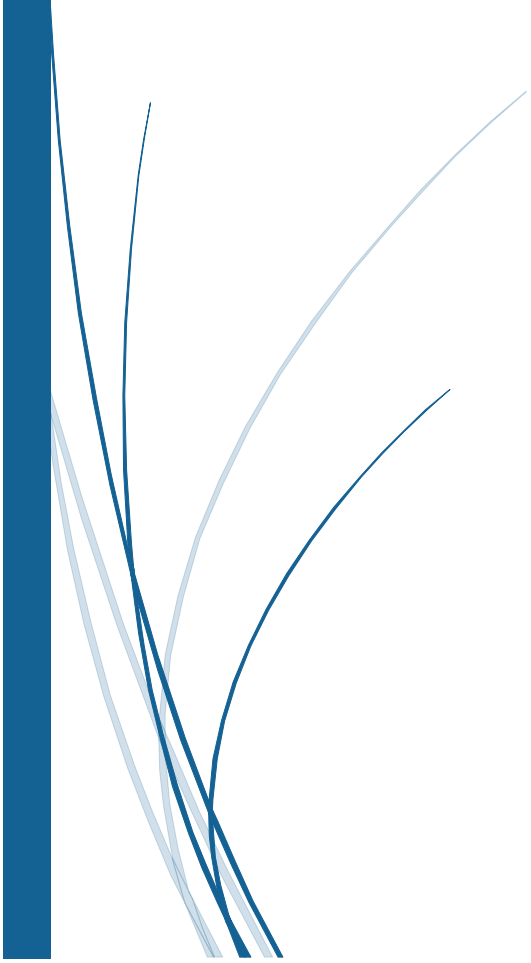
4/17/2015

# Take Home 3 Classification

ISyE6740



# CONTENTS

## DATA MANIPULATION (Q1)

The following R code specifies the necessary columns (all but EYE, GENDER, ETHNIC, HGT, WT, ASPH, ACYL, SE, AXL, CACD, AGE, CCT.OD, and PCCURV mm) from the predictors to be used in the prediction model training as well as remove rows with missing data from these columns and maintain the corresponding responses:

```
#read in data

myData = read.csv("AngleClosure.csv",header=TRUE)

myCases= read.csv("AngleClosure_ValidationCases.csv",header=TRUE)

myControls = read.csv("AngleClosure_ValidationControls.csv",header=TRUE)

myDataNew = cbind(myData[,2:14], myData[17:25])

#select necessary columns

myPredictors = myDataNew[,1:12]

#eliminate rows with NA's from predictors

myPredictors = myDataNew[rowSums(is.na(myPredictors))==0,]

#select the corresponding responses

myDataNew2 <- myDataNew[rowSums(is.na(myDataNew[,1:12]))==0,]

#convert to logical

myResponse = myDataNew2$ANGLE.CLOSURE=="YES"
```

# DEVELOP PREDICTION MODELS (Q2 AND 3)

The following 5 models are proposed to predict the response of the ultimate validation data. Most have been tuned with respect to important parameters that optimize their AUC (Area Under Curve) in the ROC graph (Risk Operating Curve):

1. Support Vector Machines (tuned Gamma and Cost)
2. Ada Boost (tuned Nu)
3. Neural Network (Decay – Lambda and Size)
4. Random Forest (mtry)
5. Logistic (Probit) Regression (Number of Steps)

The next section will focus on Parameter Selection as this one will focus on the process of Cross Validation and actually tuning the model in code.

## SUPPORT VECTOR MACHINE

The Support Vector Machine Model was tuned via two parameters:

1. **Cost of constraints violation**: tuned between *0.01 and 100*.
2. **Gamma**: how far the influence of a single training observation reaches, with low values meaning 'far' and high values meaning 'close': tuned between $10^{-6}$ *and 0.1*.

```
#SVM#MODEL 1
library(e1071)
library(pROC)

#define number of folds and iterations
nFolds=10
nIter=25
#define range of costs and Gamma values
myCosts = 10^(-2:2)
myGamma = 10^seq(-6,-1,0.5)
#define storage for AUC values
SVM_AUC_plot=matrix(NA,length(myCosts)*length(myGamma)*nIter,3)

count = 0
#forall iterations
for(iter in 1:nIter){
  #generate 1/10th sample
  testingIndices=sample(length(myLabel))[1:round(length(myLabel)/nFolds)]
  #define testing data
  myDataTesting=dataStar[testingIndices,]
  #define training data
```

```
  myDataTraining=dataStar[-testingIndices,]
  #for all gammas and costs
  for (g in 1:length(myGamma)){
    for(c in 1:length(myCosts)){
      #fit the SVM model with the given gamma and cost
      fit=svm(myLabel~.,data=myDataTraining, cost=myCosts[c], gamma = myGamma[g],
      probability = TRUE)
      #generate prediction in terms of probability
      myPreds= predict(fit,newdata=myDataTesting, probability = TRUE)
      probabilities = attr(myPreds,"probabilities")[,1]
      #increase counter
      count = count + 1
      #record AUC
      SVM_AUC_plot[count,] =
      c(myGamma[g],myCosts[c],as.numeric(roc(myDataTesting$myLabel,probabilities)$a
      uc))
    }
  }
}

#get the average over each parameter setting
attach(as.data.frame(SVM_AUC_plot))
SVM_AUC_avg <-aggregate(SVM_AUC_plot, by=list(V1,V2),
                    FUN=mean, na.rm=TRUE)

library(lattice)
AUC = SVM_AUC_avg[,5]
Gamma = SVM_AUC_avg[,3]
Costs = SVM_AUC_avg[,4]
#plot contour map
levelplot(AUC~Gamma*Costs, aspect = "fill", labels = TRUE, contour = TRUE, pretty =
TRUE)
#get best gamma and costs
BestGamma = Gamma[which.max(AUC)]
BestCosts = Costs[which.max(AUC)]
```

