# Unit 6: Classification rules

## QHS 670

Vedbar Khadka, PhD
Assistant Professor
Quantitative Health Science Department
University of Hawaii John A. Burns School of Medicine

09/23/2022

# Outline

- Understanding classification rules
- Separate and Conquer
- 1R algorithm
- RIPPER algorithm
- Rules from decision trees
- What makes trees and rules greedy?
- Example – Identifying poisonous mushrooms using rule learners
  - Step 1 – collecting data
  - Step 2 – exploring and preparing data
  - Step 3 – training a model on the data
  - Step 4 – evaluating model performance
  - Step 5 – improving model performance
- Summary

# Understanding classification rules

- Classification rules represent knowledge in an **if-else** format.
- Rules involve the terms antecedent (**if** part or precondition) and consequent (**else** or **then** part).
  - For example, if students study 5 hours a week, then they will pass the class with an A
  - Antecedent: If students study 5 hours a week
  - Consequent: then they will pass the class with an A

```
if(<condition>) {
        ## do something
}
else {
        ## do something else
}
```

# Understanding classification rules

- Classification rules are highly similar to the decision trees.
- Primary difference
  - Decision trees involve a complex step-by-step process to make a decision. We do need to be familiar with the process that generated the decision.
  - Classification rules are stand-alone rules that are abstracted from a process. We do not need to be familiar with the process that created it.
- Classification rules are not as useful for large amounts of numeric variables.
  - This is not a problem with decision trees.

# Understanding classification rules

- Classification rules use algorithms that employ a **separate and conquer** heuristic.
  - Algorithm will try to separate the data into smaller and smaller subsets by generating enough rules to make homogeneous subsets.
  - Goal is to separate the observations in the data set into subgroups that have similar characteristics.
- Classification rules are a useful way to develop clear principles as found in the data.
  - Advantage: the approach is simple.
  - Disadvantage: numeric data is harder to use when trying to develop such rules.
- Common algorithms used in classification rules include the
  - One Rule Algorithm
  - RIPPER Algorithm

# Understanding classification rules

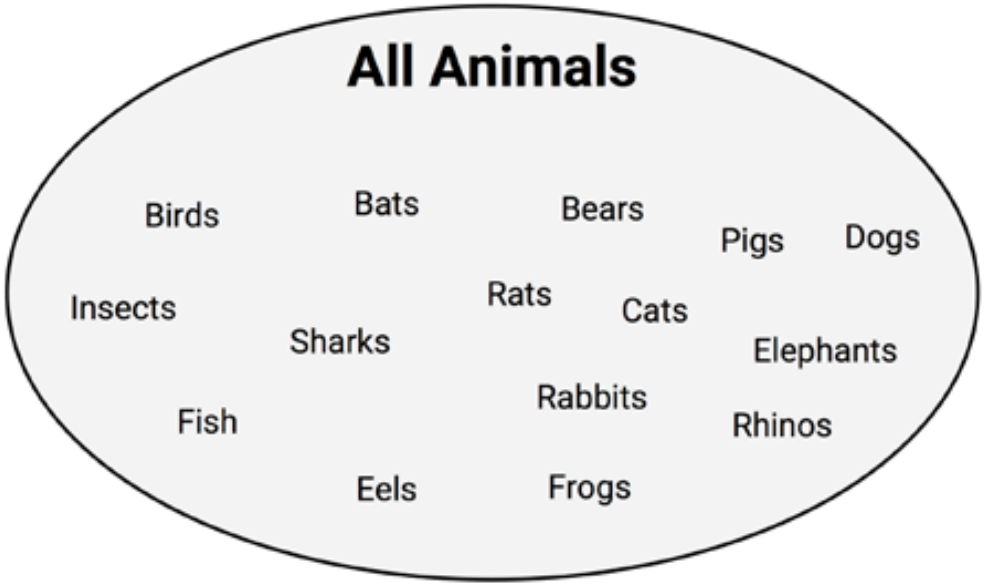
- One Rule Algorithm
  - Analyzes data and generates one all-encompassing rule.
  - Works by finding the single rule that contains the less amount of error.
  - Despite its simplicity, it is surprisingly accurate.
- RIPPER algorithm
  - Grows as many rules as possible. When a rule begins to become so complex that it no longer helps to purify the various groups the rule is pruned or the part of the rule that is not beneficial is removed.
  - This process of growing and pruning rules is continued until there is no further benefit.
- RIPPER algorithm rules are more complex than One Rule Algorithm.
  - This allows for the development of complex models.
  - Drawback is that the rules can become too complex to make practical sense.

# Separate and Conquer

- Classification rule learning algorithms utilize a heuristic known as separate and conquer.
- The process involves identifying a rule that covers a subset of observations in the training data, and then separating this partition from the remaining data.
- As the rules are added, additional subsets of the data are separated until the entire dataset has been covered and no more observations remain.

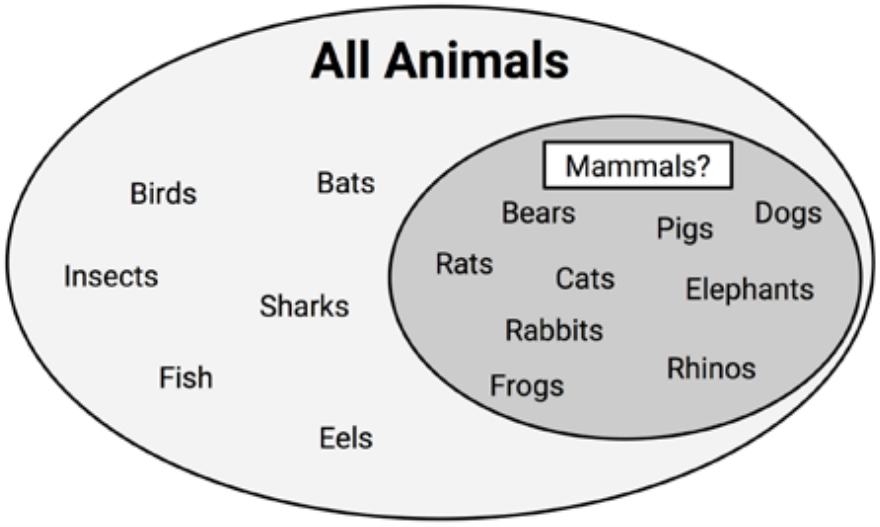
# Separate and Conquer - Example

- Suppose we were tasked with creating rules to identify whether or not an animal is a mammal.
- We could depict the set of all animals as a large space, as shown.



