Loops

Data Set Analysis

Thomas Schwarz, SJ
Marquette University

Loops

- Computer Science knows three types of loops
  - Count driven
    - The loop in C, Java, …
    - Python emulates it with ranks: `for i in range(100):`
  - Condition driven
    - This is typical for while loops
  - Collection controlled loop:
    - This is the Python for-loop
    - Collection can be any generator, file, list, dictionary, tuple, …

Python Iterators

- Python iterators are not covered in this course, but you ought to be aware of this concept
  - An iterator has a function next
  - When an iterator runs out of objects to provide on a next, it will create a StopIteration exception
  - We can emulate this behavior in a while loop

Python Iterators

```
numbers = [3,5,7,11,13,17,19,23,29,31]
num_iterator = iter(numbers)
while num_iterator:
    try:
        current_number = next(num_iterator)
        print(current_number)
    except StopIteration:
        break
```

Creating an iterator

Python Iterators

```
numbers = [3,5,7,11,13,17,19,23,29,31]
num_iterator = iter(numbers)
while True:
    try:
        current_number = next(num_iterator)
        print(current_number)
    except StopIteration:
        break
```

Looping

Python Iterators

```
numbers = [3,5,7,11,13,17,19,23,29,31]
num_iterator = iter(numbers)
while True:
    try:
        current_number = next(num_iterator)
        print(current_number)
    except StopIteration:
        break
```

Getting the next item

Python Iterators

```
numbers = [3,5,7,11,13,17,19,23,29,31]
num_iterator = iter(numbers)
while True:
    try:
        current_number = next(num_iterator)
        print(current_number)
    except StopIteration:
        break
```

Handling the exception generated when next fails

Python Generators

- Python allows you to define generators
  - We do not discuss generators in this course but you ought to be aware of their existence
- A generator object creates a sequence of objects
- A generator just creates a generator object
  - Looks like a function, but has a yield instead of a return

Python Generators

```
def fib_generator():
    previous, current = 0, 1
    while True:
        previous, current = current, previous+current
        yield current
```

Generators look like functions !

Python Generators

```
def fib_generator():
    previous, current = 0, 1
    while True:
        previous, current = current, previous+current
        yield current
```

But have a “yield” instead of a “return”

Python Generators

```
def fib_generator():
    previous, current = 0, 1
    while True:
        previous, current = current, previous+current
        yield current
```

If this were a function, it would return just one element

Python Generators

```
def fib_generator():
    previous, current = 0, 1
    while True:
        previous, current = current, previous+current
        yield current
```

But a generator keeps on yielding

Python Generator

- This Python generator will generate all the Fibonacci numbers

While Loops

While Loops

- Controlled by a condition
  - Normal way to leave a loop is for the condition to become False

```
def heron(a):
    x = 1
    while abs(x*x-a) > 1e-12:
        x = (a/x + x)/2
    return x
```

While Loop

- Loop termination statements
    - A <u>break</u> statement jumps out of a loop
    - A <u>continue</u> statement will restart the loop

While Loop

- The else statement:
    - Put after the end of the loop
    - Executed if the loop condition is false
    - “else” chosen instead of “finally” because Python did not want to introduce new key words

While Loops

- Used in searches that need post-processing if nothing is found

```
def sum_of_divisors(n):
    result = 0
    for i in range(1,n//2+1):
        if n%i==0:
            result += i
    return result
```

```
def perfect(x, y):
    for i in range(x, y):
        if sum_of_divisors(i)==i:
            return i
    else:
        print("nothing found")
```

Decision Trees

Decision Trees

- One of many machine learning methods
  - Used to learn categories
- Example:
  - The Iris Data Set
    - Four measurements of flowers
    - Learn how to predict species from them

Iris Data Set

Iris Setosa

Iris Virginica

Iris Versicolor

Iris Data Set

- Data in a .csv file
  - Collected by Fisher
  - One of the most famous datasets
    - Look it up on Kaggle or at UC Irvine Machine Learning Repository

- Want to learn to distinguish Iris Versicolor and Iris Virginica

Iris Data Set

- Read the data set
  - Program included in the attached Python file
  - You might want to follow along on by programming