## ADA BOOST

The Ada Boost Model was tuned via one parameters, Nu, the shrinkage factor, which is used to slow down learning and reduce over-fitting. This parameter was tuned between $10^{-7}$ *and 1*, where 1 means no shrinkage.

```
#ADA#MODEL 2
#define number of folds and iterations
library(ada)
nFolds=10
nIter=25
#define range of nu values
myNUs=10^seq(-7,0,0.5)
#define storage for AUC values
ADA_AUC_plot=matrix(NA,nIter,length(myNUs))
count = 0;

#forall iterations
for(iter in 1:nIter){
  #generate 1/10th sample
  testingIndices=sample(length(myLabel))[1:round(length(myLabel)/nFolds)]
  #define testing data
  myDataTesting=dataStar[testingIndices,]
  #define training data
  myDataTraining=dataStar[-testingIndices,]
  for (n in 1:length(myNUs)){
    #fit model
    fit=ada(myLabel~.,data=myDataTraining, nu=myNUs[n])
    #generate prediction in terms of probability
    myPreds= predict(fit,newdata=myDataTesting, type = "probs")[,2]
    #record AUC
    ADA_AUC_plot[iter,n] =
as.numeric(roc(as.numeric(myDataTesting$myLabel),as.numeric(myPreds))$auc)
  }
}
#plot column means (column is a setting)
plot(x=myNUs,y= colMeans(ADA_AUC_plot),   main="ADA: AUC vs Nu",
     xlab="Nu", ylab="AUC", mar = c(.1,.1,.1,.1))

#get best NU with highest AUC
BEST_NU = myNUs[which.max(colMeans(ADA_AUC_plot))]
```

## NEURAL NETWORK

The Neural Network model was tuned in two parameters:

1. **Decay (Lambda)**: a regularization parameter to avoid over-fitting, where the fit criterion is altered by a factor lambda multiplied by a cost for the "roughness of the model" – tuned between $10^{-7}$ *and 1*
2. **Size:** number of units in the hidden layer – tuned between 5 and 15 (default 10).

```
#NEURAL NET #MODEL 3
library(nnet)
#define number of folds and iterations
nFolds=10
nIter=25
#define range of decay and size values
myLambdas=10^seq(-7,1,0.5)
mySizes = 5:15
#define storage for AUC values
NN_AUC_plot=matrix(NA,length(myLambdas)*length(mySizes)*nIter,3)

count = 0;

for(iter in 1:nIter){
  testingIndices=sample(length(myLabel))[1:round(length(myLabel)/nFolds)]
  myDataTesting=dataStar[testingIndices,]
  myDataTraining=dataStar[-testingIndices,]
  #for each lamda and size combination
  for(lambda in myLambdas){
    for (size in mySizes){
      #fit model
      fit=nnet(myLabel~.,data=myDataTraining,size=10,decay=lambda)
      count = count+1;
      #generate prediction
      myPreds=predict(fit,newdata=myDataTesting, probability = TRUE)
      #store AUC
      NN_AUC_plot[count,] = c(lambda, size, as.numeric(roc(myDataTesting$myLabel,myPreds)$auc))
    }
  }
}
#get the average over each parameter setting
attach(as.data.frame(NN_AUC_plot))
NN_AUC_avg <-aggregate(NN_AUC_plot, by=list(V1,V2),
                       FUN=mean, na.rm=TRUE)
#plot AUC vs each individual parameter
plot(x=NN_AUC_avg[,1],y= NN_AUC_avg[,5],   main="Neural Net: AUC vs Lambda",
     xlab="Lambda", ylab="AUC", mar = c(.1,.1,.1,.1))
```

```
plot(x=NN_AUC_avg[,2],y= NN_AUC_avg[,5],   main="Neural Net: AUC vs Size",
    xlab="Size", ylab="AUC" )
library(lattice)
AUC = NN_AUC_avg[,5]
Lambda = NN_AUC_avg[,3]
Size = NN_AUC_avg[,4]
#plot contour map
levelplot(AUC~Lambda*Size, aspect = "fill", labels = TRUE, contour = TRUE, pretty = TRUE)
#get best lambda and size
BestLambda = Lamdba[which.max(AUC)]
BestSize = Size[which.max(AUC)]
```

## RANDOM FOREST

The Random Forest Model was tuned via the **mtry** parameter: the number of variables randomly sampled as candidates at each split – tuned between *1 and 10*.

```
#Random Forest #MODEL 4
library(randomForest)
#define number of folds and iterations
nFolds=10
nIter=25
#define range of mtry values
myMTry = seq(1,10,1)
#define storage for AUC values
RF_AUC_plot=matrix(NA,nIter,length(myMTry))

library(pROC)

for(iter in 1:nIter){
  testingIndices=sample(length(myLabel))[1:round(length(myLabel)/nFolds)]
  #define testing data
  myDataTesting=dataStar[testingIndices,]
  #define training data
  myDataTraining=dataStar[-testingIndices,]
  for(t in 1:length(myMTry)){
      #fit model
      fit=randomForest(myLabel~.,data=myDataTraining, mtry=myMTry[t])
      #generate prediction in terms of probability
      myPreds= predict(fit,newdata=myDataTesting, type = "prob")[,2]
      #record AUC
      RF_AUC_plot[iter,t]=as.numeric(roc(myDataTesting$myLabel,myPreds)$auc)
  }
}

#plot column means (column is a setting)
plot(colMeans(RF_AUC_plot), main="Random Forest: AUC vs mtry",
     xlab="mtry", ylab="AUC")
#get best mtry with highest AUC
BEST_MTRY = myMTry[which.max(colMeans(RF_AUC_plot))]
```

