Advanced Classification: Classifiers and Support Vector Machine

Support vector classifier

The e1071 library contains implementations for a number of statistical learning methods. In particular, the svm() function can be used to fit a support vector classifier when the argument kernel="linear" is used. A cost argument allows us to specify the cost of a violation to the margin. When the cost argument is small, then the margins will be wide and many support vectors will be on the margin or will violate the margin. When the cost argument is large, then the margins will be narrow and there will be few support vectors on the margin or violating the margin.

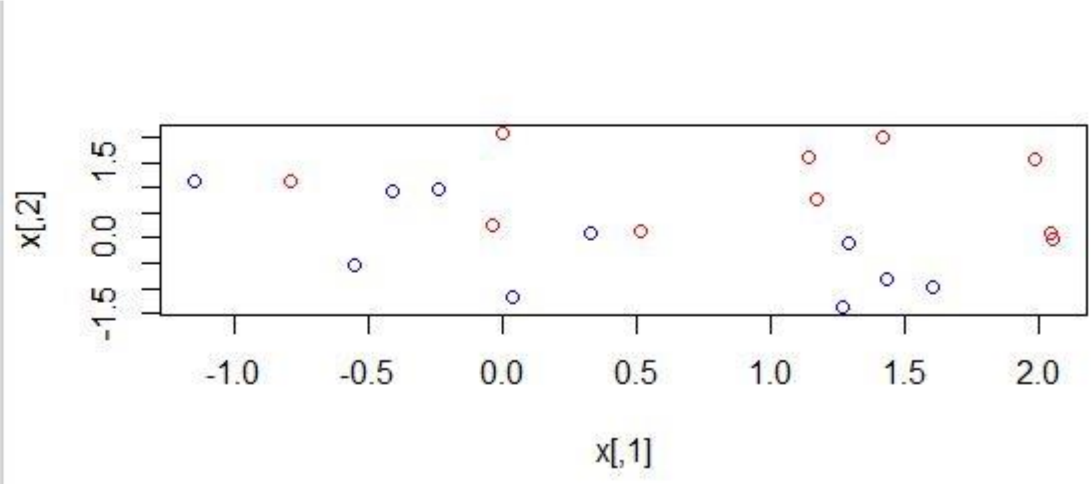
In this project we assume that students will implement independently all necessary steps like setting the working directory and connecting the library, which were explained before in HW1 and HW2.

Step 1 Set variable k equal to the last 4 digits of your student number. Then initialize the random number generator as set.seed(k). This is an important requirement which makes all project results different for all students with very high level of probability. Do not re-set this value for other steps of this work.

```
> set.seed(6388)
```

Step 2 (1 mark) We begin by generating the observations, which belong to two classes, and checking whether the classes are linearly separable. Use commands matrix to generate two sets of data.
Plot these data using command plot. Demonstrate this plot and answer to the questions if these two sets are separable.

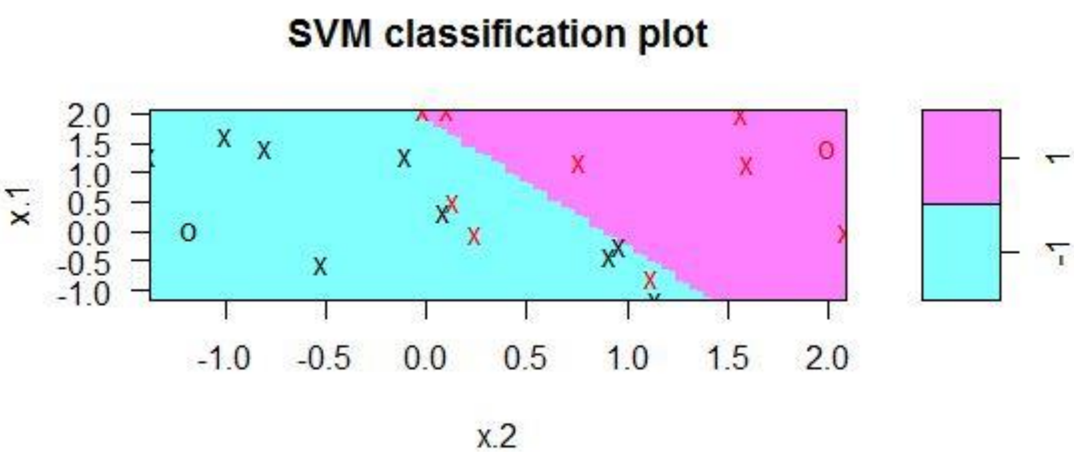
```
> x <- matrix(rnorm(20*2), ncol=2)
> y <- c(rep(-1,10), rep(1,10))
> x[y==1,]=x[y==1,] + 1
> plot(x, col=(3-y))
```



Step 3 (1 mark) Fit the support vector classifier for cost function value 0.1. Note that in order for the svm() function to perform classification (as opposed to SVM-based regression), we must encode the response as a factor variable. Provide summary of the svmfit. Plot the support vector classifier obtained.

