Classification Study of Mushroom Toxicity

Architectural Decisions Document

by

Richard Spencer Landry, Jr., Ph.D.

Contents

1 Introduction and Motivation

Deaths related to food poisoning by eating wild mushrooms still appear in the news, including one recently documented case in Feb. 2019 at a Michelin-starred restaurant in Spain. This project will be using a collection of data with species characteristics to train a model for determining edibility. This project would function as a first test to allow for early detection of poisonous varieties. The motivation for this study is to provide a first automated step to rule out verifiably inedible species a priori. This will aid to expedite and streamline more efficiently the second level of testing by expert connoisseurs in a commercial setting or production line. Mushrooms, especially foraged ones such as truffles and morels sell at very high prices and market value remains fairly constant for the varieties that cannot be commercially mass harvested. The Mushrooms Database contains data about several thousands of types of mushrooms, both edible and poisonous. We will use several predictive models to guess the type (edible or poisonous) of the species. The data contains attributes for the cap (shape, surface, color), gill (attachment, spacing, size, color), bruises, stalk (shape, root, surface above ring, surface below ring, color above ring, color below ring), veil (type, color), ring (number, type), spore print color, population, habitat.

2 Streaming Analytics and Data Integration

Because our product model does not require streaming or time train based data, streaming analytics and the technologies used for them are not implemented in this project. The classification scheme has a set number of parameters and therefore needs only to be run on a new specimen with previous training data. Moreover, we are only using one large collection of data (to be split into a testing and training set) so we will only need to load it into our

analysis software. In this project we will be using a combination of a few languages and systems.

3 Data Repository

The Mushrooms Database has been obtained through `https://archive.ics.uci.edu/ml/datasets/mushroom` and we explained in the introduction the structure of the data. Let's start by reading it in from the current working directory using the R programming language in a Jupyter notebook (see associated notebook for details):

```
library(caret)
library(dplyr)
library(ggplot2)
library(grid)
library(gridExtra)
library(pROC)
library(randomForest)
library(gbm)
library(xgboost)
library(AppliedPredictiveModeling)

raw.data <- read.csv("E:/Documents/R/mushrooms.csv")
print(sprintf("Number of data rows: %d", nrow(raw.data)))
print(sprintf("Number of data columns: %d", ncol(raw.data)))
df <- read.table("mushrooms.csv")
head(df)
```

```
[1] "Number of data rows: 8124"
[1] "Number of data columns: 23"
V1
1 class,cap-shape,cap-surface,cap-color,bruises,odor,gill-attachment,
gill-spacing,gill-size,gill-color,stalk-shape,stalk-root,
stalk-surface-above-ring,stalk-surface-below-ring,
stalk-color-above-ring,stalk-color-below-ring,veil-type,veil-color,
ring-number,ring-type,spore-print-color,population,habitat
2
p,x,s,n,t,p,f,c,n,k,e,e,s,s,w,w,p,w,o,p,k,s,u
```

```
3
e,x,s,y,t,a,f,c,b,k,e,c,s,s,w,w,p,w,o,p,n,n,g
4
e,b,s,w,t,l,f,c,b,n,e,c,s,s,w,w,p,w,o,p,n,n,m
5
p,x,y,w,t,p,f,c,n,n,e,e,s,s,w,w,p,w,o,p,k,s,u
6
e,x,s,g,f,n,f,w,b,k,t,e,s,s,w,w,p,w,o,e,n,a,g
```

This tells us we have 8124 different samples with 23 characteristics per sample. With each sample we have a classification: `edible` or `poisonous`.

4 Discovery and Exploration

We will explore the features using R in the first part of the notebook because the data set is not too large and some of the plot routines are superior to other progamming languages when initially working with the data set. The necessary libraries and utilities can be found in the fully functional notebook associated with this report. Refer to the top of that notebook and be sure those libraries are installed. The field `class` has either an `e` (`edible`) or `p` (`poisonous`) value. We should also check how many species are in each category.

```
1 class <- plyr::count(raw.data$class)
2 print(sprintf("Edible: %d | Poisonous: %d |
3       Percent of poisonous classes: %.1f%%",
4       class$freq[1],class$freq[2],
5       round(class$freq[1]/nrow(raw.data)*100,1)))
```

```
[1] "Edible: 4208 | Poisonous: 3916 | Percent of poisonous classes: 51.8%"
```

We want to create a model that allows us to determine if a mushroom is edible or poisonous, so we must analyze the features and try to understand which have either larger predictive values, or do not bring considerable predictive value.

5 Features Analysis

We reframe the features by converting the factor data into numeric data.

```
m.data = raw.data[,2:23]
m.class = raw.data[,1]
m.data <- sapply( m.data, function (x) as.numeric(as.factor(x))...
    )
```