## LOGISTIC (PROBIT) REGRESSION

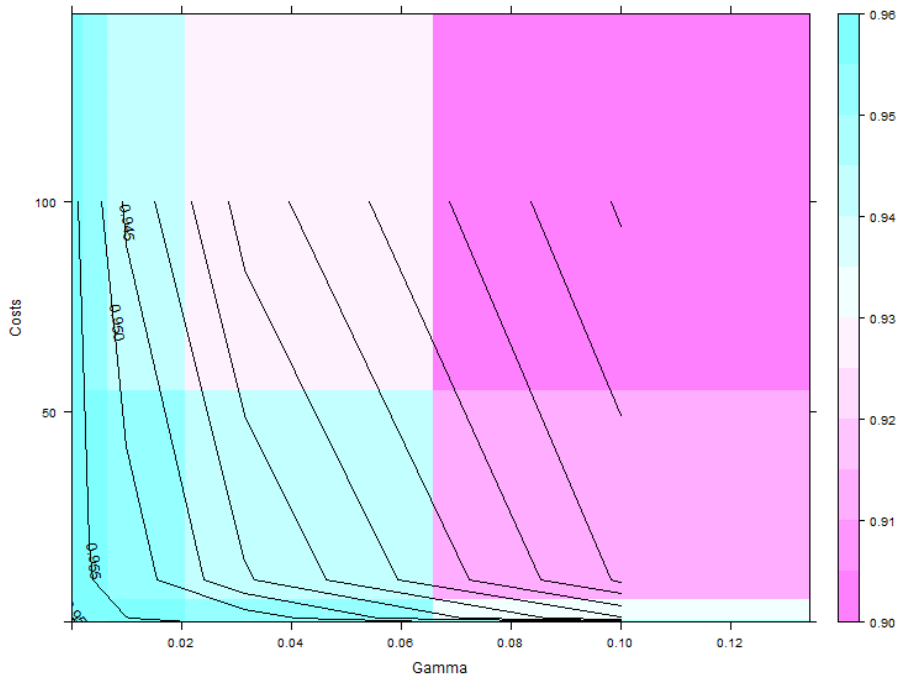
The Logistic Regression Model (using the Probit link) using two-directional stepwise AIC regression was tuned via the number of steps allowed in the stepwise regression: tuned between 1 and 11 (total number of variables).

```
#Logistic-Probit Regression#MODEL 5
library(MASS)
#define number of folds and iterations
nFolds=10
nIter=25
#define range of step values
mySteps = seq(1,11,1)
#define storage for AUC values
GLM_AUC_plot=matrix(NA,nIter,length(mySteps))
for(iter in 1:nIter){
  testingIndices=sample(length(myLabel))[1:round(length(myLabel)/nFolds)]
  #define testing data
  myDataTesting=dataStar[testingIndices,]
  # define training data
  myDataTraining=dataStar[-testingIndices,]
  for(t in 1:length(mySteps)){
    #define starting model (all predictors)
    test = glm(myLabel~.,data=myDataTraining,family=binomial(link=probit))
    #perform stepwise regression in both directions
    step = step(test, direction = "both", steps = mySteps[t])
    #store output model
    newDataTraining = step$model
    #fit output model
    fit=glm(myLabel~.,data=newDataTraining,family=binomial(link=probit))
    #generate prediction in terms of probability
    myPreds=predict(fit,newdata=myDataTesting,family=binomial(link=probit), type =
"response")
    #record AUC
    GLM_AUC_plot[iter,t]=as.numeric(roc(myDataTesting$myLabel,myPreds)$auc)
  }
}
#get best steps value
mySteps[which.max(colMeans(GLM_AUC_plot))]
```

# PARAMETER SELECTION AND VISUALIZATION (Q3 AND 6)

## SUPPORT VECTOR MACHINE

Using the R code provided in the previous section and AUC result values in Appendix 1, we generate the following contour plot of AUC vs Costs and Gamma where the gradient value depicted on the right is the AUC value, as well as the values on the contour.

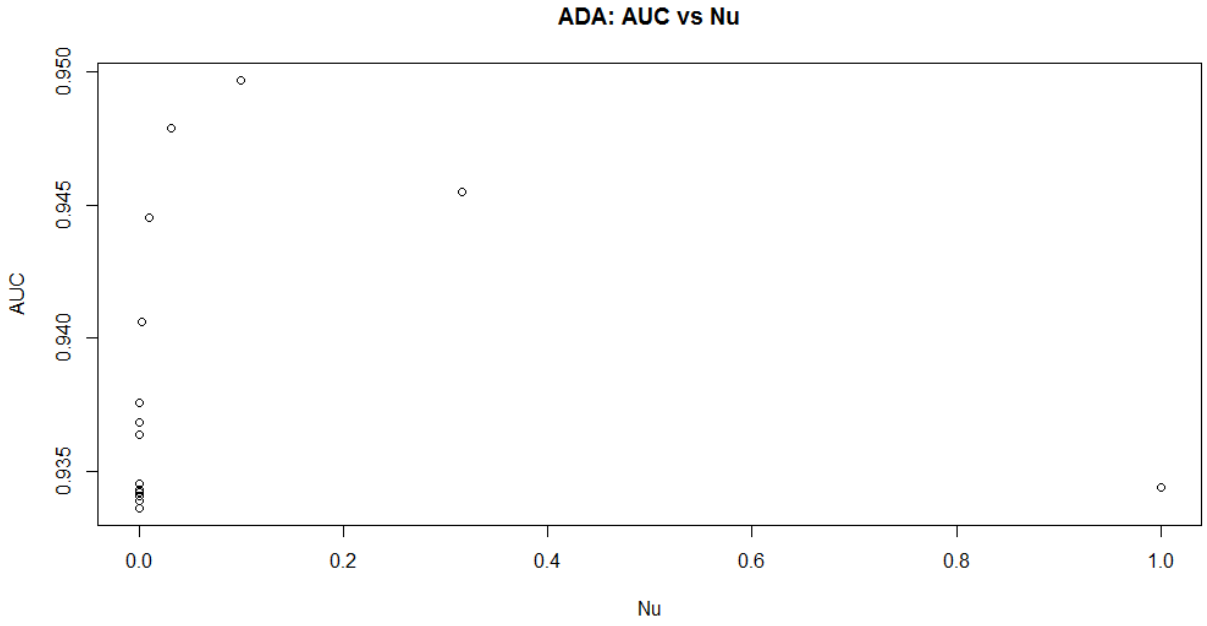



From the final part of the code in the previous section we get the best cost and gamma which are:

**Cost=10, Gamma = 0.0003162278**

## ADA BOOST

Using the R code provided in the previous section and AUC result values in Appendix 1, we generate the following scatter plot of AUC vs Nu. There is a clear trend as to where the maximum is (0.1).