The important point is that before following the instructions from the text book, or use the R commands from the website, you have to install package e1071.

```
> library(e1071)
> dat <- data.frame(x=x, y=as.factor(y))
> svm.fit<- svm(y ~., data=dat, kernel = 'linear', cost=0.1, scale=FALSE)
> plot(svm.fit, dat)
```



Step 4 (1 mark) Determine their identities of the support vectors.

```
> summary(svm.fit)

Call:
svm(formula = y ~ ., data = dat, kernel = "linear", cost = 0.1,
    scale = FALSE)
```

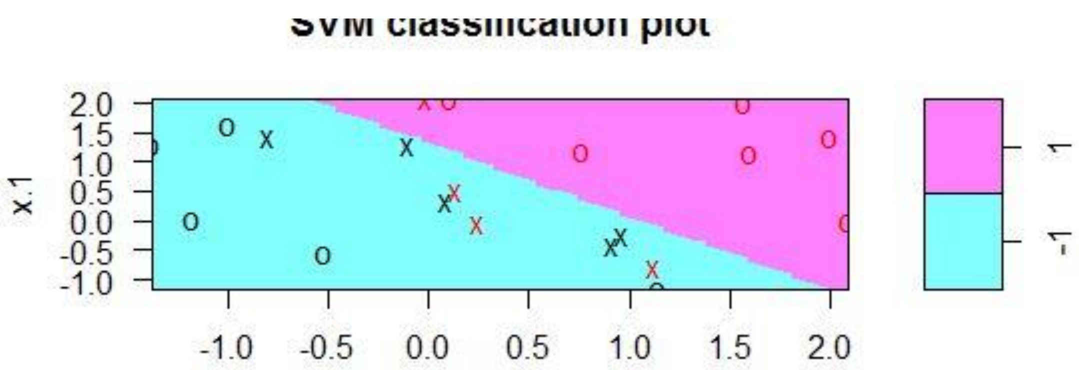
```
Parameters:
   SVM-Type:  C-classification
 SVM-Kernel:  linear
       cost:  0.1
      gamma:  0.5

Number of Support Vectors:  18

 ( 9 9 )


Number of Classes:  2

Levels:
 -1 1
```

```
> svm.fit$index
 [1]  1  2  3  4  5  6  8  9 10 11 12 13 14 15 16 17 18 19
```

The summary lets us know there were 18 support vectors which are {1 2 3 4 5 6 8 9 10 11 12 13 14 15 16 17 18 19},
9 in the first class and 9 in the second

Step 5 (1 mark) Increase number of cost parameter to 10. Check and identify the support vectors, wrote how they number changed.

```
> dat <- data.frame(x=x, y=as.factor(y))
> svm.fit1 <- svm(y ~., data=dat, kernel='linear', cost=10, scale=FALSE)
> plot(svm.fit1, dat)
```




```
> summary(svm.fit1)

Call:
svm(formula = y ~ ., data = dat, kernel = "linear",
cost = 10,
    scale = FALSE)


Parameters:
   SVM-Type:  C-classification
 SVM-Kernel:  linear
```

```
       cost:  10
      gamma:  0.5

Number of Support Vectors:  9

 ( 5 4 )


Number of Classes:  2

Levels:
 -1 1



> svm.fit1$index
[1]  1  2  3  4  9 11 16 17 19

>
```

The summary lets us know there were 18 support vectors which are {1 2 3 4 9 11 16 17 19},
5 in the first class and 4 in the second

Step 6 (1 mark) Compare SVMs with a linear kernel, using a range of values of the cost parameter. Print and interpret summary.

```
 > set.seed(6388)
>
> tune.out <- tune(svm, y ~., data=dat, kernel='linear',
+                  ranges=list(cost=c(0.001,0.01,0.1,1,5,10,100)))
> summary(tune.out)

Parameter tuning of ‘svm’:

- sampling method: 10-fold cross validation

- best parameters:
 cost
    1

- best performance: 0.25

- Detailed performance results:
   cost error dispersion
1 1e-03  0.70  0.4216370
2 1e-02  0.70  0.4216370
3 1e-01  0.30  0.3496029
4 1e+00  0.25  0.3535534
5 5e+00  0.25  0.3535534
6 1e+01  0.25  0.3535534
7 1e+02  0.25  0.3535534
```

#The best cost is 1 for the output.
best performance: 0.25

Step 7 (1 mark) The tune() function stores the best model obtained; accessed it using the command. Print summary.

```
> bestmod = tune.out$best.model
```

```
> summary(bestmod)

Call:
best.tune(method = svm, train.x = y ~ ., data = dat, ranges = list(cost = c(0.001,
    0.01, 0.1, 1, 5, 10, 100)), kernel = "linear")


Parameters:
   SVM-Type:  C-classification
 SVM-Kernel:  linear
       cost:  1
      gamma:  0.5

Number of Support Vectors:  13

 ( 7 6 )


Number of Classes:  2

Levels:
 -1 1
```

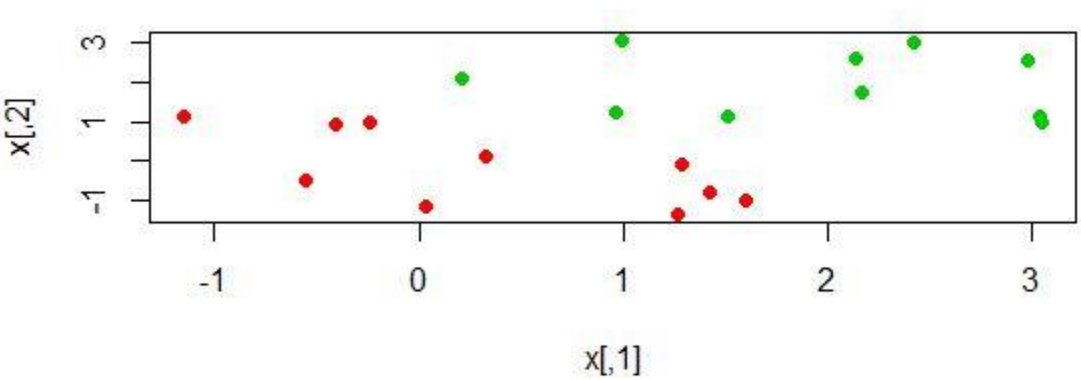
Here we see that cost= 1 results in the lowest cross-validation error rate.

