INFS_SP5_2023
Predictive Analytics
week11

Enna H

Contents


```
# Load libraries
pacman::p_load(pscl, ROCR, glmnet,mice,rpart,pROC)
# for cross validation
pacman::p_load(caret,rpart.plot)

pacman::p_load(class, mlr3, mlr3learners, mlr3measures, C50)
```

Support Vector Machine (SVM)

Understand basic principles behind SVM. • Implement SVM (with linear kernel) using R. • Tune parameters for SVM with linear kernel. • Tune parameters for SVM with polynomial kernel. • Tune parameters for SVM with rbf kernel.

Task1. Introduction to SVM

In very simple terms, the main idea of SVM is to find an optimal hyperplane that maximizes the margin between two classes. This hyperplane is simply a line in 2D space (if you plot two features like we did last week), plane in 3D and hyperplane when we have more than three dimensions.

Support vectors are data points that support hyperplane on either side, as displayed below (Figure 1).

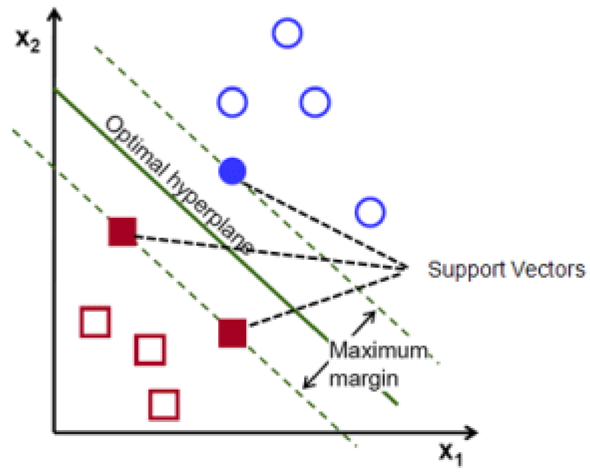

Figure 1. Support vectors for an optimal hyperplane.

SVM uses different kernels in case of non-linearly separable data points. Different kernel functions transform the data into a higher dimensional feature space to make it possible to perform the linear separation. Depending on the library you use, there are various implementations of the SVM kernel, including polynomial, Gaussian, Gaussian radial basis function (RBF), Laplace RBF, hyperbolic tangent, sigmoid, and linear kernel.

When working with the linear kernel there is only one parameter we should configure - C or regularization parameter. Can you explain what would happen when you have large C? What about when using very small C?

Linear Kernel and Regularization Parameter (C) in SVM

Basic Understanding: Support Vector Machines (SVM) are supervised learning algorithms designed for classification and regression tasks. The regularization parameter $ C $ controls the trade-off between maximizing the margin between classes and minimizing the classification error on the training set.

Technical Details: The parameter $ C $ is crucial in determining the flexibility of the decision boundary. It essentially dictates the degree to which SVM should allow misclassification.

1. **Large C Value**:
   - **Low Bias, High Variance**: A large $ C $ makes the optimization focus on correctly classifying each data point, even at the expense of a potentially convoluted decision boundary.
   - **Overfitting**: It is highly susceptible to noise and outliers in the training set, leading to overfitting.
   - **Narrow Margin**: The margin, which is the distance between the closest points of different classes, will be narrower.
   - **Optimization**: Solving the optimization problem may become computationally expensive due to the complex decision boundary.
2. **Small C Value**:
   - **High Bias, Low Variance**: A small $ C $ means the algorithm is less concerned with misclassifying some training examples, favoring a smoother and simpler decision boundary.
   - **Underfitting**: The model may become too generalized and perform poorly on unseen data.

- **Wide Margin**: The margin between classes will be wider.
- **Optimization**: Generally quicker to compute due to the simpler boundary.

Significance: Choosing an appropriate $ C $ is essential for model performance. While small $ C $ values may provide faster computation and a more generalized model, they risk underfitting. Conversely, large $ C $ values offer precise training classification but can lead to overfitting and increased computational costs.

Applications: Model selection techniques like cross-validation can be employed to find the optimal $ C $ value for specific tasks. Grid search and randomized search are commonly used methods to tune $ C $ efficiently.