Measuring Purity

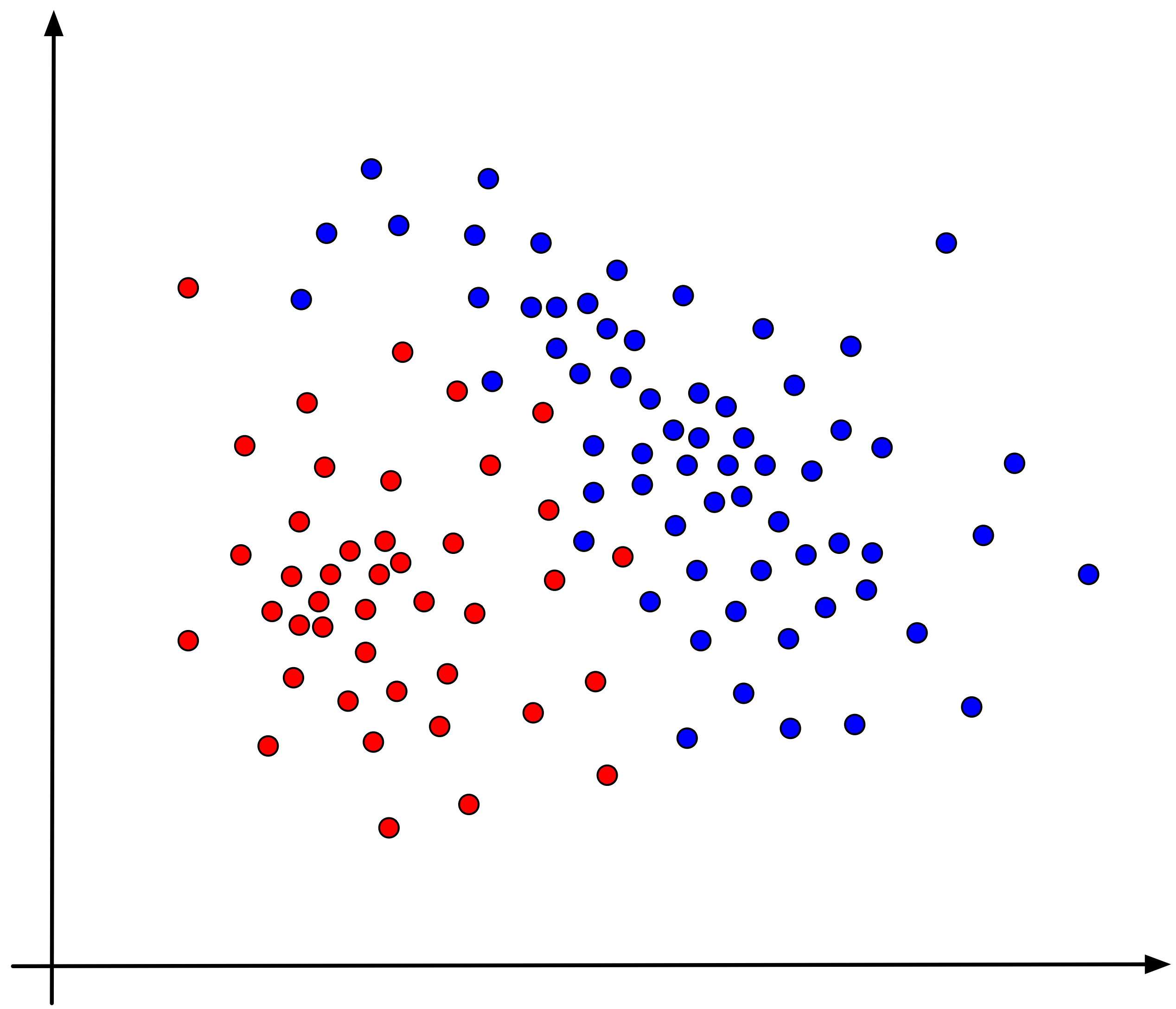
- Several measures of purity
  - Gini Index of Purity
  - Entropy
    - In the case of two categories with p and q proportions, defined as

$$\textbf{Entropy}(p, q) = \log_2(p)p + \log_2(q)q$$

    - Unless one of the proportions is zero, in which case the entropy is zero.
  - High entropy means low purity, low entropy means high purity