# Separate and Conquer - Example

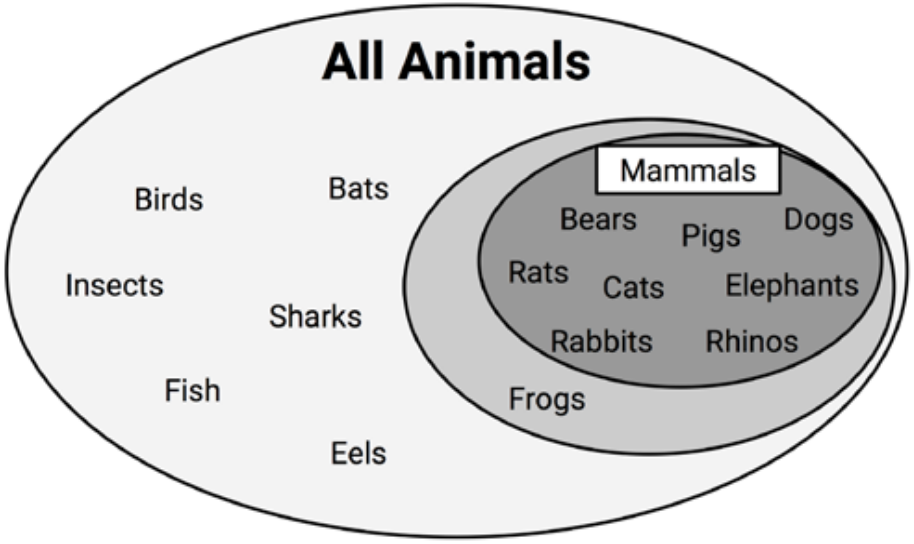
- A rule learner begins by using the available features to find homogeneous groups.
  - For example, using a feature “Travels By”, the first rule might suggest that any land-based animals are mammals.




| Animal | Travels By | Has Fur | Mammal |
|---|---|---|---|
| Bats | Air | Yes | Yes |
| Bears | Land | Yes | Yes |
| Birds | Air | No | No |
| Cats | Land | Yes | Yes |
| Dogs | Land | Yes | Yes |
| Eels | Sea | No | No |
| Elephants | Land | No | Yes |
| Fish | Sea | No | No |
| Frogs | Land | No | No |
| Insects | Air | No | No |
| Pigs | Land | No | Yes |
| Rabbits | Land | Yes | Yes |
| Rats | Land | Yes | Yes |
| Rhinos | Land | No | Yes |
| Sharks | Sea | No | No |

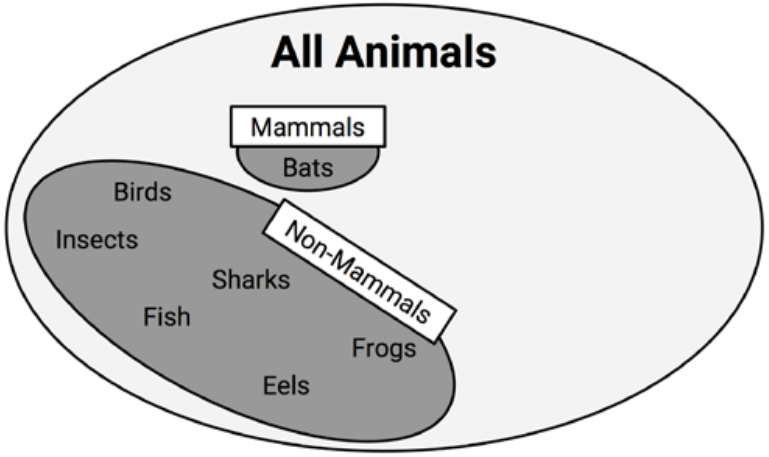
# Separate and Conquer - Example

- Do you notice any problems with this rule?
  - Frogs are amphibians, not mammals
- Our rule needs to be a bit more specific.
  - So, let's drill down further by suggesting that mammals must walk on land and have a tail:



# Separate and Conquer - Example

- An additional rule can be defined to separate out the bats, the only remaining mammal.
- A potential feature distinguishing bats from the other remaining animals would be the presence of fur.
- Using a rule built around this feature, we have then correctly identified all the animals:




**If the animal does not have fur, it is not a mammal**

Once all of the training instances have been classified, the rule learning process stops.

# Separate and Conquer - Example

- We learned a total of three rules:
  - Animals that walk on land and have tails are mammals
  - If the animal does not have fur, it is not a mammal
  - Otherwise, the animal is a mammal
- The example illustrates how rules gradually consume larger and larger segments of data to eventually classify all instances.
- As the rules seem to cover portions of the data, separate and conquer algorithms are also known as **covering algorithms**, and the resulting rules are called **covering rules**.

# ZeroR algorithm

- ZeroR (Zero Rule) is the simplest classification method which relies on the target and ignores all predictors
- ZeroR classifier simply predicts the most common class (majority category).
- There is no predictability power in ZeroR
  - Useful for determining a baseline performance as a benchmark for other classification methods.
- How it works?
  - Construct a frequency table for the target and select its most frequent value.

# ZeroR algorithm

- Suppose a wheel with 16 evenly sized colored slices. Four of the segments were colored red, four were blue, and eight were white. Prior to spinning the wheel, you are asked to choose one of these colors. When the wheel stops spinning, if the color shown matches your prediction, you will win a large cash prize. What color should you pick?
  - Answer: Of course, white. This is the most common color on the wheel.
- The ZeroR, a rule learner literally learns no rules.
  - It predicts the most common class.

