# Predict the Price of Used Corolla - SVM

Langtao Chen

Initial: Feb 20, 2017 Update: Apr 13, 2021

## Contents



In this example, we'll use SVM to predict the price of used Corolla.

# 1. Data

Import the data from the csv file.

```
# Clean the environment
rm(list = ls())

# Read data file
df <- read.csv("ToyotaCorolla.csv", stringsAsFactors = TRUE)
```

```
# Show the head of the dataset
head(df)
```

```
##   Price Age    KM FuelType HP MetColor Automatic   CC Doors Weight
## 1 13500  23 46986   Diesel 90        1         0 2000     3   1165
## 2 13750  23 72937   Diesel 90        1         0 2000     3   1165
## 3 13950  24 41711   Diesel 90        1         0 2000     3   1165
```

```
## 4 14950  26 48000   Diesel 90        0          0 2000     3    1165
## 5 13750  30 38500   Diesel 90        0          0 2000     3    1170
## 6 12950  32 61000   Diesel 90        0          0 2000     3    1170
```

```r
# Show the structure of the dataset
str(df)
```

```
## 'data.frame':    1436 obs. of  10 variables:
##  $ Price    : int  13500 13750 13950 14950 13750 12950 16900 18600 21500 12950 ...
##  $ Age      : int  23 23 24 26 30 32 27 30 27 23 ...
##  $ KM       : int  46986 72937 41711 48000 38500 61000 94612 75889 19700 71138 ...
##  $ FuelType : Factor w/ 3 levels "CNG","Diesel",..: 2 2 2 2 2 2 2 2 3 2 ...
##  $ HP       : int  90 90 90 90 90 90 90 90 192 69 ...
##  $ MetColor : int  1 1 1 0 0 0 1 1 0 0 ...
##  $ Automatic: int  0 0 0 0 0 0 0 0 0 0 ...
##  $ CC       : int  2000 2000 2000 2000 2000 2000 2000 2000 1800 1900 ...
##  $ Doors    : int  3 3 3 3 3 3 3 3 3 3 ...
##  $ Weight   : int  1165 1165 1165 1165 1170 1170 1245 1245 1185 1105 ...
```

```r
# Summary statistics
summary(df)
```

```
##      Price            Age              KM            FuelType          HP       
##  Min.   : 4350   Min.   : 1.00   Min.   :     1   CNG   :  17   Min.   : 69.0  
##  1st Qu.: 8450   1st Qu.:44.00   1st Qu.: 43000   Diesel: 155   1st Qu.: 90.0  
##  Median : 9900   Median :61.00   Median : 63390   Petrol:1264   Median :110.0  
##  Mean   :10731   Mean   :55.95   Mean   : 68533                 Mean   :101.5  
##  3rd Qu.:11950   3rd Qu.:70.00   3rd Qu.: 87021                 3rd Qu.:110.0  
##  Max.   :32500   Max.   :80.00   Max.   :243000                 Max.   :192.0  
##     MetColor        Automatic             CC            Doors      
##  Min.   :0.0000   Min.   :0.00000   Min.   :1300   Min.   :2.000  
##  1st Qu.:0.0000   1st Qu.:0.00000   1st Qu.:1400   1st Qu.:3.000  
##  Median :1.0000   Median :0.00000   Median :1600   Median :4.000  
##  Mean   :0.6748   Mean   :0.05571   Mean   :1567   Mean   :4.033  
##  3rd Qu.:1.0000   3rd Qu.:0.00000   3rd Qu.:1600   3rd Qu.:5.000  
##  Max.   :1.0000   Max.   :1.00000   Max.   :2000   Max.   :5.000  
##      Weight    
##  Min.   :1000  
##  1st Qu.:1040  
##  Median :1070  
##  Mean   :1072  
##  3rd Qu.:1085  
##  Max.   :1615  
```

From the summary statistics, we found that there is no missing value.

