Building predictive models of drug sensitivity using Synapse

Adam Margolin, Nicole Deflaux

December 8, 2011

1 Sage Bionetworks Synapse project

The recent exponential growth of biological "omics" data has occurred concurrently with a decline in the number of NMEs approved by the FDA. Sage Bionetworks believes that a fundamental reason biological research productivity does not scale with biological data generation is that the analysis and interpretation of genomic data remains largely an isolated, individual activity. Sage Bionetworks' mission is to catalyze a cultural transition from the traditional single lab, single-company, and single-therapy R&D paradigm to a model with broad precompetitive collaboration on the analysis of large scale data in medical sciences. For this to happen it is critical that: 1) human health data become accessible and reusable by people other than the original data generators allowing multiple parallel approaches to data interpretation; 2) analytical methodologies become fully reproducible and transparent so that results can be vetted and existing analysis techniques quickly applied to new application areas, and; 3) models of biological systems and networks be opened to a variety of users such that theoretical predictions can be rapidly validated experimentally and improve standards of care for patients. Sage Bionetworks is actively engaged with academic and pharmaceutical collaborators in developing technical, policy and strategic solutions to these issues. Part of Sage's solution is Synapse, a platform for open, reproducible data-driven science, which will provide support for Sage's research initiatives and serve as a resource for the broader scientific community.

Synapse will support the reusability of information facilitated by ontology-based services and applications directed at scientific researchers and data curators. Sage Bionetworks is actively pursuing the acquisition, curation, statistical quality control, and hosting of human and mouse global coherent datasets for use by Sage Bionetworks researchers, collaborators, and the broader research community. Global coherent datasets integrate both clinical phenotype and genomic data along with an intermediate molecular layer such as gene expression or proteomic data. Typically studies contain genome-wide genetic variation data and/or expression profiling data. We expect the release of these sorts of unique, integrative, high value datasets into the public domain will seed a variety of analytical approaches to drive new treatments based on better understanding of disease states and the biological effects of existing drugs.

Reproducible analysis and disease model reuse require a supporting informatics infrastructure. In the Sage Bionetworks system, users interact with resources via a number of mechanisms depending upon their interests and expertise. The Synapse web portal will be an environment for end user scientists to interact and share data, models, and analysis methods, both in the context of specific research projects, and broadly across otherwise disparate projects. Many other specialized scientific tools can be extended to load data and save results to the Sage Bionetworks platform, or to perform analysis by calling methods executed on a remote service. The Sage Bionetworks Platform is aggressively leveraging and optimizing its architecture to take full advantage of the rapidly maturing cloud computing technologies which will put on-demand supercomputing power in the hands of the average researcher. These more specialized analytical clients would support use cases in data curation and quality control as well as scientific analysis.

2 Set up your Synapse work environment

Go to `https://synapse-alpha.sagebase.org/` to register for a new account and then log into Synapse.

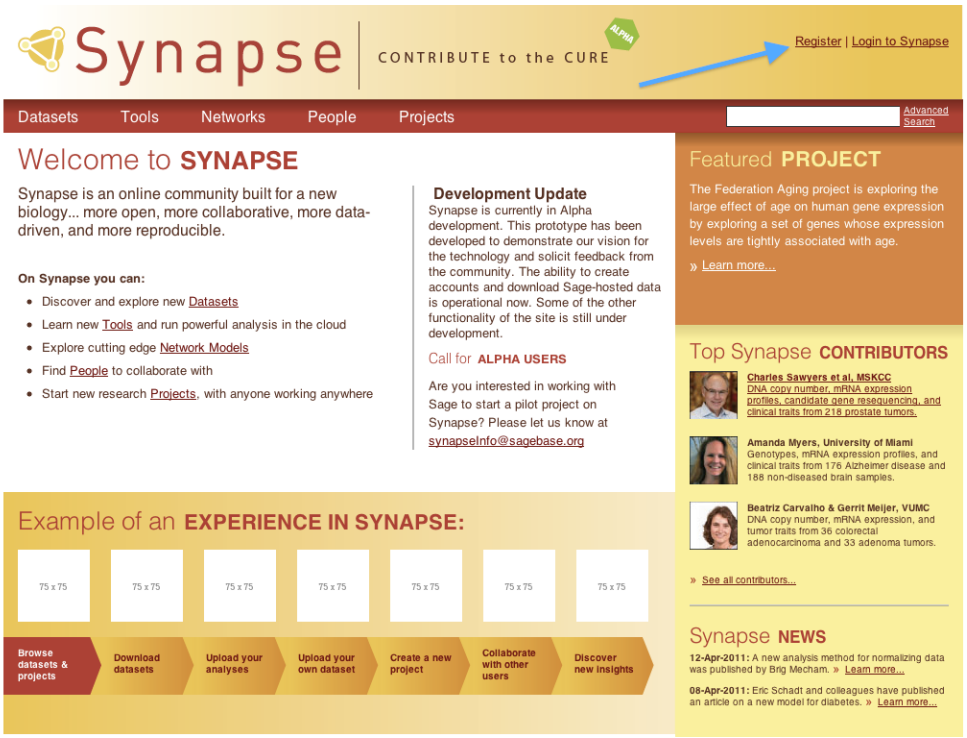


Figure 1: Register for a Synapse account and log in

Use the following R code to setup your Synapse work environment.

```
> library(predictiveModeling)
> # Change the values of these variables
> myName <- Sys.getenv("USER")
> myWorkingDirectory <- "."
> setwd(myWorkingDirectory)
```

Create a Synapse project to hold your analyses results. Be sure to type in your Synapse username and password when prompted from R.

```
> library(synapseClient)
> synapseLogin()
> project <- Project(list(
+   name=paste("Machine Learning Results - ", myName)
+   ))
> project <- createEntity(project)
> onWeb(project)
> analysis <- Analysis(list(
+     name="Elastic net versus custom methods",
+     description="Several Machine Learning methods run upon CCLE Data with Sanger Drug Response",
+     parentId=propertyValue(project, "id")
+     ))
> analysis <- createEntity(analysis)
> dataset <- Dataset(list(
+   name="Analysis Plots",
```

```
+   parentId=propertyValue(project, "id")
+   ))
> dataset <- createEntity(dataset)
```

Go back to `https://synapse-alpha.sagebase.org/` and find your newly created project. Click on "share" and share your project with the group AUTHENTICATED_USERS.