Step 8 (2 marks) Generate the test data set and predict the class labels of these test observations.

```
> xtest=matrix(rnorm(20*2), ncol=2)
> ytest=sample(c(-1,1), 20, rep=TRUE)
> xtest [ ytest ==1 ,]= xtest [ ytest ==1 ,] + 1
> testdat=data.frame(x=xtest, y=as.factor(ytest))
> yhat <- predict(tune.out$best.model, testdat)
> #install.packages("caret")
> library(caret)
> confusionMatrix(yhat, testdat$y)
Confusion Matrix and Statistics

          Reference
Prediction -1  1
        -1  5  2
        1   3 10

               Accuracy : 0.75
                 95% CI : (0.509, 0.9134)
    No Information Rate : 0.6
    P-Value [Acc > NIR] : 0.1256

                  Kappa : 0.4681
 Mcnemar's Test P-Value : 1.0000

            Sensitivity : 0.6250
            Specificity : 0.8333
         Pos Pred Value : 0.7143
         Neg Pred Value : 0.7692
             Prevalence : 0.4000
         Detection Rate : 0.2500
   Detection Prevalence : 0.3500
      Balanced Accuracy : 0.7292

       'Positive' Class : -1
```

Step 9 (2 marks) Now consider a situation in which the two classes are linearly separable. Then find a separating hyperplane using the svm() function. Separate the two classes in our simulated data so that they are linearly separable.

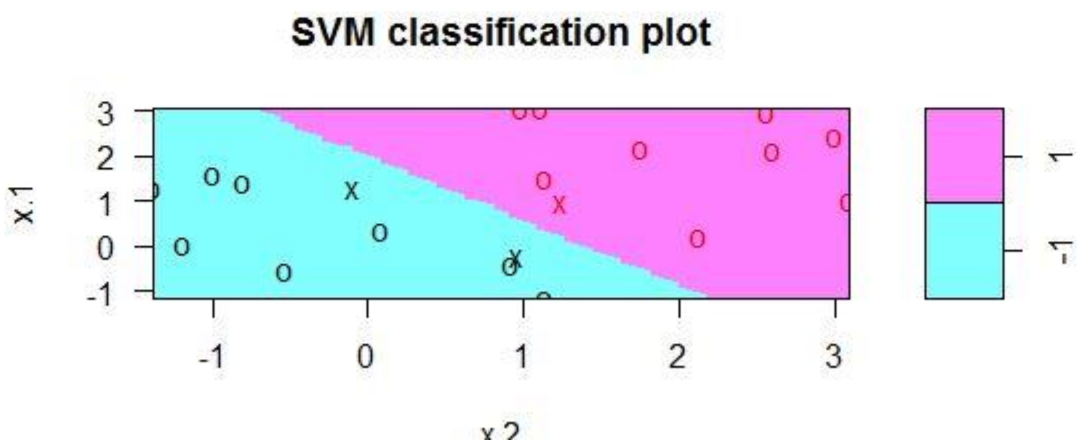
```
> x[y==1 ,]= x[y==1 ,]+01
> plot(x, col =(y+5) /2, pch =19)
```



Step 10 (2 marks) Fit the support vector classifier and plot the resulting hyperplane, using a very large value of cost so that no observations are misclassified.

```
> dat=data.frame(x=x,y=as.factor (y))
> svmfit =svm(y~ ., data=dat , kernel ="linear", cost =1e5)
> summary (svmfit)

Call:
svm(formula = y ~ ., data = dat, kernel = "linear", cost = 1e+05)


Parameters:
   SVM-Type:  C-classification
 SVM-Kernel:  linear
       cost:  1e+05
      gamma:  0.5

Number of Support Vectors:  3

 ( 2 1 )


Number of Classes:  2

Levels:
 -1 1


#No of supporting vectors is 3
```

```
> plot(svmfit , dat)
```



Step 11 (1 marks) Answer the multiple choice question:

1. Are the support vectors outside of the margin?
2. Are the support vectors on the boarder of the margin?
3. Are the support vectors within the margin?

#1. Are the support vectors outside of the margin? Yes
#2. Are the support vectors on the boarder of the margin? Yes
#3. Are the support vectors within the margin? No

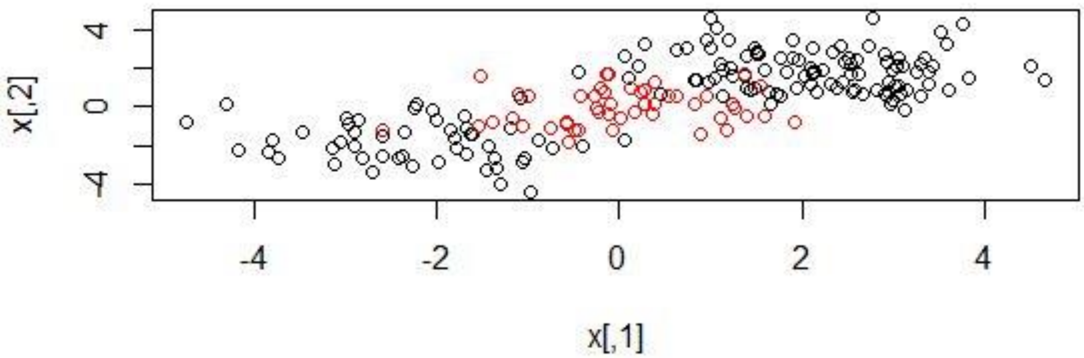
Support vector machine (Refer Section 9.6 from the text book) 5 marks