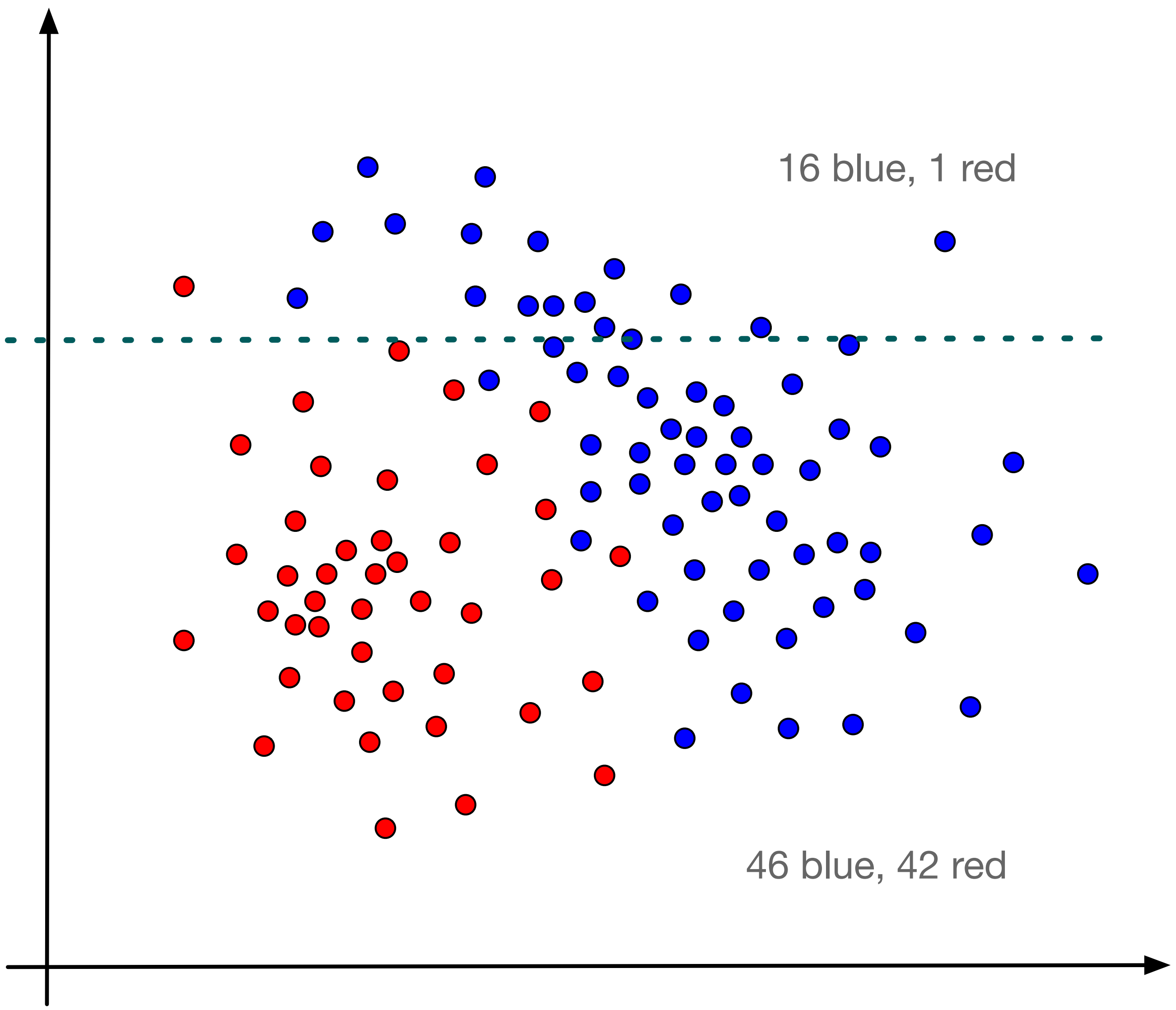
Building a Decision Tree

- A decision tree
  - Can we predict the category (red vs blue) of the data from its coordinates?



