A Brief Introduction to caretEnsemble

Zach Mayer

2016-01-31

`caretEnsemble` is a package for making ensembles of caret models. You should already be somewhat familiar with the caret package before trying out `caretEnsemble`.

`caretEnsemble` has 3 primary functions: **`caretList`**, **`caretEnsemble`** and **`caretStack`**. `caretList` is used to build lists of caret models on the same training data, with the same re-sampling parameters. `caretEnsemble` and `caretStack` are used to create ensemble models from such lists of caret models. `caretEnsemble` uses a glm to create a simple linear blend of models and `caretStack` uses a caret model to combine the outputs from several component caret models.

caretList

`caretList` is a flexible function for fitting many different caret models, with the same resampling parameters, to the same dataset. It returns a convenient `list` of caret objects which can later be passed to `caretEnsemble` and `caretStack`. `caretList` has almost exactly the same arguments as `train` (from the caret package), with the exception that the `trControl` argument comes last. It can handle both the formula interface and the explicit `x`, `y` interface to train. As in caret, the formula interface introduces some overhead and the `x`, `y` interface is preferred.

`caretEnsemble` has 2 arguments that can be used to specify which models to fit: `methodList` and `tuneList`. `methodList` is a simple character vector of methods that will be fit with the default `train` parameters, while `tuneList` can be used to customize the call to each component model and will be discussed in more detail later. First, lets build an example dataset (adapted from the caret vignette):

```
#Adapted from the caret vignette
library("caret")
library("mlbench")
library("pROC")
data(Sonar)
set.seed(107)
inTrain <- createDataPartition(y = Sonar$Class, p = .75, list = FALSE)
training <- Sonar[ inTrain,]
testing <- Sonar[-inTrain,]
my_control <- trainControl(
  method="boot",
  number=25,
  savePredictions="final",
  classProbs=TRUE,
  index=createResample(training$Class, 25),
  summaryFunction=twoClassSummary
  )
```

Notice that we are explicitly setting the resampling `index` to being used in `trainControl`. If you do not set this index manually, `caretList` will attempt to set it for automatically, but it"s generally a good idea to set it yourself.

Now we can use `caretList` to fit a series of models (each with the same `trControl`):

```
library("rpart")
library("caretEnsemble")
model_list <- caretList(
  Class~., data=training,
  trControl=my_control,
  methodList=c("glm", "rpart")
  )
```

(As with `train`, the formula interface is convienent but introduces move overhead. For large datasets the explicitly passing `x` and `y` is preferred). We can use the `predict` function to extract predicitons from this object for new data:

```
p <- as.data.frame(predict(model_list, newdata=head(testing)))
print(p)
```

glm	rpart
0.0000000	0.7794118
0.0000000	0.0882353
0.0000000	0.0882353
0.0000675	0.0882353
0.0000000	0.6666667
0.7654240	0.7794118

If you desire more control over the model fit, use the `caretModelSpec` to contruct a list of model specifications for the `tuneList` argument. This argumenent can be used to fit several different variants of the same model, and can also be used to pass arguments through `train` down to the component functions (e.g. `trace=FALSE` for `nnet`):

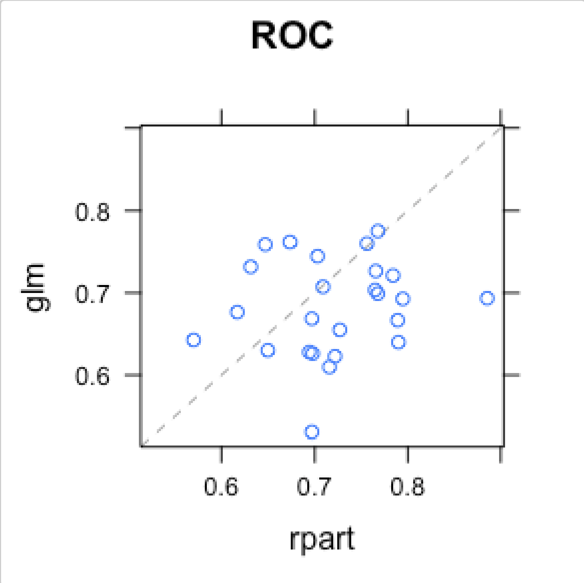
```
library("mlbench")
library("randomForest")
library("nnet")
model_list_big <- caretList(
  Class~., data=training,
  trControl=my_control,
  metric="ROC",
  methodList=c("glm", "rpart"),
  tuneList=list(
    rf1=caretModelSpec(method="rf", tuneGrid=data.frame(.mtry=2)),
    rf2=caretModelSpec(method="rf", tuneGrid=data.frame(.mtry=10), preProcess="pca"),
    nn=caretModelSpec(method="nnet", tuneLength=2, trace=FALSE)
  )
)
```

