NCBI SVM Example

Rachel Adams

Monday, February 2, 2015

Overview

As an example deliverable for upcoming weekly reports, this document describes the details of new models and their predictive performances. Each of the following reports will also contain comparisons to previously discussed models.

This Rmd file contains the source code used to perform data transformations and analytics as well as a description of the models' methods, parameters, performances, and conclusions.

In brief, this study found that Support Vector Machine (SVM) methods applied to an NCBI dataset of gene expression data can accurately (~80%) predict simulated survival outcomes.

Dataset

The original dataset was downloaded directly from the small-blue-round-cell tumor (SBRCT) microarray training set made available by Hastie, Tibshirani, and Friedman as part of their *Elements of Statistical Learning* book published online (http://statweb.stanford.edu/~tibs/ElemStatLearn/).

This dataset contains **63 training samples** of SBRCT gene expression data for **2318 genes**. The csv file contains one gene per row and one sample per column. Additional files can be downloaded that list the cancer class corresponding to each of the training samples, but they were not used for this study.

Instead, this study focused on predicting survival outcomes. Since survival outcomes were not available for this dataset, the dependent variable was generated using non-linear transformations of subsets of the independent variables. Generating known "right" answers helped test each model's accuracy and robustness, but this step will not be required in future analyses.

Support Vector Machines

Support Vector Machines (SVMs) are supervised learning methods that infer functions from training data where the correct labels or dependent values are known. SVMs can be used for classification (discrete outcomes) or for regression (continuous outcomes). A successful SVM algorithm will generalize from the training set to correctly predict the class or output value for unseen (testing) data. The accuracy of the model is based on the alignment of the predicted outcomes with the known outcomes. Choosing the right parameters for the SVM algorithm can greatly influence the performance of the model, so it is important to tune and cross-validate parameters of the SVM.

The most basic structural parameter of an SVM is the **kernel method**. Kernel methods are similarity functions that quantify the similarity between two points or features. They're extremely useful for high-dimensional computations because they do not require explicit, exhaustive calculations of all possible feature combinatons. Instead, they operate on an "instance-based" learning method that simply compares one input object to an object it has seen before. This approach is often called "the"kernel trick."

One of the most commonly used kernel methods is the Gaussian or ***radial basis function (RBF) kernel***. This method measures similarity between two samples **x** and **x'** using the squared Euclidean distance between the two feature vectors.

$$K(\mathbf{x}, \mathbf{x'}) = \exp(\gamma||\mathbf{x} - \mathbf{x'}||^2)$$
$$where$$
$$\gamma = -\frac{1}{2\sigma^2}$$

σ is a free parameter (a smaller σ^2 results in low bias, high variance while a larger σ^2 results in high bias, low variance), but γ is the parameter generally required for RBF functions.

Other common non-linear kernels include the following:

- Polynomial (homogeneous): $k(\mathbf{x_i}, \mathbf{x_j}) = (\mathbf{x_i} \cdot \mathbf{x_j})^d$
- Polynomial (inhomogeneous): $k(\mathbf{x_i}, \mathbf{x_j}) = (\mathbf{x_i} \cdot \mathbf{x_j} + 1)^d$
- Hyperbolic tangent: $k(\mathbf{x_i}, \mathbf{x_j}) = \tanh(\kappa\mathbf{x_i} \cdot \mathbf{x_j} + c)$, for some (not every) $\kappa > 0$ and $c < 0$

None of these non-linear methods scale well to large numbers of training samples or when the number of features far exceeds the number of training samples.

Linear kernel functions are much more amenable to large feature sets, generally faster to compute than non-linear methods, and are less-prone to overfitting compared to non-linear methods. There are no extra parameters to compute or tune for the linear kernel methods.

Regardless of whether one chooses a linear or non-linear kernel function, the **C (cost)** parameter is required for every SVM. Increasing C increases the complexity of the model and allows for a larger range of weights for the features, whereas lowering C decreases the complexity of the model and may allow for more misclassified/miscalculated outcomes. SVMs are often called *large-margin classifiers*, and the parameter C controls the width of this margin. Tuning this parameter is important to the overall success of the SVM prediction.

Method Description

This study implemented SVMs to predict survival outcomes on a dataset of 64 patients and 2300 genes. Survival outcomes were generated by selecting 10-20 columns of gene expression data and applying a random linear function to output the dependent variable. This approach was taken with the hopes that the SVM should be able to pick up on these 10-20 columns and accurately predict survival outcomes.

Initially, a subset of the dataset was extracted to contain 50 randomly selected genes. Out of this subset dataset, 21 patients were used for training while 43 patients were set aside for testing. Parameters were tuned for 2 types of kernels (RBF and linear), 10 models were generated for each dataset with the optimized parameters, and the R2 and MSE values of the true versus predicted survival outcomes were reported.

