Optimization Methods for Tuning Predictive Models

CRUG 2016

Max Kuhn

Pfizer R&D

Over–Fitting and Model Tuning

APM Ch. 4

Over–Fitting

Over–fitting occurs when a model inappropriately picks up on trends in the training set that do not generalize to new samples.

When this occurs, assessments of the model based on the training set can show good performance that does not reproduce in future samples.

Some models have specific "knobs" to control over-fitting

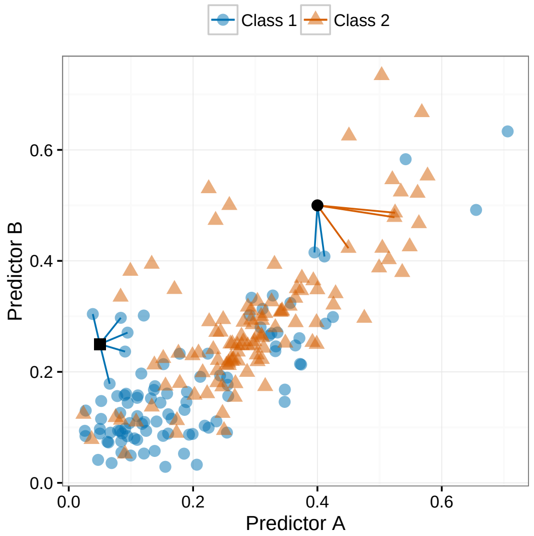
- neighborhood size in nearest neighbor models is an example
- the number if splits in a tree model

Often, poor choices for these parameters can result in over-fitting

For example, the next slide shows a data set with two predictors. We want to be able to produce a line (i.e. decision boundary) that differentiates two classes of data.

Two new points are to be predicted. A 5–nearest neighbor model is illustrated.

K–Nearest Neighbors Classification



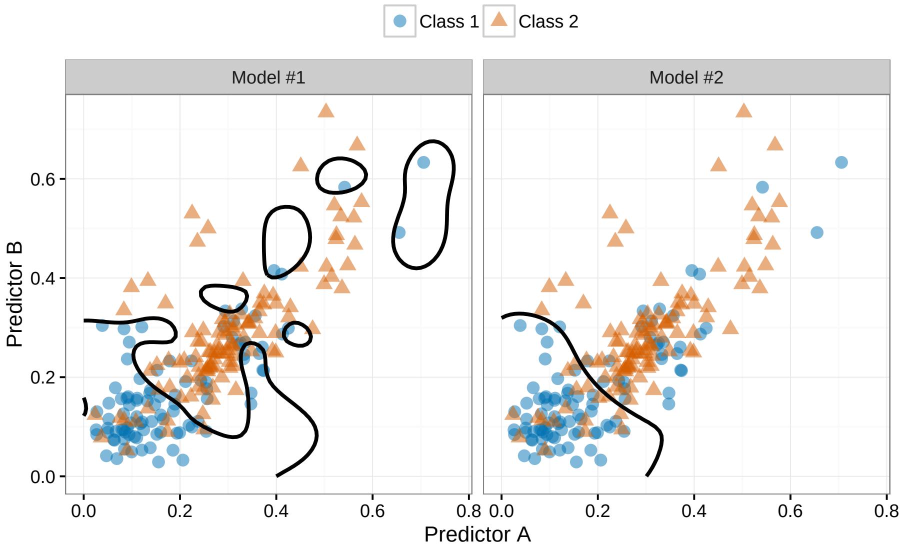
Over–Fitting

On the next slide, two classification boundaries are shown for the a different model type not yet discussed.

The difference in the two panels is solely due to different choices in tuning parameters.

One over–fits the training data.

Two Model Fits



Characterizing Over–Fitting Using the Training Set

One obvious way to detect over–fitting is to use a test set. However, repeated "looks" at the test set can also lead to over–fitting

Resampling the training samples allows us to know when we are making poor choices for the values of these parameters (the test set is not used).

Resampling methods try to "inject variation" in the system to approximate the model's performance on future samples.

We'll walk through several types of resampling methods for training set samples.