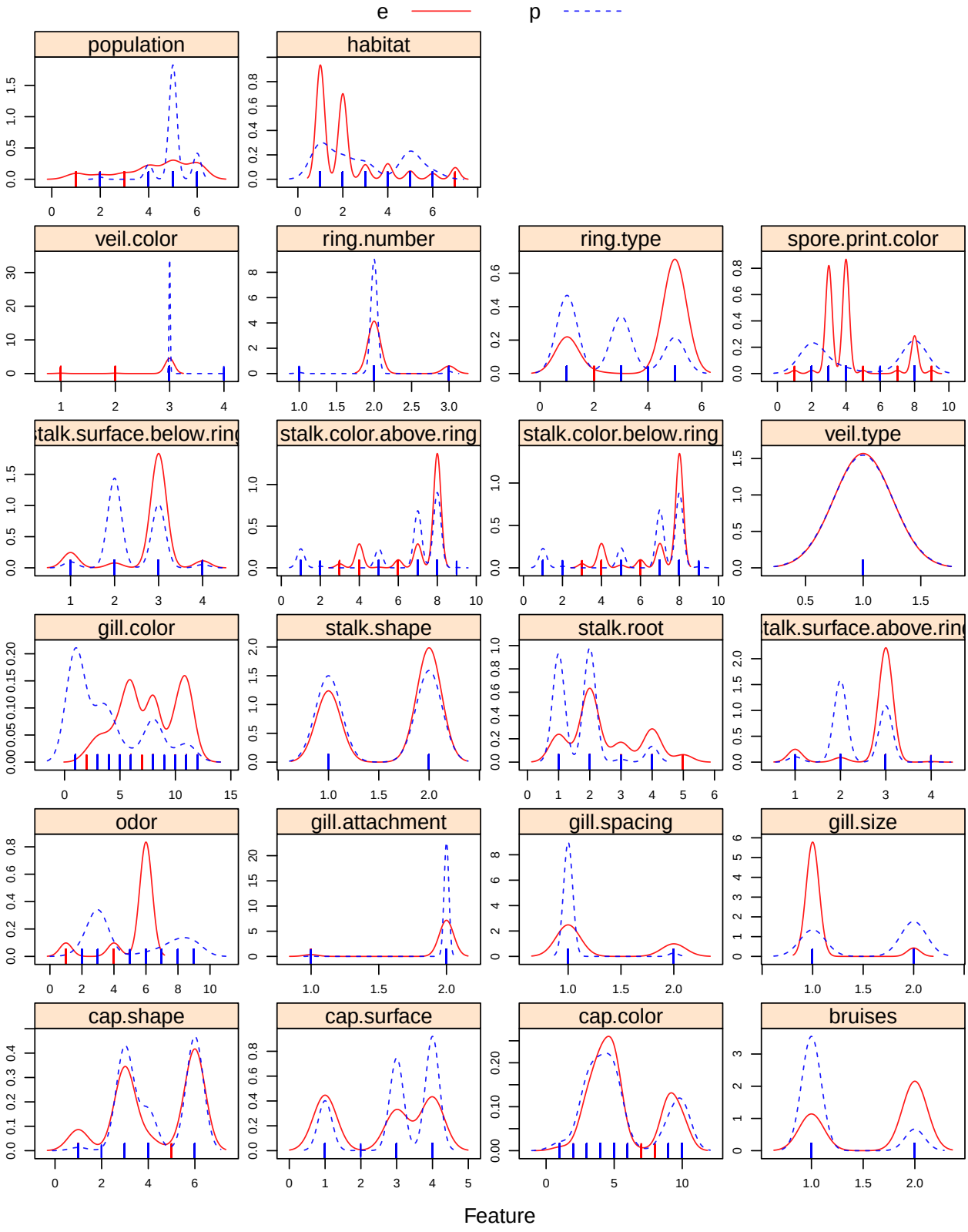
For a visualization of the features we will use density plots to represent both the values density and the degree of separation of the two sets of values, on each feature dimension.

```
scales <- list(x=list(relation="free"),y=list(relation="free"),...
    cex=0.6)
featurePlot(x=m.data, y=m.class, plot="density",scales=scales,
            layout = c(4,6), auto.key = list(columns = 2), pch ...
               = "|")
```

There is no perfect separation between any of the features, as expected, but we do have fairly good separations for spore.print.color, ring.type, population, and habitat. There is also a tight correspondence for some of the values, like veil.type and stalk.shape.

6 Predictive Models

We will use three models in R and one more model using a neural network to determine which is the best way to train the data. In our R implementation we will be implementing a Random Forest Model, a Gradient Boosting Machine Model, and XGBoost, the eXtreme Gradient Boosting Machine Model.


e
p
population
habitat
veil.color
ring.number
ring.type
spore.print.color
talk.surface.below.rin
stalk.color.above.ring
stalk.color.below.ring
veil.type
gill.color
stalk.shape
stalk.root
talk.surface.above.rin
odor
gill.attachment
gill.spacing
gill.size
cap.shape
cap.surface
cap.color
bruises
Feature

1 Random Forest Model

Ensembling is a type of supervised machine learning where multiple models or the same model with different hyperparameters are trained on a dataset with individual outputs combined with a rule to derive the final output. Training observations may differ slightly over sampling but the population remains the same. There could be multiple rules by which outputs are combined, and most commonly used ones are the average, or a decision in terms of categorical output. When different models give us numerical output, in the case of categorical output we decide an output occurring the maximum number of times as the final output.

A Random Forest is an ensembling machine learning algorithm that functions by creating multiple decision trees and then combining the output generated by each of the decision trees. A decision tree is a classification model which works on the concept of information gain at every node. For all the data points, decision tree will try to classify data points at each of the nodes and check for information gain at each node. It will then classify at the node where information gain is maximum. It will follow this process subsequently until all the nodes are exhausted or there is no further information gain. Decision trees are very simple and easy to understand models; however, they have very low predictive power. The Random Forest by combining the output of multiple decision trees and then finally comes up with its own output. Random Forest works like Decision Tress but differ in that they do not select all the data points and variables in each of the trees. It randomly samples data points and variables in each of the tree that it creates and then combines the output at the end. It removes the bias that a decision tree model might introduce in the system. Also, it improves the predictive power significantly. The first model we will try is Random forest. We set the number of trees to 100. For the rest of the parameters, we will keep the default settings from the internal R function.

```
df <- data.frame(sapply(raw.data, function (x) as.numeric(as....
    factor(x))))
df$class <- df$class - 1
nrows <- nrow(df)
set.seed(314)
indexT <- sample(1:nrow(df), 0.7 * nrows)
#separate train and validation set
trainset = df[indexT,]
testset =   df[-indexT,]
n <- names(trainset)
```

```
rf.form <- as.formula(paste("class ~", paste(n[!n %in% "class"...
    ], collapse = " + ")))
trainset.rf <- randomForest(rf.form,trainset,ntree=100,...
    importance=T)
```

To visualize the variable importance with two methods we will look at some plots of IncNodePurity and %IncMSE. Here IncNodePurity is the total decrease in node impurities, measured by the Gini Index from splitting on the variable, averaged over all trees. %IncMSE is the increase in mse of predictions as a result of a variable being permuted (here the values are randomly shuffled).

```
varimp <- data.frame(trainset.rf$importance)
vi1 <- ggplot(varimp, aes(x=reorder(rownames(varimp),...
    IncNodePurity), y=IncNodePurity)) + geom_bar(stat="identity"...
    , fill="green", colour="black") + coord_flip() + theme_bw(...
    base_size = 8) + labs(title="Prediction using RandomForest ...
    with 100 trees", subtitle="Variable importance (...
    IncNodePurity)", x="Variable", y="Variable importance (...
    IncNodePurity)")
vi2 <- ggplot(varimp, aes(x=reorder(rownames(varimp),X.IncMSE),...
     y=X.IncMSE)) + geom_bar(stat="identity", fill="lightblue", ...
    colour="black") + coord_flip() + theme_bw(base_size = 8) + ...
    labs(title="Prediction using RandomForest with 100 trees", ...
    subtitle="Variable importance [%IncMSE]", x="Variable", y="...
    Variable importance [%IncMSE]")
grid.arrange(vi1, vi2, ncol=2)
```

We observe that odor, gill.size, gill.color, spore.print.color, ring.type, population, stalk.root, and gill.spacing are the most important features. Now we present teh test data to the model.

