remove(list=ls()); gc();

## used (Mb) gc trigger (Mb) max used (Mb)  
## Ncells 503999 27.0 1135421 60.7 609151 32.6  
## Vcells 977688 7.5 8388608 64.0 1605717 12.3

knitr::opts\_chunk$set(echo = TRUE, cache=TRUE, results = TRUE, message=FALSE, warning=FALSE);  
  
#Prepare environment   
set.seed(5104);

check.packages <- function(pkg)  
{  
 new.pkg <- pkg[!(pkg %in% installed.packages()[, "Package"])];  
 if (length(new.pkg))   
 {  
 install.packages(new.pkg, dependencies = TRUE);  
 #sapply(pkg, require, character.only = TRUE);  
 }  
}  
  
# packeges required by project  
packages<-c("caret",  
 "rpart",  
 "e1071",  
 "klaR",  
 "rattle",  
 "doParallel",  
 "parallel",  
 "randomForest",  
 "gbm",  
 "MLmetrics",  
 "dplyr",  
 "ggplot2",  
 "GGally",  
 "lattice");  
check.packages(packages);  
  
library(dplyr);  
library(caret);

# Overview

This report explores the Weight Lifting Exercises Dataset and attempt to predict the type of performance based on data from various sensors on the body.

## Data

The data for this project come from [this source](http://web.archive.org/web/20161224072740/http:/groupware.les.inf.puc-rio.br/har) or [this source](https://d396qusza40orc.cloudfront.net/predmachlearn/pml-training.csv).

This data set is unique in a way that while there are many variables, each are fundamentally the same: each set of three columns represents a gyroscope attached on different parts of the body. Each gyroscope generates data according to it’s rotation around a spatial axis, giving spatial data on three dimentions. Hence all 53 columns are tantamount to each other. Since each perdictor has well-defined meaning. they should not be scaled because this will cause distortion. \* Same scale \* Same range \* All continous

So no transformation needed.

### Load data

1. The first document is the dataset for training the model.
2. The second document contains data which we are trying to predict.

dataURL <- "https://d396qusza40orc.cloudfront.net/predmachlearn/pml-training.csv";  
  
data <- read.csv(dataURL, header = TRUE);

### Data cleaning

The data are further cleaned by:  
\* Removing the first seven fields which are just descriptive data  
\* Removing near zero variance fields  
\* Removing fields with mostly NA

#Remove the first seven columns  
data <- data[,-(1:7)];  
  
#Remove NearZeroVariance variables  
nzv <- nearZeroVar(data, saveMetrics=TRUE);  
data <- data[,nzv$nzv == FALSE];  
  
#Clean variables with mostly NA  
dataNA <- apply(data, 2, function(col){sum(is.na(col))/length(col)});  
data <- data[,which(dataNA < .1)];

Now

### Pre-processing

The training data set is sliced into 80% for training and 20% for testing.

#Data slicing  
inTrain <- createDataPartition(y=data$classe, p=0.80, list=FALSE);  
train <- data[inTrain,];  
test <- data[-inTrain,];  
head(inTrain)

## Resample1  
## [1,] 1  
## [2,] 2  
## [3,] 3  
## [4,] 5  
## [5,] 7  
## [6,] 8

### Overview of dataset

str(train);

