Ch.17 Decision Trees

Decision Trees

- DataSciencester의 인재 부사장(VP of Talent)은 여러 구직자를 인터뷰했습니다.
  - 이 데이터를 사용하여 어떤 후보자가 인터뷰를 잘 볼지를 식별하는 모델을 구축할 수 있나요?

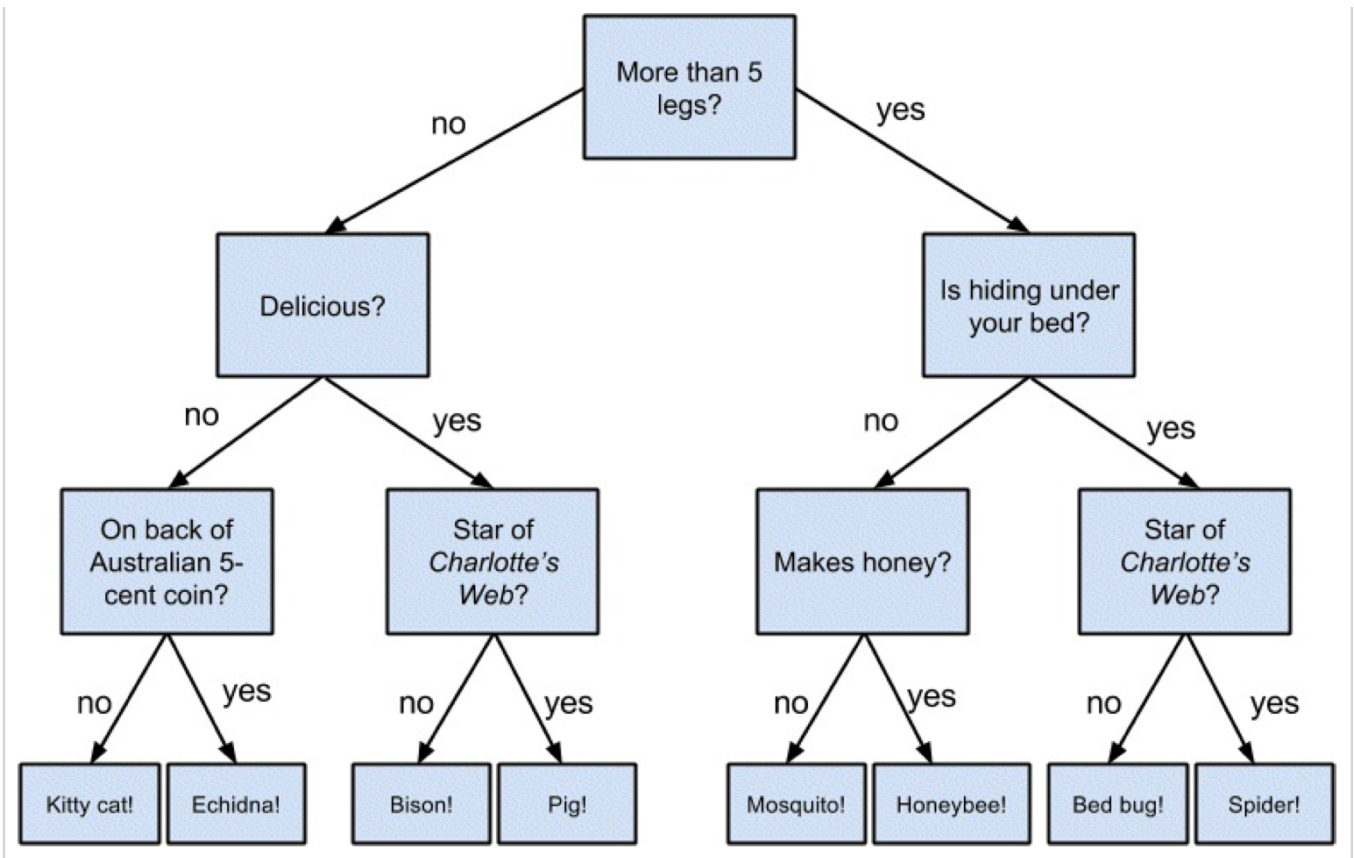
What is a Decision Trees (결정 트리란 무엇인가?)

만약 여러분이 **스무고개** 게임을 해본 적이 있다면, 결정 트리에 익숙할 것입니다. 예를 들어:

- "나는 동물을 생각하고 있어요."
- "다섯 개 이상의 다리를 가지고 있나요?"
- "아니요."
- "맛있나요?"
- "아니요."
- "호주 5센트 동전 뒷면에 나오나요?"
- "네."
- "바늘두더지인가요?"
- "네, 맞아요!"