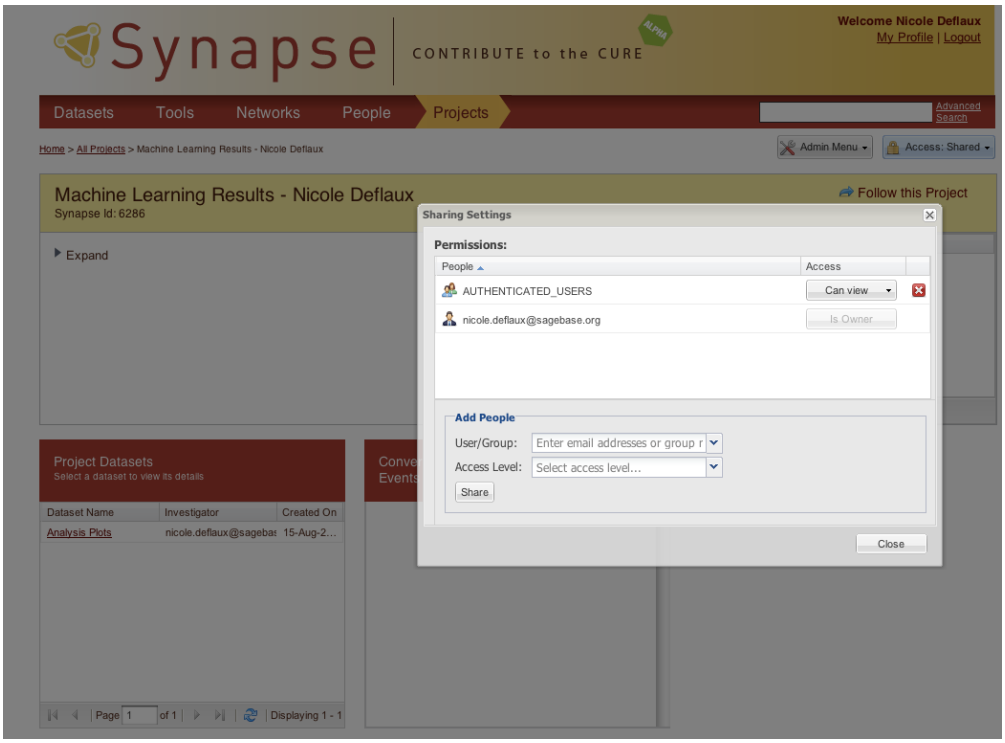


Figure 2: Find your project and share it with AUTHENTICATED_USERS

3 Load data from Synapse

Navigate to the "Cell Line Project" in Synapse `https://synapse-alpha.sagebase.org/#Project:5019`. Click on the two datasets listed there and their layers to get the IDs of the data layers for CCLE expression, CCLE copy number, CCLE oncomap, and Sanger drug response IC50s. Note that you can also browse data available in Synapse via the Synapse R Client. See the help documentation for synapseClient for more detail.

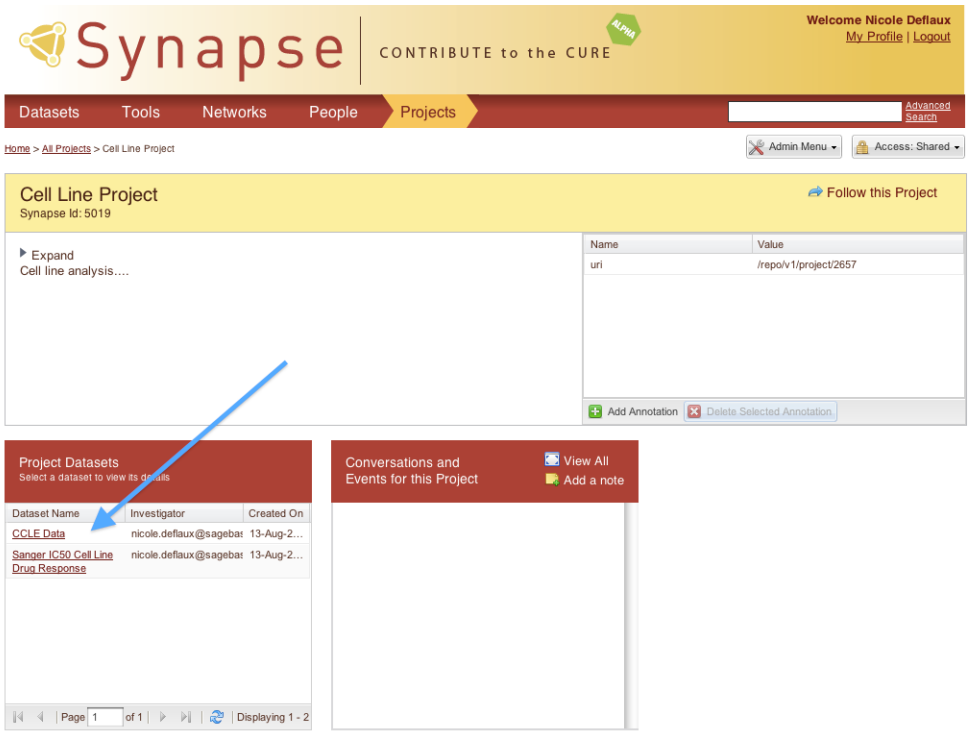


Figure 3: Find the CCLE and Sanger Dataset Layers in Synapse

```
> #### Load Data from Synapse ####
> idExpressionLayer <- "48344"
> expressionLayer <- loadEntity(idExpressionLayer)
> exprSet <- expressionLayer$objects$exprSet
> idCopyLayer <- "48339"
> copyLayer <- loadEntity(idCopyLayer)
> copySet <- copyLayer$objects$copySet
> idOncomapLayer <- "48341"
> oncomapLayer <- loadEntity(idOncomapLayer)
> oncomapSet <- oncomapLayer$objects$oncomapSet
> idSangerLayer <- "48337"
> sangerLayer <- loadEntity(idSangerLayer)
> sangerADF <- sangerLayer$objects$sangerADF

> print(colnames(pData(sangerADF)))

 [1] "Docetaxel"    "Gefitinib"    "CI1040"       "BIBW2992"     "PLX4720"      "Paclitaxel"
 [7] "HKI-272"      "Cyclopamine"  "Go_6976"      "Erlotinib"    "Geldanamycin" "AZ628"
[13] "Sorafenib"    "MK-0457"      "Imatinib"     "TAE684"       "PD173074"     "PF-2341066"
[19] "AZD0530"      "Rapamycin"    "Sunitinib"    "PHA665752"    "MG-132"
```

Combine the feature data into a single dataset. The number of features in the aggregate dataset should equal the number of features in the copy number data plus the number of features in the oncomap data.

```
> ds_features_cn_mut_ccle <- createAggregateFeatureDataSet(list(copy = copySet, mut = oncomapSet))
> checkEquals(nrow(ds_features_cn_mut_ccle), nrow(exprs(copySet)) + nrow(exprs(oncomapSet)))
```

```
[1] TRUE
```

We want to use the aggregate features to build predictive models of drug response. Run the function createFeatureAndResponseDataList, which finds the samples that are common between the feature data and response data and creates subsets of the feature and response datasets containing only these samples, in the same order. These new datasets are returned in a list with elements featureData and responseData. Check that the columns of the datasets returned in the list are the same.