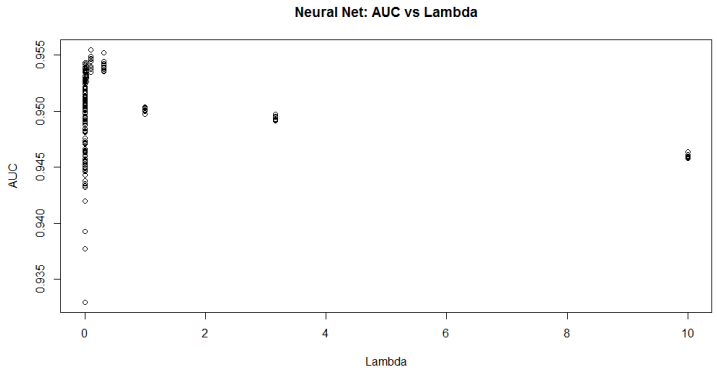


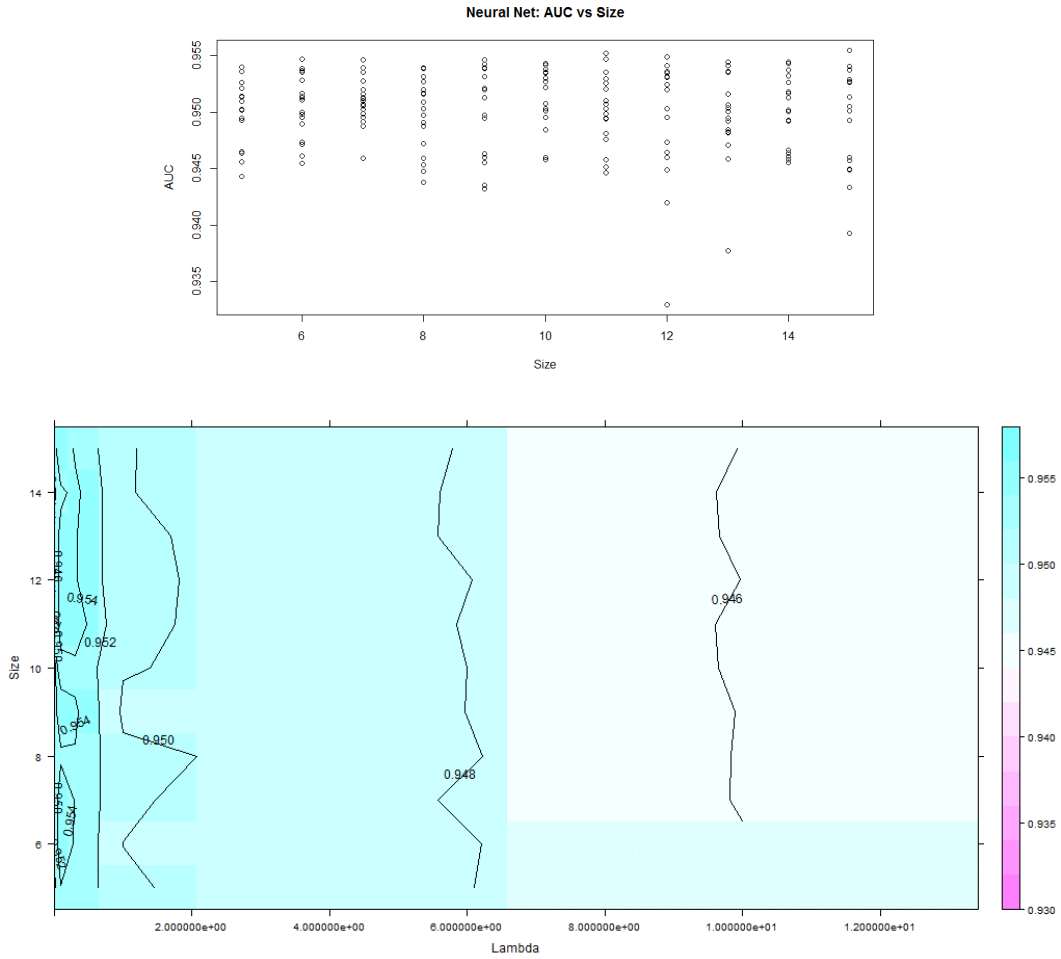

From the final part of the code in the previous section we get the best Nu which is:

**Nu = 0.1**

## NEURAL NETWORK

Using the R code provided in the previous section and AUC result values in Appendix 1, we generate the following scatter plot of AUC vs Lambda, AUC vs Size and the contour plot of AUC vs both. Lambda seems to have the strongest impact of the AUC, where values just above zero offer the best AUC. Size seems to matter less and has negligeable impact on AUC. Once again, the contour plot's right side gradient refers to AUC as it changes with Lambda and Size, as does the contour values.



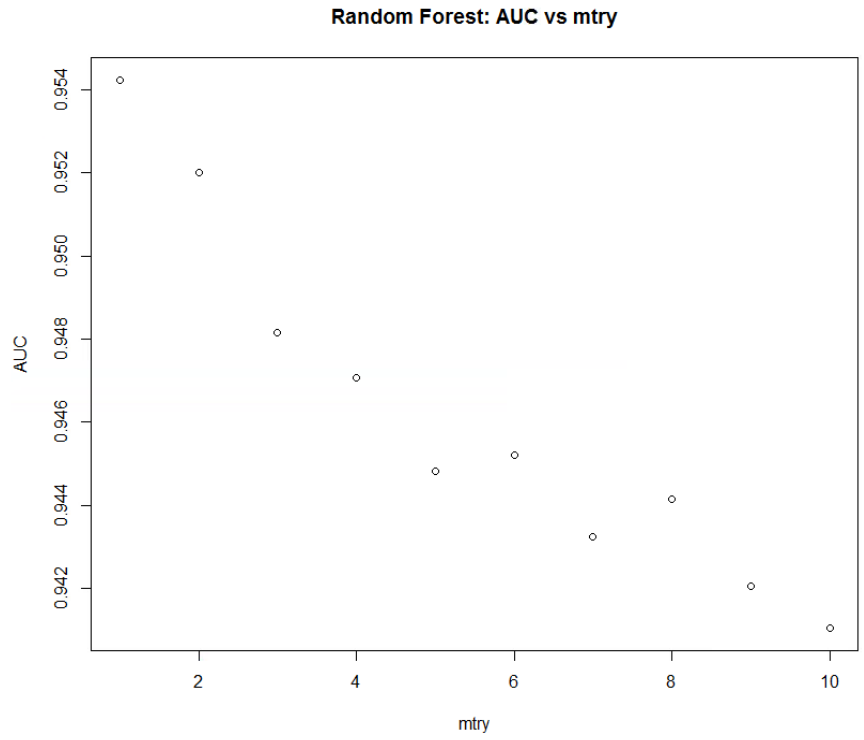



From the final part of the code in the previous section we get the best cost and gamma which are:

**Size = 10, Lambda = 0.1**

## RANDOM FOREST

Using the R code provided in the previous section and AUC result values in Appendix 1, we generate the following scatter plot of AUC vs mtry. Mtry seems have a strong impact on the AUC, where lower values offer the best AUC.

