Mushroom Data Classification
CS5950: Machine Learning
Final Project
By Chris Carlson and Ben Mechling

MUSHROOM DATA

The mushroom data was obtained from a UCI data repository (https://archive.ics.uci.edu/ml/datasets/Mushroom). The data set has 8124 rows and a total of 23 columns. All of the data is categorical. Each value is a single letter representing a value specific to a given attribute (column). The first column "class" indicates whether the mushroom is considered to be poisonous (p) or edible (e). The number of possible values for each attribute varies. The table below shows the possible of each attribute and symbol used to represent it. The largest number of possible values is 12.

Attribute	**Values**
class	edible=e, poisonous=p
cap.shape	bell=b, conical=c, convex=x, flat=f, knobbed=k, sunken=s
cap.surface	fibrous=f, grooves=g, scaly=y, smooth=s
cap.color	brown=n,buff=b,cinnamon=c,gray=g,green=r, pink=p, purple=u, red=e, white=w, yellow=y
bruises	bruises=t, no=f
odor	almond=a, anise=l, creosote=c, fishy=y, foul=f, musty=m, none=n, pungent=p, spicy=s
gill.attachment	attached=a, descending=d, free=f, notched=n
gill.spacing	close=c, crowded=w, distant=d
gill.size	broad=b, narrow=n
gill.color	black=k, brown=n, buff=b, chocolate=h, gray=g, green=r, orange=o, pink=p, purple=u, red=e, white=w, yellow=y
stalk.shape	enlarging=e, tapering=t
stalk.root	bulbous=b, club=c, cup=u, equal=e, rhizomorphs=z, rooted=r, missing=?
stalk.surface.above.ring	fibrous=f, scaly=y, silky=k, smooth=s
stalk.surface.below.ring	fibrous=f, scaly=y, silky=k, smooth=s
stalk.color.above.ring	brown=n, buff=b, cinnamon=c, gray=g, orange=o, pink=p, red=e, white=w, yellow=y
stalk.color.below.ring	brown=n, buff=b, cinnamon=c, gray=g, orange=o, pink=p, red=e, white=w, yellow=y
veil.type	partial=p, universal=u
veil.color	brown=n, orange=o, white=w, yellow=y
ring.number	none=n, one=o, two=t
ring.type	cobwebby=c, evanescent=e, flaring=f, large=l, none=n, pendant=p, sheathing=s, zone=z
spore.print.color	black=k, brown=n, buff=b, chocolate=h, green=r, orange=o, purple=u, white=w, yellow=y
population	abundant=a,clustered=c,numerous=n,scattered=s,several=v,solitary=y
habitat	grasses=g, leaves=l, meadows=m, paths=p, urban=u, waste=w, woods=d

Data Observations
The attributes values for mushrooms tend to be heavily grouped by the class, making classification somewhat easy.

Class
The number of elements in each class (edible, poisonous) is roughly equal.

Edible	Poisonous
4208	3916

Odor
The odor class is the most distinctive attribute. The table below shows the frequency of each value in the feature for each class over the entire data set. Not all the features were this segregated, but some clear division were found in other features as well.

Class	a	c	f	l	m	n (none)	p	s	y
Edible	400	0	0	400	0	3408	0	0	0
Poisonous	0	192	2160	0	36	120	256	576	576

Spore Print Color
The spore print color is also a strong classifier. Although a comparison of all of its values is not as segregated as odor, if only records where odor = n, were evaluated, then approximately 600 records remained that were ambiguous.

Veil Type
The veil type attribute is supposed to have two possible values, but in the data only one value exists. All 8124 rows have value “p” (partial). For Bayesian and regression models this column was excluded from the data.

CLASSIFICATION

Directly Available Methods
Since the data is entirely categorical, numerical methods cannot be directly applied. Methods that can be directly applied are **naive Bayesian classification** (NBC), **classification trees**, and **random forests**.

Indirect Methods
While implementing a Bayesian classifier, it was observed that the naive Bayesian approach computes a numeric probability estimate value for each element of a test vector. If a dataset was created using these estimates, then numeric methods could be applied as well. Using an estimate of the log-likelihood that the vector is edible given the value of the feature, numeric training and test sets were built. **Linear discriminate analysis** (LDA), **quadratic discriminate analysis** (QDA), and **logistic regression** (Log.Reg.) were applied.

Naive Bayesian Classification
By Ben Mechling

The naïve Bayesian classifier computes an estimate probability that a test vector belongs to a class for each class. In the mushroom data there are only two classes: edible and poisonous. The estimate probabilities are computed using a modified form of Bayes theorem. Bayes' theorem is stated as products.

$$p(c|v) = p(v|c) * p(c) / p(v)$$

As is commonly done for classification, the form is converted to sums of logs and the denominator p(v) is discarded.

$$\log(p(c|v)) = \log(p(c)) + \text{sum}(\log(p(v\_i|c)))$$

To find probabilities in a formal sense, the number of occurrences is divided by the total number of elements.

$$p(x) = \text{freq}(x)/n$$
$$\log(p(x)) = \log(\text{freq}(x)) - \log(n)$$

For this application, however, the size of the set only scales the results. Experimentally, probability estimates were better by ignoring the -log(n) term in computations.

$$\text{estimate}(c|v) = \log(\text{freq}(c)) + \text{sum}(\log(\text{freq}(v\_i|c)))$$

Thus, the estimate value use in the implementation uses log frequencies rather than proper probabilities.

