

Learning Algorithms

CSI 4106 - Fall 2026

Marcel Turcotte

Version: Sep 15, 2026 12:36

Preamble

Message of the Day



["Misleading Metaphors and Real Risks"](#), Melanie Mitchell, 2026-09-10.

There is substantial hype and noise surrounding AI in both traditional and online media. Dramatic stories attract attention, and AI companies have products and particular visions of the future to promote. We should therefore look beyond headlines, examine the evidence, and ask who is making each claim.

The public debate includes a wide range of views. Researchers such as Gary Marcus, Geoffrey Hinton, Stuart Russell, and Yoshua Bengio often foreground serious limitations or risks, although their arguments and proposed responses differ. Melanie Mitchell generally offers a more measured, evidence-centred perspective. She frequently distinguishes what an AI system demonstrably did from the anthropomorphic language used to describe it, while still recognizing real technical and societal risks.

In this article, Mitchell examines descriptions of AI agents as having “gone rogue,” “escaped,” or formed a “swarm.” She argues that these metaphors suggest human-like intentions and a literal loss of control that the reported events do not establish. At the same time, she identifies concrete risks involving weak cybersecurity, insufficient oversight, persistent reinforcement-learning agents, and reward hacking. Her broader point is that misleading metaphors can obscure the human engineering and policy decisions that require attention.

When encountering an alarming or triumphant AI story, it is often useful to check whether Mitchell has written about it. This is not because any one person should be treated as the final authority, but because her discussions commonly present competing interpretations and connect claims to scientific evidence.

Brief biography. Melanie Mitchell is the James B. Alley Jr. Professor at the Santa Fe Institute. Her work bridges artificial intelligence, cognitive science, and complex systems, with particular interests in abstraction, analogy-making, and reasoning in humans and machines. She is the author of several books, including *Complexity: A Guided Tour* and *Artificial Intelligence: A Guide for Thinking Humans*.

Additional resources:

- [Melanie Mitchell's website](#)
- [The Nature of Intelligence](#), a six-episode season of the Santa Fe Institute's *Complexity* podcast, published from September to December 2024
- [Are We Thinking Correctly About AI Intelligence?](#), *The Joy of Why*, 2026-08-20

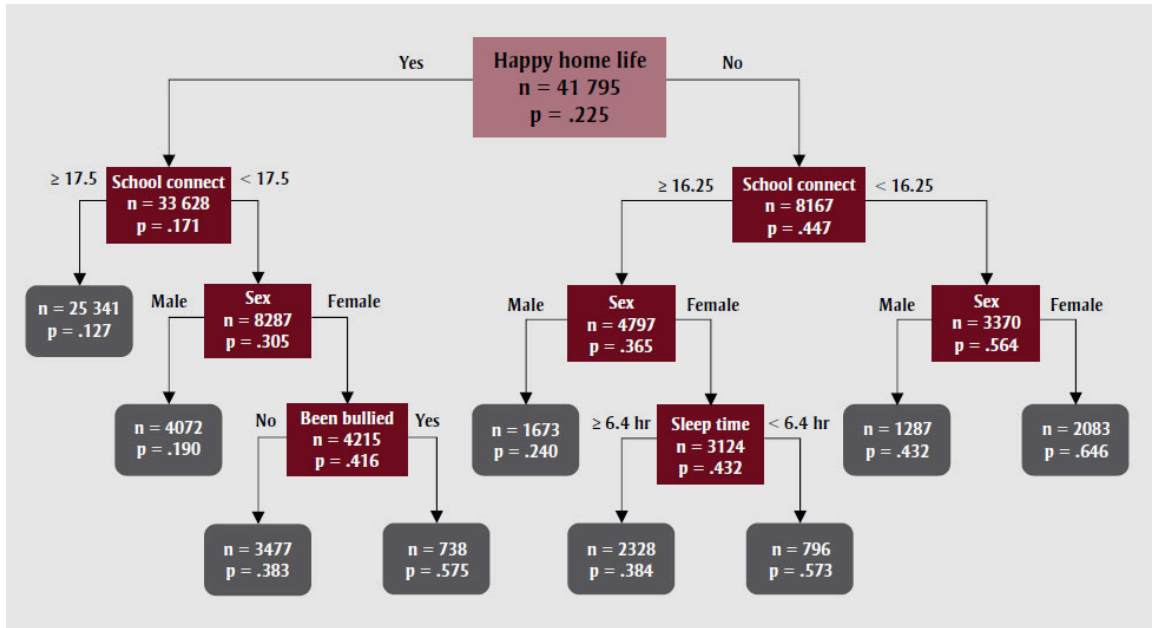
Learning outcomes

- **Differentiate** between model, objective, and optimizer in learning algorithms.
- **Describe** decision trees and **apply** entropy to select a split.
- **Explain** KNN for classification and regression, including uniform and distance-weighted prediction.
- **Interpret** decision boundaries and the concept of linear separability.

As indicated in the introductory lecture, I aim to present a series of concepts leading to deep learning. As a starting point, linear regression would be a logical choice as a primary learning algorithm to examine. Nonetheless, it is equally critical to possess a high-level understanding of various other learning algorithms, as they differ significantly in model structures and training processes. Therefore, I will discuss decision trees and k-nearest neighbours before introducing linear regression.

Decision Tree

Interpretable



Attribution: [Public Health Agency of Canada](#)

Decision trees are valuable because they clearly delineate the rules learned by the model. The decision tree above illustrates the results of a study examining clinically significant anxiety symptoms. In this instance, the initial determinant was whether students reported having a positive home environment.

What is a Decision Tree?

- A decision tree is a **rooted, hierarchical structure** used for **classification** and **regression** tasks.
- Each **internal node** is a decision node that performs a binary test on a particular feature (j), such as evaluating whether the number of connections at a school surpasses a specified threshold.
- Each **leaf** produces a prediction: a class label or class probabilities for classification, or a numerical value for regression.