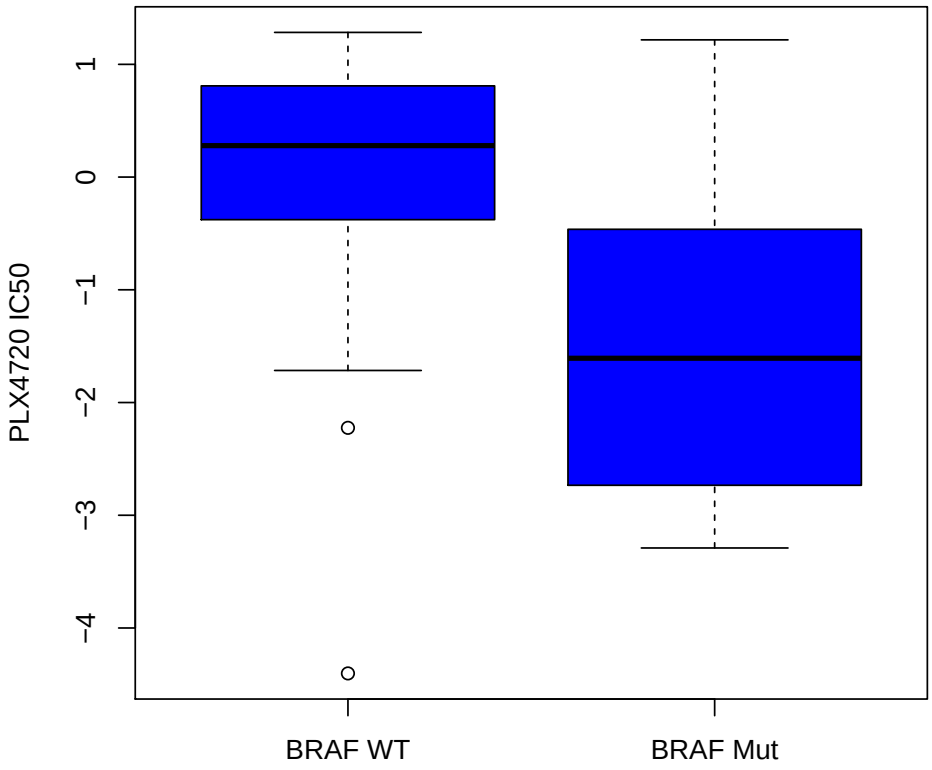
```
> dataSets_ccleFeaturesCnMut_sangerChems <- createFeatureAndResponseDataList(ds_features_cn_mut_ccle, san
> ls(dataSets_ccleFeaturesCnMut_sangerChems)

[1] "featureData"  "responseData"

> checkEquals(colnames(dataSets_ccleFeaturesCnMut_sangerChems$featureData), colnames(dataSets_ccleFeature

[1] TRUE
```

Normalize the data. The scale function only scales the columns of a matrix, so we transpose the matrices as input to the scale function and then transpose the results back to the original form. We want to build a predictive model on a single compound, so return to responseData_scaled only the row of responseData corresponding to the compound PLX4720.

```
> featureData_scaled <- t(scale(t(dataSets_ccleFeaturesCnMut_sangerChems$featureData)))
> responseData_scaled <- t(scale(t(dataSets_ccleFeaturesCnMut_sangerChems$responseData["PLX4720",,drop=FA
> print(dim(featureData_scaled))

[1] 23257   324

> print(dim(responseData_scaled))

[1]   1 324
```

PLX4720 is an inhibitor of mutant BRAF and is known to be selective for BRAF mutant tumors. Plot the IC50 values for cell lines with wild type and mutant BRAF to confirm that cell lines with mutant BRAF have lower IC50 values on average.

```
> boxplot(as.numeric(responseData_scaled) ~ featureData_scaled["BRAF_mut",], names = c("BRAF WT", "BRAF M
```



4 Fit elastic net model

Given that BRAF mutant cell lines have lower IC50 values for PLX4720, can we develop a predictive model that can discover BRAF mutations as predictive of PLX4720 in an unbiased analysis of all of the feature data?

In this demo, we can build predictive models using any object that is a sublcass of PredictiveModel. We define all classes using the new R5 syntax. For documentation see help(setRefClass) or this webpage: `https://github.com/hadley/devtools/wiki/R5`.

The generic base class PredictiveModel defines functions train() and predict(), which all subclasses must implement. Later, we will provide an example of implementing a custom class using this interface. We also provide support for most commonly used machine learning algorithms through the class CaretModel, which provides a wrapper around the R predictive modeling package caret.

To instantiate an object implementing a method supported by caret, simply call the constructure for CaretModel with the argument modelType set to a string referring to a model supported by caret. For documentation see: `http://cran.r-project.org/web/packages/caret/vignettes/caretTrain.pdf`

In this vignette, we will apply a regularized regression algorithm, called elastic net regression, that is designed to work for problems with many more variables than observations (referred to as $p \gg n$), which is the case in our example where we have tens of thousands of features and only hundreds of samples. Users

are encouraged to experiment on their own with other modeling algorithms, as well as customized novel algorithms, as described later in this vignette.

Elastic net overcomes the $p \gg n$ problem by imposing a complexity penalty on the parameters and selecting the strength of this penalty using a cross validation procedure and selecting the parameter leading to the lowest test error in the training data.

Let's construct an elastic net model, specified using the "glmnet" parameter in caret. For elastic net, and all other models that require parameter tuning, the test error will be computed across many different combinations of training parameters. This grid of tuning parameters is specified in the tuneGrid parameter to fitPredictiveModel. If it is not specified it will be created by default. However, we have included a customized tuneGrid for elastic net that sets one of the parameters (alpha) to 1 (corresponding to lasso regression) and varies the other parameter (lambda) across a range of values. The lambda parameter controls the strength of the penalty imposed on the complexity of the model – in this case, the L1 norm, or sum of absolute values of the coefficients.

To train the predictive model, call the train() function on the instantiated CaretModel object, with arguments corresponding to the featureData, responseData, and optionally the tuning parameter grid.

All source code is contained in the directory COMPBIO/trunk/predictiveModeling/R. To explore source code for the CaretModel class, look at the file CaretModel.R.

