Letter Recognition

(Adapted from Bertsimas 22.3)

Introduction

One of the most widespread applications of machine learning is to do optical character/letter recognition, which is used in applications from physical sorting of mail at post offices, to reading license plates at toll booths, to processing check deposits at ATMs. Optical character recognition by now involves very well-tuned and sophisticated models. In this problem, we will build a simple model that uses attributes of images of four letters in the Roman alphabet – A, B, P, and R – to predict which letter a particular image corresponds to.

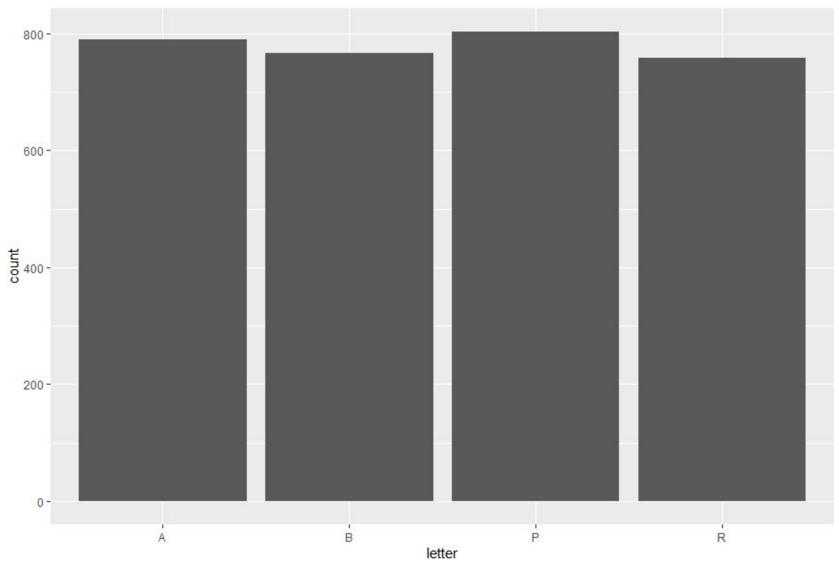
This is a multi-class classification problem. The file Letters242.csv contains 3116 observations, each of which corresponds to a certain image of one of the four letters A, B, P and R. The images came from 20 different fonts, which were then randomly distorted to produce the final images; each such distorted image is represented as a collection of pixels, and each pixel is either "on" or "off". For each such distorted image, we have available certain attributes of the image in terms of these pixels, as well as which of the four letters the image is. These features are described in Table 1. Note that feature selection is not required.

Exploratory Data Analysis

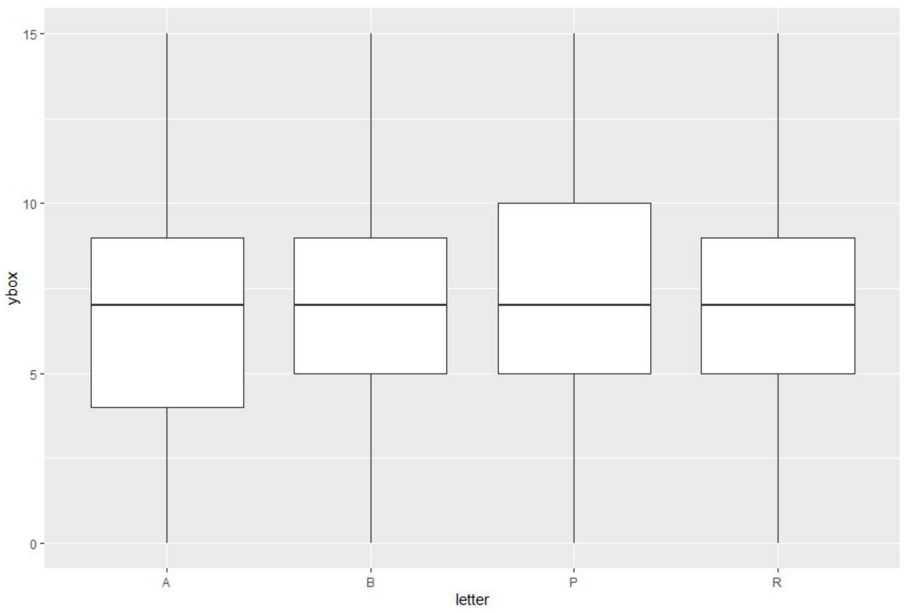
In order to undergo the Exploratory Data Analysis, we first start by looking at the distribution of the letters in the data using a bar chart to check if there is any notable unbalancing and to see what kind of sample we have. We see in the plot below that the data is fairly balanced, with each letter being represented almost equally. This means that we can kind of anticipate that a simple baseline that predicts a certain majority class would probably perform poorly on this dataset.

