Building and submitting predictive models for the breast cancer prognosis challenge

Erhan Bilal, Erich Huang, Adam Margolin

September 25, 2012

1 Load required libraries

```
> library(predictiveModeling)

randomSurvivalForest 3.6.3

Type rsf.news() to see new features, changes, and bug fixes.

> library(BCC)
> library(survival)
> library(MASS)
```

2 Load training data

```
> ## Before downloading data from the R client you must sign the terms of use through the web client.
> ## This is done by attempting to download any dataset included in the competition and agreeing to the t
> ## For example, clicking the "Download" link at https://synapse.sagebase.org/\#Synapse:syn375502 will b
> ## the terms of use dialogue if you have not already agreed to them.
>
> # synapseLogin() ### not required if configured for automatic login
> trainingData <- loadMetabricTrainingData()
```

3 Create custom predictive model

A predicitve model for the competition is implemented as an R5 class that implements the methods customTrain() and customPredict(). We recommend defining this class in a separate file with the same name as the class name. For an example, see DemoModel.R in the BCC package source code (file BCC/R/DemoModel.R).

The customTrain() method should take arguments exprData (an ExpressionSet), copyData (an ExpressionSet), clinicalFeaturesData (a data.frame), and clinicalSurvData (a Surv object). It is the user's responsibility to pre-process and combine these data as needed for the model. The predictiveModeling package provides some convenience functions to do so, such as createAggregateFeaturesDataSet().

The customPredict() method will be run on the validation dataset, in the same format as the training data, so this method should take arguments exprData (an ExpressionSet), copyData (an ExpressionSet), and clinicalFeaturesData (a data.frame). Again it is the user's responsibility to pre-process and combine these data as needed for their model.

Both the customTrain() and customPredict() method may choose to use only subsets of the input arguments, and may incorporate any additional datasets. A recommended practice is to store additional data

in Synapse in the user's custom project and load the data from Syanpse within customTrain() or customPredict() using the loadEntity() function. The user may also add additional data directly to the R5 class as additional fields.

4 Train the model

```
> modelClassFile <- "~/BCC/R/DemoModel.R"
> source(modelClassFile)

> demoPredictiveModel <- DemoModel$new()
> ### class can be instantiated directly from source file with the syntax:
> ### demoPredictiveModel <- source(modelClassFile)$value$new()
> demoPredictiveModel$customTrain(trainingData$exprData, trainingData$copyData,
+                                 trainingData$clinicalFeaturesData, trainingData$clinicalSurvData)
> trainPredictions <- demoPredictiveModel$customPredict(trainingData$exprData, trainingData$copyData,
+                                                       trainingData$clinicalFeaturesData)
```

5 Compute concordance index on training dataset

We provide a class called SurvivalModelPerformance (see the file SurvivalModelPerformance.R in the predictiveModeling package) to calculate statistics used to evaluate the accuracy of predictive model results. This class holds a vector of hazard ratio predictions (though only the sort order matters for concordance index calculations) and a Survival object, corresponding to the known (censored) survival times. The most common method of assessing the accuracy of survival model predictions is to calculate the concordance index. We also provide a function of the SurvivalModelPerformance class called getExactConcordanceIndex() that is not subject to stochastic sampling issues with concordance index calculations and returns an exact calculation of the concordance index statistic. This function is used for model comparison in the breast cancer competition.

```
> trainPerformance <- SurvivalModelPerformance$new(trainPredictions, trainingData$clinicalSurvData)
> print(trainPerformance$getExactConcordanceIndex())

[1] 0.8115018
```

6 Test that the model runs successfully on the test dataset

A valid model submission will return a vector of predicted hazard ratios in the validation dataset. We therefore provide users access to the validation data used as input to predictive models, including the expression data, copy number data, and clinical covariates. Our evaluation script will call customPredict() on the submitted model with these data as parameters, as shown below, and compare the prediction results to the known survival times from the validation set (which is hidden from users).

Run the code below on your predictive model to ensure it returns a valid vector of predictions on the test data.

```
> testData <- loadMetabricTestData(loadSurvData=FALSE)
> testPredictions <- demoPredictiveModel$customPredict(testData$exprData, testData$copyData,
+                                                      testData$clinicalFeaturesData)
```

7 Submit the model to Synapse

After training the model, the trained object should be loaded to Synapse. To compute the validation score, the customPredict() method of the trained object will be called with arguments corresponding to the expression data, copy data, and clinical features data from the test dataset.

