Data Science 'Mushrooms' Capstone Project

András Gelencsér

26 December 2019

Contents

Overview

This document describes the result of the capstone project for the HarvardX Data Science course based on the Mushrooms dataset available in the UCI Machine Learning Repository (Dua, D. and Graff, C. (2019). UCI Machine Learning Repository [http://archive.ics.uci.edu/ml]. Irvine, CA: University of California, School of Information and Computer Science.)

The "Mushrooms" dataset was recorded down from The Audubon Society Field Guide to North American Mushrooms (1981). G. H. Lincoff (Pres.), New York: Alfred A. Knopf and is available on the https://archive.ics.uci.edu/ml/datasets/mushroom url.

The goal of this project is to develop a model based on data science and machine learning techniques to predict if a mushroom is eatable based on the known attributes.

The dataset contains 8,124 records and uses 22 attributes to describe the mushrooms. Each dataset contains additional attribute to flag if the mushroom is edible or not.

I used 80% of the available data for train the model. The remaining 20% will be used as test dataset for evaluating the developed model.

For the developed model is very important to detect poisonous mushrooms to avoid possible health damages or death. Therefore, the model performance will be evaluated with the F1 score instead of the overall accuracy. Because the models use the edible prediction as true value, I will try to get a high specificity with the developed models.

The following steps were done during the project for developing the algorithm:

1. Download the data and prepare for the analysis
2. Analyze the data structure
3. Define the classification model based on the result the data analysis
4. Implement and train the model based on the train set
5. Review the model based on the test set

I developed two different prediction model. The first model uses a decision tree, the second model uses a reduced set of the features with high correlation to the edibility.

Both developed models reached an F1 score of 1 with sensitivity 1 and specificity 1. That shows, that both models had a perfect prediction for the test set.

Methods & analysis

Preparation

For the analysis the data we will use the following R packages:

- tidyverse
- caret
- gridExtra
- ggplot2

the following R code loads and install the required packages if needed:

```
if(!require(tidyverse)) install.packages("tidyverse",
                         repos = "http://cran.us.r-project.org", dependencies = TRUE)
if(!require(caret)) install.packages("caret",
                         repos = "http://cran.us.r-project.org", dependencies = TRUE)
if(!require(gridExtra)) install.packages("gridExtra",
                         repos = "http://cran.us.r-project.org", dependencies = TRUE)
if(!require(ggplot2)) install.packages("ggplot2",
                         repos = "http://cran.us.r-project.org", dependencies = TRUE)
```

At first, we need to prepare the data set for the analysis. The mushrooms dataset contains 8,124 records about different mushrooms.

The following R code downloads the data from the https://archive.ics.uci.edu/ml/datasets/mushroom site. To avoid unnecessary data traffic, the data will be downloaded only if not done yet.

```
#create directory for the data file if necessary
if (!dir.exists("mushrooms")){
  dir.create("mushrooms")
}
baseurl = "https://archive.ics.uci.edu/ml/machine-learning-databases/mushroom/"
files_to_download = c("agaricus-lepiota.data", "agaricus-lepiota.names")

for (f in files_to_download){
  #download the files only if not done yet
  if (!file.exists(paste("./mushrooms/",f,sep='')))
  {
    download.file(paste(baseurl,f,sep=''), paste("./mushrooms/",f,sep=''))
  }
}

fl <- file("mushrooms/agaricus-lepiota.data")
mushroom_data <- str_split_fixed(readLines(fl), ",", 23)
close(fl)
```

The dataset is stored in the “agaricus-lepiota.data” file. This file contains 23 columns separated via comma. There is no header line in the file. The description file “agaricus-lepiota.names” gives us information about the classes and the 22 attributes in the data file.

So we can extract the data by parsing the data file. The following R code reads the data and stores in a data frame:

```
fl <- file("mushrooms/agaricus-lepiota.data")
mushroom_data <- str_split_fixed(readLines(fl), ",", 23)
close(fl)
```

```r
#set the column names for the features
colnames(mushroom_data) <- c("classes", "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")
#convert to data frame
mushroom_data <- as.data.frame(mushroom_data)

#remove temporary file variable
rm(fl)
```

For training the classification method we will use 80% of the available data as training set. The remaining data will be used for evaluating the developed method. The following R code creates the train and test set:

```r
#initialize random sequenz
set.seed(1, sample.kind = "Rounding")
#create index for train and test set
#20% of the data will be used for the test set
test_idx = createDataPartition(y = mushroom_data$classes, times=1, p=0.2, list=FALSE)
train_data = mushroom_data[-test_idx,]
test_data = mushroom_data[test_idx,]
#remove temporary variables
rm(mushroom_data, test_idx)
```

Available Data

The dataset contains 8,124 records. For the analysis we have data about 6,500 mushrooms (80% of all data). All the features are categorical with different possible values.

1. cap-shape: bell=b,conical=c,convex=x,flat=f, knobbed=k,sunken=s
2. cap-surface: fibrous=f,grooves=g,scaly=y,smooth=s
3. cap-color: brown=n,buff=b,cinnamon=c,gray=g,green=r, pink=p,purple=u,red=e,white=w,yellow=y
4. bruises?: bruises=t,no=f
5. odor: almond=a,anise=l,creosote=c,fishy=y,foul=f, musty=m,none=n,pungent=p,spicy=s
6. gill-attachment: attached=a,descending=d,free=f,notched=n
7. gill-spacing: close=c,crowded=w,distant=d
8. gill-size: broad=b,narrow=n
9. 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
10. stalk-shape: enlarging=e,tapering=t
11. stalk-root: bulbous=b,club=c,cup=u,equal=e, rhizomorphs=z,rooted=r,missing=?
12. stalk-surface-above-ring: fibrous=f,scaly=y,silky=k,smooth=s
13. stalk-surface-below-ring: fibrous=f,scaly=y,silky=k,smooth=s
14. stalk-color-above-ring: brown=n,buff=b,cinnamon=c,gray=g,orange=o, pink=p,red=e,white=w,yellow=y
15. stalk-color-below-ring: brown=n,buff=b,cinnamon=c,gray=g,orange=o, pink=p,red=e,white=w,yellow=y
16. veil-type: partial=p,universal=u
17. veil-color: brown=n,orange=o,white=w,yellow=y
18. ring-number: none=n,one=o,two=t
19. ring-type: cobwebby=c,evanescent=e,flaring=f,large=l, none=n,pendant=p,sheathing=s,zone=z
20. spore-print-color: black=k,brown=n,buff=b,chocolate=h,green=r, orange=o,purple=u,white=w,yellow=y
21. population: abundant=a,clustered=c,numerous=n, scattered=s,several=v,solitary=y

22. habitat: grasses=g,leaves=l,meadows=m,paths=p, urban=u,waste=w,woods=d

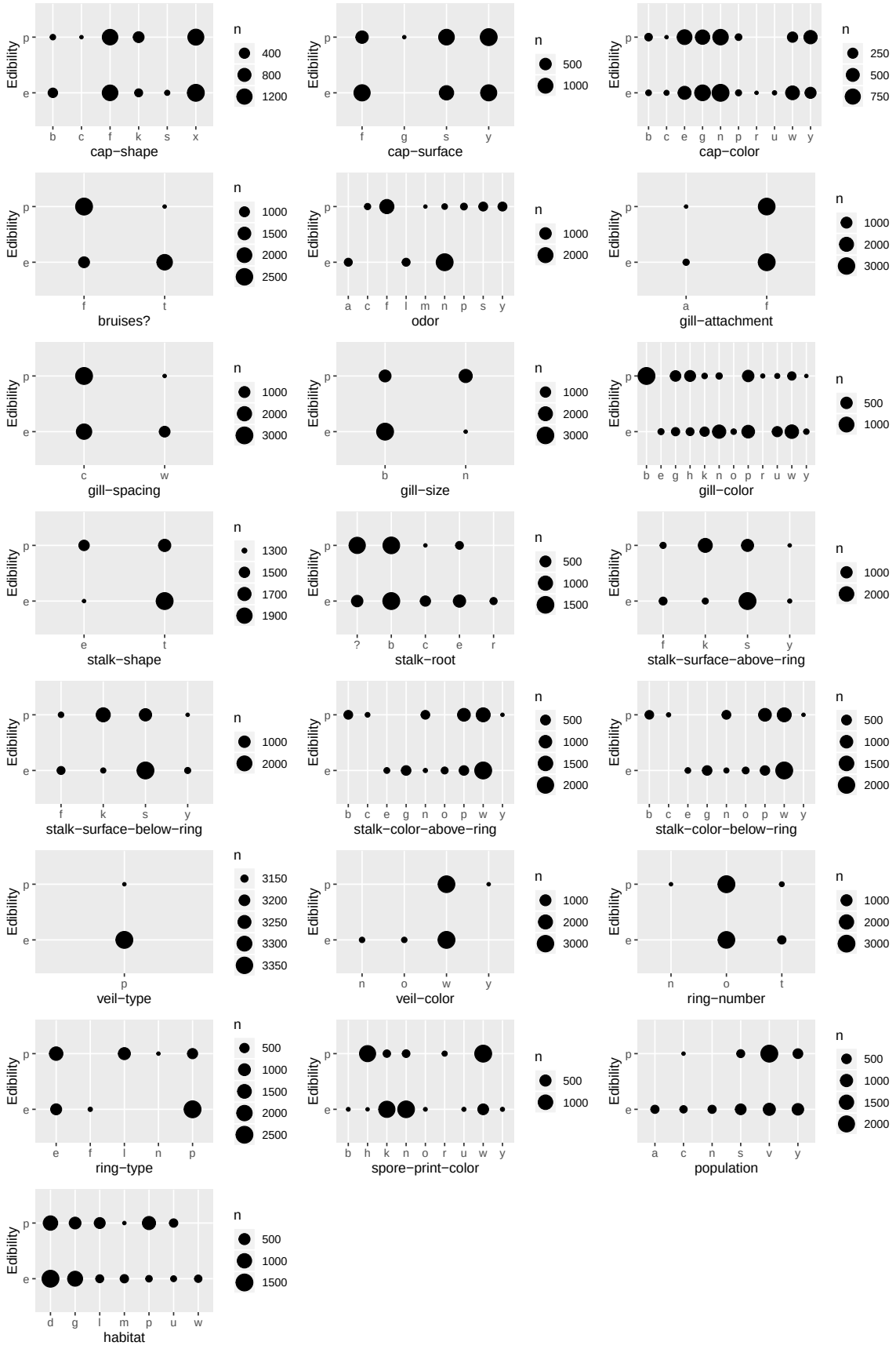
For testing the performance the prediction model we have about 1,600 mushrooms data (20% of all data) The dataset contains only the short 1 character code of the attribute values, but it's not disturbing for the model development.