이는 다음 경로에 해당합니다:

- "다섯 개 이상의 다리를 가지지 않음" → "맛있지 않음" → "5센트 동전에 나옴" → "바늘두더지!"
- 이는 특이하고 (아주 포괄적이지 않은) "동물 맞추기" 결정 트리입니다.



Decision Trees (결정 트리)

장점:

- 이해하고 **해석**하기 매우 쉽습니다.
- 예측에 도달하는 과정이 완전히 **투명**합니다.
- 숫자형 (예: 다리 수)과 범주형 (예: 맛있다/맛없다) 속성의 혼합을 쉽게 처리할 수 있습니다.

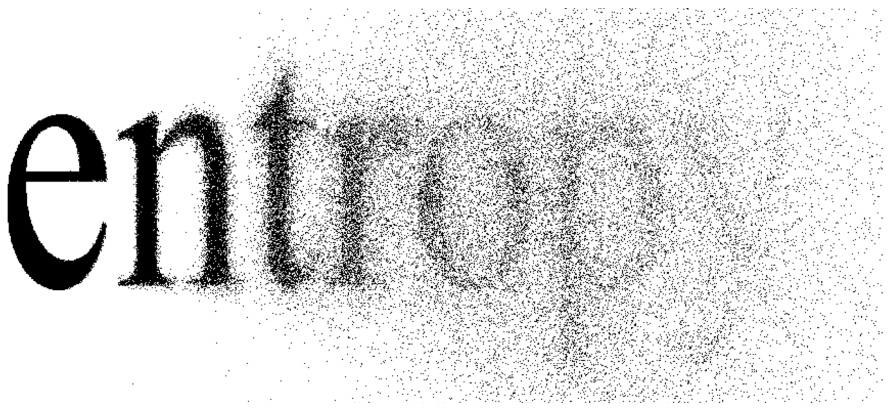
단점:

- 훈련 데이터 세트에 대해 "최적의" 결정 트리를 찾는 것은 **계산적으로 매우 어려운** 문제입니다.
- 훈련 데이터에 **과적합**되고 보지 못한 데이터에 잘 일반화되지 않는 결정 트리를 쉽게 (그리고 매우 나쁘게) 만들 수 있습니다.

ID3 알고리즘

- 라벨이 지정된 데이터 세트로부터 결정 트리를 학습하는 알고리즘입니다.
- 이진 분류에 중점을 둡니다:
  - "이 후보자를 고용해야 할까요?" 또는
  - "이 웹사이트 방문자에게 광고 A를 보여줘야 할까요, 아니면 광고 B를 보여줘야 할까요?"

Entropy (엔트로피)



데이터 집합 S가 있고, 각 데이터가 유한한 수의 클래스 C1,...,Cn 중 하나에 속한다고 가정해봅시다.

- **낮은 엔트로피**: 모든 데이터 포인트가 하나의 클래스에 속한다면, **실질적인 불확실성**이 없습니다.
- **높은 엔트로피**: 데이터 포인트가 클래스에 고르게 분포되어 있다면, **불확실성이 많습니다**.
- 간단히 말해서, 여기서 H(S)는 집합 S의 엔트로피를 나타내고, Pi는 클래스 Ci 에 속하는 데이터 포인트의 비율입니다.

$$H(S) = -\sum_{i} p_i \log(p_i)$$

Entropy in Binary Classes: 엔트로피 계산 (이진 클래스)

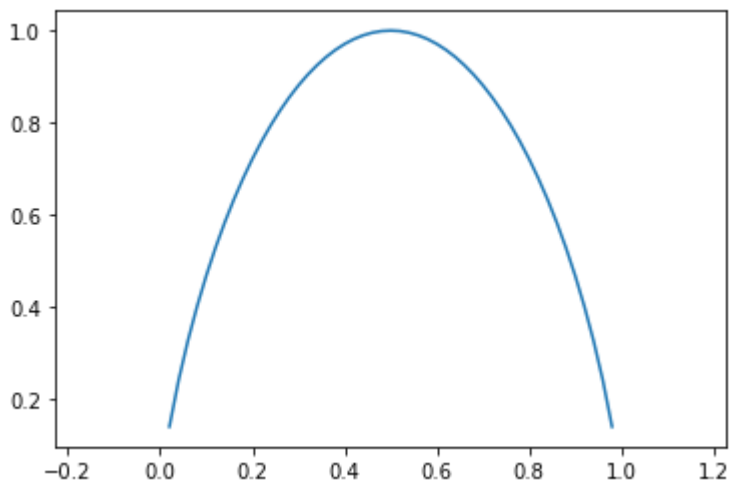
```
import numpy as np
import matplotlib.pyplot as plt

# 0과 1 사이를 균등하게 나눈 51개의 점 중 끝점을 제외한 부분 설정
x = np.linspace(0, 1, 51, endpoint=True)[1:-1]

# 엔트로피 함수 정의 (이진 클래스)
entropy = lambda p: -(p * np.log2(p) + (1 - p) * np.log2(1 - p))
```

```
# 엔트로피 함수 그래프 그리기
plt.plot(x, entropy(x))
plt.xlabel('Probability of class 1')
plt.ylabel('Entropy')
plt.title('Entropy for Binary Classification')
plt.axis('equal')
plt.show()
```

