Ch4Report

Madi Kassymbekov

Question 1

```
#Slide 28 code to fit gbm on Ames data
library(gbm)
```

```
## Loaded gbm 2.1.8
```

```
set.seed(33967)
gbmgc=gbm(Sale_Price~.,data=amestrain,distribution="gaussian",
          n.trees=100,interaction.depth = 5,shrinkage =0.1)
```

1) By referring to the algorithm presented in slide 22, explain what the value of the initF component represents

```
print(sprintf("initF: %f",gbmgc$initF))
```

```
## [1] "initF: 179.894944"
```

```
initF <- gbmgc$initF
```

Answer: initF is the initial predicted value according to which adjustments (boosting) are made. For ames example the initial value of residuals (intercept term) is 179.894944

2) What does the component fit represent ? Use it to compute the MSE in the training sample

```
print(sprintf("MSE of fitted values: %f",
              gbmgcMSE <- mean((gbmgc$fit-amestrain$Sale_Price)^2)))
```

```
## [1] "MSE of fitted values: 218.691773"
```

```
fit.error <- mean((gbmgc$fit-amestrain$Sale_Price)^2)
```

Answer: fit is the vector of fitted values of L2-boosting algorithm on the training set. MSE calculated from fitted and actual values on the training set is 218.6917732.

3) What does the component train.error represent ? Explain the link with the MSE value computed in 2)

```
print(sprintf("Train error of last 100th iteration: %f",
              gbmgc$train.error[100]))
```

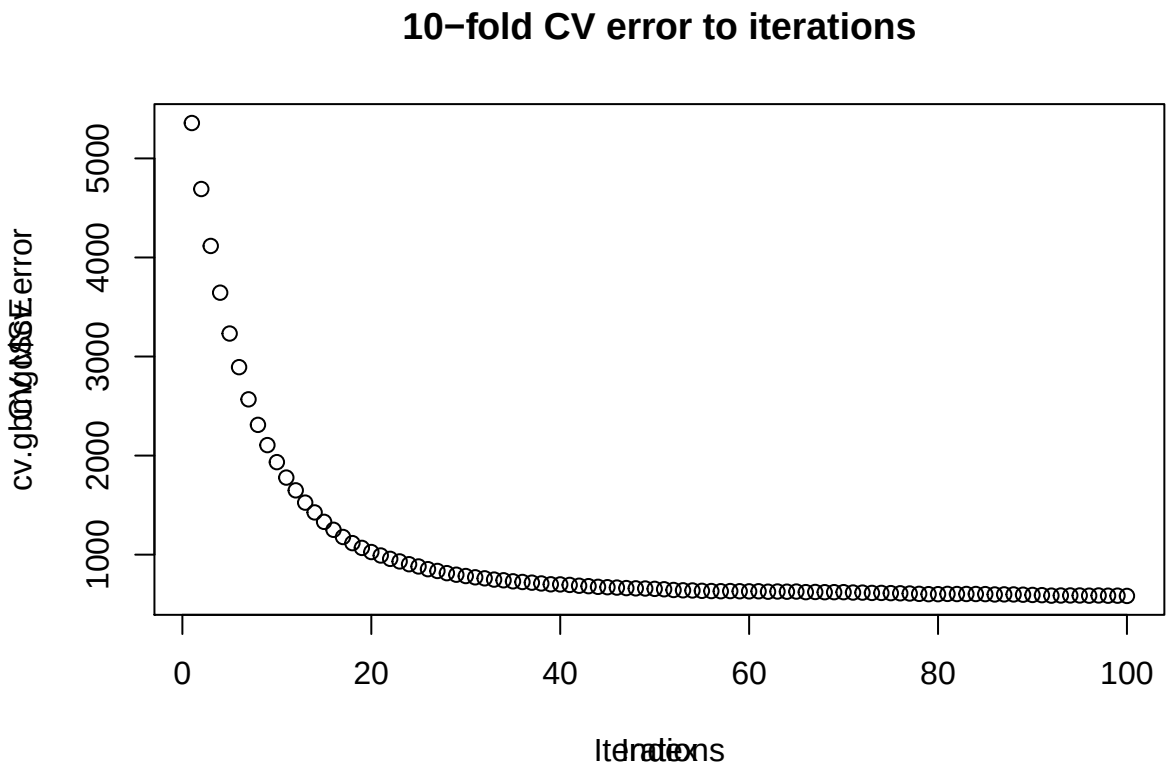
```
## [1] "Train error of last 100th iteration: 218.691773"
```