## 'data.frame': 15699 obs. of 53 variables:  
## $ roll\_belt : num 1.41 1.41 1.42 1.48 1.42 1.42 1.43 1.45 1.45 1.43 ...  
## $ pitch\_belt : num 8.07 8.07 8.07 8.07 8.09 8.13 8.16 8.17 8.18 8.18 ...  
## $ yaw\_belt : num -94.4 -94.4 -94.4 -94.4 -94.4 -94.4 -94.4 -94.4 -94.4 -94.4 ...  
## $ total\_accel\_belt : int 3 3 3 3 3 3 3 3 3 3 ...  
## $ gyros\_belt\_x : num 0 0.02 0 0.02 0.02 0.02 0.02 0.03 0.03 0.02 ...  
## $ gyros\_belt\_y : num 0 0 0 0.02 0 0 0 0 0 0 ...  
## $ gyros\_belt\_z : num -0.02 -0.02 -0.02 -0.02 -0.02 -0.02 -0.02 0 -0.02 -0.02 ...  
## $ accel\_belt\_x : int -21 -22 -20 -21 -22 -22 -20 -21 -21 -22 ...  
## $ accel\_belt\_y : int 4 4 5 2 3 4 2 4 2 2 ...  
## $ accel\_belt\_z : int 22 22 23 24 21 21 24 22 23 23 ...  
## $ magnet\_belt\_x : int -3 -7 -2 -6 -4 -2 1 -3 -5 -2 ...  
## $ magnet\_belt\_y : int 599 608 600 600 599 603 602 609 596 602 ...  
## $ magnet\_belt\_z : int -313 -311 -305 -302 -311 -313 -312 -308 -317 -319 ...  
## $ roll\_arm : num -128 -128 -128 -128 -128 -128 -128 -128 -128 -128 ...  
## $ pitch\_arm : num 22.5 22.5 22.5 22.1 21.9 21.8 21.7 21.6 21.5 21.5 ...  
## $ yaw\_arm : num -161 -161 -161 -161 -161 -161 -161 -161 -161 -161 ...  
## $ total\_accel\_arm : int 34 34 34 34 34 34 34 34 34 34 ...  
## $ gyros\_arm\_x : num 0 0.02 0.02 0 0 0.02 0.02 0.02 0.02 0.02 ...  
## $ gyros\_arm\_y : num 0 -0.02 -0.02 -0.03 -0.03 -0.02 -0.03 -0.03 -0.03 -0.03 ...  
## $ gyros\_arm\_z : num -0.02 -0.02 -0.02 0 0 0 -0.02 -0.02 0 0 ...  
## $ accel\_arm\_x : int -288 -290 -289 -289 -289 -289 -288 -288 -290 -288 ...  
## $ accel\_arm\_y : int 109 110 110 111 111 111 109 110 110 111 ...  
## $ accel\_arm\_z : int -123 -125 -126 -123 -125 -124 -122 -124 -123 -123 ...  
## $ magnet\_arm\_x : int -368 -369 -368 -374 -373 -372 -369 -376 -366 -363 ...  
## $ magnet\_arm\_y : int 337 337 344 337 336 338 341 334 339 343 ...  
## $ magnet\_arm\_z : int 516 513 513 506 509 510 518 516 509 520 ...  
## $ roll\_dumbbell : num 13.1 13.1 12.9 13.4 13.1 ...  
## $ pitch\_dumbbell : num -70.5 -70.6 -70.3 -70.4 -70.2 ...  
## $ yaw\_dumbbell : num -84.9 -84.7 -85.1 -84.9 -85.1 ...  
## $ total\_accel\_dumbbell: int 37 37 37 37 37 37 37 37 37 37 ...  
## $ gyros\_dumbbell\_x : num 0 0 0 0 0 0 0 0 0 0 ...  
## $ gyros\_dumbbell\_y : num -0.02 -0.02 -0.02 -0.02 -0.02 -0.02 -0.02 -0.02 -0.02 -0.02 ...  
## $ gyros\_dumbbell\_z : num 0 0 0 0 0 0 0 0 0 0 ...  
## $ accel\_dumbbell\_x : int -234 -233 -232 -233 -232 -234 -232 -235 -233 -233 ...  
## $ accel\_dumbbell\_y : int 47 47 46 48 47 46 47 48 47 47 ...  
## $ accel\_dumbbell\_z : int -271 -269 -270 -270 -270 -272 -269 -270 -269 -270 ...  
## $ magnet\_dumbbell\_x : int -559 -555 -561 -554 -551 -555 -549 -558 -564 -554 ...  
## $ magnet\_dumbbell\_y : int 293 296 298 292 295 300 292 291 299 291 ...  
## $ magnet\_dumbbell\_z : num -65 -64 -63 -68 -70 -74 -65 -69 -64 -65 ...  
## $ roll\_forearm : num 28.4 28.3 28.3 28 27.9 27.8 27.7 27.7 27.6 27.5 ...  
## $ pitch\_forearm : num -63.9 -63.9 -63.9 -63.9 -63.9 -63.8 -63.8 -63.8 -63.8 -63.8 ...  
## $ yaw\_forearm : num -153 -153 -152 -152 -152 -152 -152 -152 -152 -152 ...  
## $ total\_accel\_forearm : int 36 36 36 36 36 36 36 36 36 36 ...  
## $ gyros\_forearm\_x : num 0.03 0.02 0.03 0.02 0.02 0.02 0.03 0.02 0.02 0.02 ...  
## $ gyros\_forearm\_y : num 0 0 -0.02 0 0 -0.02 0 0 -0.02 0.02 ...  
## $ gyros\_forearm\_z : num -0.02 -0.02 0 -0.02 -0.02 0 -0.02 -0.02 -0.02 -0.03 ...  
## $ accel\_forearm\_x : int 192 192 196 189 195 193 193 190 193 191 ...  
## $ accel\_forearm\_y : int 203 203 204 206 205 205 204 205 205 203 ...  
## $ accel\_forearm\_z : int -215 -216 -213 -214 -215 -213 -214 -215 -214 -215 ...  
## $ magnet\_forearm\_x : int -17 -18 -18 -17 -18 -9 -16 -22 -17 -11 ...  
## $ magnet\_forearm\_y : num 654 661 658 655 659 660 653 656 657 657 ...  
## $ magnet\_forearm\_z : num 476 473 469 473 470 474 476 473 465 478 ...  
## $ classe : Factor w/ 5 levels "A","B","C","D",..: 1 1 1 1 1 1 1 1 1 1 ...