- 엔트로피는 확률이 0이나 1에 가까울수록 낮고, 0.5일 때 가장 높습니다. 이는 클래스 분포의 불확실성을 나타냅니다.




```
from collections import Counter, defaultdict
from functools import partial
import math, random
```

```
def entropy(class_probabilities):
    """주어진 클래스 확률 목록을 기반으로 엔트로피를 계산합니다."""
    return sum(-p * math.log(p, 2) for p in class_probabilities if p)
```

- `class_probabilities`는 클래스 확률 목록입니다.
- 각 확률 `p`에 대해, `p * math.log(p, 2)` 값을 계산하여 엔트로피를 구합니다.
- `p`가 0인 경우에는 로그가 정의되지 않으므로 이를 제외합니다.

```
# 예제 엔트로피 계산
print(entropy([1]))               # 0.0
print(entropy([0.5, 0.5]))        # 1.0
print(entropy([0, 1]))            # 0.0
print(entropy([1/3, 1/3, 1/3]))   # 1.584962500721156
print(entropy([1/4, 1/4, 1/4, 1/4])) # 2.0
```

- 클래스 확률이 `[1]`인 경우, 엔트로피는 `0.0`입니다. 이는 불확실성이 없음을 의미합니다.
- 클래스 확률이 `[0.5, 0.5]`인 경우, 엔트로피는 `1.0`입니다. 이는 최대 불확실성을 의미합니다.
- 클래스 확률이 `[0, 1]`인 경우, 엔트로피는 `0.0`입니다. 이는 불확실성이 없음을 의미합니다.
- 클래스 확률이 `[1/3, 1/3, 1/3]`인 경우, 엔트로피는 약 `1.584`입니다.
- 클래스 확률이 `[1/4, 1/4, 1/4, 1/4]`인 경우, 엔트로피는 `2.0`입니다.

💡 print(entropy([1])) → -log2

```
def class_probabilities(labels):
    """주어진 레이블 목록을 기반으로 클래스 확률을 계산합니다."""
    total_count = len(labels)
    return [count / total_count for count in Counter(labels).values()]
```

```
# 예제 클래스 확률 계산
print(class_probabilities([0, 0, 1, 1, 1]))  # [0.4, 0.6]
print(entropy(class_probabilities([1, 1, 1, 1, 1])))  # 0.0
```

```
def data_entropy(labeled_data):
    """주어진 레이블된 데이터를 기반으로 엔트로피를 계산합니다."""
    labels = [label for _, label in labeled_data]
    probabilities = class_probabilities(labels)
    return entropy(probabilities)
```

The Entropy of a Partition (파티션의 엔트로피)

- 의사 결정 트리의 각 단계는 **데이터를 하나 또는 (가능하면) 더 많은 하위 집합으로 나누는 질문을 하는 것**입니다.
  - 예를 들어, "다리가 다섯 개 이상 있나요?"
- 우리는 파티션이 낮은 엔트로피를 가지기를 원합니다:
  - 엔트로피가 낮은 하위 집합(즉, 매우 확실한)으로 데이터를 나누는 것입니다.

이 질문이 좋은 질문인가요?

- "Australian five-cent coin?"
  - echidna (작고 낮은 엔트로피 그룹)와
  - 나머지 모든 것 (크고 높은 엔트로피 그룹)

수학적으로, 데이터를 S1, ..., Sn 비율 q1, ..., qn 으로 포함하는 하위 집합으로 나누면, 파티션의 엔트로피를 가중 합으로 계산할 수 있습니다:

$$H = q_1 H(S_1) + \ldots + q_m H(S_m)$$

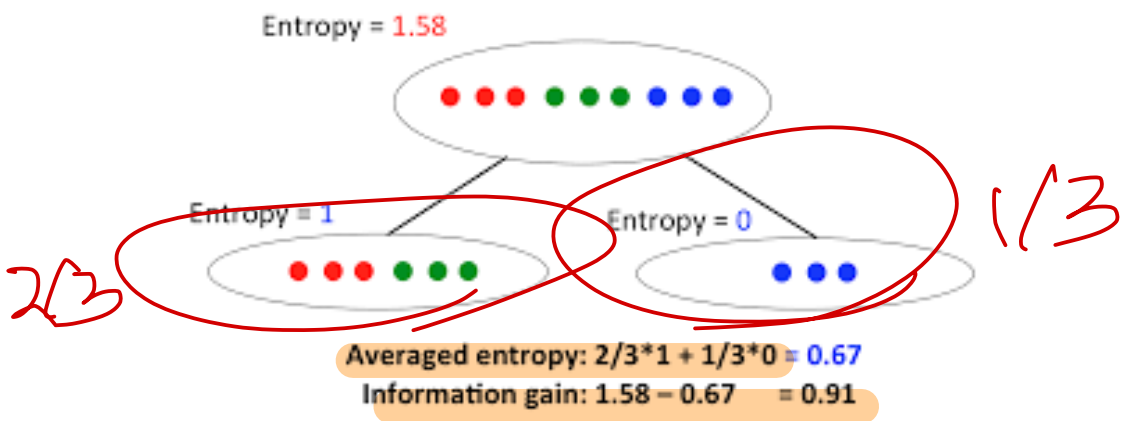
```
def partition_entropy(subsets):
    """하위 집합으로 데이터를 나누어 엔트로피를 계산합니다."""
    total_count = sum(len(subset) for subset in subsets)  # 모든 하위 집합의 총 개수

    # 각 하위 집합의 엔트로피에 하위 집합의 비율을 곱하여 가중 합을 구합니다.
    return sum(data_entropy(subset) * len(subset) / total_count
               for subset in subsets)
```

```
partition_entropy([[(_,0),(_,1)],[(_,1),(_,1)]])

# 0.5
```

Information Gain



Dataset

- 데이터:
  - 각 후보자의 수준(level)
  - 선호하는 언어(preferred language)
  - Twitter 활동 여부(active on Twitter)
  - 박사 학위 소지 여부(PhD)
- 라벨:
  - True (후보자가 인터뷰를 잘 봤음) 또는
  - False (후보자가 인터뷰를 잘 못 봤음)
- 이 간단한 구현에서는 카테고리 속성만 고려하며(연속 속성은 제외) 데이터셋을 구성합니다.

```
inputs = [
    ({'level':'Senior','lang':'Java','tweets':'no','phd':'no'},   False),
    ({'level':'Senior','lang':'Java','tweets':'no','phd':'yes'},  False),
    ({'level':'Mid','lang':'Python','tweets':'no','phd':'no'},     True),
    ({'level':'Junior','lang':'Python','tweets':'no','phd':'no'},  True),
    ({'level':'Junior','lang':'R','tweets':'yes','phd':'no'},      True),
    ({'level':'Junior','lang':'R','tweets':'yes','phd':'yes'},    False),
    ({'level':'Mid','lang':'R','tweets':'yes','phd':'yes'},        True),
    ({'level':'Senior','lang':'Python','tweets':'no','phd':'no'}, False),
    ({'level':'Senior','lang':'R','tweets':'yes','phd':'no'},      True),
    ({'level':'Junior','lang':'Python','tweets':'yes','phd':'no'}, True),
    ({'level':'Senior','lang':'Python','tweets':'yes','phd':'yes'},True),
    ({'level':'Mid','lang':'Python','tweets':'no','phd':'yes'},    True),
    ({'level':'Mid','lang':'Java','tweets':'yes','phd':'no'},      True),
    ({'level':'Junior','lang':'Python','tweets':'no','phd':'yes'},False)
]
```

Simple ID3 Algorithm (간단한 ID3 알고리즘):

- 만약 데이터의 모든 레이블이 **동일하다면**, 해당 레이블을 예측하는 리프 노드를 생성하고 멈춥니다.
- 속성 리스트가 비어있다면(즉, 더 이상 질문할 수 있는 속성이 없다면), **가장 일반적인 레이블**을 예측하는 리프 노드를 생성하고 멈춥니다.
- 그렇지 않다면, 각 속성으로 데이터를 분할해봅니다. (**몇 개로?**).
- **엔트로피가 가장 낮은 분할**을 선택합니다.
- 선택한 속성에 기반한 의사 결정 노드를 추가합니다.
- **남은 속성들을 사용하여 각 분할된 하위 집합에 대해 재귀적으로 처리**합니다.