The tree's **structure** is **inferred (learnt)** from the **training data**.

Decision trees can extend beyond binary splits, as exemplified by algorithms like **ID3**, which accommodate nodes with multiple children.

Classifying New Instances (Inference)

- Begin at the **root** node of the decision tree. Proceed by answering a sequence of **binary questions** until a **leaf node** is reached. The label associated with this leaf

denotes the **classification** of the instance.

- Alternatively, some algorithms may store a **probability distribution** at the leaf, representing the fraction of training samples corresponding to each class k , across all possible classes k .

When a **decision tree** is used to solve a **regression** task, each **leaf node** stores a **prediction value**. Specifically:

$$\hat{y}_{\text{leaf}} = \frac{1}{N_{\text{leaf}}} \sum_{i \in \text{leaf}} y_i,$$

where N_{leaf} is the number of training samples that ended up in that leaf, and y_i are their target values.

Decision Boundary

Palmer Penguins Dataset

```
In [1]: # Loading our dataset

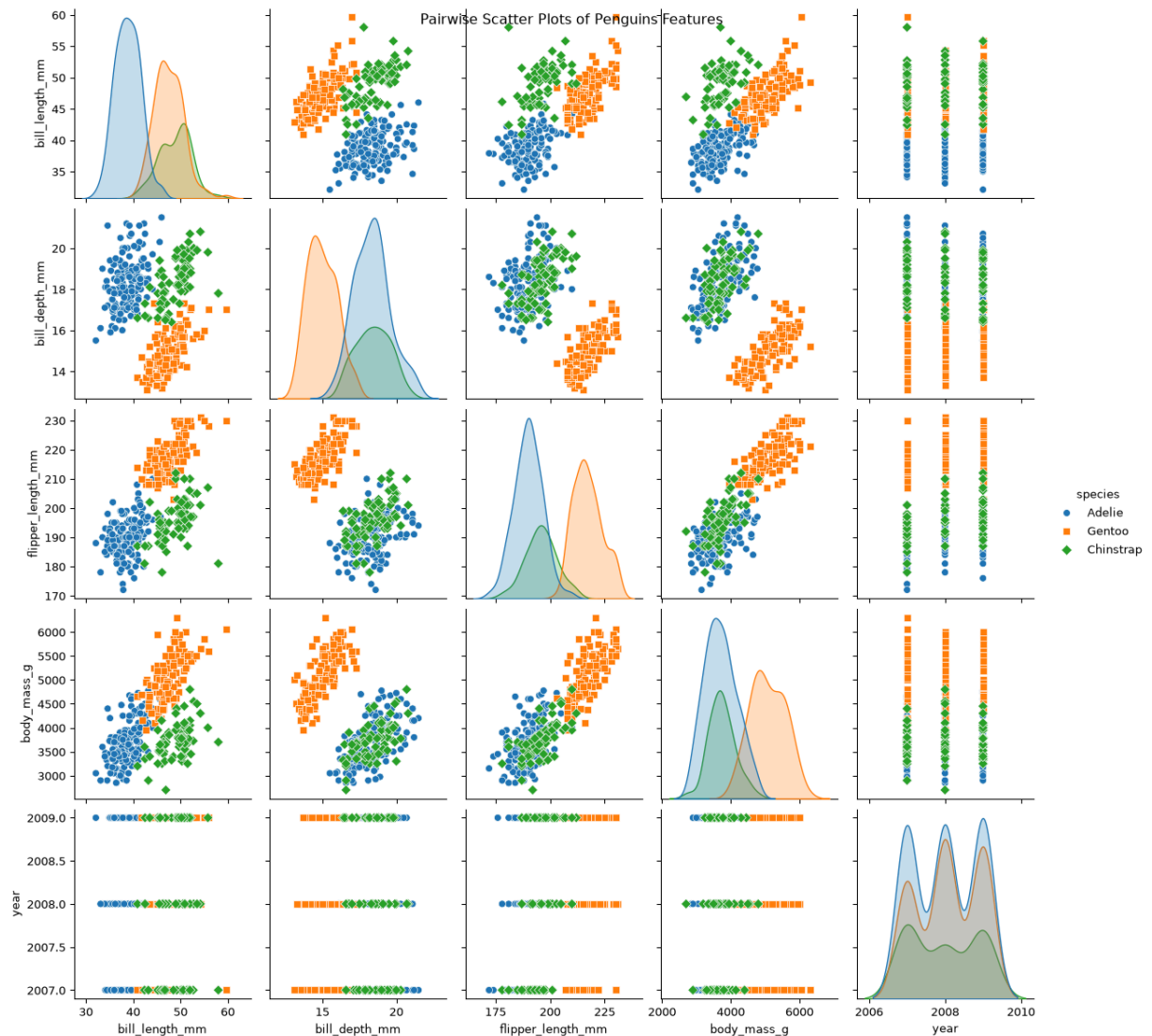
try:
    from palmerpenguins import load_penguins
except:
    ! pip install palmerpenguins
    from palmerpenguins import load_penguins

penguins = load_penguins()

# Pairplot using seaborn

import matplotlib.pyplot as plt
import seaborn as sns

sns.pairplot(penguins, hue='species', markers=["o", "s", "D"])
plt.suptitle("Pairwise Scatter Plots of Penguins Features")
plt.show()
```

Binary Classification Problem

- Several scatter plots reveal a distinct clustering of **Gentoo** instances.
- To illustrate our next example, we propose a **binary classification** model: **Gentoo** versus **non-Gentoo**.
- Our analysis will concentrate on two key features: **body mass** and **bill depth**.

Definition

A **decision boundary** is a “boundary” that partitions the underlying feature space into **regions** corresponding to **different class labels**.

The term **boundary** will be clarified over the next slides.