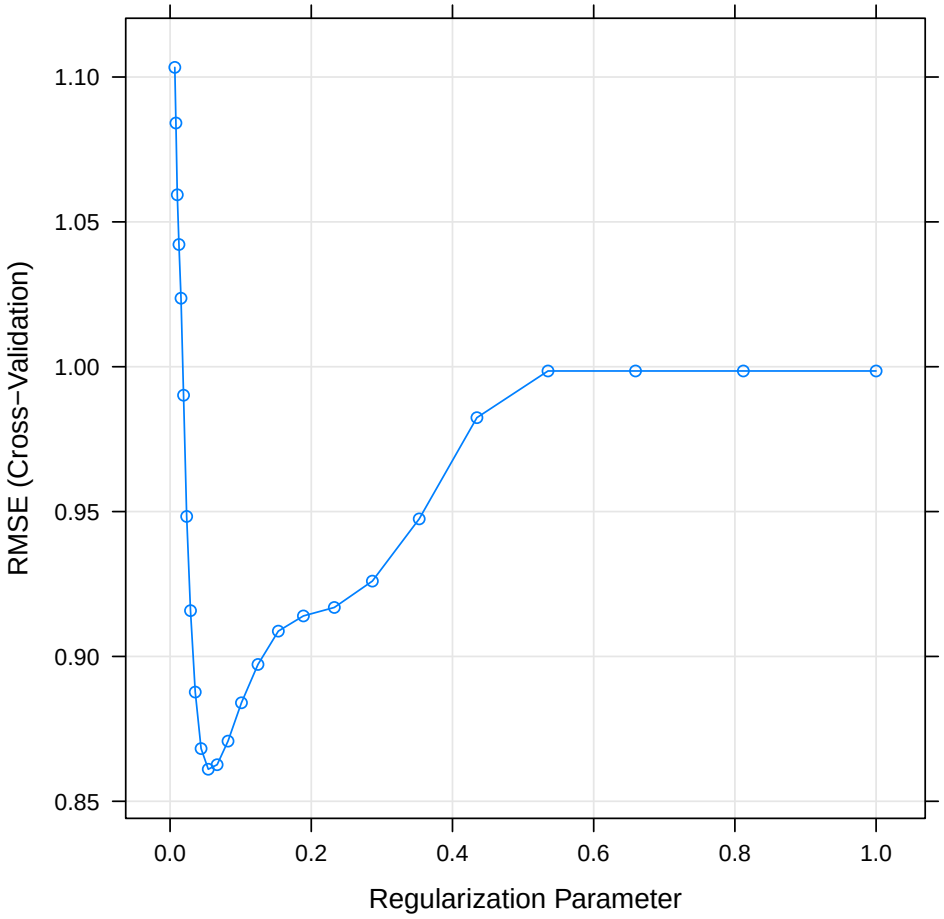
```
> predictiveModel_eNet <- CaretModel$new(modelType = "glmnet")
> predictiveModel_eNet$train(t(featureData_scaled), t(responseData_scaled), tuneGrid=createENetTuneGrid(a

Fitting: alpha=1, lambda=NA
Fitting: alpha=1, lambda=NA
Fitting: alpha=1, lambda=NA
Aggregating results
Selecting tuning parameters
Fitting model on full training set
```

Now the model field in predictiveModel_eNet should contain an elastic net model fit to the data. This is implemented as a wrapper of a train object in caret. To see this, we can extrat the caret object by calling rawCaretModel().

The default plotting method for train objects displays the test error associated with the model parameters used in tuneGrid. To ensure that we have tested the proper range of tuning parameters we would expect the lowest test error to occur at intermediate tuning parameter values, with larger test errors at the low and high tuning parameter values, which cause the model to be overfit and underfit.

The model selected as optimal is returned in the finalModel element of the train object. This model is fit using the tuning parameters that were computed as optimal, and these optimal tuning parameters are specified in the tuneValue element of finalModel. The optimal alpha is 1 (since this is the only alpha value we tested). Ensure that the optimal lambda value corresponds to the lambda producing the minimal test error, as seen in the plot below.

```
> caretModel_eNet <- predictiveModel_eNet$rawCaretModel()
> print(plot(caretModel_eNet))
> print(caretModel_eNet$finalModel$tuneValue)

   .alpha    .lambda
11      1 0.05411377
```

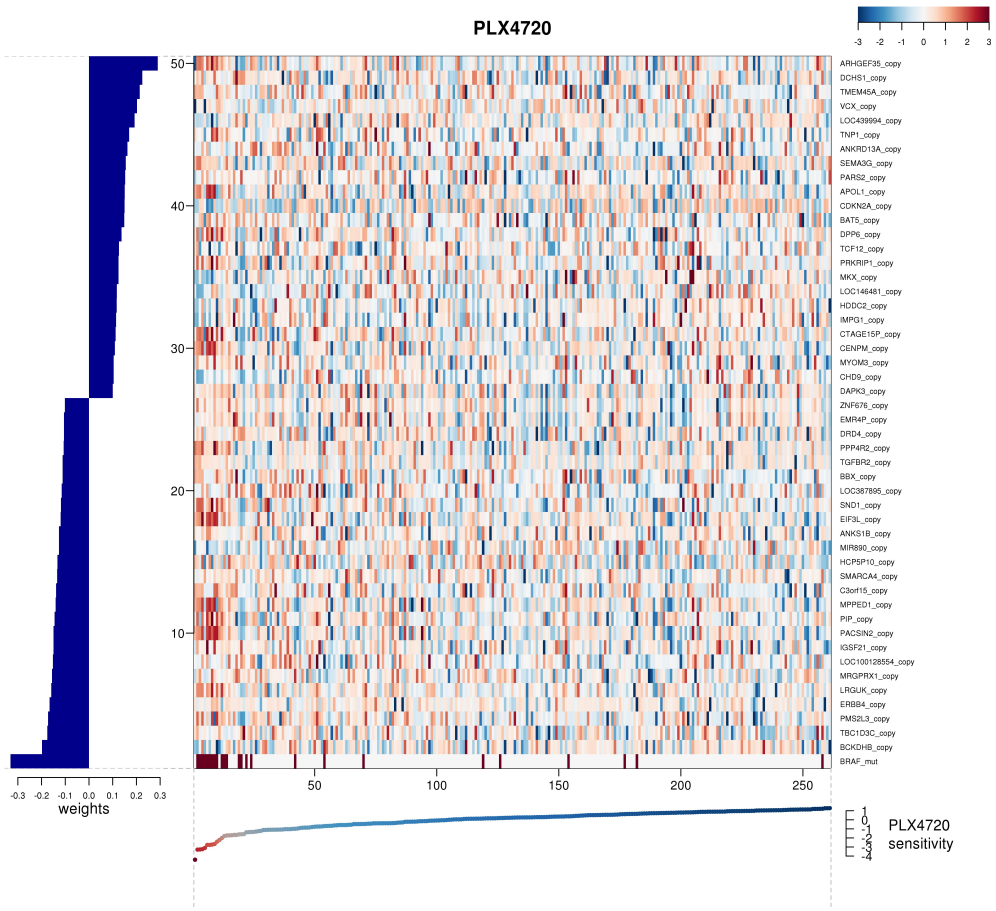


Elastic net computes coefficients associated with a subset of the features, such that response is modeled as a weighted sum of these feature values. We can visualize the results of the model by plotting a heatmap of the inferred predictive features and the inferred weights associated with each feature. To do this, extract the coefficients of the model and call the plotPredictiveModelHeatmap function. Note that the elastic net model returns the entire solution path in the beta element, where the final beta values are at the termination of the solution path, and therefore the last column of the beta element.

