ADVANCE DATA MINING AND PREDICTIVE ANALYTICS - 3

shiva gadila

2023-12-06

QA1. What is the difference between SVM with hard margin and soft margin?

In Support Vector Machines (SVM), there are two main approaches: hard margin and soft margin.

Hard Margin SVM: Imagine you have data points of two different classes, and you want to draw a line (hyperplane) that perfectly separates them. This works well when the data is cleanly separable, but in real life, there might be some noisy or outlier points. Hard Margin SVM doesn't handle these outliers gracefully. It insists on a perfect division, and if there's even one misclassified point, it struggles.

Soft Margin SVM: Understanding that perfect separation might not be achievable in the real world, especially with noisy data, the soft margin approach is more forgiving. It introduces a concept called "slack variable." This allows for a bit of flexibility, meaning it permits some points to be on the wrong side of the dividing line. The challenge now becomes finding a balance between having a wide margin (space between classes) and allowing for a few misclassifications. There's a parameter called "C" that decides how much penalty is given for each misclassified point. You can think of it as the cost of allowing mistakes.

Summary: Hard Margin: Insists on a perfect division, not great with noisy data. Soft Margin: More flexible, accepts some misclassifications, and adjusts the balance with the "C" parameter. In simple terms, it's like trying to draw a line between two groups of points. Hard Margin wants a flawless line, while Soft Margin allows for a bit of messiness, deciding how much mess is acceptable with the "C" parameter. It's a way of handling the imperfect nature of real-world data.

QA2. What is the role of the cost parameter, C, in SVM (with soft margin) classifie?

The importance of the cost parameter, C, in SVM with a soft margin cannot be overstated. It acts as the guiding force in balancing the priorities of maximizing the margin and minimizing classification errors.

SVM focuses on creating a broader margin to enhance generalization, acknowledging the challenges of achieving perfect separation in real-world scenarios. C allows for some misclassifications but strives to keep them minimal while maintaining a significant margin.

The delicate balance controlled by the C parameter involves a trade-off. Lower C values prioritize a wider margin, accepting a higher tolerance for misclassifications. Conversely, higher C values emphasize reducing misclassifications, potentially leading to a narrower margin.

The impact of the chosen C value is significant. A high C risks overfitting by closely fitting the training data, capturing noise and outliers. In contrast, a low C may result in underfitting as the model leans heavily towards maximizing the margin, potentially neglecting some misclassifications.

Selecting the right C value is crucial for optimal SVM performance. Techniques like cross-validation help explore various C values, ensuring the choice that delivers the best performance on a validation set. This meticulous approach ensures the development of a well-balanced SVM model proficient in managing both margin maximization and classification error minimization.

Q3. Will the following perceptron be activated (2.8 is the activation threshold)?

Activation function = (input 1 x weight 1) + (input 2 x weight 2) = (0.1 x 0.8) + (11.1 x -0.2)

= 0.08 - 2.22

= -2.14

The activation function value is -2.14, which falls below the activation threshold of 2.8. Consequently, the perceptron will remain inactive in this scenario.

QA4. What is the role of alpha, the learning rate in the delta rule?

In the delta rule, a widely used algorithm for adjusting neural network weights during training, the learning rate, represented by the symbol alpha (α), plays a crucial role. Alpha is a hyperparameter that determines the size of the steps taken for weight updates generated by the delta rule.

Here’s how it works: The delta rule calculates the error between the predicted output of the neural network and the actual target output. This error is then utilized to tweak the network’s weights, aiming to enhance its overall performance. The extent of weight adjustment is linked to both the magnitude of the error and the learning rate.

Choosing the right value for alpha is key. A higher alpha leads to more substantial weight updates, facilitating faster convergence of weights. However, this can also result in overshooting the optimal weights and hinder convergence. Conversely, a lower alpha results in smaller weight updates and a longer convergence process. While this approach may assist in reaching a more accurate minimum, it requires patience during training.

Selecting an appropriate learning rate is a critical aspect of the delta rule, impacting the algorithm’s performance. A common strategy involves starting with a small alpha, such as 0.1 or 0.01, and then fine-tuning it through experimentation to achieve optimal results on the training data.