# 1R algorithm

- The 1R algorithm (One Rule or OneR) improves over ZeroR by selecting a single rule.
  - This may seem overly simplistic but performs better than you might expect.
  - Accuracy of this algorithm can approach that of much more sophisticated algorithms for many real-world tasks.
- This algorithm works in a simple way.
  - For each feature, 1R divides the data into groups based on similar values of the feature.
  - Then, for each segment, the algorithm predicts the majority class.
  - The error rate for the rule based on each feature is calculated and the rule with the fewest errors is chosen as the one rule.

# 1R algorithm

Tables show how this would work for the animal data

| Animal | Travels By | Has Fur | Mammal |
|---|---|---|---|
| Bats | Air | Yes | Yes |
| Bears | Land | Yes | Yes |
| Birds | Air | No | No |
| Cats | Land | Yes | Yes |
| Dogs | Land | Yes | Yes |
| Eels | Sea | No | No |
| Elephants | Land | No | Yes |
| Fish | Sea | No | No |
| Frogs | Land | No | No |
| Insects | Air | No | No |
| Pigs | Land | No | Yes |
| Rabbits | Land | Yes | Yes |
| Rats | Land | Yes | Yes |
| Rhinos | Land | No | Yes |
| Sharks | Sea | No | No |

**Full Dataset**

| Travels By | Predicted | Mammal | |
|---|---|---|---|
| Air | No | Yes | ✖ |
| Air | No | No | |
| Air | No | No | |
| Land | Yes | Yes | |
| Land | Yes | Yes | |
| Land | Yes | Yes | |
| Land | Yes | Yes | |
| Land | Yes | No | ✖ |
| Land | Yes | Yes | |
| Land | Yes | Yes | |
| Land | Yes | Yes | |
| Land | Yes | Yes | |
| Sea | No | No | |
| Sea | No | No | |
| Sea | No | No | |

**Rule for "Travels By"**
**Error Rate = 2 / 15**

bats
frogs

| Has Fur | Predicted | Mammal | |
|---|---|---|---|
| No | No | No | |
| No | No | No | |
| No | No | Yes | ✖ |
| No | No | No | |
| No | No | No | |
| No | No | No | |
| No | No | Yes | ✖ |
| No | No | Yes | ✖ |
| No | No | No | |
| Yes | Yes | Yes | |
| Yes | Yes | Yes | |
| Yes | Yes | Yes | |
| Yes | Yes | Yes | |
| Yes | Yes | Yes | |
| Yes | Yes | Yes | |

**Rule for "Has Fur"**
**Error Rate = 3 / 15**

elephants,
pigs,
rhinos

# 1R algorithm

- For the “Travels By” feature, the dataset was divided into three groups: Air, Land, and Sea.
  - Animals in the Air and Sea groups were predicted to be non-mammals, while animals in the Land group were predicted to be mammals.
  - This resulted in two errors: bats and frogs.
- For the “Has Fur” feature, the dataset was divided animals into two groups: Yes and No
  - Those with fur were predicted to be mammals, while those without fur were not predicted to be mammals.
  - Three errors were counted: pigs, elephants, and rhinos.

# 1R algorithm

- As the “Travels By” feature results in fewer errors, the 1R algorithm will return the following "one rule" based on Travels By:
  - If the animal travels by air, it is not a mammal
  - If the animal travels by land, it is a mammal
  - If the animal travels by sea, it is not a mammal
- The algorithm stops here, having found the single most important rule.
- Obviously, this rule learning algorithm may be too basic for some tasks.
  - Would you want a medical diagnosis system to consider only a single symptom, or an automated driving system to stop or accelerate your car based on only a single factor?
  - For these types of tasks, a more sophisticated rule learner might be useful.

# 1R algorithm

**Strength**

- Generates a single, easy-to-understand, human-readable rule of thumb
- Often performs surprisingly well
- Can serve as a benchmark for more complex algorithms

**Weaknesses**

- Uses only a single feature
- Probably overly simplistic

# RIPPER algorithm

- Early rule learning algorithms (Separate and conquer, and 1R algorithm) were plagued by a couple of problems.
  - First, they were notorious for being slow, which made them ineffective for the increasing number of large datasets.
  - Secondly, they were often prone to being inaccurate on noisy data.
- Johannes Furnkranz and Gerhard Widmer in 1994 proposed a step toward solving these problems.
  - Their **I**ncremental **R**educed **E**rror **P**runing (IREP) algorithm uses a combination of pre-pruning and post-pruning methods that grow very complex rules and prune them before separating the instances from the full dataset.

# RIPPER algorithm

- The RIPPER algorithm was introduced by William W. Cohen in 1995
  - Improve on IREP to generate rules that match or exceed the performance of decision trees.
  - RIPPER stands for **R**epeated **I**ncremental **P**runing to **P**roduce **E**rror **R**eduction
- Having evolved from several iterations of rule learning algorithms, the RIPPER algorithm can be understood in a three-step process.
  - They are: Grow, Prune, Optimize

# RIPPER algorithm

1. Grow:
   - Uses the separate and conquer technique to greedily add conditions to a rule until it perfectly classifies a subset of data or runs out of attributes for splitting.
   - Similar to decision trees, the information gain criterion is used to identify the next splitting attribute.
2. Prune:
   - When increasing a rule's specificity no longer reduces entropy, the rule is immediately pruned.
3. Optimize:
   - Steps one and two are repeated until it reaches a stopping criterion, at which point the entire set of rules is optimized using a variety of heuristics.

# RIPPER algorithm

- The RIPPER algorithm can create much more complex rules than can the 1R algorithm
  - It can consider more than one feature.
  - This means that it can create rules with multiple antecedents such as "**if** an animal flies **and** has fur, **then** it is a mammal."
- This improves the algorithm's ability to model complex data, but just like decision trees, it means that the rules can quickly become more difficult to comprehend.