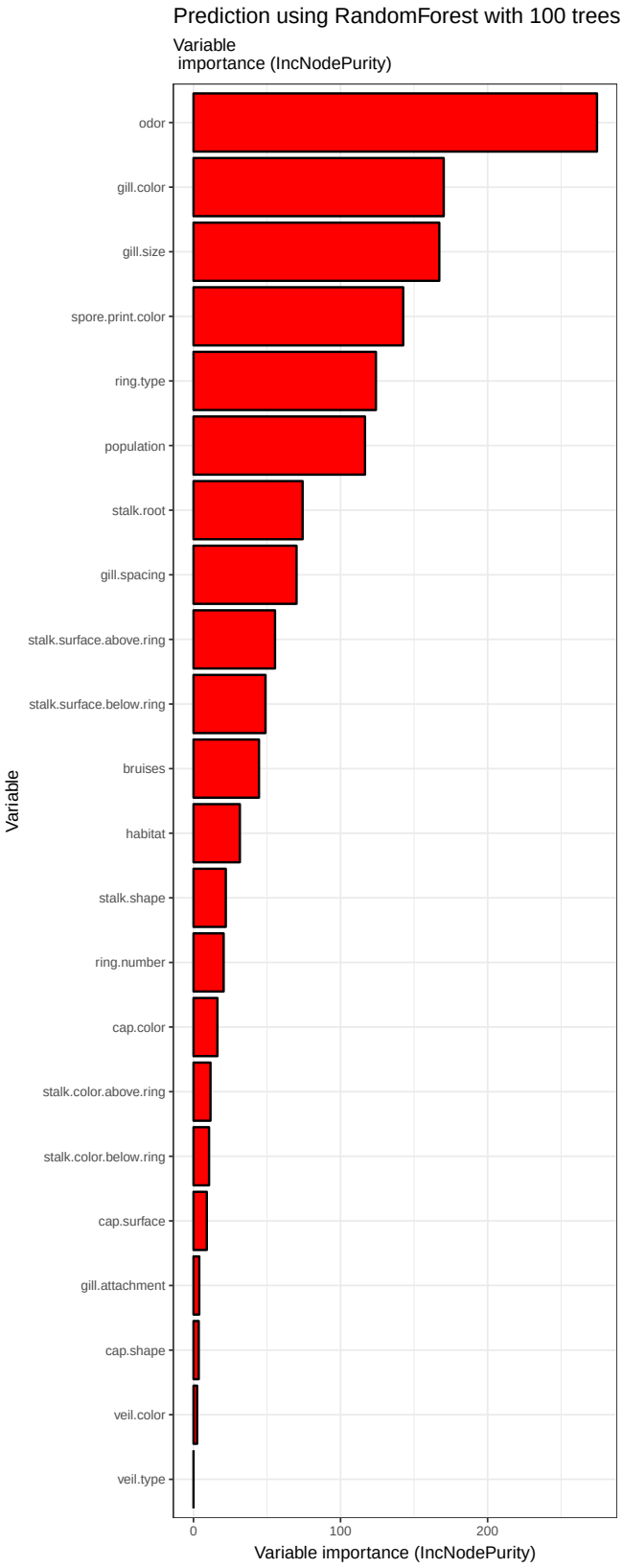
```
testset$predicted <- round(predict(trainset.rf ,testset),0)
testset_rf <- testset$predicted;
```

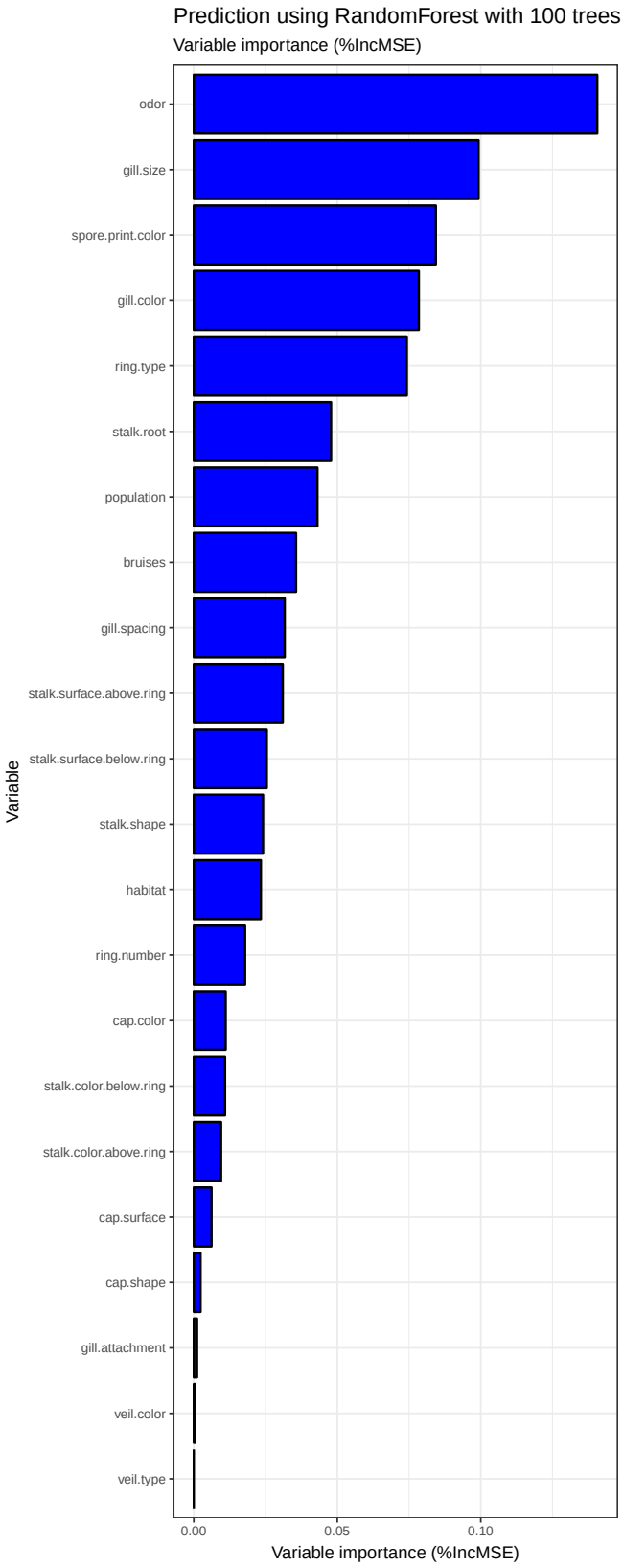
In a two-class problem, we are often looking to discriminate between observations with a specific outcome, from normal observations, such as a disease state or event from no disease state or no event. In our case we are looking at poisonous, nonpoisonous. In this way, we can assign the event row as "positive" and the no-event row as "negative." We can then assign the event column of predictions as "true" and the no-event as "false".