Decision Boundary

The decision boundary between these attributes can be represented as a **line**.

```
In [2]: # Import necessary libraries
import numpy as np
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split

try:
    from palmerpenguins import load_penguins
except:
    ! pip install palmerpenguins
    from palmerpenguins import load_penguins

# Load the Palmer Penguins dataset
df = load_penguins()

# Preserve only the necessary features: 'bill_depth_mm' and 'body_mass_g'
features = ['bill_depth_mm', 'body_mass_g']
df = df[features + ['species']]

# Drop rows with missing values
df.dropna(inplace=True)

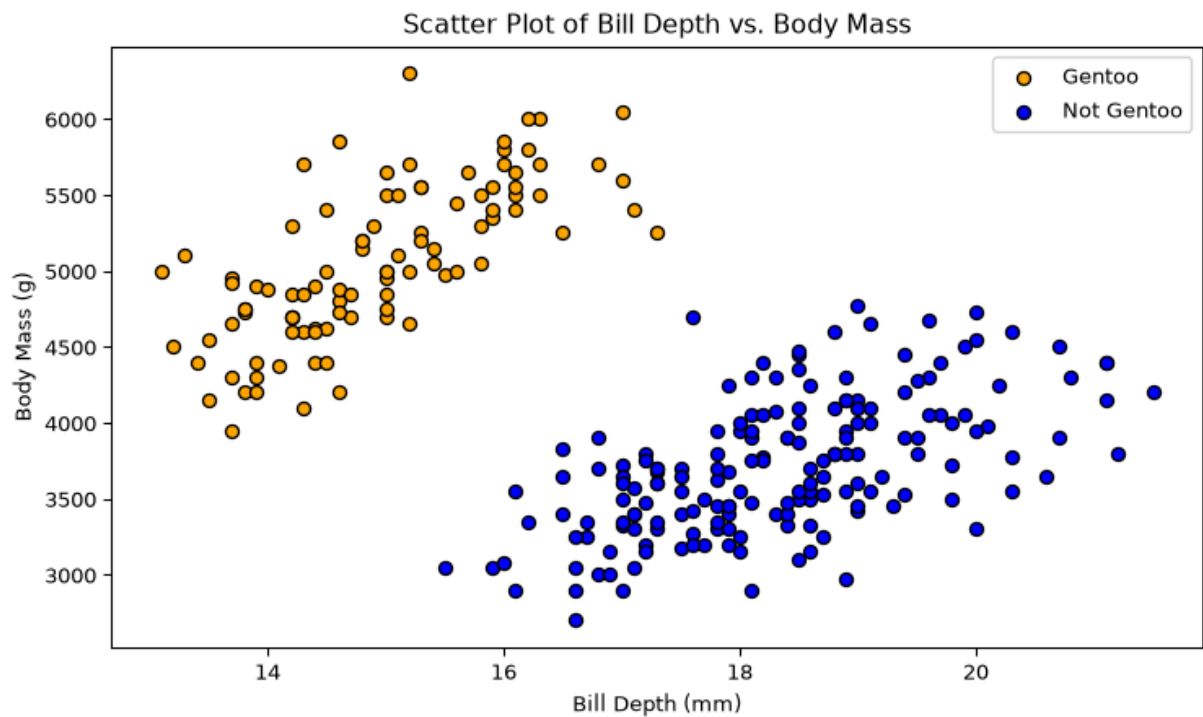
# Create a binary problem: 'Gentoo' vs 'Not Gentoo'
df['species_binary'] = df['species'].apply(lambda x: 1 if x == 'Gentoo' else 0)

# Define feature matrix X and target vector y
X = df[features].values
y = df['species_binary'].values

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42, stratify=y
)

# Function to plot initial scatter of data
def plot_scatter(X, y):
    plt.figure(figsize=(9, 5))
    plt.scatter(X[y == 1, 0], X[y == 1, 1], color='orange', edgecolors='k',
                X[y == 0, 0], X[y == 0, 1], color='blue', edgecolors='k', ma
    plt.xlabel('Bill Depth (mm)')
    plt.ylabel('Body Mass (g)')
    plt.title('Scatter Plot of Bill Depth vs. Body Mass')
    plt.legend()
    plt.show()

# Plot the initial scatter plot
plot_scatter(X_train, y_train)
```