The freq(v_i|c) value of is the number of occurrences of the given feature value in a given class. A heuristic was used to replace log(freq(v_i|c)) the value never occurred in class c. Naturally the arithmetic results in negative infinity, which is difficult to use in computation. In such cases, the -log(freq(v_i)), the negative log frequency over all classes used as a more reasonable penalty. With this approach if the edible class never have value v_i, but the number of occurrences was large and all poisonous, a large penalty would be given to the estimate probability of the test being edible. On the other hand, if the number of occurrences was small and all were poisonous, then a smaller penalty was given.

*For the mushroom data, features values had a tendency exclusively used by one class. For Bayesian classification, the experimental results improved when the penalty was large. Some of the best results were found for a penalty of -log(freq(v_i)) * 100. The author theorizes that these large penalties push that results strongly to one side or the other and that because the data is very segregated this strong push tends to be accurate. In the general scope of this assignment, the * 100 multiplier was not used.*

Indirect Numerical Classification
By Ben Mechling

The computation process for NBC computes sum(log(freq(v_i|c))) for each class. Effectively the NBC model has two training data sets: one set with edible mushrooms and one set with poisonous mushrooms. In order to run numerical methods, a single numerical training data needed to be created. Thus for a given value v_i a single and useful number was needed. In experimentation, the best form found was a subtraction to the two values.

$$\text{numericalData}(v\_i) = \text{freq}(v\_i|\text{edible}) - \text{freq}(v\_i|\text{poisonous})$$

This form seemed theoretically reasonable given the form for the odds' of a value.

$$\text{odds}(x) = p(x)/(1-p(x))$$

$$\text{logit}(x) = \log(\text{odds}) = \log(p(x)) - \log(1-p(x))$$

For this implementation, proper probabilities were not used, thus 1-p(x) is not a good estimate. The estimate of p(poisonous) should reasonable inverse for p(edible).

$$\text{estimate logit}(v\_i) = \log(\text{edible}) - \log(\text{poisonous})$$

For practical use, the value will be positive if the data supports edible and negative if the data supports poisonous.

For each training set of original data, a version of the mushroom data was created as numerical data. From this numerical data, train and test data sets were partitioned in the same manner as the original data. Using the standard R modeling functions on the training data, LDA, QDA, and logistic regression models were built and predictions were made over the numerical test data.

Bayesian Based Classification Results

Using 5-fold cross validation on NBC, LDA, QDA, and logistic regression gave the following results.

Regular Penalty	**Actually Edible**	**Actually Poisonous**	**Heavy Penalty**	**Actually Edible**	**Actually Poisonous**
Predicted Edible	4206	78	Predicted Edible	4206	3
Predicted Poisonous	0	3838	Predicted Poisonous	2	3913

LDA Classifier

Regular Penalty	**Actually Edible**	**Actually Poisonous**	**Heavy Penalty**	**Actually Edible**	**Actually Poisonous**
Predicted Edible	4193	62	Predicted Edible	4208	40
Predicted Poisonous	15	3854	Predicted Poisonous	0	3876

QDA Classifier

Regular Penalty	**Actually Edible**	**Actually Poisonous**	**Heavy Penalty**	**Actually Edible**	**Actually Poisonous**
Predicted Edible	4208	0	Predicted Edible	4208	22
Predicted Poisonous	0	3916	Predicted Poisonous	0	3894