Answer: train.error is the vector of all MSE errors calculated for each tree/boosting iteration. Each boosting iteration improves/shrinks MSE value. As we specified 100 trees/iterations for boosting, last 100th final MSE of train.error is equal to our calculated MSE from fit vector.

4) We mentioned in slide 32 that gbm allows to estimate the error with cross validation. By default (which was the case in the analysis presented in the notes), no cross-validation is performed. Use the option cv.folds in the function to perform a 10-fold CV and use the results

in the component cv.error to plot the error as a function of the iterations (trees). What can you say about the optimal number of trees and comment about our choice to use 100 trees in our analysis.

```
cv.gbmgc=gbm(Sale_Price~.,data=amestrain,distribution="gaussian",
           n.trees=100,interaction.depth = 5,shrinkage =0.1, cv.folds=10)
plot(cv.gbmgc$cv.error)
title(main="10-fold CV error to iterations",
   xlab="Iterations", ylab="CV MSE")
```



Answer: By looking at the results and plot of cv.error, it is obvious that 100 trees are more than needed and model will tend to overfit due to increased model complexity. Based on the plot results, MSE stabilizes around 40 iterations/trees which suggests to be an optimal number of trees for boosting.

5) From the results obtained in 4), what is the final cross-validated MSE of the model? Compare this value to the MSE computed in 2) and comment.

```
print(sprintf("CV error: %f, iteration: %i",min(cv.gbmgc$cv.error),
              which(cv.gbmgc$cv.error==min(cv.gbmgc$cv.error))))
```

```
## [1] "CV error: 583.425369, iteration: 100"
```

```
cv.error <- min(cv.gbmgc$cv.error)
```

Answer: Final CV MSE error is 583.4253687 which is much higher than MSE of 218.6917732 obtained in step 2. However, it should be noted that MSE obtained in step 2 is on training set, while CV error is done on different validation folds to better represent generalization error, and by definition training errors are much lower than validation error as model is learning on the training dataset.

6) Using the cross validation results in 4), explain the difference between the component

cv.fitted and the component fit from the analysis without the cross validation. Use the cv.fitted to recompute the cross-validation error obtained in 5).

```
print(sprintf("CV MSE calculated error: %f",
              mean((cv.gbmgc$cv.fitted-amestrain$Sale_Price)^2)))
```