Building a Decision Tree

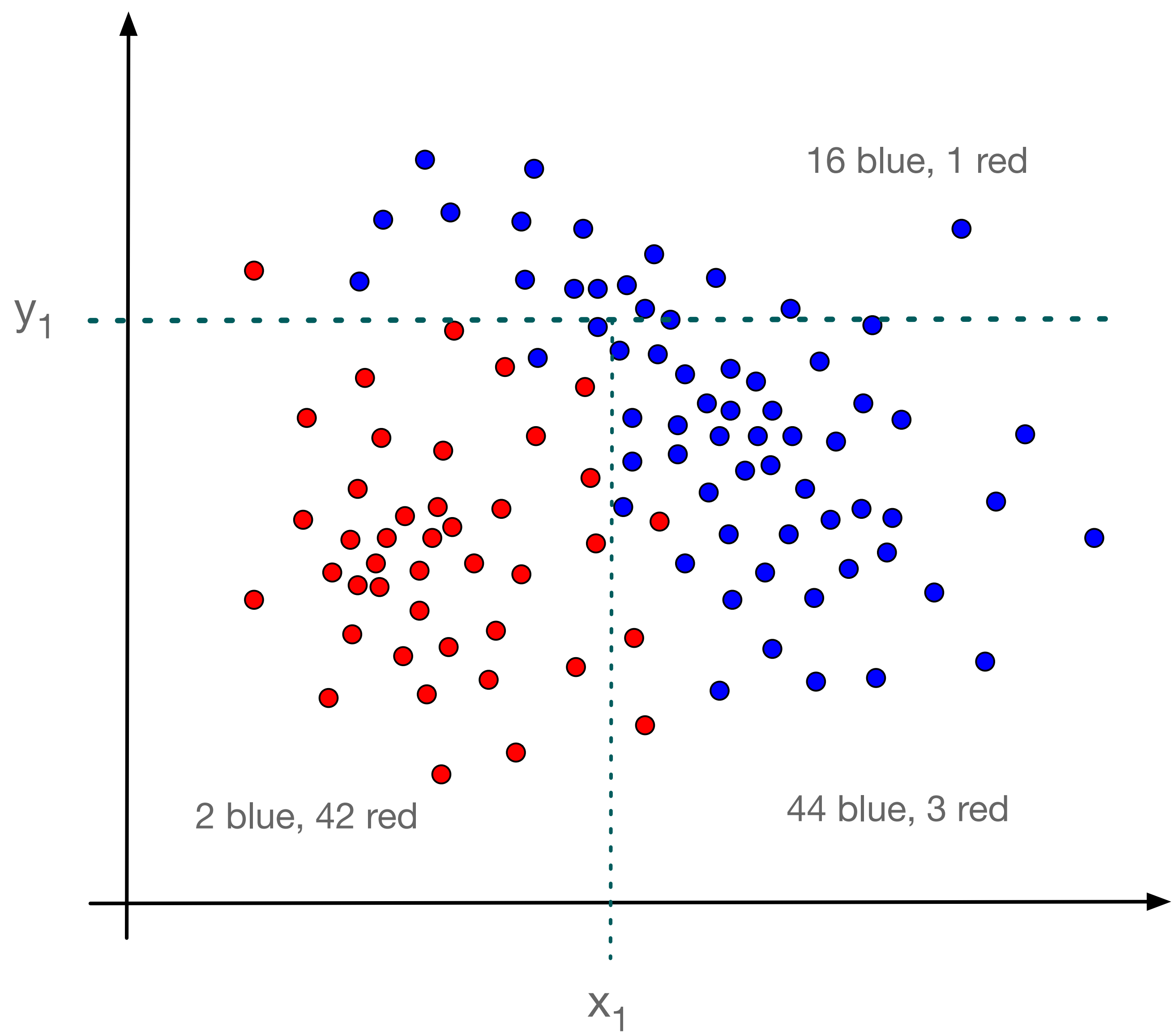
- Introduce a single boundary



Almost all points above the line are

Building a Decision Tree

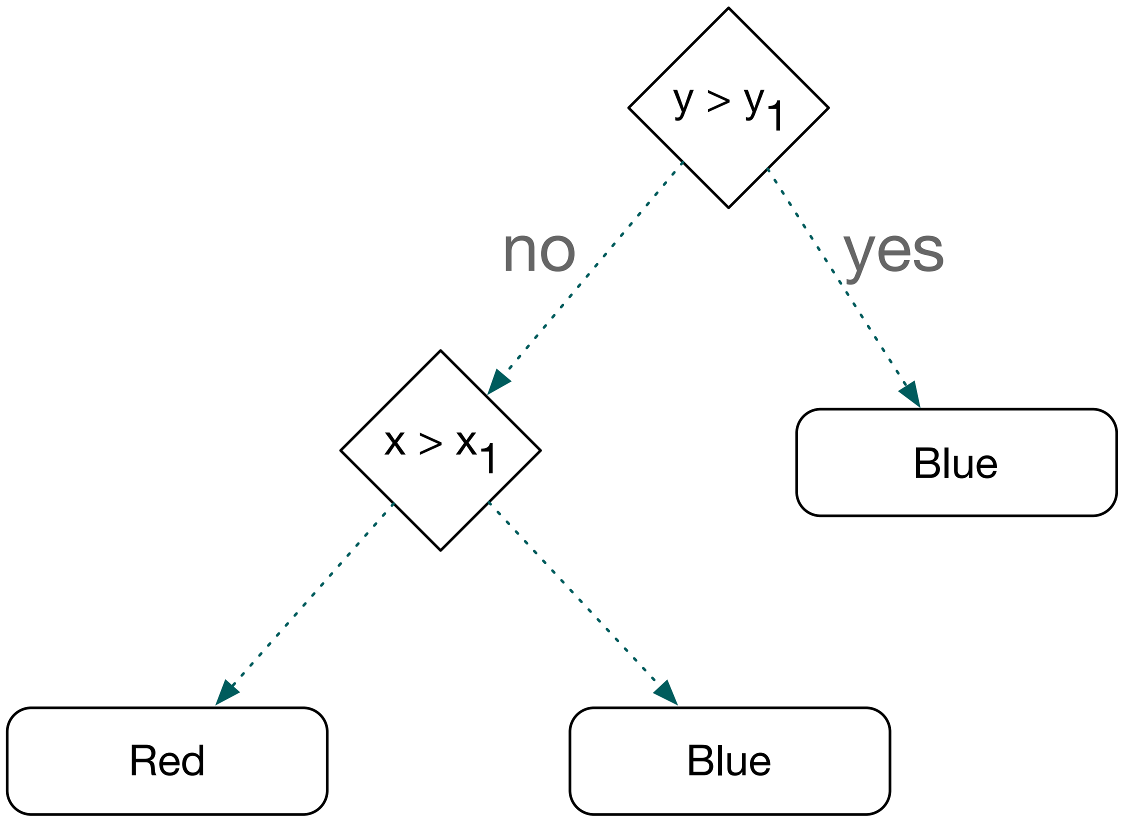
- Subdivide the area below the line

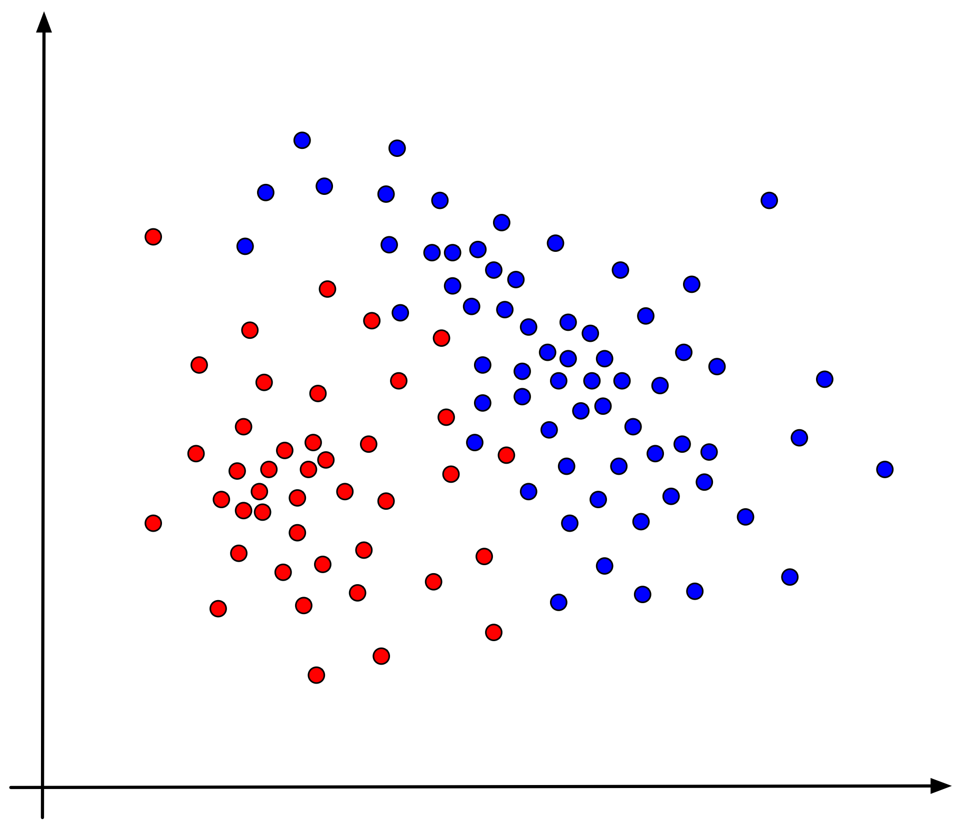


Defines three almost homogeneous r

Building a Decision Tree

- Express as a decision tree





Building a Decision Tree

- If a new point with coordinates (x, y) is considered
  - Use the decision tree to predict the color of the point
- Decision tree is not always correct even on the points used to develop it
  - But it is mostly right
- If new points behave like the old ones
  - Expect the rules to be mostly correct

Building a Decision Tree

- Decision trees can be used to predict behavior
  - People with similar behavior have stopped patronizing the enterprise