Feature analysis

As first step we can investigate the distribution of the features corresponding to the edibility classification.

The following code plots this distribution for each feature.

```
#plot function for the value distribution for all feature values
#to edibility classification
#All features must be factor and the first feature must be the edibility classification
#
#parameters:
#     dataset:  the data set containing the data for the plot
#

feature_ditribution_plot <- function(data){

  #initialize the plot list
  plots <- list()

  #loop for all features except edibility
  for (i in 1:(ncol(data)-1))
  {
    #calculate the count of the feature values per edibility classification
    summarized_data <- data %>% group_by(classes, .[,i+1]) %>% summarise(n = n())
    #rename the feature column(it's easier to handle in the further code)
    names(summarized_data)[2] <- "attr"
    #create the plot for the current feature
    plot <- summarized_data %>% ggplot(aes(attr , classes)) + geom_point(aes(size=n)) +
      xlab(names(data)[i+1]) + ylab("Edibility")
    #add the plot to the plot list
    plots[[i]] <- plot
  }
  #remove the unnecessary variables
  rm(summarized_data, i, plot)

  #draw all the created plots in a grid with 3 columns
  grid.arrange(grobs=plots,ncol=3)
}

#plot all the feature distribution for the train dataset
feature_ditribution_plot(train_data)
```


Edibility
p
e
b c f k s x
cap−shape
f g s y
cap−surface
b c e g n p r u w y
cap−color
f t
bruises?
a c f l m n p s y
odor
a f
gill−attachment
c w
gill−spacing
b n
gill−size
b e g h k n o p r u w y
gill−color
e t
stalk−shape
? b c e r
stalk−root
f k s y
stalk−surface−above−ring
f k s y
stalk−surface−below−ring
b c e g n o p w y
stalk−color−above−ring
b c e g n o p w y
stalk−color−below−ring
p
veil−type
n o w y
veil−color
n o t
ring−number
e f l n p
ring−type
b h k n o r u w y
spore−print−color
a c n s v y
population
d g l m p u w
habitat
n

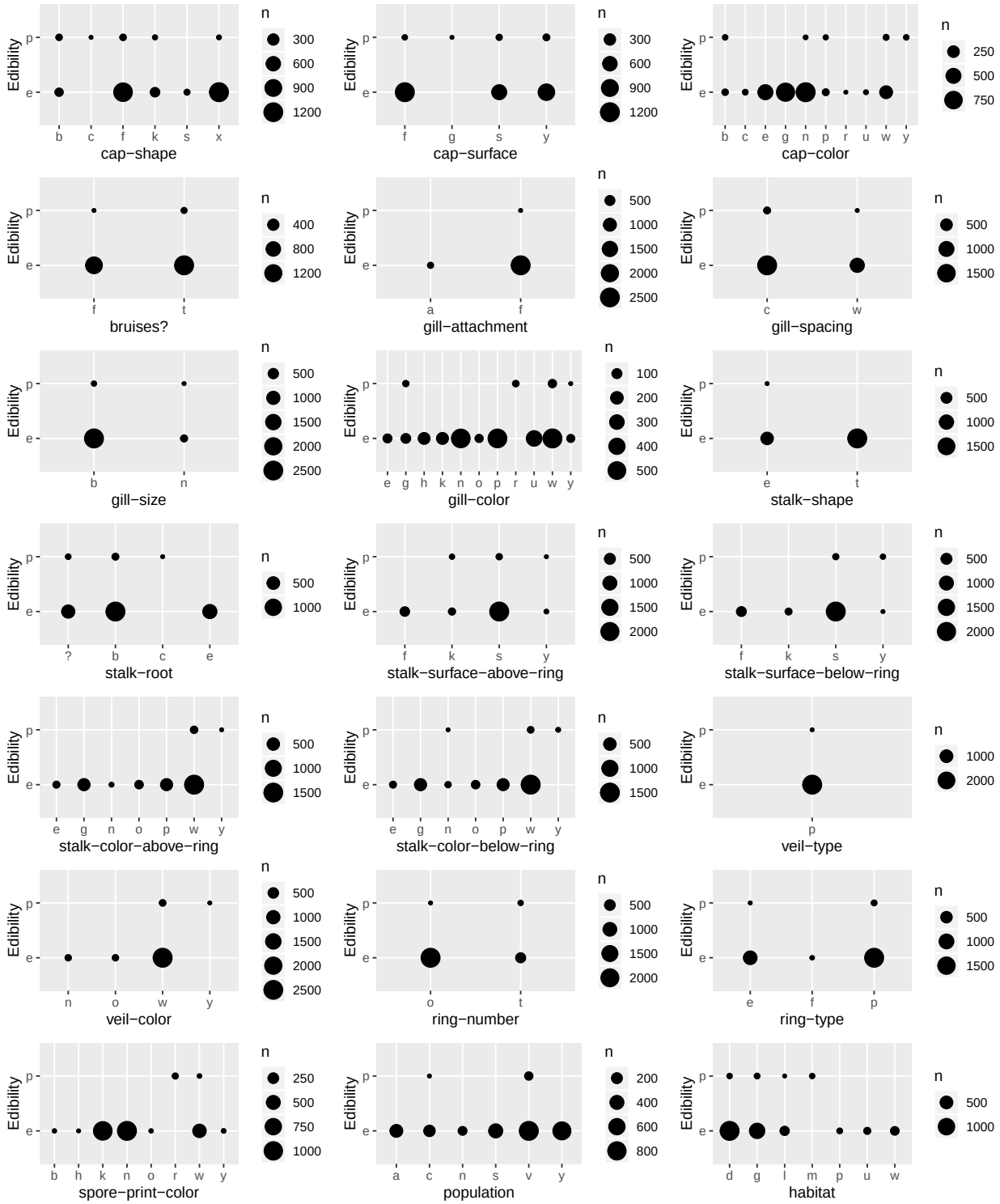
We can see that the feature veil-type has only the value 'p' (partial), there is no data record with the veil-type value 'u' (universal). This feature cannot provide decision information about the mushrooms, therefor we can remove this feature from the dataset.

On the other side, there are some features with values that can decide the classification. E.g. if we see a mushroom with the value 'f' (foul) for the feature odor, we can classify the mushroom as poisonous. The feature values 'a','c', 'f', 'l', 'm', 'p', 's', 'y' for the attribute odor can give us either 'e' or 'p' edibility classification. With this information we can classify 3,689 mushrooms. We cannot classify only the mushrooms with the odor value 'n' (none). There are 2,809 mushrooms remaining in the dataset without classification.

We can try to analyze all the mushrooms, where the odor value is 'n'. For this analysis we can remove the attribute odor from the dataset, because we examine further only a specific odor value.

The following code removes the odor attribute and plots the feature distribution for the remaining data:

```
#get the data only for odor value 'n'
odor_n <- train_data %>% filter(`odor` == 'n') %>% select(-`odor`)

#plot all the feature distribution for the remaining data
feature_ditribution_plot(odor_n)
```



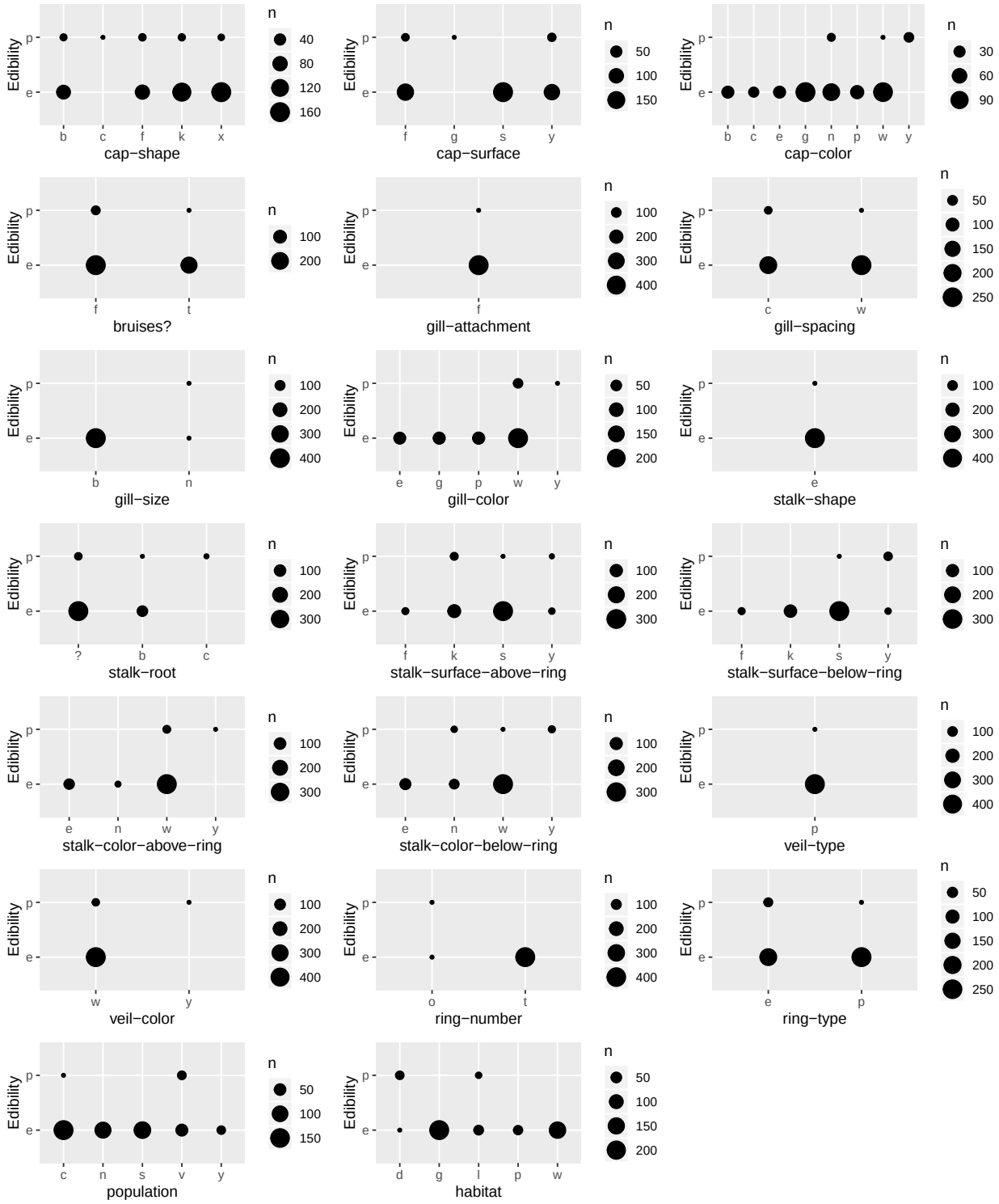
In the reduced data set we see again some features that we can use for classification the mushrooms.