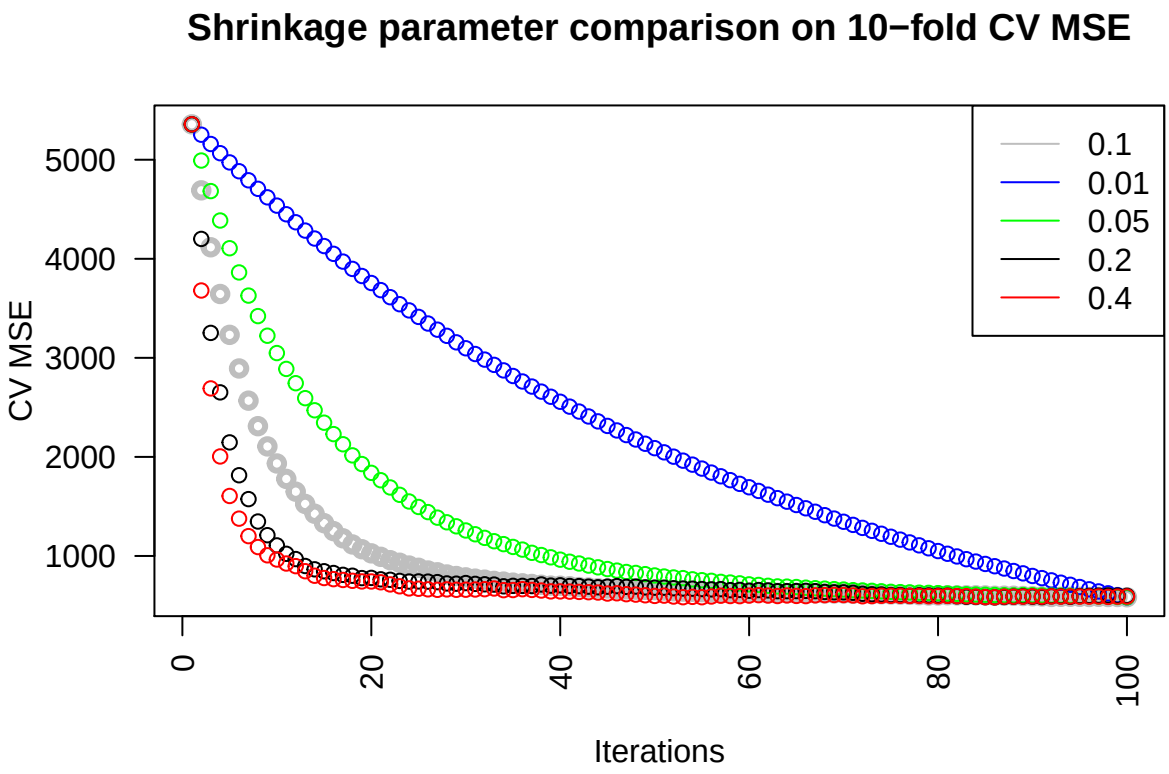
```
## [1] "CV MSE calculated error: 583.425369"
```

Answer: cv.fitted are the values fitted on the validation fold based on training model on other training folds, while fitted values are the values fit on the whole training dataset, therefore cv.fitted are generalizable fitted values and the residuals will be bigger compared to train fitted values. Recomputed MSE based on cv.fitted is equal to the best iteration of cv.error that was seen in step 5.

7) Draw a graph showing the 10-fold cross validation error of the model as a function of the number of trees (like you did in 4), for different values of the shrinkage parameter epsilon (use 0.01, 0.05, 0.1, 0.2 and 0.3). Comment your results and choose the optimal value of the shrinkage parameter.

```
cv.gbmgc1=gbm(Sale_Price~.,data=amestrain,distribution="gaussian",
           n.trees=100,interaction.depth = 5,shrinkage =0.01, cv.folds=10)
cv.gbmgc2=gbm(Sale_Price~.,data=amestrain,distribution="gaussian",
           n.trees=100,interaction.depth = 5,shrinkage =0.05, cv.folds=10)
cv.gbmgc3=gbm(Sale_Price~.,data=amestrain,distribution="gaussian",
           n.trees=100,interaction.depth = 5,shrinkage =0.2, cv.folds=10)
cv.gbmgc4=gbm(Sale_Price~.,data=amestrain,distribution="gaussian",
           n.trees=100,interaction.depth = 5,shrinkage =0.3, cv.folds=10)
```

```
plot(cv.gbmgc$cv.error, lwd=3, col="grey", ann=FALSE, las=2)
par(new=TRUE)
plot(cv.gbmgc1$cv.error, ann=FALSE, axes=FALSE,col='blue')
par(new=TRUE)
plot(cv.gbmgc2$cv.error, ann=FALSE, axes=FALSE,col='green')
par(new=TRUE)
plot(cv.gbmgc3$cv.error, ann=FALSE, axes=FALSE,col='black')
par(new=TRUE)
plot(cv.gbmgc4$cv.error, ann=FALSE, axes=FALSE,col='red')
legend("topright", legend=c("0.1", "0.01", "0.05","0.2","0.4"),
       col=c("grey", "blue", "green", "black", "red"), lty=1)
title(main="Shrinkage parameter comparison on 10-fold CV MSE",
   xlab="Iterations", ylab="CV MSE")
```