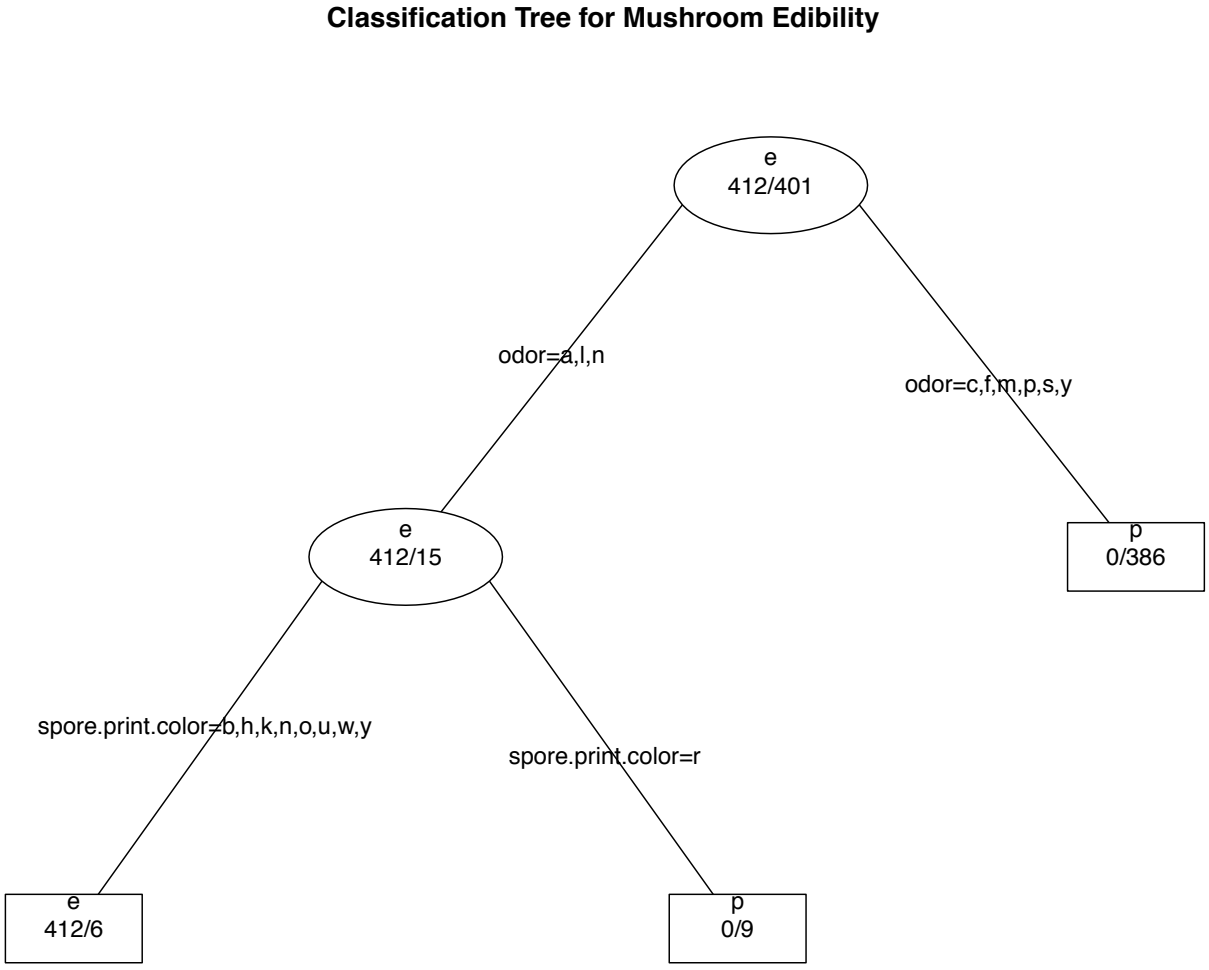
Logistic Regression Classifier

Regular Penalty	**Actually Edible**	**Actually Poisonous**	**Heavy Penalty**	**Actually Edible**	**Actually Poisonous**
Predicted Edible	4208	0	Predicted Edible	4208	0
Predicted Poisonous	0	3916	Predicted Poisonous	0	3916

A number of minor modifications to the algorithm were attempted. Overall, NBC (without the heavy artificial weights) was the worst. LDA and QDA were generally better the NBC, but one of the two was not always better than the other. Logistic regression was by far the best of the classifiers. on the quantitative data.

Decision Tree Classification
By Chris Carlson

The Decision tree classifier was constructed in R using the 'rpart' package. Due to the nature of the data the decision tree classifier was a simple and effective classifier to build. As mentioned in the introduction, the data can be split using only two feature vectors, odor and spore-print-color, with excellent results. The sample plot shown below is representative of the trees rpart builds. In fact, without imposing additional constraints, the rpart package always splits this data on these two features regardless of how the data subsets are arranged.



Random Forest Classification
By Chris Carlson

The Random Forest classifier was produced using the r package called 'randomForest'. The results of random forest classification were much better than the simple decision tree, which is expected. Typical test error rates for random forest models were less than 0.01, which is the second best result we achieved - the best being logistic regression on Bayesian probabilities. The random forest classifier was tested with as many as 500 trees and as few as 150 trees, and the number of variables available per split was also adjusted from between 3 up to 8. Overall adjusting these characteristics didn't affect test error rates significantly. Shown below are confusion matrices produced by the random forest algorithm in a typical k-folds cross validation.

```
Confusion Matrix averages from each iteration:
   e      p
e 846.75 0.75
p 0     777.25

   e    p
e 836.5 1
p 0.25 787

   e     p
e 839.25 1
p 0.25 784.25

    e    p
e 844.5 1.25
p 0     779

    e     p
e 840.25 0.75
p 0.25  783.75


Average Confusion Matrix from 5-folds cross validation:
   e      p
e 841.45 0.95
p 0.15  782.25
```