E.g. if we see a mushroom with the value ‘k’ (black) for the feature spore-print-color, we can classify the mushroom as edible. The feature values ‘b’, ‘h’, ‘k’, ‘n’, ‘o’, ‘r’, ‘y’ for the feature spore-print-color can give us a classification for the remaining data set. We cannot make a classification only it the spore-print-color value is ‘w’ (white).

We can reuse the same logic like for the feature odor and filter the remaing data with the spore-print-color feature value ‘w’.

The following code removes the odor attribute and plots the feature distribution for the remain data:

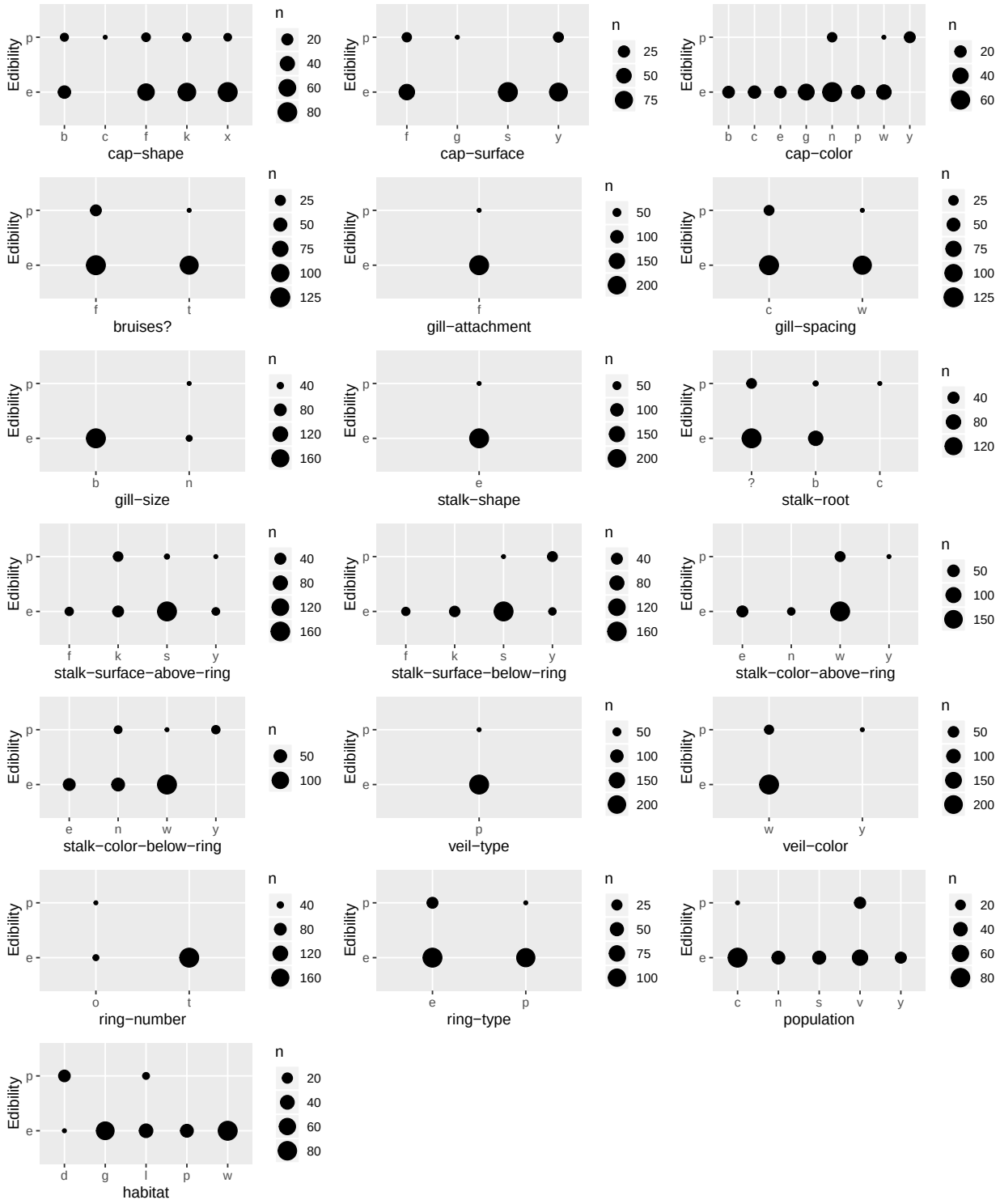
```
#get the data for the 'spore-print-color' value 'w'
spore_print_color_w <- odor_n %>% filter(`spore-print-color` == 'w') %>%
                       select(-`spore-print-color`)

#plot all the feature distribution for the remaining data
feature_ditribution_plot(spore_print_color_w)
```



We can repeat this analysis and look for features that we can use for classifying the remaining data set. As next step we can get the gill-color feature and examine only the mushrooms with the grill color 'w' (white).

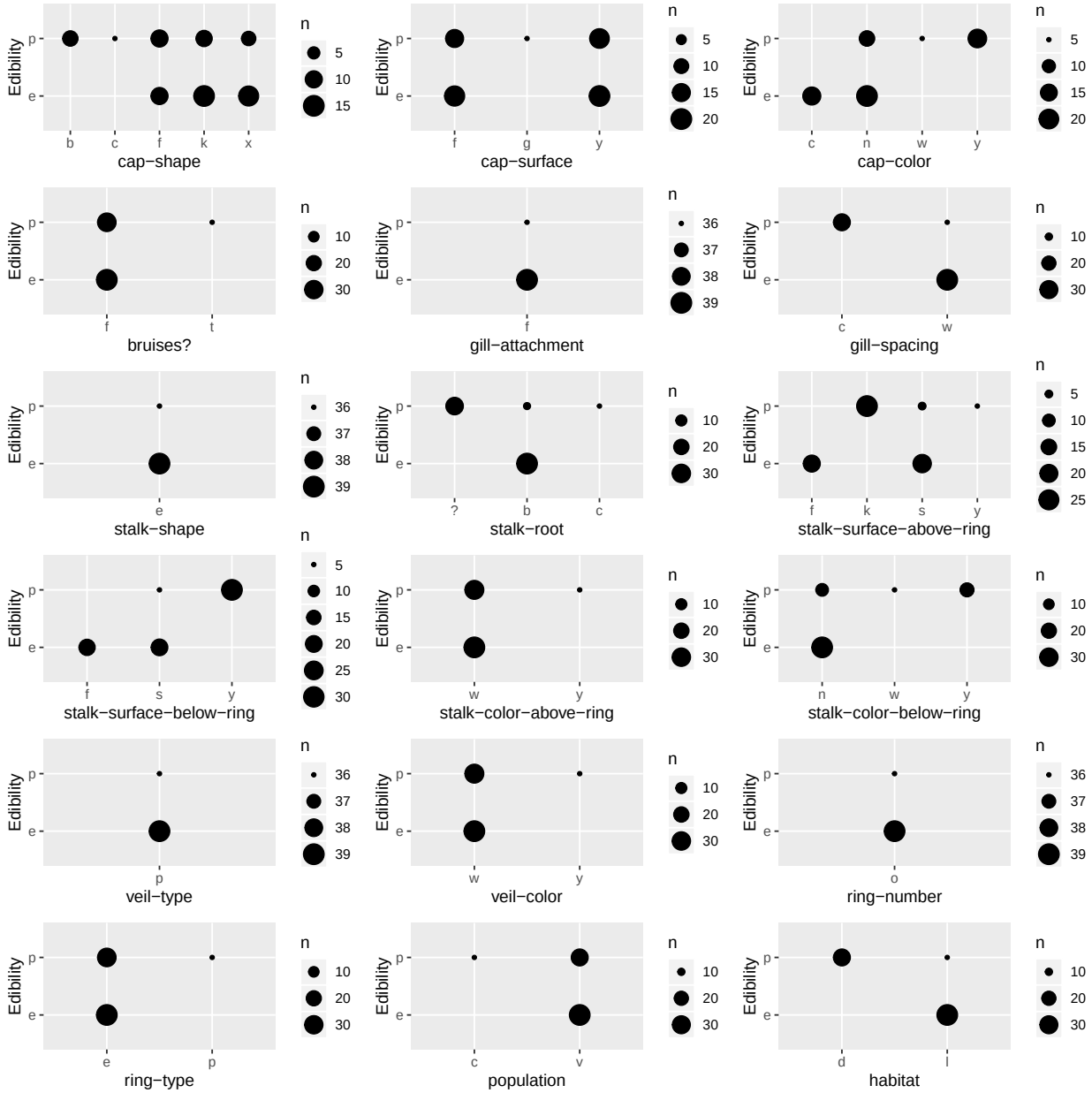
```
#get the data for the 'gill-color' feature value'w'
gill_color_w <- spore_print_color_w %>% filter(`gill-color` == 'w') %>%
                select(-`gill-color`)
```

```
#plot all the feature distribution for the remaining data
feature_ditribution_plot(gill_color_w)
```



As next step we can get the gill-size feature and examine only the mushrooms with the grill size ‘n’ (narrow).