Task2. Implement SVM (with linear kernel) using R

```r
# Load libraries
pacman::p_load(e1071, caret, ggplot2, GGally, kernlab)
```

At the end this practical, we will compare results we obtained with other three classifiers (rule-based, kNN and Naïve Bayes) versus SVM.

```r
# Load data
data <-read.csv(url("https://raw.githubusercontent.com/sreckojoksimovic/infs5100/main/wine-data.csv"))
```

make sure outcome variable is in the right format

```r
data$quality_class <- as.factor(data$quality_class)
```

summarize the obtained dataset.

```r
# Data input validation
head(data)
```

```
##   fixed_acidity volatile_acidity citric_acid residual_sugar chlorides
## 1           7.3             0.65        0.00            1.2     0.065
## 2           7.8             0.58        0.02            2.0     0.073
## 3           8.5             0.28        0.56            1.8     0.092
## 4           8.1             0.38        0.28            2.1     0.066
## 5           7.5             0.52        0.16            1.9     0.085
## 6           8.0             0.59        0.16            1.8     0.065
##   free_sulfur_dioxide total_sulfur_dioxide density   pH sulphates alcohol
## 1                  15                   21  0.9946 3.39      0.47    10.0
## 2                   9                   18  0.9968 3.36      0.57     9.5
## 3                  35                  103  0.9969 3.30      0.75    10.5
## 4                  13                   30  0.9968 3.23      0.73     9.7
## 5                  12                   35  0.9968 3.38      0.62     9.5
## 6                   3                   16  0.9962 3.42      0.92    10.5
##   quality_class
## 1             1
## 2             1
## 3             1
```

```
## 4              1
## 5              1
## 6              1
```

```
summary(data)
```

```
##  fixed_acidity    volatile_acidity  citric_acid    residual_sugar
##  Min.   : 4.700   Min.   :0.1200   Min.   :0.000   Min.   : 0.900
##  1st Qu.: 7.100   1st Qu.:0.3900   1st Qu.:0.100   1st Qu.: 1.900
##  Median : 7.900   Median :0.5100   Median :0.260   Median : 2.200
##  Mean   : 8.338   Mean   :0.5198   Mean   :0.275   Mean   : 2.533
##  3rd Qu.: 9.300   3rd Qu.:0.6300   3rd Qu.:0.430   3rd Qu.: 2.600
##  Max.   :15.900   Max.   :1.3300   Max.   :0.790   Max.   :15.500
##    chlorides       free_sulfur_dioxide total_sulfur_dioxide    density
##  Min.   :0.01200   Min.   : 1.00       Min.   :  6.00       Min.   :0.9901
##  1st Qu.:0.07000   1st Qu.: 8.00       1st Qu.: 22.00       1st Qu.:0.9956
##  Median :0.07900   Median :14.00       Median : 38.00       Median :0.9968
##  Mean   :0.08713   Mean   :16.03       Mean   : 46.96       Mean   :0.9967
##  3rd Qu.:0.09000   3rd Qu.:22.00       3rd Qu.: 63.00       3rd Qu.:0.9979
##  Max.   :0.61100   Max.   :72.00       Max.   :289.00       Max.   :1.0037
##        pH          sulphates         alcohol      quality_class
##  Min.   :2.860   Min.   :0.3700   Min.   : 8.40   0:1319
##  1st Qu.:3.210   1st Qu.:0.5500   1st Qu.: 9.50   1: 217
##  Median :3.310   Median :0.6200   Median :10.20
##  Mean   :3.308   Mean   :0.6609   Mean   :10.43
##  3rd Qu.:3.400   3rd Qu.:0.7300   3rd Qu.:11.10
##  Max.   :4.010   Max.   :1.9800   Max.   :14.90
```