CROSS VALIDATION

K-Folds Cross Validation

K-Folds Cross Validation was used for all model types, though the specific implementation differs between the tree based approaches and the Bayesian approaches. For the Bayesian approaches cross validation was computed precisely according to the method description; the data is dived into k equal size subsets, and for k iterations one section of the data is held out as testing data, and the remaining data is used to build a model which is subsequently tested on the test data. The results of all iterations are reported individually and an average across iterations is reported. For the decision tree methods cross validation was implemented slightly differently; in this approach a model was generated for each of the training folds individually, and each of these models was tested on the test data. The results of each model were averaged to determine the overall test error for the iteration.

```
#########################################################################
#           Functions .R
#########################################################################
#
#
#
#     ###     Description         ###
#
# Functions contains various generally aplicable methods/functions
# that I desire to have scripted and available in a tested
# and reliable manner.  Anything that can be refactored should
# eventually find it's way here.
#
#     ###     By              ###
#
# Written by Christopher Carlson
#
#     ###     For             ###
#
# Written initially for Western Michigan University's Summer 1 2015
#     Semester course, CS 5950 - Machine Learning.
#
#     ###     Date            ###
#
# Monday June 29, 2015
#
#     INDEX
#
# 0. Preload Necessary Libraries
#
# 1. "Round-Mean" - round_mean(X, digits)
#
# 2. "Ceiling-Mean" - ceil_mean(X)
#
# 3. "Floor-Mean" - floor(X)
#
# 4. "Print-Summary" - print_summary(X)
#
# 5. "Subset-Folds" - subset_flds(folds, i_test)
#
# 6. "Grow Forest" - grow_forest(data_frame, test_data)
#
# 7. "Grow Tree" - grow_tree(data_frame)
#
# 8. "Test Tree" - test_tree(model, test_data)
#
# 9. "Generate Tree-Fit Confusion Matrix" - gen_tree_conf_mat(pred, test_data)
```

```
#
#  10. "Plot Tree" - plot_tree(tree_model, main_message)
#
#  11. "Get Random Forest Confusion Matrix" - get_rf_conf_mat(rf_object)
#
#  12. "Apply K-Folds to Tree" - k_folds(tree)
#
#  13. "Write Confusion Matrix" - write_confusion_matrix(c_matrix, r_file)
#
#  14. "Write Message" - write_message(msg, r_file)
#
#  15. "Load Data" - load_data(data_file)
#
#########                              ##########

# 0

## Load Libraries
library('randomForest')
library(rpart)
library(plyr)
##

# 1

##  "Round-Mean"
## Find the mean of some collection X and then round the result to
## digits places.  Used with apply functions.
round_mean <- function(X, digits)
{
  round(data.frame(mean(X)), digits)
}

# 2

## "Ceiling-Mean"
## Find the mean of collection X and then round the result up to the
## nearest integer value.
ceil_mean <- function(X)
{
  ceiling(mean(X))
}

#3

## "Floor-Mean"
## Find the mean of collection X and then round the result down to the
```

```
## nearest integer value.
floor_mean <- function(X)
{
  floor(mean(X))
}

# 4

## "Print Summary"
## Print the summary of item X.
print_summary <- function(X)
{
  print(summary(X))
}

# 5

## "Subset Folds"
## Subsets K folds into Training and Testing Data where the i'th fold
## becomes test and the other four folds become training folds.
subset_flds <- function(folds, i_test)
{
  env <- parent.frame()
  env$test  <- folds[[i_test]]
  env$train <- folds[c(seq(1:(length(folds))))[-i_test]]
}

# 6

## "Grow Forest
## Function to generate an RF model given a data frame object from the
## mushrooms data set, and some test data.
grow_forest <- function(data_frame, test_data) {

# Random Forest Object is generated by the 'randomForest' command.
# mtry is the number of variables tested for each split, ntree is the
# number of trees grown, xtest and ytest specify respectively the
# set of predictors to test the data on, and the set of correct responces
# corresponding with the predictors.
  randomForest(edibility~., data=data_frame, mtry=3, ntree=150,
      xtest=test_data[-1], ytest=test_data$edibility,
      importance=TRUE, proximity=TRUE)
}

# 7

# "Grow Tree"
```

```
# Function to generate a model given a data frame object from the
# mushrooms data set.
grow_tree <- function(data_frame) {
  rpart(edibility~., data=data_frame, method="class")
}

# 8

# "Test Tree"
# A function to classify some data using a tree model produced by
# the grow_tree, (rpart), function.   It does not return typical
# results, instead it simply returns an array of P's and E's
# indicating which of the two values the probilities indicated.
test_tree <- function(model, test_data) {
  pred <- data.frame(predict(model, test_data, type='prob'))

  results <- array("p", nrow(test_data))

  for(i in 1:nrow(results))
  {