PART B

```
#QB1. Build a linear SVM regression model to predict Sales based on all other
attributes ("Price","Advertising", "Population", "Age", "Income" and
"Education"). Hint: use caret train() with method set to “svmLinear”. What is
the R-squared of the model?

library(ISLR)
library(dplyr)

##
## Attaching package: 'dplyr'

## The following objects are masked from 'package:stats':
##
##     filter, lag

## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union

library(glmnet)

## Loading required package: Matrix
```

```
## Warning: package 'Matrix' was built under R version 4.2.3

## Loaded glmnet 4.1-8

library(caret)

## Loading required package: ggplot2

## Loading required package: lattice

Carseats_Filtered <- Carseats %>% select("Sales", "Price",
"Advertising","Population","Age","Income","Education")

set.seed(123)
trainIndex <- createDataPartition(Carseats_Filtered$Sales, p = 0.7, list =
FALSE)
Train_Data <- Carseats_Filtered[trainIndex, ]
Test_Data <- Carseats_Filtered[-trainIndex, ]

#Set up the model.
Support_Vector_Machine_Model <- train(Sales ~.,
               data = Train_Data,
               method = "svmLinear",
               trControl = trainControl(method ="cv",number
                                        = 10))

#Presenting the model's summary information.
summary(Support_Vector_Machine_Model)

## Length  Class   Mode
##      1   ksvm     S4

#Make predictions using the test dataset.
Predictions <- predict(Support_Vector_Machine_Model, newdata = Test_Data)

#Determine the R-squared value.
R_squared<- postResample(Predictions, Test_Data$Sales)
R_squared

##     RMSE Rsquared      MAE
## 2.297064 0.385761 1.876610

#QB2. Customize the search grid by checking the model’s performance for C
parameter of 0.1,.5,1 and 10 using 2 repeats of 5-fold cross validation.

library(caret)
Grid <- expand.grid(C = c(0.1,0.5,1,10))
trctrl2 <- trainControl(method = "repeatedcv", number = 5,repeats = 2)
svm_Linear_Grid <- train(Sales~., data = Carseats_Filtered, method =
"svmLinear",
                         trControl=trctrl2,
                         preProcess = c("center", "scale"),
                         tuneGrid = Grid,
```

```
                          tuneLength = 10)
svm_Linear_Grid

## Support Vector Machines with Linear Kernel
##
## 400 samples
##   6 predictor
##
## Pre-processing: centered (6), scaled (6)
## Resampling: Cross-Validated (5 fold, repeated 2 times)
## Summary of sample sizes: 319, 320, 320, 321, 320, 320, ...
## Resampling results across tuning parameters:
##
##   C     RMSE      Rsquared   MAE
##    0.1  2.270125  0.3649327  1.818221
##    0.5  2.269631  0.3642787  1.816916
##    1.0  2.269468  0.3642989  1.816430
##   10.0  2.269407  0.3643307  1.816397
##
## RMSE was used to select the optimal model using the smallest value.
## The final value used for the model was C = 10.

#QB3. Train a neural network model to predict Sales based on all other
attributes ("Price","Advertising", "Population", "Age", "Income" and
"Education"). Hint: use caret train() with method set to "nnet". What is the
R-square of the model with the best hyper parameters (using default caret
search grid) – hint: don't forget to scale the data.


#Data scaling
scaled <- preProcess(Carseats_Filtered[-1], method = "scale")
train <- predict(scaled, Carseats_Filtered[-1])
#Train control
set.seed(27)
folds <- trainControl(method = "repeatedcv",
                      number = 5,
                      repeats = 2,
                      verboseIter = FALSE)
# Using default search grid, training the neural network model
set.seed(123)
nnet_Cars <- train(Sales~.,data = Carseats_Filtered ,
                    method = "nnet",
                    trControl = folds)

## # weights:  9
## initial  value 17583.375508
## final  value 15744.713000
## converged
## # weights:  25
## initial  value 18250.380996
```

```
