076
  16    0.7569    0.3650084
Kappa was used to select the optimal model using  the largest value.
The final value used for the model was mtry = 16.
```

Evaluating random forest performance

Now, we will create an auto-tuned boosted C5.0 model:

```
> grid_c50 <- expand.grid(.model = "tree", .trials = c(10, 20, 30, 40),
.winnow = "FALSE")
> m_c50 <- train(default ~ ., data = credit, method = "C5.0", metric =
"Kappa", trControl = ctrl, tuneGrid = grid_c50)
> m_c50
```

The result is shown on the next slide. The best C.5.0 model has 40 trials and kappa of 0.343, whereas the best random forest uses 16 features and has kappa of 0.365

Evaluating random forest performance

```
C5.0
1000 samples
  16 predictor
   2 classes: 'no', 'yes'
No pre-processing
Resampling: Cross-Validated (10 fold, repeated 10 times)
Summary of sample sizes: 900, 900, 900, 900, 900, 900, ...
Resampling results across tuning parameters:
  trials  Accuracy  Kappa
  10      0.7354    0.3245479
  20      0.7396    0.3352073
  30      0.7415    0.3399885
  40      0.7430    0.3437301
Tuning parameter 'model' was held constant at a value of tree
Tuning
 parameter 'winnow' was held constant at a value of FALSE
Kappa was used to select the optimal model using  the largest value.
The final values used for the model were trials = 40, model = tree and winnow
 = FALSE.
```