# Context Aware with Different Application

As for the data comes from different application, we can classify them according to the application they belong to. Then we send the data to the learners which are available and efficient.

Before we send the data to the potential learner, we need some training data and test data to label the error rate and the time delay of every learner over the data according to the application.

We have used three kinds of data: handwritten digits, kddcup99 security data and climate data (<https://archive.ics.uci.edu/ml/datasets/Climate+Model+Simulation+Crashes>) . These data are all in the file final report. The summary of the error rate and the time delay are in the following：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Handwritten Digits | Logistic | EM | LDA | Random Forest | SVM |
| Error rate | 13.08% | 10.07% | 14.51% | 6.55% | 0.05% |
| Time(10(-4)s) | 1.81 | 4.32 | 1.31 | 1.78 | 13.36 |

Table 2:

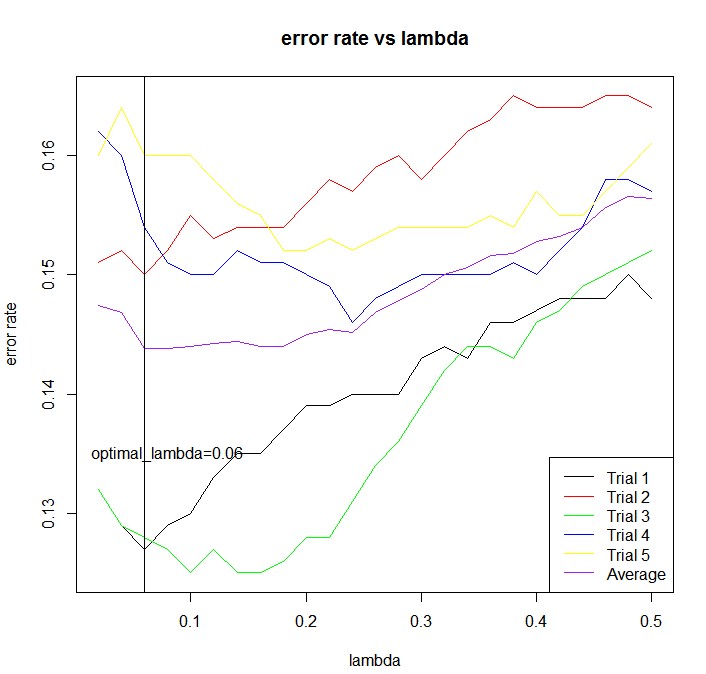
|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Kddcup Security Data | Logistic | | SVM | | Random Forest | Always 1 | | Always 0 |
| Error rate | 0.84% | | 0.42% | | 0.26% | 80.44% | | 19.57% |
| Time(10(-4)s) | 1.33 | | 10 | | 4.7 | - | | - |
| Climate data | | AdaBoost | | RandomForest | | | SVM | |
| Error Rate | | 2.47% | | 4.9% | | | 6.17% | |
| Time(10(-6)s) | | 540 | | 10.8 | | | 5.3 | |

The detailed analysis is in the following:

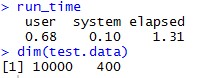
## Handwritten Digits:

### LDA classification of the handwritten digits:

First of all, we should determine an optimal lambda. Here we choose lambda from 0.02to 0.5 and the interval is 0.02. The error rate plots after running the algorithm 5 times with lambda ranging from 0.02 to o.5 are in the following:



From the above we can see that when lambda is 0.04, the error rate is the minimum. Then we use the lambda to train the 5000 training data and test on the testing data.When test the testing data, the total time cost is in the following:



Here user time is how many seconds the computer spent doing your calculations. System time is how much time the operating system spent responding to your program's requests. Elapsed time is the sum of those two, plus whatever "waiting around" your program and/or the OS had to do.

Therefore, I use the elapsed time to denote the time cost. Meaningfully, it is better to use the average time which indicates the time cost of one data stream.

We also record the error rate.

C:\Users\Christina\AppData\Roaming\Tencent\Users\919834852\QQ\WinTemp\RichOle\@]CBL(GEZ4V_[0S@V@P%4O2.jpg

These kind of information are in the following grid:

|  |  |  |
| --- | --- | --- |
|  | Average time of one data stream(s) | The error rate of LDA method |
|  | 1.31xe(-4) | 0.1451 |

### SVM method to classify the handwritten digits

Here we use the library “e1071” of R

R instruction of svm:

svm(x, y = NULL, scale = TRUE, type = NULL, kernel =

"radial", degree = 3, gamma = if (is.vector(x)) 1 else 1 / ncol(x),

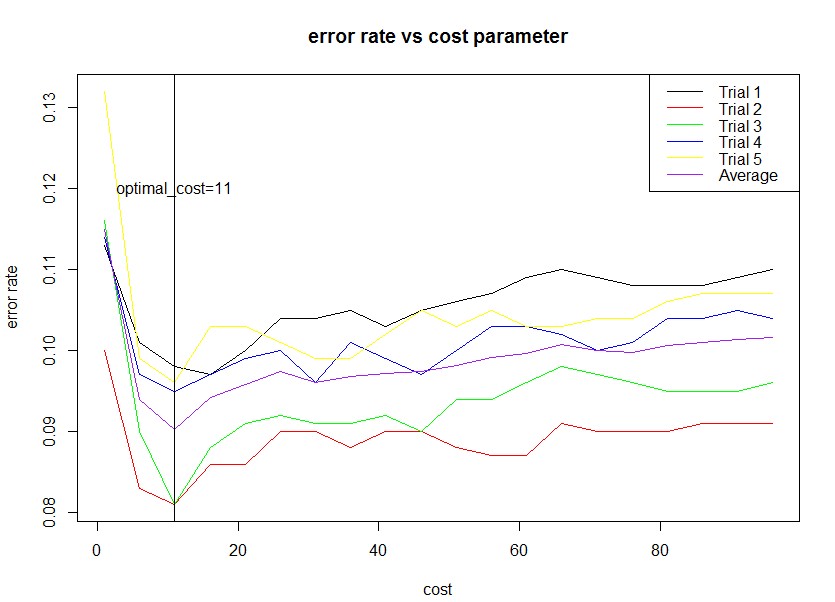
coef0 = 0, cost = 1, nu = 0.5,

class.weights = NULL, cachesize = 40, tolerance = 0.001, epsilon = 0.1,

shrinking = TRUE, cross = 0, probability = FALSE, fitted = TRUE,

..., subset, na.action = na.omit)

After trying some parameters, I find that the cost parameter affect the error rate a lot. Therefore, I decide to run the algorithm 5 times with the training data by ranging the c parameter from 1to 100. The plots of error rate vs cost parameter are in the following and we can find that the optimal cost is 11 with the average corresponding error rate 0.0902



With the optimal cost, I then train the whole training data and test on the testing data. Related codes are in the following:

#deal with the data

training.data=as.data.frame(cbind(training.label,training.data))

training.data[,1]=as.factor(training.data[,1])

test.data=as.data.frame(cbind(test.label,test.data))

test.data[,1]=as.factor(test.data[,1])

colnames(training.data)=c("Y",paste("x.",1:400,sep=""))

colnames(test.data)=c("Y",paste("x.",1:400,sep=""))

svm.model1=svm(training.data$Y~.,data = training.data, cost = 11,method="class")