# 2. Data Partitioning

We use a single 80/20% split.

```r
library(caret)
```

```
## Loading required package: lattice

## Loading required package: ggplot2
```

```r
set.seed(1234)
trainIndex <- createDataPartition(df$Price, p = .8, list = FALSE)

train_data <- df[ trainIndex,]
test_data  <- df[-trainIndex,]

nrow(train_data)
```

```
## [1] 1150
```

```r
nrow(test_data)
```

```
## [1] 286
```

# 3. Data Normalization

It's usually recommended to normalize data before apply support vector machines. Here we use the preProcess() method in caret package to normalize variables.

Note: In the method parameter, we set "scale" to transform the standard deviation as 1, and set "center" to center the variable such that the mean will be zero. Use the "scale" and "center" methods together, we can normalize the variables such that mean = 0, sd = 1.

```r
# Load library
library(caret)

# calculate the pre-process parameters from the training dataset
preprocessParams <- preProcess(train_data, method = c("scale","center"))

# summarize transform parameters
print(preprocessParams)
```

```
## Created from 1150 samples and 10 variables
##
## Pre-processing:
##   - centered (9)
##   - ignored (1)
##   - scaled (9)
```

```r
# transform the training dataset using the parameters
train_scaled <- predict(preprocessParams, train_data)

# summarize the transformed dataset
stargazer::stargazer(train_scaled, type = 'text')
```

```
##
## ============================================================
## Statistic   N    Mean  St. Dev.  Min   Pctl(25) Pctl(75)  Max
## ------------------------------------------------------------
## Price     1,150 -0.000  1.000   -1.723  -0.627   0.332   5.963
## Age       1,150 0.000   1.000   -2.977  -0.651   0.756   1.297
## KM        1,150 -0.000  1.000   -1.868  -0.679   0.492   4.565
## HP        1,150 -0.000  1.000   -2.175  -0.780   0.548   5.994
## MetColor  1,150 0.000   1.000   -1.451  -1.451   0.688   0.688
## Automatic 1,150 -0.000  1.000   -0.249  -0.249   -0.249  4.019
## CC        1,150 0.000   1.000   -1.419  -0.883   0.190   2.334
## Doors     1,150 -0.000  1.000   -2.162  -1.110   0.995   0.995
## Weight    1,150 -0.000  1.000   -1.345  -0.608   0.221   9.983
## ------------------------------------------------------------
```

It's important to use the same parameters to transform the test dataset.

```
# transform the test dataset using the parameters
test_scaled <- predict(preprocessParams, test_data)

# summarize the transformed dataset
stargazer::stargazer(test_scaled, type = 'text')
```

```
##
## ==========================================================
## Statistic  N   Mean  St. Dev.  Min   Pctl(25) Pctl(75)  Max
## ----------------------------------------------------------
## Price     286 -0.011  0.970   -1.750  -0.627   0.332   3.771
## Age       286 -0.021  1.032   -2.652  -0.705   0.797   1.297
## KM        286 0.125   1.165   -1.868  -0.700   0.702   4.843
## HP        286 -0.082  0.973   -2.175  -1.046   0.548   5.994
## MetColor  286 -0.037  1.015   -1.451  -1.451   0.688   0.688
## Automatic 286 -0.055  0.890   -0.249  -0.249   -0.249  4.019
## CC        286 0.060   1.018   -1.419  -0.883   0.190   2.334
## Doors     286 -0.113  1.009   -2.162  -1.110   0.995   0.995
## Weight    286 -0.051  0.837   -1.345  -0.608   0.221   4.549
## ----------------------------------------------------------
```

From the above summary statistics, we notice that the mean and sd of the variables in the test dataset are not necessarily 0 and 1 respectively. This is because the rescaling parameters are only obtained from the training dataset.

# 4. Support Vector Machine

## 4.1. Try a Linear Kernel

We can use the svm() method in the e1071 package to implement the SVM algorithm. Let's start with a linear kernel and parameter cost = 100.

```
library(e1071)
```