The colored line at the bottom of the plot displays the sensitivity of each cell line to PLX4720 (in normalized IC50 values), sorted from the most to least sensitive cell line. The heatmap displays each inferred predictive feature, with columns representing cell lines in the same order as the sensitivity plot on the bottom, with red corresponding to high feature values (e.g. mutations or amplifications) and blue corresponding to low feature values. The bar plot to the left of the heatmap displays the weights associated with each feature.

Verify that BRAF mutation is inferred as the most predictive feature.

```
> coefs_eNet <- caretModel_eNet$finalModel$beta[, ncol(caretModel_eNet$finalModel$beta)]
> outputFileElasticNet <- 'PLX4720_ElasticNetModel.jpg'
> plotPredictiveModelHeatmap(coefs_eNet, featureData_scaled, responseData_scaled, outputFile=outputFileEl

RStudioGD
        2
```



Store the Elastic net analysis result in Synapse.

```
> elasticNetLayer <- Layer(list(
+                          name="ElasticNet Results for PLX4720",
+                          type="M",
+                          parentId=propertyValue(dataset, "id")))
> elasticNetLayer <- addFile(elasticNetLayer, outputFileElasticNet)
> storeEntity(elasticNetLayer)
> step1 <- stopStep()
> onWeb(step1)
> propertyValue(step1, 'name') <- "Single run using elastic net"
> propertyValue(step1, 'description') <- "I found that ... looked very interesting due to ..."
> step1 <- updateEntity(step1)
> step2 <- startStep(analysis)
> propertyValue(step2, 'name') <- "Cross validation using elastic net"
> propertyValue(step2, 'input') <- propertyValue(step1, "input")
> step2 <- updateEntity(step2)
> onWeb(analysis)
>
>
```

5 Evaluate model performance

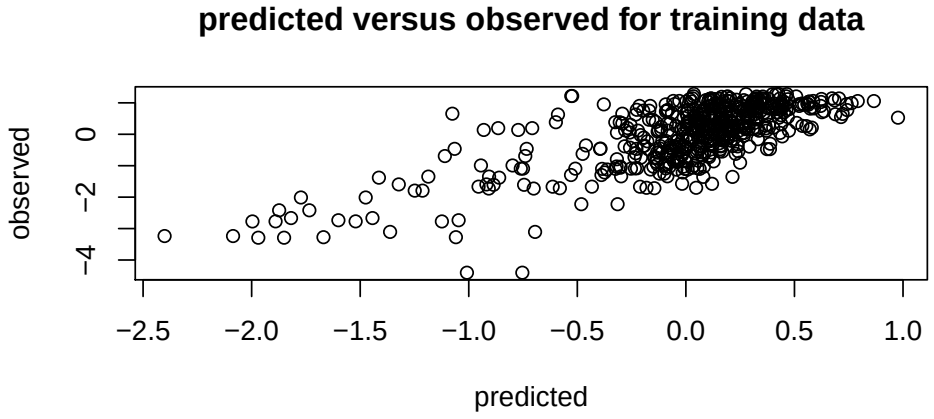
We can evaluate the performance of a predictive model using a cross validation procedure in which parts of the data are successively held out, the model is fit using the remaining data, and the predictions made from the held out features are compared against the held out response data. This cross validation procedure is implemented in the crossValidatePredictiveModel() function. Call this function with the arguments corresponding to the featureData, responseData, and any object that is a subclass of PredictiveModel. Optionally pass arguments for the tuning parameter grid and the number of cross validation folds.

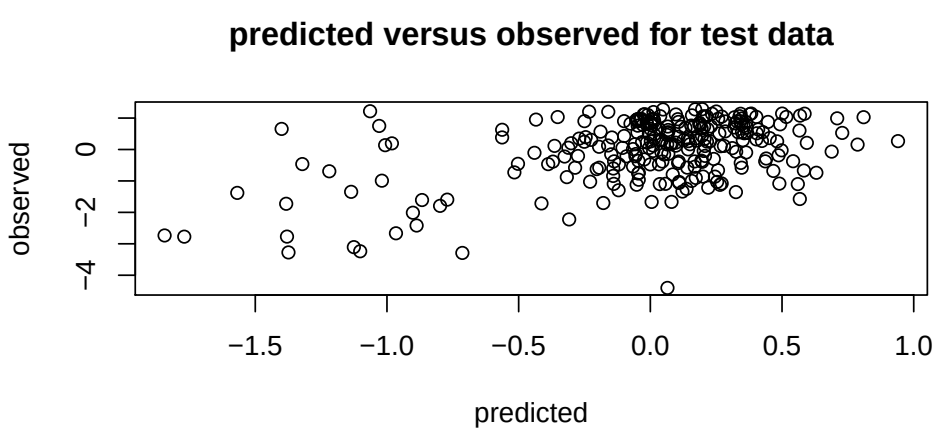
```
> cvResults_eNet <- crossValidatePredictiveModel(t(featureData_scaled), t(responseData_scaled), model=Car
```

```
Fitting: alpha=1, lambda=NA
Fitting: alpha=1, lambda=NA
Fitting: alpha=1, lambda=NA
Aggregating results
Selecting tuning parameters
Fitting model on full training set
Fitting: alpha=1, lambda=NA
Fitting: alpha=1, lambda=NA
Fitting: alpha=1, lambda=NA
Aggregating results
Selecting tuning parameters
Fitting model on full training set
Fitting: alpha=1, lambda=NA
Fitting: alpha=1, lambda=NA
Fitting: alpha=1, lambda=NA
Aggregating results
Selecting tuning parameters
Fitting model on full training set
```