See the two blog posts "Comparing Different Species of Cross-Validation" at `http://bit.ly/1yE0Ss5` and `http://bit.ly/1zfoFj2`

K–Fold Cross–Validation

Here, we randomly split the data into K distinct blocks of roughly equal size.

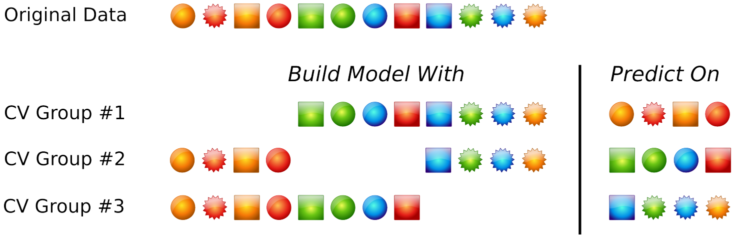
1. We leave out the first block of data and fit a model.
2. This model is used to predict the held-out block
3. We continue this process until we've predicted all K held–out blocks

The final performance is based on the hold-out predictions

K is usually taken to be 5 or 10 and leave one out cross–validation has each sample as a block

Repeated K–fold CV creates multiple versions of the folds and aggregates the results (I prefer this method)

K–Fold Cross–Validation



The Big Picture – Grid Search

We think that resampling will give us honest estimates of future performance, but there is still the issue of which model to select.

One algorithm to select models:

```
Define sets of model parameter values to evaluate;
for each parameter set do
    for each resampling iteration do
        Hold–out specific samples ;
        Fit the model on the remainder;
        Predict the hold–out samples;
    end
    Calculate the average performance across hold–out predictions
end
Determine the optimal parameter set;
```

Regular Grid Search vs Random Search

In the previous workflow, a candidate set of tuning parameters is define *a priori*.

This is usually a regular grid in (multidimensional) tuning parameter space.

However, as the parameter space dimensionality increases, a regular grid *usually* becomes expensive, especially in over a wide range.

Random search is a simple an effective way to do an efficient broad search. Random uniform parameter combinations are simulated across a wide range.

Grid Search optimizations

While random search is a great idea when you have limited knowledge of the parameter space or there are a number of large parameters, it does have drawbacks.

There are optimizations that can be used with a regular grid that are eliminated when using random parameter values.

For example, the typical tuning parameters for a stochastic gradient boosting machine include the number of boosting iterations, the model complexity, learning rate, and other values.

However, with the other parameters fixed, a model can be fit with the maximum number of iterations and all sub–models with fewer iterations can be *derived*. This can lead to speedups that are in the 10's of folds.

Other models with this property: PLS, PCR, glmnet, LARS, MARS, rotation forest, and others.

Example Data

Sacramento Housing Prices

The data used to illustrate the models are sale prices of homes in Sacramento CA.

The original data were obtained from the website for the SpatialKey software. From their website:

> *The Sacramento real estate transactions file is a list of 985 real estate transactions in the Sacramento area reported over a five-day period, as reported by the Sacramento Bee.*

Google was used to fill in missing/incorrect data.

CA House Data Set

```
> library(caret)
> data(Sacramento)
> str(Sacramento)
```

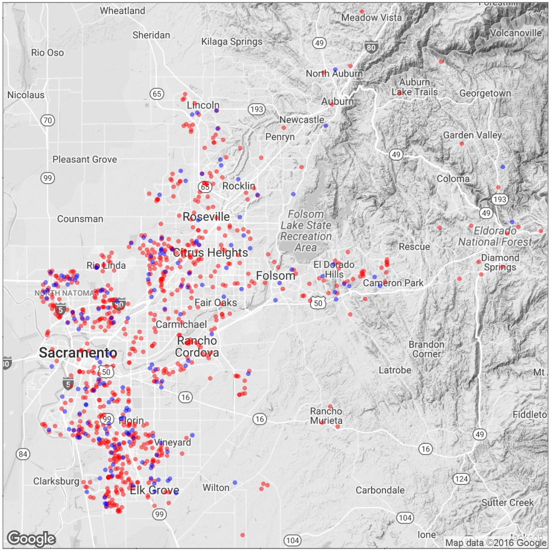
```
'data.frame': 932 obs. of  9 variables:
 $ city     : Factor w/ 37 levels "ANTELOPE","AUBURN",..: 34 34 34 34 34 34 34 34 2
 $ zip      : Factor w/ 68 levels "z95603","z95608",..: 64 52 44 44 53 65 66 49 24
 $ beds     : int  2 3 2 2 2 3 3 3 2 3 ...
 $ baths    : num  1 1 1 1 1 1 2 1 2 2 ...
 $ sqft     : int  836 1167 796 852 797 1122 1104 1177 941 1146 ...
 $ type     : Factor w/ 3 levels "Condo","Multi_Family",..: 3 3 3 3 3 1 3 3 1 3 ...
 $ price    : int  59222 68212 68880 69307 81900 89921 90895 91002 94905 98937 ...
 $ latitude : num  38.6 38.5 38.6 38.6 38.5 ...
 $ longitude: num  -121 -121 -121 -121 -121 ...
```