This gives us "true positive" for correctly predicted event values, "false positive" for incorrectly predicted event values, "true negative" for correctly predicted no-event values, and "false negative" for incorrectly predicted no-event values.

Let's visualize the confusion matrix, to see how accurate the results were that we obtained.

```
plotConfusionMatrix <- function(testset, sSubtitle) {
    tst <- data.frame(testset$predicted, testset$class)
    opts <- c("Predicted", "True")
```


Prediction using RandomForest with 100 trees
Variable
importance (IncNodePurity)
odor
gill.color
gill.size
spore.print.color
ring.type
population
stalk.root
gill.spacing
stalk.surface.above.ring
stalk.surface.below.ring
bruises
habitat
stalk.shape
ring.number
cap.color
stalk.color.above.ring
stalk.color.below.ring
cap.surface
gill.attachment
cap.shape
veil.color
veil.type
Variable
0
100
200
Variable importance (IncNodePurity)

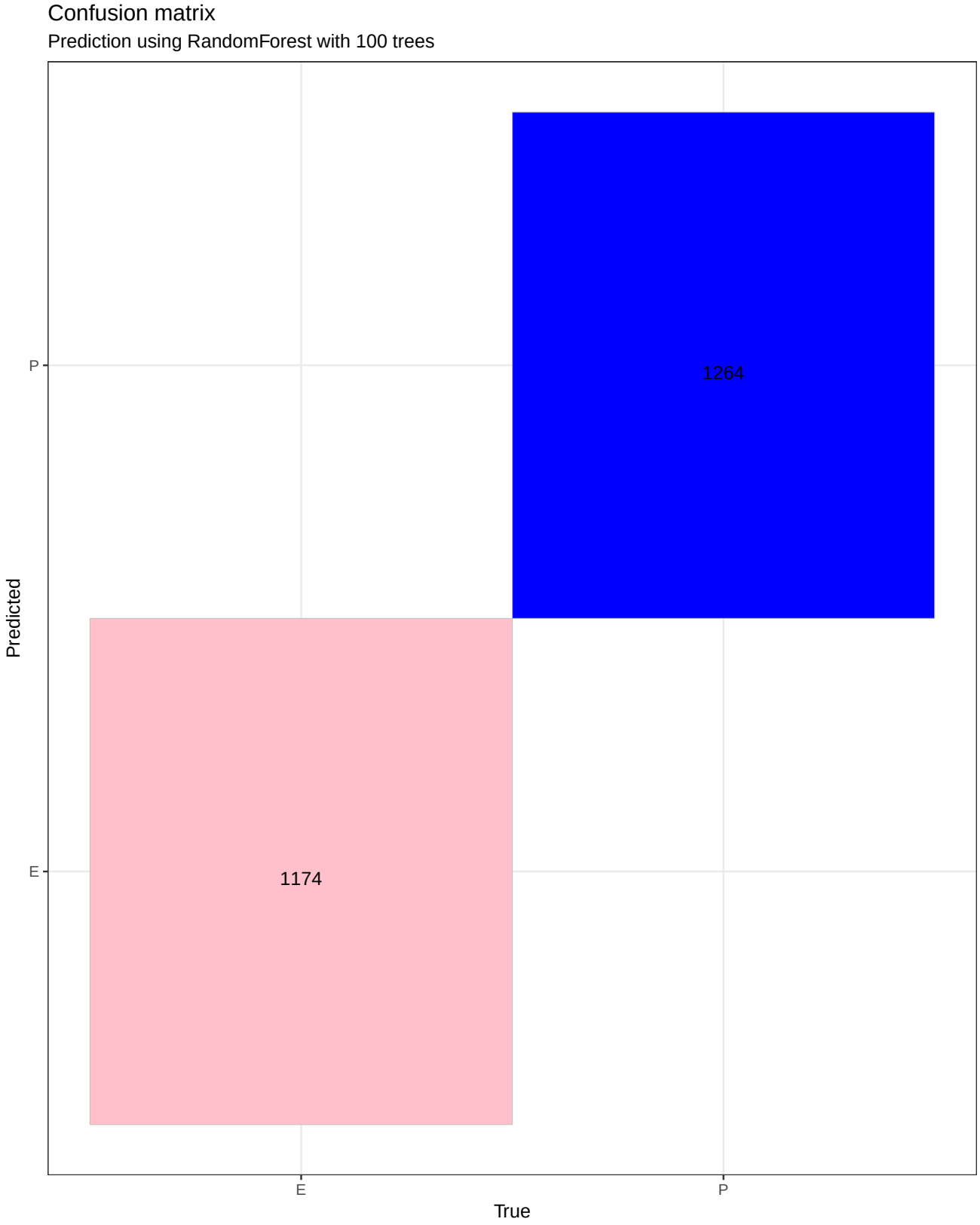

Prediction using RandomForest with 100 trees
Variable importance (%IncMSE)
odor
gill.size
spore.print.color
gill.color
ring.type
stalk.root
population
bruises
gill.spacing
stalk.surface.above.ring
stalk.surface.below.ring
stalk.shape
habitat
ring.number
cap.color
stalk.color.below.ring
stalk.color.above.ring
cap.surface
cap.shape
gill.attachment
veil.color
veil.type
Variable
0.00
0.05
0.10
Variable importance (%IncMSE)

```
 4     names(tst) <- opts
 5     cf <- plyr::count(tst)
 6     cf[opts][cf[opts]==0] <- "P"
 7     cf[opts][cf[opts]==1] <- "E"
 8
 9     ggplot(data =  cf, mapping = aes(x = True, y = Predicted)) ...
         +
10       labs(title = "Confusion matrix", subtitle = sSubtitle) +
11       geom_tile(aes(fill = freq), colour = "grey") +
12       geom_text(aes(label = sprintf("%1.0f", freq)), vjust = 1)...
           +
13       scale_fill_gradient(low = "lightblue", high = "blue") +
14       theme_bw() + theme(legend.position = "none")
15 }
16 plotConfusionMatrix(testset,"Prediction using RandomForest with...
     100 trees")
```