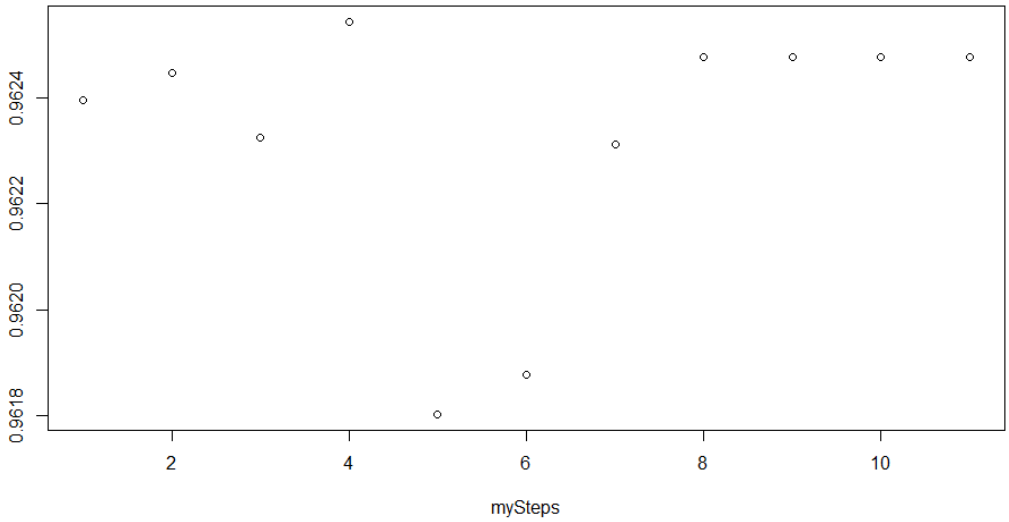



From the final part of the code in the previous section we get the best mtry which is:

**mtry = 1**

## LOGISTIC (PROBIT) REGRESSION

Using the R code provided in the previous section and AUC result values in Appendix 1, we generate the following scatter plot of AUC vs number of steps. Smaller values (mainly 4) and larger values (above 7) offer the best AUC. This refers to the number of steps we iterate through stepwise AIC regression in both directions.




From the final part of the code in the previous section we get the best steps value which is:

**Steps = 4**

## STACKING

The following R code iterates through 10 fold cross validation once again with the selected models on all the original training (AngleClosure.csv) data. In each iteration we fit and predict each model for the testing fold data and we then store this as well as the corresponding response to assemble our model matrix for the constrained and unconstrained optimizations.

```
#define number of folds and iterations
nFolds=10
nIter=50

count = 0
iter = 1
n = 1
#create dummy matrices to which we will append and remove dummy rows
X = matrix(0,,5)
Y = matrix(0,,1)

for(iter in 1:nIter){
  testingIndices=sample(length(myLabel))[1:round(length(myLabel)/nFolds)]
  myDataTesting=dataStar[testingIndices,]
  myDataTraining=dataStar[-testingIndices,]

  subX = matrix(0,dim(myDataTesting)[1],)

  #SVM fit/predict/store
  fitSVM=svm(myLabel~.,data=myDataTraining, cost=10, gamma = 0.0003162278,
probability = TRUE)
  myPreds= predict(fitSVM,newdata=myDataTesting, probability = TRUE)
  probabilities = attr(myPreds,"probabilities")[,1]
  subX = cbind(subX,probabilities)

  #ADA fit/predict/store
  fitADA=ada(myLabel~.,data=myDataTraining, nu=0.1)
  myPreds= predict(fitADA,newdata=myDataTesting, type = "probs")[,2]

  subX = cbind(subX,myPreds)

  #NNET fit/predict/store
  fitNNET=nnet(myLabel~.,data=myDataTraining,size=10,decay=0.1)
  myPreds=predict(fitNNET,newdata=myDataTesting, probability = TRUE)
  subX = cbind(subX,myPreds)
  #RF fit/predict/store
  fitRF=randomForest(myLabel~.,data=myDataTraining, mtry = 1)
  myPreds= predict(fitRF,newdata=myDataTesting, type = "prob")[,2]
  subX = cbind(subX,myPreds)
```

```
#GLM fit/predict/store
  test = glm(myLabel~.,data=myDataTraining,family=binomial(link=probit))
  step = step(test, direction = "both", steps = 4)
  newDataTraining = step$model
  fitGLM=glm(myLabel~.,data=newDataTraining,family=binomial(link=probit))
  myPreds=predict(fitGLM,newdata=myDataTesting,family=binomial(link=probit), type =
"response")

  subX = cbind(subX,myPreds)
  #append to Y, response, and X, predictors
  Y = rbind(Y,as.matrix(myDataTesting$myLabel))
  X = rbind(X, subX[,2:6])
}

Ycopy = Y
Xcopy = X

Ycopy = as.numeric(Ycopy[2:dim(Ycopy)[1],])
Xcopy = Xcopy[2:dim(Xcopy)[1],]
#set up quadratic constrained optimization problem
Dmat = t(Xcopy)%*%Xcopy
dvec = t(Ycopy)%*%Xcopy

C <- cbind(rep(1,5), diag(5))
b <- c(1,rep(0,5))
#solve and get weights
weightsConstrained = as.numeric(solve.QP(Dmat = Dmat, dvec = dvec, Amat = C, bvec =
b, meq = 1)$solution)
#set up regression model to represent the unconstrained case, no intercept
regression = lm(Ycopy~ 0 + Xcopy)
#get weights
weightsUnConstrained = summary(regression)$coef[,1]
```

THE WEIGHT OPTIMIZATION RESULTS ARE AS FOLLOWS:

| ***Stacked Model*** | **SVM** | **ADA** | **NNET** | **RF** | **REGRESSION** |
|---|---|---|---|---|---|
| *Constrained* | 0.4391 | 0.1338 | -5.111e-18 | 4.004e-02 | 0.3871 |
| *Unconstrained* | 0.4459 | 0.1128 | -0.03761 | 0.1118 | 0.3923 |

Thus, we can see both put quite a high weight particularly on SVM and the Regression model, with slightly less weight on ADA.