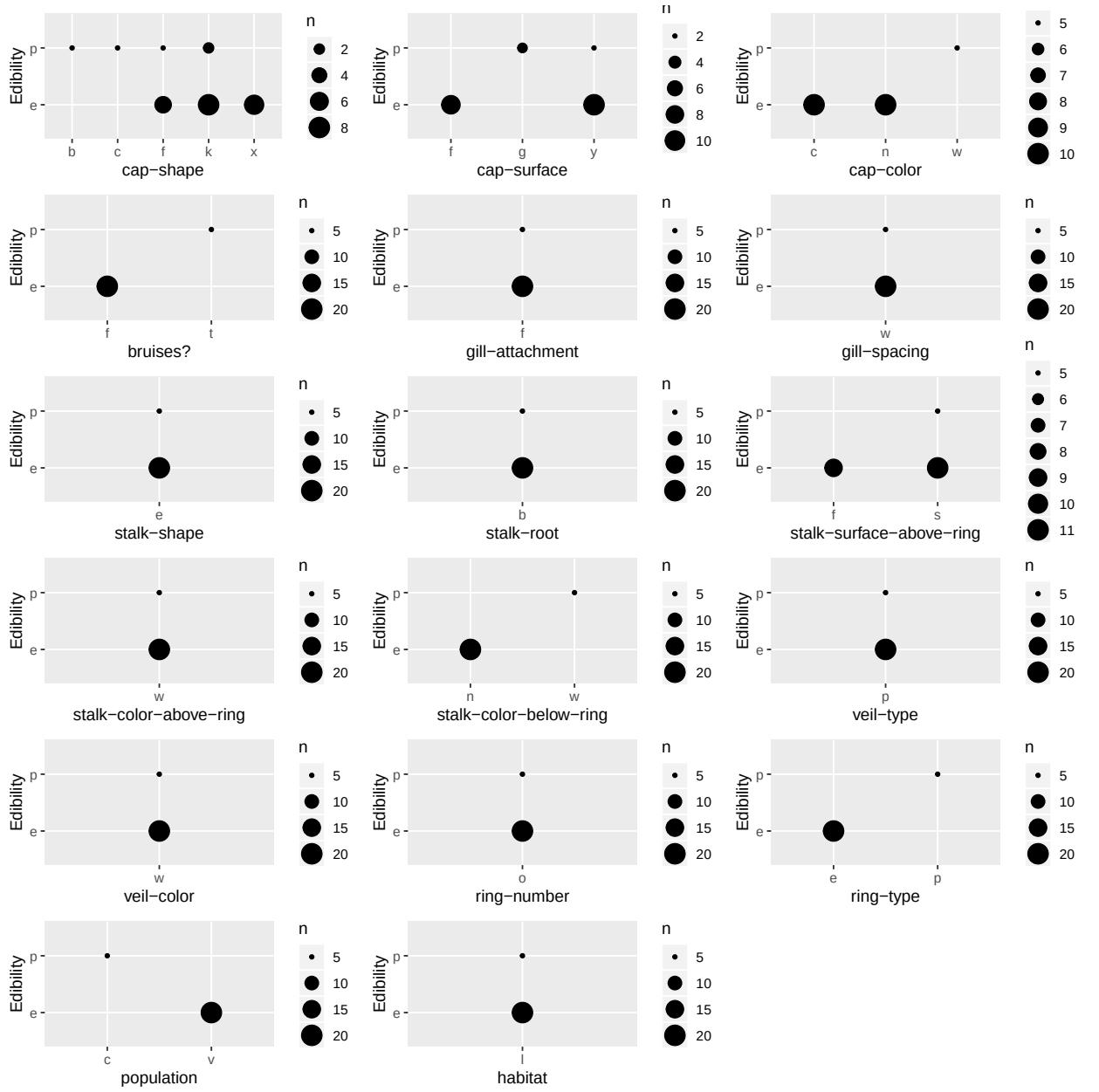
```
#get the data to the 'gill-size' feature value 'n'
gill_size_n <- gill_color_w %>% filter(`gill-size` == 'n') %>% select(-`gill-size`)

#plot all the feature distribution for the remaining data
feature_ditribution_plot(gill_size_n)
```



As next step we can get the stalk-surface-below-ring feature and examine only the mushrooms with the stalk-surface-below-ring 's' (smooth).

```
#get the data for the 'stalk-surface-below-ring' feature value 's'
stalk_surface_below <- gill_size_n %>% filter(`stalk-surface-below-ring` == 's') %>%
                    select(-`stalk-surface-below-ring`)

#plot all the feature distribution for the remaining data
```

```
feature_ditribution_plot(stalk_surface_below)
```



The remaining dataset has only 25 mushrooms. As we can see on the plot, now we can use some feature for classifying these remaining mushrooms. E.g. the feature ring-type give us a good classification: if we have a ring type of 'e' (evanescent) than the mushroom is edible. On the other case, if we have a ring-type of 'p' (pedant), the mushroom is poisonous.

With the steps about we defined a decision tree for classifying all the mushrooms in the train set. We can use this information to develop a classification model for the mushrooms.

The analyze above give us another important information: some features are more relevant for the classification some features gives no additional information for the method. That means, the different features have different information weight for the classification.

We must try to calculate a correlation between the features and the edibility. For the unordered categorical

features, we can use the uncertainty coefficient as correlation score. (See: https://en.wikipedia.org/wiki/Uncertainty_coefficient)

The uncertainty score builds on the conditional entropy form the information theory.

The entropy for a single feature is defined as follows:

$$H(X) = -\sum_{x} P_X(x) log P_X(x)$$

The conditional entropy of X given Y is defined as follows:

$$H(X|Y) = -\sum_{x,y} P_{X,Y}(x,y) log P_{X|Y}(x|y)$$

The uncertainty coefficient of the two features can be calculated as follows:

$$U(X|Y) = \frac{H(X) - H(X|Y)}{H(X)}$$

We can implement an R function to calculate the entropy, conditional entropy and the uncertainty coefficient.

The following R code implements the calculation functions:

```
#function for calculating the entropy of one feature in the dataset
#the feature must be a factor!
#
#parameters:
#     dataset:  the data set containing data for calculation
#     colX:     the index of the feature column for calculating the entropy
#               (indexing starts with 1)
#
#return:
#     the entropy of the feature in the dataset as numeric value

entropy <- function(dataset, colX){

  #the colX index must be between 1 and the count of the columns
  #if not, than the entropy is not calculable and we return 0 as entropy
  if (ncol(dataset) < colX | colX < 1){
    return (0)
  }

  #initialize the return variable
  ret <- 0
  #calculate the count of the records for each value of the given feature
  summarized_data <- dataset %>% group_by(.[,colX]) %>% summarise(n = n())
  #the row count of the whole dataset
  rowCount <- nrow(dataset)

  #get all possible feture values
  level_items <- levels(dataset[,colX])

  #calculate for all feature values..
  for (item in level_items){
    #probability of the actual feature value
    prob <- summarized_data %>% filter(.[,1] == item) %>% pull(n) / rowCount
```