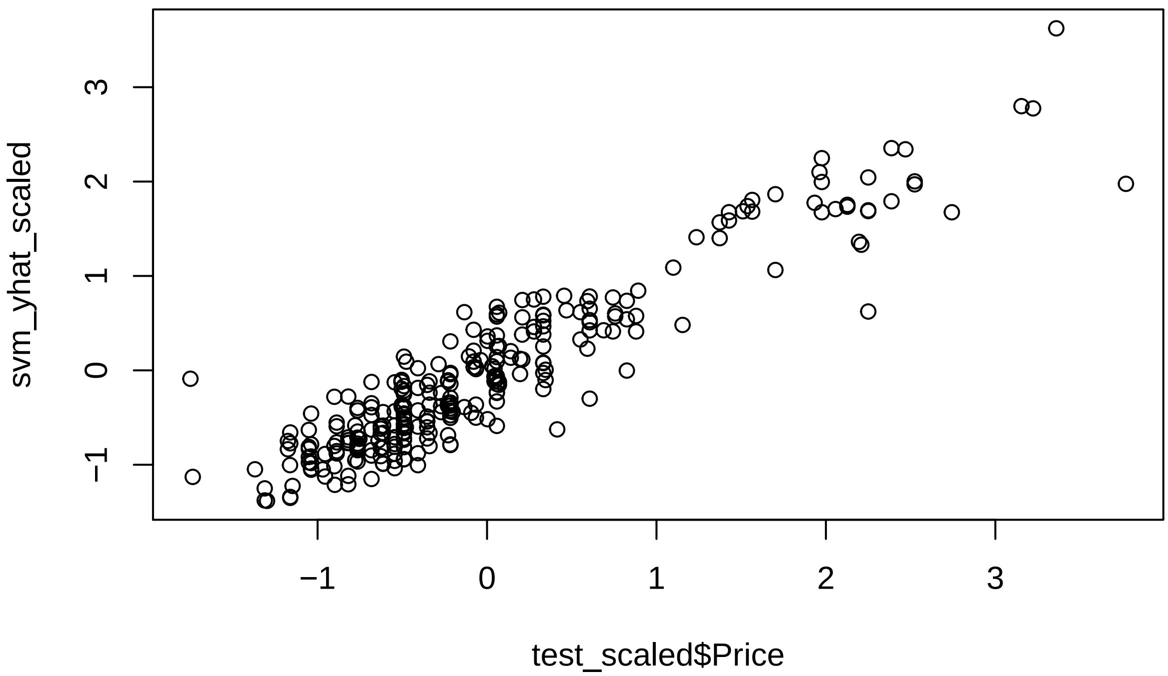
```
## Warning: package 'e1071' was built under R version 4.0.4
```

```r
svm_corolla <- svm(Price~., data = train_scaled,
                   kernel = 'linear', cost = 100, scale = TRUE)

summary(svm_corolla)
```

```
##
## Call:
## svm(formula = Price ~ ., data = train_scaled, kernel = "linear",
##     cost = 100, scale = TRUE)
##
##
## Parameters:
##    SVM-Type:  eps-regression
##  SVM-Kernel:  linear
##        cost:  100
##       gamma:  0.09090909
##     epsilon:  0.1
##
##
## Number of Support Vectors:  815
```

Test the performance of the SVM on the test dataset.