## VALIDATION - FINAL AUC AND ROC CURVES (Q5 AND Q6)

Using the following R code, we first manipulated the Cases and Controls data to ensure we only have the necessary columns and that we preferentially choose right eye data over left. Then we generate predictions with each model trained with the complete initial data set (AngleClosure.csv).

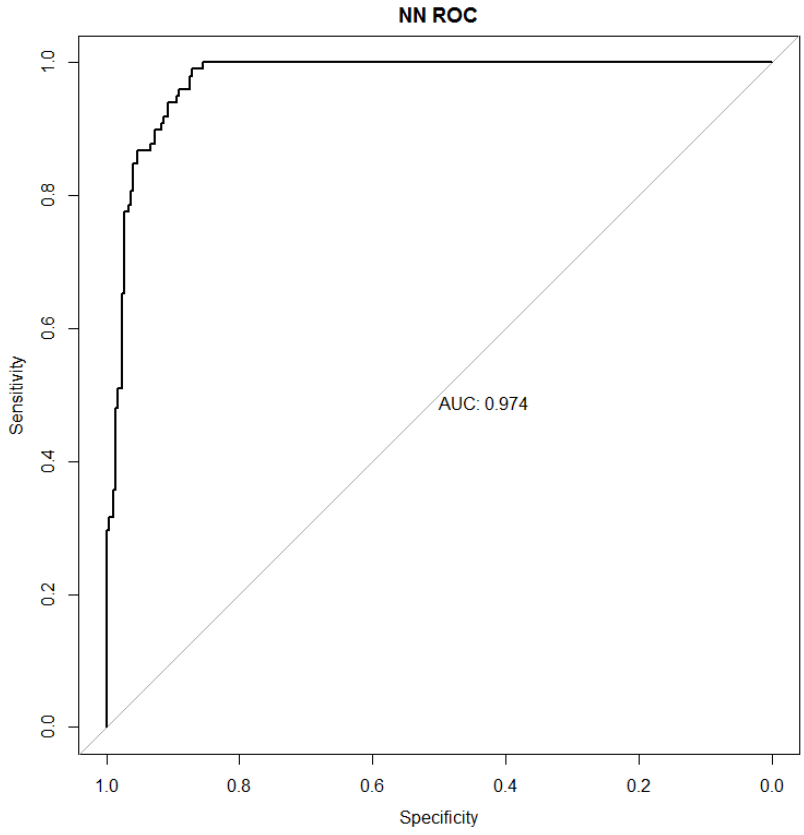
```
#get left and right cases and controls corresponding to each other for comparison purposes
myCasesNewR = myCases[,c(19,21,23,24,25,26,27,31,32,36)]
myCasesNewL = myCases[,c(7,9,11,12,13,14,15,30,31,32,36)]
myCasesNewR = myCases[,c(19,21, 23:27,30:32,36)]
myControlsNewL = myControls[,c(6,8,10:14,29:31,35)]
myControlsNewR = myControls[,c(6,8,10,11,12,13,14,29,30,31,35)]

myCasesFinal = matrix(NA,dim(myCasesNewL)[1],dim(myCasesNewL)[2])

myControlsFinal = matrix(NA,dim(myControlsNewL)[1],dim(myControlsNewL)[2])

#for each row and column of the ultimate controls matrix
for (i in 1:dim(myControlsNewL)[1])
{
  for (j in 1:dim(myControlsNewL)[2])
  {
    #if no right value, put left, if right, put right, leave NA otherwise
    if(is.na(myControlsNewR[i,j]) && !is.na(myControlsNewL[i,j]))
    {
      myControlsFinal[i,j] = myControlsNewL[i,j]
    } else if (!is.na(myControlsNewR[i,j])){
      myControlsFinal[i,j] = myControlsNewR[i,j]
    }
  }
}

#for each row and column of the ultimate cases matrix
for (i in 1:dim(myCasesNewL)[1])
{
  for (j in 1:dim(myCasesNewL)[2])
  {
    #if no right value, put left, if right, put right, leave NA otherwise
    if(is.na(myCasesNewR[i,j]) && !is.na(myCasesNewL[i,j]))
    {
      myCasesFinal[i,j] = myCasesNewL[i,j]
    } else if (!is.na(myCasesNewR[i,j])){
      myCasesFinal[i,j] = myCasesNewR[i,j]
    }
  }
}

myCasesFinal2 <- myCasesFinal[rowSums(is.na(myCasesFinal))==0,]

caseY = matrix(1,dim(myCasesFinal2)[1],1)
controlY = matrix(0,dim(myControlsFinal)[1],1)

finalV_X = rbind(myControlsFinal, myCasesFinal2)
```

```
myLabelY = as.factor(rbind(controlY, caseY))
#assemble final validation data
finalData = data.frame(myLabelY, finalV_X)
colnames(finalData) = colnames(dataStar)

#SVM - fit/predict/get ROC and AUC using original training data
fit=svm(myLabel~.,data=dataStar, cost=10, gamma = 0.0003162278, probability = TRUE)
myPredsSVM= predict(fit,newdata=finalData, probability = TRUE)
probabilities = attr(myPredsSVM,"probabilities")[,1]
ROC_SVM = roc(as.numeric(myLabelY),as.numeric(probabilities))
AUC_SVM = ROC_SVM$auc

#ADA - fit/predict/get ROC and AUC
fit=ada(myLabel~.,data=dataStar, nu=0.1)
myPredsADA= predict(fit,newdata=finalData, type = "probs")[,2]
ROC_ADA = roc(as.numeric(myLabelY),as.numeric(myPredsADA))
AUC_ADA = ROC_ADA$auc

#NNET - fit/predict/get ROC and AUC
fitNN=nnet(myLabel~.,data=dataStar,size=10,decay=0.1)
myPredsNN=predict(fitNN,newdata=finalData, probability = TRUE)
ROC_NN = roc(as.numeric(myLabelY),as.numeric(myPredsNN))
AUC_NN = ROC_NN$auc

#RF - fit/predict/get ROC and AUC
fit=randomForest(myLabel~.,data=dataStar, mtry = 1)
myPredsRF= predict(fit,newdata=finalData, type = "prob")[,2]
ROC_RF = roc(as.numeric(myLabelY),as.numeric(myPredsRF))
AUC_RF = ROC_RF$auc

#GLM - fit/predict/get ROC and AUC
test = glm(myLabel~.,data=dataStar,family=binomial(link=probit))
step = step(test, direction = "both", steps = 4)
newdataStar = step$model
fit=glm(myLabel~.,data=newdataStar,family=binomial(link=probit))
fit=glm(myLabel~.,data=dataStar,family=binomial)
myPredsGLM=predict(fit,newdata=finalData,family=binomial(link=probit), type = "response")
ROC_GLM = roc(as.numeric(myLabelY),as.numeric(myPredsGLM))
AUC_GLM = ROC_GLM$auc
#prepare columns of all prection models for stacking predictions
P = probabilities
P = cbind(P, myPredsADA)
P = cbind(P, myPredsNN)
P = cbind(P, myPredsRF)
P = cbind(P, myPredsGLM)
#use the P matrix and the weights of constrained and unconstrained to get predictions
#get ROC and AUC after
#stacked constrained
myPredsStackedC = P%*%weightsConstrained
ROC_SC = roc(as.numeric(myLabelY),as.numeric(myPredsStackedC))
AUC_SC = ROC_SC$auc
#stacked unconstrained
myPredsStackedU = P%*%weightsUnConstrained
ROC_SU = roc(as.numeric(myLabelY),as.numeric(myPredsStackedU))
AUC_SU = ROC_SU$auc
```