In order to fit an SVM using a non-linear kernel, use the svm() function. Use a different value of the parameter kernel. To fit an SVM with a polynomial kernel use kernel="polynomial", and to fit an SVM with a radial kernel use kernel="radial". In the former case we also use the degree argument to specify a degree for the polynomial kernel (this is *d* in (9.22)), and in the latter case we use gamma to specify a value of γ for the radial basis kernel (9.24).

Step 1 (1 marks) Generate some data with a non-linear class boundary and plot them.

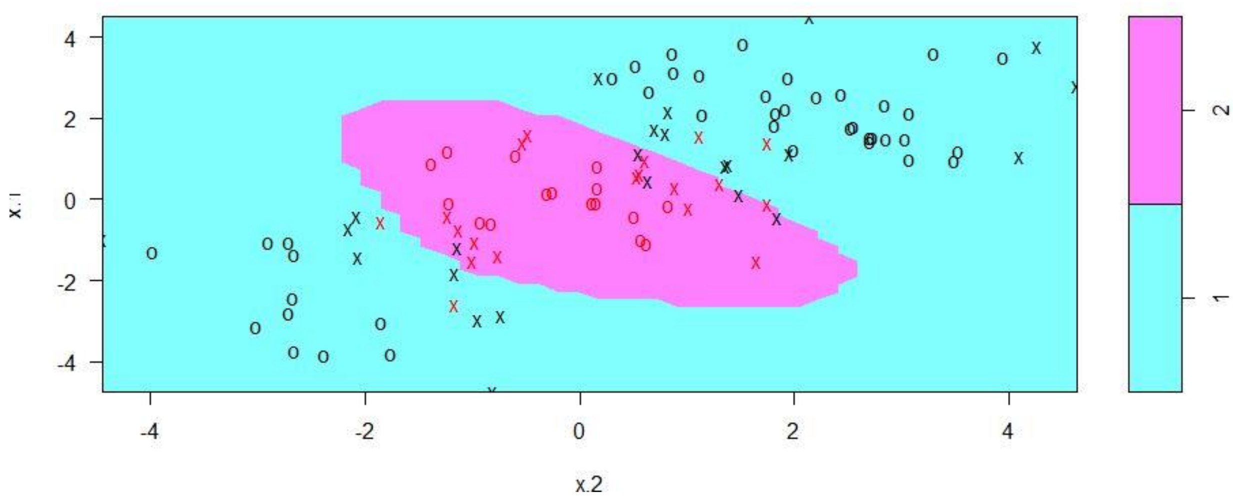
```
> plot(svmfit , dat)
> set.seed (6388)
> x=matrix (rnorm (200*2) , ncol =2)
> x[1:100 ,]=x[1:100 ,]+2
> x[101:150 ,]= x[101:150 ,] -2
> y=c(rep (1 ,150) ,rep (2 ,50) )
```

```
> dat=data.frame(x=x,y=as.factor (y))
> plot(x, col=y)
```



Step 2 (1 marks) Fit the training data using the svm() function with a radial kernel and γ = 1.

```
> train=sample (200 ,100)
> svmfit =svm(y~., data=dat [train, ], kernel ="radial", gamma =1,cost =1)
> plot(svmfit , dat[train ,])
```



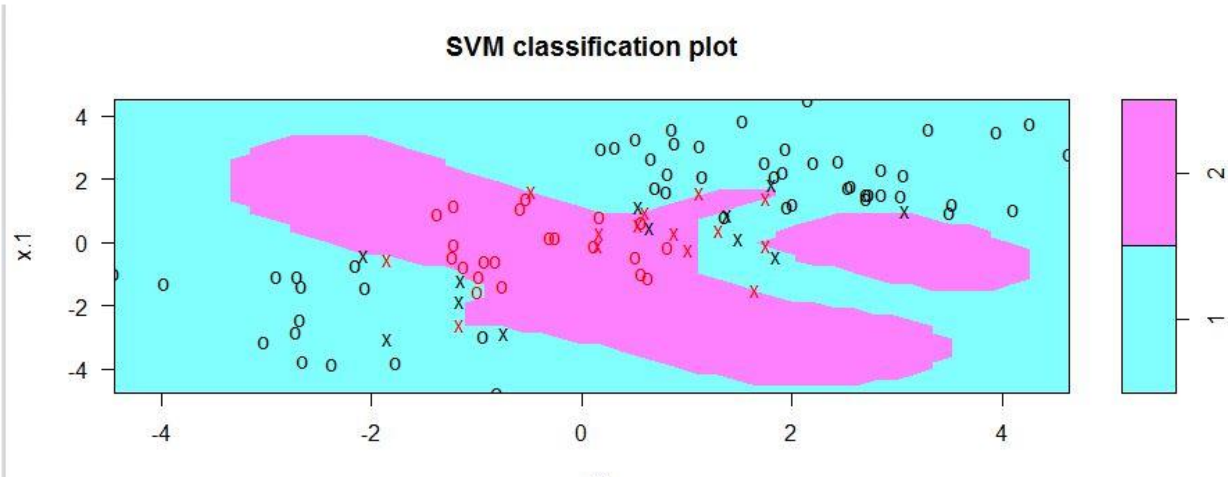
Step 3 (1 marks) Print summary. What can you tell about of the error? Re-fit the SVM classification with higher cost. Print summary and plot results. What are your major concern about these results?

```
> summary(svmfit)

Call:
svm(formula = y ~ ., data = dat[train, ], kernel = "radial",
    gamma = 1, cost = 1)


Parameters:
   SVM-Type:  C-classification
```

```
 SVM-Kernel:  radial
       cost:  1
      gamma:  1

Number of Support Vectors:  43

 ( 19 24 )


Number of Classes:  2

Levels:
 1 2
```