The value returned from this function is an object of class PredictiveModelPerformance (see PredictiveModelPerformance.R). This object contains the predicted and observed values for the training and testing data, concatenated across the cross validation folds. Methods of this object are used to evaluate the performance of predictive models, and provide objective benchmarks to compare different models. To visualize the performance of the model on the training and testing data, call plotPredAndObs() on the returned object.

```
> cvResults_eNet$plotPredAndObs()
```



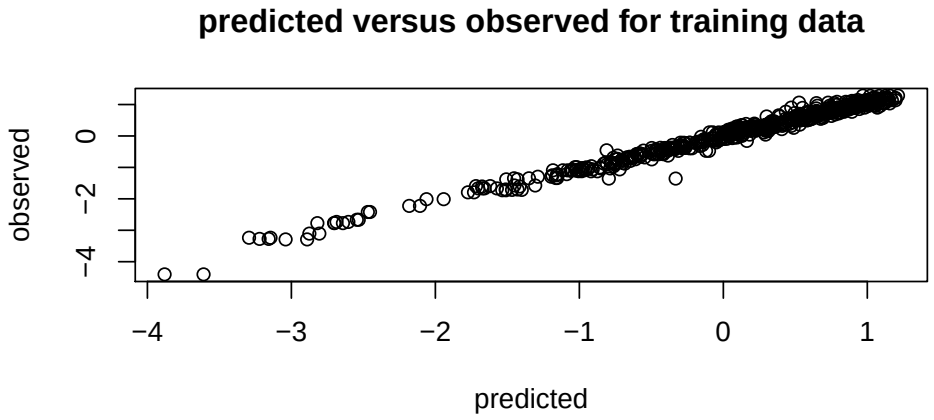


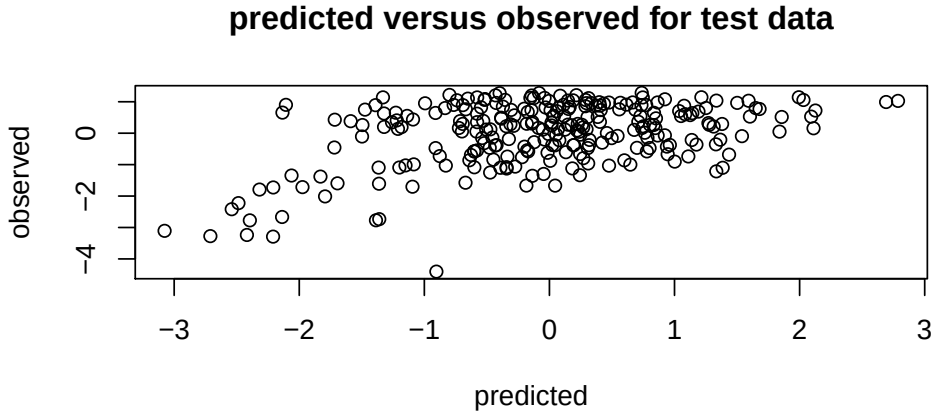
The predictions generalize pretty well from the training to the test data. This is due to the complexity penalty, which controls for overfitting. What would have happened if we constructed a model with a lower complexity penalty and allowed the model to overfit the data? Try this by passing low lambda values as parameters to the createENetTuneGrid function.

```
> cvResults_eNet_lowLambda <- crossValidatePredictiveModel(t(featureData_scaled), t(responseData_scaled),

Fitting: alpha=1, lambda=NA
Fitting: alpha=1, lambda=NA
Fitting: alpha=1, lambda=NA
Aggregating results
Selecting tuning parameters
Fitting model on full training set
Fitting: alpha=1, lambda=NA
Fitting: alpha=1, lambda=NA
Fitting: alpha=1, lambda=NA
Aggregating results
Selecting tuning parameters
Fitting model on full training set
Fitting: alpha=1, lambda=NA
```

```
Fitting: alpha=1, lambda=NA
Fitting: alpha=1, lambda=NA
Aggregating results
Selecting tuning parameters
Fitting model on full training set
```

```
> cvResults_eNet_lowLambda$plotPredAndObs()
```

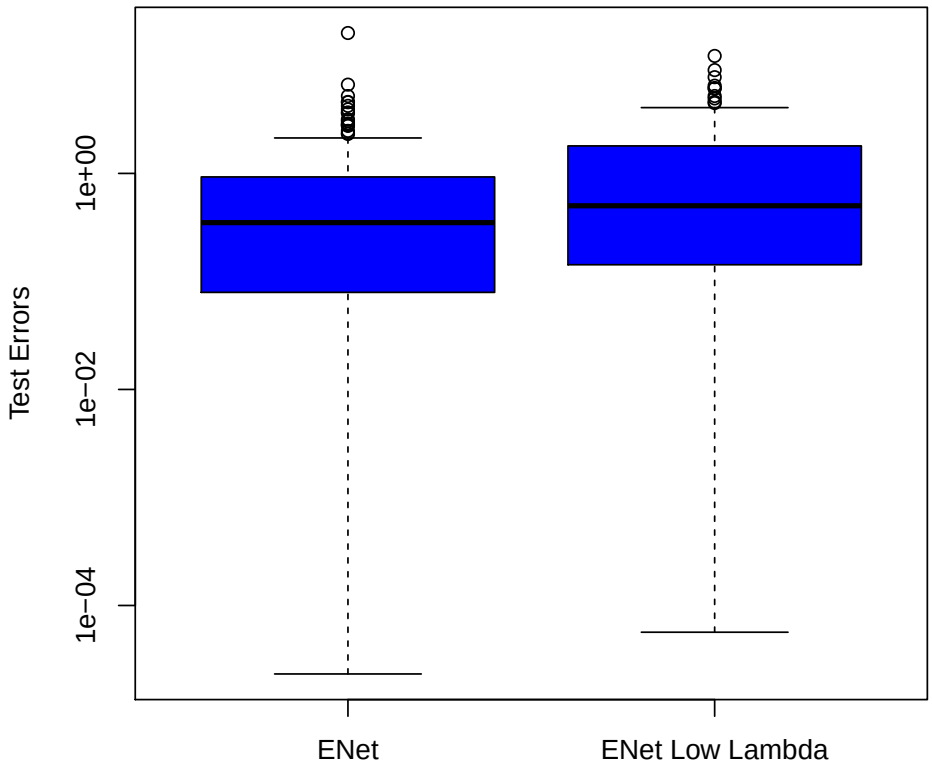




With out adequately controlling for overfitting, the model fits the training data very well, but does not generalize well to the test data and is therefore unlikely to give accurate predictions for new samples.

```
> boxplot(cvResults_eNet$getTestError(), cvResults_eNet_lowLambda$getTestError(), log="y", names = c("ENe
> t.test(cvResults_eNet$getTestError(), cvResults_eNet_lowLambda$getTestError())

        Welch Two Sample t-test

data:  cvResults_eNet$getTestError() and cvResults_eNet_lowLambda$getTestError()
t = -2.5927, df = 519.674, p-value = 0.009789
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -0.61889464 -0.08531301
```

```
sample estimates:
mean of x mean of y
 0.779550  1.131654
```



The test error using the optimal complexity penalty is statistically significantly lower than the test error using a low complexity penalty. This provides an objective test to evaluate the relative performance of different models. Note that this single example contains relatively few sensitive cell lines and robust assessments of relative performance will require comparisons across many different predictions (e.g. sensitivity to different drugs).