Let's calculate as well the AUC for the prediction.

```
1 print(confusionMatrix(factor(testset$class), factor(testset$...
    predicted)))
2 print(sprintf(
3     "Area under curve (AUC) : %.3f",
4     auc(testset$class, testset$predicted)
5 ))
```

```
Confusion Matrix and Statistics

          Reference
Prediction    0    1
         0 1264    0
         1    0 1174

               Accuracy : 1
                 95% CI : (0.9985, 1)
    No Information Rate : 0.5185
    P-Value [Acc > NIR] : < 2.2e-16

                  Kappa : 1

 Mcnemar's Test P-Value : NA

            Sensitivity : 1.0000
```


Confusion matrix
Prediction using RandomForest with 100 trees
Predicted
P
E
1264
1174
E
P
True

```
           Specificity : 1.0000
        Pos Pred Value : 1.0000
        Neg Pred Value : 1.0000
            Prevalence : 0.5185
        Detection Rate : 0.5185
  Detection Prevalence : 0.5185
     Balanced Accuracy : 1.0000

      'Positive' Class : 0

[1] "Area under curve (AUC) : 1.000"
```

2 Gradient Boosting Machine (GBM) Model

Gradient boosted machines (GBMs) are a machine learning algorithm that have proven successful across many domains. Whereas random forests build an ensemble of deep independent trees, GBMs build an ensemble of shallow and weak successive trees with each tree learning and improving on the previous. When combined, these many weak successive trees produce a powerful collective that are often hard to beat with other algorithms. Some setbacks are that GBMs will continue improving to minimize all errors. This can overemphasize outliers and cause overfitting so we need to use cross-validation to neutralize. The main idea of boosting is to add new models to the ensemble sequentially. At each particular iteration, a new weak, base-learner model is trained with respect to the error of the whole ensemble learned so far. Gradient boosting is considered a gradient descent algorithm that can be performed on any loss function that is differentiable. Consequently, this allows GBMs to optimize different loss functions as desired. An important parameter in gradient descent is the size of the steps which is determined by the learning rate. If the learning rate is too small, then the algorithm will take many iterations to find the minimum. On the other hand, if the learning rate is too high, you might jump cross the minimum and end up further away than when you started. We will prepare a simple GBM model and use cross validation with 5 folds.

```
1 n<-names(trainset)
2 gbm.form <- as.formula(paste("class ~", paste(n[!n %in% "class"...
    ], collapse = " + ")))
3 gbmCV = gbm(formula = gbm.form,
4               distribution = "bernoulli",
5               data = trainset,
6               n.trees = 500,
7               shrinkage = .1,
8               n.minobsinnode = 15,
9               cv.folds = 5,
```

```
10                  n.cores = 1)
```

In order to find the best number of trees to use for the prediction for the test data, we can use `gbm.perf` function. This function returns the optimal number of trees for prediction.

```
1 optimalTreeNumberPredictionCV = gbm.perf(gbmCV)
```

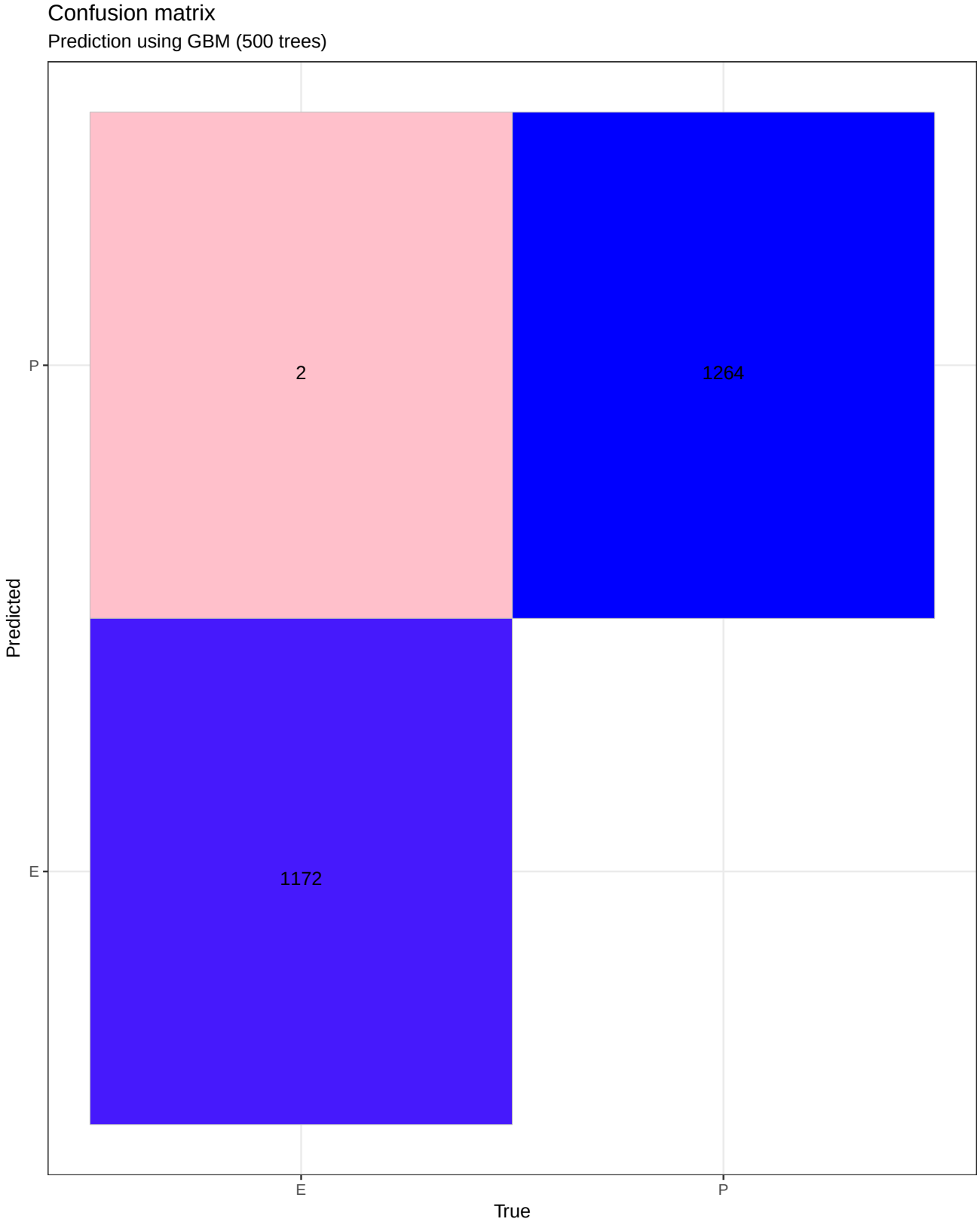
```
1 gbmTest = predict(object = gbmCV,
2                           newdata = testset,
3                           n.trees = ...
                               optimalTreeNumberPredictionCV,
4                           type = "response")
5 testset$predicted <- round(gbmTest,0)
6 testset_gbm <- testset$predicted
```

```
1 plotConfusionMatrix(testset,sprintf("Prediction using GBM (%d ...
    trees)",optimalTreeNumberPredictionCV))
```