Answer: Based on graph comparison for different shrinkage parameter, all of the models reach similar MSE with 100 iterations. However, it is noticeable that lower shrinkage rate requires more iterations to reach optimal MSE. It can be noticed, that shrinkage 0.4 starts overfitting near 100 iterations as the MSE starts slowly to increase. Considering current iteration number of 100, 0.01 is the optimal shrinkage parameter for 100 iterations as MSE keeps decreasing across all iterations while others reached optimality much earlier. However, if we consider to find optimality with less iterations, either 0.1 or 0.05 will be better for such a task with optimal number of iterations ranging from 40 to 70 iterations. Slow learning is preferred for a model to capture relevant features and representation of the dataset for better generalization on the test and validation sets.

Question 2

Use the following methods and compare the misclassification rate:

```
trainset <- read.csv("data/ionosphere_traindata.csv", header = TRUE)
testset <- read.csv("data/ionosphere_testdata.csv", header = TRUE)
trainset$Class <- as.factor(ifelse(trainset$Class==-1, 0, trainset$Class))
testset$Class <- as.factor(ifelse(testset$Class==-1, 0, testset$Class))
```

1) Single tree (pruned)

```
library(rpart)
rptree <- rpart(Class~., data=trainset, method="class",
        control = rpart.control(xval = 10, minsplit=10, minbucket = 3, cp = 0))
rptreepruned=prune(rptree, cp=rptree$cp[which.min(rptree$cp[,"xerror"]),"CP"])
predrpart=predict(rptreepruned,newdata=testset,type="class")
results <- data.frame("Single tree (pruned)", mean(predrpart!=testset$Class))
names(results) <- c("Model", "Error_Rate")
```

```
knitr::kable(results, caption="Single tree (pruned) Model Performance")
```

Table 1: Single tree (pruned) Model Performance

Model	Error_Rate
Single tree (pruned)	0.1324503

2) Random forest with 100, 200, 300, 400, 500 trees. Select the optimal number of trees.

```
library(randomForest)
```

```
## randomForest 4.6-14
```

```
## Type rfNews() to see new features/changes/bug fixes.
```

```
opttrees <- 0
opterror <- 9999999999
ntrees <- c(100, 200, 300, 400, 500)
for (tree in ntrees) {
  rf=randomForest(Class~.,data=trainset,ntree=tree)
  predrf=predict(rf, newdata=testset)
  print(sprintf("RF ntrees %f, error %f", tree, mean(predrf!=testset$Class)))
  if (mean(predrf!=testset$Class) < opterror) {
    opterror <- mean(predrf!=testset$Class)
    opttrees <- tree
  }
}
```

```
## [1] "RF ntrees 100.000000, error 0.079470"
## [1] "RF ntrees 200.000000, error 0.092715"
## [1] "RF ntrees 300.000000, error 0.092715"
## [1] "RF ntrees 400.000000, error 0.099338"
## [1] "RF ntrees 500.000000, error 0.086093"
```

```
results[nrow(results) + 1,] = list(Model=paste(c("Random forest optimal trees",
                                                 opttrees),collapse=" "),
                                   Error_Rate=opterror)
knitr::kable(results[2,], caption="Random forest optimal trees Model
             Performance")
```