When the tuning parameters span 5 orders of magnitude, the analyses take approximately 3 minutes to run. Increasing the tuning parameters to 10 orders of magnitude or adding another kernel method, such as polynomial, takes 10-15 minutes to process.

Method Implementation

Loading Libraries

Two R libraries are required for this particular study of SVM models: **e1071** and **Rmisc**; two additional graphical libraries have also been used to read and write images for the report: **ggplot2** and **png**. The e1071 package is a popular, user-friendly implementation of modeling, predicting, and fitting SVMs that takes advantage of the C++ implementation of *libsvm*. Rmisc contains "miscellaneous" accessory functions that supplement basic R functions, including additional methods for quickly summarizing data objects. ggplot2 is a well-known graphics utility package that facilitates clean, customizable graphs. Once the plots have been created, the png package allows the report to read in and display specific images by name. This way, the program can be run with multiple parameters, generate separate graphs, and display the results side-by-side.

```
## load/install required libraries
repos="http://cran.rstudio.com/"
require(e1071, quietly=T) || install.packages("e1071", repos)
```

```
## [1] TRUE
```

```
require(ggplot2, quietly=T) || install.packages("ggplot2", repos)
```

```
## [1] TRUE
```

```
require(plyr, quietly=T) || install.packages("plyr", repos)
```

```
## [1] TRUE
```

```
require(Rmisc, quietly=T) || install.packages("Rmisc", repos)
```

```
## [1] TRUE
```

```
require(png, quietly=T) || install.packages("png", repos)
```

```
## [1] TRUE
```

Reading in Data

The current dataset is a csv file without headers. It contains one gene per row and one patient per column. There is an optional parameter `max_genes` to set the max number of genes to be stored in the main data table. Its default value (-1) allows all of the genes to be used for analyses.

```
## function to read in data
## two optional arguments: data_file and max_genes
##   - data_file by default "ncbi.data"
##   - max_genes by default -1 (no max; read all rows);
##     if specified, limits the number of rows
## returns a data frame
read_data_from_file <- function(data_file = "ncbi.data", max_genes = -1) {

  ## read in data from file
  table1 <- read.table(data_file, header=F)

  ## select a subset of genes to work with
  if(max_genes > 1 && max_genes <= nrow(table1)) {
    table1 <- table1[1:max_genes,]
    }

  table1
  }
```

Generating Survival Outcomes

Although this dataset did not come with survival information, generating these values provides substantial advantages for model development and evaluation. For example, controlling whether these outcomes are completely random or directly calculable from the input variables provides extra insight into the inferred models' robustness.

Two main approaches were taken for generating survival outcomes: linear and non-linear transformations of a subset of variables. The function that generates survival times contains an internal variable `{r} method_opt` that allows the user to select which transformation to apply.

```r
## function to generate a continuous value for the dependent variable
##   based on a linear or non-linear transformation of 10-20 columns (genes)
## one required argument: data table with variables in columns, samples in rows
## returns the input table with an additional column (survival_days)
generate_survival_info <- function(table1) {

  ## add the new column with default value 0
  table1$survival_days <- 0

  ## transformation method can be 'nonlinear' or 'linear'
  method_opt = "linear"
  if(method_opt == "nonlinear") {

    ## non-linear transformation has two options:
    ##   - "a": additional computation generates larger range of outcomes
    ##   - "b": generates smaller range of outcomes
    opt = "b"
    if(opt == "a") {
      ## choose 10-20 variables (columns) to USE
      num_cols <- sample(10:20, 1)
      cols_to_use <- sample(1:(ncol(table1)-1), num_cols, replace=T)

      ## select a random coefficient for each column to change
      random_coefs <- sample((-50):50, num_cols, replace=T)

      ## apply non-linear transformations to selected columns
      ## save the output in the new column, survival_days
      for (iCol in 1:num_cols) {
        table1$survival_days <- table1$survival_days +
          table1[, cols_to_use[iCol]] * random_coefs[iCol]
        }
      table1$survival_days <- table1$survival_days + rnorm(nrow(table1))
      }

    ## choose 10-30 additional variables (columns) to USE
    aFewMoreVars <- sample(10:30, 1)
    moreColsToUse <- sample(1:(ncol(table1)-1), aFewMoreVars)
    newCol <- 0
    for (iCol in moreColsToUse) {
      newCol <- newCol + table1[, iCol] * rnorm(1)
      }

    table1$survival_days <- table1$survival_days + exp(newCol)
    ## else linear transformation
    } else {
      ## choose 10-20 variables (columns) to USE
      num_cols <- sample(10:20, 1)
      cols_to_use <- sample(1:(ncol(table1)-1), num_cols, replace=T)

      ## select a random coefficient for each column to change
      random_coefs <- sample((-50):50, num_cols, replace=T)

      ## apply linear transformations to selected columns
      ## save the output in the new column, survival_days
      for (iCol in 1:num_cols) {
        table1$survival_days <- table1$survival_days +
          table1[, cols_to_use[iCol]] * random_coefs[iCol]
        }

      table1$survival_days <- table1$survival_days + rnorm(nrow(table1))
      }

  table1
  }
```