House Price Data

For these data, let's take a stratified random sample of houses for training.

```
> set.seed(955)
> in_train <- createDataPartition(log10(Sacramento$price), p = .8, list = FALSE)
> head(in_train)

     Resample1
[1,]         1
[2,]         2
[3,]         3
[4,]         4
[5,]         5
[6,]         6

> training <- Sacramento[ in_train,]
> testing  <- Sacramento[-in_train,]
```

Training in Blue, Testing in Red



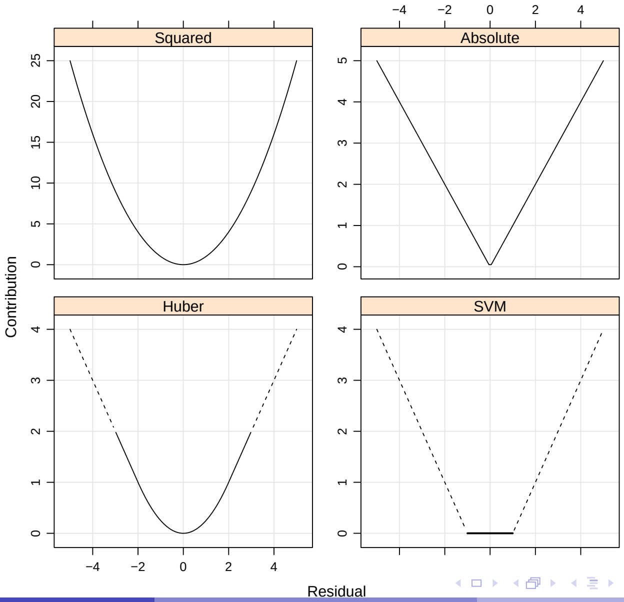
Support Vector Machines (SVM)

This is a class of powerful and very flexible models originally designed for classification using a new type of objective function called the *margin*.

SVMs for regression have similar properties as robust regression models, specifically M–estimation.

In this objective function, training set points with the *smallest* errors do not influence the training set points.

SVM Objective Function



SVM Prediction Function

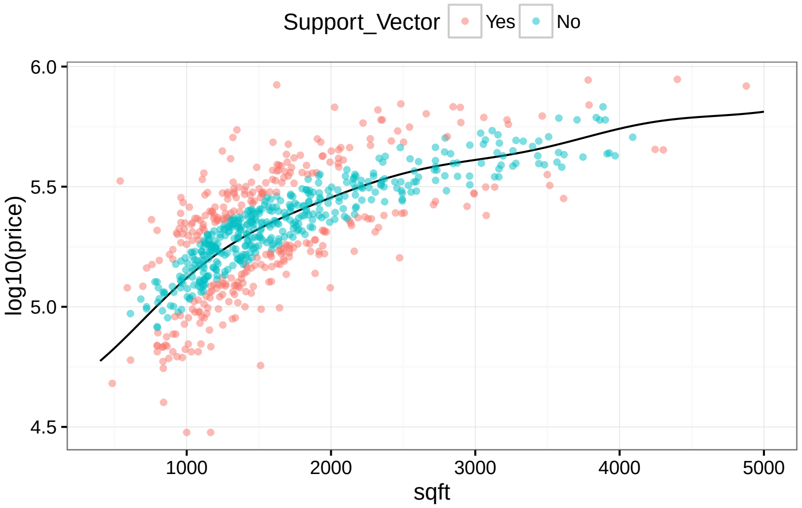
The SVM model estimates n parameters $(\alpha_1 \ldots \alpha_n)$ for the model. Regularization is used to avoid saturated models (more on that in a minute).