    if(pred[i, ]$e > pred[i, ]$p)
    {
      results[i] = "e"
    }
  }
  results
}

# 9

# "Generate Tree-Fit Confusion Matrix"
# A function to compare the predictions with the data labeles
# to construct a confusion matrix based of the tree clasifiers
# results.
gen_tree_conf_mat <- function(pred, test_data) {

  pp = 0 # counter for poisonous mushrooms predicted as poisonous
  pe = 0 # counter for poisonous mushrooms predicted as edible
  ep = 0 # counter for edible mushrooms predicted as poisonous
  ee = 0 # counter for edible mushrooms predicted as edible

  for(i in 1:nrow(pred))
  {
    if(pred[i] == test_data$edibility[i])
    {
      if(pred[i] == 'e')
      {
```

```
        ee = ee+1
      }
      else
      {
        pp = pp+1
      }
    }
    else
    {
      if(pred[i] == 'e')
      {
        pe = pe+1
      }
      else
      {
        ep = ep+1
      }
    }
  }
  conf_matrix <- matrix(c(ee, ep, pe, pp), nrow=2, ncol=2)
  rownames(conf_matrix)<-c("e", "p")
  colnames(conf_matrix)<-c("e", "p")
  conf_matrix
}

# 10

## "Plot Tree"
## Given a tree model, plot the tree model.
plot_tree <- function(tree_model, main_label="Classification Tree for Mushroom
{
  par(mfrow=c(1,2), xpd=NA)
  plot(tree_model, uniform=TRUE, main=main_label)
  text(tree_model, use.n=TRUE, all=TRUE, cex=.8)
  post(tree_model, file=paste0("results_temp/fit_", i), title="Classification T
}

# 11

## "Apply K-Folds to Tree"
## Applies K-Folds using classification trees.  Returns a list
## of confusion matrices.
kfolds_tree <- function(train, test)
{
  # First get the parent environment to ease command line use.
  env <- parent.frame()
  # Now generate the models for each of the training data lists.
```

```
  env$fits  <- lapply(train, FUN=grow_tree)

  # Next generate predictions from each model using the test data.
  env$preds <- lapply(fits, FUN=test_tree, test_data=test)

  # Generate a list of confusion matrices.
  env$conf_mats <- lapply(preds, FUN=gen_tree_conf_mat, test_data=test)
}

# 12

## "Prune Tree"
## Given a tree model, prune the tree.
prune_tree <- function(fit)
{
  # The prune cp parameter or "Complexity Parameter" is the measure
  # to use to prune on.  Here we decide which to use based on which
  # has the smallest cross-validation error.
  prune(fit, cp=fit$cptable[which.min(fit$cptable[,"xerror"]), "CP"])
}

# 13

## "Get Random Forest Confusion Matrix"
## A function to extract the confusion matrices from the random
## forest object returned by randomforest.
get_rf_conf_mat <- function(rf_object)
{
  conf<-as.array(rf_object$confusion)
  p<-conf[,c(1,2)]
  conf_mat<-matrix(c(p[1],p[2], p[3], p[4]), nrow=2, ncol=2)
  conf_mat
}

# 14

## "Write Confusion matrix"
## A function to write a confusion matrix to the results file.
write_confusion_matrix <- function(c_matrix, r_file)
{
  write(c_matrix, file=r_file, ncolumns=2, append=TRUE)
  write("\n", file=r_file, ncolumns=1, append=(TRUE))
}

#15

## "Write Message"
```

```
## Write message to file.
write_message <- function(msg, r_file)
{
  write(msg, file=r_file, append=TRUE)
}

#16

## "Load Data"
## Runs data_setup.R
load_data <- function(data_file)
{
if(is.null(data_file))
{
  data_file<-'../Data/agaricus-lepiota.data'
}

## Get the parent environment
env <- parent.frame()
## Load Data
env$mushrooms=read.csv(data_file, header=TRUE, sep=",")

## Specify Some Variables
n_folds <- 5
index_folds <- list()
env$folds <- list()

n_entries_per_fold <- floor(nrow(env$mushrooms)/(n_folds))

## Generate the indices we will use to segment the data.
all_indices <- seq_len(nrow(mushrooms))
while( length(all_indices) > n_entries_per_fold)
{
  temp <- sample(all_indices, size = n_entries_per_fold)
  all_indices <- setdiff(all_indices, temp)
  index_folds <- c(index_folds, list(temp))
}

## At this point there are a few indices that were not used.  These
## indices are added to the index_folds vectors starting at vector 1.
i = 1
while( length(all_indices) > 0 )
{
  index_folds[[i]] <- append( index_folds[[i]], all_indices[1])
  all_indices <- setdiff(all_indices, all_indices[1])
  i = i+1
  if( i > n_folds ) i = 1
```

```
}

## Now we have a list of vectors such that all the vectors contain
## all of indices of the data set, all the vectors are
## mutually disjoint (no repeates among them), and all of them are
## randomly selected.  Now, using these sets of indices, we will
## subset the data into ten subsets.
  env$folds <- list(data.frame(mushrooms[index_folds[[1]], ]), data.frame(mushr
                data.frame(mushrooms[index_folds[[3]], ]), data.frame(mushrooms
                data.frame(mushrooms[index_folds[[5]], ]))
                #, data.frame(mushrooms[index_folds[[6]], ]),
                #data.frame(mushrooms[index_folds[[7]], ]), data.frame(mushroom
                #data.frame(mushrooms[index_folds[[9]], ]), data.frame(mushroom


# Now the i'th fold can be accessed as a list item by: folds[[i]]
# categories can be accessed by: folds[[i]]$category_name
}