Optimizing Models

In order to develop an accurate model, two main factors need to be considered: the kernel type and their accompanying parameters. The e1071 package supports four main kernel methods: "radial" (Gaussian), "linear", "polynomial", and "sigmoid". Each of the kernel types requires a cost (or complexity) parameter, which allows the user to indicate how many misclassifications are acceptable. In general, larger cost parameters can lead to underfitting, whereas smaller costs can result in overfitting. The non-linear kernel methods (radial, polynomial, and sigmoid) all require an additional gamma parameter. Polynomial kernels also make use of the degree parameter.

Tuning parameters involves generating many models over a range of parameter values and looking at the performance (measured as error). Optimized parameter values produce better performance metrics with lower errors.

```
## function to find the best model (kernel) type and tune its parameters
## seven required arguments: train_set, max_gamma, min_gamma, max_cost, min_cost,
##   max_degree, min_degree
## returns the model's kernel and parameters with the best performance
find_best_params<- function(train_set,
                            max_gamma, min_gamma,
                            max_cost, min_cost,
                            max_degree, min_degree) {

  ## possible kernel_types: "radial", "linear", "polynomial", "sigmoid"
  kernel_types <- c("radial", "linear")

  ## save the best kernel name and its performance (error)
  best_performance <- 10^15
  best_kernel <- kernel_types[1]

  ## save performances for each combination of parameters for all kernel methods
```

```r
all_tuned_kernels <- as.data.frame(rbind(rep(0, 2), rep(0, 2)))
colnames(all_tuned_kernels) <- c("ParamValue", "Error")

## use Rmisc's summarySSE function to give count, mean, standard deviation,
##   standard error of the mean, and confidence interval (default 95%)
##   for a variable (performance) grouped by other variables (cost, gamma, etc)
all_tuned_kernels <- summarySE(all_tuned_kernels, measurevar = "Error",
                               groupvars=c("ParamValue"), na.rm=T)
all_tuned_kernels$Parameter <- NA
all_tuned_kernels$kernel <- NA

## iterate through each kernel type
for(i in 1:length(kernel_types)) {

  ## save the parameters and performances for this particular kernel
  all_tune_param_values <- data.frame()

  ## tune the kernel X times to get enough data for error bars (avg & std err)
  for(j in 1:10) {

    ## tune parameters for the specific kernel type
    tune_output <- tune_model(train_set, kernel_types[i],
                              max_gamma, min_gamma,
                              max_cost, min_cost)

    all_tune_param_values <- rbind(all_tune_param_values,
                                   tune_output$performances)
    }

  ## extract the cost variable and its associated errors
  cost_table <- unique(all_tune_param_values[,c('cost','error')])
  colnames(cost_table) <- c("Cost", "Error")
  cost_summary <- summarySE(cost_table, measurevar="Error",
                            groupvars = c("Cost"), na.rm=T)
  cost_summary$Parameter <- "Cost"
  colnames(cost_summary)[1] <- "ParamValue"
  cost_summary <- na.omit(cost_summary)

  if(kernel_types[i] != "linear") {

    ## extract the gamma variable and its associated errors
    gamma_table <- unique(all_tune_param_values[,c('gamma','error')])
    colnames(gamma_table) <- c("Gamma", "Error")
    gamma_summary <- summarySE(gamma_table, measurevar="Error",
                               groupvars = c("Gamma"), na.rm=T)
    gamma_summary$Parameter <- "Gamma"
    colnames(gamma_summary)[1] <- "ParamValue"
    summary_complete <- rbind(cost_summary, gamma_summary)

    if(kernel_types[i] == "polynomial") {

      ## extract the degree variable and its associated errors
      degree_table <- unique(all_tune_param_values[,c('degree','error')])
```

```
        colnames(degree_table) <- c("Degree", "Error")
        degree_summary <- summarySE(degree_table, measurevar="Error",
                                    groupvars = c("Degree"), na.rm=T)
        degree_summary$Parameter <- "Degree"
        colnames(degree_summary)[1] <- "ParamValue"
        summary_complete <- rbind(cost_summary, degree_summary)

        }
      } else {
        summary_complete <- cost_summary
        }

    summary_complete$kernel <- kernel_types[i]
    all_tuned_kernels <- rbind(all_tuned_kernels, summary_complete)

    ## plot tuning parameters for a kernel
    plot_model_tuning_param_performance(tune_output, kernel_types[i])

    ## determine which of the models is the best (lowest error rate)
    ## save the information about the kernel, parameters, and performance
    if(best_performance > tune_output$best.performance) {
      best_params <- tune_output
      best_kernel <- kernel_types[i]
      best_performance <- tune_output$best.performance
      }

    }

  compare_kernels_plot <- plot_all_tuned_kernels(all_tuned_kernels)
  compare_kernels_plot

  ## return the kernel name and parameters for the best model
  return (as.list(c(best_kernel, best_params, compare_kernels_plot)))

  }

## function to tune the parameters for a specific kernel
tune_model <- function(train_set,
                       kernel_type = "radial",
                       max_gamma = -2, min_gamma = -6,
                       max_cost = 3, min_cost = -6,
                       max_degree = 5, min_degree = 2) {

  if(kernel_type == "linear") {

    tuned <- tune.svm(survival_days ~ ., data = train_set,
                      kernel = kernel_type,
                      cost = 10^(min_cost:max_cost))

    } else if(kernel_type == "polynomial") {

      tuned <- tune.svm(survival_days ~ ., data = train_set,
                        kernel = kernel_type,
                        gamma = 10^(min_gamma:max_gamma),
                        cost = 10^(min_cost:max_cost),
                        degree = (min_degree:max_degree))

      } else {

        tuned <- tune.svm(survival_days ~ ., data = train_set,
                          kernel = kernel_type,
                          gamma = 10^(min_gamma:max_gamma),
                          cost = 10^(min_cost:max_cost))
        }

  }
```