Table 1: Variables in the dataset Letters242.

Variable	Description
`letter`	The letter that the image corresponds to (A, B, P or R).
`xbox`	The horizontal position of where the smallest box enclosing the letter shape begins.
`ybox`	The vertical position of where the smallest box enclosing the letter shape begins.
`width`	The width of this smallest box.
`height`	The height of this smallest box.
`onpix`	The total number of "on" pixels in the character image.
`xbar`	The mean horizontal position of all of the "on" pixels.
`ybar`	The mean vertical position of all of the "on" pixels.
`x2bar`	The mean squared horizontal position of all of the "on" pixels in the image.
`y2bar`	The mean squared vertical position of all of the "on" pixels in the image.
`xybar`	The mean of the product of the horizontal and vertical position of all of the "on" pixels in the image.
`x2ybar`	The mean of the product of the squared horizontal position and the vertical position of all of the "on" pixels.
`xy2bar`	The mean of the product of the horizontal position and the squared vertical position of all of the "on" pixels.
`xedge`	The mean number of edges (the number of times an "off" pixel is followed by an "on" pixel, or the image boundary is hit) as the image is scanned from left to right, along the whole vertical length of the image.
`xedgeycor`	The mean of the product of the number of horizontal edges at each vertical position and the vertical position.
`yedge`	The mean number of edges as the images is scanned from top to bottom, along the whole horizontal length of the image.
`yedgexcor`	The mean of the product of the number of vertical edges at each horizontal position and the horizontal position.



After that, we try to visualize the distribution of the variables per letter. We select the variable xbox because it seems that it could be letter specific, and plot a box plot of the variable against the letters, shown below:



We can clearly see that the mean xbox value for the letter A is lower than the other letters, which themselves are quite similar. This is logical and can be expected if we look at the shape of the letter A, which would tend to have less width than the fairly similar B, P and R, on average. This lets us do a quick sanity check on the data and see if it makes sense at a quick glance.

Base Model

Predicting ‘B’

To warm up, we’ll start by predicting whether or not the letter is “B”. To do so, first create a new variable called isB in your dataset, which takes value “Yes” if the letter is B, and “No” if the letter is not B. Assuming that you named your data frame Letters242, this can be done by running the following command in your R console

Then randomly split your dataset into a training set and a test set, putting 65% of the data in the training set. Just do a standard random split (do not use stratified sampling) with the following commands:

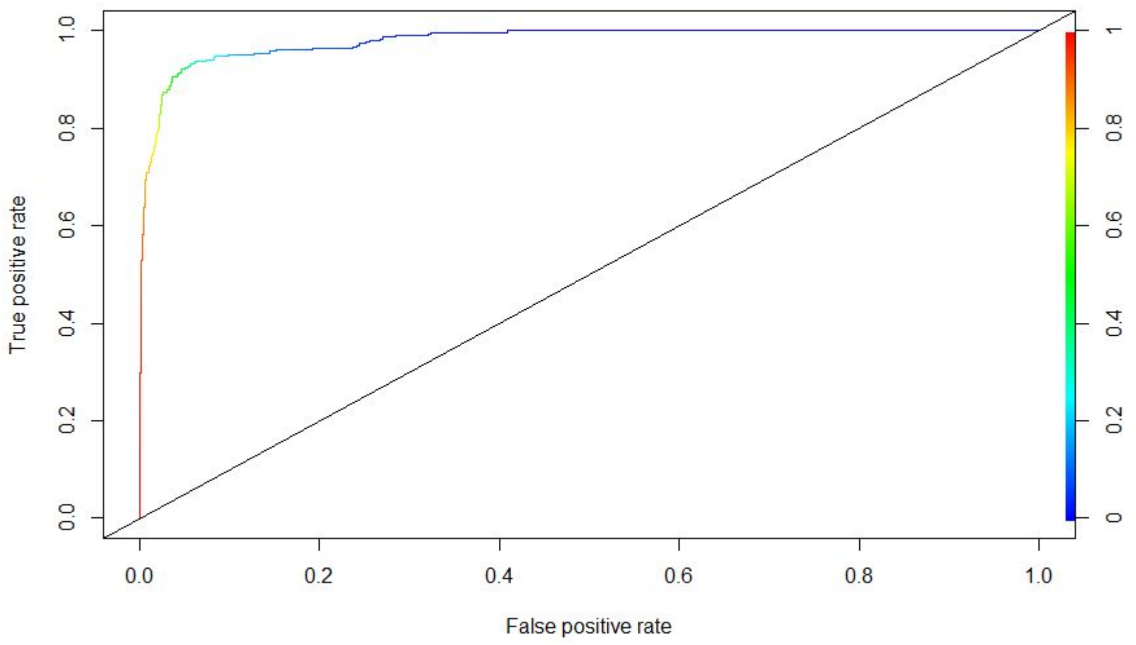
We first check the proportion of isB and isNotB in Letters. Upon doing so, we find that we have 1532 instances of isNotB and 493 of isB in the training set. Thus, a simple baseline would just be to predict isNotB all the time. In test set, we have 818 isNotB and 273 isB. The baseline would thus have an accuracy of 818 818+273 = 0.749 on the Test Set.