Let's calculate as well the AUC for the prediction.

```
1 print(confusionMatrix(factor(testset$class), factor(testset$...
    predicted)))
2 print(sprintf(
3     "Area under curve (AUC) : %.3f",
4     auc(testset$class, testset$predicted)
5 ))
```

```
Confusion Matrix and Statistics

          Reference
Prediction    0    1
         0 1264    0
         1    2 1172

               Accuracy : 0.9992
                 95% CI : (0.997, 0.9999)
    No Information Rate : 0.5193
    P-Value [Acc > NIR] : <2e-16

                  Kappa : 0.9984
```


Confusion matrix
Prediction using GBM (500 trees)
Predicted
True
P
E
2
1264
1172
E
P

```
 Mcnemar's Test P-Value : 0.4795

            Sensitivity : 0.9984
            Specificity : 1.0000
         Pos Pred Value : 1.0000
         Neg Pred Value : 0.9983
             Prevalence : 0.5193
         Detection Rate : 0.5185
   Detection Prevalence : 0.5185
      Balanced Accuracy : 0.9992

       'Positive' Class : 0

[1] "Area under curve (AUC) : 0.999"
```

3 Extreme Gradient Boosting (XGBoost) Model

Let's try now the XGBoost model. We prepare the data to run the model. We create `xgb.DMatrix` objects for each trainand test set.

```
dMtrain <- xgb.DMatrix(as.matrix(trainset %>% select(-class)), ...
    label = trainset$class)
dMtest <- xgb.DMatrix(as.matrix(testset %>% select(-class,-...
    predicted)), label = testset$class)
```

We set the XGBoost parameters for the model. We will use a binary logistic objective function. The evaluation metric will be AUC (Area under curve) as before.

```
params <- list(
  "objective"           = "binary:logistic",
  "eval_metric"         = "auc",
  "eta"                 = 0.012,
  "subsample"           = 0.8,
  "max_depth"           = 8,
  "colsample_bytree"    = 0.9,
  "min_child_weight"    = 5
)
```

We train the model using cross validation with 5 folds as before. We are using a number of rounds equal with 5000, with early stopping criteria for 100 steps. We are also setting the frequency of printing partial results every 100 steps.

```
nRounds <- 5000
earlyStoppingRound <- 100
printEveryN = 100
model_xgb.cv <- xgb.cv(params=params,
                       data = dMtrain,
                       maximize = TRUE,
                       nfold = 5,
                       nrounds = nRounds,
                       nthread = 3,
                       early_stopping_round=earlyStoppingRound,
                       print_every_n=printEveryN)
```

The AUC for train and test set obtained using the training with cross validation have

```
model_xgb <- xgboost(params=params,
                       data = dMtrain,
                       maximize = TRUE,
                       nrounds = nRounds,
                       nthread = 3,
                       early_stopping_round=earlyStoppingRound,
                       print_every_n=printEveryN)
```

Let's use the model now to predict the test data:

```
testset$predicted <- round(predict(model_xgb ,dMtest),0)
testset_xgboost <- testset$predicted
```

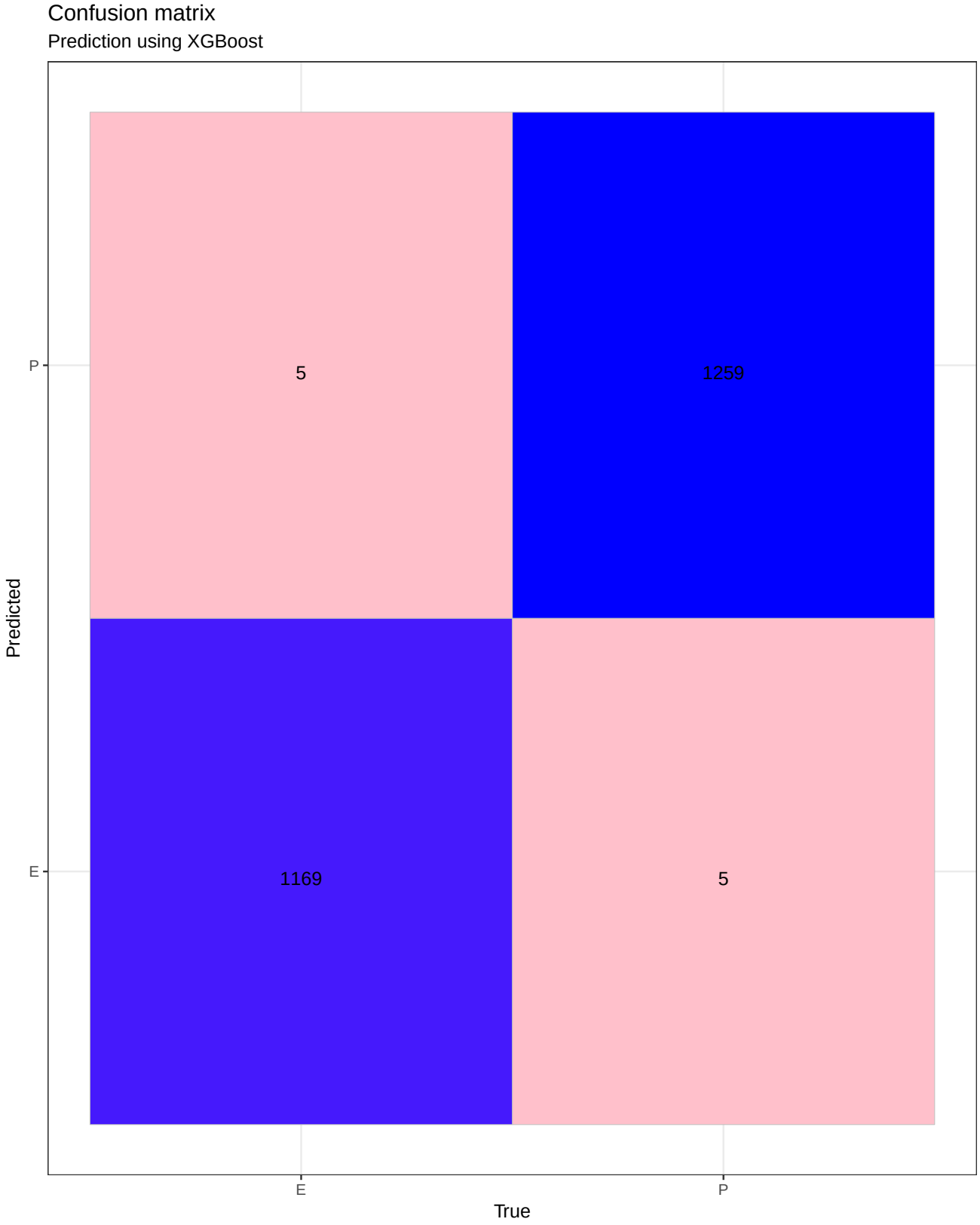
Let's visualize the confusion matrix, to see how accurate are the results we obtained.

```
plotConfusionMatrix(testset,"Prediction using XGBoost")
```