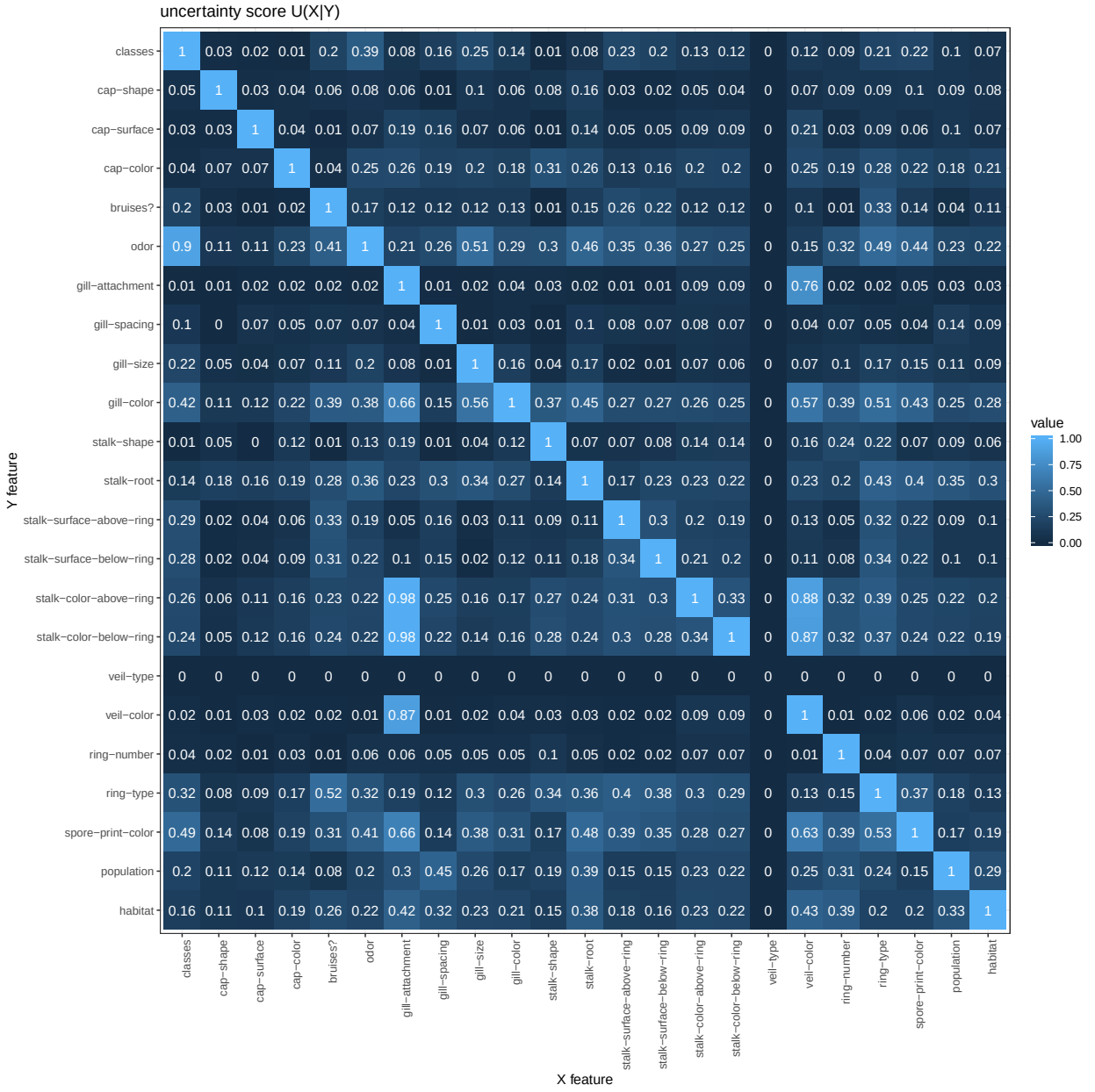
```
    #add probability multiplied by the 2 base log of the probability to the overall summ
    ret <- ret + prob * log(prob,base = 2)
  }
  #return the entropy value
  return (-ret)
}


#function for calculating the conditional entropy between two features in the dataset
#Both features must be a factor! The feature indexes start with 1.
#
#parameters:
#     dataset:  the data set containing the data for calculation
#     colX:     the index of the feature column for calculating
#               the conditinal entropy for
#     colY:     the index of the condition feature column for calculating
#               the conditional entropy
#
#return:
#     the conditional entropy of the two features in the dataset as numeric value

cond_entropy <- function(dataset, colX, colY){

  #the colX index must be between 1 and the count of the columns
  #if not, than the condiitonal entropy is not calculable and we return 0 as entropy
  if (ncol(dataset) < colX | colX < 1){
    return (0)
  }

  #the colY index must be between 1 and the count of the columns
  #if not, than the conditional entropy is not calculable and we return 0 as entropy
  if (ncol(dataset) < colY | colY < 1){
    return (0)
  }


  #initialize the return variable
  ret <- 0
  #the count of the data rows in the dataset
  rowCount <- nrow(dataset)

  #calculate the count of the records for each value of the condition feature
  summarized_y <- dataset %>% group_by(.[,colY]) %>% summarise(n = n())
  #rename the feature col name to Y (it's easier to handle in the further code)
  names(summarized_y)[1] <- "Y"

  #get all values for the condition feature
  level_itemsy <- levels(summarized_y$Y)

  #calculation loop for all possible condition values
  for (itemy in level_itemsy){

    #calculate the probability of the condition
```

```
    proby <- summarized_y %>% filter(Y == itemy) %>% pull(n) / rowCount

    #calculate the count of the compare feature values
    #using the condition value as filter
    summarized_x <- dataset %>% filter(.[,colY] == itemy) %>%
        group_by(.[,colX]) %>% summarise(n = n())

    #rename the compare feature col name to X (it's easier to handle in the further code)
    names(summarized_x)[1] <- "X"

    #get all values for the compare feature
    level_itemsx <- levels(summarized_x$X)

    #calculation loop for all possible compare values
    for (itemx in level_itemsx){
      #calculate the count of the condition feature value
      filtered <- summarized_x %>% filter(X == itemx)
      #if there are values and the condition probability is not 0
      if (nrow(filtered) > 0 && proby != 0){
        #calculate the joined probabiliy P(X,Y)
        probxy <-  filtered$n / rowCount
        #calculate the conditional probabilit p(X|Y)
        probx_at_y <- probxy / proby
        #add conditinal probability multiplied by the 2 base log
        #of the conditional probability to the overall summ
        ret <- ret + probxy * log(probx_at_y,base = 2)
      }
    }
  }
  #return the conditional entropy
  return (-ret)
}


#function for calculating the uncertainty score between two features in the dataset
#Both features must be a factor! The feature indexes start with 1.
#
#parameters:
#     dataset:  the data set containing the data for calculation
#     colX:     the index of the feature column for calculating
#               the conditinal entropy for
#     colY:     the index of the condiion feature column for calculating
#               the conditional entropy
#
#return:
#     the uncertainty score of the two features in the dataset as numeric value


uncertainty <- function(dataset, colX, colY){
  #calculate the entropy for the compare feature
  entr <- entropy(dataset,colX)

  #if the enropy is 0, we return 0 as uncertainty
```

```
  if (entr == 0){
    return(0)
  }

  #calculate the conditional entropy for the two feature
  cond_entr <- cond_entropy(dataset, colX, colY)

  #return the uncertainty score for the features
  return ( (entr - cond_entr) / entr)
}
```

We can plot the uncertainty coefficient between all the features as a matrix. The following R code creates the plot:

```
uncertainty_plot <- function(dataset){
  #calculate uncertainty for all feature pairs
  uncertainty_df <- data_frame(X=character(), idx=numeric(), Y=character(),
                               idy=numeric(), value=numeric())
  labels <- names(dataset)
  for (i in 1:ncol(dataset))
  {
    for(j in 1:ncol(dataset))
    {
      uncertainty_df <- bind_rows(uncertainty_df, data_frame(Y = labels[j], idy=j ,
                                  X=labels[i], idx=i, value=uncertainty(train_data,i,j)))
    }
  }

  #plot the uncertainty with colors
  uncertainty_df %>% ggplot(aes(x=reorder(X,idx), y=reorder(Y,-idy), fill=value)) +
    geom_tile() + geom_text(aes(label=round(value,2)), color='white') +
    theme_bw() + theme(axis.text.x = element_text(angle = 90, hjust = 1)) +
    xlab('X feature') + ylab('Y feature') + ggtitle('uncertainty score U(X|Y)')
}

uncertainty_plot(train_data)
```



This plot shows the uncertainty coefficient between all feature pairs. The light blue values show a high uncertainty coefficient (high correlation) the dark blue values show a low uncertainty coefficient (low correlation) The value 1 in the matrix shows a perfect correlation, therefore the matrix diagonal elements are 1. (The diagonal elements show the correlation of the features with himself, therefore there is a perfect correlation)

We see that the feature veil-type has 0 correlation with all other features. In the first data analysis we saw that the feature veil-type has only one value and has no prediction information, that causes the 0 correlation.

For the classification model we examine the first column of the matrix. This column shows the correlation between edibility and another feature. As we see, the feature odor has the highest correlation with the edibility. The other features have only a correlation belowe 0.5.

We can use this information for developing a classification model that use only the features with high correlation with the edibility.

The classification models

The classification model will be evaluated with the F_1 score. This score includes not only the overall accuracy, it considers the prediction specificity and sensitivity too.

The F_1 score is the harmonic average of the precision and recall:

$$F_1 = \frac{1}{\frac{1}{2}(\frac{1}{recall} + \frac{1}{precision})}$$

where recall is the true positive rate (sensitivity) and precision is the positive prediction value (specificity).

The model should predict the edibility of the mushrooms, therefore it's more important to make the right prediction for the poisonous mushrooms as the edible. Therefore, if two models have the same F_1 score, the model with the higher true negativ rate (specificity) will be declared as better model.

After the data analysis we can develop two different model:

- model based on simple decision rules
- model based on the feature value combinations

The following two chapter describe the two models.

Decision tree model

As we saw before, we can develop a prediction model based on the feature distribution between the edible and poisonous classes. A very easy way to define a decision tree is to examine the generated charts like we did it before.

On the other side, this method is fixed on the current train data set. If the dataset changes, we must to make the analysis again manually.

Therefore, we need to develop a method to create a decision tree automatically. For this method we can use the same logic as we did:

1. create the edibility distribution for all features for the dataset
2. search for features, that have values with only either 'e' or 'p' edibility. If we don't find a feature, then we can't improve our decision tree anymore
3. if we found some feature in step 2, we select one of them and add to the decision tree
4. filter out all records from the dataset that match the feature values with distinct edibility values
5. if there is no remaining record, then we reached a full decision tree for the train set
6. if there are remaining records, we start at step 1 again, but we use only the remaining records for the next loop

With this method we can find a decision tree for the train set, that we can use for predicting the edibility for the test dataset.

For the prediction, we can go through the features in the prediction tree and check if the feature in the actual record matches either the feature values with distinct 'e' or 'p' edibility. If we find a match, we use the corresponding edibility value as prediction. If there is no match for all the feature in the prediction tree, we predict 'p' edibility. This is, because, we want to be sure not to predict edible for a possible poisonous mushroom.

Feature based model

The uncertainty score matrix give us an overview about the correlation between the edibility classification and other features. With this information we can develop a prediction model. We can assume, that the most relevant features give use enough information to predict the edibility and we can ignore further features.

For the prediction we take the n most relevant features and select the edibility to all the feature value combination to the features and examine which edibility values occures to this combination. For example, we

can use the features 'odor' and 'spore-print-color' and generate a summary for all existing feature combination to this two features und the edibility classification.