Generating graphs

There are 4 main graphs generated for this analysis: two plots of the trained model's residual values, an example linear regression of a model's predicted and true values, and a plot of tuning parameters' values versus performance (error).

```
## function to generate 2 plots (boxplot and histogram)
## depicts residuals of the svm model's fitted data compared to the training
##   set's true survival outcomes
plot_model_residuals <- function(best_tuned_kernel, svm.model) {

  xlab <- paste("Model Residuals from", best_tuned_kernel,
                "kernel on a dataset of", max_genes, "genes")

  ## make a boxplot of residual values
  g <- ggplot() + geom_boxplot(aes(x="Model Residuals",y=svm.model$residuals))
  filename <- generate_filename(dir="nonlinear_output",best_tuned_kernel,
                                "model_residuals",".png")
  ggsave(filename, g, height=5,width=9)

  ## make a histogram of residual values
  g <- ggplot() + geom_histogram(aes(svm.model$residuals))
  filename <- generate_filename(dir="nonlinear_output",best_tuned_kernel,
                                "model_residuals_hist",".png")
  ggsave(filename, g, height=5,width=9)

  }

## function to generate linear reg plot of predicted vs true survival outcomes
## label includes an adjusted R2 value as an approx for the accuracy of the model
plot_prediction_vs_true_r2 <- function(best_tuned_params, best_tuned_kernel,
                                       svm.pred, test_set) {

  filename <- generate_filename(dir="nonlinear_output",best_tuned_kernel,
                                "predict_corr_r2",".png")
  png(filename)
  params_with_labels <- as.vector(sapply(names(best_tuned_params),
                                         function(i)
                                           paste(i, ":",
                                                  best_tuned_params[i][[1]])))
  plot(test_set$survival_days, svm.pred,
       main = paste(best_tuned_kernel, params_with_labels, sep=", "),
       xlab = "True Survival Days", ylab = "Predicted Survival Days")

  par(new=T)

  fit2 <- lm(svm.pred ~ test_set$survival_days)
  abline(fit2)
  legend("topleft", bty="n",
         legend=paste("R2 is", format(summary(fit2)$adj.r.squared, digits=4)))

  dev.off()
  }

## function to generate a plot of a model's performance (x-axis) vs a range of
##   tuning parameters (y-axis)
## for linear models, the y-axis is the log10 of the cost parameter
## for non-linear models, there is a second y-axis with the log10 of the gamma
plot_model_tuning_param_performance <- function(tune_output, best_tuned_kernel){

  x <- tune_output$performances$error
  y1 <- log(tune_output$performances$cost, 10)

  filename <- generate_filename(dir="nonlinear_output",best_tuned_kernel,
                                "model_tuning_param_perf",".png")
  png(filename)

  plot(x, y1, col="red",
       main = "Tuning Model's Parameters to Reduce Error",
       xlab = "Error", ylab = "log10(Cost)",
       xlim=c(min(x)-10, max(x)+10))

  if(best_tuned_kernel != "linear") {
    y2 <- log(tune_output$performances$gamma, 10)
    par(new=TRUE)
    plot(x, y2, col="blue", pch=2,
         xaxt="n", yaxt="n",
         ylab="", xlab="",
         xlim=c(min(x)-10, max(x)+10))
    axis(4)
    mtext("log10(Gamma)",side=4,line=3)
    legend("topleft",col=c("red","blue"),lty=1, legend=c("Cost","Gamma"), pch=1,
           cex=0.7)
    }
  dev.off()
  }

## function to generate a graph that compares all of the tuning parameters
##  attempted for a kernel and their corresponding performances
## the graph is a collection of plots for each kernel method that was considered
plot_all_tuned_kernels <- function(all_tuned_kernels) {
  all_tuned_kernels <- na.omit(all_tuned_kernels)
  g<- ggplot(all_tuned_kernels,
             aes(x=log10(ParamValue), y=Error, colour=Parameter)) +
    geom_errorbar(aes(ymin=Error-se, ymax=Error+se), width=.1) +
    geom_line() +
    geom_point() + facet_grid(.~kernel) +
    ggtitle("Tuning Parameters For Each Kernel Method")
  filename <- generate_filename(dir="nonlinear_output","all_kernels",
                                 "tuning_param_perf",".png")
  ggsave(filename, g, height=5,width=9)
  print(g, height=5,width=9)
  }

## function to generate both model residual plots and predictor regression plots
## this is intended to be called after the parameters for a specific kernel type
##    have been tuned
generate_model_pred_plots <- function(best_tuned_params, best_tuned_kernel,
                                      svm.model, test_set, svm.pred) {

  ## plot model residuals of train set
  plot_model_residuals(best_tuned_kernel, svm.model)

  ## plot prediction outcomes and true values
  plot_prediction_vs_true_r2(best_tuned_params, best_tuned_kernel, svm.pred,
                             test_set)
  }
```