#No of support vectors is 43.

```
> svmfit =svm(y~., data=dat [train ,], kernel ="radial",gamma =1, cost=1e5)
> plot(svmfit ,dat [train ,])
```



#We can see from the figure that there are a fair number of training errors in this SVM fit. If we increase the value of cost, we can reduce the number of training errors. However, this comes at the price of a more irregular decision boundary that seems to be at risk of overfitting the data.

Step 4 (1 marks) Perform cross-validation using tune() to select the best choice of *γ* and cost for an SVM with a radial kernel.

```
> set.seed (6388)
> tune.out=tune(svm , y~., data=dat[train ,], kernel
="radial", ranges =list(cost=c(0.1 ,1 ,10 ,100 ,1000),
gamma=c(0.5,1,2,3,4) ))
> summary(tune.out)

Parameter tuning of ‘svm’:

- sampling method: 10-fold cross validation

- best parameters:
 cost gamma
```

```
     1   0.5

- best performance: 0.1

- Detailed performance results:
    cost gamma error dispersion
1  1e-01   0.5  0.15 0.08498366
2  1e+00   0.5  0.10 0.04714045
3  1e+01   0.5  0.12 0.06324555
4  1e+02   0.5  0.11 0.05676462
5  1e+03   0.5  0.16 0.08432740
6  1e-01   1.0  0.16 0.09660918
7  1e+00   1.0  0.11 0.05676462
8  1e+01   1.0  0.12 0.06324555
9  1e+02   1.0  0.16 0.08432740
10 1e+03   1.0  0.21 0.11972190
11 1e-01   2.0  0.28 0.11352924
12 1e+00   2.0  0.13 0.08232726
13 1e+01   2.0  0.13 0.06749486
14 1e+02   2.0  0.19 0.09944289
15 1e+03   2.0  0.19 0.11005049
16 1e-01   3.0  0.29 0.11972190
17 1e+00   3.0  0.13 0.08232726
18 1e+01   3.0  0.14 0.06992059
19 1e+02   3.0  0.17 0.08232726
20 1e+03   3.0  0.22 0.09189366
21 1e-01   4.0  0.35 0.14337209
22 1e+00   4.0  0.13 0.08232726
23 1e+01   4.0  0.15 0.08498366
24 1e+02   4.0  0.17 0.08232726
25 1e+03   4.0  0.23 0.08232726


>
```

Therefore, the best choice of parameters involves cost=1 and gamma=0.5

Step 5 (1 marks) Interpret results: what si the optimal values of cost and *γ* and what is the lowastt percent of misclassified objects?

```
> yhat <- predict(tune.out$best.model, dat[-train,])
> confusionMatrix(yhat, dat[-train, 'y'])
Confusion Matrix and Statistics

          Reference
Prediction  1  2
         1 75  0
         2 10 15

               Accuracy : 0.9
                 95% CI : (0.8238, 0.951)
    No Information Rate : 0.85
    P-Value [Acc > NIR] : 0.099447

                  Kappa : 0.6923
 Mcnemar's Test P-Value : 0.004427

            Sensitivity : 0.8824
            Specificity : 1.0000
         Pos Pred Value : 1.0000
         Neg Pred Value : 0.6000
```

```
            Prevalence : 0.8500
        Detection Rate : 0.7500
  Detection Prevalence : 0.7500
     Balanced Accuracy : 0.9412

      'Positive' Class : 1


>
```

The optimal values of cost is 1 and gamma=0.5. Percentage of misclassified objects is 10 percent.

Decision trees for classification (Refer Section 8.3 from the text book) 7 marks

Step 1 The ISLR and tree libraries are used to construct classification and regression trees. First use classification trees to analyze the Carseats data set. In these data, Sales is a continuous variable, and so we begin by recoding it as a binary variable. Use the ifelse() function to create a variable, called High, which takes on a value of Yes if the Sales variable exceeds 8, and takes on a value of No otherwise. Do not forget to install relevant packages. The description of ISLR package including Carseats (which contains Sales) data set is available on the course website (R language page).

```
> #install.packages('tree', dependencies=TRUE)
> library (tree)
Warning message:
package 'tree' was built under R version 3.4.3
> library (ISLR)

>
```