Let's calculate as well the AUC for the prediction.

```
print(confusionMatrix(factor(testset$class), factor(testset$...
    predicted)))
print(sprintf(
    "Area under curve (AUC) : %.3f",
    auc(testset$class, testset$predicted)
))
```

```
Confusion Matrix and Statistics
```


Confusion matrix
Prediction using XGBoost
5
1259
1169
5
P
E
Predicted
E
P
True

```
          Reference
Prediction    0    1
         0 1259    5
         1    5 1169

               Accuracy : 0.9959
                 95% CI : (0.9925, 0.998)
    No Information Rate : 0.5185
    P-Value [Acc > NIR] : <2e-16

                  Kappa : 0.9918

 Mcnemar's Test P-Value : 1

            Sensitivity : 0.9960
            Specificity : 0.9957
         Pos Pred Value : 0.9960
         Neg Pred Value : 0.9957
             Prevalence : 0.5185
         Detection Rate : 0.5164
   Detection Prevalence : 0.5185
      Balanced Accuracy : 0.9959

       'Positive' Class : 0

[1] "Area under curve (AUC) : 0.996"
```

4 Deep Learning Neural Network Model

For this final example we will implement a deep learning model with a Neural Network to compare and contrast against our previous model choices. Instead of using R for this model we will be using the Tensorflow backend under the Keras framework in python, so first we need to import the required libraries into our notebook. Once this is done, any number of optimizations can be called if necessary. For the purposes of this project we will only look at the "adam" optimization, which is a variation on a stochastic gradient optimizing scheme.

Keras is a high-level neural network API which is written in Python. It is capable of running on top of Tensorflow. Keras can be used as a deep learning library. It supports convolutional and recurrent Neural Networks. Prototyping with Keras is fast and easy. We first import the keras library to create the neural network layers.

```
1 import numpy as np
```

```
from sklearn.ensemble import RandomForestClassifier as RFC
from sklearn.tree import DecisionTreeClassifier as DTC
from matplotlib import pyplot as plt
%matplotlib inline
import datetime
import pandas as pd
from sklearn import preprocessing as pp
from sklearn.linear_model import LogisticRegression as LR
from sklearn.neighbors import KNeighborsClassifier as KNN

import keras
from keras.models import Sequential
from keras.layers import Dense, Activation
from keras.utils import np_utils
```

We will use Sequential in Keras to build our neural network. We use Dense library to build input, hidden and output layers of a neural network. ReLu will be the activation function for hidden layers. We will use softmax as the activation function. Kernel is the weight matrix. Its initialization defines the way to set the initial random weights of Keras layers. Random normal initializer generates tensors with a normal distribution. For uniform distribution, we can use Random uniform initializers. Keras provides multiple initializers for both kernel or weights as well as for bias units.

```
ds = pd.read_csv('mushrooms.csv')
dat = ds.values
headers = list(ds.columns.values)
#Data Preprocessing
l = pp.LabelEncoder()
l.fit(dat[:, 0])
dataa = l.transform(dat[:, 0])

for ix in range(1, dat.shape[1]):
    le = pp.LabelEncoder()
    le.fit(dat[:, ix])
    y = le.transform(dat[:, ix])
    dataa = np.vstack((dataa , y))

data = dataa.T
cate = data[:, 0] #One hot encoding for Neural Network ...
    implementation
split = int(0.80 * data.shape[0])

x_train = data[:split , 1:]
y_train = data[:split, 0]

```

```
x_test = data[split: , 1:]
y_test = data[split: , 0]
y = np_utils.to_categorical(cate)
Y_train = y[:split]
Y_test = y[split:]

print(x_train.shape, x_test.shape)
print(y_train.shape, y_test.shape)
print(data.shape)
```

We perform some preprocessing on the data as before and view the output, noting thtat we have used 80 percent of the data for training:

```
(6499, 22) (1625, 22)
(6499,) (1625,)
(8124, 23)
```