source('Functions.R')
#############################################################################
#           data_setup.R
#############################################################################
#
#
#
#     ###     Description         ###
#
#   Partitions the data into k-index_folds which can be used
# with any of the various models we might want to try out with the
#   data.  The goal of this is to be able to run this, and to initiate
#   the data into the R-workspace so models can be trained and tested
#   using the data.
#
#     ###     By              ###
#
# Written by Christopher Carlson
#
#     ###     For             ###
#
# Written initially for Western Michigan University's Summer 1 2015
#     Semester course, CS 5950 - Machine Learning.
#
#########                           ##########
```

```
# Load Data
mushrooms=read.csv("../Data/agaricus-lepiota.data", header=TRUE, sep=",")


# Specify Some Variables
n_folds <- 3
index_folds <- list()
folds <- list()

n_entries_per_fold <- floor(nrow(mushrooms)/(n_folds))

# Generate the indices we will use to segment the data.
all_indices <- seq_len(nrow(mushrooms))
while( length(all_indices) > n_entries_per_fold)
{
  temp <- sample(all_indices, size = n_entries_per_fold)
  all_indices <- setdiff(all_indices, temp)
  index_folds <- c(index_folds, list(temp))
}

  ## At this point there are a few indices that were not used.  These
  ## indices are added to the index_folds vectors starting at vector 1.
i = 1
while( length(all_indices) > 0 )
{
  index_folds[[i]] <- append( index_folds[[i]], all_indices[1])
  all_indices <- setdiff(all_indices, all_indices[1])
  i = i+1
  if( i > 10 ) i = 1
}

  ## Now we have a list of vectors such that all the vectors contain
  ## all of indices of the data set, all the vectors are
  ## mutually disjoint (no repeates among them), and all of them are
  ## randomly selected.  Now, using these sets of indices, we will
  ## subset the data into ten subsets.

  folds <- list(data.frame(mushrooms[index_folds[[1]], ]), data.frame(mushrooms
                data.frame(mushrooms[index_folds[[3]], ]))#
                # , data.frame(mushrooms[index_folds[[4]], ]),
                # data.frame(mushrooms[index_folds[[5]], ]))
                # , data.frame(mushrooms[index_folds[[6]], ]),
                # data.frame(mushrooms[index_folds[[7]], ]), data.frame(mushroo
                # data.frame(mushrooms[index_folds[[9]], ]), data.frame(mushroo
```

```r
# Now the i'th fold can be accessed as a list item by: folds[[i]]
# categories can be accessed by: folds[[i]]$category_name



source('Functions.R')
#########################################################################
#           classification_tree.R
#########################################################################
#
#
#
#     ###     Description         ###
# Runs the classification tree algorithm k-folds time and performs
# k-folds cross validation on the results.  Writes the results to
# the file "classification_tree_cv_results.txt" as confusion matrices.
#
#     ###     By              ###
#
# Written by Christopher Carlson
#
#     ###     For             ###
#
# Written initially for Western Michigan University's Summer 1 2015
#   Semester course, CS 5950 - Machine Learning.
#
#########                               ##########


# This matrix will hold the final results after running the complete
# cross validation.
confusion_matrix_averages <- vector("list", (length(folds)-1))

results_file <- "results/tree_classification.txt"

# In this section of the code, the goal is to cross-validate over each
# of the folds.  So for i in nFolds, it will make the i'th fold the
# testing data, and it will build a tree from each of the remaining
# folds.  Then it will test the fit of each tree on the test fold.
#
# The results of each test are added to a list, and finally at the
# end all the average test perfomance is calculated and reported.
for( i in 1:(length(folds)))
{
  # Name i'th fold 'test' and add the
```

```
  # remaining folds to a list called 'train'
  test  <- folds[[i]]
  train <- folds[c(seq(1:(length(folds))))[-i]]

  # First generate the models for each of the training data lists.
  fits  <- lapply(train, FUN=grow_tree)

  # Next generate predictions from each model using the test data.
  preds <- lapply(fits, FUN=test_tree, test_data=test)

  # Generate a list of confusion matrices.
  conf_mats <- lapply(preds, FUN=gen_tree_conf_mat, test_data=test)

  # Write results to file.
  write_message(paste("Iteration #", i, ":\n", sep=""), results_file)
  lapply(conf_mats, FUN=write_confusion_matrix, r_file=results_file)

  # Add the average confusion matrix for this iteration to the
  # confusion_matrix_averages list.
  confusion_matrix_averages[[i]]<- apply(simplify2array(conf_mats),
      c(1,2), mean)
}
# Write the results averages to the results file.
write_message("Averages of each CV iteration:\n", results_file)