    - Assume that we can predict clients likely to jump ship
    - Offer special incentives so that they stay with us
  - This is called churn management and it can make lots of money

Building a Decision Tree

- How do we build decision trees
  - First rule: Decisions should be simple, involving only one coordinate
  - Second rule: If decision rules are complex they are likely to not generalize
    - E.g.: the lone red point in the upper region is probably an outlier and not indicative of general behavior

Building a Decision Tree

- Algorithm for decision trees:
  - Find a simple rule that yields a division into two regions that are <u>more</u> homogeneous than the original one
  - Continue sub-diving the regions
    - Stop when a region is homogeneous or almost homogeneous
    - Stop when a region becomes too small

Building a Decision Tree

- We need to try all possible boundaries and all possible regions
  - We better write some helper functions to help us

Processing Iris

- First, get the data

```
>>> irises = get_data()
>>> len(irises)
100
>>> count(irises)
(50, 50)
>>> entropy(irises)
1.0
>>>
```

- 100 tuples, half with Virginica, half with Versicolor

Processing Iris

```
[(7.0, 3.2, 4.7, 1.4, 'Iris-versicolor'),
(6.4, 3.2, 4.5, 1.5, 'Iris-versicolor'),
(6.9, 3.1, 4.9, 1.5, 'Iris-versicolor'),
(5.5, 2.3, 4.0, 1.3, 'Iris-versicolor'),
(6.5, 2.8, 4.6, 1.5, 'Iris-versicolor'),
…
…
(6.7, 3.0, 5.2, 2.3, 'Iris-virginica'),
(6.3, 2.5, 5.0, 1.9, 'Iris-virginica'),
(6.5, 3.0, 5.2, 2.0, 'Iris-virginica'), (
6.2, 3.4, 5.4, 2.3, 'Iris-virginica'),
(5.9, 3.0, 5.1, 1.8, 'Iris-virginica')]
```