Step 2 (1 marks) Use the data.frame() function to merge High with the rest of the Carseats data. Use the tree() function to fit a classification tree in order to predict High using all variables but Sales. The syntax of the tree() function is quite similar to that of the lm() function. Use summary() function lists the variables that are used as internal nodes in the tree, the number of terminal nodes, and the (training) error rate. What is the training error rate?

```
> attach (Carseats )
The following objects are masked from Carseats (pos =
3):

    Advertising, Age, CompPrice, Education, Income,
    Population, Price, Sales, ShelveLoc, Urban, US

> View(Carseats)
> High=ifelse (Sales <=8," No"," Yes ")
> Carseats =data.frame(Carseats ,High)
> tree.carseats =tree(High~.-Sales ,Carseats )
> summary (tree.carseats )

Classification tree:
tree(formula = High ~ . - Sales, data = Carseats)
Variables actually used in tree construction:
```

```
[1] "ShelveLoc"   "Price"       "Income"
"CompPrice"
[5] "Population"  "Advertising" "Age"         "US"
Number of terminal nodes:  27
Residual mean deviance:  0.4575 = 170.7 / 373
Misclassification error rate: 0.09 = 36 / 400
```

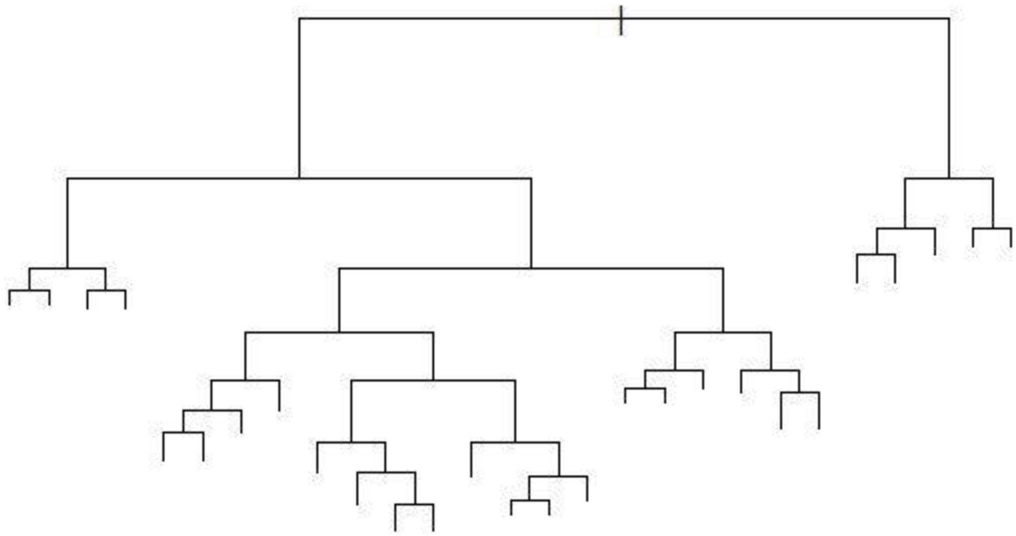
```
>
```

	Sales	CompPrice	Income	Advertising	Population	Price	ShelveLoc	Age	Education	Urban	US	High	High.1
1	9.50	138	73	11	276	120	Bad	42	17	Yes	Yes	Yes	Yes
2	11.22	111	48	16	260	83	Good	65	10	Yes	Yes	Yes	Yes
3	10.06	113	35	10	269	80	Medium	59	12	Yes	Yes	Yes	Yes
4	7.40	117	100	4	466	97	Medium	55	14	Yes	Yes	No	No
5	4.15	141	64	3	340	128	Bad	38	13	Yes	No	No	No
6	10.81	124	113	13	501	72	Bad	78	16	No	Yes	Yes	Yes
7	6.63	115	105	0	45	108	Medium	71	15	Yes	No	No	No
8	11.85	136	81	15	425	120	Good	67	10	Yes	Yes	Yes	Yes
9	6.54	132	110	0	108	124	Medium	76	10	No	No	No	No
10	4.69	132	113	0	131	124	Medium	76	17	No	Yes	No	No
11	9.01	121	78	9	150	100	Bad	26	10	No	Yes	Yes	Yes
12	11.96	117	94	4	503	94	Good	50	13	Yes	Yes	Yes	Yes

Mis classication rate is .09

Step 3 (1 marks) Plot and text the car seat tree. Provide in your answer the tree **without** texts.

```
> plot(tree.carseats )
```