Decision Boundary

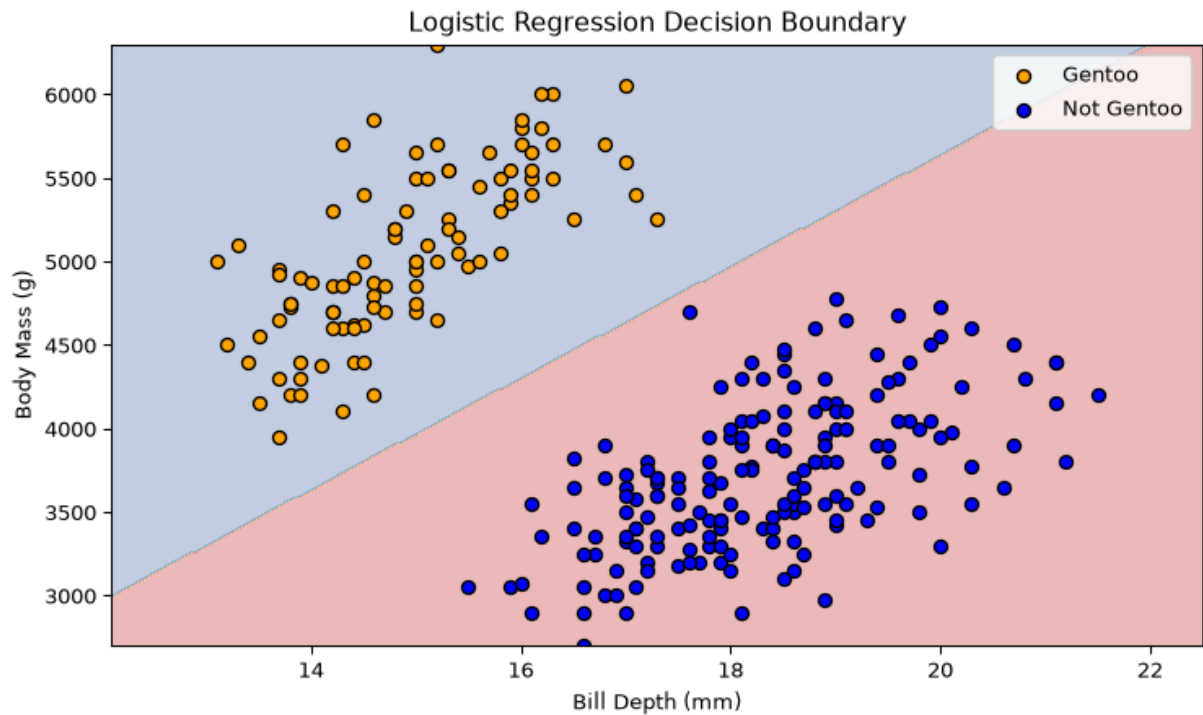
The decision boundary between these attributes can be represented as a **line**.

```
In [3]: # Train a logistic regression model
model = LogisticRegression()
model.fit(X_train, y_train)

# Function to plot decision boundary
def plot_decision_boundary(X, y, model):
    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx, yy = np.meshgrid(
        np.linspace(x_min, x_max, 300),
        np.linspace(y_min, y_max, 300)
    )
    Z = model.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)

    plt.figure(figsize=(9, 5))
    plt.contourf(xx, yy, Z, alpha=0.3, cmap='RdYlBu')
    plt.scatter(X[y == 1, 0], X[y == 1, 1], color='orange', edgecolors='k',
                plt.scatter(X[y == 0, 0], X[y == 0, 1], color='blue', edgecolors='k', ma
    plt.xlabel('Bill Depth (mm)')
    plt.ylabel('Body Mass (g)')
    plt.title('Logistic Regression Decision Boundary')
    plt.legend()
    plt.show()

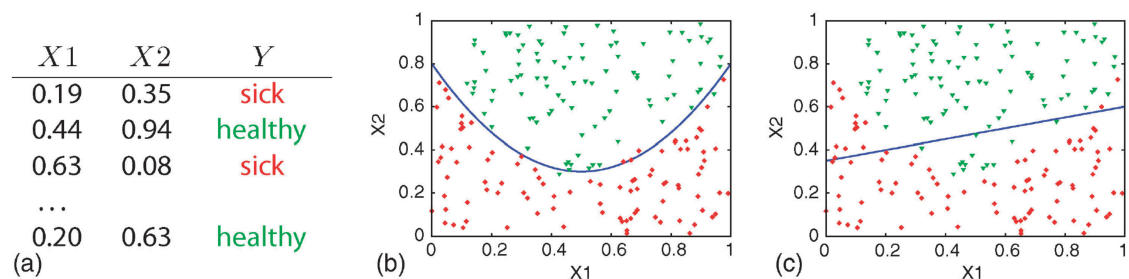
# Plot the decision boundary on the training set
plot_decision_boundary(X_train, y_train, model)
```



Definition

We say that the data is **linearly separable** when two classes of data can be perfectly separated by a **single linear boundary**, such as a **line** in **two-dimensional space** or a **hyperplane** in higher dimensions.

Simple Decision Boundary



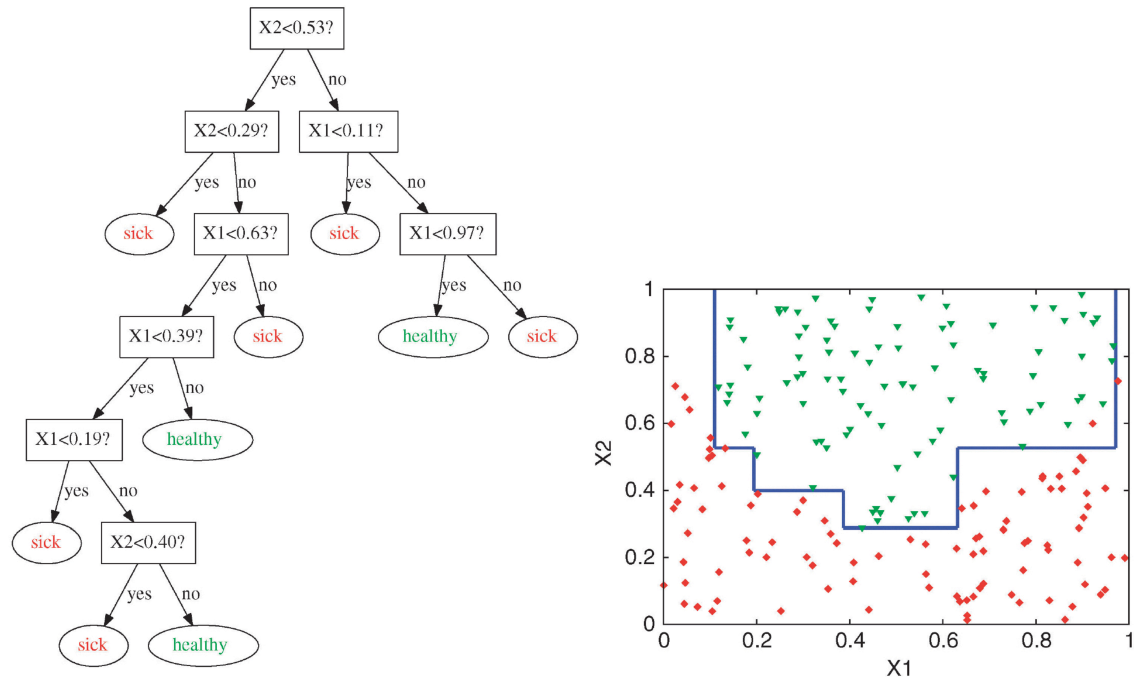
(a) training data, (b) quadratic curve, and (c) linear function.