### Principal Component Analysis

Principal Component Analysis (PCA) is a dimention reduction technique. A reduced dataset allows faster processing and smaller storage. In the context of data mining, PCA reduce the number of variables to be used in a model by focusing only on the components accounting for the majority of the variance. Highly correlated variables are also removed as a result of PCA.

In this report, PCA is perofrmed on the original dataset to reduce the number of dimention while retaining 99% of the information.

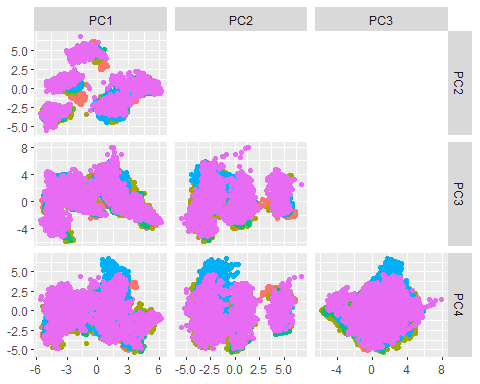
prComp <- preProcess(train[,-length(train)], method = "pca", thresh = 0.99);  
  
trainPC <- predict(prComp, train[,1:ncol(train)-1]);  
trainPC$classe <- train$classe;  
testPC <- predict(prComp, test[,1:ncol(test)-1]);  
testPC$classe <- test$classe;

Here, PCA is able to reduce the dimention of the datasets from 52 to 38 while retaining 99% of the information. This reduces model complexity and improves scalibility.

As a side note, PCA is ususlly performed on scaled/standardised dataset to prevent the resulting principle sub-space being dominated by variables with large scales. As mentioned above, because the variables in the dataset are of similar nature, scaling or standardisation provdes no added benifits. Hence such procedures are not used.

### Graphs

gpairs\_lower <- function(g)  
{  
 g$plots <- g$plots[-(1:g$nrow)];  
 g$yAxisLabels <- g$yAxisLabels[-1];  
 g$nrow <- g$nrow - 1;  
   
 g$plots <- g$plots[-(seq(g$ncol, length(g$plots), by = g$ncol))];  
 g$xAxisLabels <- g$xAxisLabels[-g$ncol];  
 g$ncol <- g$ncol - 1;  
   
 return(g);  
}  
  
g <- GGally::ggpairs(trainPC[, c(1:4)],  
 upper = list(continuous = "blank"),  
 diag = list(continuous = "blankDiag"),  
 lower = list(continuous = "points",  
 mapping = ggplot2::aes(colour = trainPC$classe)  
 )  
 );  
gpairs\_lower(g)



### Classification And REgression Training (caret) ([Help](http://topepo.github.io/caret/index.html))

* machine learning algorithms in caret package
  + linear discriminant analysis
  + regression
  + naive Bayes
  + support vector machines
  + classification and regression trees
  + random forests
  + boosting
  + many others
* caret provides uniform framework to build/predict using different models
  + create objects of different classes for different algorithms, and caret package allows algorithms to be run the same way through predict() function

### Model Specification and Cross Validation

Learning models in this report will be genereted using the train function. train accepts a trControl parameter to control the computational nuances of the train function.