## final  value 15744.713000
## converged
## # weights:  41
## initial  value 17969.872043
## final  value 15744.713000
## converged
## # weights:  9
## initial  value 17485.862634
## iter  10 value 15752.495629
## iter  20 value 15749.181183
## final  value 15749.165782
## converged
## # weights:  25
## initial  value 16731.281643
## iter  10 value 15750.726329
## iter  20 value 15747.342759
## iter  30 value 15747.245743
## iter  30 value 15747.245604
## iter  30 value 15747.245554
## final  value 15747.245554
## converged
## # weights:  41
## initial  value 17717.818624
## iter  10 value 15761.704139
## iter  20 value 15746.593674
## final  value 15746.528022
## converged
## # weights:  9
## initial  value 18907.374924
## iter  10 value 15751.310870
## iter  20 value 15744.789068
## iter  20 value 15744.789038
## final  value 15744.789038
## converged
## # weights:  25
## initial  value 17691.631949
## iter  10 value 15766.322138
## iter  20 value 15744.962136
## iter  30 value 15744.727758
## iter  30 value 15744.727752
## iter  30 value 15744.727746
## final  value 15744.727746
## converged
## # weights:  41
## initial  value 18730.502990
## iter  10 value 15752.934200
## iter  20 value 15744.986373
## final  value 15744.735157
## converged
## # weights:  9
```

```
## initial  value 18150.262633
## final  value 16133.992900
## converged
## # weights:  25
## initial  value 18057.338235
## final  value 16133.992900
## converged
## # weights:  41
## initial  value 17359.482931
## final  value 16133.992900
## converged
## # weights:  9
## initial  value 17989.498119
## iter  10 value 16200.459178
## iter  20 value 16138.492102
## final  value 16138.454408
## converged
## # weights:  25
## initial  value 18745.698886
## iter  10 value 16153.006756
## iter  20 value 16137.256703
## iter  30 value 16136.548298
## final  value 16136.530460
## converged
## # weights:  41
## initial  value 17459.477458
## iter  10 value 16143.635989
## iter  20 value 16136.062684
## iter  30 value 16135.826166
## iter  40 value 16135.818629
## iter  40 value 16135.818513
## final  value 16135.811494
## converged
## # weights:  9
## initial  value 17665.772198
## iter  10 value 16142.809710
## iter  20 value 16134.094551
## iter  20 value 16134.094510
## final  value 16134.094510
## converged
## # weights:  25
## initial  value 17838.240242
## iter  10 value 16143.041882
## iter  20 value 16134.097228
## iter  30 value 16134.005971
## iter  30 value 16134.005867
## iter  30 value 16134.005862
## final  value 16134.005862
## converged
## # weights:  41
```

```
## initial  value 17513.611852
## iter  10 value 16144.718662
## iter  20 value 16134.116560
## final  value 16134.008897
## converged
## # weights:  9
## initial  value 18527.696365
## final  value 16054.878300
## converged
## # weights:  25
## initial  value 17788.524754
## final  value 16054.878300
## converged
## # weights:  41
## initial  value 17446.488348
## final  value 16054.878300
## converged
## # weights:  9
## initial  value 18790.788824
## iter  10 value 16060.229113
## iter  20 value 16059.341696
## final  value 16059.340150
## converged
## # weights:  25
## initial  value 18626.938869
## iter  10 value 16082.809489
## iter  20 value 16057.679412
## iter  30 value 16057.431616
## final  value 16057.416615
## converged
## # weights:  41
## initial  value 18338.133205
## iter  10 value 16153.067658
## iter  20 value 16058.367215
## iter  30 value 16057.518420
## iter  40 value 16056.957883
## iter  50 value 16056.696790
## iter  50 value 16056.696704
## iter  50 value 16056.696682
## final  value 16056.696682
## converged
## # weights:  9
## initial  value 17590.373920
## iter  10 value 16059.385823
## iter  20 value 16054.930268
## iter  20 value 16054.930247
## final  value 16054.930247
## converged
## # weights:  25
## initial  value 17195.152150
```

```