6 Advanced topic: Build a custom model

An active area of research in computational biology applications in cancer medicine is to develop predictive models able to discover relationships between cancer genotypes and drug response, especially for drugs where the association is weaker than between BRAF and PLX4720.

The predictiveModeling package provides a customizable interface for developing novel predictive models and testing their performance against other predictive models. In order to define a custom predictiveModel, we need to create a subclass of the abstract PredictiveModeling class and define two functions – train() and predict(). The train() function uses a training set of feature and response data to fit the parameters of

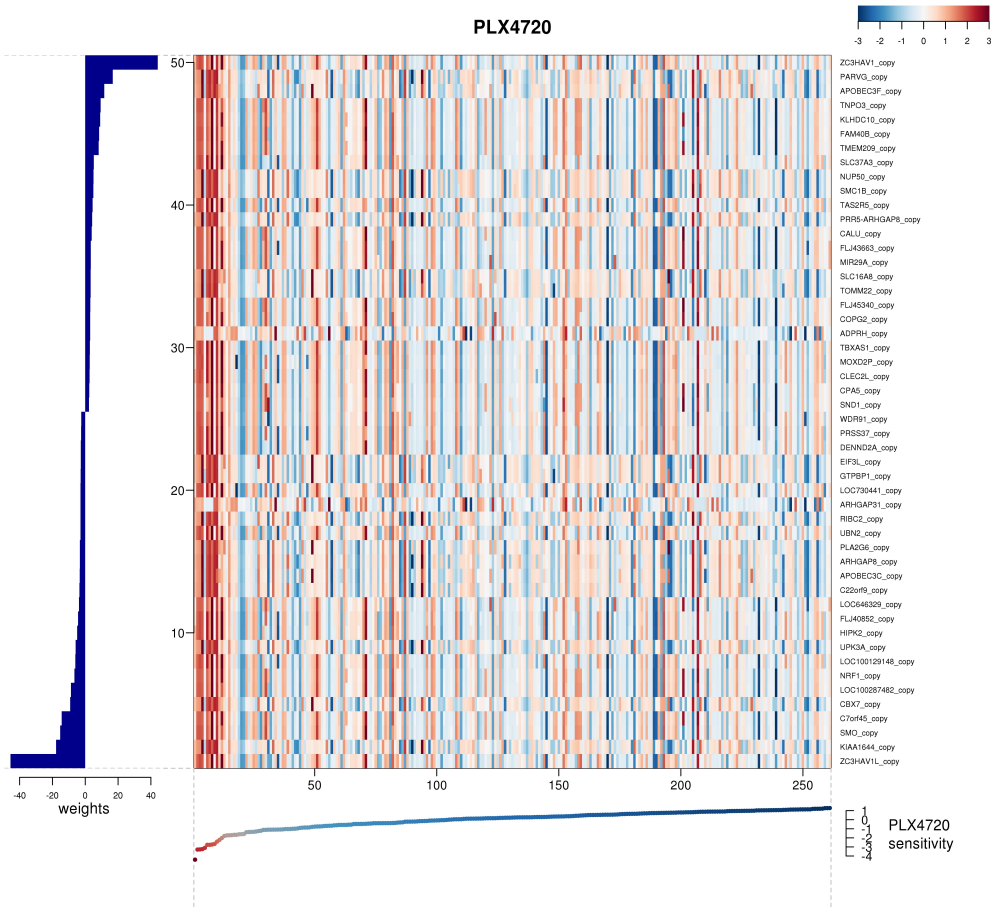
the custom model. The predict() function uses the model that was built in train() to predict the expected response for new observations that were not used in the model fitting procedure.

A simple example of a custom class is provided in MostCorrelatedFeatures.R. This class simply builds a linear regression model using a subset of features that are most correlated with response (by default, 200 features). This model imposes no penalty on parameter complexity and is likely to overfit the data. Indeed, we see that BRAF mutation is no longer the feature with the highest weight.

This object can be used the same way as the CaretModel objects described above. For example, we can call the object's train() method

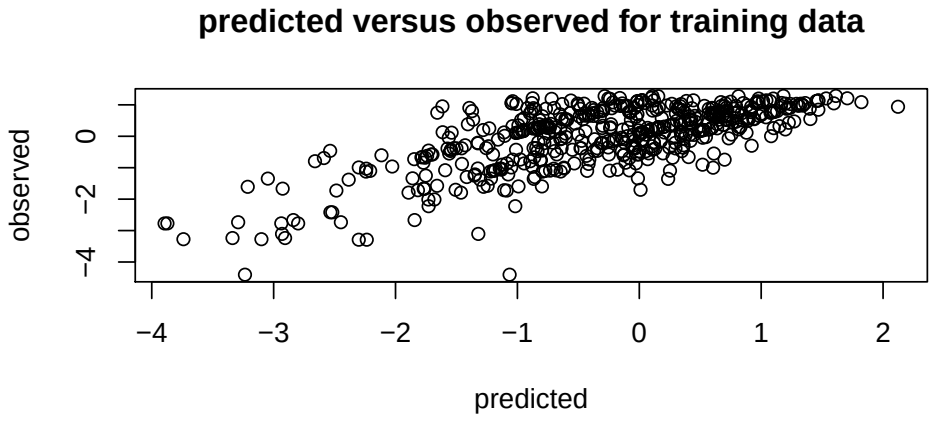
```
> predictiveModel_mostCorr <- MostCorrelatedFeatures$new()
> predictiveModel_mostCorr$train(t(featureData_scaled), t(responseData_scaled))
> outputFileMostCorr <- 'PLX4720_MostCorrModel.jpg'
> coefs_mostCorr <- predictiveModel_mostCorr$coefficients[2:length(predictiveModel_mostCorr$coefficients)
> plotPredictiveModelHeatmap(coefs_mostCorr, featureData_scaled, responseData_scaled, outputFile=outputFi
```

