Ensemble methods

Introduction to Computational BioStatistics with R

Alexey Fedoseev

November 25, 2025

Ensemble methods

It is possible to bundle multiple machine-learning, or regression, models together into a single model. Such techniques are known as “ensemble methods”, and the results as “ensemble models”. Why would we want to do that?

- Some methods are stronger on certain data sets than others
- Different algorithms have different weaknesses or strengths than others
- By combining the models together we hope to have the different models complement each other, resulting in an aggregate model which is more accurate than the individual models themselves

We will focus on classification examples today, but many of these models also have regression versions, which could be used instead.

Voting

The simplest of the ensemble methods are

- voting, for classification problems
- or averaging, for regression problems

In these methods, multiple classification or regression models are trained. These models can be created by

- using different algorithms (kNN, decision tree, etc),
- using different splits of the training data set,
- both

We then generate the predictions for the test data for all the models.

Majority voting

The simplest voting method is majority voting.

- For each test data point the final result is the result which gets more than half of the votes from the various models
- If no prediction gets more than half of the votes, we say that the ensemble method could not make a stable prediction
- In this case we would usually pick the prediction that gets the most votes. This is sometimes called “plurality voting”
- Another example is weighted voting: the votes from the different models are weighed unequally. The prediction with the highest weighted value of votes is the winner

For continuous models the predicted values are either simply averaged, or a weighted average is used.

Ionosphere data set

Let's use something new, the Ionosphere data set which consists of radar data. The targets were free electrons in the ionosphere. "Good" radar returns are those showing evidence of some type of structure in the ionosphere. "Bad" returns are those that do not; their signals pass through the ionosphere.

To load this data set we would require the `mlbench` library:

```
library(caret)
library(mlbench)
data(Ionosphere)
```

The second column `V2` contains only zeros and we will exclude it from our analysis. We also rename the first column to avoid error messages:

```
mydata <- Ionosphere[,-2]
names(mydata)[1] <- "x1"
```

Stratified random split

Our data set has around 64% of the "good" values and approximately 36% of the "bad" values. We would like to preserve this balance in our testing and training data sets. To achieve this, we will use the function `createDataPartition` from the `caret` library.

```
index <- createDataPartition(mydata$Class, p=0.75, list=FALSE)
trainSet <- mydata[index,]
testSet <- mydata[-index,]
```

Here we are asking for a 75% split between the training and testing data and that the resulting `index` should be a matrix instead of a list.

Let's create a function to confirm the ratios of "good" and "bad" values in our vector:

```
calc.perc <- function(vec, val) {
  return(sum(vec == val) / length(vec))
}
```

Stratified random split

To make things simple we will just display on the values in a row:

```
cat(calc.perc(mydata$Class,   "good"),
    calc.perc(trainSet$Class, "good"),
    calc.perc(testSet$Class,  "good"), "\n")

cat(calc.perc(mydata$Class,   "bad"),
    calc.perc(trainSet$Class, "bad"),
    calc.perc(testSet$Class,  "bad"), "\n")
```

The resulting ratios are very close to each other:

```
0.6410256 0.6401515 0.6436782
0.3589744 0.3598485 0.3563218
```

Now we are sure that we do not have underrepresented categories in both of our data sets.

Building models

We will proceed by building several models.

To help build our models we will use 10-fold cross-validation:

```
fitControl <- trainControl(method = "cv", number = 10, classProbs = TRUE)

model_dt <- train(Class ~ ., data = trainSet,
  method = "rpart", trControl = fitControl)

model_lr <- train(Class ~ ., data = trainSet,
  method = "glm", trControl = fitControl, preProc = c("center", "scale"))

model_svm <- train(Class ~ ., data = trainSet,
  method="svmLinear", trControl=fitControl, preProc=c("center", "scale"))
```

Notice that in our logistic regression and SVM models we also center and scale our data.

Building models

We can use our `calc.perc` function to calculate the accuracy of our predictions on the test set:

```
pred_dt <- predict(model_dt, testSet)
pred_lr <- predict(model_lr, testSet)
pred_svm <- predict(model_svm, testSet)

cat("DT ", calc.perc(testSet$Class, pred_dt), "\n")
cat("LR ", calc.perc(testSet$Class, pred_lr), "\n")
cat("SVM", calc.perc(testSet$Class, pred_svm),"\n")
```

The resulting accuracy values are as follows:

```
DT  0.8850575
LR  0.8850575
SVM 0.8965517
```

Forming an ensemble: Averaging

Averaging means taking the average of predictions from models in case of regression problem or while predicting probabilities for the classification problem.

Since our predictions are either "good" or "bad", averaging doesn't make much sense for this binary classification. Instead, we can average the probabilities of observations:

```
pred_dt_prob  <- predict(model_dt,  testSet, type = 'prob')
pred_lr_prob  <- predict(model_lr,  testSet, type = 'prob')
pred_svm_prob <- predict(model_svm, testSet, type = 'prob')
```

```
> head(pred_dt_prob, 5)
          bad      good
2  0.05952381 0.9404762
7  0.05952381 0.9404762
13 0.05952381 0.9404762
14 0.78571429 0.2142857
15 0.78571429 0.2142857
```

Forming an ensemble: Averaging

Now we can calculate the average probability:

```
pred_avg <- (pred_dt_prob$good + pred_lr_prob$good + pred_svm_prob$good) / 3
```

Since we are not really interested in probabilities, we will convert them to values:

```
pred_avg <- as.factor(ifelse(pred_avg > 0.5, "good", "bad"))
```

We can use our `calc.perc` function to check the accuracy of our model on the testing data:

```
> calc.perc(pred_avg, testSet$Class)
[1] 0.908046
```

Comparing results

Method	Accuracy
DT	0.8850575
LR	0.8850575
SVM	0.8965517
Averaging	0.9080460

The ensemble method performed slightly better than our individual models.