We build a logistic regression model and train it on the Training Set. We then evaluate its performance on the Test Set using a threshold of p = 0.5 and obtain the following metrics:

True Negatives: 792	True Positives: 237
False Positives: 26 False	Negatives : 36

The accuracy of the model is thus 792+237 792+237+36+26 = 0.943

We compute the ROC curve and observe that is completely outperforming the baseline and is actually excellent on the test set. We already know that the model achieves a high accuracy and we can see that this good performance is also translated in the AUC with a value of 0.978.

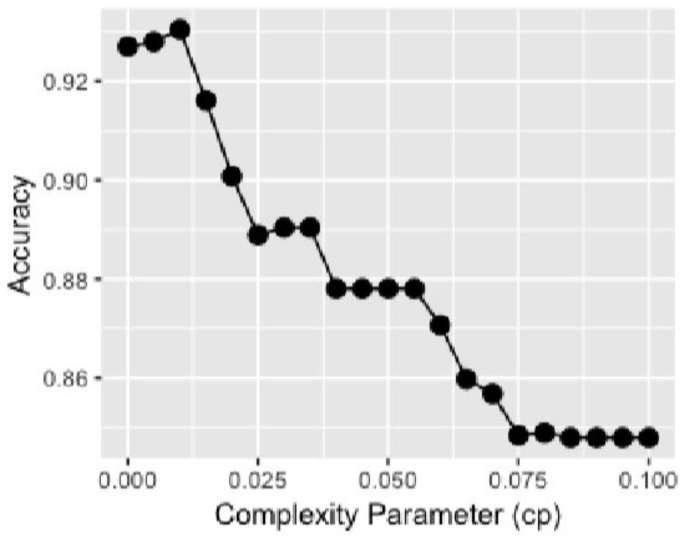


True positive rate
False positive rate
0.0
0.2
0.4
0.6
0.8
1.0
0
0.2
0.4
0.6
0.8
1

CART Tree

To construct the CART tree, we first start with cross-validation to determine the parameter cp. We decided to use the cross-validation using 10 folds (k is usually taken as 5 or 10), and try all the values of cp between 0 and 0.1 (which is a typical range) with increments of 0.005.

We train the model using the various values of cp, and leaving one fold out at each of the 10 fold iteration as a validation set to evaluate performance on. We then chose the value of cp for which the average accuracy over the 10 folds is the highest. We obtain a cp value of 0.01. A plot of the various Accuracies vs cp can be seen below, where we can see a peak Accuracy at cp = 0.01:



The following metrics were obtained:

True Positives: 790	True Negatives: 228
False Positives: 28	False Negatives: 45

The accuracy of the model on the Test set is thus 790+228/790+228+28+45 = 0.933

Random Forest

We use the Random Forest model to predict if it the letter is a B or not. We train the model on the Training set, using the default R parameters (we omit them from the training function call). We then evaluate the performance of the model on the Test set, and obtain the following metrics:

True Positives: 811 True Negatives: 255
False Positives: 7 False Negatives: 18

The accuracy thus comes to be: 811+255 811+255+7+18 = 0.977.

One might be surprised to see the model performing so well without tuning. However, for Random Forest, the model is actually robust to the choice hyperparameters used, and can give good results without tuning.