For a new point u, the model predicts:

$$f(u) = \beta_0 + \sum_{i=1}^{n} \alpha_i y_i x_i' u$$

Data points that are support vectors have $\alpha_i \neq 0$, so the prediction equation is only affected by the support vectors.

SVMs and the Margin



The Kernel Trick

You may have noticed that the prediction function was a function of an inner product between two samples vectors ($x_i'u$). It turns out that this opens up some new possibilities.

Nonlinear regression models can be computed using the "kernel trick".

The predictor space can be expanded by adding nonlinear functions in x. These functions, which must satisfy specific mathematical criteria, include common functions:

$$\text{Polynomial} : K(x,u) = (1+x'u)^p$$

$$\text{Radial basis function} : K(x,u) = exp\left[\frac{-\sigma}{2}(x-u)^2\right]$$

We don't need to store the extra dimensions; these functions can be computed quickly.

SVM Regularization

As previously discussed, SVMs also include a regularization parameter that controls how much the regression line can adapt to the data smaller values result in more linear (i.e. flat) surfaces

This parameter is generally referred to as “Cost”

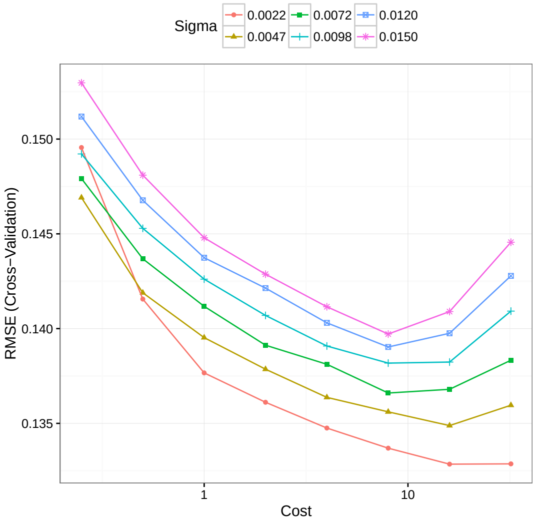
If the cost parameter is large, there is a significant penalty for having samples within the margin $\Rightarrow$ the regression line becomes very flexible.

Tuning the cost parameter, as well as any kernel parameters, becomes very important as these models have the ability to greatly over–fit the training data.

Grid Search for SVM

```
> set.seed(3313)
> index <- createFolds(training$price, returnTrain = TRUE, list = TRUE)
> ctrl <- trainControl(method = "cv", index = index)
>
> set.seed(30218)
> grid_search <- train(log10(price) ~ ., data = training,
+                      method = "svmRadialSigma",
+                      ## Will create 48 parameter combinations
+                      tuneLength = 8,
+                      metric = "RMSE",
+                      preProc = c("center", "scale", "zv"),
+                      trControl = ctrl)
> getTrainPerf(grid_search)

  TrainRMSE TrainRsquared         method
1 0.1328485     0.6637799 svmRadialSigma
```

Grid Search for SVM



Nonlinear Optimization

We can treat the process of finding the best tuning parameter values as any other nonlinear optimization problem and use all of the machinery to bear such as

- Simulated annealing (via `optim`), Nelder–Mead (`optim`), gradient–based methods
- Genetic algorithms (via `GA::ga`), particle swarm optimization (via `pso::psoptim`), etc.

The trick is that we still need an objective function estimate that generalizes well (e.g. resampled RMSE)

We will use the same cross–validation folds when we measure performance across the candidate models.