Greedy Algorithm (탐욕 알고리즘):

- 이 알고리즘은 "탐욕" 알고리즘으로 알려져 있습니다. 왜냐하면 각 단계에서 **가장 즉각적으로 최선의 옵션**을 선택하기 때문입니다.
- 주어진 데이터 셋에서, **첫 번째 선택이 덜 좋아 보이는 경우 더 나은 트리**가 있을 수 있습니다.
- 그렇다면, 이 알고리즘은 그것을 찾지 못할 것입니다.
- 그럼에도 불구하고, 상대적으로 이해하기 쉽고 구현하기 쉬워서 의사결정 트리를 탐색하는 좋은 출발점이 됩니다.

```
def group_by(items, key_fn):
    """key_fn(item)의 결과를 키로 하여 items를 defaultdict(list)에 그룹화하여 반환합니다"""
    groups = defaultdict(list)
    for item in items:
        key = key_fn(item)
        groups[key].append(item)
    return groups

def partition_by(inputs, attribute):
    """attribute에 의해 inputs를 분할하여 dict로 반환합니다
    각 input은 (attribute_dict, label) 쌍입니다"""
    return group_by(inputs, lambda x: x[0][attribute])

def partition_entropy_by(inputs, attribute):
    """주어진 분할에 해당하는 엔트로피를 계산합니다"""
    partitions = partition_by(inputs, attribute)
    return partition_entropy(partitions.values())
```

```
# 각 속성에 대해 분할 엔트로피를 계산하고 출력
for key in ['level', 'lang', 'tweets', 'phd']:
    print(f"{key}: {partition_entropy_by(inputs, key):.4f}")
```

```
level 0.6935361388961919
lang 0.8601317128547441
tweets 0.7884504573082896
phd 0.8921589282623617
```

- 가장 낮은 엔트로피는 **level**을 기준으로 분할할 때 발생하므로, 각 가능한 **level** 값에 대해 서브트리를 만들어야 합니다.

```
# 'inputs' 리스트에서 'level'이 'Senior'인 데이터만 필터링하여 'senior_inputs' 리스트 생성
senior_inputs = [(input, label) for input, label in inputs if input["level"] == "Senior"]

# 'senior_inputs' 데이터에 대해 'lang', 'tweets', 'phd' 속성을 기준으로 엔트로피 계산
for key in ['lang', 'tweets', 'phd']:
    print(key, partition_entropy_by(senior_inputs, key))
```

```
lang 0.4
tweets 0.0
phd 0.9509775004326938
```

```
# 'inputs' 리스트에서 'level'이 'Mid'인 데이터만 필터링하여 'mid_inputs' 리스트 생성
mid_inputs = [(input, label) for input, label in inputs if input["level"] == "Mid"]

# 'mid_inputs' 데이터에 대해 'lang', 'tweets', 'phd' 속성을 기준으로 엔트로피 계산
```

```
for key in ['lang', 'tweets', 'phd']:
    print(key, partition_entropy_by(mid_inputs, key))
```

```
lang 0.0
tweets 0.0
phd 0.0
```

```
data_entropy(mid_inputs)

# 0.0
```

```
mid_inputs
```