The following R code lists all features value combination for the mentioned features:

```
odor_sporeprintcolor_values <- train_data %>%
          group_by(classes, odor, `spore-print-color`) %>% summarise()
odor_sporeprintcolor_values %>% print(n = nrow(odor_sporeprintcolor_values))
```

```
## # A tibble: 24 x 3
## # Groups:   classes, odor [10]
##    classes odor  `spore-print-color`
##    <fct>   <fct> <fct>
##  1 e       a     k
##  2 e       a     n
##  3 e       a     u
##  4 e       l     k
##  5 e       l     n
##  6 e       l     u
##  7 e       n     b
##  8 e       n     h
##  9 e       n     k
## 10 e       n     n
## 11 e       n     o
## 12 e       n     w
## 13 e       n     y
## 14 p       c     k
## 15 p       c     n
## 16 p       f     h
## 17 p       f     w
## 18 p       m     w
## 19 p       n     r
## 20 p       n     w
## 21 p       p     k
## 22 p       p     n
## 23 p       s     w
## 24 p       y     w
```

There are some odor/spore-print-color value combination that have both 'p' and 'e' edibility classification. for this combination we don't have a definitely prediction. If we want to have a high specificity (we want to predict the poisonous mushrooms as good as possible), we must predict 'p' in this case.

There are some odor/spore-print-color value combination with exclusive 'p' edibility classification. In this case we can predict 'p'.

There are some odor/spore-print-color value combination with exclusive 'e' edibility classification. In this case (and only in this case) we can predict that the mushroom is edible.

Our prediction model will search all feature value combination with exclusive 'e' edibility form the train set and predict for any record from the test set that match to one of this value combination 'e' edibility. For all other records in the test set we predict 'p' edibility.

The number of the used feature influences the F_1 score of our prediction. With more used feature we can make better prediction. After a specific number of used features, we will get worse F_1 score again. That is the effect of the overtraining. Therefore, we must find out the number of the features to use for the best prediction performance.

Implementation & result

In this chapter I will implement the defined two prediction model and analyze the result. For the feature-based model i will use n-fold cross validation. For this purpose, I implemented a function to calculate the n-fold cross validation for given train und test dataset calling a given function.

The following R code implements the n-fold cross validation (The function has to be called with cv_n >= 2):

```
#function for cross validation
#parameters:
#     trainset: the train set to use for the cross validation
#     cv_n:     the count of the cross validation
#     FUNC:     the function to call for the actual cross validation train and
#               test set (calculated from the param trainset)
#     ...:      additional parameter necessary for calling the provided function
#
#return:
#     dataframe with the function result for the cross validations
#     (the data frame has cv_n items)

cross_validation <- function(trainset, cv_n, FUNC,...){

  #get the count of the data rows on the train set
  data_count = nrow(trainset)

  #initialize the data frame for the result
  values_from_cv = data_frame()

  #randomise the trainset.
  #If the train set is ordered the cross validation
  #will not be independent and provide wrong result
  trainset_randomised <- trainset[sample(nrow(trainset)),]

  #create the train- and testset for the cross validation
  #we need cv_n run, therefore we use a loop
  for (i in c(1:cv_n)){
    #evaulate the size of the test set. This will be the 1/cv_n part of the data
    part_count = data_count / cv_n

    #select the data from the parameter train set
    #we get the part_count size elements from the parameter train set
    idx = c(   (trunc((i-1) * part_count) + 1) : trunc(i * part_count) )

    #tmp holds the new test set
    test = trainset_randomised[idx,]
    #train holds the new test set
    train = trainset_randomised[-idx,]

    #call the provided function to the actual train and test set.
    akt_value <- FUNC(train, test,...)

    #add the result to the data frame
    #the column 'cv' contains the idx of the cross validation run
    values_from_cv <- bind_rows(values_from_cv, akt_value %>% mutate(cv = i))
  }
```

```
  #return the results of each cross validation
  return(values_from_cv)
}
```

For calculation of the F_1 score, i will us the confuseMatrix R function.

Decision tree model

As first iteration I implemented a naive decision tree model based on the analysis in the previous chapters. This implementation is not trainable, it depends on the observation about the train dataset. The R implementation this naive decision tree is as follow:

```
#implementation of the naiv decision tree model based on the data analysis observation
#This implementation uses fix decision tree, therefore is not suitable
#for machine learning
#
#parameters:
#     dataset:  the data set containing the data for prediction
#
#return:
#     the predicted classification list

predict_dectree_naive <- function(dataset){

  #initalization for the result variable
  predicted <- tibble(y = character())

  #loop all the data in the dataset
  for (i in 1 : nrow(dataset))
  {
    y <- tryCatch({
        #get the feature values relevant for the fixed decision tree

        odor <- dataset[i,]$odor
        spore_print_color <- dataset[i,]$`spore-print-color`
        gill_color <- dataset[i,]$`gill-color`
        gill_size <- dataset[i,]$`gill-size`
        stalk_surface <- dataset[i,]$`stalk-surface-below-ring`
        ring_type <- dataset[i,]$`ring-type`

        #go throug the decision tree rules in an if - then tree
        if (odor %in% c('c','f','m','p','s','y')){
          'p'
        }
        else if (odor  %in% c('a','l')){
          'e'
        }
        else if (spore_print_color %in% c('e','g', 'n','o','p','y')){
          'e'
        }
        else if (spore_print_color %in% c('r')){
          'p'
        }
        else if (gill_color %in% c('y')){
```

```
          'p'
        }
        else if (gill_color %in% c('e','g','p')){
          'e'
        }
        else if (gill_size %in% c('b')){
          'e'
        }
        else if (stalk_surface %in% c('f')){
          'e'
        }
        else if (stalk_surface %in% c('y')){
          'p'
        }
        else if (ring_type %in% c('e')){
          'e'
        }
        #if not match in the tree, we predict poisonous
        else{
          'p'
        }
    }, warning = function(w){
        return('p')
    }, error = function(e) {
        return('p')
    })
    #add the current prediction to the result list
    predicted <- bind_rows(predicted, data_frame(y = y))
  }

  #convert the result to a factor with two values
  predicted <- predicted %>% mutate(y=factor(y, levels=c('e','p'))) %>% pull(y)
  #reutrn the prediction list
  return(predicted)
}
```

With this naive decision tree function, we can predict the edibility for the test set and calculate the result F_1 score:

```
predicted_naive <- predict_dectree_naive(test_data %>% select(-classes))
result_naive_tree_model <- confusionMatrix( predicted_naive, test_data$classes)
result_naive_tree_model
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction   e   p
##          e 831   0
##          p  11 784
##
##                Accuracy : 0.9932
##                  95% CI : (0.9879, 0.9966)
##     No Information Rate : 0.5178
##     P-Value [Acc > NIR] : < 2.2e-16
##
```

```
##                  Kappa : 0.9865
##
##  Mcnemar's Test P-Value : 0.002569
##
##             Sensitivity : 0.9869
##             Specificity : 1.0000
##          Pos Pred Value : 1.0000
##          Neg Pred Value : 0.9862
##              Prevalence : 0.5178
##          Detection Rate : 0.5111
##    Detection Prevalence : 0.5111
##       Balanced Accuracy : 0.9935
##
##        'Positive' Class : e
##
```

The F_1 score is 0.9934 with a specificity of 1. This is a good result, but we have 11 edible mushrooms that we predict as poisonous.

The next step is to develop a method, that builds the decision tree based on the train set automatically. For the development we use the logic described in the chapter before. We can try to implement a function, that try to build all the possible decision trees and select the best decision tree based on the F_1 score with the n-fold cross validation.

Unfortunately, such a recursive function would find to many decision trees. In the first level we can find 14 features that could be used for a decision rule. If we try to find all possible decision rule combination for these 14 rules, we would get about 120 decision trees with two decision rules. In the 3rd level the count of the decision trees explodes about 2.000. If we want to find all the possible decision trees and select the best of them, we would need a strong computer.

Therefore, the implementation takes on each level only the decision rule, that give us classification to the most data from the remaining data set. With this limitation we have only one decision tree at the end of the function calls.