Table 2: Random forest optimal trees Model Performance

	Model	Error_Rate
2	Random forest optimal trees 100	0.0794702

3) Adaboost based on a stump with 100 trees

```
library(adabag)
```

```
## Loading required package: caret
```

```
## Loading required package: lattice
```

```
## Loading required package: ggplot2
```

```
##
```

```
## Attaching package: 'ggplot2'

## The following object is masked from 'package:randomForest':
##
##     margin

## Loading required package: foreach

## Loading required package: doParallel

## Loading required package: iterators

## Loading required package: parallel
```

```
boostgc1=boosting(Class~.,data=trainset,boos = FALSE, mfinal = 100,
                coeflearn = 'Freund', control=rpart.control(maxdepth=1))
predboostgc1=predict(boostgc1, newdata=testset)
results[nrow(results) + 1,] = list(Model="Adaboost stump 100 trees",
                            Error_Rate=mean(predboostgc1$class!=testset$Class))
knitr::kable(results[3,], caption="Adaboost stump 100 trees Model Performance")
```

Table 3: Adaboost stump 100 trees Model Performance

	Model	Error_Rate
3	Adaboost stump 100 trees	0.0927152

4) Adaboost based on trees with maxdepth=5 with 100 trees

```
boostgc5=boosting(Class~.,data=trainset,boos = FALSE, mfinal = 100,
                  coeflearn = 'Freund', control=rpart.control(maxdepth=5))
predboostgc5=predict(boostgc5, newdata=testset)
results[nrow(results) + 1,] = list(Model="Adaboost depth no stump 100 trees",
                            Error_Rate=mean(predboostgc5$class!=testset$Class))
knitr::kable(results[4,], caption="Adaboost depth 5 100trees Model Performance")
```

Table 4: Adaboost depth 5 100trees Model Performance

	Model	Error_Rate
4	Adaboost depth no stump 100 trees	0.0794702

5) Gradient boosting based on a stump with 100 trees using a shrinkage parameter of (0.01, 0.05, 0.1, 0.2). Select the optimal shrinkage.

```
optshrink <- 0
opterror <- 1
shrinks <- c(0.01, 0.05, 0.1, 0.2)
for (shrink in shrinks) {
  gbmfit=gbm(as.character(Class)~.,data=trainset,distribution = "bernoulli",
        n.trees = 100, interaction.depth = 1, shrinkage = shrink)
  predgbm=predict(gbmfit, newdata=testset, n.trees = 100, type="response")
  predgbm <- ifelse(predgbm>=0.5,1,0)
  print(sprintf("GBM stump shrink %f, error %f", shrink,
                mean(predgbm!=testset$Class)))
  if (mean(predgbm!=testset$Class) < opterror) {
    opterror <- mean(predgbm!=testset$Class)
```

```
    optshrink <- shrink
  }
}
```

```
## [1] "GBM stump shrink 0.010000, error 0.185430"
## [1] "GBM stump shrink 0.050000, error 0.086093"
## [1] "GBM stump shrink 0.100000, error 0.086093"
## [1] "GBM stump shrink 0.200000, error 0.092715"
```

```
results[nrow(results) + 1,] = list(Model=paste(
c("Gradient boosting stump optimal shrinkage",optshrink),collapse=" "),
      Error_Rate=opterror)
knitr::kable(results[5,],
caption="Gradient boosting stump optimal shrinkage Model Performance")
```