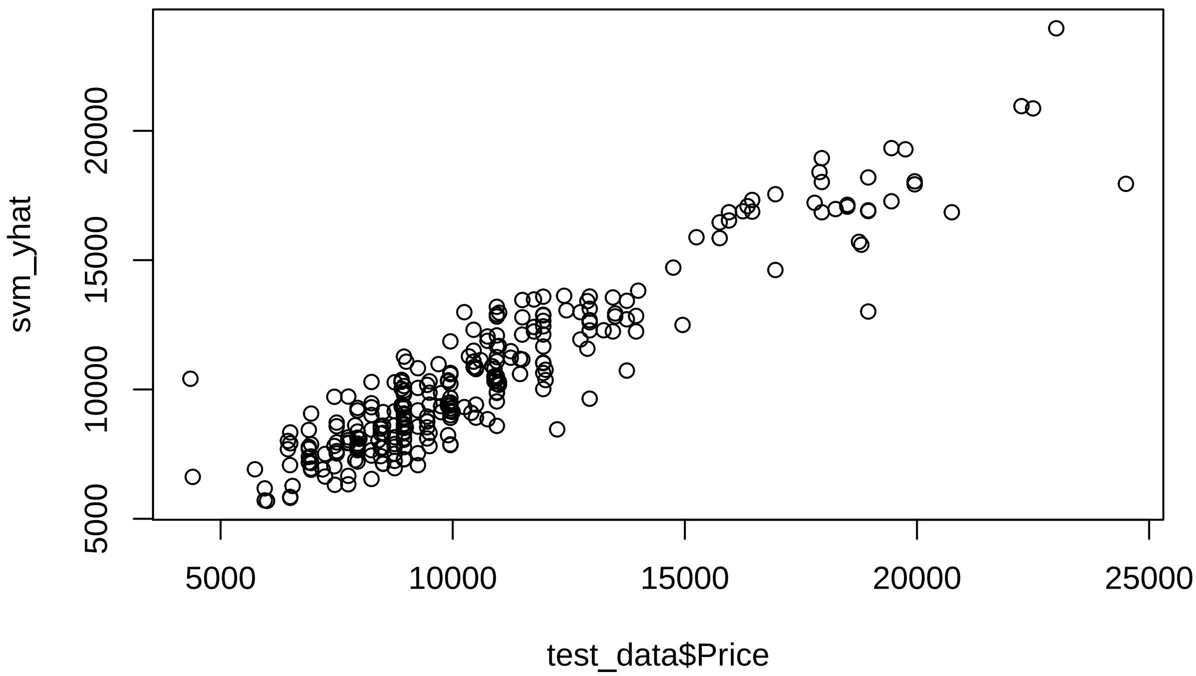
```r
# Predict on the scaled test dataset
svm_yhat_scaled <- predict(svm_corolla, newdata = test_scaled)

# Plot the normalized price and predicted normalized price
plot(test_scaled$Price,svm_yhat_scaled)
```




From the above plot, we notice that the price is in a very small range. This is because we normalize data before predictive modeling.

It makes more sense to transform the normalized data back to original scale when we evaluate the performance of the predictive model.

```
# Transform the normalized price prediction to original scale
svm_yhat <- svm_yhat_scaled*sd(train_data$Price) + mean(train_data$Price)

# Plot the price and predicted price
plot(test_data$Price,svm_yhat)
```




```r
# Calcalate prediction performane measures
postResample(svm_yhat, test_data$Price)
```

```
##         RMSE     Rsquared          MAE
## 1307.3803997    0.8648937  960.5435638
```

### 4.2. Tune a Linear Kernel

Use 10-fold cross validation to fine tune a linear kernel.

```r
set.seed(1)
# Tune the cost parameter for linear kernel
tune_svm_linear <- tune(svm, Price~., data = train_scaled,
                        kernel = 'linear',
                        tunecontrol=tune.control(cross=10,sampling="cross"),
                        ranges =list(cost=10^(-2:2)))

summary(tune_svm_linear)
```

```
##
## Parameter tuning of 'svm':
##
## - sampling method: 10-fold cross validation
##
```

```
## - best parameters:
##  cost
##  0.01
##
## - best performance: 0.1441666
##
## - Detailed performance results:
##    cost     error dispersion
## 1 1e-02 0.1441666 0.06468844
## 2 1e-01 0.1481840 0.08112129
## 3 1e+00 0.1466461 0.07750109
## 4 1e+01 0.1463488 0.07666046
## 5 1e+02 0.1463374 0.07667789
```

```r
# Print the best parameters
tune_svm_linear$best.parameters
```

```
##   cost
## 1 0.01
```

```r
# Print the best performance
tune_svm_linear$best.performance
```

```
## [1] 0.1441666
```

## 4.3. Tune a Radial (RBF) Kernel

```r
set.seed(1)
# Tune the cost and gamma parameters for radial kernel
tune_svm_radial <- tune(svm, Price~., data = train_scaled,
                       kernel = 'radial',
                       tunecontrol=tune.control(cross=10,sampling="cross"),
                       ranges =list(cost=10^(-2:2),gamma=10^(-2:2)))

# Print the best parameters
tune_svm_radial$best.parameters
```

```
##   cost gamma
## 4   10  0.01
```

```r
# Print the best performance
tune_svm_radial$best.performance
```

```
## [1] 0.09586199
```