# RIPPER algorithm

**Strengths**

- Generates easy-to-understand, human-readable rules
- Efficient on large and noisy datasets
- Generally produces a simpler model than a comparable decision tree

**Weaknesses**

- May result in rules that seem to defy common sense or expert knowledge
- Not ideal for working with numeric data
- Might not perform as well as more complex models

# Rules from decision trees

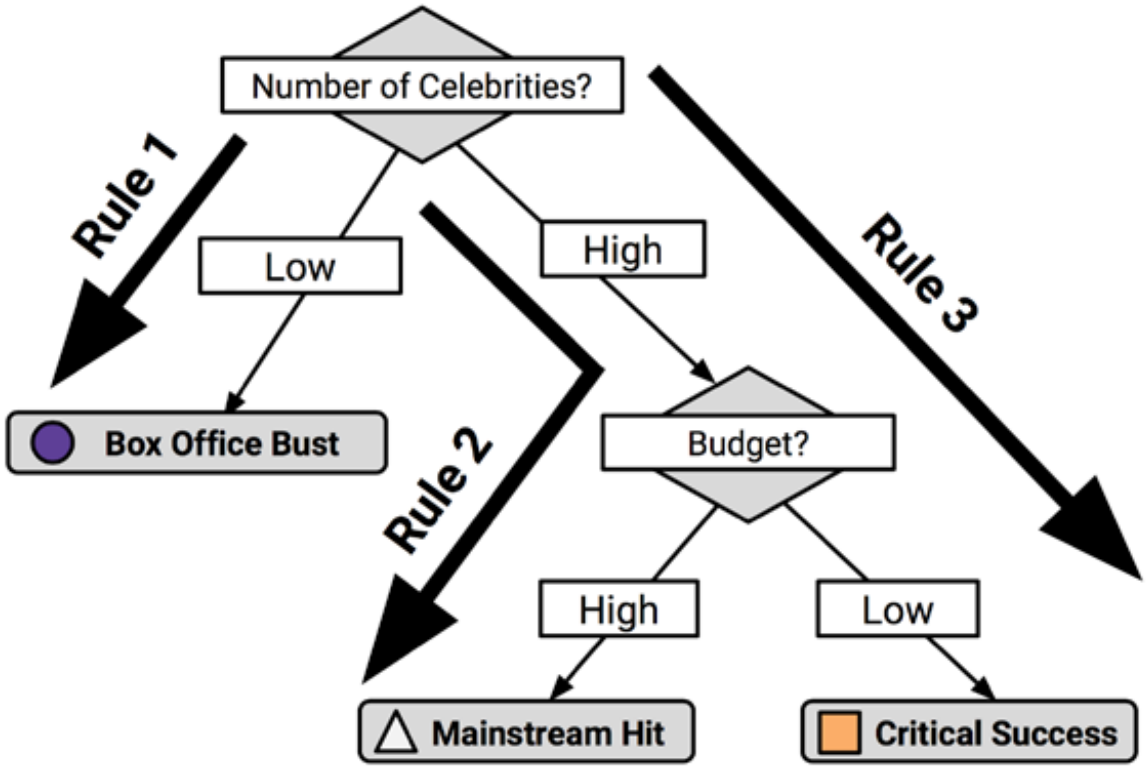



Figure shows how rules could be constructed from the decision tree to predict movie success:

# Rules from decision trees

- Classification rules can also be obtained directly from decision trees.
- Beginning at a leaf node and following the branches back to the root, you will have obtained a series of decisions.
- These can be combined into a single rule.
- The following Following the paths from the root node down to each leaf, the rules would be:
  1. If the number of celebrities is low, then the movie will be a Box Office Bust.
  2. If the number of celebrities is high and the budget is high, then the movie will be a Mainstream Hit.
  3. If the number of celebrities is high and the budget is low, then the movie will be a Critical Success.

# Rules from decision trees

- The main downside to using a decision tree to generate rules is that the resulting rules are often more complex than those learned by a rule learning algorithm.
- The divide and conquer strategy employed by decision trees biases the results differently than that of a rule learner.
- On the other hand, it is sometimes more computationally efficient to generate rules from trees.

# Greedy Algorithm

- An approach for solving a problem by selection the best option available at the moment (always chooses the steps which provide immediate profit/benefit).
- Does not worry whether the current best result will bring the overall optimal result.
- The algorithm never reverses the earlier decision even if the choice is wrong.
  - It works in a top-down approach.

# What makes trees and rules greedy?

- Decision trees and rule learners are known as greedy learners
  - They use data on a first-come, first-served basis.
- Both the “divide and conquer” heuristic used by decision trees and the “separate and conquer” heuristic used by rule learners attempt to make partitions one at a time, finding the most homogeneous partition first, followed by the next best, and so on, until all examples have been classified.

# What makes trees and rules greedy?

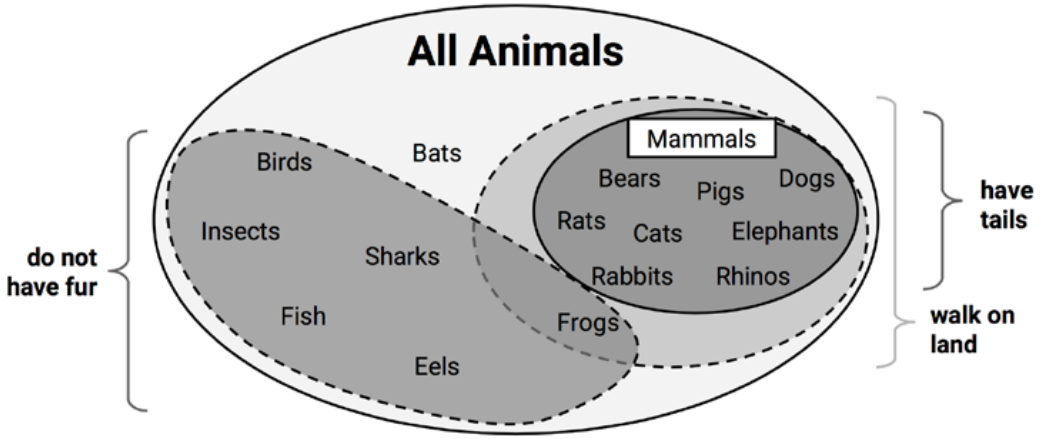
- The downside to the greedy approach is that greedy algorithms are not guaranteed to generate the optimal, most accurate, or smallest number of rules for a particular dataset.
  - By taking the low-hanging fruit early, a greedy learner may quickly find a single rule that is accurate for one subset of data;
  - However, in doing so, the learner may miss the opportunity to develop a more nuanced set of rules with better overall accuracy on the entire set of data.
- Without using the greedy approach, rule learning would be computationally infeasible for larger datasets