Stacking

Stacking involves building of a meta-model from some base models

- We first train a bunch of models on the training data set
- These models are then used to predict the targets of the training data
- We then create a new data set, consisting of these target *predictions* from the trained models, and the correct target from the original training data set
- We then train a meta-model on this new data set
- By using the predictions of the different models as features, we can let the meta-model determine when certain models perform well, and when certain models perform poorly

This technique is particularly useful for combining models of different types.

A note on performance

Not all ensemble models will outperform the models from which they are built.

- Sometimes the underperforming base models drag the ensemble model down with them
- Sometimes the choice of base models or meta-models is not appropriate for the data set in question
- Certain data sets lend themselves better to different types of base models; the same applies to ensemble methods
- That being said, many of the latest machine learning competition winning models have been ensemble methods
- If you go down this road you will need to experiment with many combinations of base models and parameters to find the best-performing combination

Bootstrap aggregating (bagging)

Another ensemble technique is Bootstrap Aggregating (commonly known as "Bagging").

- This is useful for algorithms that have a high variance (decision trees)
- We first train a bunch of models of the same type on different versions of the training data set
- These data set versions are generated using bootstrapping (sampling from the training data with replacement)
- These models are then aggregated using voting (classification) or averaging (regression)
- The models can be generated in parallel

This is a very commonly-used technique if you have a base model that you're confident in.

Bagging, variations

You may run into a variety of different variations on Bagging:

- "Pasting": samples are drawn from the data set without replacement. This was originally designed for large data sets
- "Bagging": samples are drawn from the data set with replacement (the data is bootstrapped)
- "Random Subspaces": Each model is trained on a subset of the features. Also known as "feature bagging"
- "Random Patches": the combination of Bagging and Random Subspaces, the models are trained on subsets of both the samples and the features

Both Bagging and Random Patches are worth exploring if you end up going down this road.

Random Forests

Random Forests are a special variation of bagging, based entirely on decision trees.

- As with regular bagging, the data used in training are sampled from the training data, with replacement
- But rather than allow the tree to split on all available features, only a randomly-chosen subset of the full set of features is available at each split
- Random Forests use a subset of features for each split, but the model itself has access to all features
- Each tree is grown as far as possible, or close to it, without pruning
- The results of all the trees are then aggregated

This reduces the correlation between individual models and the high variance which is inherent to decision trees.

Random Forests, example

We can use the caret package to train a model using random forests:

```
model_rf <- train(Class ~ ., data = trainSet, method = "rf")

pred_rf <- predict(model_rf, testSet)
```

The accuracy of the model is slightly better than accuracy of our decision tree model:

```
> calc.perc(testSet$Class, pred_rf)
[1] 0.9425287
```

Boosting

Boosting is used to convert weak models into strong models.

- In this case, "weak" means a poor classification rate
- The skeleton of the algorithm is as follows:
  1. Start with a starting model, and the whole data set
  2. Train the existing model on the remaining data
  3. Remove the data which the model gets correct
  4. Create a new model; train it on the remaining data (the data the model gets wrong)
  5. Aggregate the new model with the existing models, using weighted majority vote (classification) or weighted sum (regression)
  6. Repeat, starting at step 3
- The algorithm actively attempts to correct for mistakes in the existing model

AdaBoost (adaptive boosting) was an early example of this type of boosting algorithm.

AdaBoost example

Library adabag provides a convenient way to build models:

```
library(adabag)
model_ada <- boosting(Class ~ ., data = trainSet)
pred_ada <- predict(model_ada, testSet)
```

```
> calc.perc(testSet$Class, pred_ada$class)
[1] 0.9310345
```

Gradient Boosting

Gradient Boosting (also called Accelerated Gradient Boosting, or Gradient Tree Boosting) is a generalization of boosting. It's based on three parts:

- A loss function L. This depends on the problem type, but must be differentiable
- A (weak) model, usually a decision tree
- A meta-model, which combines the weak models to minimize the loss function. The loss function is minimized using gradient descent

$$F_m(x) = F_{m-1}(x) + \gamma_m h_m(x)$$

where m is the iteration step. The model is built sequentially. At each step the decision tree $h_m(x)$ is chosen to minimize the loss function L.

XGBoost

If you start using Gradient Boosting, you'll quickly run into XGBoost ("eXtreme Gradient Boosting"). This is a very popular modelling algorithm.

- There are important differences, under the hood, between XGBoost and Gradient Boosting
  - XGBoost is fast; it was designed for speed
  - XGBoost can be run in parallel, either single- or multi-node
  - XGBoost uses less memory
  - You need to install the "xgboost" package

Running XGBoost with Caret

```
> # Train XGBoost model
> model_xgb <- train(Class ~ ., data = trainSet, method = "xgbTree",
+   trControl = trainControl(method = "cv", number = 5), verbosity = 0)
>
> # Predict and evaluate
> pred_xgb <- predict(model_xgb, testSet)
> calc.perc(pred_xgb, testSet$Class)
0.9425287
```

- Caret enables parallel CV. Set `allowParallel = TRUE` in `trainControl` for faster runs.

Summary

Some things to remember:

- Ensemble models combine a bunch of other base models to (hopefully) create a more-accurate model than the base models themselves
- Voting is the simplest approach, either doing majority voting or averaging of results
- Stacking involves creating a meta-model which models the results of the base models, hopefully learning where certain base models are weak
- Bagging creates multiple versions of the same base model, but each trained on different boostrapped versions of the original data set
- Random Forests is a tree-based versions of bagging, where only subsets of features are available at each split.
- Gradient Boosting uses Gradient Descent to iteratively build a better model, by focusing on what the model gets wrong. XGBoost is a super version.