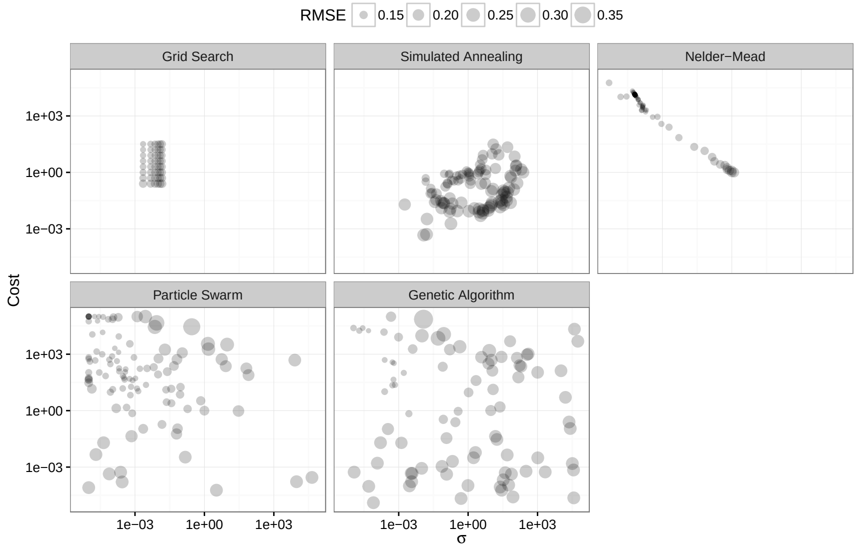
Nonlinear Optimization

There is an important trade–off between computational time and the precision of estimates

Many nonlinear optimization methods can require a lot of iterations to converge.

Limiting ourselves to 100 model evaluations, here is what the results look like for several approaches.

Nonlinear Optimization Results



Nonlinear Optimization Results

A few methods (NM, GA, PS) converged to similar values around $C = 10^4$ and $\sigma = 10^{-4}$ while SA just seemed ... confused.

For Nelder–Mead, most of the time was spent converging around the eventual estimate. It also converged quickly but is generally bad. It is typical to restart NM in different locations to combat this issue

The two batch–oriented methods (GA, PS) cast a wide net and do well but would need more iterations to refine the estimate. Their batch sizes were 50 (GA) and 12 (PS) but could be changed.

None of these methods extensively *learn* from the entire set of previous iterations. Bayesian optimization via Gaussian processes have the ability to do this.

Bayesian Optimization via Gaussian Processes

Let's denote

- the tuning parameter vector as $\boldsymbol{z}_i \; i = 1 \ldots M$
- the objective function for tuning parameter i estimated from resample $j = 1 \ldots B$ as Q_{ij}.

The model is

$$Q_{ij} = \boldsymbol{z}_i' \boldsymbol{\beta} + \epsilon_{ij}$$

A Gaussian process assumes that the parameters follow a multivariate Gaussian prior: $Pr(\boldsymbol{\beta}) \sim N(\mathbf{0}, \Sigma)$ and $\epsilon_{ij} \sim N(0, \sigma_Q^2)$.

The likelihood function for the parameters is $Pr(Q|\boldsymbol{z}, \boldsymbol{\beta}) \sim N(\boldsymbol{z}_i' \boldsymbol{\beta}, \sigma_Q^2 I)$ and the resulting posterior is Gaussian distribution with a fairly complex covariance function.

Gaussian Processes and Kernels

In the same manner as SVMs, nonlinear GP's can be created using the kernel trick with the model.

$$Q_{ij} = \phi(\boldsymbol{z}_i)'\boldsymbol{\beta} + \epsilon_{ij}$$

where $\phi(\cdot)$ is the kernel basis function expansion (e.g. $\phi(z) = (1, z, z^2)$)

This leads to nonlinear mean and variance functions but with closed form solutions.

For this application, profile likelihood is used to optimize the kernel parameter(s).

Bayesian Optimization via Gaussian Processes

Given an initial set of tuning parameter combinations, a GP can be created to relate the tuning parameters to the performance metric.

The estimated GP can be used to predict which part of the tuning parameter space is best to be explore and a new value of z is determined and Q is estimated using resampling.

This new point is used to update the GP and the process continues.