Processing Iris

- We can divide the list according to coordinate and value
  - We can see an increase in homogeneity, but it is not substantial

```
>>> l1, l2 = divide(irises, 1, 3.0)
>>> count(l1)
(33, 42)
>>> count(l2)
(17, 8)
```

Processing Iris

- We pick a coordinate.
  - We sort the tuple values in this coordinate
  - We make sure that they are unique
  - We then create a list of midpoints

```
sorted(tupla[1] for tupla in irises)
[2.0, 2.2, 2.2, 2.2, 2.3, 2.3, 2.3, 2.4,
2.4, 2.4, 2.5, 2.5, 2.5, 2.5, 2.5, 2.5,
2.5, 2.5, 2.6, 2.6, 2.6, 2.6, 2.6, 2.7,
2.7, 2.7, 2.7, 2.7, 2.7, 2.7, 2.7, 2.7,
2.8, 2.8, 2.8, 2.8, 2.8, 2.8, 2.8, 2.8,
2.8, 2.8, 2.8, 2.8, 2.8, 2.8, 2.9, 2.9,
2.9, 2.9, 2.9, 2.9, 2.9, 2.9, 2.9, 3.0,
3.0, 3.0, 3.0, 3.0, 3.0, 3.0, 3.0, 3.0,
3.0, 3.0, 3.0, 3.0, 3.0, 3.0, 3.0, 3.0,
3.0, 3.0, 3.0, 3.1, 3.1, 3.1, 3.1, 3.1,
3.1, 3.1, 3.2, 3.2, 3.2, 3.2, 3.2, 3.2,
3.2, 3.2, 3.3, 3.3, 3.3, 3.3, 3.4, 3.4,
3.4, 3.6, 3.8, 3.8]
>>> midpoints(tupla[1] for tupla in
irises)
[2.1, 2.25, 2.3499999999999996, 2.45,
2.55, 2.6500000000000004, 2.75,
2.8499999999999996, 2.95, 3.05,
3.1500000000000004, 3.25,
3.3499999999999996, 3.5, 3.7]
```

Processing Iris

- For each midpoint, we split the set and calculate the weighted entropy of the resulting split
- We do this for all coordinates:

```
>>> for i in range(4):
   print(i, find_best_value(irises, i))


0 (5.75, 0.1682616579400087)
1 (2.45, 0.0739610509320755)
2 (4.75, 0.7268460660521441)
3 (1.65, 0.6474763214577008)
```

- And select the best gain: coordinate 2 with 4.75

Processing Iris

- We split into two lists: left and right

```
>>> left, right = divide(irises, 2, 4.75)
>>> count(left)
(1, 44)
>>> count(right)
(49, 6)
```

  - left is almost completely Iris Versicolor
  - right needs to be subdivided

Processing Iris

- Since the right set is already pretty homogeneous, the gains are not as large as before

```
>>> for i in range(4):
   print(i, find_best_value(right, i))


0 (7.0, 0.00522989837660498)
1 (3.25, 0.0031757407862335607)
2 (5.05, 0.041343407685332456)
3 (1.75, 0.07488163300231473)
```

- Select coordinate 3 with value 1.75

Processing Iris

- We split the right list accordingly

```
>>> rightleft, rightright = divide(right, 3, 1.75)
>>> count(rightleft)
(4, 5)
>>> count(rightright)
(45, 1)
```

- The list rightright looks good, but rightleft can be improved

Processing Iris

- We find the best way to split

```
>>> for i in range(4):
   print(i, find_best_value(rightleft, i))


0 (6.5, 0.10417849406014013)
1 (2.75, 0.007965292443227856)
2 (5.05, 0.2472576473434122 7)
3 (1.45, 0)
```

- and split again in coordinate 2, but with value 5.05

Processing Iris

- The results now fulfill our stopping criteria:

```
>>> rightleftleft, rightleftright = divide(rightleft, 2, 5.05)
>>> count(rightleftleft)
(1, 4)
>>> count(rightleftright)
(3, 1)
```

Processing Iris

- We summarize (and use the names of the columns instead of the number)