Running the Models and Predicting Outcomes

Given a training set, a kernel, and tuned parameters, the `run_model()` function will generate an SVM model. An optional parameter `cross` can be used to indicate how many cross-validation models should be run, allowing the MSE of the fitted model to be extracted.

Given a trained model and a test set, the `predict_model()` function will predict outcomes for the unseen input values.

```
## run model
run_model <- function(train_set, kernel, tuned_cost, tuned_gamma=0, cross=0) {
  svm.model <- svm(survival_days ~ ., data = train_set,
                   kernel = best_tuned_kernel, cost = tuned_cost,
                   gamma = tuned_gamma, cross = cross)
  }

## run prediction
predict_model<- function(svm.model, test_set) {
  svm.pred <- predict(svm.model, test_set[,1:(ncol(test_set)-1)])
  }
```

Accessory/Utility Functions

```
generate_filename <- function(dir="", kernel,  desc_name, extension) {
  filename = ""
```

```r
  if(dir != "") {
    filename = paste(dir,"/", sep="")
    }
  filename = paste(filename, max_genes, "genes_", kernel,"_", desc_name,
                  extension, sep="")
  }

## function to set working directory
set_workdir <- function (my_workdir = ".") {

  ## set working directory
  if(!file.exists(my_workdir)) {
    stop(paste("Working directory is not valid or does not exist:\n'",
               my_workdir, "'", sep=""))
    }
  setwd(my_workdir)

  }
```