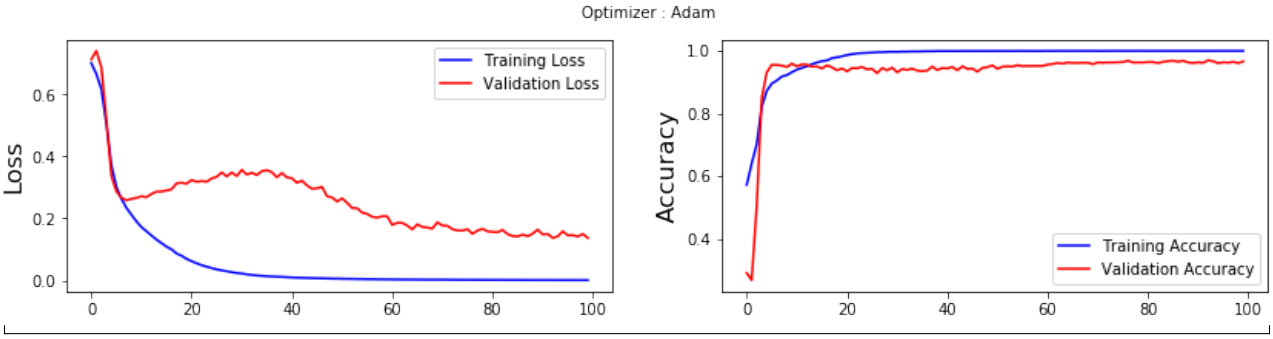
Now we contruct the model with a few hidden layers:

```
model = Sequential()

model.add(Dense(11, input_shape=(22,)))
model.add(Activation('relu'))

model.add(Dense(5))
model.add(Activation('relu'))

model.add(Dense(2))
model.add(Activation('softmax'))

model.summary()
model.compile(loss='categorical_crossentropy', optimizer='adam'...
    , metrics=['accuracy'])
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_13 (Dense)             (None, 11)                253
_________________________________________________________________
activation_13 (Activation)   (None, 11)                0
_________________________________________________________________
dense_14 (Dense)             (None, 5)                 60
_________________________________________________________________
activation_14 (Activation)   (None, 5)                 0
```

```
_________________________________________________________________
dense_15 (Dense)             (None, 2)                 12
_________________________________________________________________
activation_15 (Activation)   (None, 2)                 0
=================================================================
Total params: 325
Trainable params: 325
Non-trainable params: 0
_________________________________
```

```
hist = model.fit(x_train, Y_train,
          epochs=100,
          shuffle=True,
          batch_size=128,
          validation_data=(x_test, Y_test))

plt.figure(figsize=(14,3))
plt.subplot(1, 2, 1)
plt.suptitle('Optimizer : Adam', fontsize=10)
plt.ylabel('Loss', fontsize=16)
plt.plot(hist.history['loss'], 'b', label='Training Loss')
plt.plot(hist.history['val_loss'], 'r', label='Validation Loss'...
    )
plt.legend(loc='upper right')

plt.subplot(1, 2, 2)
plt.ylabel('Accuracy', fontsize=16)
plt.plot(hist.history['acc'], 'b', label='Training Accuracy')
plt.plot(hist.history['val_acc'], 'r', label='Validation ...
    Accuracy')
plt.legend(loc='lower right')
plt.show()
from sklearn.metrics import confusion_matrix,accuracy_score, ...
    roc_curve, auc

#Predict on test set
predictions_NN_prob = model.predict(x_test)
predictions_NN_prob = predictions_NN_prob[:,0]

predictions_NN_01 = np.where(predictions_NN_prob < 0.5, 1, 0) #...
    Turn probability to 0-1 binary output
#Print accuracy
acc_NN = accuracy_score(y_test, predictions_NN_01)
print('Overall accuracy of Neural Network model:', acc_NN)
```

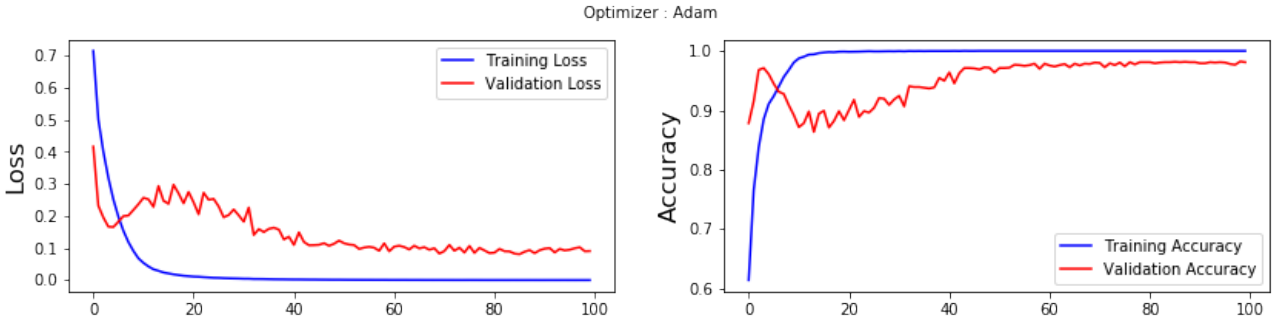



```
Overall accuracy of Neural Network model: 0.9661538461538461
```

From a few tests we found that the overall best situation requires that the hidden layers retain the same number of nodes as the input. After a few tests we view the output below, which is performing better than before.

```
model = Sequential()

model.add(Dense(22, input_shape=(22,)))
model.add(Activation('relu'))

model.add(Dense(22))
model.add(Activation('relu'))


model.add(Dense(2))
model.add(Activation('softmax'))

model.summary()
model.compile(loss='categorical_crossentropy', optimizer='adam'...
    , metrics=['accuracy'])
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_12 (Dense)             (None, 22)                506
_________________________________________________________________
activation_12 (Activation)   (None, 22)                0
_________________________________________________________________
dense_13 (Dense)             (None, 22)                506
_________________________________________________________________
activation_13 (Activation)   (None, 22)                0
_________________________________________________________________
```




```
dense_14 (Dense)             (None, 2)                 46
_________________________________________________________________
activation_14 (Activation)   (None, 2)                 0
=================================================================
Total params: 1,058
Trainable params: 1,058
Non-trainable params: 0
_________________________________________________________________
```

```
Overall accuracy of Neural Network model: 0.9809230769230769
```

7 Actionable Insights / Applications / Data Products

As this is a preliminary study that would need to be fine tuned before porting the fully functional product for end consumers, our project has shown that using the data available is enough to train a Neural Network to predict which mushrooms are poisonous with a great

amount of accuracy. However, it seems at this level of scrutiny our original model using a random forest performs the best.

8 Security, Information Governance, Systems Management, and Future Directions

In this particular case study, the need for security measures on a database lookup table would be minimal. The main database would need only retain the original measurement of classifications and the relevant charactersics used in this experimental model. If a future design would permit updating the data set with new measurements on new specimens determined to be either poisonous or edible, then they would be added to the overall set as needed.

9 Conclusions

We were able to predict with very high accuracy the poisonous and edible mushrooms based on the four models used, `Random Forest`, `Gradient Boosting Machine (GBM)`, `XGBoost`, and finally a neural network using the adam optimizer. For the `GBM` and `XGBoost` we were also using cross validation. The best prediction was obtained using the `Random Forest` model.

1 References

https://machinelearningmastery.com/confusion-matrix-machine-learning/
https://www.r-bloggers.com/how-to-implement-random-forests-in-r/
http://uc-r.github.io/gbm_regression/