#### Choosing between LOOCV and K-Fold

Leave-One-Out Cross-Validation (LOOCV) and K-Fold are common resampling methods for accessing model performance. While LOOCV estimates test error with lowest bias (averaging valodation errors across n models), K-Fold CV is much less computationally intensive. Yet there is another advantage to using K-fold CV. This has to do with a bias-variance trade-off.

Estimates produced by LOOCV is plagued by high variance compared to that produced by K-fold CV. This is because each of the n validation errors in LOOCV are produced by models trained on virtually identical dataset. The final statistic is an average of the n validation errors which are highly positively correlated. On the other hand, K-fold CV outputs K (which is usually much less than n) validation errors which are less correated as there are less overlap among models. The average of strongly correlated quantities has higher variance than the average of weakly correlated quantities; hence the estimated test error from LOOCV tends to have higher variance that that from K-fold.

The dataset in the report consists of relatively large number of observations (38 rows). Hence a 10 fold cross-validation is performed.

#### Performance Measures for Multi-Class Problems

* Accuracy and Kappa
* RMSE and
* Area Under ROC Curve
* Logarithmic Loss

tc <- trainControl(method = "cv", #resampling method = cross validation  
 number = 10, #10-fold validation  
 classProbs = TRUE,  
 summaryFunction = multiClassSummary,  
 verboseIter=FALSE,  
 allowParallel=TRUE);  
  
metric <- "logLoss";  
  
cl <- parallel::makeCluster(parallel::detectCores()- 1);#Parallel Processing, leaves you one core for other stuff.  
 #Plz try not to do CPU intensive tasks while modelling.  
doParallel::registerDoParallel(cl);

## Lazy learners

if (file.exists("nb.rds"))  
{  
 nb <- readRDS(file = "nb.rds");  
} else  
{  
 nbTime <- system.time(nb <- train(classe ~ ., data = trainPC, method = "nb", metric = metric, trControl= tc)); #naive bayes  
 saveRDS(nb, file = "nb.rds");  
 saveRDS(nbTime, file = "nbTime.rds");  
}  
  
if (file.exists("knn.rds"))  
{  
 knn <- readRDS(file = "knn.rds");   
} else  
{  
 knnTime <- system.time(knn <- train(classe ~ ., data = trainPC, method = "knn", metric = metric, trControl= tc)); #knn  
 saveRDS(knn, file = "knn.rds");  
 saveRDS(knnTime, file = "knnTime.rds");  
}

### Multinomial logistic regression

if (file.exists("multinom.rds"))  
{  
 multinom <- readRDS(file = "multinom.rds");  
} else  
{  
 multinomTime <- system.time(multinom <- train(classe ~ ., data = trainPC, method = "multinom", metric = metric, trControl= tc));  
 saveRDS(multinom, file = "multinom.rds");  
 saveRDS(multinomTime, file = "multinomTime.rds");  
}

### Tree based models

tree-based methods tend to perform well on unprocessed data (i.e. without normalizing, centering, scaling features).

Decision Trees often produce predictions with low bias but high variance. The more complex the tree, the more apparent this becomes (overfitting). Methods have been proposed to overcome this issue. This includes Bootstrap Aggregation (Bagging), as well as Random Forest.

The idea behind tree bagging is to create many trees, each trained from bootstrapped data from the original data set. Each trees are slightly different from each other because they are trained with mildly different datasets. Classification decision is then performed by popular vote across all trees. This method reduces variance by averaging decisions among many trees. There is a caveat though: tress turn out to be very similar to each other when there exists a (or few) extremely strong predictor, following by some moderately strong predictors. Each tree will have similar node splitting because of these strong predictors, which renders each trees to have practicality the same decision rules. Unfortunately, as mentioned above, the variance of the averages of highly correlated quantities is also high. This means tree bagging provides little improvments in terms of variance reduction.