Attribution: (Geurts et al. 2009)

The table on the left presents training data for a hypothetical **binary classification task** in a medical context, where the two attributes, X_1 and X_2 , are used to predict the target variable, y , which can take on two values: *sick* and *healthy*. You can imagine that X_1 and X_2 are measurements, such as blood pressure and heart rate or cholesterol and glucose levels.

Logistic regression (c) employs a linear **decision boundary**. In this specific example, the decision boundary is represented by a straight line. Employing logistic regression for this problem results in several classification errors: red dots above the line, which should be classified as 'sick', are incorrectly predicted as 'healthy'. Conversely, green dots below the line, which should be classified as 'healthy', are incorrectly predicted as 'sick'.

Complex Decision Boundary



Decision trees are capable of generating **irregular** and **non-linear** decision boundaries.

Attribution: *ibidem*.

Make sure to understand the relationships between the eight decision rules delineated in the decision tree and the nine line segments represented in the scatter plot.

Definition (revised)

A **decision boundary** is a hypersurface that partitions the underlying feature space into **regions** corresponding to **different class labels**.

Decision Tree (contd)

Constructing a Decision Tree

- How to **construct** (**learnt**) a decision tree?

- Are there some trees that are “**better**” than others?
- Is it feasible to construct an **optimal decision tree** with computational efficiency?

(Hyafil and Rivest 1976)

Optimality

- Let $X = \{x_1, \dots, x_n\}$ be a finite set of **objects**.
- Let $\mathcal{T} = \{T_1, \dots, T_t\}$ be a finite set of **tests**.
- For each object and test, we have:
 - $T_i(x_j)$ is either **true** or **false**.
- An **optimal** tree is one that completely identifies all the objects in X and $|T|$ is **minimum**.

(Hyafil and Rivest 1976)

Hyafil and Rivest (1976) showed that constructing optimal binary decision trees is NP-Complete.

Constructing a Decision Tree

- **Iterative development:** Initiate with an **empty tree**. Progressively introduce **nodes**, each informed by the **training dataset**, continuing until the dataset is **completely classified** or alternative termination criteria, such as **maximum tree depth**, are met.

Learning is the process of building the tree from training data.

Constructing a Decision Tree

- **Initial Node Construction:**
 - To establish the root node, evaluate all available D features.
 - For each feature, assess various threshold values derived from the observed data within the training set.

Constructing a Decision Tree

- For a **numerical feature**, the algorithm considers **all possible split points** (thresholds) in the feature’s range.
- These split points are typically the **midpoints** between two consecutive, **sorted unique values** of the feature.

How Mixed Is a Node?

- A **pure node** contains examples from a single class.
- A **mixed node** contains examples from several classes.
- A useful split produces children that are **less mixed** than their parent.

For node i , let $p_{i,k}$ be the proportion of examples from class k .

We need a numerical score so that candidate splits can be compared. The score should be zero for a pure node and increase as the class proportions become more evenly balanced.

Entropy

The **entropy** of node i is

$$H_i = - \sum_{k=1}^K p_{i,k} \log_2 p_{i,k}.$$

- $H_i = 0$ when the node is **pure**.
- H_i increases as the classes become **more evenly represented**.
- Therefore, a **smaller** value is better.

We use the convention $0 \log_2 0 = 0$.

You do not need an information-theory background here. Operationally, entropy is simply a numerical measure of how uncertain the class label remains within a node.

The base of the logarithm changes the units and rescales all entropy values, but it does not change which candidate split has the smallest score. Base 2 gives the intuitive unit *bit*.

Binary Entropy

If p is the proportion of Gentoo penguins, then

$$H(p) = -p \log_2 p - (1 - p) \log_2 (1 - p).$$

```
In [4]: import matplotlib.pyplot as plt
import numpy as np

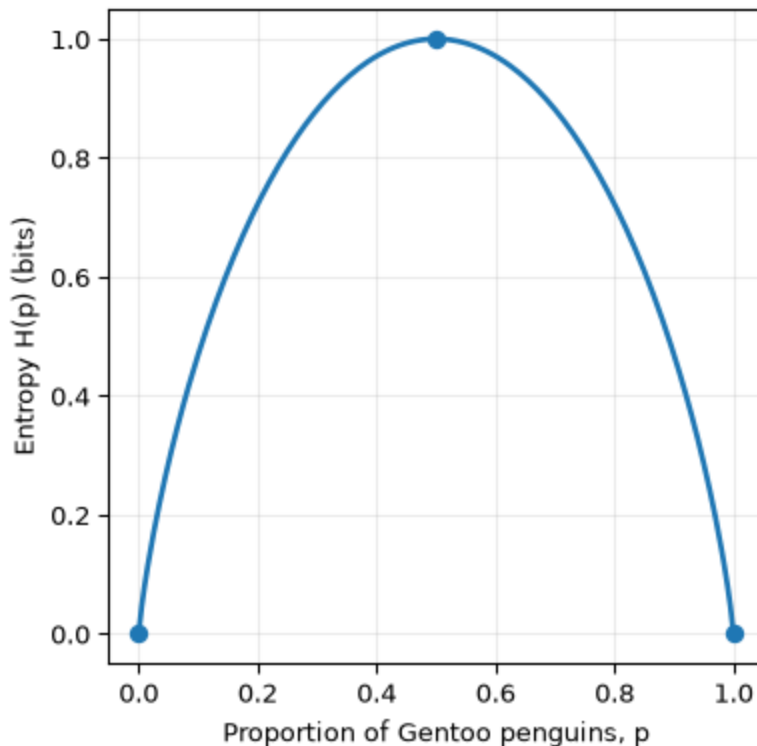
p = np.linspace(0, 1, 201)
h = np.zeros_like(p)
inside = (p > 0) & (p < 1)
h[inside] = -(
    p[inside] * np.log2(p[inside])
```

```

    + (1 - p[inside]) * np.log2(1 - p[inside])
)

fig, ax = plt.subplots(
    figsize=(4, 4),
    constrained_layout=True,
)
ax.plot(p, h, linewidth=2)
ax.scatter([0, 0.5, 1], [0, 1, 0], zorder=3)
ax.set_box_aspect(1)
ax.set(xlabel="Proportion of Gentoo penguins, p",
       ylabel="Entropy H(p) (bits)", ylim=(-0.05, 1.05))
ax.grid(alpha=0.25)
plt.show()