split the data into training and testing sets.

```
# Split data into training and test datasets. We will use 70%/30% split
# again.
set.seed(123)
dat.d <- sample(1:nrow(data),size=nrow(data)*0.7,replace = FALSE) #random selection of 70% data.
train.data <- data[dat.d,] # 70% training data
test.data <- data[-dat.d,] # remaining 30% test data
head(train.data)
```

```
##     fixed_acidity volatile_acidity citric_acid residual_sugar chlorides
## 415           7.8             0.34        0.37            2.0     0.082
## 463          11.5             0.18        0.51            4.0     0.104
## 179          10.6             0.36        0.57            2.3     0.087
## 526          13.5             0.53        0.79            4.8     0.120
## 195           7.0             0.60        0.12            2.2     0.083
## 938           9.9             0.25        0.46            1.7     0.062
##     free_sulfur_dioxide total_sulfur_dioxide density   pH sulphates alcohol
## 415                  24                   58 0.99640 3.34      0.59     9.4
## 463                   4                   23 0.99960 3.28      0.97    10.1
## 179                   6                   20 0.99676 3.14      0.72    11.1
## 526                  23                   77 1.00180 3.18      0.77    13.0
## 195                  13                   28 0.99660 3.52      0.62    10.2
## 938                  26                   42 0.99590 3.18      0.83    10.6
##     quality_class
```

```
## 415              0
## 463              0
## 179              1
## 526              0
## 195              1
## 938              0
```

```r
head(test.data)
```

```
##    fixed_acidity volatile_acidity citric_acid residual_sugar chlorides
## 3            8.5             0.28        0.56            1.8     0.092
## 12           5.2             0.48        0.04            1.6     0.054
## 14          15.0             0.21        0.44            2.2     0.075
## 15          10.0             0.31        0.47            2.6     0.085
## 20           7.7             0.27        0.68            3.5     0.358
## 21           8.9             0.40        0.32            5.6     0.087
##    free_sulfur_dioxide total_sulfur_dioxide density   pH sulphates alcohol
## 3                   35                  103 0.99690 3.30      0.75    10.5
## 12                  19                  106 0.99270 3.54      0.62    12.2
## 14                  10                   24 1.00005 3.07      0.84     9.2
## 15                  14                   33 0.99965 3.36      0.80    10.5
## 20                   5                   10 0.99720 3.25      1.08     9.9
## 21                  10                   47 0.99910 3.38      0.77    10.5
##    quality_class
## 3              1
## 12             1
## 14             1
## 15             1
## 20             1
## 21             1
```

```r
nrow(train.data)
```

```
## [1] 1075
```

```r
nrow(test.data)
```

```
## [1] 461
```

Building an SVM model using the linear kernel is rather straightforward.

```r
# Build SVM model using linear kernel
svm.model <- svm(quality_class ~ ., data = train.data, kernel = "linear")
```

To be able to obtain a confusion matrix and calculate model parameters, we will call predict function on the test data. Please note that here we are removing class variable, as that is what we are trying to predict.

```r
svm.pred = predict(svm.model,test.data[, -12])
```

obtain the confusion matrix.

```
svm.results = confusionMatrix(table(predicted = svm.pred,
              actual = test.data$quality_class))
svm.results
```

```
## Confusion Matrix and Statistics
##
##          actual
## predicted   0   1
##         0 391  70
##         1   0   0
##
##                Accuracy : 0.8482
##                  95% CI : (0.8121, 0.8797)
##     No Information Rate : 0.8482
##     P-Value [Acc > NIR] : 0.5318
##
##                   Kappa : 0
##
##  Mcnemar's Test P-Value : <2e-16
##
##             Sensitivity : 1.0000
##             Specificity : 0.0000
##          Pos Pred Value : 0.8482
##          Neg Pred Value :    NaN
##              Prevalence : 0.8482
##          Detection Rate : 0.8482
##    Detection Prevalence : 1.0000
##       Balanced Accuracy : 0.5000
##
##        'Positive' Class : 0
##
```

We mentioned in the lecture that we should always tune SVM parameters to find an optimal model. Also, as we can see above that although the accuracy is considerably high, Kappa value is 0. Looking into the confusion matrix, it is obvious that this classifier simply assigns 0 to each data instance.

One way to tune parameters for SVM with linear kernel would be as follows:

```
# Parameter tuning - linear kernel
set.seed(999)
svm.linear.tune = tune.svm(quality_class~.,
    data=train.data,
    kernel="linear",
    cost=c(0.001, 0.01, 0.1, 1, 5, 10)) # cost parameter
summary(svm.linear.tune)
```

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

```
##  0.001
##
## - best performance: 0.1366822
##
## - Detailed performance results:
##    cost     error dispersion
## 1 1e-03 0.1366822 0.02907684
## 2 1e-02 0.1366822 0.02907684
## 3 1e-01 0.1366822 0.02907684
## 4 1e+00 0.1366822 0.02907684
## 5 5e+00 0.1366822 0.02907684
## 6 1e+01 0.1366822 0.02907684
```

setting different values for the cost parameter. Once we finish, we can use the parameters for the best model .

```
# Optimal model for linear kernel
svm.best.linear = svm.linear.tune$best.model
svm.tune.linear.pred = predict(svm.best.linear, newdata=test.data[, -12])
confusionMatrix(svm.tune.linear.pred, test.data$quality_class)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction   0   1
##          0 391  70
##          1   0   0
##
##                Accuracy : 0.8482
##                  95% CI : (0.8121, 0.8797)
##     No Information Rate : 0.8482
##     P-Value [Acc > NIR] : 0.5318
##
##                   Kappa : 0
##
##  Mcnemar's Test P-Value : <2e-16
##
##             Sensitivity : 1.0000
##             Specificity : 0.0000
##          Pos Pred Value : 0.8482
##          Neg Pred Value :    NaN
##              Prevalence : 0.8482
##          Detection Rate : 0.8482
##    Detection Prevalence : 1.0000
##       Balanced Accuracy : 0.5000
##
##        'Positive' Class : 0
##
```

Not much has changed, compared to the first model we tried. That is why we will test other kernels. In this practical, we will explore polynomial and rbf kernels.