#calculate the time that used to predict and repot the errorrate

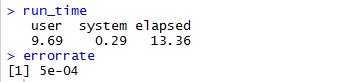
start\_time=proc.time()

svm.pred1=predict(svm.model1,newdata=test.data,type="class")

end\_time=proc.time()

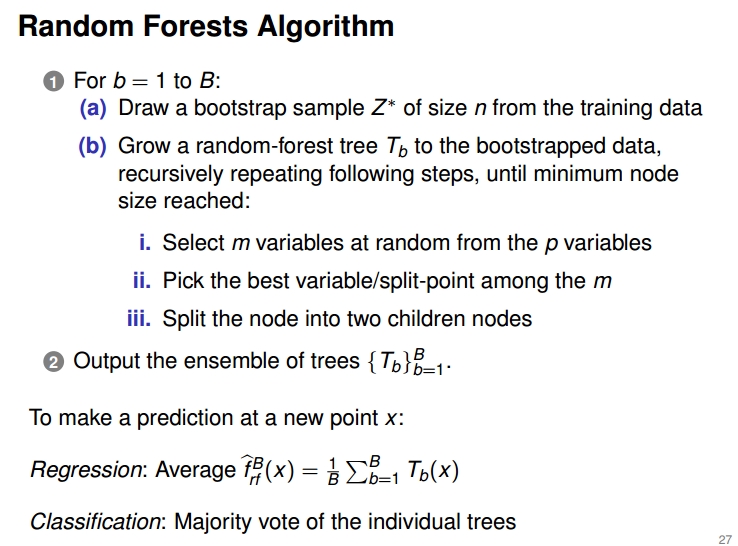
run\_time=end\_time-start\_time

errorrate=sum(svm.pred1!=test.label)/nrow(test.data)



|  |  |
| --- | --- |
| Average time of one data stream(s) | Error rate |
| 1.3e(-3) | 5e-04 |

### Random Forest Method to Classify the Handwritten Digits



**Tune RandomForest for the optimal mtry parameter**

**Description**: Starting with the default value of mtry, search for the optimal value (with respect to Out-of-Bag error estimate) of mtry for randomForest.

**Usage:** tuneRF(x, y, mtryStart, ntreeTry=50, stepFactor=2, improve=0.05,

trace=TRUE, plot=TRUE, doBest=FALSE, ...)

**Arguments**

|  |  |
| --- | --- |
| x | matrix or data frame of predictor variables |
| y | response vector (factor for classification, numeric for regression) |
| mtryStart | starting value of mtry; default is the same as in [randomForest](http://127.0.0.1:46782/help/library/randomForest/help/randomForest) |
| ntreeTry | number of trees used at the tuning step |
| stepFactor | at each iteration, mtry is inflated (or deflated) by this value |
| improve | the (relative) improvement in OOB error must be by this much for the search to continue |
| trace | whether to print the progress of the search |
| plot | whether to plot the OOB error as function of mtry |
| doBest | whether to run a forest using the optimal mtry found |
| ... | options to be given to [randomForest](http://127.0.0.1:46782/help/library/randomForest/help/randomForest) |

> trf=tuneRF(training.data[,-1],training.data$Y,ntreeTry=500,n.var=400)

mtry = 20 OOB error = 7.36%

Searching left ...

mtry = 10 OOB error = 7.78%

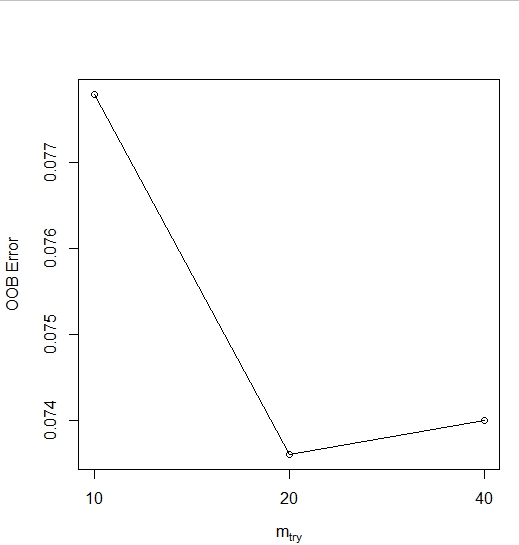
-0.05706522 0.05

Searching right ...

mtry = 40 OOB error = 7.4%

-0.005434783 0.05

**The optimal mtry is 20 which is same with the default mtry**

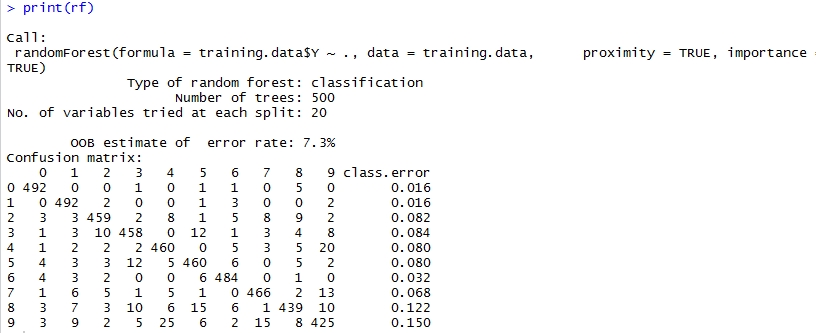


We use the following code to train the whole training data:

rf=randomForest(training.data$Y~.,training.data,proximity = TRUE,importance=TRUE)

plot(rf,type="p",cex=0.5)

print(rf)