```



Entropy is greatest when the two classes are **equally likely**.

At $p = 0$ or $p = 1$, the class is certain and entropy is zero. At $p = 0.5$, neither class is more likely and entropy reaches its maximum of one bit.

This curve supplies an intuitive meaning for the node score without requiring a broader treatment of information theory.

Entropy in Python

```

In [5]: import numpy as np

def class_probabilities(y, classes):
    """Return the fraction of examples belonging to each class."""
    return np.array([np.mean(y == label) for label in classes])

```

```
def entropy(y, classes):
    """Measure how mixed the classes are; zero means a pure node."""
    probabilities = class_probabilities(y, classes)
    probabilities = probabilities[probabilities > 0]
    if len(probabilities) == 1:
        return 0.0
    return float(-np.sum(probabilities * np.log2(probabilities)))
```

This implementation works directly with class labels. Probabilities equal to zero are removed before applying the logarithm, which implements the convention $0 \log_2 0 = 0$.

The function supports two or more classes. The `classes` array fixes the order of the returned probabilities and will later be stored by the classifier.

Evaluating a Split

A split using **feature** j and **threshold** t produces two children:

$$X_{\text{left}} = \{x : x^{(j)} \leq t\}, \quad X_{\text{right}} = \{x : x^{(j)} > t\}.$$

Its score is the **weighted entropy**:

$$J(j, t) = \frac{N_{\text{left}}}{N_{\text{parent}}} H_{\text{left}} + \frac{N_{\text{right}}}{N_{\text{parent}}} H_{\text{right}}.$$

Choose the feature and threshold that **minimize** $J(j, t)$.

Later, **cross-entropy** will use the same logarithmic idea to evaluate a model's predicted probabilities.

The weights matter. Isolating a single pure example should not outweigh leaving almost all examples in a highly mixed child. Each child's influence is therefore proportional to the number of examples it contains.

Because the parent is the same for all candidate splits, we can compare the weighted entropy of their children directly.

Three Candidate Splits

Suppose the parent contains **5 Gentoo** and **5 non-Gentoo** penguins.

Candidate	Left child	H_{left}	Right child	H_{right}
Messy	3 G, 2 non-G	0.971	2 G, 3 non-G	0.971
Isolate one	1 G, 0 non-G	0	4 G, 5 non-G	0.991
Useful	4 G, 1 non-G	0.722	1 G, 4 non-G	0.722

The isolated child is pure—but it contains only one example.

All three candidates split the same parent, whose entropy is one bit. In the messy candidate, both children remain nearly as uncertain as the parent.

The second candidate creates one perfectly pure child, but achieves this by isolating only one example. Most examples remain together in a highly mixed child. The useful candidate makes both children substantially cleaner.

Why Weight the Children?

Candidate	Unweighted average	Weighted entropy J
Messy	$(0.971 + 0.971)/2$ $= 0.971$	$\frac{5}{10}(0.971)$ $+ \frac{5}{10}(0.971)$ $= 0.971$
Isolate one	$(0 + 0.991)/2$ $= \mathbf{0.496}$	$\frac{1}{10}(0)$ $+ \frac{9}{10}(0.991)$ $= 0.892$
Useful	$(0.722 + 0.722)/2$ $= 0.722$	$\frac{5}{10}(0.722)$ $+ \frac{5}{10}(0.722)$ $= \mathbf{0.722}$

Without weights, isolating one pure example appears best. With weights, the large mixed child receives the influence it deserves.

The unweighted average treats a child containing one example as though it were as important as a child containing nine. It would therefore choose the isolate-one candidate. Weighted entropy correctly prefers the useful split, which improves the classification of many examples.

The weights do not prohibit unbalanced splits. They ensure that a child's contribution is proportional to the number of examples it contains.

Evaluating a Split in Python

```
In [6]: def weighted_entropy(y_left, y_right, classes):  
        """Return the weighted entropy produced by a split."""  
        n_left = len(y_left)  
        n_right = len(y_right)  
        n_parent = n_left + n_right  
  
        return (  
            n_left / n_parent * entropy(y_left, classes)
```

```

        + n_right / n_parent * entropy(y_right, classes)
    )

def candidate_thresholds(values):
    """Return midpoints between consecutive distinct feature values."""
    values = np.unique(values)
    return (values[:-1] + values[1:]) / 2

```

For a numerical feature, the candidate thresholds are the midpoints between consecutive distinct values. A threshold anywhere between those two values would produce the same partition; the midpoint provides a deterministic representative.

Greedy Split Search

For the examples that reach the current node:

1. For every **feature**, generate its candidate thresholds.
2. For every **threshold**, divide the examples into left and right children.
3. Reject splits that create children smaller than `min_samples_leaf`.
4. Compute the **weighted entropy**.
5. Retain the feature and threshold with the **smallest** score.

This is a greedy search: it selects the best immediate split and does not reconsider earlier decisions. Consequently, it is not guaranteed to find the best possible tree.