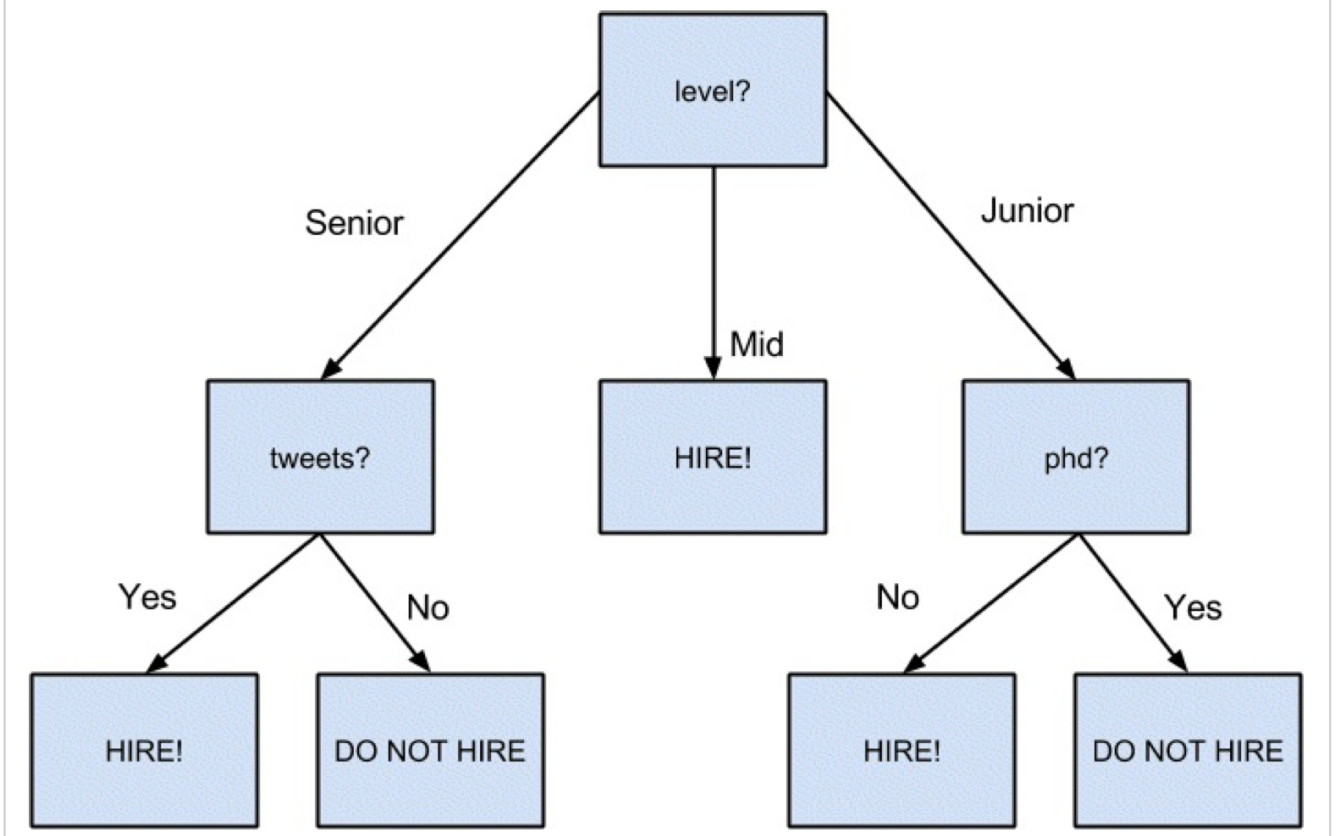
```
[({'level': 'Mid', 'lang': 'Python', 'tweets': 'no', 'phd': 'no'}, True),
 ({'level': 'Mid', 'lang': 'R', 'tweets': 'yes', 'phd': 'yes'}, True),
 ({'level': 'Mid', 'lang': 'Python', 'tweets': 'no', 'phd': 'yes'}, True),
 ({'level': 'Mid', 'lang': 'Java', 'tweets': 'yes', 'phd': 'no'}, True)]
```

```
# 'inputs' 리스트에서 'level'이 'Junior'인 데이터만 필터링하여 'junior_inputs' 리스트 생성
junior_inputs = [(input, label) for input, label in inputs if input["level"] == "Junior"]

# 'junior_inputs' 데이터에 대해 'lang', 'tweets', 'phd' 속성을 기준으로 엔트로피 계산
for key in ['lang', 'tweets', 'phd']:
    print(key, partition_entropy_by(junior_inputs, key))
```

```
lang 0.9509775004326938
tweets 0.9509775004326938
phd 0.0
```

Finally, we have a complete decision tree:



Putting It All Together

```
def classify(tree, input):
    """주어진 결정 트리를 사용하여 입력 데이터를 분류"""

    # 만약 이 노드가 리프 노드(끝 노드)라면, 그 값을 반환
    if tree in [True, False]:
        return tree

    # 그렇지 않으면, 올바른 서브트리를 찾음
    attribute, subtree_dict = tree

    subtree_key = input.get(attribute)  # 만약 입력 데이터에 해당 속성이 없으면 None 반환

    if subtree_key not in subtree_dict: # 만약 해당 키에 대한 서브트리가 없으면,
        subtree_key = None              # None 서브트리를 사용

    subtree = subtree_dict[subtree_key] # 적절한 서브트리를 선택
    return classify(subtree, input)     # 선택한 서브트리를 사용하여 입력 데이터를 분류
```