## iter  10 value 16063.994339
## iter  20 value 16054.983401
## final  value 16054.887538
## converged
## # weights:  41
## initial  value 19381.816748
## iter  10 value 16074.408053
## iter  20 value 16055.103463
## iter  30 value 16054.916332
## final  value 16054.893441
## converged
## # weights:  9
## initial  value 18955.393490
## final  value 16058.233500
## converged
## # weights:  25
## initial  value 18731.157319
## final  value 16058.233500
## converged
## # weights:  41
## initial  value 17519.676097
## final  value 16058.233500
## converged
## # weights:  9
## initial  value 18443.945205
## iter  10 value 16062.703228
## final  value 16062.691766
## converged
## # weights:  25
## initial  value 17324.504702
## iter  10 value 16063.061711
## iter  20 value 16060.975591
## iter  30 value 16060.769179
## iter  30 value 16060.769058
## iter  30 value 16060.768932
## final  value 16060.768932
## converged
## # weights:  41
## initial  value 17182.149602
## iter  10 value 16062.517552
## iter  20 value 16060.350351
## iter  30 value 16060.151610
## iter  40 value 16060.063571
## final  value 16060.050628
## converged
## # weights:  9
## initial  value 17573.438755
## iter  10 value 16062.663964
## iter  20 value 16058.284580
## iter  20 value 16058.284559
```

```
## final  value 16058.284559
## converged
## # weights:  25
## initial  value 17709.202204
## iter  10 value 16070.257094
## iter  20 value 16058.372123
## final  value 16058.252642
## converged
## # weights:  41
## initial  value 18558.603853
## iter  10 value 16075.463049
## iter  20 value 16058.432143
## final  value 16058.249622
## converged
## # weights:  9
## initial  value 18722.292207
## final  value 16260.862700
## converged
## # weights:  25
## initial  value 17799.426161
## final  value 16260.862700
## converged
## # weights:  41
## initial  value 18255.742928
## final  value 16260.862700
## converged
## # weights:  9
## initial  value 17481.607340
## iter  10 value 16273.839719
## iter  20 value 16265.331018
## final  value 16265.323262
## converged
## # weights:  25
## initial  value 18295.365403
## iter  10 value 16326.205795
## iter  20 value 16265.170871
## iter  30 value 16263.576040
## final  value 16263.399405
## converged
## # weights:  41
## initial  value 17990.022132
## iter  10 value 16263.841721
## iter  20 value 16262.695365
## final  value 16262.680613
## converged
## # weights:  9
## initial  value 18295.931560
## iter  10 value 16271.756840
## iter  20 value 16260.988301
## iter  20 value 16260.988251
```

```
## final  value 16260.988251
## converged
## # weights:  25
## initial  value 18603.949520
## iter  10 value 16276.382456
## iter  20 value 16261.041631
## iter  30 value 16260.871843
## iter  30 value 16260.871810
## final  value 16260.870307
## converged
## # weights:  41
## initial  value 18012.253619
## iter  10 value 16277.963842
## iter  20 value 16261.059863
## iter  30 value 16260.871990
## iter  30 value 16260.871973
## final  value 16260.871512
## converged
## # weights:  9
## initial  value 17699.831901
## final  value 16275.959300
## converged
## # weights:  25
## initial  value 19641.795202
## final  value 16275.959300
## converged
## # weights:  41
## initial  value 18752.645654
## final  value 16275.959300
## converged
## # weights:  9
## initial  value 17240.269435
## iter  10 value 16283.782698
## final  value 16280.421142
## converged
## # weights:  25
## initial  value 18553.272473
## iter  10 value 16279.424338
## iter  20 value 16278.558130
## iter  30 value 16278.527772
## final  value 16278.496768
## converged
## # weights:  41
## initial  value 17756.873961
## iter  10 value 16280.244531
## iter  20 value 16277.829378
## iter  30 value 16277.780248
## final  value 16277.777787
## converged
## # weights:  9
```

```