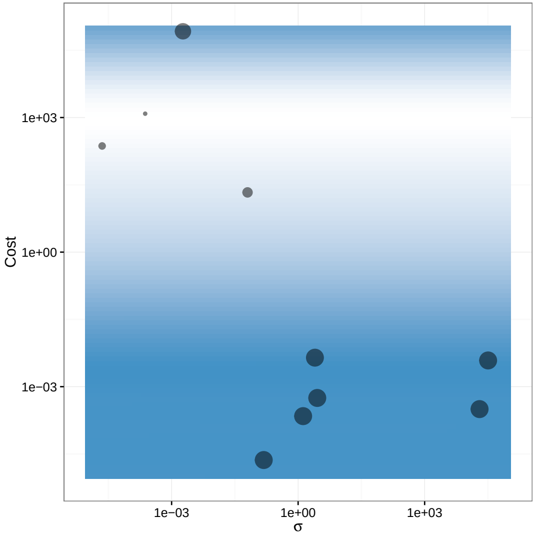
Gaussian Processes Acquisition Functions

The GP can produce predictions for the average values of Q as well as the variation in Q at z.

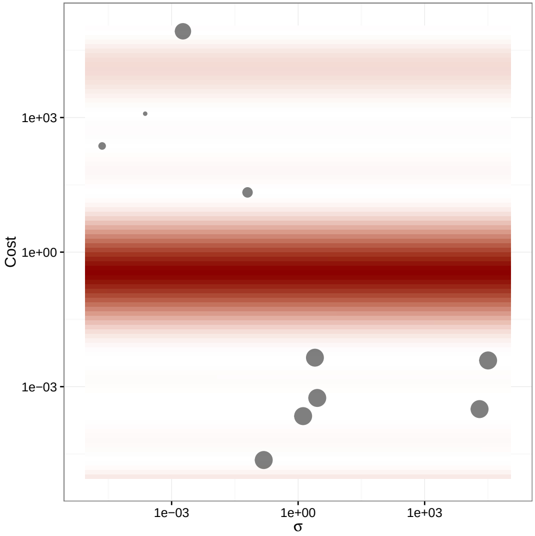
There are several methods of quantifying which points should be sampled in the next iteration. Probability of improvement and the expected improvement are two methods (see references in last slide).

We will score new parameter values ousing the predicted *lower confidence bound* $\mu_Q - \kappa\sigma_Q$ and use $\kappa = 1$