Task 3. SVM in R - polynomial and rbf kernels

Finding an optimal model for polynomial and rbf kernels is quite similar to what we had for the linear kernel. If you would like to explore what are the parameters that we need to configure for each kernel, you can run ?tune.svm() in R console.

```
# Parameter tuning - polynomial kernel
set.seed(999)
svm.poly.tune = tune.svm(quality_class~., data=train.data,
                kernel="polynomial",
                degree=c(3,4,5), coef0=c(0.001, 0.01, 0.1, 1, 5, 10))
summary(svm.poly.tune)
```

```
##
## Parameter tuning of 'svm':
##
## - sampling method: 10-fold cross validation
##
## - best parameters:
##  degree coef0
##       3     5
##
## - best performance: 0.1134476
##
## - Detailed performance results:
##    degree coef0     error dispersion
## 1       3 1e-03 0.1152821 0.02074193
## 2       4 1e-03 0.1310921 0.02645591
## 3       5 1e-03 0.1273451 0.03583300
## 4       3 1e-02 0.1162167 0.01984027
## 5       4 1e-02 0.1283056 0.02625936
## 6       5 1e-02 0.1245587 0.03453361
## 7       3 1e-01 0.1162080 0.02206846
## 8       4 1e-01 0.1143648 0.01981162
## 9       5 1e-01 0.1180599 0.03592229
## 10      3 1e+00 0.1171772 0.01737077
## 11      4 1e+00 0.1143821 0.01934059
## 12      5 1e+00 0.1264798 0.02579180
## 13      3 5e+00 0.1134476 0.01820294
## 14      4 5e+00 0.1190723 0.02932737
## 15      5 5e+00 0.1366390 0.03570928
## 16      3 1e+01 0.1153340 0.01701622
## 17      4 1e+01 0.1264884 0.03271407
## 18      5 1e+01 0.1375909 0.03353378
```

we can obtain the confusion matrix.

```
svm.best.poly = svm.poly.tune$best.model
svm.tune.poly.pred = predict(svm.best.poly, newdata=test.data[, -12])
confusionMatrix(svm.tune.poly.pred, test.data$quality_class)
```

```
## Confusion Matrix and Statistics
##
```

```
##           Reference
## Prediction   0   1
##          0 376  40
##          1  15  30
##
##                Accuracy : 0.8807
##                  95% CI : (0.8476, 0.9088)
##     No Information Rate : 0.8482
##     P-Value [Acc > NIR] : 0.027123
##
##                   Kappa : 0.4572
##
##  Mcnemar's Test P-Value : 0.001211
##
##             Sensitivity : 0.9616
##             Specificity : 0.4286
##          Pos Pred Value : 0.9038
##          Neg Pred Value : 0.6667
##              Prevalence : 0.8482
##          Detection Rate : 0.8156
##    Detection Prevalence : 0.9024
##       Balanced Accuracy : 0.6951
##
##        'Positive' Class : 0
##
```

As we can notice, model parameters look much better now. We will repeat the same process for the rbf kernel.

```
# Parameter tuning - rbf kernel
set.seed(999)
svm.rbf.tune = tune.svm(quality_class~., data=train.data,
               kernel="radial",
               gamma=c(0.001, 0.1, 0.5, 1, 5, 10))
summary(svm.rbf.tune)
```

```
##
## Parameter tuning of 'svm':
##
## - sampling method: 10-fold cross validation
##
## - best parameters:
##  gamma
##      1
##
## - best performance: 0.1022759
##
## - Detailed performance results:
##   gamma     error dispersion
## 1 1e-03 0.1366822 0.02907684
## 2 1e-01 0.1125476 0.01770730
## 3 5e-01 0.1032278 0.02401985
## 4 1e+00 0.1022759 0.02454381
```

```
## 5 5e+00 0.1078574 0.03235309
## 6 1e+01 0.1087833 0.03199823
```

run the model evaluation.

```
svm.best.rbf = svm.rbf.tune$best.model
svm.tune.rbf.pred = predict(svm.best.rbf, newdata=test.data[, -12])
confusionMatrix(svm.tune.rbf.pred, test.data$quality_class)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction   0   1
##          0 389  46
##          1   2  24
##
##                Accuracy : 0.8959
##                  95% CI : (0.8643, 0.9222)
##     No Information Rate : 0.8482
##     P-Value [Acc > NIR] : 0.001802
##
##                   Kappa : 0.4552
##
##  Mcnemar's Test P-Value : 5.417e-10
##
##             Sensitivity : 0.9949
##             Specificity : 0.3429
##          Pos Pred Value : 0.8943
##          Neg Pred Value : 0.9231
##              Prevalence : 0.8482
##          Detection Rate : 0.8438
##    Detection Prevalence : 0.9436
##       Balanced Accuracy : 0.6689
##
##        'Positive' Class : 0
##
```