Finally, you should note that `caretList` does not support custom caret models. Fitting those models are beyond the scope of this vignette, but if you do so, you can manually add them to the model list (e.g. `model_list_big[["my_custom_model"]] <- my_custom_model`). Just be sure to use the same re-sampling indexes in `trControl` as you use in the `caretList` models!

caretEnsemble

`caretList` is the preferred way to construct list of caret models in this package, as it will ensure the resampling indexes are identical across all models. Lets take a closer look at our list of models:

```
xyplot(resamples(model_list))
```



As you can see from this plot, these 2 models are un-correlated, and the rpart model is ocassionally anti-predictive, with a few re-samples showing AUCS around 0.3 to 0.4.

We can confirm the 2 model"s correlation with the `modelCor` function from caret (caret has a lot of convienent functions for analyzing lists of models):

```
modelCor(resamples(model_list))
```

```
##             glm     rpart
## glm   1.0000000 0.1426353
## rpart 0.1426353 1.0000000
```

These 2 models make a good candidate for an ensemble: their predicitons are fairly un-correlated, but their overall accuaracy is similar. We do a simple, linear greedy optimization on AUC using caretEnsemble:

```
greedy_ensemble <- caretEnsemble(
  model_list,
  metric="ROC",
  trControl=trainControl(
    number=2,
    summaryFunction=twoClassSummary,
    classProbs=TRUE
    ))
summary(greedy_ensemble)
```

```
## The following models were ensembled: glm, rpart
## They were weighted:
## 1.4489 -0.9559 -2.0442
## The resulting ROC is: 0.7573
## The fit for each individual model on the ROC is:
##  method       ROC      ROCSD
##     glm 0.6829333 0.05890797
##   rpart 0.7206765 0.06849524
```

The ensemble"s AUC on the training set resamples is 0.76, which is about 7% better than the best individual model. We can confirm this finding on the test set:

```
library("caTools")
model_preds <- lapply(model_list, predict, newdata=testing, type="prob")
model_preds <- lapply(model_preds, function(x) x[,"M"])
model_preds <- data.frame(model_preds)
ens_preds <- predict(greedy_ensemble, newdata=testing, type="prob")
```

```
model_preds$ensemble <- ens_preds
caTools::colAUC(model_preds, testing$Class)
```

```
##                 glm     rpart  ensemble
## M vs. R 0.6496914 0.6566358 0.6967593
```

The ensemble"s AUC on the test set is about 6% higher than the best individual model.

We can also use varImp to extract the variable importances from each member of the ensemble, as well as the final ensemble model:

```
varImp(greedy_ensemble)
```

	overall	glm	rpart
V60	0.0000000	0.0000000	0.000000
V5	0.0195047	0.0612158	0.000000
V53	0.0304331	0.0955147	0.000000
V43	0.0462375	0.1451172	0.000000
V45	0.0944099	0.2963071	0.000000
V57	0.1006198	0.3157968	0.000000
V55	0.1285907	0.4035841	0.000000
V46	0.1399427	0.4392124	0.000000
V26	0.1422233	0.4463702	0.000000
V28	0.1470963	0.4616641	0.000000
V39	0.2137933	0.6709937	0.000000
V40	0.2209835	0.6935602	0.000000
V29	0.2225302	0.6984145	0.000000
V38	0.2370211	0.7438946	0.000000
V47	0.2919185	0.9161909	0.000000
V37	0.2960336	0.9291063	0.000000
V6	0.3016812	0.9468313	0.000000
V52	0.3122835	0.9801068	0.000000
V33	0.3365294	1.0562030	0.000000
V23	0.3470675	1.0892770	0.000000
V1	0.3610996	1.1333169	0.000000
V14	0.3788974	1.1891755	0.000000
V7	0.3805820	1.1944628	0.000000
V25	0.4560018	1.4311691	0.000000
V44	0.4772194	1.4977606	0.000000
V32	0.5022176	1.5762180	0.000000
V51	0.5358167	1.6816694	0.000000