Step 4 (1 marks) Type the name of the tree object, and analyze the R prints output corresponding to each branch of the tree. R displays the split criterion (e.g. Price<92.5), the number of observations in that branch, the deviance, the overall prediction for the branch (Yes or No), and the fraction of observations in that branch that take on values of Yes and No. Branches that lead to terminal nodes are indicated using asterisks.

```
> tree.carseats
node), split, n, deviance, yval, (yprob)
      * denotes terminal node

  1) root 400 541.500  No ( 0.59000 0.41000 )
    2) ShelveLoc: Bad,Medium 315 390.600  No ( 0.68889 0.31111 )
      4) Price < 92.5 46  56.530  Yes  ( 0.30435 0.69565 )
        8) Income < 57 10  12.220  No ( 0.70000 0.30000 )
         16) CompPrice < 110.5 5   0.000  No ( 1.00000 0.00000 ) *
         17) CompPrice > 110.5 5   6.730  Yes  ( 0.40000 0.60000 ) *
        9) Income > 57 36  35.470  Yes  ( 0.19444 0.80556 )
         18) Population < 207.5 16  21.170  Yes  ( 0.37500 0.62500 ) *
         19) Population > 207.5 20   7.941  Yes  ( 0.05000 0.95000 ) *
      5) Price > 92.5 269 299.800  No ( 0.75465 0.24535 )
       10) Advertising < 13.5 224 213.200  No ( 0.81696 0.18304 )
         20) CompPrice < 124.5 96  44.890  No ( 0.93750 0.06250 )
           40) Price < 106.5 38  33.150  No ( 0.84211 0.15789 )
             80) Population < 177 12  16.300  No ( 0.58333 0.41667 )
              160) Income < 60.5 6   0.000  No ( 1.00000 0.00000 ) *
              161) Income > 60.5 6   5.407  Yes  ( 0.16667 0.83333 ) *
             81) Population > 177 26   8.477  No ( 0.96154 0.03846 ) *
           41) Price > 106.5 58   0.000  No ( 1.00000 0.00000 ) *
         21) CompPrice > 124.5 128 150.200  No ( 0.72656 0.27344 )
           42) Price < 122.5 51  70.680  Yes  ( 0.49020 0.50980 )
             84) ShelveLoc: Bad 11   6.702  No ( 0.90909 0.09091 ) *
             85) ShelveLoc: Medium 40  52.930  Yes  ( 0.37500 0.62500 )
              170) Price < 109.5 16   7.481  Yes  ( 0.06250 0.93750 ) *
              171) Price > 109.5 24  32.600  No ( 0.58333 0.41667 )
                342) Age < 49.5 13  16.050  Yes  ( 0.30769 0.69231 ) *
                343) Age > 49.5 11   6.702  No ( 0.90909 0.09091 ) *
```

```
          43) Price > 122.5 77  55.540  No ( 0.88312 0.11688 )
            86) CompPrice < 147.5 58  17.400  No ( 0.96552 0.03448 ) *
            87) CompPrice > 147.5 19  25.010  No ( 0.63158 0.36842 )
             174) Price < 147 12  16.300  Yes  ( 0.41667 0.58333 )
               348) CompPrice < 152.5 7   5.742  Yes  ( 0.14286 0.85714 ) *
               349) CompPrice > 152.5 5   5.004  No ( 0.80000 0.20000 ) *
             175) Price > 147 7   0.000  No ( 1.00000 0.00000 ) *
      11) Advertising > 13.5 45  61.830  Yes  ( 0.44444 0.55556 )
        22) Age < 54.5 25  25.020  Yes  ( 0.20000 0.80000 )
          44) CompPrice < 130.5 14  18.250  Yes  ( 0.35714 0.64286 )
            88) Income < 100 9  12.370  No ( 0.55556 0.44444 ) *
            89) Income > 100 5   0.000  Yes  ( 0.00000 1.00000 ) *
          45) CompPrice > 130.5 11   0.000  Yes  ( 0.00000 1.00000 ) *
        23) Age > 54.5 20  22.490  No ( 0.75000 0.25000 )
          46) CompPrice < 122.5 10   0.000  No ( 1.00000 0.00000 ) *
          47) CompPrice > 122.5 10  13.860  No ( 0.50000 0.50000 )
            94) Price < 125 5   0.000  Yes  ( 0.00000 1.00000 ) *
            95) Price > 125 5   0.000  No ( 1.00000 0.00000 ) *
   3) ShelveLoc: Good 85  90.330  Yes  ( 0.22353 0.77647 )
     6) Price < 135 68  49.260  Yes  ( 0.11765 0.88235 )
      12) US: No 17  22.070  Yes  ( 0.35294 0.64706 )
        24) Price < 109 8   0.000  Yes  ( 0.00000 1.00000 ) *
        25) Price > 109 9  11.460  No ( 0.66667 0.33333 ) *
      13) US: Yes 51  16.880  Yes  ( 0.03922 0.96078 ) *
     7) Price > 135 17  22.070  No ( 0.64706 0.35294 )
      14) Income < 46 6   0.000  No ( 1.00000 0.00000 ) *
      15) Income > 46 11  15.160  Yes  ( 0.45455 0.54545 ) *
```

Step 5 (1 marks) Evaluate the performance of a classification tree on these data and the training error. Split the observations into a training set (200 records) and a test set, build the tree using the training set, and evaluate its performance on the test data. The predict() function can be used for this purpose. In the case of a classification tree, the argument type="class" instructs R to return the actual class prediction.

```
> set.seed (6388)
> train=sample (1: nrow(Carseats), 200)
> Carseats.test=Carseats [-train ,]
> High.test=High[-train ]
> tree.carseats =tree(High~.-Sales ,Carseats ,subset =train )
> tree.pred=predict (tree.carseats ,Carseats.test ,type ="class")
> table(tree.pred ,High.test)
         High.test
tree.pred  No  Yes
     No    95   26
     Yes   26   53
```

#This approach leads to correct predictions for around 74% of the locations in the test data set.
#(26+26) /200 = 0.26 , 26% is the error date.

Step 6 (1 marks) Consider whether pruning the tree might lead to improved results. The function cv.tree() performs cross-validation in order to cv.tree() determine the optimal level of tree complexity; cost complexity pruning is used in order to select a sequence of trees for consideration. Use the argument FUN=prune.misclass in order to indicate that we want the classification error rate to guide the cross-validation and pruning process, rather than the default for the cv.tree() function, which is deviance. The cv.tree() function reports the number of terminal nodes of each tree considered (size) as well as the corresponding error rate and the value of the cost-complexity parameter used (k, which corresponds to α in (8.4)).