## initial  value 19121.132256
## iter  10 value 16284.843993
## iter  20 value 16276.061734
## iter  20 value 16276.061693
## final  value 16276.061693
## converged
## # weights:  25
## initial  value 18036.303840
## iter  10 value 16281.085424
## iter  20 value 16276.018400
## final  value 16275.980929
## converged
## # weights:  41
## initial  value 19355.400213
## iter  10 value 16288.718731
## iter  20 value 16276.106406
## final  value 16275.970733
## converged
## # weights:  9
## initial  value 18050.702295
## final  value 15928.817500
## converged
## # weights:  25
## initial  value 18673.513835
## final  value 15928.817500
## converged
## # weights:  41
## initial  value 18478.275519
## final  value 15928.817500
## converged
## # weights:  9
## initial  value 18032.649747
## iter  10 value 15935.432118
## final  value 15933.274268
## converged
## # weights:  25
## initial  value 18953.220557
## iter  10 value 15937.188701
## iter  20 value 15932.025175
## iter  30 value 15931.704036
## iter  40 value 15931.355821
## final  value 15931.352118
## converged
## # weights:  41
## initial  value 17775.320098
## iter  10 value 15931.261131
## iter  20 value 15930.730135
## iter  30 value 15930.634420
## iter  30 value 15930.634354
## iter  30 value 15930.634354
```

```
## final  value 15930.634354
## converged
## # weights:  9
## initial  value 17439.224458
## iter  10 value 15933.303079
## iter  20 value 15928.869215
## iter  20 value 15928.869195
## final  value 15928.869195
## converged
## # weights:  25
## initial  value 17831.381529
## iter  10 value 15939.629190
## iter  20 value 15928.942150
## final  value 15928.827000
## converged
## # weights:  41
## initial  value 16885.798107
## iter  10 value 15943.011273
## iter  20 value 15928.981143
## final  value 15928.827662
## converged
## # weights:  9
## initial  value 19001.093244
## final  value 16026.916900
## converged
## # weights:  25
## initial  value 18234.740751
## final  value 16026.916900
## converged
## # weights:  41
## initial  value 17444.773107
## final  value 16026.916900
## converged
## # weights:  9
## initial  value 19226.609443
## iter  10 value 16031.661108
## final  value 16031.376945
## converged
## # weights:  25
## initial  value 17912.050438
## iter  10 value 16099.176152
## iter  20 value 16031.084078
## iter  30 value 16030.127163
## final  value 16030.125564
## converged
## # weights:  41
## initial  value 19636.734646
## iter  10 value 16041.491939
## iter  20 value 16030.108778
## iter  30 value 16028.945067
```

```
## iter  40 value 16028.760813
## iter  50 value 16028.737686
## final  value 16028.735586
## converged
## # weights:  9
## initial  value 17894.055196
## iter  10 value 16037.383250
## iter  20 value 16027.037569
## iter  20 value 16027.037521
## final  value 16027.037521
## converged
## # weights:  25
## initial  value 17695.042357
## iter  10 value 16036.926838
## iter  20 value 16027.032307
## final  value 16026.947386
## converged
## # weights:  41
## initial  value 16889.040753
## iter  10 value 16033.782658
## iter  20 value 16026.996057
## final  value 16026.924364
## converged
## # weights:  9
## initial  value 18092.146826
## final  value 16199.115500
## converged
## # weights:  25
## initial  value 18173.879067
## final  value 16199.115500
## converged
## # weights:  41
## initial  value 19348.973466
## final  value 16199.115500
## converged
## # weights:  9
## initial  value 17686.176332
## iter  10 value 16205.168079
## final  value 16203.578217
## converged
## # weights:  25
## initial  value 18221.044579
## iter  10 value 16206.649650
## iter  20 value 16201.780743
## iter  30 value 16201.659845
## final  value 16201.653530
## converged
## # weights:  41
## initial  value 18007.644613
## iter  10 value 16236.916646
```

```