Arguments for the model submission include: 1) The name of the model; 2) The trained model object; 3) A list or character vector containing paths to source files needed to run the model, including the model class file that defines customTrain() and customPredict(), as well as any dependent files; 4) A binary argument isPracticeModel, which is set equal to true so the model is uploaded to a project containing practice models that are not scored for the competition. When you are ready to submit a real model, set isPracticeModel equal to FALSE and a model score will be automatically computed.

See Syanpse R client documentation to learn how to save and retrieve data from Syanpse. For example, see help("addObject") or help("getEntity"). For a list of all methods supported by the Synapse client type ls("package:synapseClient").

```
> myModelName = "Erhan demo model" #change this name to something unique
> submitCompetitionModel(modelName = myModelName, trainedModel=demoPredictiveModel,
+                        rFiles=c(modelClassFile))

[1] "syn1416762"

> onWeb("syn1125643")
> ## model results will display automatically at http://validation.bcc.sagebase.org/bcc-leaderboard-publi
```

8 View model results

Your uploaded model should now be visible in Synapse in the "METABRIC competition models" project, which you can access by typing onWeb("syn375517") or navigating to synapse.sagebase.org/#Synapse:syn375517. Because the isPracticeModel flag was set to TRUE, a model with the name you specified should be available by clicking "Unevaluated practice models".

When you are ready to submit your customized model, set the isPracticeModel flag to FALSE and your model will be uploaded to "Evaluated metabric models", corresponding to Synapse Entity syn375380. Models in this project will be scored for their accuracy in predicting survival the samples from the validation set. The initial data release contains 1,000 samples released for training and around 500 held out for testing. The concordance index will be calculated on these 500 samples for all models submitted to this project and results displayed in real time at http://validation.bcc.sagebase.org/bcc-leaderboard-public.php.

As the competition progresses, additional Sage-curated breast cancer datasets will be released to assist users with innovative ideas of how additional data may be incorporated in model training. We will also add the feature of evaluating accuracy of models training on the METABRIC data in predicting survival in other breast cancer datasets.

The scores of all models submitted with isPracticeModel set to false are displayed in real time at http://validation.bcc.sagebase.org/bcc-leaderboard-public.php.

9 Evaluate model performance using cross validation (optional)

The recommended method for evaluating the performance of custom models prior to submission is to use cross validation on the training dataset. To do so, we provide a convenience function called crossValidatePredictiveSurvivalModel that runs cross validation on a custom model. This script is called with the first argument corresponding to the custom model, and the following arguments the same as used in the customTrain() function.

The function crossValidatePredictiveModel returns a list with elements trainPerformanceCV and testPerformanceCV, each of which are an object of type SurvivalModelPerformanceCV (see predictiveModeling source code). This object contains a list of elements of type SurvivalModelPerformance, corresponding to the predicted and observed values for each cross validation fold. As shown below, calling the function getFoldCIndices() on a SurvivalModelPerformanceCV object returns a vector of concordance indices for each of the folds.

It is recommended, though optional, to compute and upload the cross validation performance object with the model submission. As the competition progresses, final submissions will require cross validation results in order to compare cross validation and test data performance.

```
> cvPerformance <- crossValidatePredictiveSurvivalModel(DemoModel$new(), trainingData$exprData,
+                                                       trainingData$copyData,
+                                                       trainingData$clinicalFeaturesData,
+                                                       trainingData$clinicalSurvData, numFolds = 3)

> cvPerformance$trainPerformanceCV$getFoldCIndices()

[1] 0.8089860 0.8185439 0.8129256

> cvPerformance$testPerformanceCV$getFoldCIndices()

[1] 0.6988507 0.7044735 0.6977526
```

10 Advanced topic: uploading models with specific parameter settings

Some users may want to build models that use specific parameter settings of a more general algorithm. Because models must be able to be re-trained by calling the object constructor with no arguments, it would be cumersome and undesirable coding practice to re-implement all logic for an algorithm for each custom parameter setting. However, it is simple to upload models with specific parameter settings by defining a subclass of an algorithm that calls the superclass constructor with specific parameter settings.

As an example, see the class definintions in predictiveModeling/R/RSFmodel.R and predictiveModeling/R/RSFmodel_500feat_100trees.R. In this example, RSFmodel defines a random survival forest algorithm, with customizable parameters for the number of significant features to include and the number of trees to build in the random forest. RSFmodel_500feat_100trees defines a subclass of RSFmodel that instantiates an RSFmodel with parameter settings corresponding to 500 features and 100 trees. Simply upload both files in the rFiles argument of submitCompetitionModel.