# What makes trees and rules greedy?

- Though both trees and rules employ greedy learning heuristics, there are subtle differences in how they build rules.
  - Trees: once divide and conquer splits on a feature, the partitions created by the split may not be re-conquered, only further subdivided. A tree is permanently limited by its history of past decisions.
  - Rules: once separate and conquer finds a rule, any examples not covered by all of the rule's conditions may be re-conquered.

# What makes trees and rules greedy?

- To illustrate this contrast, consider the previous case in which we built a rule learner to determine whether an animal was a mammal.
- The rule learner identified three rules that perfectly classify the example animals:
  - Animals that walk on land and have tails are mammals (bears, cats, dogs, elephants, pigs, rabbits, rats, rhinos)
  - If the animal does not has fur, it is not a mammal (birds, eels, fish, frogs, insects, sharks)
  - Otherwise, the animal is a mammal (bats)

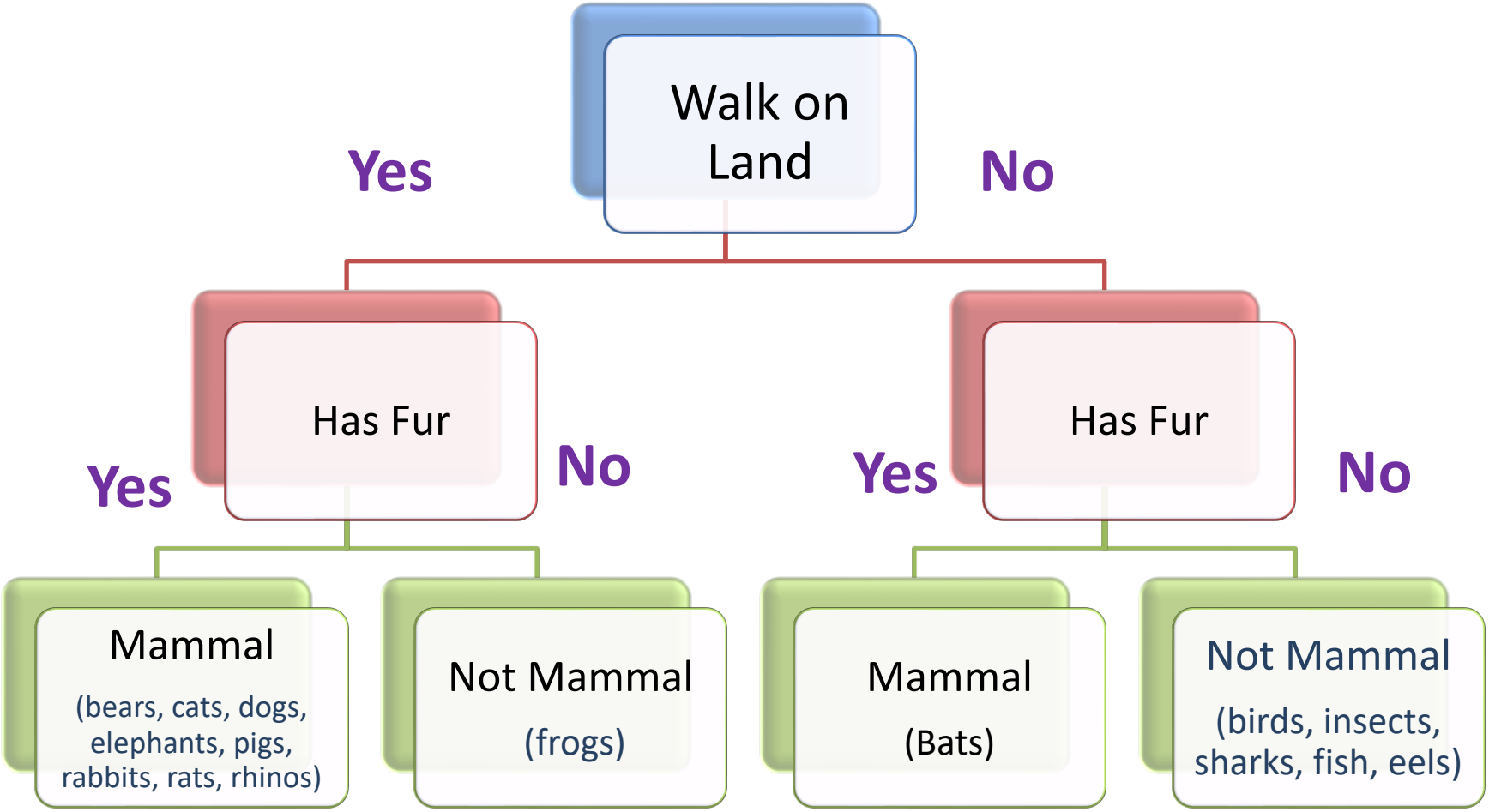



Frogs re-conquered!