lapply(confusion_matrix_averages, FUN=write_confusion_matrix, r_file=results_fi


# Finally, generate a single confusion matrix from all the average
# confusion matrices and write it to results.
final_confusion_matrix <- apply(simplify2array(confusion_matrix_averages),
              c(1,2), mean)
write_message(paste0("Overall Test Error for ", length(folds),
          "-folds cross validation:\n"), results_file)
write_confusion_matrix(final_confusion_matrix, results_file)



source('Functions.R')
#############################################################################
#           random_forest.R
#############################################################################
#
#
#
#     ###     Description         ###
# Runs k-folds cross validation on random forest classifier for
# mushroom data k times and displayes the results in the file
```

```
#   'rand_forest_results.txt'.  (This is major overkill because
# the package 'random-forest' conducts it's own cross validation
# as a part of model generation, but I am doing it explicitly to
#   demonstrate the results over k-folds first hand.)
#
#     ###     By              ###
#
# Written by Christopher Carlson
#
#     ###     For             ###
#
# Written initially for Western Michigan University's Summer 1 2015
#     Semester course, CS 5950 - Machine Learning.
#
#########                           ##########

results_file='results/rf_results.txt'

# This matrix will hold the final results after running the complete
# cross validation.
confusion_matrix_averages <- vector("list", (length(folds)-1))



# Begin results file.
write("\t\tBEGIN RANDOM FOREST RESULTS\n", file=results_file,
    ncolumns = 1, append=FALSE)


# In this section of the code, the goal is to cross-validate over each
# of the folds.  So for i in nFolds, it will make the i'th fold the
# testing data, and it will build a tree from each of the remaining
# folds.  Then it will test the fit of each tree on the test fold.
#
# The results of each test are added to a list, and finally at the
# end all the average test perfomance is calculated and reported.
for( i in 1:(length(folds)))
{
  # Name i'th fold 'test' and add the
  # remaining folds to a list called 'train'
  test  <- folds[[i]]
  train <- folds[c(seq(1:(length(folds))))[-i]]



  # First generate the models for each of the training data lists.
  fits  <- lapply(train, FUN=grow_forest, test_data=test)
```

```
  # Generate a list of confusion matrices.
  conf_mats <- lapply(fits, FUN=get_rf_conf_mat)

  # Add the average confusion matrix for this iteration to the
  # confusion_matrix_averages list.
  confusion_matrix_averages[[i]]<- apply(simplify2array(conf_mats),
      c(1,2), mean)

  # Print the confusion matrices to the file.
  write_message(paste0("Iteration ", i, ":\n"), results_file)
  lapply(conf_mats, FUN=write_confusion_matrix, r_file=results_file)

 }

# Write the average confusion matrices to the results file.
write_message("Averages of each CV iteration:\n", results_file)
lapply(confusion_matrix_averages, FUN=write_confusion_matrix, r_file=results_fi

# Finally, generate a single confusion matrix from all the average
# confusion matrices and write it to file.
final_confusion_matrix <- apply(simplify2array(confusion_matrix_averages),
              c(1,2), mean)
write_message(paste0("Average Confusion Matrix from ", length(folds),
          "-folds cross validation:\n"), results_file)
write_confusion_matrix(final_confusion_matrix, results_file)
```

```r
### init.R

library(MASS)
mush0 = read.csv("mydata.csv")
mush = mush0[,-17]


### util.R

cmp = function(data, cIndex) {
  print(paste("colname:", names(data)[cIndex], " [edible,poisonous]"))
  print(summary(data[which(data$class=="e"),cIndex]))
  print(summary(data[which(data$class=="p"),cIndex]))
}


### myclass.R

#myclass = rep("?", nrow(mush))

#for (i in 1:nrow(mush))
#{
# if (mush[i,]$odor %in% c("a","l") | mush[i,]$spore.print.color %in% c("b
# {
#   myclass[i] = "e"
# }
# else if (mush[i,]$odor %in% c("c","f","m","p","s","y") | mush[i,]$spore.
# {
#   myclass[i] = "p"
# }
#}

myclass = ifelse(mush$odor %in% c("a","l") | mush$spore.print.color %in% c
  ifelse(mush$odor %in% c("c","f","m","p","s","y") | mush$spore.print.colo
    ifelse(mush$spore.print.color != "w", "e",
      ifelse(mush$gill.size=="b", "e",
        ifelse(mush$gill.spacing=="c" | mush$stalk.surface.above.ring=="k'
          ifelse(mush$population=="c", "p",
"?"
          )
        )
      )
    )
  )
)
```

```r
myclass = factor(myclass)
foo = mush[which(myclass=="?"),]

print("-------------------------------------------------------------")
for (i in (1:ncol(mush))[-c(1,17)])
{
  cmp(foo,i)
}

print(table(myclass, mush$class))

### br.R

start = Sys.time()
print(start)

# Constants
nCol = ncol(mush)
index = 1:nrow(mush)
k = 5

# Feature value summary table setup.
setsize = 1:nCol
for (i in 1:nCol)
{
  setsize[i] = length(unique(mush[,i]))
}
maxFcnt = max(setsize)