The following R code implements the building of the decision tree based on a given dataset:

```
#Function to create a decision tree model based on the given dataset
#
#parameters:
#     dataset:  the data set containing the data for train the decision tree
#
#return:
#     the trained decisoin tree. The tree is a tibble with decision rules as row.
#     the columns contains the feature name for the rule, the values
#     to predict 'e' edibility
#     and the valus to predict 'p' edibility

train_decision_tree_model <- function(dataset){

  #initialize the return tibble as empty
  tree <- data_frame(feature=character(), decided_proz=numeric(),
                     e_values=character(), p_values=character())
  #call the recursive function to get the decision rules for the decision tree
  #we start the recursion with the whole dataset and an empty tree
  tree <- extend_decision_tree(dataset, tree)
  #return the decision tree
  return(tree)
```

```
}


#Recursive function to create the decision rules for the decision tree
#based on the given (remaining) dataset- The function assumes, that the edibility
#classification is the first column in the dataset! All the features must be a factor
#
#parameters:
#     dataset:        the data set containing the data for train the decision tree
#     decision_tree:  the decision tree, we extend recursive with new rules based
#                     on the given dataset
#
#return:
#     the trained decision tree. The tree is a tibble with decision rules as row.
#     the columns contains the feature name for the rule, the values
#     to predict 'e' edibility and the valus to predict 'p' edibility

extend_decision_tree <- function(dataset, decision_tree){

  #initialize the frame with the possible new rules
  newRules <- data_frame(feature=character(),decided_proz=numeric(),
                         e_values=character(), p_values=character())
  #the count of all data in the dataset
  count = nrow(dataset)

  #we go throug all the features except the edibility classification (col = 1)
  for (i in 1:(ncol(dataset)-1))
  {
    #Get the current feature name
    feature <- names(dataset)[i+1]
    #calculate the count of the feature values
    summarized_data <- dataset %>% group_by(classes, .[,i+1]) %>% summarise(n = n())
    #rename the feature col name to attr (it's easier to handle in the further code)
    names(summarized_data)[2] <- "attr"
    #get the possible feature values
    values <- levels(summarized_data$attr)
    #initialize the edible values list
    e_values <- list()
    #intialize the poisonous value list
    p_values <- list()
    #initialize the counter for the feature pereformance score
    decided_count = 0
    #initialize the index for the list entries
    e_pos <- 1
    p_pos <- 1

    #loop through all feature values
    for(val in values){

      #calculate the count of the edible entries for the feature value
      e_count <- summarized_data %>% filter(classes=='e' & attr == val) %>% pull(n)
      #if no definitely edible entries than set the count to 0
      e_count <- ifelse(is_empty(e_count),0,e_count)
```

```
    #calculate the count of the poisonous entries for the feature value
    p_count <- summarized_data %>% filter(classes=='p' & attr == val) %>% pull(n)
    #if no definitely edible entries than set the count to 0
    p_count <- ifelse(is_empty(p_count),0,p_count)

    #if the feature value has only p entries, than we can use it as p rule
    if (e_count == 0 && p_count > 0){
      #add the value to the p rules
      p_values[[p_pos]] <- val
      #update the feture performance score
      decided_count <- decided_count + p_count
      #update the p rule list index
      p_pos <- p_pos + 1
    }

    #if the feature value has only e entries, than we can use it as e rule
    if (e_count > 0 && p_count == 0){
      #add the value to the e rules
      e_values[[e_pos]] <- val
      #update the feature performance score
      decided_count <- decided_count + e_count
      #update the e rule list index
      e_pos <- e_pos + 1
    }
  }

  #if the feature performance score is greater as 0
  #(we can use the feature as decision rule)
  if (decided_count > 0){
    #convert the lists to comma separated string
    e_values <- paste(e_values, collapse=",")
    p_values <- paste(p_values,collapse=",")

    #add the rule as possible new decision rule to the tibble
    #we add the percent of the data, that could be classified with the actuall rule
    #we will use this score for selecting the best rule
    newRules <- bind_rows(newRules, data_frame(feature=feature,
                          decided_proz=decided_count / count, e_values = e_values,
                          p_values = p_values))

  }
}

#if we found at least one decision rule for the given dataset
if (nrow(newRules) > 0){
  #look for the decision rule with the highest percentage of classified data
  idx <- which.max(newRules$decided_proz)
  bestRule <- newRules[idx,]

  #get the e rules from the best performing decision rule
  e_values <- unlist(str_split(bestRule$e_values, ','))
  #get the p rules from the best performing decision rule
  p_values <- unlist(str_split(bestRule$p_values, ','))
```

```
    #get the data from the dataset, that couldn't classify with the e and p rules
    remaining_data <- dataset %>% filter( !(.[,bestRule$feature] %in% e_values) &
                                          !(.[,bestRule$feature] %in% p_values))
    #add the best rule to the tree
    decision_tree <-bind_rows(decision_tree, bestRule)

    #if we have remaining data
    if (nrow(remaining_data) > 0){
      #than call the function recursvie again with the remaining data
      #and the extended tree
      decision_tree <- extend_decision_tree(remaining_data, decision_tree)
    }
  }

  #return the decision tree if we are ready with the recursion
  return(decision_tree)
}
```

Additionally, we need a function to predict the edibility for the test set based on the trained decision tree. The function must check the first decision rules existing in the decision tree and check if the actual record match either the values for the edible or the poisonous classification. If there is no match, the function musst go through the following rules until a match is found or the decision tree has no more rules. If there was no match in the entire decision tree, we predict poisonous classification. (We want to be on the safe side with the poisonous mushrooms.)

The following R code implements the predict function for the trained decision tree:

```
#Function for predict the edibility based on the given decision tree
#
#parameters:
#     dataset:  the data set containing the data for prediction
#     tree:     the decision tree for to use for the prediction
#
#return:
#     the predicted classification list

predict_decision_tree <- function(dataset, tree){
  #initialize the result list
  ret <- list()
  #loop through all data in the dataset
  for(i in 1:nrow(dataset)){
    #initialize the current classification
    #we initialize to the poisonous value, because we want to be sure,
    #that we classify 'p' for unknown results
    y <- 'p'
    #the decision rule count in the tree
    count_rules <- nrow(tree)
    #loop throug all the decision rules in the tree
    for(j in 1:count_rules){
      #get the feature name from the current decision rule
      feature <- tree$feature[j]
      #get the e rules from the current decision rule
      e_values <- unlist(str_split(tree$e_values[j], ','))
      #get the p rules from the current decision rule
```

```
      p_values <- unlist(str_split(tree$p_values[j], ','))

      #if the current data match to one of the e rule values
      if (dataset[i,feature] %in% e_values){
        #than we predic edible
        y <- 'e'
        #the current row is predicted, we can add the precition to the result
        break;
      }

      #if the current data macht to one of the p rules
      if (dataset[i,feature] %in% p_values){
        #we predict poisonous
        y <- 'p'
        #the current row is predicted, we can add the precition to the result
        break;
      }
    }
    #add the predicted value to the result
    ret <- c(ret, y)
  }
  #return the prediction list as factor
  return(factor(ret, levels = c('e','p')))
}
```

Now we can train the decision tree and calculate the performance of our decision tree model on the test set. The following R code trains the decision tree on the train dataset and predict the edibility values for the test set. Afterwards calculate the F_1 score for the decision tree model:

```
predict_tree <- train_decision_tree_model(train_data)
predicted <- predict_decision_tree(test_data, predict_tree)
result_decision_tree_model <- confusionMatrix( predicted, test_data$classes)
result_decision_tree_model
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction   e   p
##          e 842   0
##          p   0 784
##
##                Accuracy : 1
##                  95% CI : (0.9977, 1)
##     No Information Rate : 0.5178
##     P-Value [Acc > NIR] : < 2.2e-16
##
##                   Kappa : 1
##
##  Mcnemar's Test P-Value : NA
##
##             Sensitivity : 1.0000
##             Specificity : 1.0000
##          Pos Pred Value : 1.0000
##          Neg Pred Value : 1.0000
##              Prevalence : 0.5178
```

```
##          Detection Rate : 0.5178
##    Detection Prevalence : 0.5178
##       Balanced Accuracy : 1.0000
##
##        'Positive' Class : e
##
```

With the trained decision tree, we have F_1 score of 1 with a specificity 1. This model made a perfect prediction for the test set.

Feature-based model

The implementation of the feature-based model uses an implementation based on the logic described before. Additionally, the implementation can handle different number of used features. This kind of implementation can be used to train the model with different features count and select the model with the best performance.

The following R code implements the train function for the feature model:

```
#The function trains the feature model with the given amount of the features.
#To select the most relevant features, the function uses the
#uncertainty score calculation
#All the features must be a factor. The function assumes, that the edibility
#classification is the first column in the dataset
#
#parameters:
#     dataset:        the data set containing the data for train the feature model
#     feat_count:     the count of the feature to use for the training
#
#return:
#     the trained feature model. Thre result is a tibble containing the
#     feature value combination for that we predict 'e' edibility

train_feature_model <- function(dataset, feat_count){

  #initialize the tibble for the uncertainty score
  uncertainty_df <- data_frame(X=character(), idx=numeric(), value=numeric())
  #get the feature name
  labels <- names(dataset)
  #loop all the features in the dataset
  for (i in 1:ncol(dataset))
  {
      #calculate the uncertainty score for the current feature with the first feature
      #(first feature contains the edibility classification)
      uncertainty_df <- bind_rows(uncertainty_df, data_frame(X=labels[i],
                                  idx=i, value=uncertainty(dataset,1,i)))
  }
  #order the data by the uncertainty score descending
  uncertainty_df <- uncertainty_df %>% arrange(-value)
  #get the edible classification + the following feat_count features
  features <- head(uncertainty_df$X,feat_count+1)

  #select the unique data for all the features combination,
  #where the classification is 'p'
  p_values <- dataset[,features] %>% filter(classes=='p') %>%
                  select(-classes) %>% unique()
```

```
  #select the unique data for all the features combination,
  #where the classification is 'e'
  e_values <- dataset[,features] %>% filter(classes=='e') %>%
                  select(-classes) %>% unique()

  #filter out all the data from the e_values, that occures also in the p values
  e_values <- e_values %>% anti_join(p_values, by=names(e_values))
  #so we have the feature combinations with definitly 'e' classification
  #add this classification as column to the result
  e_values <- e_values %>% mutate(pred='e')

  #return the result
  return(e_values)
}
```

With the trained model we can predict the edibility classification for the test set. The predict function search for all data in the test set in the trained model (that contains all feature combination with definitely edible mushrooms) if there is any entry. If we find the current data combination in the trained model, we predict 'e' edibility. In all other case we predict a poisonous mushroom. (We want to predict the poisonous mushrooms with high specificity, therefor we predict 'p' if we are not sure that the mushroom is edible.)

The following R code implements the predict function:

```
#The function predict the edibility classification for the given dataset
#based on the given feature model
#All the features must be a factor.
#
#parameters:
#     dataset:        the data set containing the data for train the feature model
#     feat_count:     the feature model (the e value combinations)
#
#return:
#     the predicted classification list

predict_feature_model <- function(dataset, e_values){
  #get the features containing in the feature model
 feature_names = names(e_values %>% select(-pred))

 #join the dataset to the feature model based on the selected features
 #we use left join, therefore there can be data without founded entry in the feature model
 #if we found an entry in the feture model (e values), than we predict 'e'
 #other case we predict 'p'
 #So we predict 'e' only for combination where we are sure, that the mushroom is edible
 pred <- dataset %>% left_join(e_values, by=feature_names) %>%
            mutate(y = factor(ifelse(is.na(pred),'p', pred), c('e','p'))) %>% pull(y)

 #return the predicted values
 return(pred)
}
```

To find the optimum feature count for the feature model, we need a function that calculates the F1 score for a given feature count. We can call this function with different feature counts and look for the feature count with the best F1 score.