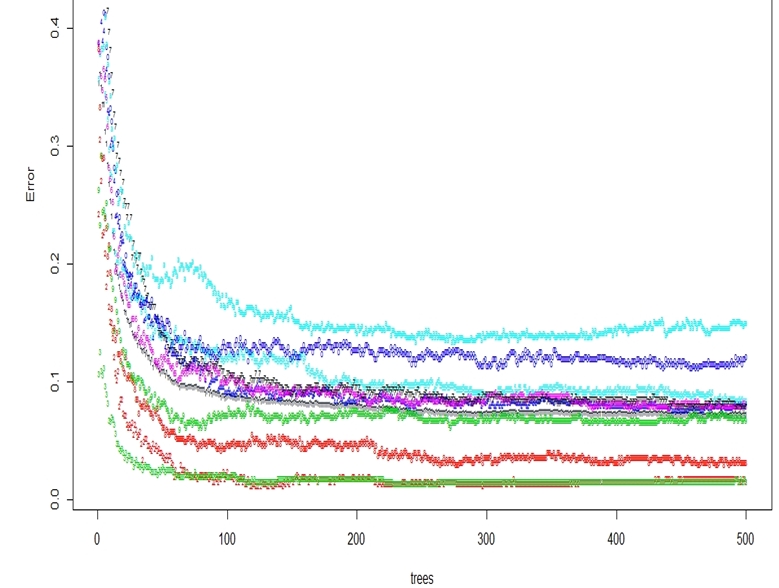


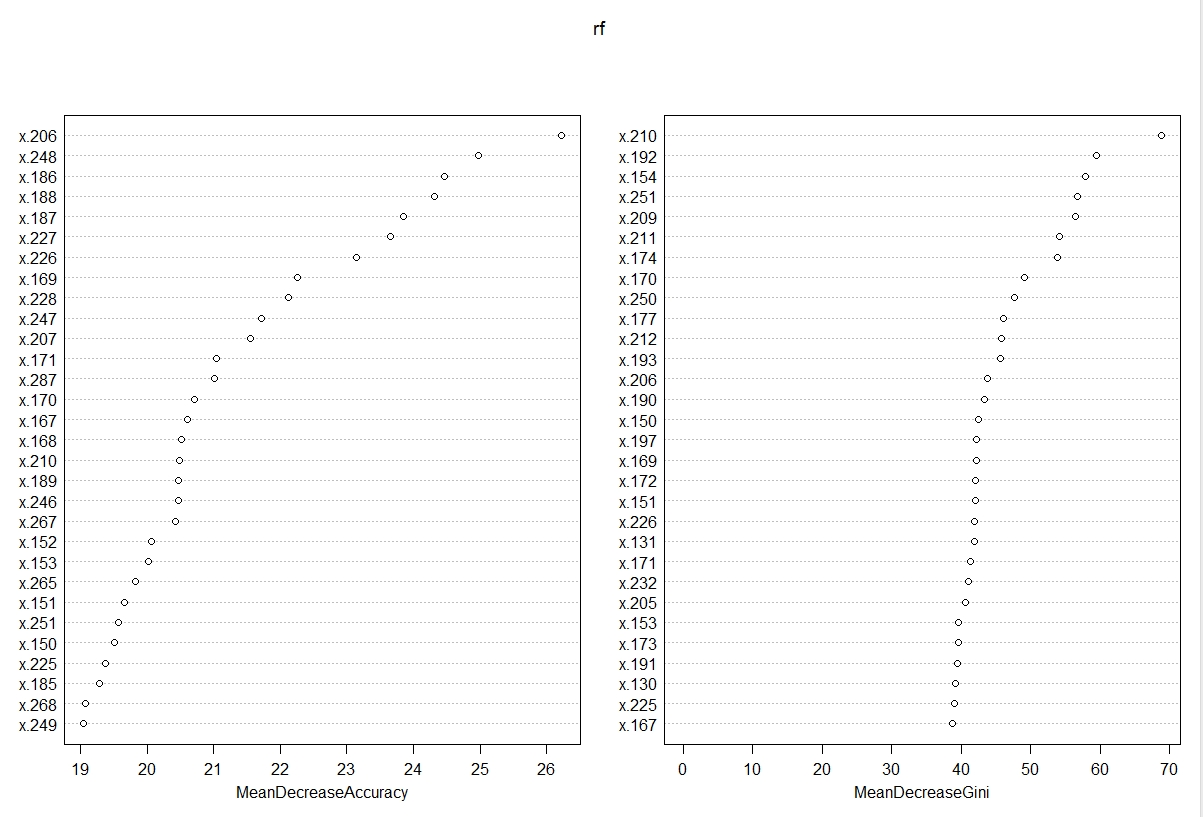
Figure 1: the error rate of different class along with the number of trees

From figure 1 we can see that the error rate will not change after 300 and therefore we set the number of trees as 500 is appropriate.

The importance of the variable plot are in the following, the larger the MeanDecreaseAccuracy is, the more important the feature is:

imp=importance(rf)

varImpPlot(rf)



Then we test on the test data:

start\_time=proc.time()

pre\_test=predict(rf,test.data[,-1])

end\_time=proc.time()

run\_time=start\_time-end\_time

error\_rate=sum(pre\_test!=test.label)/nrow(test.data)

> run\_time

user system elapsed

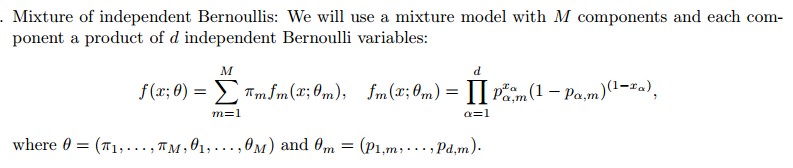
1.61 0.08 1.78

> error\_rate

[1] 0.0655

### EM Classification of the Handwritten Digits

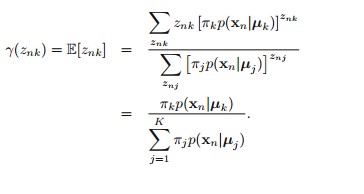
Theory:



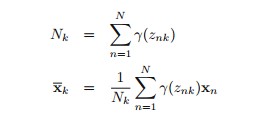
C:\Users\Christina\AppData\Roaming\Tencent\Users\919834852\QQ\WinTemp\RichOle\%9FQ}@VOUSC}OPM`0KLEFKH.jpg

C:\Users\Christina\AppData\Roaming\Tencent\Users\919834852\QQ\WinTemp\RichOle\LXI6J1QM]J%TR8{2LH5{K89.jpg

E STEP:



M STEP:



C:\Users\Christina\AppData\Roaming\Tencent\Users\919834852\QQ\WinTemp\RichOle\[2{R]5PL~S]ZY$8)1G~PFU6.jpg

C:\Users\Christina\AppData\Roaming\Tencent\Users\919834852\QQ\WinTemp\RichOle\)7P}JQ3CZU_OV{L{WKJ@G%U.jpg

#compute initial values of p and pi

p=matrix(0,d,num.class)

for(i in 1:num.class){

p[,i]=(colSums(training.data[i,,])+1)/(nrow(training.data[i,,])+2)

}

pi=matrix(0,1,num.class)

for(i in 1:num.class){

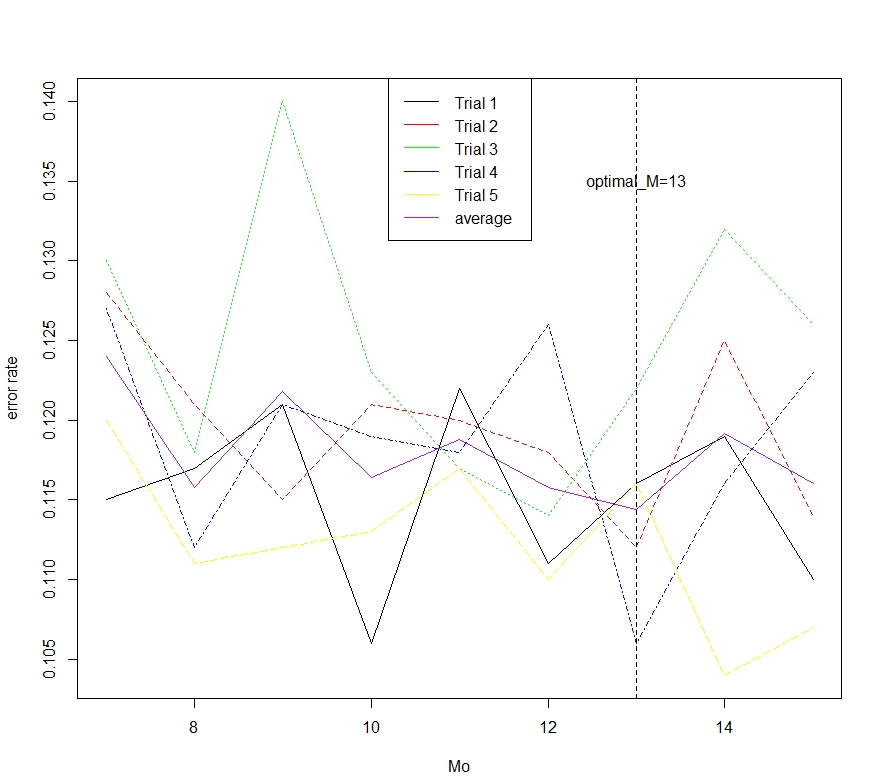
pi[,i]=(nrow(training.data[i,,])+1)/(num.class \* num.training+10)