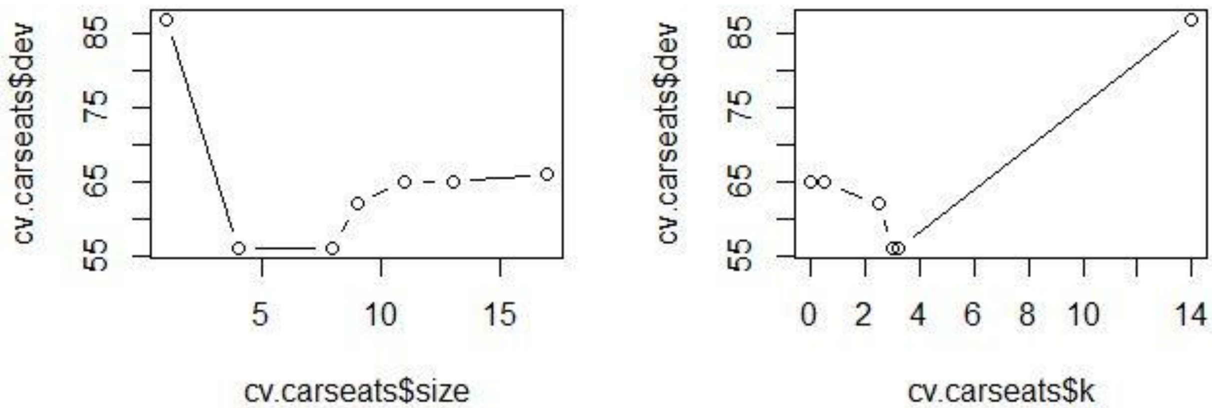
What is the optimal pruning (optimal number of leaves)?

```
> set.seed (6388)
> cv.carseats =cv.tree(tree.carseats ,FUN=prune.misclass )
> names(cv.carseats)
[1] "size"   "dev"    "k"      "method"
> cv.carseats
$size
[1] 17 13 11  9  8  4  1

$dev
[1] 66 65 65 62 56 56 87

$k
[1]  -Inf  0.00  0.50  2.50  3.00  3.25 14.00

$method
[1] "misclass"

attr(,"class")
[1] "prune"         "tree.sequence"
```

Note that, despite the name, dev corresponds to the cross-validation error rate in this instance. The tree with 4 terminal nodes results in the lowest cross-validation error rate, with 56 cross-validation errors.

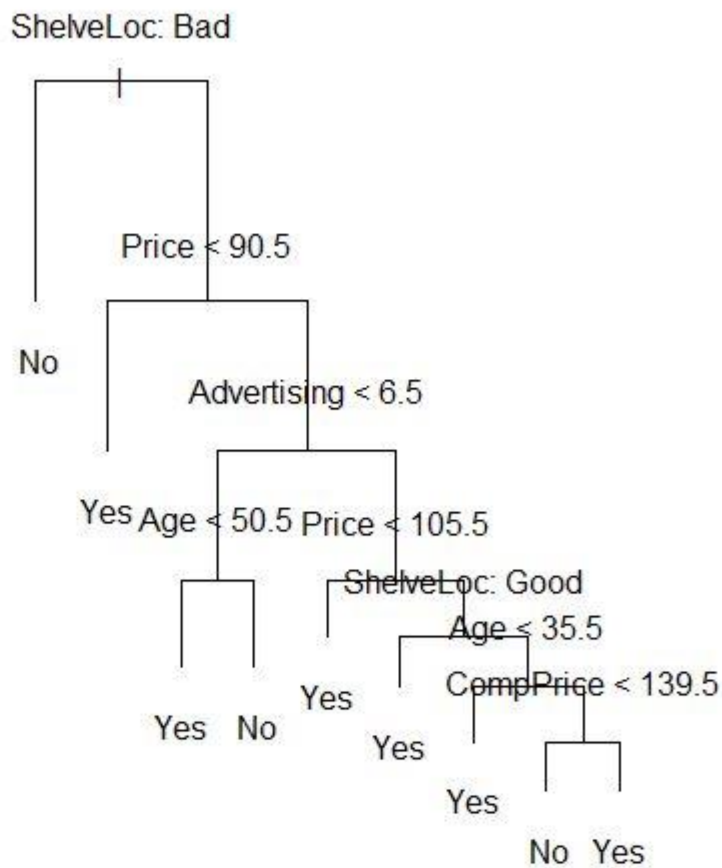
Step 7 (1 marks) Plot the error rate as a function of both size and k.

```
> par(mfrow =c(1,2))
> plot(cv.carseats$size ,cv.carseats$dev ,type="b")
> plot(cv.carseats$k ,cv.carseats$dev ,type="b")

>
```



Step 8 (1 marks) Apply the prune.misclass() function in order to prune the tree to prune. Obtain the nine-node tree. Plot it **with** text (do not care about overlapping!).

```
> prune.carseats =prune.misclass(tree.carseats,best =9)
> plot(prune.carseats)
> text(prune.carseats,pretty=0)
```


ShelveLoc: Bad
Price < 90.5
No
Advertising < 6.5
Yes
Age < 50.5
Price < 105.5
ShelveLoc: Good
Age < 35.5
Yes
No
Yes
CompPrice < 139.5
Yes
Yes
No
Yes