# What makes trees and rules greedy?

- In contrast, a decision tree built on the same data might have come up with four rules to achieve the same perfect classification:
  - If an animal walks on land and has fur, then it is a mammal (bears, cats, dogs, elephants, pigs, rabbits, rats, rhinos)
  - If an animal walks on land and does not have fur, then it is not a mammal (frogs)
  - If the animal does not walk on land and has fur, then it is a mammal (bats)
  - If the animal does not walk on land and does not have fur, then it is not a mammal (birds, insects, sharks, fish, eels)

# What makes trees and rules greedy?



# What makes trees and rules greedy?

- The different result across these two approaches has to do with what happens to the frogs after they are separated by the "walk on land" decision.
  - Decision tree cannot modify the existing partitions, and therefore must place the frog into its own rule.
  - Rule learner allows frogs to be re-conquered by the "does not have fur" decision.
- Rule learners often find a more parsimonious set of rules than those generated from decision trees
  - Can reexamine cases that were considered but ultimately not covered as part of prior rules
- Computational cost of rule learners may be somewhat higher than for decision trees
  - Due to reuse of data

# Identifying poisonous mushrooms

- Example with rule learners

# Identifying poisonous mushrooms

Each year, many people fall ill and sometimes even die from ingesting poisonous wild mushrooms. Since many mushrooms are very similar to each other in appearance, occasionally even experienced mushroom gatherers are poisoned. Unlike the identification of harmful plants such as a poison oak or poison ivy, there are no clear rules such as "leaves of three, let them be" to identify whether a wild mushroom is poisonous or edible. Complicating matters, many traditional rules, such as "poisonous mushrooms are brightly colored," provide dangerous or misleading information. If simple, clear, and consistent rules were available to identify poisonous mushrooms, they could save the lives of foragers.

Because one of the strengths of rule learning algorithms is the fact that they generate easy-to-understand rules, they seem like an appropriate fit for this classification task. However, the rules will only be as useful as they are accurate.

# Step 1 – collecting data

- To identify rules for distinguishing poisonous mushrooms, we will utilize the Mushroom dataset by Jeff Schlimmer of Carnegie Mellon University.
- The raw dataset is available freely at the UCI Machine Learning Repository (http://archive.ics.uci.edu/ml).
- The dataset includes information on 8,124 mushroom samples from 23 species of gilled mushrooms listed in Audubon Society Field Guide to North American Mushrooms (1981).
- In the Field Guide, each of the mushroom species is identified "definitely edible," "definitely poisonous," or "likely poisonous, and not recommended to be eaten."
- For the purposes of this dataset, the latter group was combined with the "definitely poisonous" group to make two classes: poisonous and edible (nonpoisonous ).

# Step 2 – Exploring and preparing the data

- Import the data for the analysis

```
mushrooms <- read.csv("mushrooms.csv", stringsAsFactors = TRUE)
# examine the structure of the data frame
str(mushrooms)
```

# Step 2 – Exploring and preparing the data

- There are 8124 observations and 23 variables.
- Data dictionary is available on the UCI website for the mushroom samples.

```
> str(mushrooms)
'data.frame':   8124 obs. of  23 variables:
 $ type                    : Factor w/ 2 levels "edible","poisonous": 2 1 1 2 1 1 1 1 2 1 ...
 $ cap_shape               : Factor w/ 6 levels "bell","conical",..: 3 3 1 3 3 3 1 1 3 1 ...
 $ cap_surface             : Factor w/ 4 levels "fibrous","grooves",..: 4 4 4 3 4 3 4 3 3 4 ...
 $ cap_color               : Factor w/ 10 levels "brown","buff",..: 1 10 9 9 4 10 9 9 9 10 ...
 $ bruises                 : Factor w/ 2 levels "no","yes": 2 2 2 2 1 2 2 2 2 2 ...
 $ odor                    : Factor w/ 9 levels "almond","anise",..: 8 1 2 8 7 1 1 2 8 1 ...
 $ gill_attachment         : Factor w/ 2 levels "attached","free": 2 2 2 2 2 2 2 2 2 2 ...
 $ gill_spacing            : Factor w/ 2 levels "close","crowded": 1 1 1 1 2 1 1 1 1 1 ...
 $ gill_size               : Factor w/ 2 levels "broad","narrow": 2 1 1 2 1 1 1 1 2 1 ...
 $ gill_color              : Factor w/ 12 levels "black","brown",..: 1 1 2 2 1 2 5 2 8 5 ...
 $ stalk_shape             : Factor w/ 2 levels "enlarging","tapering": 1 1 1 1 2 1 1 1 1 1 ...
 $ stalk_root              : Factor w/ 5 levels "bulbous","club",..: 3 2 2 3 3 2 2 2 3 2 ...
 $ stalk_surface_above_ring: Factor w/ 4 levels "fibrous","scaly",..: 4 4 4 4 4 4 4 4 4 4 ...
 $ stalk_surface_below_ring: Factor w/ 4 levels "fibrous","scaly",..: 4 4 4 4 4 4 4 4 4 4 ...
 $ stalk_color_above_ring  : Factor w/ 9 levels "brown","buff",..: 8 8 8 8 8 8 8 8 8 8 ...
 $ stalk_color_below_ring  : Factor w/ 9 levels "brown","buff",..: 8 8 8 8 8 8 8 8 8 8 ...
 $ veil_type               : Factor w/ 1 level "partial": 1 1 1 1 1 1 1 1 1 1 ...
 $ veil_color              : Factor w/ 4 levels "brown","orange",..: 3 3 3 3 3 3 3 3 3 3 ...
 $ ring_number             : Factor w/ 3 levels "none","one","two": 2 2 2 2 2 2 2 2 2 2 ...
 $ ring_type               : Factor w/ 5 levels "evanescent","flaring",..: 5 5 5 5 1 5 5 5 5 5 ...
 $ spore_print_color       : Factor w/ 9 levels "black","brown",..: 1 2 2 1 2 1 1 2 1 1 ...
 $ population              : Factor w/ 6 levels "abundant","clustered",..: 4 3 3 4 1 3 3 4 5 4 ...
 $ habitat                 : Factor w/ 7 levels "grasses","leaves",..: 5 1 3 5 1 1 3 3 1 3 ...
 $ habitat                 : Factor w/ 7 levels "grasses","leaves",..: 5 1 3 5 1 1 3 3 1 3 ...
```

# Step 2 – Exploring and preparing the data

- Anything peculiar about the veil_type variable in the following line?

  $ veil_type : Factor w/ 1 level "partial": 1 1 1 1 1 1 ...