}

C:\Users\Christina\AppData\Roaming\Tencent\Users\919834852\QQ\WinTemp\RichOle\7_(PVR2QCL1GLHR$ND}7Q30.jpgC:\Users\Christina\AppData\Roaming\Tencent\Users\919834852\QQ\WinTemp\RichOle\LU60%J24W9]ZQJL0C_7XYB0.jpg

C:\Users\Christina\AppData\Roaming\Tencent\Users\919834852\QQ\WinTemp\RichOle\NQFC5)O@[%DNIA_0NWM[L2L.jpg

We run the algorithm five times and record the error rate over the M parameter. The results is in the following figure, from the following figure, we find that the optimal M parameter is 13



With the optimal cost, I then train the whole training data and test on the testing

data

> run\_time

user system elapsed

1.32 0.18 4.32

> testerror

[1] 0.1007

### Logistic Model to Classify the Handwritten Digits

We use the glmnet package , the description of the function are in the following:

fit a GLM with lasso or elasticnet regularization

**Description**

Fit a generalized linear model via penalized maximum likelihood. The regularization path is computed for the lasso or elasticnet penalty at a grid of values for the regularization parameter lambda. Can deal with all shapes of data, including very large sparse data matrices. Fits linear, logistic and multinomial, poisson, and Cox regression models.

**Usage**

glmnet(x, y,family=c("gaussian","binomial","poisson","multinomial","cox","mgaussian"),

weights, offset=NULL, alpha = 1, nlambda = 100,

lambda.min.ratio = ifelse(nobs<nvars,0.01,0.0001), lambda=NULL,

standardize = TRUE, intercept=TRUE, thresh = 1e-07, dfmax = nvars + 1,

pmax = min(dfmax \* 2+20, nvars), exclude, penalty.factor = rep(1, nvars),

lower.limits=-Inf, upper.limits=Inf, maxit=100000,

type.gaussian=ifelse(nvars<500,"covariance","naive"),

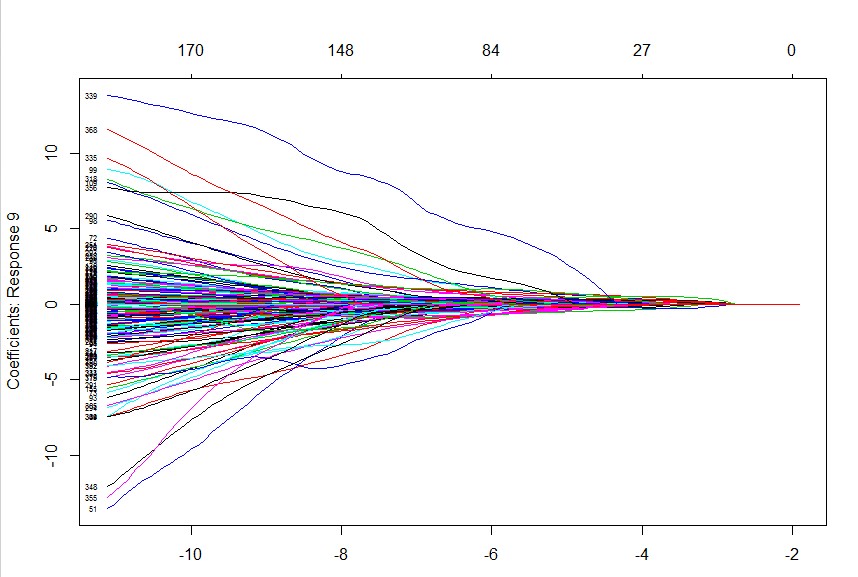
type.logistic=c("Newton","modified.Newton"),

standardize.response=FALSE, type.multinomial=c("ungrouped","grouped"))