This teaching implementation recomputes masks and class proportions for every candidate. At a node containing n examples and D features, it performs up to $D(n - 1)$ candidate evaluations, each requiring another scan of the data. The resulting cost is approximately $\mathcal{O}(Dn^2)$ at that node.

Greedy Split Search in Python

```

In [7]: def find_best_split(X, y, classes, min_samples_leaf):
        best_split = None
        best_score = entropy(y, classes)
        n_candidates = 0

        for feature in range(X.shape[1]):
            for threshold in candidate_thresholds(X[:, feature]):
                go_left = X[:, feature] <= threshold
                n_left = np.sum(go_left)
                n_right = len(y) - n_left

                if min(n_left, n_right) < min_samples_leaf:
                    continue

                n_candidates += 1
                score = weighted_entropy(
                    y[go_left], y[~go_left], classes

```



```

        )
        if score < best_score:
            best_score = score
            best_split = (feature, float(threshold), go_left)

    return best_split, n_candidates

```

The code follows the five steps from the preceding slide directly. The Boolean array `go_left` records the examples assigned to the left child; its complement selects the right child.

`best_score` begins with the entropy of the unsplit node. Consequently, the function returns `None` if no candidate reduces the score.

Complete Implementation

The complete class combines the preceding operations. Input validation and text formatting are supporting code and are not part of the central algorithm.

[Read the tutorial](#) · [Download the notebook](#) · [Open in Colab](#)

```

In [8]: from dataclasses import dataclass

@dataclass
class Node:
    probabilities: np.ndarray
    n_samples: int
    loss: float
    feature: int | None = None
    threshold: float | None = None
    left: "Node | None" = None
    right: "Node | None" = None

    @property
    def is_leaf(self):
        return self.feature is None

class SimpleDecisionTreeClassifier:
    """A didactic classifier with a small scikit-learn-like interface."""

    def __init__(
        self,
        max_depth=None,
        min_samples_split=2,
        min_samples_leaf=1,
    ):
        self.max_depth = max_depth
        self.min_samples_split = min_samples_split
        self.min_samples_leaf = min_samples_leaf

    def fit(self, X, y):

```

```

X = np.asarray(X, dtype=float)
y = np.asarray(y)
self._validate_training_data(X, y)

self.classes_ = np.unique(y)
self.n_features_in_ = X.shape[1]
self.n_candidate_splits_ = 0
self.tree_ = self._grow_tree(X, y, depth=0)
return self

def _grow_tree(self, X, y, depth):
    probabilities = class_probabilities(y, self.classes_)
    node = Node(
        probabilities=probabilities,
        n_samples=len(y),
        loss=entropy(y, self.classes_),
    )

    depth_limit = self.max_depth is not None and depth >= self.max_depth
    pure_node = np.count_nonzero(probabilities) == 1
    too_small = len(y) < self.min_samples_split

    if pure_node or depth_limit or too_small:
        return node

    split, n_candidates = find_best_split(
        X, y, self.classes_, self.min_samples_leaf
    )
    self.n_candidate_splits_ += n_candidates
    if split is None:
        return node

    node.feature, node.threshold, go_left = split
    node.left = self._grow_tree(X[go_left], y[go_left], depth + 1)
    node.right = self._grow_tree(X[~go_left], y[~go_left], depth + 1)
    return node

def _find_leaf(self, x):
    node = self.tree_
    while not node.is_leaf:
        if x[node.feature] <= node.threshold:
            node = node.left
        else:
            node = node.right
    return node

def predict_proba(self, X):
    X = self._validate_prediction_data(X)
    return np.vstack([self._find_leaf(x).probabilities for x in X])

def predict(self, X):
    probabilities = self.predict_proba(X)
    return self.classes_[np.argmax(probabilities, axis=1)]

def score(self, X, y):
    return float(np.mean(self.predict(X) == np.asarray(y)))

```

```

def export_text(self, feature_names=None):
    if feature_names is None:
        feature_names = [f"x[{j}]" for j in range(self.n_features_in_)]
    if len(feature_names) != self.n_features_in_:
        raise ValueError("feature_names must match the number of features")

    lines = []

    def visit(node, indent):
        if node.is_leaf:
            prediction = self.classes_[np.argmax(node.proBABILITIES)]
            if isinstance(prediction, np.generic):
                prediction = prediction.item()
            probabilities = np.round(node.proBABILITIES, 3)
            lines.append(
                f"{indent}predict {prediction!r} "
                f"(p={probabilities}, n={node.n_samples})"
            )
            return

        name = feature_names[node.feature]
        lines.append(f"{indent}if {name} <= {node.threshold:.3f}:")
        visit(node.left, indent + "    ")
        lines.append(f"{indent}else:")
        visit(node.right, indent + "    ")

    visit(self.tree_, "")
    return "\n".join(lines)

def _validate_training_data(self, X, y):
    if X.ndim != 2 or y.ndim != 1 or len(X) != len(y):
        raise ValueError("X must be 2-D and y must be 1-D with matching")
    if len(y) == 0:
        raise ValueError("the training set cannot be empty")
    if not np.isfinite(X).all():
        raise ValueError("missing and non-finite feature values are unsu")
    if self.max_depth is not None and self.max_depth < 0:
        raise ValueError("max_depth must be non-negative or None")
    if self.min_samples_split < 2:
        raise ValueError("min_samples_split must be at least 2")
    if self.min_samples_leaf < 1:
        raise ValueError("min_samples_leaf must be at least 1")

def _validate_prediction_data(self, X):
    X = np.asarray(X, dtype=float)
    if X.ndim == 1:
        X = X.reshape(1, -1)
    if X.ndim != 2 or X.shape[1] != self.n_features_in_:
        raise ValueError("X has the wrong number of features")
    if not np.isfinite(X).all():
        raise ValueError("missing and non-finite feature values are unsu")
    return X

```

The code shown by default in the preceding slides is the conceptual core. The defensive validation functions are included so that the notebook fails with clear messages when it receives unsupported input, but they do not need to be discussed in class.