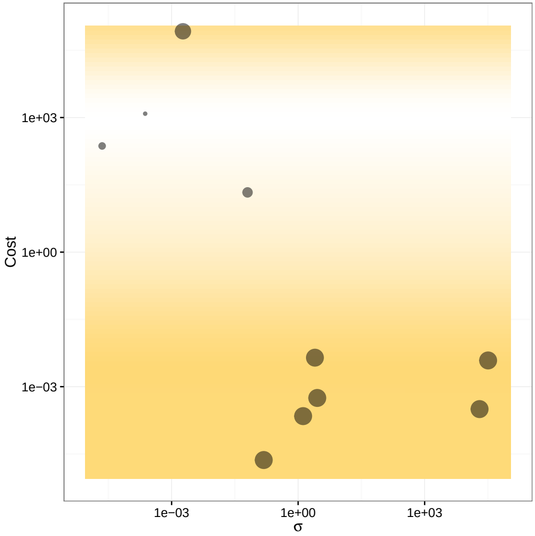
Initial GP Model - Mean Prediction



Initial GP Model - Variance Prediction



Initial GP Model - Acquisition Function



R Code – Objective Function

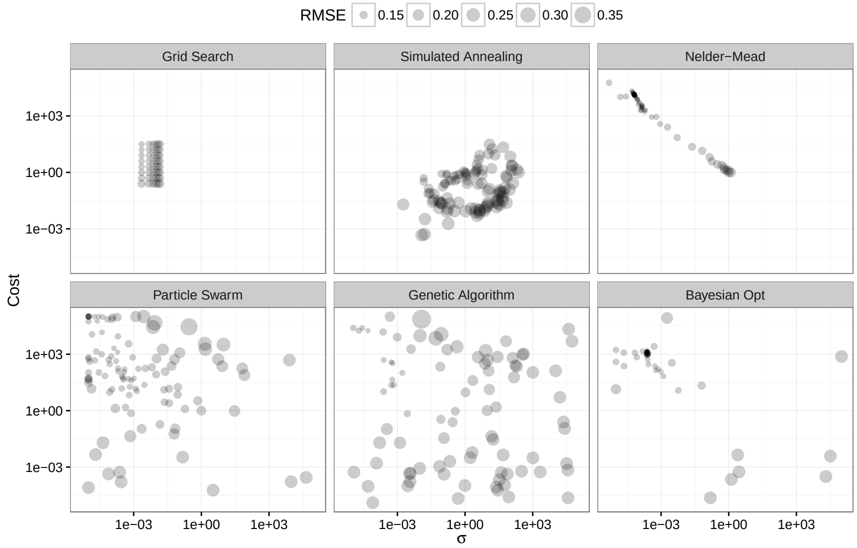
```
> ## Use this function to optimize the model. The two parameters are
> ## evaluated on the log scale given their range and scope.
> svm_fit_bayes <- function(logC, logSigma) {
+   ## Use the same model code but for a single (C, sigma) pair.
+   mod <- train(log10(price) ~ ., data = training,
+                method = "svmRadial",
+                preProc = c("center", "scale", "zv"),
+                metric = "RMSE",
+                trControl = ctrl,
+                tuneGrid = data.frame(C = 10^(logC), sigma = 10^(logSigma)))

+   ## The function wants to _maximize_ the outcome so we return
+   ## the negative of the resampled RMSE value. `Pred` can be used
+   ## to return predicted values but we'll avoid that and use NULL
+   list(Score = -getTrainPerf(mod)[, "TrainRMSE"], Pred = 0)
+ }
```

R Code – Run Optimization

```
> library(rBayesianOptimization)
>
> bounds <- list(logC = c(-2,  5), logSigma = c(-7, -2))
>
> set.seed(8606)
> bo_search <- BayesianOptimization(svm_fit_bayes,
+                                   bounds = bounds,
+                                   init_points = 10,
+                                   n_iter = 100,
+                                   acq = "ucb",
+                                   kappa = 1,
+                                   eps = 0.0)
```

The package also has an argument called `init_grid_dt` that can be used to input previously generated tuning parameter results to start the GP fitting process (e.g. from grid search).

Expanded Nonlinear Optimization Results



Resampling Results

	$log_{10}\, Cost$	$log_{10}\, \sigma$	RMSE
SA	-0.303	-1.83	0.148
NM	4.138	-4.19	0.130
PSO	4.763	-4.62	0.131
GA	4.249	-4.30	0.130
BO	3.083	-3.63	-0.130

Nelder-Mead Test Set Results

```
> set.seed(30218)
> nm_mod <- train(log10(price) ~ ., data = training,
+                 method = "svmRadialSigma",
+                 tuneGrid = data.frame(C = 10^nm_res$par[1],
+                                       sigma = 10^nm_res$par[2]),
+                 metric = "RMSE",
+                 trControl = ctrl,
+                 preProc = c("center", "scale", "zv"))
> postResample(predict(nm_mod, testing), log10(testing$price))

     RMSE  Rsquared
0.1202089 0.7209441
```

Bayesian Optimization Test Set Results

```
> set.seed(30218)
> bo_mod <- train(log10(price) ~ ., data = training,
+                 method = "svmRadialSigma",
+                 tuneGrid = data.frame(C = 10^bo_search$Best_Par[1],
+                                       sigma = 10^bo_search$Best_Par[2]),
+                 metric = "RMSE",
+                 trControl = ctrl,
+                 preProc = c("center", "scale", "zv"))
> postResample(predict(bo_mod, testing), log10(testing$price))

     RMSE  Rsquared
0.1201116 0.7209088
```

References

- Brochu, E., Cora, V. M., & De Freitas, N. (2010). A tutorial on Bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning. *arXiv*.
- Kuhn, M., & Johnson, K. (2013). *Applied Predictive Modeling*. Springer Verlag.
- Rasmussen, C. E., & Williams, C. K. I. (2006). *Gaussian Processes for Machine Learning*. MIT Press.
- Smola, A., & Scholkopf, B. (2004). A tutorial on support vector regression. *Statistics and Computing*, 14(3), 199222.
- Smola, A., Vapnik, V. (1997). Support vector regression machines. *Advances in neural information processing systems*, 9, 155-161..
- Snoek, J., Larochelle, H., Adams, R. P. (2012). Practical Bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems* (pp. 2951-2959).