# Results structures.
preds = data.frame(mush$class,mush$class,mush$class,mush$class,mush$class,
names(preds) = c("real", "nbc", "lda", "qda", "log.reg", "fold")

# Set of test features. (not including class)
testFeatureSet = 2:nCol

for (fold in 0:(k-1))
{
  # Partition Data for NBC
  test = mush[which(index %% k == fold),]
  train = mush[which(index %% k != fold),]
  traine = train[which(train$class == "e"),]
  trainp = train[which(train$class == "p"),]

  # Initialize results structures.
```

```
  test.preds = preds[which(index %% k == fold),]
  test.preds$fold = fold
  test.preds$real = test$class

  pe = log(nrow(traine)) # P(edible)
  pp = log(nrow(trainp)) # P(poisonous)

  # Create concise matrix of log(P(feature value|class)) estimates
  pfe = matrix(nrow=nCol, ncol=maxFcnt)
  pfp = matrix(nrow=nCol, ncol=maxFcnt)
  eFeatValLogit = matrix(nrow=nCol, ncol=maxFcnt)
  for (c in 1:nCol)
  {
    l = levels(mush[1,c])
    for (v in 1:length(l))
    {
      valueTotal = max(log(sum(train[,c] == l[v])), 0) * 100
      pfe[c,v] = max(log(sum(traine[,c] == l[v])) , -valueTotal)
      pfp[c,v] = max(log(sum(trainp[,c] == l[v])) , -valueTotal)
    }
  }
  # Logit(ish) Estimate for numerical methods
  eFeatValLogit = pfe - pfp # Matrix subtraction

  # Create probabalistic data table.
  mush.eprob = data.frame(mush[,1],rep(list(rep(-1,nrow(mush))),nCol))
  names(mush.eprob) = names(mush)
  for (c in 2:nCol)
  {
    # Fancy vector method for setting a column of values at a time.
    mush.eprob[,c] = eFeatValLogit[(as.integer(mush[,c])-1) * nCol + c]
  }
  test.eprob = mush.eprob[which(index %% k == fold),]
  train.eprob = mush.eprob[which(index %% k != fold),]

  # NBC
  res = rep("?", nrow(test))
  for (ti in 1:nrow(test))
  {
    pi.e = pe + sum(pfe[(as.integer(test[ti,testFeatureSet])-1)*nCol + tes
    pi.p = pp + sum(pfp[(as.integer(test[ti,testFeatureSet])-1)*nCol + tes

    res[ti] = ifelse(pi.e > pi.p, "e", "p")
  }
  test.preds$nbc = factor(res, levels=levels(mush[1,1]))
```

```
136
137   f = as.formula(paste("class~",paste(names(mush)[testFeatureSet], collaps
138   # LDA
139   lda.fit = lda(f, data=train.eprob)
140   test.preds$lda = predict(lda.fit, test.eprob)$class
141
142   # QDA
143   qda.fit = qda(f, data=train.eprob)
144   test.preds$qda = predict(qda.fit, test.eprob)$class
145
146   # Logistic Regression
147   glm.fit = glm(f, family=binomial, data=train.eprob)
148   glm.prob = predict(glm.fit, test.eprob, type="response")
149   test.preds$log.reg = factor(ifelse(glm.prob < .5, "e", "p"), levels=leve
150
151   preds[which(index %% k == fold),] = test.preds
152
153 }
154 print(table(preds$nbc, preds$real))
155 print(table(preds$lda, preds$real))
156 print(table(preds$qda, preds$real))
157 print(table(preds$log.reg, preds$real))
158 print(Sys.time() - start)
159
160
161 ### nbc.R
162
163 mylog = log
164
165 index = 1:nrow(mush)
166 k = 5
167 caution = .5
168 cautionOff = 0
169 cautionRange = .01
170 setsize = 1:ncol(mush)
171 for (i in 1:ncol(mush))
172 {
173   setsize[i] = length(unique(mush[,i]))
174 }
175 maxFcnt = max(setsize)
176
177 pred = factor(c("e", "p"))
178 real = factor(c("e", "p"))
179 confuse = table(pred, real)
180 confuse[,] = 0
```

```
181
182 coef = matrix(nrow=nrow(mush), ncol=2)
183 pprob = rep(0, nrow(mush))
184 eprob = pprob
185 realClass = rep('?', nrow(mush))
186 coefi = 0
187
188 for (fold in 0:(k-1))
189 {
190   test = mush[which(index %% k == fold),]
191   train = mush[which(index %% k != fold),]
192   traine = train[which(train$class == "e"),]
193   trainp = train[which(train$class == "p"),]
194
195   pe = mylog(nrow(traine))# * 2
196   pp = mylog(nrow(trainp))# * 2
197
198   pfe = matrix(nrow=ncol(mush), ncol=maxFcnt)
199   pfp = pfe
200   for (c in 1:ncol(mush))
201   {
202     l = levels(mush[1,c])
203     for (v in 1:length(l))
204     {
205       emptyWeight = 0#-max(mylog(sum(train[,c] == l[v])), 0)
206       pfe[c,v] = max(mylog(sum(traine[,c] == l[v])), emptyWeight)
207       pfp[c,v] = max(mylog(sum(trainp[,c] == l[v])), emptyWeight)
208     }
209   }
210
211   res = rep("?", nrow(test))
212   for (ti in 1:nrow(test))
213   {
214     pi.e = pe
215     pi.p = pp
216     #print(paste(pi.e, pi.p))
217     #for (fi in 2:ncol(mush))
218     for (fi in c(6,21))
219     {
220       pi.e = pi.e + pfe[fi,as.integer(test[ti,fi])]
221       pi.p = pi.p + pfp[fi,as.integer(test[ti,fi])]
222     }
223
224     res[ti] = #ifelse(
225       #abs(pi.e - pi.p) < cautionRange
```

```
      #abs(.5 - pi.e / (pi.e + pi.p)) < cautionRange
      #, "?",
      ifelse(pi.e * (1-caution) > pi.p * caution, "e", "p")
    #)

    #coef[coefi,] = c(pi.e / (pi.e + pi.p), ifelse(res[ti] == test[ti,1],
    pprob[coefi] = pi.e
    eprob[coefi] = pi.p
    realClass[coefi] = test[ti,1]
    coefi = coefi + 1
  }
  resf = factor(res)#, levels=c(1,2,3), labels=c("e","p","?"))

  curTab = table(res, test$class)
  print(curTab)
  confuse = confuse + curTab
}
print(confuse)
```