To recap, the accuracies of the models selected are the following: Logistic Regression, 0.943; CART, 0.933; Random Forest, 0.977; We can see that, perhaps unsurprisingly, the Random Forest model performs the best on the Test Set. This is usually the case, as the model is bound to usually outperform simpler models such as Logistic Regression and CART, and generalize better, even without tuning parameters. On this application, I think that accuracy is much more important than interpretability. For such a type of application, even a small error rate could result in huge costs and operational problems. Taking the example of recognizing letters on mail, even with 2.3 percent error rate (97.7 percent accuracy), with a really high number of mail sent everyday, we would still sort a huge amount of mail in the wrong manner and cause a disruption in the service. Similarly with license plate recognition, we would send fines wrongly to thousands of people. As such, I think that interpretability is not the most important issue here, as we only care about the accuracy and performance of the model: if the model classifies the letter correctly or not.

Final Model

Now let us move on to the original problem of interest, which is to predict whether or not a letter is one of the four letters A, B, P or R. The variable in the dataset which you will try to predict is letter.

Baseline

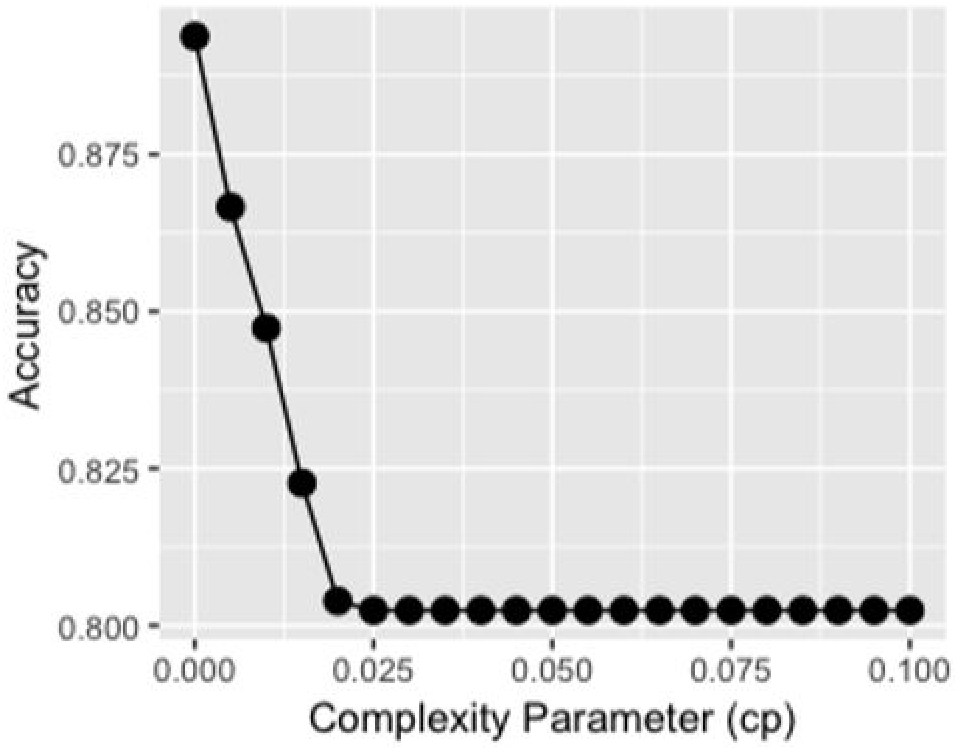
The most represented category in the letters of the training set is the letter A, with 525 occurrences. Thus, a simple baseline would be to always predict A. Such a baseline would have an accuracy of