  – Since the veil type does not vary across samples, it does not provide any useful information for prediction.
  – We will drop this variable from our analysis using the following command:

```
# drop the veil_type feature
mushrooms$veil_type <- NULL
# examine the class distribution
table(mushrooms$type)
```

```
> table(mushrooms$type)

edible poisonous
  4208      3916
```

# Step 2 – Exploring and preparing the data

- Distribution of the mushroom type class variable
  - About 52% of the mushroom samples (N = 4,208) are edible, while 48% (N = 3,916) are poisonous.
- We will consider the 8,214 samples in the mushroom data to be an exhaustive set of all the possible wild mushrooms.
- This is an important assumption!
  - It means that we do not need to hold some samples out of the training data for testing purposes.
  - We are not trying to develop rules that cover unforeseen types of mushrooms
- We are simply trying to find rules that accurately depict the complete set of known mushroom types.
  - Therefore, we can build and test the model on the same data.

# Step 3 – Training a model on the data

- If we trained a hypothetical ZeroR classifier on this data, what would it predict?
  - ZeroR ignores all of the features and simply predicts the target's mode
  - Its rule would state that all the mushrooms are edible.
  - Obviously, this is not a very helpful classifier
- Our rules will need to do much better and need to be simple/easy to remember.
- Simple rules can often be extremely predictive
  - So, let's see how a very simple rule learner performs on the mushroom data.

# Step 3 – Training a model on the data

- We will apply the 1R classifier, which will identify the most predictive single feature of the target class and use it to construct a set of rules.
- We will use the 1R implementation in the RWeka package called OneR().
- install.packages("RWeka") command and have Java installed on your system.
- Then, load the package by typing library(RWeka):

# 1R classification rule syntax

**1R classification rule syntax**

using the `OneR()` function in the `RWeka` package

**Building the classifier:**

```
m <- OneR(class ~ predictors, data = mydata)
```

- `class` is the column in the `mydata` data frame to be predicted
- `predictors` is an R formula specifying the features in the `mydata` data frame to use for prediction
- `data` is the data frame in which `class` and `predictors` can be found

The function will return a 1R model object that can be used to make predictions.

**Making predictions:**

```
p <- predict(m, test)
```

- `m` is a model trained by the `OneR()` function
- `test` is a data frame containing test data with the same features as the training data used to build the classifier.

The function will return a vector of predicted class values.

**Example:**

```
mushroom_classifier <- OneR(type ~ odor + cap_color,
                            data = mushroom_train)
mushroom_prediction <- predict(mushroom_classifier,
                               mushroom_test)
```

# Step 3 – Training a model on the data

- OneR() implementation uses the R formula syntax to specify the model to be trained.
- The formula syntax uses the ~ operator (known as the tilde) to express the relationship between a target variable and its predictors.
- The class variable to be learned goes to the left of the tilde, and the predictor features are written on the right, separated by + operators.
  - To model the relationship between the y class and predictors x1 and x2, you could write the formula as y ~ x1 + x2.
  - To include all the variables in the model, the special term . can be used.

```
#install.packages("OneR")
library(OneR)
# train OneR() on the data
mushroom_1R <- OneR(type ~ ., data = mushrooms)
mushroom_1R
```

# Step 3 – Training a model on the data

```
> mushroom_1R

Call:
OneR.formula(formula = type ~ ., data = mushrooms)

Rules:
If odor = almond   then type = edible
If odor = anise    then type = edible
If odor = creosote then type = poisonous
If odor = fishy    then type = poisonous
If odor = foul     then type = poisonous
If odor = musty    then type = poisonous
If odor = none     then type = edible
If odor = pungent  then type = poisonous
If odor = spicy    then type = poisonous

Accuracy:
8004 of 8124 instances classified correctly (98.52%)
```

**In the first line of the output, the odor feature was selected for rule generation.**

**For the purposes of a field guide for mushroom gathering, these rules could be summarized in a simple rule of thumb: "if the mushroom smells unappetizing, then it is likely to be poisonous."**

# Step 4 – Evaluating model performance

```
mushroom_1R_pred <- predict(mushroom_1R, mushrooms)
table(actual = mushrooms$type, predicted = mushroom_1R_pred)
```

```
> mushroom_1R_pred <- predict(mushroom_1R, mushrooms)
> table(actual = mushrooms$type, predicted = mushroom_1R_pred)
           predicted
actual      edible poisonous
  edible      4208         0
  poisonous    120      3796
```

- The rules correctly predicted the edibility of 8,004 of the 8,124 mushroom samples or nearly 99% of the mushroom samples.
- Although the 1R classifier did not classify any edible mushrooms as poisonous, it did classify 120 poisonous mushrooms as edible—**which makes for an incredibly dangerous mistake!**

# Step 4 – Evaluating model performance

- Considering that the learner utilized only a single feature, it did reasonably well
  - If one avoids unappetizing smells when seeking mushrooms, they will almost avoid a trip to the hospital.
- Let's see if we can add a few more rules and develop an even better classifier.

# Step 5 – improving model performance

- For a more sophisticated rule learner, we will use JRip(), a Java-based implementation of the RIPPER rule learning algorithm.
- JRip() is included in the RWeka package.
- Load the package using the library(RWeka) command:

```
#install.packages("RWeka")
library(RWeka)
mushroom_JRip <- JRip(type ~ ., data = mushrooms)
mushroom_JRip
summary(mushroom_JRip)
```

# RIPPER classification rule sysntax

**RIPPER classification rule syntax**

using the `JRip()` function in the `RWeka` package

**Building the classifier:**

```
m <- JRip(class ~ predictors, data = mydata)
```

- `class` is the column in the `mydata` data frame to be predicted
- `predictors` is an R formula specifying the features in the `mydata` data frame to use for prediction
- `data` is the data frame in which `class` and `predictors` can be found

The function will return a RIPPER model object that can be used to make predictions.

**Making predictions:**

```
p <- predict(m, test)
```

- `m` is a model trained by the `JRip()` function
- `test` is a data frame containing test data with the same features as the training data used to build the classifier.

The function will return a vector of predicted class values.