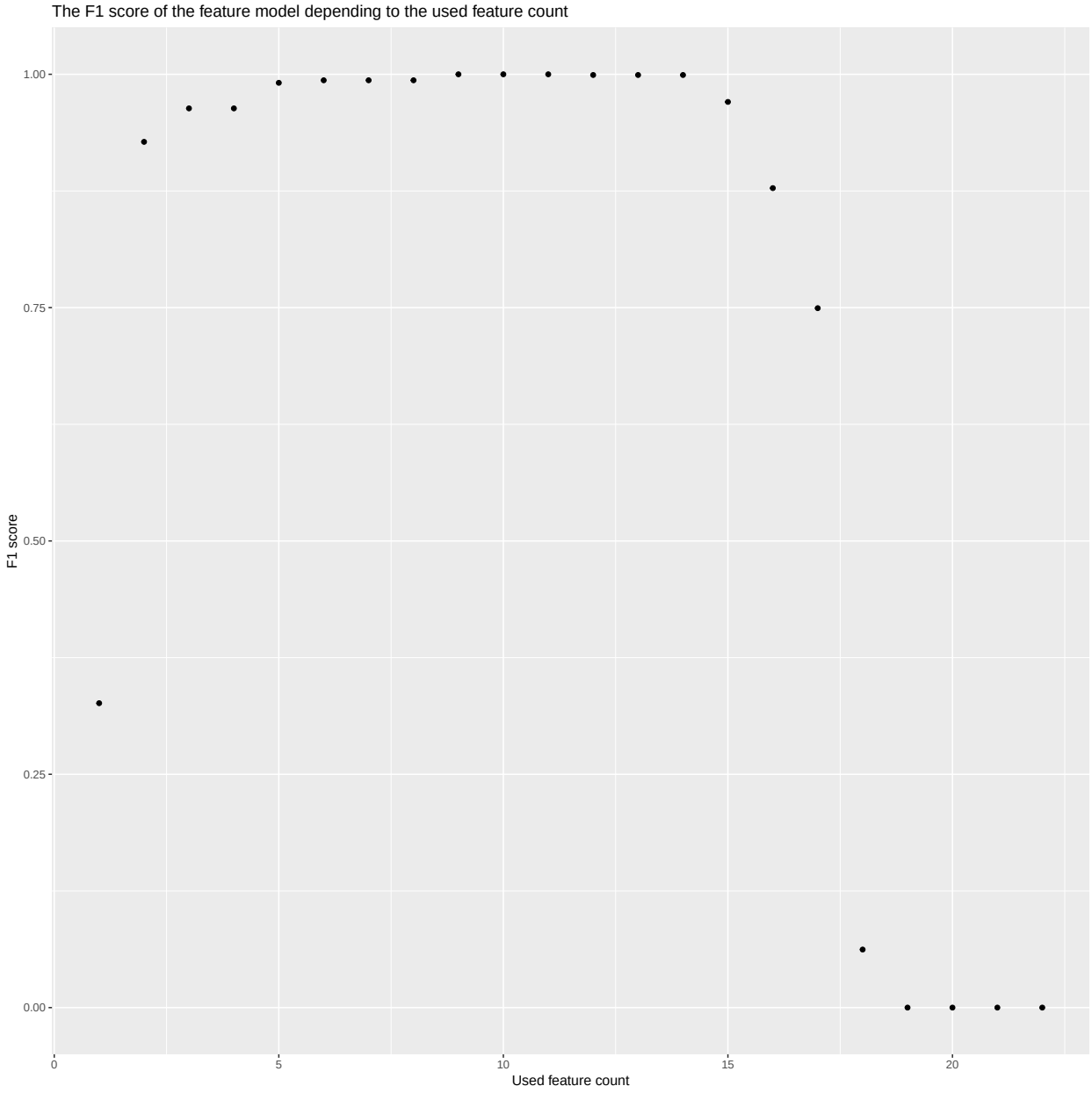
The following function implements the F1 score calculation function for the feature model for a given feature count, test and train set:

```r
#This function calculates the F1 score for the feature model for a given
#feature_count by given train and test set. This function can be called by
#the n-fold cross validation function
#
#parameters:
#     train:            the train data set containing the data for train
#                       the feature model
#     test:             the test data set to predict the edibility classification
#     features_count:   the count of the feature to use for the training
#
#return:
#     the F1 score to the trained model

calculate_F1_score <- function(train, test, features_counts){
  F1_all = data_frame(feature_count = numeric(), F1 = numeric())
  for(i in features_counts){
    train_model <- train_feature_model(train, i)
    predicted <- predict_feature_model(test, train_model)
    result <- confusionMatrix(predicted, test$classes)
    F1_score <- result$byClass[["F1"]]
    F1_score <- ifelse(is.na(F1_score),0,F1_score)
    F1_all <- bind_rows(F1_all, data_frame(feature_count = i, F1=F1_score))
  }
  return(F1_all)
}
```

The dataset has 22 features to predict the edibility classification. We can try the feature model with all the feature count between 1 and 22 and examine the F_1 scores. For the F1 score calculation we need test and train sets, but we sould only use our defined train set for training the model. Thefore we should use the n-fold cross validation function (introduced in previuous chapter) to train the model.

The following code trains the feature model with the n-fold cross validation and plots the F_1 score to all the used feature counts. (I use 5-fold cross validation):

```r
#train features model with cross validation
F1_scores <- cross_validation(train_data,5, calculate_F1_score, 1:22) %>%
  group_by(feature_count) %>% summarise(F1=mean(F1))
#plot the F1 scores depending on the used feature count
F1_scores %>% ggplot(aes(feature_count, F1)) + geom_point() +
  xlab('Used feature count') + ylab('F1 score') +
  ggtitle('The F1 score of the feature model depending to the used feature count')
```



As we see, the best F_1 score is about 10 used features. On the other side, we can see, that with more than 18 used feature we have no F_1 score anymore. This is because the overtraining effect. If we call the model trained with 22 features on the train set itself, we get a perfect prediction, but on the test set we get a bad performance. Too many features cause the model to predict always 'p' for the test set, therefore we have 1 specificity but 0 sensitivity with the model.

We can get the exact feature count for the maximum F_1 score with the following R code:

```
#get the optimum feature count
opt_feature_count <- which.max(F1_scores$F1)
opt_feature_count
```

```
## [1] 9
```

The feature model reaches the best performance with 9 used features. The performance with this optimum is the following:

```r
#calculate the result for our feature count based model
result_feature_model <- confusionMatrix(predict_feature_model(test_data,
                        train_feature_model(train_data, opt_feature_count)), test_data$classes)
result_feature_model
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction   e   p
##          e 842   0
##          p   0 784
##
##                Accuracy : 1
##                  95% CI : (0.9977, 1)
##     No Information Rate : 0.5178
##     P-Value [Acc > NIR] : < 2.2e-16
##
##                   Kappa : 1
##
##  Mcnemar's Test P-Value : NA
##
##             Sensitivity : 1.0000
##             Specificity : 1.0000
##          Pos Pred Value : 1.0000
##          Neg Pred Value : 1.0000
##              Prevalence : 0.5178
##          Detection Rate : 0.5178
##    Detection Prevalence : 0.5178
##       Balanced Accuracy : 1.0000
##
##        'Positive' Class : e
##
```

With the optimum feature count trained feature model, we have F_1 score of 1 with a specificity 1. This model made a perfect prediction for the test set.

Conclusion

The mushrooms data set is a special data set, because it contains only categorical features.

I developed two different models for predict the edibility classification of the mushrooms. Both modesl reached a perfect prediction on the test set.

The developed decision tree model has a limitation because of the complexity of the decision trees. We can create all possible decision trees with a recursive function, but this function produces a huge number of decision trees. (I implemented the recursive function at first, but I had to limit the depth of the tree to get any result. with 4 tree levels I had more than 2.000 possible trees but not all of them were "completed".) The chosen selection of the used decision tree rule is not trainable. It would be interesting to investigate; how different selection criteria influences the decision tree performance. The found decision tree has only a depth of 4 levels and uses 4 features for prediction.

The decision tree model has limitation: if we don't have any feature with definitely 'e' or 'p' edibility, we cannot extend the decision tree. In this case, the decision tree model is not applicable.

The feature-based model uses the uncertainty score for ordering the features based on the importance. The model matches feature value combination to find the edible mushrooms in the test set. This model was built with 5-fold cross validation. The optimum feature model uses 9 features to predict the edible classification.

For this model we could try to calculate the F_1 score for all feature combination and choose the best combination. Unfortunately, the count of the combination explodes if we use more feature for the model. E.g. if we build the model with 4 features, we have $\binom{22}{4} = 175.560$ 4 feature combination.

The feature model uses feature value combinations, therefore is usable in situations where the decision tree is not useful.

For the mushrooms dataset we can use the knn method too to predict the edibility classification. The knn method reaches a perfect prediction with F_1 score of 1. (To use the knn method, we must remove the feature 'veil-type', because it has only one value.)

The following R code trains the knn method and calculates the method performance:

```
train_data2 <- train_data %>% select(-`veil-type`)
train_knn <- train(classes~., method='knn', data=train_data2)
confusionMatrix(predict(train_knn, test_data, type="raw"), test_data$classes)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction   e   p
##          e 842   0
##          p   0 784
##
##                Accuracy : 1
##                  95% CI : (0.9977, 1)
##     No Information Rate : 0.5178
##     P-Value [Acc > NIR] : < 2.2e-16
##
##                   Kappa : 1
##
##  Mcnemar's Test P-Value : NA
##
##             Sensitivity : 1.0000
##             Specificity : 1.0000
##          Pos Pred Value : 1.0000
##          Neg Pred Value : 1.0000
```

```
##              Prevalence : 0.5178
##          Detection Rate : 0.5178
##    Detection Prevalence : 0.5178
##       Balanced Accuracy : 1.0000
##
##        'Positive' Class : e
##
```