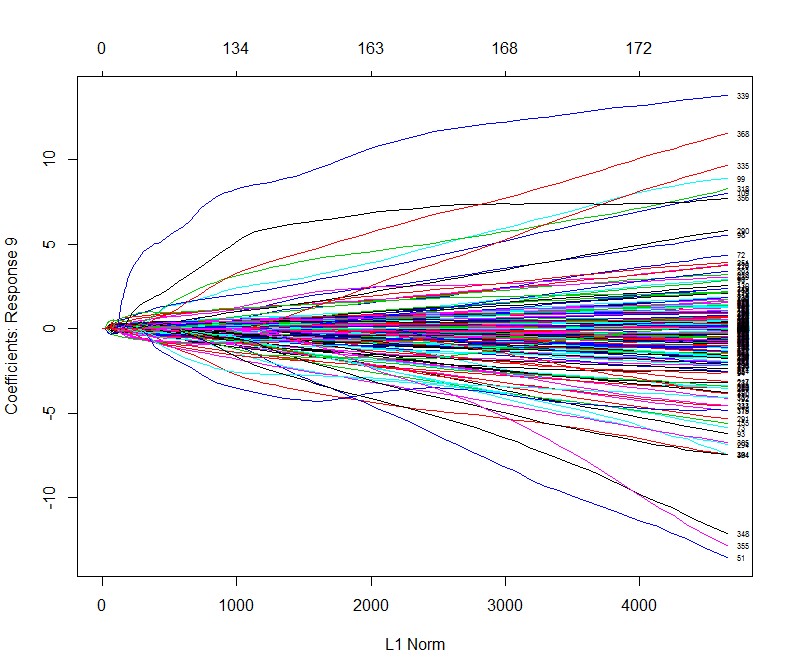
**Arguments**

|  |  |
| --- | --- |
| x | input matrix, of dimension nobs x nvars; each row is an observation vector. Can be in sparse matrix format (inherit from class "sparseMatrix" as in package Matrix; not yet available for family="cox") |
| y | response variable. Quantitative for family="gaussian", or family="poisson" (non-negative counts). For family="binomial" should be either a factor with two levels, or a two-column matrix of counts or proportions (the second column is treated as the target class; for a factor, the last level in alphabetical order is the target class). Forfamily="multinomial", can be a nc>=2 level factor, or a matrix with nc columns of counts or proportions. For either "binomial" or "multinomial", if y is presented as a vector, it will be coerced into a factor. For family="cox", y should be a two-column matrix with columns named 'time' and 'status'. The latter is a binary variable, with '1' indicating death, and '0' indicating right censored. The function Surv() in package **survival** produces such a matrix. For family="mgaussian", y is a matrix of quantitative responses. |
| family | Response type (see above) |
| weights | observation weights. Can be total counts if responses are proportion matrices. Default is 1 for each observation |
| offset | A vector of length nobs that is included in the linear predictor (a nobs x nc matrix for the"multinomial" family). Useful for the "poisson" family (e.g. log of exposure time), or for refining a model by starting at a current fit. Default is NULL. If supplied, then values must also be supplied to the predict function. |
| alpha | The elasticnet mixing parameter, with *0≤α≤ 1*. The penalty is defined as  *(1-α)/2||β||\_2^2+α||β||\_1.*  alpha=1 is the lasso penalty, and alpha=0 the ridge penalty. |
| nlambda | The number of lambda values - default is 100. |
| lambda.min.ratio | Smallest value for lambda, as a fraction of lambda.max, the (data derived) entry value (i.e. the smallest value for which all coefficients are zero). The default depends on the sample size nobs relative to the number of variables nvars. If nobs > nvars, the default is0.0001, close to zero. If nobs < nvars, the default is 0.01. A very small value oflambda.min.ratio will lead to a saturated fit in the nobs < nvars case. This is undefined for "binomial" and "multinomial" models, and glmnet will exit gracefully when the percentage deviance explained is almost 1. |
| lambda | A user supplied lambda sequence. Typical usage is to have the program compute its ownlambda sequence based on nlambda and lambda.min.ratio. Supplying a value oflambda overrides this. WARNING: use with care. Do not supply a single value for lambda(for predictions after CV use predict() instead). Supply instead a decreasing sequence of lambda values. glmnet relies on its warms starts for speed, and its often faster to fit a whole path than compute a single fit. |
| standardize | Logical flag for x variable standardization, prior to fitting the model sequence. The coefficients are always returned on the original scale. Default is standardize=TRUE. If variables are in the same units already, you might not wish to standardize. See details below for y standardization with family="gaussian". |
| intercept | Should intercept(s) be fitted (default=TRUE) or set to zero (FALSE) |
| thresh | Convergence threshold for coordinate descent. Each inner coordinate-descent loop continues until the maximum change in the objective after any coefficient update is less than thresh times the null deviance. Defaults value is 1E-7. |
| dfmax | Limit the maximum number of variables in the model. Useful for very large nvars, if a partial path is desired. |
| pmax | Limit the maximum number of variables ever to be nonzero |
| exclude | Indices of variables to be excluded from the model. Default is none. Equivalent to an infinite penalty factor (next item). |
| penalty.factor | Separate penalty factors can be applied to each coefficient. This is a number that multiplieslambda to allow differential shrinkage. Can be 0 for some variables, which implies no shrinkage, and that variable is always included in the model. Default is 1 for all variables (and implicitly infinity for variables listed in exclude). Note: the penalty factors are internally rescaled to sum to nvars, and the lambda sequence will reflect this change. |
| lower.limits | Vector of lower limits for each coefficient; default -Inf. Each of these must be non-positive. Can be presented as a single value (which will then be replicated), else a vector of lengthnvars |
| upper.limits | Vector of upper limits for each coefficient; default Inf. See lower.limits |
| maxit | Maximum number of passes over the data for all lambda values; default is 10^5. |
| type.gaussian | Two algorithm types are supported for (only) family="gaussian". The default whennvar<500 is type.gaussian="covariance", and saves all inner-products ever computed. This can be much faster than type.gaussian="naive", which loops throughnobs every time an inner-product is computed. The latter can be far more efficient for nvar >> nobs situations, or when nvar > 500. |
| type.logistic | If "Newton" then the exact hessian is used (default), while "modified.Newton" uses an upper-bound on the hessian, and can be faster. |
| standardize.response | This is for the family="mgaussian" family, and allows the user to standardize the response variables |
| type.multinomial | If "grouped" then a grouped lasso penalty is used on the multinomial coefficients for a variable. This ensures they are all in our out together. The default is "ungrouped" |

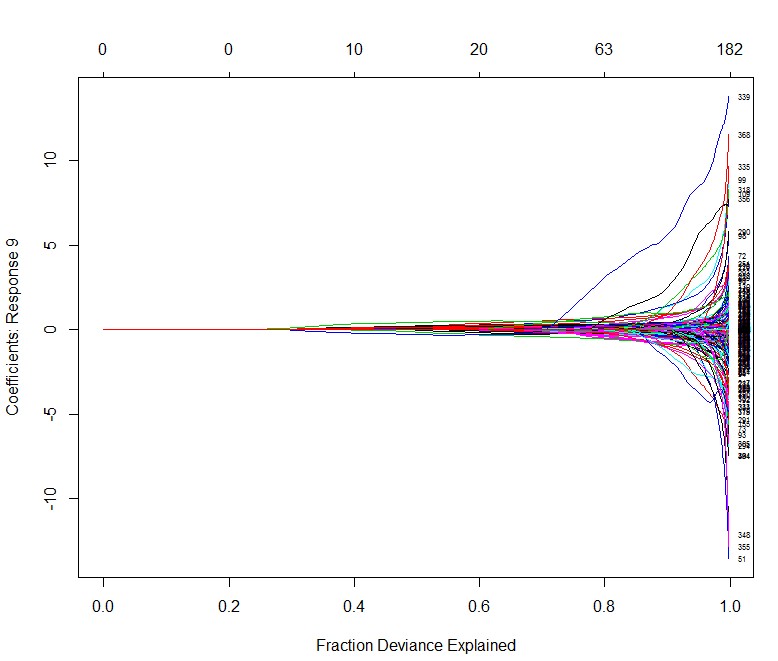
The three pictures below is used for my understanding of the GLM method



This figure shows how the coefficient change along with the log(lambda)



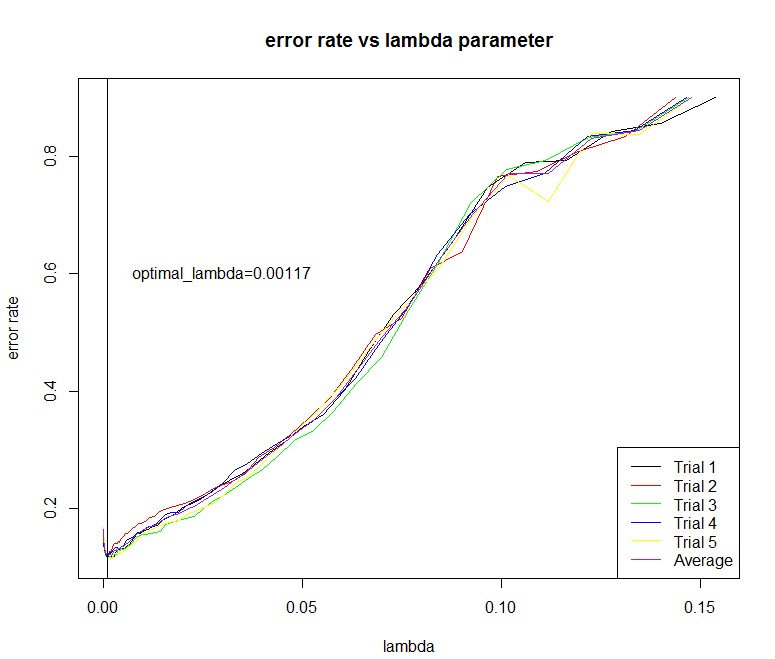
This Figure shows how the coefficient change along with the L1 Norm ( the L1 Norm of the coefficient)



This figure shows how the coefficient change along with Fraction Deviance Explaned

Here we use the method family=”multinormial” The error rate plots after running the algorithm 5 times with default 100 lambda are in the following plot.