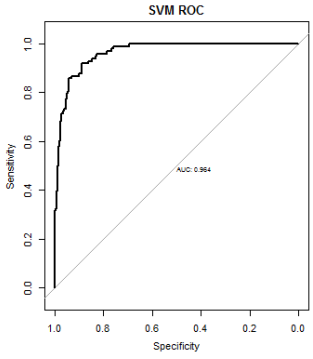
| *Model* | AUC |
|---|---|
| *Support Vector Machine* | 0.9640 |
| *ADA Boost* | 0.9609 |
| *Neural Network* | **0.9738** |
| *Random Forest* | 0.9644 |
| *Regression* | 0.9635 |
| *Stacked Constrained* | **0.9665** |
| *Stacked Unconstrained* | 0.9655 |

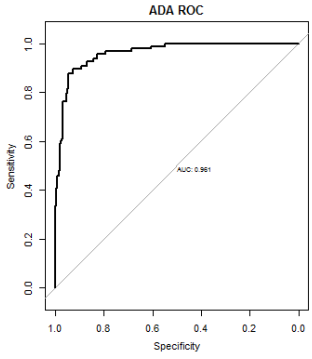
As we can see, the best model generated in terms of AUC performance on the validation data set was the **Neural Network with an AUC of 0.9738**, followed by the **Stacked Constrained model with and AUC of 0.9665**. It is interesting to note that the stacked models did not put much weight on the Neural Network in the optimization, but it ends up giving the best AUC.

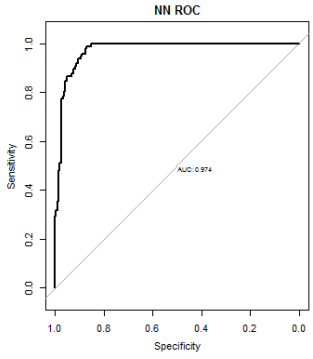
This is the ROC plot for the best model, the Neural Network:

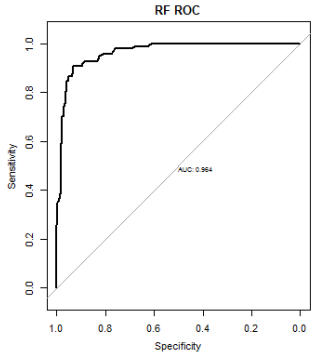


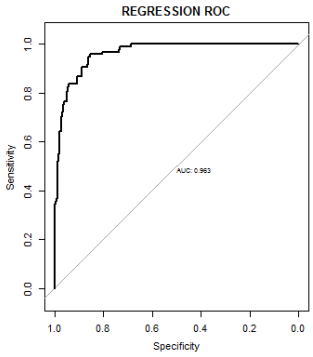
The following are ROC plots for each model:

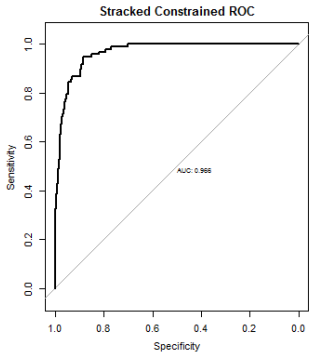


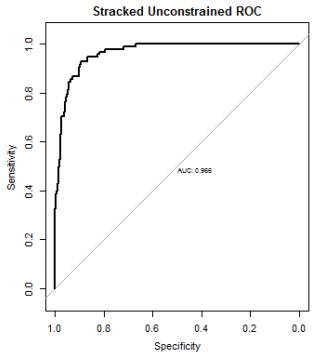



# APPENDIX

## SVM AUC VS VALUES TESTED

| Gamma | Costs | AUC |
|---|---|---|
| **1.00E-06** | 1.00E-02 | 0.954681 |
| **3.16E-06** | 1.00E-02 | 0.954773 |
| **1.00E-05** | 1.00E-02 | 0.954752 |
| **3.16E-05** | 1.00E-02 | 0.95489 |
| **1.00E-04** | 1.00E-02 | 0.954948 |
| **3.16E-04** | 1.00E-02 | 0.954898 |
| **1.00E-03** | 1.00E-02 | 0.955182 |
| **3.16E-03** | 1.00E-02 | 0.95482 |
| **1.00E-02** | 1.00E-02 | 0.955033 |
| **3.16E-02** | 1.00E-02 | 0.954244 |
| **1.00E-01** | 1.00E-02 | 0.946171 |
| **1.00E-06** | 1.00E-01 | 0.954695 |
| **3.16E-06** | 1.00E-01 | 0.95484 |
| **1.00E-05** | 1.00E-01 | 0.954843 |
| **3.16E-05** | 1.00E-01 | 0.954856 |
| **1.00E-04** | 1.00E-01 | 0.955053 |
| **3.16E-04** | 1.00E-01 | 0.954795 |
| **1.00E-03** | 1.00E-01 | 0.955075 |
| **3.16E-03** | 1.00E-01 | 0.95497 |
| **1.00E-02** | 1.00E-01 | 0.955291 |
| **3.16E-02** | 1.00E-01 | 0.954753 |
| **1.00E-01** | 1.00E-01 | 0.945311 |
| **1.00E-06** | 1.00E+00 | 0.954809 |
| **3.16E-06** | 1.00E+00 | 0.95481 |
| **1.00E-05** | 1.00E+00 | 0.954961 |
| **3.16E-05** | 1.00E+00 | 0.955003 |
| **1.00E-04** | 1.00E+00 | 0.955123 |
| **3.16E-04** | 1.00E+00 | 0.954989 |
| **1.00E-03** | 1.00E+00 | 0.955405 |
| **3.16E-03** | 1.00E+00 | 0.955607 |
| **1.00E-02** | 1.00E+00 | 0.955067 |
| **3.16E-02** | 1.00E+00 | 0.952954 |
| **1.00E-01** | 1.00E+00 | 0.930494 |
| **1.00E-06** | 1.00E+01 | 0.955038 |