## iter  20 value 16201.368158
## iter  30 value 16200.961996
## final  value 16200.935019
## converged
## # weights:  9
## initial  value 18647.736276
## iter  10 value 16206.810589
## iter  20 value 16199.204218
## iter  20 value 16199.204183
## final  value 16199.204183
## converged
## # weights:  25
## initial  value 17681.544301
## iter  10 value 16203.519924
## iter  20 value 16199.166280
## final  value 16199.138361
## converged
## # weights:  41
## initial  value 18978.786685
## iter  10 value 16214.630998
## iter  20 value 16199.294382
## iter  30 value 16199.129205
## iter  30 value 16199.129191
## final  value 16199.128531
## converged
## # weights:  9
## initial  value 18177.074240
## final  value 15821.871200
## converged
## # weights:  25
## initial  value 18778.113264
## final  value 15821.871200
## converged
## # weights:  41
## initial  value 16909.381852
## final  value 15821.871200
## converged
## # weights:  9
## initial  value 17119.260930
## iter  10 value 15826.599396
## iter  20 value 15826.324908
## final  value 15826.324509
## converged
## # weights:  25
## initial  value 18703.665234
## iter  10 value 15832.759440
## iter  20 value 15826.738148
## iter  30 value 15824.517679
## iter  40 value 15824.417539
## iter  50 value 15824.405208
```

```
## final  value 15824.404153
## converged
## # weights:  41
## initial  value 18070.771384
## iter  10 value 15824.511212
## iter  20 value 15823.959776
## iter  30 value 15823.687857
## final  value 15823.686523
## converged
## # weights:  9
## initial  value 18880.142624
## iter  10 value 15828.619435
## iter  20 value 15821.949002
## iter  20 value 15821.948971
## final  value 15821.948971
## converged
## # weights:  25
## initial  value 17263.346538
## iter  10 value 15823.989488
## iter  20 value 15821.994572
## final  value 15821.983647
## converged
## # weights:  41
## initial  value 17737.097268
## iter  10 value 15841.716075
## iter  20 value 15822.099996
## final  value 15821.883136
## converged
```

```
## Warning in nominalTrainWorkflow(x = x, y = y, wts = weights, info =
trainInfo,
## : There were missing values in resampled performance measures.
```

```
## # weights:  9
## initial  value 23385.579068
## iter  10 value 20079.130735
## iter  20 value 20063.354114
## iter  20 value 20063.354040
## final  value 20063.354040
## converged
```

```
nnet_Cars
```

```
## Neural Network
##
## 400 samples
##   6 predictor
##
## No pre-processing
## Resampling: Cross-Validated (5 fold, repeated 2 times)
## Summary of sample sizes: 320, 321, 319, 320, 320, 319, ...
```

```
## Resampling results across tuning parameters:
##
##   size  decay  RMSE      Rsquared    MAE
##   1     0e+00  7.080831         NaN  6.511777
##   1     1e-04  7.080831         NaN  6.511777
##   1     1e-01  7.081018  0.02150646  6.511976
##   3     0e+00  7.080831         NaN  6.511777
##   3     1e-04  7.080831  0.05189222  6.511777
##   3     1e-01  7.080934  0.03272925  6.511887
##   5     0e+00  7.080831         NaN  6.511777
##   5     1e-04  7.080831  0.03626897  6.511777
##   5     1e-01  7.080900  0.02608450  6.511851
##
## RMSE was used to select the optimal model using the smallest value.
## The final values used for the model were size = 1 and decay = 1e-04.
```

```r
#The optimal model was chosen based on the RMSE (root mean squared error),
featuring a size of 1 and a decay of 0. The RMSE for this model was 7.081637,
accompanied by a corresponding MAE (mean absolute error) value of 6.511536.
While the Rsquared value was 'not available' for the optimal model, it ranged
from NaN to 0.02538470 for other models.

#QB4 - “Consider the following input: Sales=9, Price=6.54, Population=124,
Advertising=0, Age=76, Income= 110, Education=10 What will be the estimated
Sales for this record using the above neuralnet model?”

Sales <- c(9)
Price <- c(6.54)
Population <- c(124)
Advertising <- c(0)
Age <- c(76)
Income <- c(110)
Education <- c(10)
Test <- data.frame(Sales, Price, Population, Advertising, Age, Income,
Education)

# Making Predictions
Predicting_sales <- predict(nnet_Cars, Test)
Predicting_sales
```

```
## 1
## 1
```