Table 5: Gradient boosting stump optimal shrinkage Model Performance

	Model	Error_Rate
5	Gradient boosting stump optimal shrinkage 0.05	0.0860927

6) Gradient boosting on trees with maxdepth=5 with 100 trees using a shrinkage parameter of (0.01, 0.05, 0.1, 0.2). Select the optimal shrinkage.

```
optshrink <- 0
opterror <- 1
shrinks <- c(0.01, 0.05, 0.1, 0.2)
for (shrink in shrinks) {
  gbmfit=gbm(as.character(Class)~.,data=trainset,
             distribution = "bernoulli",
             n.trees = 100, interaction.depth = 5, shrinkage = shrink)
  predgbm=predict(gbmfit, newdata=testset, n.trees = 100)
  predgbm <- ifelse(predgbm>=0.5,1,0)
  print(sprintf("GBM no stump shrink %f, error %f",
                shrink, mean(predgbm!=testset$Class)))
  if (mean(predgbm!=testset$Class) < opterror) {
    opterror <- mean(predgbm!=testset$Class)
    optshrink <- shrink
  }
}
```

```
## [1] "GBM no stump shrink 0.010000, error 0.119205"
## [1] "GBM no stump shrink 0.050000, error 0.079470"
## [1] "GBM no stump shrink 0.100000, error 0.079470"
## [1] "GBM no stump shrink 0.200000, error 0.086093"
```

```
results[nrow(results) + 1,] = list(Model=paste(
c("Gradient boosting no stump optimal shrinkage",optshrink), collapse=" "),
                                    Error_Rate=opterror)
knitr::kable(results[6,],
caption="Gradient boosting no stump optimal shrinkage Model Performance")
```

Table 6: Gradient boosting no stump optimal shrinkage Model Performance

	Model	Error_Rate
6	Gradient boosting no stump optimal shrinkage 0.05	0.0794702

```
#Overall results comparison
knitr::kable(results)
```

Model	Error_Rate
Single tree (pruned)	0.1324503
Random forest optimal trees 100	0.0794702
Adaboost stump 100 trees	0.0927152
Adaboost depth no stump 100 trees	0.0794702
Gradient boosting stump optimal shrinkage 0.05	0.0860927
Gradient boosting no stump optimal shrinkage 0.05	0.0794702

Based on the error rate results, Optimized Random Forest, Adaboost with no stump and depth of 5 and Gradient Boosting with optimal shrinkage performed the same with 0.07947020 error rate which is the best error across all the models tested. Single tree error is way higher with 13% and was the worst across the competition. Trend can be seen as following in terms of models performance (from best to worst for this particular dataset): 1. optimized Boosting and Random Forest models 2. unoptimized Boosting and Random Forest models 3. Single Tree with pruning.

Overall, we notice that boosting methods along with random forest are very powerful models for regression and classification especially with hyperparameter tuning. Due to small dataset, differences between different boosting and random forest algorithms are hard to notice.

Question 3

Question 1

Which of the following statements is true about boosting methods?

A. Base boosting learners can be parallelized to reduce training time.

Answer: False. Main idea of boosting is to use previous model performance results for reweighting predictors of the following model to reduce error and improve performance. Therefore, it is sequential and cannot be parallelized.

B.Adaboost is immune to outliers in data.

Answer: False. Adaboost is sensitive to the outliers due to the fact it's loss function is exponential and therefore all errors are exponentiated which will lead to much higher error rate if the outliers are present.

C. Bigger learning rate in boosting leads to faster model training to converge and therefore saves time to reach the same generalization performance as slower learning rate.

Answer: False. Higher learning rates do indeed lead to faster training phase, however model may not capture all the space features and may miss local minimas that will lower the loss function and lead to better generalization error.

D.Tuning number of iterations/trees in gradient boosting will help to prevent overfitting

Answer: True. Very high or very low number of iterations in gradient boosting will lead to poor generalization performance. Very high number of iterations will lead to near perfect training loss while validation error will

increase which means overfitting. Cross-validation or OOB error can be used to tune hyperparameters for optimal model capacity.

Question 2

What conclusion about gradient boosting and random forest is true?

A. Gradient boosting is better performing model than random forest.

Answer: False. There is no such thing as always better performing or superior model for any task. It depends on the characteristics of the data. If data has high bias than generally gradient boosting will have better performance as at each step it will try to reduce this bias, while if data has high variance random forest will probably perform better due to randomnes of parallel trees and features selected.

B. Gradient boosting requires feature normalization for reliable output, while random forest doesn't need it to be done before.

Answer: False. Both of these models use decision trees which by definition requires only absolute values for branching.