- 이 `classify` 함수는 주어진 결정 트리를 사용하여 입력 데이터를 분류합니다.
- 먼저, 현재 노드가 리프 노드인지 확인하고, 그렇다면 해당 노드의 값을 반환합니다.
- 리프 노드가 아니라면, 분할에 사용된 속성과 서브트리 사전을 가져옵니다.
- 입력 데이터에서 현재 속성의 값을 찾아 서브트리 사전에 있는지 확인하고, 없으면 `None` 서브트리를 사용합니다.
- 그런 다음 적절한 서브트리를 선택하여 재귀적으로 함수를 호출하여 입력 데이터를 분류합니다.
- 이 과정을 반복하여 최종적으로 입력 데이터의 분류 결과를 얻습니다.

```
def build_tree_id3(inputs, split_candidates=None):
    """
    주어진 데이터셋을 사용하여 ID3 알고리즘을 통해 결정 트리를 생성
    """

    # 첫 번째 호출일 경우, 첫 입력 데이터의 모든 키를 분할 후보로 설정
    if split_candidates is None:
        split_candidates = inputs[0][0].keys()

    # 입력 데이터에서 True와 False의 개수를 세기
    num_inputs = len(inputs)
    num_trues = len([label for item, label in inputs if label])
    num_falses = num_inputs - num_trues

    if num_trues == 0:  # 입력 데이터가 모두 False일 경우
        return False    # "False" 리프 노드를 반환

    if num_falses == 0:  # 입력 데이터가 모두 True일 경우
        return True     # "True" 리프 노드를 반환

    if not split_candidates:  # 더 이상 분할할 후보가 없을 경우
        return num_trues >= num_falses  # 다수결로 리프 노드를 반환

    # 그렇지 않으면, 가장 좋은 속성을 기준으로 분할
    best_attribute = min(split_candidates,
                         key=partial(partition_entropy_by, inputs))
```

```
    # 가장 좋은 속성을 기준으로 데이터를 분할
    partitions = partition_by(inputs, best_attribute)
    new_candidates = [a for a in split_candidates if a != best_attribute]

    # 재귀적으로 서브트리를 생성
    subtrees = {attribute: build_tree_id3(subset, new_candidates)
                for attribute, subset in partitions.items()}

    # 기본 경우를 위한 None 서브트리 설정
    subtrees[None] = num_trues > num_falses

    return (best_attribute, subtrees)
```

- 첫 번째 호출 시, 첫 입력 데이터의 모든 키를 분할 후보로 설정합니다.
- 입력 데이터에서 True와 False의 개수를 셉니다.
- 입력 데이터가 모두 False일 경우 "False" 리프 노드를, 모두 True일 경우 "True" 리프 노드를 반환합니다.
- 더 이상 분할할 후보가 없을 경우, 다수결에 따라 리프 노드를 반환합니다.
- 그렇지 않으면, 가장 낮은 엔트로피를 가진 속성을 기준으로 데이터를 분할합니다.
- 가장 좋은 속성을 기준으로 데이터를 분할하고, 재귀적으로 서브트리를 생성합니다.
- 기본 경우를 위한 None 서브트리를 설정합니다.
- 최종적으로 결정 트리를 반환합니다.

```
print("building the tree")
tree = build_tree_id3(inputs)  # 주어진 데이터를 사용하여 결정 트리를 생성
print(tree)  # 생성된 트리를 출력

# 트리 분류 테스트: Junior / Java / tweets / no phd
print("Junior / Java / tweets / no phd", classify(tree,
    { "level" : "Junior",
      "lang" : "Java",
      "tweets" : "yes",
      "phd" : "no"} ))

# 트리 분류 테스트: Junior / Java / tweets / phd
print("Junior / Java / tweets / phd", classify(tree,
    { "level" : "Junior",
              "lang" : "Java",
              "tweets" : "yes",
              "phd" : "yes"} ))

# 트리 분류 테스트: Intern
print("Intern", classify(tree, { "level" : "Intern" } ))

# 트리 분류 테스트: Senior
print("Senior", classify(tree, { "level" : "Senior" } ))
```

```
building the tree
('level', {'Senior': ('tweets', {'no': False, 'yes': True, None: False}), 'Mid': True, 'Junior'
Junior / Java / tweets / no phd True
Junior / Java / tweets / phd False
Intern True
Senior False
```

- **트리 생성**:
  - `build_tree_id3(inputs)`를 호출하여 주어진 데이터를 사용하여 결정 트리를 생성합니다.
  - 생성된 트리를 `tree` 변수에 저장하고 이를 출력합니다.
- **트리 분류 테스트**:
  - **Junior / Java / tweets / no phd**:
    - 주어진 특성(`level`이 "Junior", `lang`이 "Java", `tweets`가 "yes", `phd`가 "no")을 가진 입력을 분류합니다.
    - 결과를 출력합니다.
  - **Junior / Java / tweets / phd**:
    - 주어진 특성(`level`이 "Junior", `lang`이 "Java", `tweets`가 "yes", `phd`가 "yes")을 가진 입력을 분류합니다.
    - 결과를 출력합니다.
  - **Intern**:
    - 특성(`level`이 "Intern")을 가진 입력을 분류합니다.
    - 결과를 출력합니다.
  - **Senior**:
    - 특성(`level`이 "Senior")을 가진 입력을 분류합니다.
    - 결과를 출력합니다.