**Example:**

```
mushroom_classifier <- JRip(type ~ odor + cap_color,
                            data = mushroom_train)
mushroom_prediction <- predict(mushroom_classifier,
                               mushroom_test)
```

# Step 5 – improving model performance

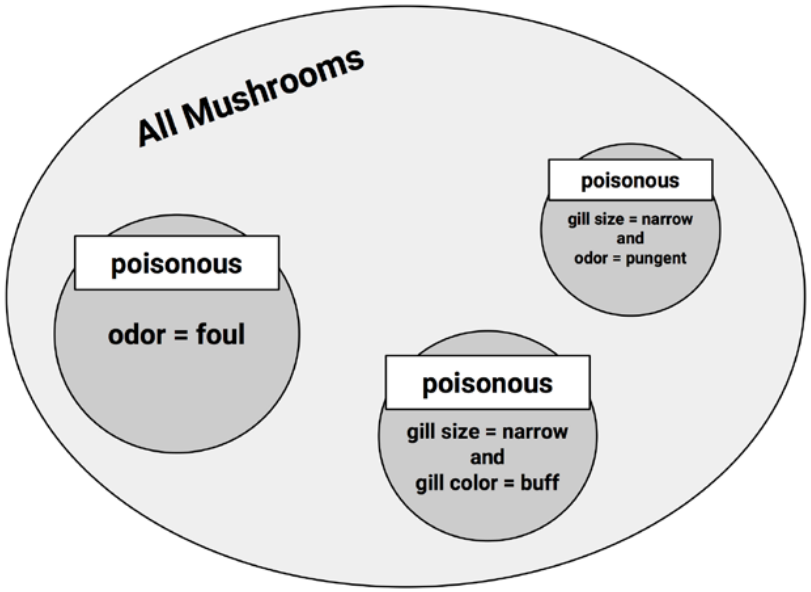
```
> mushroom_JRip
JRIP rules:
===========

(odor = foul) => type=poisonous (2160.0/0.0)
(gill_size = narrow) and (gill_color = buff) => type=poisonous (1152.0/0.0)
(gill_size = narrow) and (odor = pungent) => type=poisonous (256.0/0.0)
(odor = creosote) => type=poisonous (192.0/0.0)
(spore_print_color = green) => type=poisonous (72.0/0.0)
(stalk_surface_below_ring = scaly) and (stalk_surface_above_ring = silky) => type=poisonous (68.0/0.0)
(habitat = leaves) and (cap_color = white) => type=poisonous (8.0/0.0)
(stalk_color_above_ring = yellow) => type=poisonous (8.0/0.0)
 => type=edible (4208.0/0.0)

Number of Rules : 9


> summary(mushroom_JRip)

=== Summary ===

Correctly Classified Instances        8124               100      %
Incorrectly Classified Instances         0                 0      %
Kappa statistic                          1
Mean absolute error                      0
Root mean squared error                  0
Relative absolute error                  0      %
Root relative squared error              0      %
Total Number of Instances             8124

=== Confusion Matrix ===

    a    b   <-- classified as
 4208    0 |    a = edible
    0 3916 |    b = poisonous
```

# Step 5 – improving model performance

- The JRip() classifier learned a total of 9 rules from the mushroom data.
- Read these rules as a list of if-else statements.
- The first three rules could be expressed as:
  - If the odor is foul, then the mushroom type is poisonous
  - If the gill size is narrow and the gill color is buff, then the mushroom type is poisonous
  - If the gill size is narrow and the odor is pungent, then the mushroom type is poisonous
- Finally, the ninth rule implies that any mushroom sample that was not covered by the preceding eight rules is edible.
- Following the example of our programming logic (if-else statements), this can be read as:
  - Else, the mushroom is edible.

# Step 5 – improving model performance

- Notably, there were no misclassified mushroom samples using these nine rules (**achieve 100% accuracy!**).
- Number of instances covered by the last rule is exactly equal to the number of edible mushrooms in the data (N = 4,208).
- The algorithm found a group of poisonous mushrooms uniquely distinguished by their foul odor, some smaller and more specific groups of poisonous mushrooms
  - Except for those, all of the remaining mushrooms were found to be edible.




**Rough illustration on how the rules are applied to the mushroom data.**

# Step 5 – improving model performance

- Rule Learner using C5.0 Decision Trees

```
library(C50)
mushroom_c5rules <- C5.0(type ~ odor + gill_size, data = mushrooms, rules = TRUE)
summary(mushroom_c5rules)
```

```
> summary(mushroom_c5rules)

Call:
C5.0.formula(formula = type ~ odor + gill_size, data = mushrooms, rules = TRUE)


C5.0 [Release 2.07 GPL Edition]         Sat Feb 26 16:23:04 2022
-------------------------------

Class specified by attribute `outcome'

Read 8124 cases (3 attributes) from undefined.data

Rules:

Rule 1: (4328/120, lift 1.9)
        odor in {almond, anise, none}
        ->  class edible  [0.972]

Rule 2: (3796, lift 2.1)
        odor in {creosote, fishy, foul, musty, pungent, spicy}
        ->  class poisonous  [1.000]

Default class: edible


Evaluation on training data (8124 cases):

             Rules
       ----------------
         No      Errors

          2  120( 1.5%)   <<


          (a)   (b)    <-classified as
         ----  ----
         4208          (a): class edible
          120  3796    (b): class poisonous


        Attribute usage:

        100.00% odor


Time: 0.0 secs
```

# Summary

- Two classification methods (Decision trees and classification rules) use so-called "greedy" algorithms to partition the data according to feature values.
  - Decision trees use a divide and conquer strategy to create flowchart-like structures,
  - Rule learners use a separate and conquer strategy to identify logical if-else rules.
- Both methods produce models that can be interpreted without a statistical background.
- Two rule learners, 1R and RIPPER, develop rules to identify poisonous mushrooms.
  - The 1R algorithm used a single feature to achieve 99% accuracy in identifying potentially fatal mushroom samples.
  - More sophisticated RIPPER algorithm correctly identified the edibility of each mushroom generating a set of nine rules.