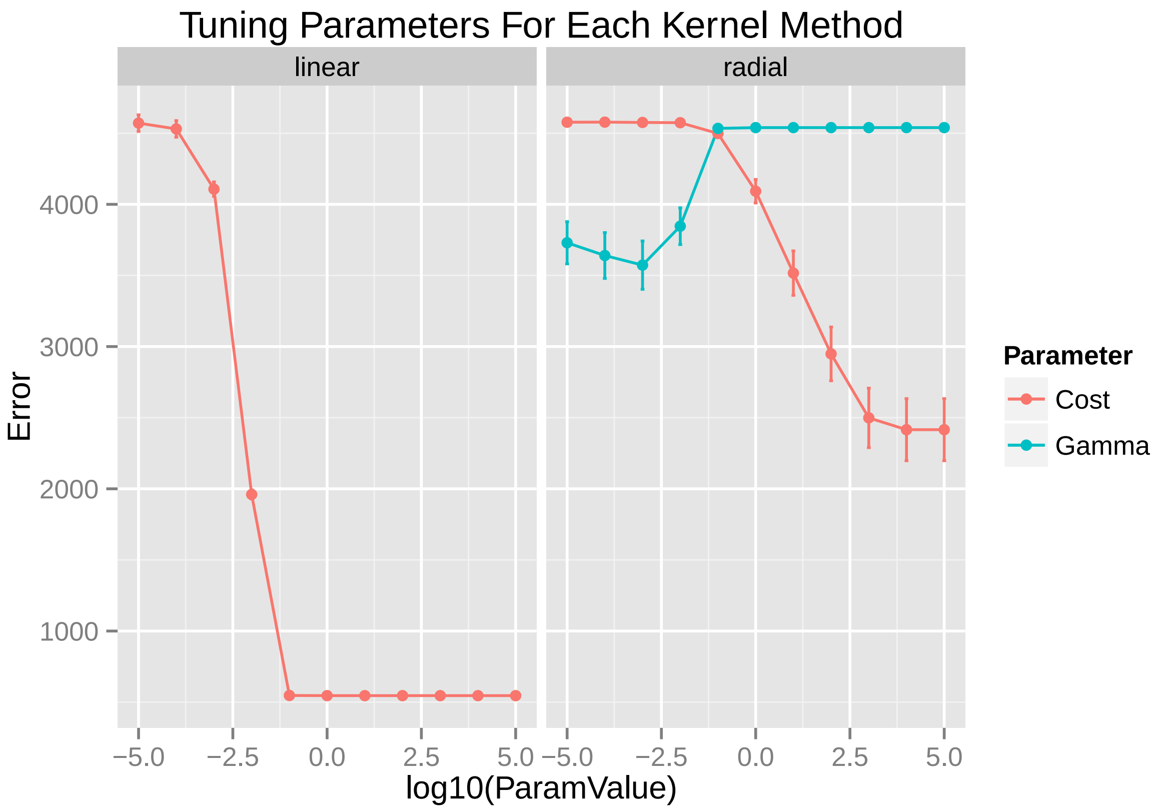
Model Generation

Data Preparation

```r
#####################
# MAIN FUNCTIONS
#####################
set_workdir()
max_genes = 50
table1 <- read_data_from_file(max_genes=max_genes)

## table format is genes (rows) x patients (columns)
## initially, no survival data is available
num_patients <- ncol(table1)
num_genes <- nrow(table1)

## designate Patient Ids ("Patient1", "Patient2", etc)
colnames(table1) <- lapply(seq(1,num_patients),
                           function(i) paste("Patient", i, sep=""))

## reformat the table to contain patients (rows) x genes (columns)
matrix1 <- t(as.matrix(table1))
table1 <- as.data.frame(matrix1)

## generate and add survival_days column to end of table
table2 <- generate_survival_info(table1)

## split data into a train and test set
index <- 1:nrow(table2)
test_index <- sample(index, trunc(length(index)/3))
test_set <- table2[test_index,]
train_set <- table2[-test_index,]
```

Model Tuning and Selection

```
## tune model
max_gamma <- 5
min_gamma <- -5
max_cost <- 5
min_cost <- -5
min_degree <- 2
max_degree <- 5
all_tune_output <- find_best_params(train_set,
                                    max_gamma, min_gamma,
                                    max_cost, min_cost,
                                    max_degree, min_degree)
```



```
best_tuned_kernel <- all_tune_output[[1]]
best_tuned_params <- all_tune_output[[2]]
compare_kernels_plot <- all_tune_output[[3]]
```

The selected model for this dataset (using non-linear transformations to generate survival data): `{r} best_tuned_kernel` kernel with parameters `{r} best_tuned_params`. As evidenced by the plot above, different values for the cost parameter can dramatically impact the model's performance, especially in the linear model. The performances by the radial parameters are much more consistent, but there are still noticable drops in error for certain cost and parameter values. These graphs highlight the advantages of tuning parameters over large ranges and provide visual clues as to how much complexity a model can handle before it overfits the data.

```
## use the tuned parameters to generate a model based on the training data
## and predict outcomes based on the test set
tuned_cost <- best_tuned_params["cost"][[1]]
tuned_gamma <- 0
if(best_tuned_kernel != "linear") {

  tuned_gamma <- best_tuned_params["gamma"][[1]]

  ## run model
  svm.model <- run_model(train_set, kernel, tuned_cost, tuned_gamma)
  } else {

    ## run model
    svm.model <- run_model(train_set, kernel, tuned_cost)
    }

## run prediction
svm.pred <- predict_model(svm.model, test_set)

## compare prediction and real values
comparison <- cbind(svm.pred, test_set$survival_days)
```