위 코드는 ID3의 매우 간단한 구현입니다. 우리는 다음을 고려하지 않았습니다:

- 연속 변수 (Continuous Variable)
- 과적합 (Overfitting)
- 비탐욕적 알고리즘 (Nongreedy Algorithm)

Random Forest (랜덤 포레스트)

- 과적합(Overfitting)을 피하는 한 가지 방법은 "랜덤 포레스트(Random Forests)"라는 기술입니다:
- 앙상블 학습(Ensemble Learning)의 한 유형으로, 여러 약한 학습자(일반적으로 높은 편향, 낮은 분산 모델)를 결합하여 전체적으로 강력한 모델을 생성하는 기법입니다.
- 여러 개의 결정 트리를 만들고
  - 각 트리를 bootstrap_sample(inputs)의 결과로 학습시킵니다.
  - 부트스트랩 새 데이터 세트를 데이터에서 중복 선택으로 n개의 데이터 포인트를 선택하여 생성합니다.
- 입력을 분류하는 방법에 대해 투표하게 합니다.

```
def bootstrap_sample(data):
    return [random.choice(data) for _ in data]
```

- 데이터에서 len(data) 개의 요소를 중복 선택하여 무작위로 샘플링합니다.

```
def forest_classify(trees, input):
    votes = [classify(tree, input) for tree in trees]
    vote_counts = Counter(votes)
    return vote_counts.most_common(1)[0][0]
```

- 여러 결정 트리를 사용하여 입력을 분류하고, 가장 많이 등장한 예측 결과를 반환합니다.

Sklearn implements the following

Decision tree learning

Decision tree learning is a supervised learning approach used in statistics, data mining and machine learning. In this formalism, a classification or regression decision tree is used as a predictive model to draw conclusions about a set of observations.

𝕎 https://www.google.com/url?q=https://en.wikipedia.org/wiki/Decision_tree_learning

```
from sklearn import tree

# 학습 데이터 설정
X = [[0, 0], [1, 1]]  # 특성 데이터
Y = [0, 1]            # 레이블 데이터

# 결정 트리 분류기 생성
clf = tree.DecisionTreeClassifier()

# 모델 학습
clf = clf.fit(X, Y)  # 특성 데이터 X와 레이블 데이터 Y를 사용하여 모델을 학습시킴

# 새로운 데이터에 대한 예측 수행
prediction = clf.predict([[2., 2.]])
```

- 새로운 데이터 포인트 `[[2., 2.]]`에 대한 예측 값이 출력됩니다.
- 예측 값은 `1`로 출력될 가능성이 높습니다. 이는 모델이 학습 데이터에 기반하여 새로운 데이터 포인트를 분류한 결과

Decision Tree(결정 트리) 생성 및 학습

```
from sklearn.datasets import load_iris
from sklearn import tree

# 아이리스 데이터셋 로드
iris = load_iris()

# 결정 트리 분류기 생성
clf = tree.DecisionTreeClassifier()

# 모델 학습
clf = clf.fit(iris.data, iris.target)
```

Decision Tree(결정 트리) 시각화

```
import graphviz

# 결정 트리 시각화를 위한 그래프 데이터 생성
dot_data = tree.export_graphviz(clf, out_file=None)

# 그래프 데이터로부터 시각화 객체 생성
graph = graphviz.Source(dot_data)

# 결정 트리를 'iris' 파일로 저장 및 렌더링
graph.render("iris")

# 'iris.pdf'
```

교차 검증을 통한 모델 평가 (결정 트리)

```
from sklearn.model_selection import cross_val_score

# 교차 검증을 통한 결정 트리 모델 평가
scores = cross_val_score(clf, iris.data, iris.target, cv=10, scoring="accuracy")

# 결과 출력
print("Decision Tree Scores:", scores)
print("Mean:", scores.mean())
print("Standard Deviation:", scores.std())
```

```
Scores: [1.         0.93333333 1.         0.93333333 0.93333333 0.86666667
 0.93333333 0.93333333 1.         1.        ]
Mean: 0.9533333333333334
Standard Deviation: 0.042687494916218 98
```

랜덤 포레스트 생성 및 학습

```
from sklearn.ensemble import RandomForestClassifier

# 랜덤 포레스트 분류기 생성
rf = RandomForestClassifier(n_estimators=100)  # 트리의 개수 설정

# 교차 검증을 통한 랜덤 포레스트 모델 평가
scores = cross_val_score(rf, iris.data, iris.target, cv=10, scoring="accuracy")

# 결과 출력
print("Random Forest Scores:", scores)
print("Mean:", scores.mean())
print("Standard Deviation:", scores.std())
```

```
Scores: [1.         0.93333333 1.         0.93333333 0.93333333 0.93333333
 0.86666667 1.         1.         1.        ]
Mean: 0.96
Standard Deviation: 0.044221663871405324
```