C. Gradient boosting is a high bias and low variance model, while random forest is low bias and high variance model.

Answer: True. Gradient boosting is iterative process based on weak learners (small trees or even stumps) which makes them high bias and low variance model. Sequentially, gradient boosting is reducing bias as well as variance to improve performance and optimality. In contrast, random forest uses fully grown trees with randomized feature samples and random splitting that are trained in parallel and end result is a weighted average of these parallel tree results. This means that random forest has low bias from start but high variance compared to boosting.

D. Both gradient boosting and random forest use sequential approach for final convergence.

Answer: False. Gradient boosting is a sequential model which re-weights predictors to reduce error and improve performance, while random forest aggregates results from parallelized trees to derive a final prediction.

Question 4 CV Ch2q1b redoing attempt

```
library(glmnet)
#b.Write your own code in order to simultaneously optimize the a and l parameters
#by cross validation. Compute the MAE and MSE on the Ames data and compare your
#answer with a).

cv.optimalElasticNet <- function(seed, x, y, fold, alpha, lambda, xtrain, ytrain) {

  set.seed(seed)

  #Create equally sized folds
  folds <- cut(seq(1,nrow(x)),breaks=fold,labels=FALSE)
  #alpha lambda grid search
  searchgrid <- expand.grid(alpha, lambda)
  grid <- nrow(searchgrid)
  #Store intermediate results for MSE and MAE for alpha-lambda combinations
  tuneResults <- cbind(searchgrid, data.frame(matrix(0, ncol = fold, nrow=grid)))
  #Perform cross validation to tune alpha and lambda based on MSE like caret does
  for (i in 1:grid) {
    for(j in 1:fold){
      #Segment data by fold using the which() function
      testIndexes <- which(folds==j,arr.ind=TRUE)
```

```
      #Getting split for train and validation (test) fold
      testx <- x[testIndexes, ]
      testy <- y[testIndexes]
      trainx <- x[-testIndexes, ]
      trainy <- y[-testIndexes]
      #Fit and test the alpha-lambda combination for elastic net
      model.fit <- glmnet(trainx , trainy , alpha = tuneResults[i,1] ,
                          lambda = tuneResults[i,2])
      #Getting performance measures of cross-validation for alpha
      #and lambda combination
      model.pred <- predict(model.fit , testx , s=model.fit$lambda)
      meanSquaredError <- mean((model.pred - testy)^2)

      tuneResults[i, j+2] <- meanSquaredError
    }
  }
  #Calculate average MSE and use that alpha-lambda for optimal model
  tuneResults$MSE_AVG <- rowMeans(tuneResults[,3:(3+fold-1)])
optimalAlpha<-tuneResults[which(tuneResults$MSE_AVG==min(tuneResults$MSE_AVG)),1]
optimalLambda<-tuneResults[which(tuneResults$MSE_AVG==min(tuneResults$MSE_AVG)),2]
  #Use CV alpha and lambda on final optimal model
  optimal.model.fit <- glmnet(xtrain , ytrain , alpha = optimalAlpha ,
                              lambda = optimalLambda)
  optimal.model.pred <- predict(optimal.model.fit , xtest , s=model.fit$lambda)

  optimalMSE <- mean((optimal.model.pred - ytest)^2)
  optimalMAE <- mean(abs(optimal.model.pred - ytest))

  optimalResult <- data.frame("Optimal Elastic Net No Caret", optimalMSE,
                              optimalMAE, optimalAlpha, optimalLambda)
  names(optimalResult) <- c("Model", "MSE", "MAE", "Alpha", "Lambda")
  return(optimalResult)
}

#Run 10fold CV Elastc Net
alpha.grid <- seq(0, 1, 0.01)
lambda.grid <- seq(1,3, 0.1)
ElasticNetNoCaret <- cv.optimalElasticNet(123456, xdum, amesdum$Sale_Price, 10,
                    alpha.grid, lambda.grid, xdumtrain, amesdumtrain$Sale_Price)
```