	overall	glm	rpart
V54	0.5535618	1.7373624	0.000000
V58	0.5602788	1.7584441	0.000000
V56	0.5829064	1.8294611	0.000000
V59	0.5851514	1.8365071	0.000000
V49	0.6398756	2.0082597	0.000000
V24	0.7070269	2.2190151	0.000000
V4	0.8515139	2.6724900	0.000000
V22	0.8705632	2.7322766	0.000000
V3	0.8954663	2.8104353	0.000000
V36	0.9063754	2.8446737	0.000000
V41	0.9114068	2.8604649	0.000000
V8	0.9414496	2.9547546	0.000000
V19	0.9592699	3.0106838	0.000000
V2	0.9876133	3.0996403	0.000000
V48	1.0037327	3.1502311	0.000000
V30	1.0763736	3.3782158	0.000000
V21	1.1406271	3.5798764	0.000000
V50	1.1555815	3.6268111	0.000000
V34	1.2249664	3.8445765	0.000000
V20	1.3424172	4.2131978	0.000000
V42	1.4504082	4.5521294	0.000000
V35	1.5006324	4.7097589	0.000000
V31	1.5014484	4.7123197	0.000000
V27	3.0365663	0.5780564	4.186200
V15	4.0576503	0.3819398	5.776464
V18	4.2110509	0.3369883	6.022617
V17	4.4701477	0.8271323	6.173673
V16	5.3660529	0.2577508	7.754766
V13	7.3251110	0.9620234	10.300580
V10	8.2586938	0.3012508	11.979705
V12	10.8256937	1.3496604	15.256820
V9	11.5410394	3.2537912	15.416273
V11	11.8386126	0.5166883	17.132903

(The columns each sum up to 100.)

caretStack

caretStack allows us to move beyond simple blends of models to using “meta-models” to ensemble collections of predictive models. DO NOT use the `trainControl` object you used to fit the training models to fit the ensemble. The re-sampling indexes will be wrong. Fortunately, you don“t need to be fastidious with re-sampling indexes for caretStack, as it only fits one model, and the defaults `train` uses will usually work fine:

```
glm_ensemble <- caretStack(
  model_list,
  method="glm",
  metric="ROC",
  trControl=trainControl(
    method="boot",
    number=10,
    savePredictions="final",
    classProbs=TRUE,
    summaryFunction=twoClassSummary
  )
)
model_preds2 <- model_preds
model_preds2$ensemble <- predict(glm_ensemble, newdata=testing, type="prob")
CF <- coef(glm_ensemble$ens_model$finalModel)[-1]
colAUC(model_preds2, testing$Class)
```

```
##                glm     rpart  ensemble
## M vs. R 0.6496914 0.6566358 0.6967593
```

```
CF/sum(CF)
```

```
##       glm     rpart
## 0.3186219 0.6813781
```

Note that `glm_ensemble$ens_model` is a regular caret object of class `train`. The glm-weighted model weights (glm vs rpart) and test-set AUCs are extremely similar to the caretEnsemble greedy optimization.

We can also use more sophisticated ensembles than simple linear weights, but these models are much more succeptible to over-fitting, and generally require large sets of resamples to train on (n=50 or higher for bootstrap samples). Lets try one anyways:

```
library("gbm")
gbm_ensemble <- caretStack(
  model_list,
  method="gbm",
  verbose=FALSE,
  tuneLength=10,
  metric="ROC",
  trControl=trainControl(
    method="boot",
    number=10,
    savePredictions="final",
    classProbs=TRUE,
    summaryFunction=twoClassSummary
  )
)
model_preds3 <- model_preds
model_preds3$ensemble <- predict(gbm_ensemble, newdata=testing, type="prob")
colAUC(model_preds3, testing$Class)
```

```
##                glm     rpart  ensemble
## M vs. R 0.6496914 0.6566358 0.7006173
```

In this case, the sophisticated ensemble is no better than a simple weighted linear combination. Non-linear ensembles seem to work best when you have:

1. Lots of data.
2. Lots of models with similar accuracies.
3. Your models are un-correllated: each one seems to capture a different aspect of the data, and different models perform best on different subsets of the data.