## 4.4. Tune a Polynomial Kernel

To simplify the hyper-parameter tuning, here we only set to fine tune the cost parameter.

```
set.seed(1)
# Tune the cost and gamma parameters for polynormial kernel
tune_svm_poly <- tune(svm, Price~., data = train_scaled,
                      kernel = 'polynomial', degree = 2,
                      tunecontrol=tune.control(cross=10,sampling="cross"),
                      ranges =list(cost=10^(-2:2)))

# Print the best parameters
tune_svm_poly$best.parameters
```

```
##   cost
## 3    1
```

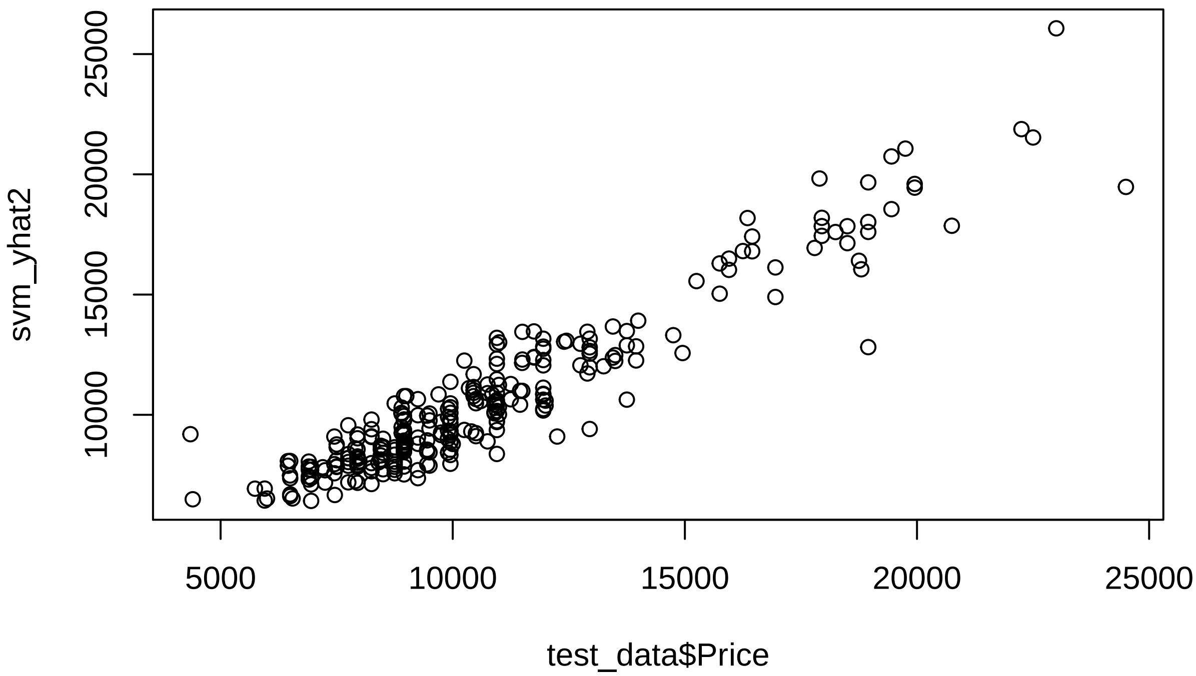
```
# Print the best performance
tune_svm_poly$best.performance
```

```
## [1] 0.1277619
```

### 4.5. Test the Best Model

From the above model tuning process, we know that the best model is the radial kernel with cost = 10 and gamma = 0.01 as it has the best performance (lowest error). Let's test the performance of the best SVM on the test dataset.

```
# Predict on the scaled test dataset
svm_yhat2_scaled <- predict(tune_svm_radial$best.model, newdata = test_scaled)

# Transform the normalized price prediction to original scale
svm_yhat2 <- svm_yhat2_scaled*sd(train_data$Price) + mean(train_data$Price)

# Plot the price and predicted price
plot(test_data$Price,svm_yhat2)
```




```r
# Calcalate prediction performane measures
postResample(svm_yhat2, test_data$Price)
```

```
##         RMSE     Rsquared          MAE
## 1173.1031957    0.8906976  870.1413565
```

The best model has a better performance than the linear kernel in section 4.1.