Observing all the models we built today (SVM with linear, polynomial, and rbf kernels) and in Practical 8 (rule-based, kNN, and Naïve Bayes), which model achieved the best performance?

Sigmoid Kernel in SVM

Challenge 1. We talked about three kernels in this practical. Another commonly used is sigmoid kernel. Can you try to run the above procedure (finding an optimal model and running the model evaluation) for sigmoid kernel? Please note that for the sigmoid kernel, you should optimize values for gamma and coef0 parameters.

Parameter Tuning for Sigmoid Kernel:

```
set.seed(999)
svm.sigmoid.tune = tune.svm(quality_class~., data=train.data,
               kernel="sigmoid",
```

```r
               gamma=c(0.001, 0.01, 0.1, 1, 10),
               coef0=c(-1, 0, 1))
summary(svm.sigmoid.tune)
```

```
##
## Parameter tuning of 'svm':
##
## - sampling method: 10-fold cross validation
##
## - best parameters:
##  gamma coef0
##    0.1    -1
##
## - best performance: 0.1311267
##
## - Detailed performance results:
##    gamma coef0     error dispersion
## 1  1e-03    -1 0.1366822 0.02907684
## 2  1e-02    -1 0.1366822 0.02907684
## 3  1e-01    -1 0.1311267 0.01802472
## 4  1e+00    -1 0.1897975 0.01798997
## 5  1e+01    -1 0.1971357 0.03337466
## 6  1e-03     0 0.1366822 0.02907684
## 7  1e-02     0 0.1366822 0.02907684
## 8  1e-01     0 0.1655936 0.03344130
## 9  1e+00     0 0.1925320 0.02335392
## 10 1e+01     0 0.2000000 0.02748888
## 11 1e-03     1 0.1366822 0.02907684
## 12 1e-02     1 0.1366822 0.02907684
## 13 1e-01     1 0.2092073 0.03000143
## 14 1e+00     1 0.1906542 0.03137866
## 15 1e+01     1 0.2045864 0.03317739
```

Train and Evaluate the Optimal Sigmoid Kernel SVM Model:

```r
svm.best.sigmoid = svm.sigmoid.tune$best.model
svm.tune.sigmoid.pred = predict(svm.best.sigmoid, newdata=test.data[, -12])
confusionMatrix(svm.tune.sigmoid.pred, test.data$quality_class)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction   0   1
##          0 360  43
##          1  31  27
##
##                Accuracy : 0.8395
##                  95% CI : (0.8027, 0.8718)
##     No Information Rate : 0.8482
##     P-Value [Acc > NIR] : 0.7237
##
##                   Kappa : 0.3296
```

```
##
##  Mcnemar's Test P-Value : 0.2010
##
##             Sensitivity : 0.9207
##             Specificity : 0.3857
##          Pos Pred Value : 0.8933
##          Neg Pred Value : 0.4655
##              Prevalence : 0.8482
##          Detection Rate : 0.7809
##    Detection Prevalence : 0.8742
##       Balanced Accuracy : 0.6532
##
##        'Positive' Class : 0
##
```

Main Concepts: The Sigmoid kernel is often used in Support Vector Machines (SVM) as one of the alternatives to linear, polynomial, or RBF kernels. Mathematically, the sigmoid kernel is defined as $K(x, y) = \tanh(\gamma\langle x, y\rangle + c_0)$, where γ is the scale parameter and c_0 is the coefficient coef0.

Technical Considerations:

- **Gamma (γ)**: The scale parameter; a small value will yield a more flexible decision boundary, while a large value will yield a more rigid decision boundary.
- **Coef0 (c_0)**: The independent coefficient; this impacts how the input data is scaled before the sigmoid function is applied.

Significance: Choosing optimal parameters for the Sigmoid kernel can substantially influence model performance. It's essential to tune both `gamma` and `coef0` to avoid overfitting or underfitting.

Applications: Like other kernels, the sigmoid kernel can be applied in various classification tasks. However, it's worth noting that sigmoid kernels have lost popularity compared to RBF kernels due to their behavior and output range.