From the below plot we can see that the optimal lambda is 0.0017.



#Then we do logistic regression on the whole training data with the optimal parameter

model=glmnet(training.data,training.label,"multinomial")

#do test on test data

start\_time=proc.time()

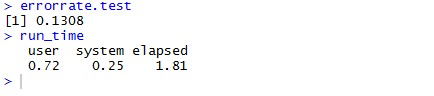
pre.test=predict(model,test.data,lambda.optimal,"class")

end\_time=proc.time()

run\_time=end\_time-start\_time

errorrate.test=sum((pre.test!=test.label))/nrow(test.data)

The results we get are in the following:



|  |  |
| --- | --- |
| Average predict time of one data stream(s) | The error rate of logistic method |
| 1.81e(-4) | 0.1308 |

# KDDCUP 99 Data Analysis(kddcup.data\_10\_percent\_corrected)

## Logistic classification of the KDDCUP 99 Data Analysis

**Process of the data**

* The last column is the label, we treat them as two part. One is” normal “and the other is” attack”
* The second, third and fourth column are characters. We use the following code to transfer these character into numbers:

for (i in 1:41){

kddcup\_train[,i]=as.numeric(kddcup\_train[,i])

kddcup\_test[,i]=as.numeric(kddcup\_test[,i])

}

* Finally we transfer the data into matrix only with numbers

**Implement of the data**

**Cross validation to find the optimal lambda**

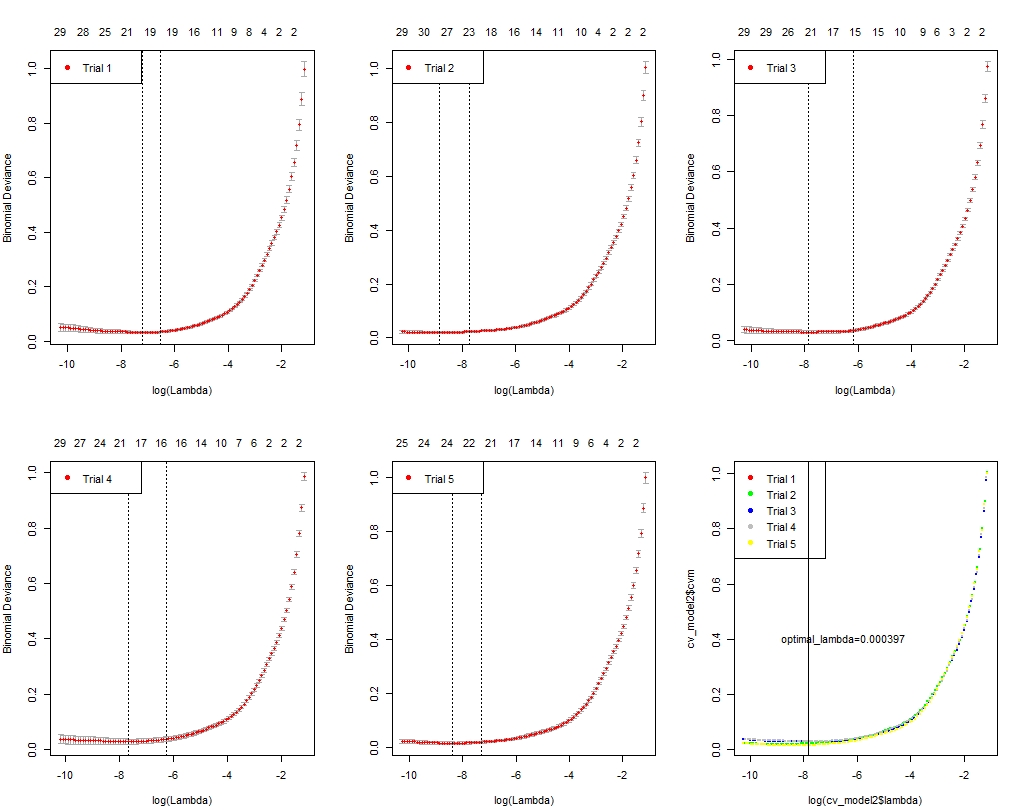
We implement the data with cross validation (generalized linear model) to find the optimal lambda with the following model:

cv\_model=cv.glmnet (a1[1:5000,-42],a1[1:5000,42],family="binomial")

I sample the data with size 5000 and 2000 respectively. 5000 data is the training data and 2000 data is the testing data.

I run the algorithm 5 times and you can see the figures in the next page.

From the figures, we can see that the optimal lambda is 0.000397.



**Train with larger size training data (20000) and larger size test data (5000)**

model=glmnet(kddcup\_train[1:20000,-42],kddcup\_train[1:20000,42],family="binomial")

#do test on test data

start\_time=proc.time()

pre.test=predict(model,kddcup\_test[1:5000,-42],0.000397,"class")

end\_time=proc.time()

run\_time=end\_time-start\_time

errorrate.test=sum((pre.test!=kddcup\_test[,42]))/nrow(kddcup\_test)

The result we get are in the following:

> errorrate.test

[1] 0.0084

> run\_time

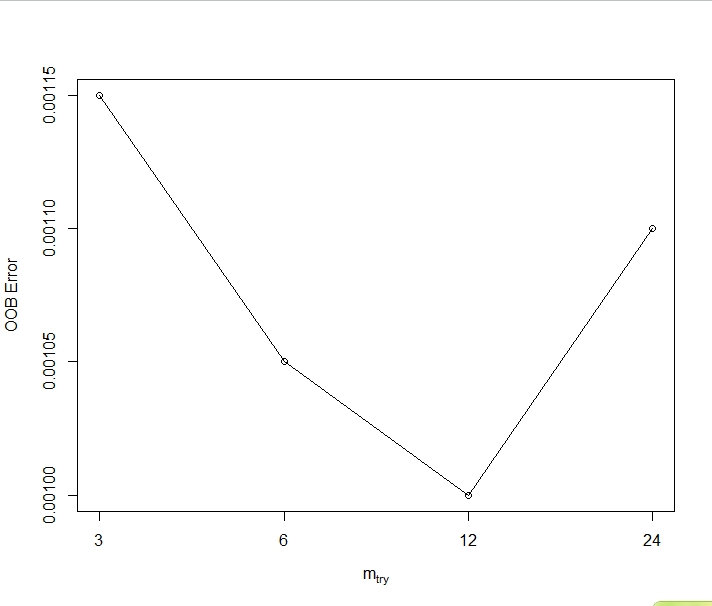
user system elapsed

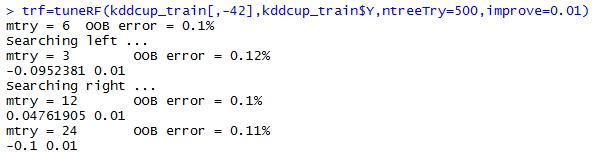
0.02 0.03 2.66

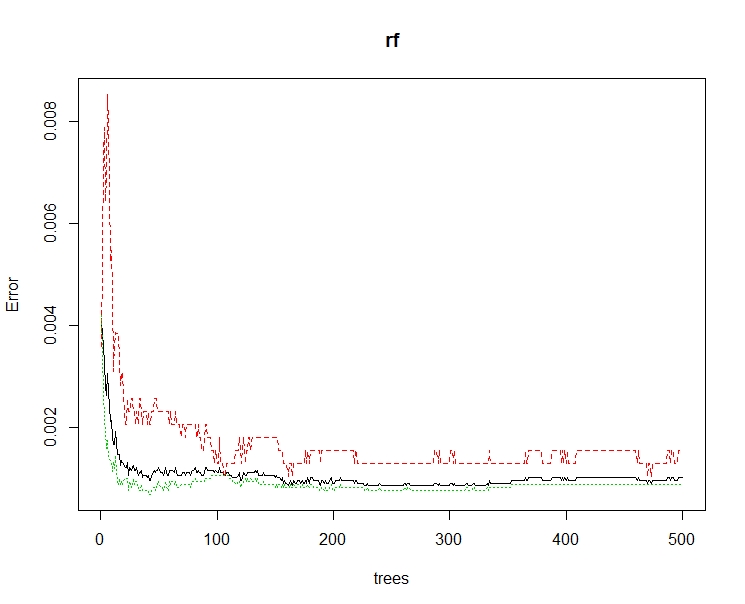
|  |  |
| --- | --- |
| Average time of one data stream(s) | Error rate |
| 1.33e(-4) | 0.0084 |

# RandomForest classification of Kddcup data

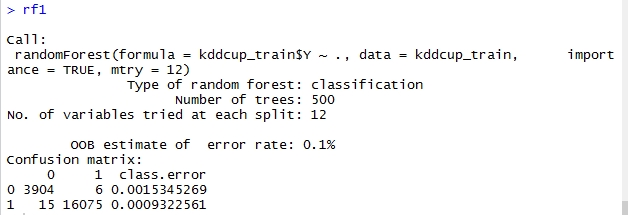
We process the data as we state above. Then we randomly select 20000 data as the training data and 5000 data as the test data. We need to find the optimal mtry first. From the following Figure, we can find that the optimal mytry is 12.





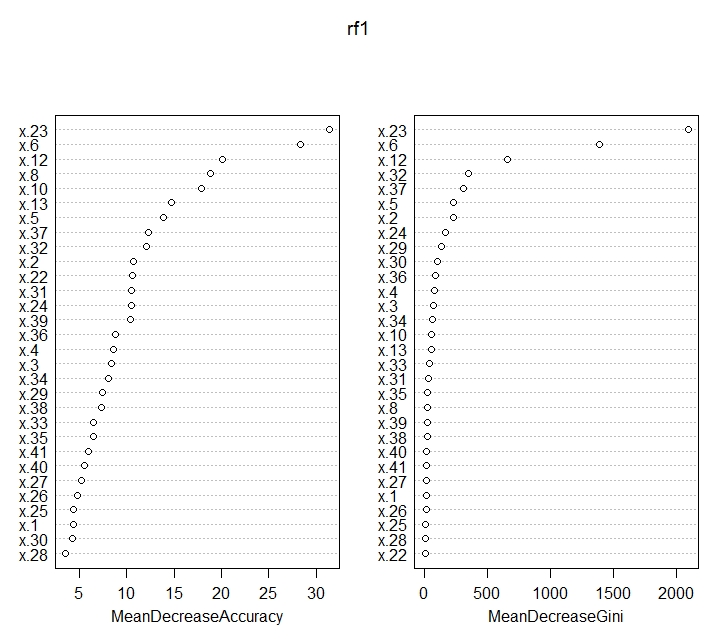


The picture above shows how OOB error changes over the number of trees. We can see that the OOB error almost stays the same when the number of trees is larger than 300. Here, we use the default 500 to train the whole data.



This is the model of the random forest when we train the whole 20000 training data.

The picture in the below shows the importance of the features when we apply the method of random forest



> run\_time

user system elapsed

0.20 0.12 2.39

> error\_rate

[1] 0.0026

## SVM Method to Classify the Kddcup99 Data

**Process of the Data:**

* The last column is the label, we treat them as two part. One is” normal “and the other is” attack”
* The second, third and fourth column are characters. We use the following code to transfer these character into numbers and treat the last column as factor

for (i in 1:41){

kddcup\_train[,i]=as.numeric(kddcup\_train[,i])

kddcup\_test[,i]=as.numeric(kddcup\_test[,i])

}

* Finally we find that some columns are constant. We use the following command to remove these columns:

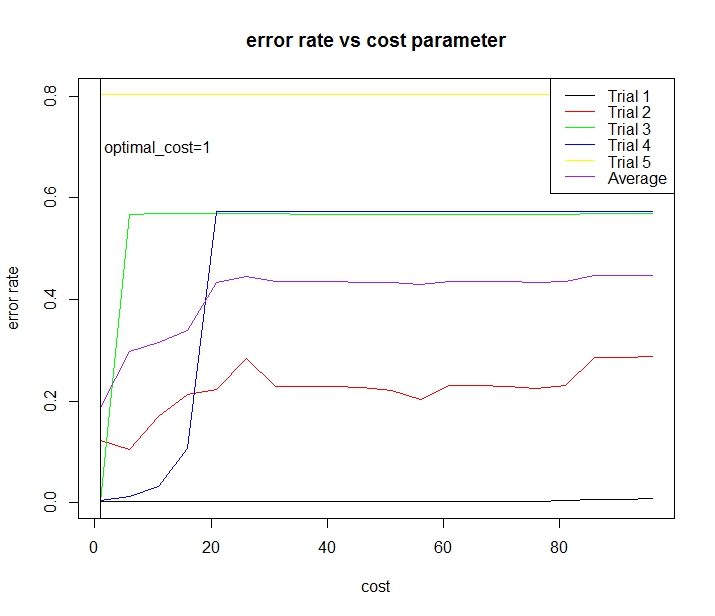
kddcup\_train1=subset(kddcup\_train,select=c(x.7,x.21,x.9,x.11,x.15,x.22,x.20,x.14,x.18))

kddcup\_test1=subset(kddcup\_test,select=-c(x.7,x.21,x.9,x.11,x.15,x.22,x.20,x.14,x.18))

**Implement the Data to find the optimal cost**

Sample the data with size 5000 and 2000 respectively. 5000 data is the training data and 2000 data is the testing data. Run the algorithm 5 times with the cost parameter ranging from 1 to one hundred.