264/264+273+283+271 = 0.242,

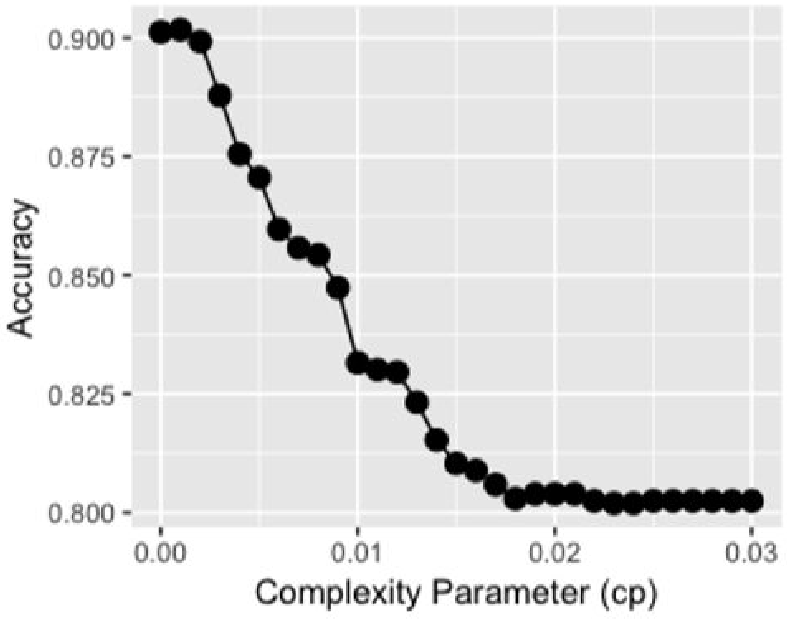
which is expected, since the test data letter distribution is somewhat evenly spread among letters.

CART

For the CART model, we first try to do cross-validation to determine cp. We try 10-fold cross validation with cp values from 0 to 0.1 in 0.005 increments, however, this leads to an optimal cp = 0, which is confirmed by plotting:



The plot shows that after cp around 0.03. the increase in cp is meaningless. The plot also shows that the step might be too big. As such we reduce the interval to be from 0 to 0.03 in 0.001 intervals. We obtain an optimal cp = 0.001 and the following plot:



Thus, the final CART model has cp = 0.001.
We evaluate this model's performance on the Test Set and obtain an accuracy of

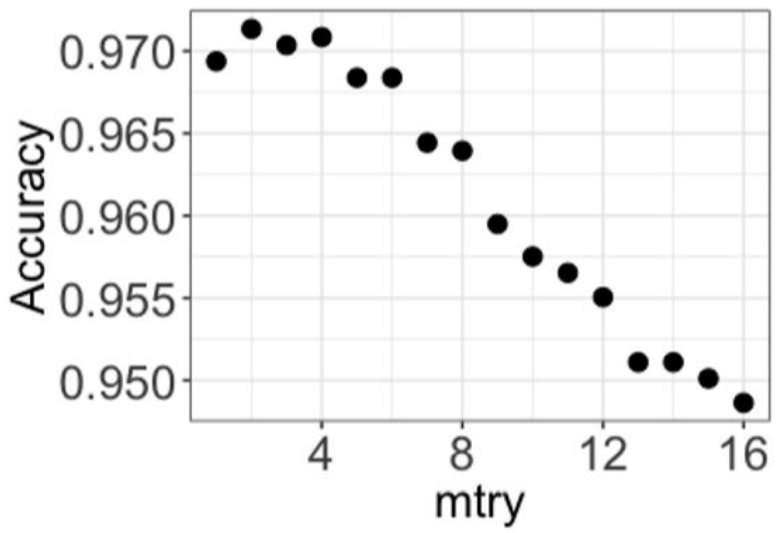
258+240+268+237/1091 = 0.919.

Random Forest

Next, we train a Random Forest model on the Training set, and evaluate its performance on the Test Set in order to predict letters. Again, we use the default parameters provided by R by omitting them in the function call. We obtain an accuracy of

263+262+280+260/1091 = 0.977

In this section, we first undergo 5-fold cross-validation to chose the best value of mtry for the Random Forest model. We chose k=5 here instead of 10 because we thought it would be too computationally intensive. We try values of mtry from 1 (minimum, only 1 feature is considered at each split) to 16 (maximum, all features are considered at each split, similar to Bagging) and chose the model with the highest average accuracy on the 5-folds. We obtain a value of mtry = 2, as confirmed from the plot below:



The performance of the Random Forest model with mtry = 2 is then evaluated on the Test Set. We obtain an accuracy of 263+262+280+262 1091 = 0.978. It is interesting to note that in the previous model using the default parameters, mtry was equal to sqrt(number of features) = √16 = 4. In this model, we chose a slightly higher mtry, choosing between 2 features for each split. We can see that although we did go through tedious cross validation, the mtry value chosen is close to the default, and the accuracy is basically similar. This confirms the fact that Random Forest can perform very well without tuning its parameters.