Model Assessment

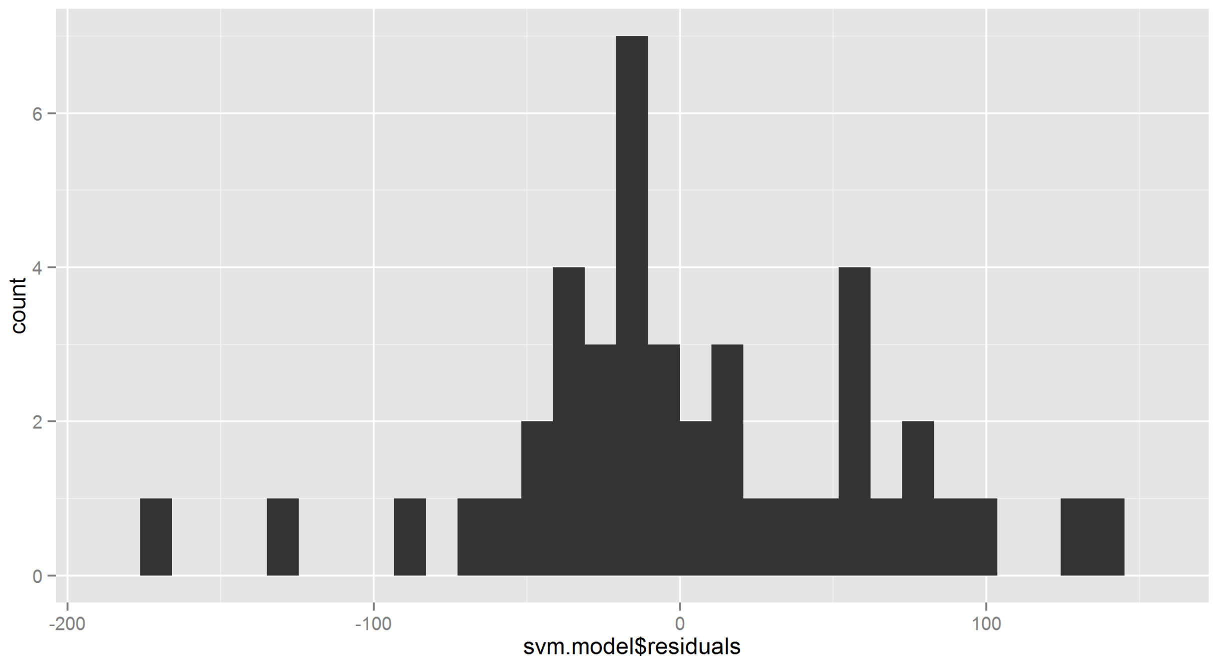
Two easy-to-interpret metrics of how well an SVM aligns with the true data are residual plots and linear regressions. The residual plots show the distribution of differences between the true survival datapoints in the training set and the SVM's fitted values and in doing so, illustrates what the model has learned from the given data. As a complement to those graphs, linear regression plots of the true survival datapoints in the test set compared to the SVM's predicted values showcases how well the model applies what it has learned to unseen data.

```
## plot model residuals of train set
plot_model_residuals(best_tuned_kernel, svm.model)
```

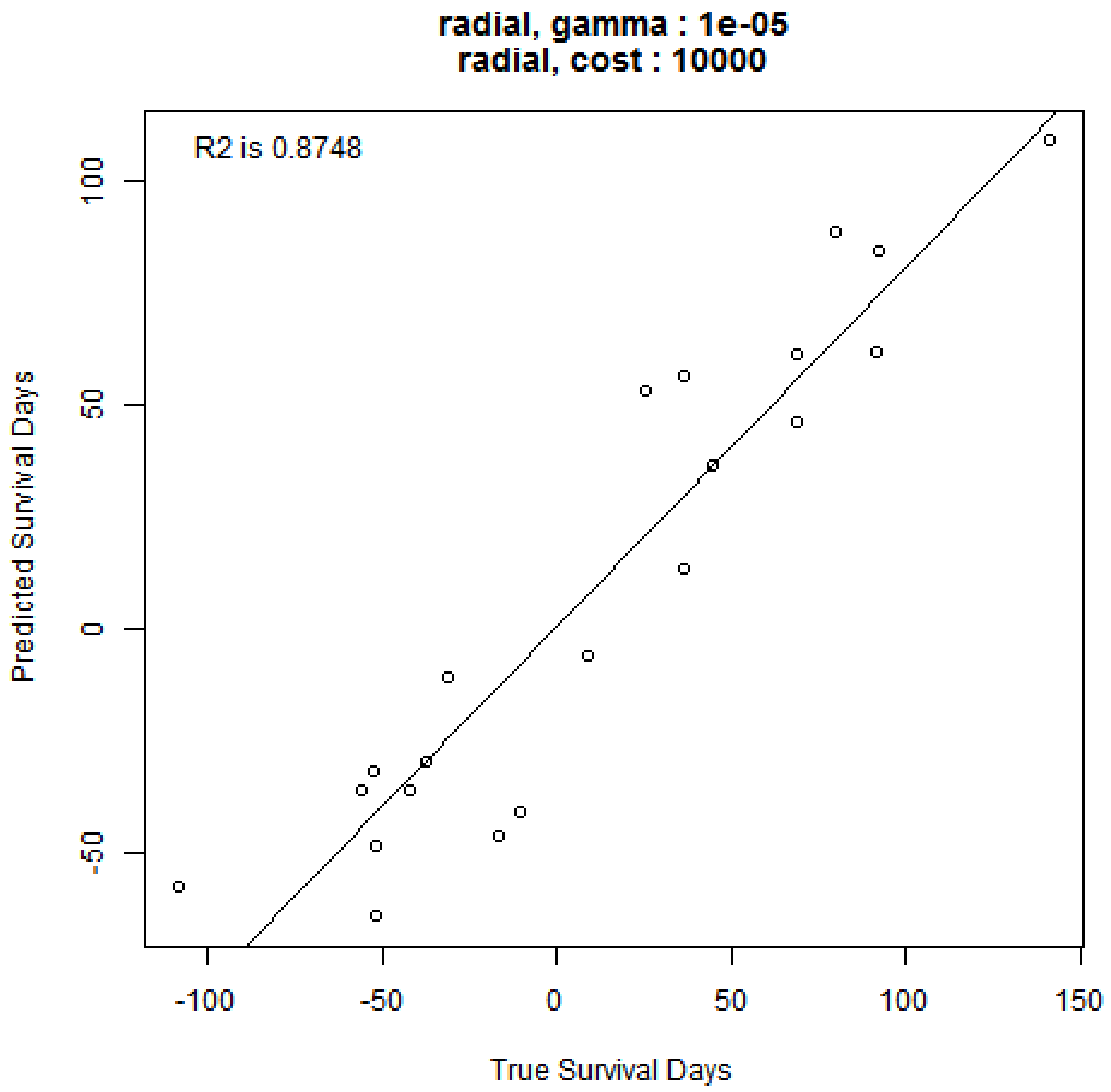
```
## stat_bin: binwidth defaulted to range/30. Use 'binwidth = x' to adjust this.
```

```
## plot prediction outcomes and true values
plot_prediction_vs_true_r2(best_tuned_params, best_tuned_kernel,
                           svm.pred, test_set)
```

```
## pdf
##   2
```