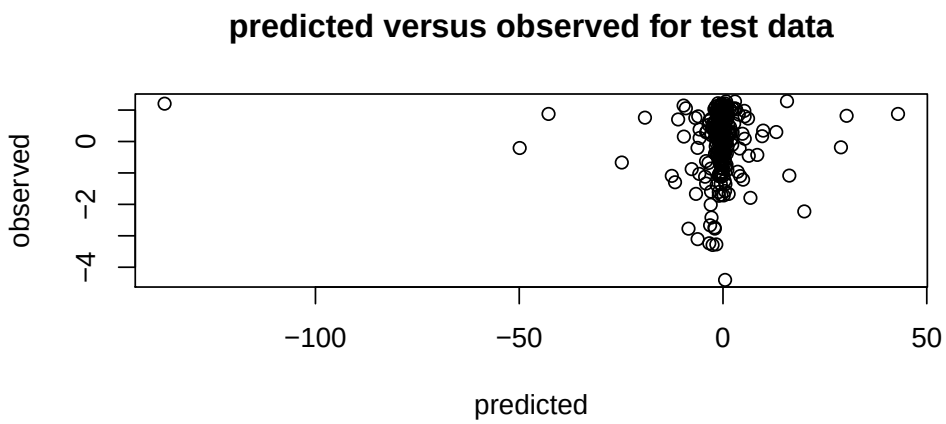
```
RStudioGD
        2
```



We can pass an object of class MostCorrelatedFeatures to crossValidatePredictiveModel(). Plotting the predicted vs. observed values for both the training and test data confirms the overfitting and shows that the training data is fairly well fit, but the test data is not.

```
> cvResults_mostCorr <- crossValidatePredictiveModel(t(featureData_scaled), t(responseData_scaled), model
> cvResults_mostCorr$plotPredAndObs()
```





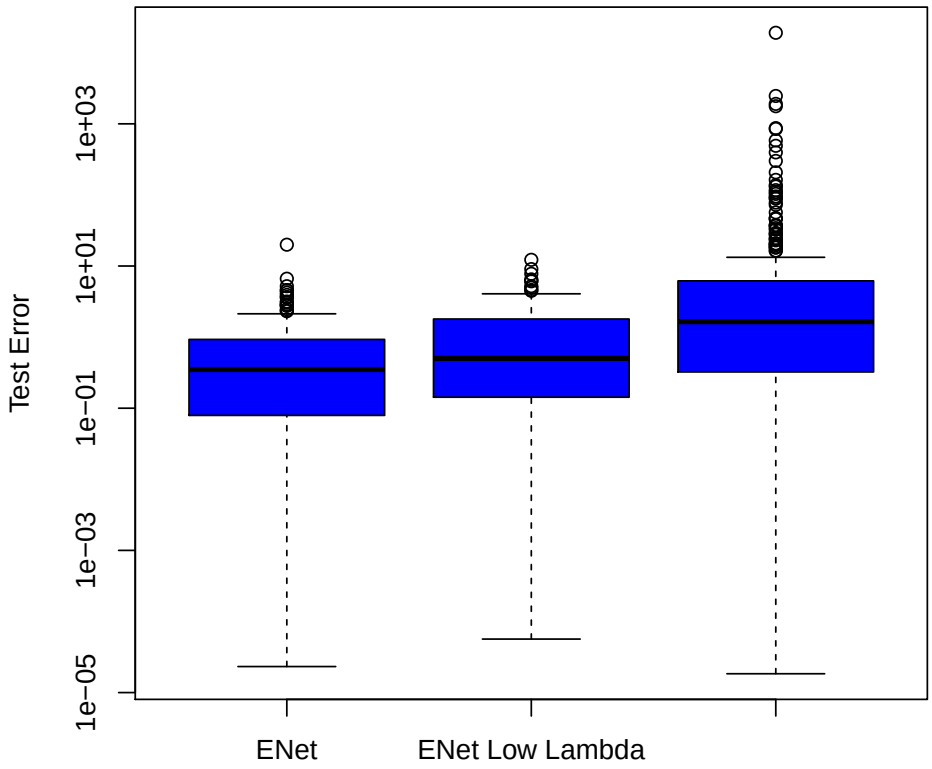
7 Compare the performance of different models

Putting it all together, we can compare the test errors from the 3 models. Compute the R squared of predicted vs. observed values for each model, and plot the full distribution of test errors for each model.

Indeed, we verify that performance progressively diminishes as we allow models to overfit the data. Elastic net with parameter optimization gives the lowest test error. Performance is decreased when running elastic net with a low complexity penalty, and further decreased by using the simple multiple regression model that does not penalize parameter complexity.

```
> cvResults_eNet$getR2()

[1] 0.2175988

> cvResults_eNet_lowLambda$getR2()

[1] 0.1968051

> cvResults_mostCorr$getR2()

[1] 0.0003786744
```

```
> boxplot(cvResults_eNet$getTestError(), cvResults_eNet_lowLambda$getTestError(), cvResults_mostCorr$getT
```



8 Next steps

Implement your own customized predictive models and use the cross validation evaluation procedures to test if your method achieves more accurate predictive performance than currently used methods. Try to beat the best methods!

Try out predictive models on other drugs and other datasets containing molecular feature and phenotypic data!

9 Session Information

```
> toLatex(sessionInfo())
```

- R version 2.14.0 (2011-10-31), `x86_64-unknown-linux-gnu`
- Locale: `LC_CTYPE=en_US.UTF-8`, `LC_NUMERIC=C`, `LC_TIME=en_US.UTF-8`, `LC_COLLATE=en_US.UTF-8`, `LC_MONETARY=en_US.UTF-8`, `LC_MESSAGES=en_US.UTF-8`, `LC_PAPER=C`, `LC_NAME=C`, `LC_ADDRESS=C`, `LC_TELEPHONE=C`, `LC_MEASUREMENT=en_US.UTF-8`, `LC_IDENTIFICATION=C`

- Base packages: base, datasets, graphics, grDevices, methods, stats, utils
- Other packages: affy 1.32.0, Biobase 2.14.0, caret 5.08-011, cluster 1.14.1, codetools 0.2-8, foreach 1.3.2, glmnet 1.7.1, iterators 1.0.5, lattice 0.20-0, Matrix 1.0-2, plyr 1.6, predictiveModeling 0.9.0, reshape 0.8.4, RUnit 0.4.26, synapseClient 0.8-5
- Loaded via a namespace (and not attached): affyio 1.22.0, BiocInstaller 1.2.1, compiler 2.14.0, digest 0.5.1, grid 2.14.0, preprocessCore 1.16.0, RCurl 1.8-0, RJSONIO 0.96-0, tcltk 2.14.0, tools 2.14.0, zlibbioc 1.0.0