Random Forest enhances tree bagging through a tweak: at each node split, the algorithm randomly picks a subset of size predictors out of all , then choose the best predictor for this node split as normally seen in decision trees. This way, each trees are more likely to be different from each other. And hence the their averages are less varying. The choice of is often the square root of but other method of chosing m also exists.

if (file.exists("ctree.rds"))  
{  
 ctree <- readRDS(file = "ctree.rds");  
} else  
{  
 ctreeTime <-system.time(ctree <- train(classe ~ ., data = trainPC, method = "rpart", metric = metric, trControl= tc)); #decision tree  
 saveRDS(ctree, file = "ctree.rds");  
 saveRDS(ctreeTime, file = "ctreeTime.rds");  
}  
  
if (file.exists("treebag.rds"))  
{  
 treebag <- readRDS(file = "treebag.rds");  
} else  
{  
 treebagGrid <- expand.grid(.mtry = ncol(trainPC) - 1);  
 treebagTime <- system.time(treebag <- train(classe ~ ., data = trainPC, method = "rf", metric = metric, tuneGrid = treebagGrid, trControl= tc)); #bagging  
 saveRDS(treebag, file = "treebag.rds");  
 saveRDS(treebagTime, file = "treebagTime.rds");  
}  
  
if (file.exists("rf.rds"))  
{  
 rf <- readRDS(file = "rf.rds");  
} else  
{  
 rfGrid <- expand.grid(.mtry = sqrt(ncol(trainPC) - 1));  
 rfTime <- system.time(rf <- train(classe ~ ., data = trainPC, method = "rf", metric = metric, tuneGrid = rfGrid, trControl= tc)); #Random Forest  
 saveRDS(rf, file = "rf.rds");  
 saveRDS(rfTime, file = "rfTime.rds");  
}

Note that in the code above, both models treebag and rf employ the training method rf. This is because tree bagging is in fact a special case of Random Forest where = .

### Neuro-Net

if (file.exists("NN.rds"))  
{  
 NN <- readRDS(file = "NN.rds");  
} else  
{  
 nnetGrid <- expand.grid(  
 size = seq(from = 1, to = 10, by = 1),  
 decay = c(0.5, 0.1, 1e-2, 1e-3, 1e-4, 1e-5, 1e-6, 1e-7));  
 NNTime <- system.time(NN <- train(classe ~ ., data = trainPC, method = "nnet", metric = metric, tuneGrid = nnetGrid, trControl= tc, verbose=FALSE));  
 saveRDS(NN, file = "NN.rds");  
 saveRDS(NNTime, file = "NNTime.rds");  
}

#remember to run this after training model(s)!!!  
parallel::stopCluster(cl);

## Compare Models

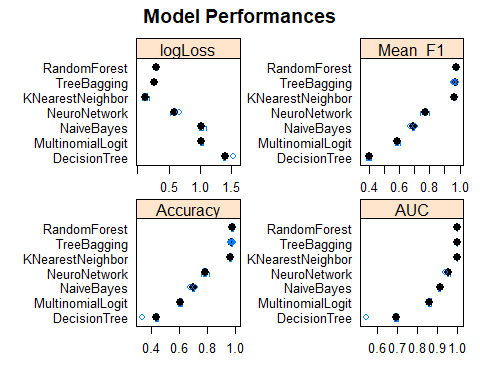
results <- resamples(list(NaiveBayes = nb,  
 KNearestNeighbor = knn,  
 MultinomialLogit = multinom,  
 DecisionTree = ctree,  
 TreeBagging = treebag,  
 RandomForest = rf,  
 NeuroNetwork = NN));  
  
  
results$metrics

## [1] "Accuracy" "AUC"   
## [3] "Kappa" "logLoss"   
## [5] "Mean\_Balanced\_Accuracy" "Mean\_Detection\_Rate"   
## [7] "Mean\_F1" "Mean\_Neg\_Pred\_Value"   
## [9] "Mean\_Pos\_Pred\_Value" "Mean\_Precision"   
## [11] "Mean\_Recall" "Mean\_Sensitivity"   
## [13] "Mean\_Specificity" "prAUC"

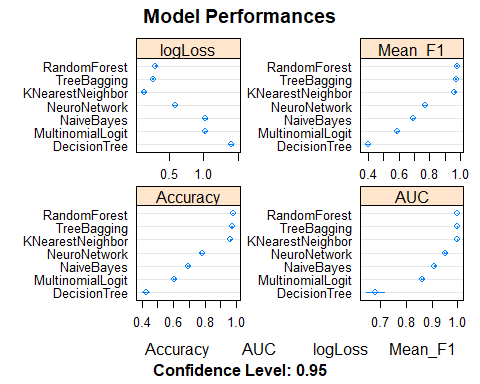
There are a total of 14 metrics for comparing models. Decision Tree is excluded here because