Gradient Boost

In this section, we train a Gradient Boosting Machine model on the Training set, using the values of the parameters already determined from 5-fold cross validation. Setting the interaction depth to 10, we run the method for 3300 iterations, and leave all other parameters at their default values. We then evaluate the performance of the model on the Test Set, and obtain an accuracy of

$$262+264+280+266/1091 = 0.982$$

Summary

To recap, we obtained the following accuracy using different models:

CART,	0.919;
Random Forest,	0.978;
Gradient Boosting Machine,	0.982.

We can see that the Boosting model gives the best accuracy, and so we would recommend it for this problem. In letter classification, accuracy is very important and a few percentage increase in accuracy can lead to drastically better performances in numbers when implemented on large scale applications. However, the Boosting model would require much more tuning that the Random Forests, and if we are short on computational power, we might want to use the Random Forest, which has somewhat comparable accuracy. In question b), we chose the Random Forest as the best model, as we also care more about accuracy than interpretability. However, we didn't try Boosting. Maybe if we did, the model's performance would be similar or might be even better, like in this case.

Code Review

```
library(dplyr)
library(ggplot2)
library(caTools) # splits
library(GGally)
library(ROCR)
library(MASS)
library(rpart) # CART
library(rpart.plot) # CART plotting
library(caret)


Letters = read.csv('Letters242.csv')
str(Letters)

## a) Exploratory Data Analysis

# Bar Chart:
ggplot(data = Letters) +  geom_bar(mapping = aes(x= letter))

# we can see that there is almost the same number of letters , so no major unbalancing
# we are interested in seeing each variable's distribution, so we look at the diagonal:
ggscatmat(Letters, columns = 2:17, alpha = 0.8)

#density <- density(Letters$letter)
#plot(density)

#Box Plots
ggplot(data = Letters, mapping = aes(x = letter, y = xbox)) +  geom_boxplot()
ggplot(data = Letters, mapping = aes(x = letter, y = ybox)) +  geom_boxplot()

## b)
Letters$isB = as.factor(Letters$letter == "B")
str(Letters)
set.seed(456)

# standard split data

train.ids = sample(nrow(Letters), 0.65*nrow(Letters))
Letters.train = Letters[train.ids,]
Letters.test = Letters[-train.ids,]
```

```
# b) i)
#baseline:

table(Letters.train$isB)

# We have 1532 instances of isNotB and 493 of isB in training set
# thus predict: isNotB all the time

#On test set, we have 818 isNotB and 273 isB
table(Letters.test$isB)
#Baseline accuracy on test  = 0.749:
818/(818+273)




## b) ii)
#Logistic Regression model
log_model = glm(isB ~ xbox + ybox + width + height + onpix +
            xbar + ybar + x2bar + y2bar + xybar + x2ybar +
            xy2bar + xedge + xedgeycor + yedge + yedgexcor, data = Letters.train, family = "binomial")
summary(log_model)

#predict on TestSet
log_predTest = predict(log_model, newdata = Letters.test, type = "response")

#get Accuracy = 0.943
table(Letters.test$isB, log_predTest > 0.5)
log_accuracy = (792+237)/(792+237+26+36)

## b) iii)
# ROC curve and AUC = 0.978
rocr.log.pred <- prediction(log_predTest, Letters.test$isB)
logPerformance <- performance(rocr.log.pred, "tpr", "fpr")
plot(logPerformance, colorize = TRUE)
abline(0, 1)
as.numeric(performance(rocr.log.pred, "auc")@y.values)


## b) iv) CART model, we obtain cp = 0.01
train.cart = train(isB ~ xbox + ybox + width + height + onpix +
              xbar + ybar + x2bar + y2bar + xybar + x2ybar +
              xy2bar + xedge + xedgeycor + yedge + yedgexcor,
             data = Letters.train, method = "rpart",tuneGrid = data.frame(cp=seq(0, 0.1, 0.005)),
                   trControl = trainControl(method="cv", number=10),
                   metric = "Accuracy")

train.cart$results
train.cart

# plot the results
ggplot(train.cart$results, aes(x=cp, y=Accuracy)) + geom_point(size=3) + xlab("Complexity Parameter
(cp)") + geom_line()

# Extract the best model and make predictions
train.cart$bestTune cart_final = train.cart$finalModel
prp(cart_final, digits=3)

#no need for model matrix because no factor variables in data, all continuous