For this dataset (which used linear transformations to generate the “true” survival outcomes), a radial kernel was optimized using parameters cost=1e4 and gamma=1e-5. In the histogram of residuals, one can see that the center of the distribution of the differences between true and fitted values is between -25 and 15. Considering that the true values ranged from -150 to 180 (data not shown), these residuals are within acceptable bounds.



There is a high degree of correlation (R2 = 0.87) between the test set's true values and the SVM's predicted values.

Model Testing- Resampling the Dataset

Since our selection of the data (50 genes) was such a small percentage of the total dataset, running the optimized model multiple times provided additional information about how well the model could learn and predict survival outcomes. The following code runs 10 iterations of the model on new test and training subsets and records the correlations between true and predicted values.

```
## run model X times
iterations_to_repeat_model <- 10
multiple_model_table <- as.data.frame(cbind(row.names(table2),
                                           table2$survival_days))
```

```r
colnames(multiple_model_table)[1:2] <- c("PatientID", "Survival Time")
multiple_model_corr_table <- as.data.frame(c("tot.MSE","R2"))
colnames(multiple_model_corr_table)[1] = "PredCorr"

for(i in 1:iterations_to_repeat_model) {

  svm.model <- run_model(train_set, kernel, tuned_cost, tuned_gamma, cross=10)
  svm.pred <- predict_model(svm.model, test_set)

  if(i == 1) {
    #generate_model_pred_plots(best_tuned_params, best_tuned_kernel, svm.model,
    #  test_set, svm.pred)
    }

  new_col_name <- paste(best_tuned_kernel,"Run", i)
  svm.table <- as.data.frame(cbind(names(svm.pred),svm.pred))
  colnames(svm.table) <- c("PatientID", "svm.pred")
  multiple_model_table <- merge(multiple_model_table, svm.table, by="PatientID",
                               all=T, sort=T)
  colnames(multiple_model_table)[i+2] <- paste(best_tuned_kernel,"Run", i)

  fit2 <- lm(svm.pred ~ test_set$survival_days)
  model_corr_values <- c(svm.model$tot.MSE,
                         format(summary(fit2)$adj.r.squared, digits=4))
  multiple_model_corr_table <- cbind(multiple_model_corr_table,
                                     model_corr_values)
  colnames(multiple_model_corr_table)[i+1] <- paste(best_tuned_kernel,"Run", i)

  test_index <- sample(index, trunc(length(index)/3))
  test_set <- table2[test_index,]
  train_set <- table2[-test_index,]
  }
```

Tables/Reports

```r
filename <- generate_filename(dir="nonlinear_output",best_tuned_kernel,
                              "model_predictions",".csv")
write.csv(multiple_model_table, file=filename, quote=F)

filename <- generate_filename(dir="nonlinear_output",best_tuned_kernel,
                              "model_MSE",".csv")
write.csv(multiple_model_corr_table, file=filename, quote=F)
```

Discussion

SVMs are useful tools for translating feature vectors of training sets into succinct functions that can reliably predict values for unseen data. Choosing an appropriate kernel method and optimizing its parameters are crucial steps in generating a successful model.

In future reports, this section will discuss the overall conclusions based on the statistical analyses as well as touch on the biological implications of the findings. Each week, there will also be a cumulative comparison of the new model with respect to the past algorithms.