## Summary Statistics

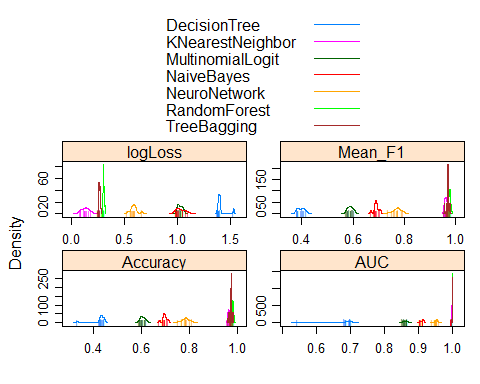
summaryStat <- summary(results)$statistics;  
  
scales <- list(x=list(relation="free"), y=list(relation="free"));  
metrics <- c("Accuracy", "AUC", "logLoss", "Mean\_F1");  
  
lattice::bwplot(results,  
 scales=scales,  
 metric=metrics,  
 main="Model Performances");



lattice::dotplot(results, scales=scales, metric=metrics, main="Model Performances");



lattice::densityplot(results, scales=scales, pch = "|", metric=metrics, auto.key=TRUE);



## Confusion Matrices of final model

confusionMatrix(predict(knn), trainPC$classe);

## Confusion Matrix and Statistics  
##   
## Reference  
## Prediction A B C D E  
## A 4442 25 0 2 0  
## B 8 2980 6 0 7  
## C 8 29 2710 51 4  
## D 5 4 19 2515 7  
## E 1 0 3 5 2868  
##   
## Overall Statistics  
##   
## Accuracy : 0.9883   
## 95% CI : (0.9865, 0.9899)  
## No Information Rate : 0.2843   
## P-Value [Acc > NIR] : < 2.2e-16   
##   
## Kappa : 0.9852   
##   
## Mcnemar's Test P-Value : 3.233e-09   
##   
## Statistics by Class:  
##   
## Class: A Class: B Class: C Class: D Class: E  
## Sensitivity 0.9951 0.9809 0.9898 0.9775 0.9938  
## Specificity 0.9976 0.9983 0.9929 0.9973 0.9993  
## Pos Pred Value 0.9940 0.9930 0.9672 0.9863 0.9969  
## Neg Pred Value 0.9980 0.9954 0.9978 0.9956 0.9986  
## Prevalence 0.2843 0.1935 0.1744 0.1639 0.1838  
## Detection Rate 0.2829 0.1898 0.1726 0.1602 0.1827  
## Detection Prevalence 0.2847 0.1912 0.1785 0.1624 0.1833  
## Balanced Accuracy 0.9963 0.9896 0.9913 0.9874 0.9965

confusionMatrix(predict(knn, testPC), testPC$classe); #don't use testPC until end of report

## Confusion Matrix and Statistics  
##   
## Reference  
## Prediction A B C D E  
## A 1105 15 0 3 0  
## B 3 727 7 1 4  
## C 4 16 667 23 3  
## D 3 1 7 613 1  
## E 1 0 3 3 713  
##   
## Overall Statistics  
##   
## Accuracy : 0.975   
## 95% CI : (0.9696, 0.9797)  
## No Information Rate : 0.2845   
## P-Value [Acc > NIR] : < 2.2e-16   
##   
## Kappa : 0.9684   
##   
## Mcnemar's Test P-Value : 0.0008391   
##   
## Statistics by Class:  
##   
## Class: A Class: B Class: C Class: D Class: E  
## Sensitivity 0.9901 0.9578 0.9751 0.9533 0.9889  
## Specificity 0.9936 0.9953 0.9858 0.9963 0.9978  
## Pos Pred Value 0.9840 0.9798 0.9355 0.9808 0.9903  
## Neg Pred Value 0.9961 0.9899 0.9947 0.9909 0.9975  
## Prevalence 0.2845 0.1935 0.1744 0.1639 0.1838  
## Detection Rate 0.2817 0.1853 0.1700 0.1563 0.1817  
## Detection Prevalence 0.2863 0.1891 0.1817 0.1593 0.1835  
## Balanced Accuracy 0.9919 0.9765 0.9805 0.9748 0.9934

#talk about why choose specific model first.