pred = predict(cart_final, newdata=Letters.test, type="class")
table(Letters.test$isB, pred)

# accuracy = 0.933 (790+228)/(790+228+45+28)

## b) v) Random Forest using default values

mod.rf <- randomForest(isB ~ xbox + ybox + width + height + onpix +
                         xbar + ybar + x2bar + y2bar + xybar + x2ybar +
                         xy2bar + xedge + xedgeycor + yedge + yedgexcor,
                       data = Letters.train)
importance(mod.rf)
pred.rf <- predict(mod.rf, newdata = Letters.test)

#Get Accuracy = 0.977 table(Letters.test$isB, pred.rf)
(811+255)/(811+255+7+18)

## c) i) Baseline
# get most represented category: A with 525

table(Letters.train$letter)

#check the same on test and assess baseline performance
table(Letters.test$letter)
(264)/(264+273+283+271)

## c) ii)
```

```
train.cart_2 = train(letter ~ xbox + ybox + width + height + onpix +
                       xbar + ybar + x2bar + y2bar + xybar + x2ybar +
                       xy2bar + xedge + xedgeycor + yedge + yedgexcor ,
                     data = Letters.train, method = "rpart",
                     tuneGrid = data.frame(cp=seq(0, 0.1, 0.005)),
                     trControl = trainControl(method="cv", number=10),
                     metric = "Accuracy")

train.cart_2 = train(letter ~ xbox + ybox + width + height + onpix +
                       xbar + ybar + x2bar + y2bar + xybar + x2ybar +
                       xy2bar + xedge + xedgeycor + yedge + yedgexcor ,
                     data = Letters.train,
                     method = "rpart",
                     tuneGrid = data.frame(cp=seq(0, 0.03, 0.001)),
                     trControl = trainControl(method="cv", number=10),
                     metric = "Accuracy")

train.cart_2$results
train.cart_2


# plot the results
ggplot(train.cart_2$results, aes(x=cp, y=Accuracy)) + geom_point(size=3) + xlab("Complexity
Parameter (cp)") + geom_line()

# Extract the best model and make predictions

train.cart_2$bestTune cart_final_2 = train.cart_2$finalModel
prp(cart_final_2, digits=3)

#no need for model matrix because no factor variables in data, all continuous

pred_2 = predict(cart_final_2, newdata=Letters.test, type="class")
table(Letters.test$letter, pred_2)

# accuracy = (258+240+268+237)/(nrow(Letters.test))

## c) iii) Random Forests
mod_2.rf <- randomForest(letter ~ xbox + ybox + width + height + onpix +
                           xbar + ybar + x2bar + y2bar + xybar + x2ybar +
                           xy2bar + xedge + xedgeycor + yedge + yedgexcor,
                         data = Letters.train)
```

```
importance(mod_2.rf)

pred_2.rf <- predict(mod_2.rf, newdata = Letters.test)
summary(mod_2.rf)

#Get Accuracy = 0.977
table(Letters.test$letter, pred_2.rf)
(263+262+280+260)/(1091)

## c) iv) Random Forests: with CV

train.rf <- train(letter ~ xbox + ybox + width + height + onpix +
                    xbar + ybar + x2bar + y2bar + xybar + x2ybar +
                    xy2bar + xedge + xedgeycor + yedge + yedgexcor,
                  data = Letters.train,
                  method = "rf",
                  tuneGrid = data.frame(mtry=1:16),
                  trControl = trainControl(method="cv", number=5, verboseIter = TRUE),
                  metric = "Accuracy")

train.rf$results
train.rf

ggplot(train.rf$results, aes(x = mtry, y = Accuracy)) +
  geom_point(size = 3) +   ylab("Accuracy") + theme_bw() +
  theme(axis.title=element_text(size=18), axis.text=element_text(size=18))

best.rf <- train.rf$finalModel
pred.best.rf <- predict(best.rf, newdata = Letters.test)

table(Letters.test$letter, pred.best.rf)

#accuracy = 0.978 (263+262+280+262)/(1091)

## c) v) Boosting

mod.boosting <- gbm(letter ~ xbox + ybox + width + height + onpix +
                      xbar + ybar + x2bar + y2bar + xybar + x2ybar +
                      xy2bar + xedge + xedgeycor + yedge + yedgexcor,
                    data = Letters.train, distribution = 'multinomial', n.trees = 3300,interaction.depth = 10)

pred.boosting <- predict(mod.boosting, newdata = Letters.test, n.trees=3300, type = 'response')
p.pred.boosting <- apply(pred.boosting, 1, which.max)
```

```
table(Letters.test$letter, p.pred.boosting)

#accuracy : 0.982 (262+264+280+266)/(1091)
```