| | | |
|---|---|---|
| **3.16E-06** | 1.00E+01 | 0.954787 |
| **1.00E-05** | 1.00E+01 | 0.954992 |
| **3.16E-05** | 1.00E+01 | 0.955056 |
| **1.00E-04** | 1.00E+01 | 0.955221 |
| **3.16E-04** | 1.00E+01 | 0.955732 |
| **1.00E-03** | 1.00E+01 | 0.955492 |
| **3.16E-03** | 1.00E+01 | 0.955177 |
| **1.00E-02** | 1.00E+01 | 0.953331 |
| **3.16E-02** | 1.00E+01 | 0.940669 |
| **1.00E-01** | 1.00E+01 | 0.914333 |
| **1.00E-06** | 1.00E+02 | 0.955022 |
| **3.16E-06** | 1.00E+02 | 0.955035 |
| **1.00E-05** | 1.00E+02 | 0.955211 |
| **3.16E-05** | 1.00E+02 | 0.955777 |
| **1.00E-04** | 1.00E+02 | 0.955511 |
| **3.16E-04** | 1.00E+02 | 0.95543 |
| **1.00E-03** | 1.00E+02 | 0.955192 |
| **3.16E-03** | 1.00E+02 | 0.953353 |
| **1.00E-02** | 1.00E+02 | 0.943929 |
| **3.16E-02** | 1.00E+02 | 0.927694 |
| **1.00E-01** | 1.00E+02 | 0.904367 |

## ADA: AUC VS VALUES TESTED

| NU | 1.00E-07 | 3.16E-07 | 1.00E-06 | 3.16E-06 | 1.00E-05 | 3.16E-05 | 1.00E-04 | 3.16E-04 | 1.00E-03 | 3.16E-03 | 1.00E-02 | 3.16E-02 | 1.00E-01 | 3.16E-01 | 1.00E+00 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| AUC | 0.9363701 | 0.934215 | 0.9340735 | 0.9343125 | **0.9336347** | 0.9338952 | 0.9345385 | 0.9375675 | 0.936832 | 0.940598 | 0.944509 | 0.947871 | 0.949664 | 0.945498 | 0.934387 |

## NEURAL NETWORK: AUC VS VALUES TESTED

| Lambda | Size | AUC |
|---|---|---|
| **1.00E-05** | 7 | 0.953543 |
| **1.00E-04** | 7 | 0.950804 |
| **1.00E-03** | 7 | 0.953821 |
| **1.00E-02** | 7 | 0.95484 |
| **1.00E-01** | 7 | 0.956298 |
| **1.00E+00** | 7 | 0.952521 |
| **1.00E+01** | 7 | 0.949273 |
| **1.00E+02** | 7 | 0.936669 |
| **1.00E-05** | 8 | 0.949537 |
| **1.00E-04** | 8 | 0.955669 |
| **1.00E-03** | 8 | 0.954776 |
| **1.00E-02** | 8 | 0.95631 |
| **1.00E-01** | 8 | 0.954788 |
| **1.00E+00** | 8 | 0.953117 |
| **1.00E+01** | 8 | 0.949575 |
| **1.00E+02** | 8 | 0.936908 |
| **1.00E-05** | 9 | 0.949609 |
| **1.00E-04** | 9 | 0.951548 |
| **1.00E-03** | 9 | 0.952876 |
| **1.00E-02** | 9 | 0.954775 |
| **1.00E-01** | 9 | 0.956099 |
| **1.00E+00** | 9 | 0.952744 |
| **1.00E+01** | 9 | 0.949261 |
| **1.00E+02** | 9 | 0.936906 |
| **1.00E-05** | 10 | 0.950312 |
| **1.00E-04** | 10 | 0.950205 |
| **1.00E-03** | 10 | 0.955023 |
| **1.00E-02** | 10 | 0.955337 |
| **1.00E-01** | **10** | **0.95658** |
| **1.00E+00** | 10 | 0.952564 |

| | | |
|---|---|---|
| **1.00E+01** | 10 | 0.949374 |
| **1.00E+02** | 10 | 0.936917 |
| **1.00E-05** | 11 | 0.951857 |
| **1.00E-04** | 11 | 0.952818 |
| **1.00E-03** | 11 | 0.95514 |
| **1.00E-02** | 11 | 0.955808 |
| **1.00E-01** | 11 | 0.955635 |
| **1.00E+00** | 11 | 0.952585 |
| **1.00E+01** | 11 | 0.94923 |
| **1.00E+02** | 11 | 0.936937 |
| **1.00E-05** | 12 | 0.95105 |
| **1.00E-04** | 12 | 0.949128 |
| **1.00E-03** | 12 | 0.953974 |
| **1.00E-02** | 12 | 0.954968 |
| **1.00E-01** | 12 | 0.95567 |
| **1.00E+00** | 12 | 0.952958 |
| **1.00E+01** | 12 | 0.9496 |
| **1.00E+02** | 12 | 0.936867 |

## RANDOM FOREST: AUC VS VALUES TESTED

| mTry | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| **AUC** | 0.954236 | 0.952 | 0.94815 | 0.947075 | 0.944815 | 0.945197 | 0.943243 | 0.944137 | 0.942055 | 0.941043 |

## LOGISTIC (PROBIT) REGRESSION: AUC VS VALUES TESTED

| K | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **AUC** | 0.9623 | 0.9624 | 0.9623 | **0.9625** | 0.9618 | 0.9618 | 0.9623 | 0.9624 | 0.9624 | 0.9624 | 0.9624 |