The figure of error\_rate vs cost parameter is in the following picture, we can find that the optima cost is 1



**Train with larger size training data (20000) and larger size test data(5000)**

model\_svm=svm(kddcup\_train2$Y~.,method="class",data=kddcup\_train2,cost=1)

start\_time=proc.time()

pre\_svm2=predict(model\_svm,newdata=kddcup\_test2,type="class")

end\_time=proc.time()

run\_time=end\_time-start\_time

error\_rate=sum(kddcup\_test[,42]!=pre\_svm2)/nrow(kddcup\_test)

The result are in the following:

> error\_rate

[1] 0.0042

> run\_time

user system elapsed

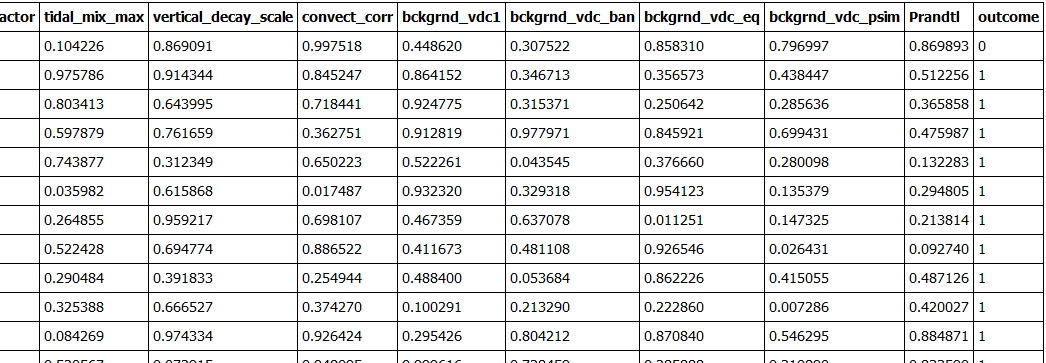
0.19 0.00 0.19

|  |  |
| --- | --- |
| Average time of one data stream(s) | Error\_rate |
| e(-5) | 0.0042 |

## Climate Data Analysis:(for detailed refer to the climate data

## SVM to classify the climate data: (Python sklearn)

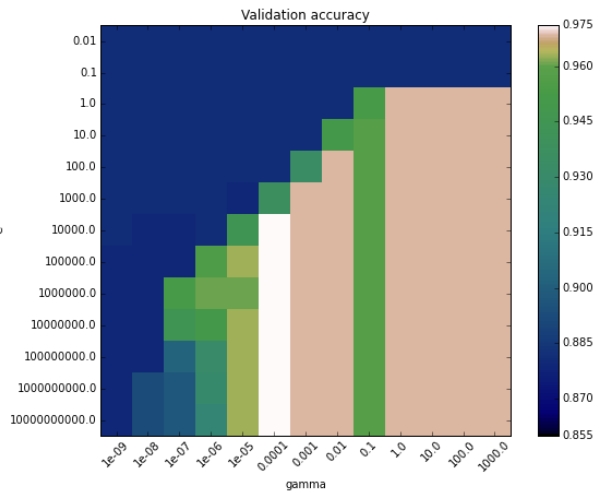
The following is the format of the data with 20 features:



Split the data into training set(0.7) and test set(0.3) randomly and train the training set with respect to the parameter "gamma" and "C" and plot the heatmap to visualize the result

A heat map is a two-dimensional representation of data in which values are represented by colors. A simple heat map provides an immediate visual summary of information. More elaborate heat maps allow the viewer to understand complex data sets.

Draw heatmap of the validation accuracy as a function of gamma and C The score are encoded as colors with the hot colormap which varies from dark red to bright yellow. As the most interesting scores are all located in the 0.95 to 0.97 range we use a custom normalizer to set the mid-point to 0.96 so as to make it easier to visualize the small variations of score values in the interesting range while not brutally collapsing all the low score values to the same color.



From the picture above, we can see that the best gamma and C is 0.0001 and 10000 with the accuracy rate 0.97. Then, we train the SVM model with the whole training data and test on the testing data with the optimal parameters.

Error Rate: 6.17%, Running Time per Stream:5.3e-06

## RandomForest classification of climate data

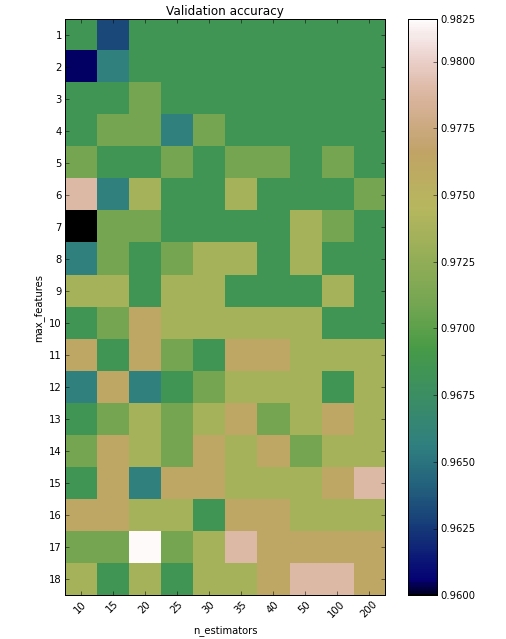
Split the data into taining set(0.7) and test set(0.3) randomly and train the training set with respect to the parameter "n\_estimators" and "max\_features" and plot the heatmap to visiualize the result.

For an initial search, we set the feature is from 2 to 18 and the n\_estimators is from 10 to 2000. Draw heatmap of the validation accuracy as a function of n\_estimator and max\_features. The score are encoded as colors with the hot colormap which varies from dark red to bright yellow. As the most interesting scores are all located in the 0.96 to 0.9825 range we use a custom normalizer to set the mid-point to 0.97 so as to make it easier to visualize the small variations of score values in the interesting range while not brutally collapsing all the low score values to the same color.

From the picture below, we can see that the optimal n\_estimators is 20 and max\_features is 17 with the accuracy 0.97. Then we train the whole training data with the optimal parameters and test on the test data set

Error Rate: 4.9%

Running Time per Stream: 1.08e(-05)



**Show the importance of the features**

Feature ranking:

1. feature 2 (0.320250)

2. feature 3 (0.287561)

3. feature 14 (0.102951)

4. feature 19 (0.048369)

5. feature 12 (0.032164)

6. feature 13 (0.030537)

7. feature 5 (0.029772)

8. feature 16 (0.022735)

9. feature 8 (0.022478)

10. feature 1 (0.018122)

11. feature 11 (0.016845)

12. feature 15 (0.016615)

13. feature 17 (0.011008)

14. feature 18 (0.009362)

15. feature 7 (0.008403)

16. feature 4 (0.007262)

17. feature 10 (0.006733)

18. feature 9 (0.006600)

19. feature 6 (0.002233)

20. feature 0 (0.000000)