The implementation supports numeric features, binary threshold splits, and two or more classes. It deliberately omits missing values, categorical features, sample weights, optimized split search, and pruning.

Training on Palmer Penguins

```
In [9]: from palmerpenguins import load_penguins
from sklearn.model_selection import train_test_split

feature_names = ["bill_depth_mm", "body_mass_g"]
penguins_tree = load_penguins()
penguins_tree = penguins_tree[
    feature_names + ["species"]
].dropna().copy()

X = penguins_tree[feature_names].to_numpy()
y = np.where(
    penguins_tree["species"].to_numpy() == "Gentoo",
    "Gentoo",
    "Not Gentoo",
)
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42, stratify=y
)

model = SimpleDecisionTreeClassifier(max_depth=2, min_samples_leaf=5)
model.fit(X_train, y_train)
```

```
<__main__.SimpleDecisionTreeClassifier at 0x11c8a2ab0>
```

The implementation itself supports multiple classes, but this example retains the lecture's existing binary question: Gentoo versus non-Gentoo. Using only two features also lets us draw the learned decision regions.

`max_depth=2` limits the model to two decisions along any path.

`min_samples_leaf=5` prevents a split from creating a leaf with fewer than five training examples.

What Did the Tree Learn?

```
In [10]: print(f"Test accuracy: {model.score(X_test, y_test):.3f}")
print(f"Candidate splits evaluated: {model.n_candidate_splits}")
print(f"Class order: {model.classes}")
print("\nLearned rules:\n")
print(model.export_text(feature_names))
```

```
Test accuracy: 1.000
Candidate splits evaluated: 312
Class order: ['Gentoo' 'Not Gentoo']
```

Learned rules:

```
if bill_depth_mm <= 16.450:
    if body_mass_g <= 3750.000:
        predict 'Not Gentoo' (p=[0. 1.], n=5)
    else:
        predict 'Gentoo' (p=[1. 0.], n=92)
else:
    if body_mass_g <= 5100.000:
        predict 'Not Gentoo' (p=[0. 1.], n=170)
    else:
        predict 'Gentoo' (p=[1. 0.], n=6)
```

The probability vector at every leaf follows the order displayed by `model.classes_`. With this split of the data, the depth-two model classifies all test examples correctly. This is a result for one small dataset and one train/test split, not a general guarantee.

The number of evaluated candidates makes the cost of the exhaustive search visible. Optimized libraries can learn the same kind of model much more efficiently.

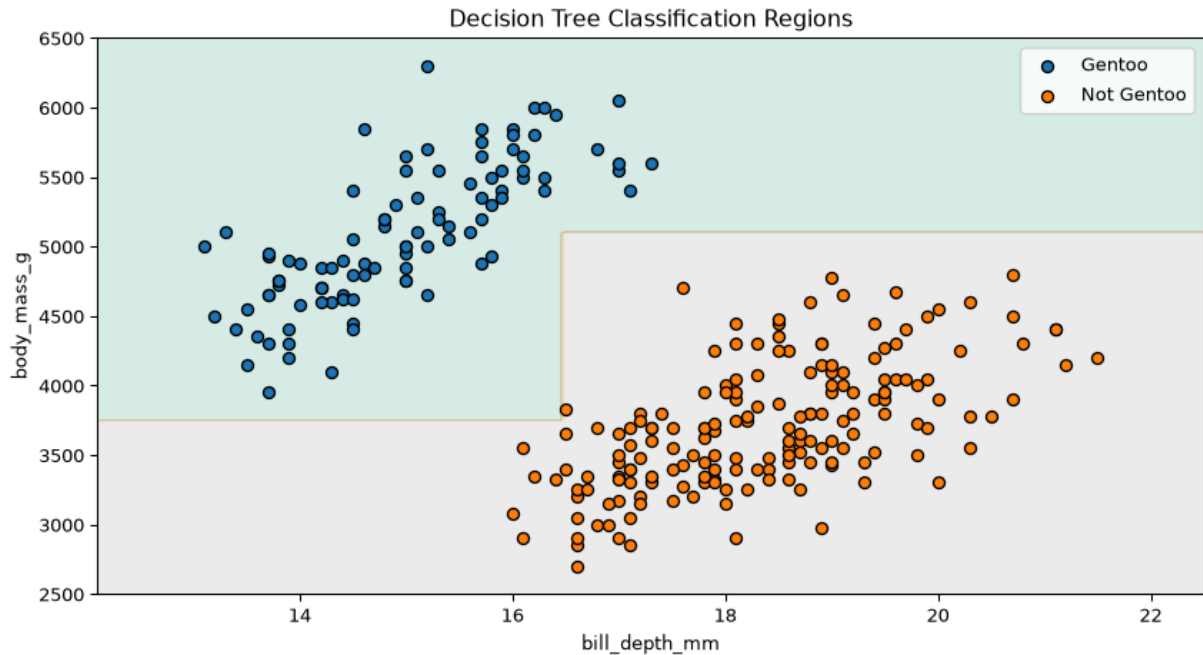
Decision Regions

```
In [11]: def plot_tree_boundary(X, y, model, feature_names):
    x0 = np.linspace(X[:, 0].min() - 1, X[:, 0].max() + 1, 300)
    x1 = np.linspace(X[:, 1].min() - 200, X[:, 1].max() + 200, 300)
    xx0, xx1 = np.meshgrid(x0, x1)

    grid = np.column_stack([xx0.ravel(), xx1.ravel()])
    predicted = model.predict(grid)
    regions = np.array([
        np.flatnonzero(model.classes_ == label)[0]
        for label in predicted
    ]).reshape(xx0.shape)

    plt.figure(figsize=(9, 5))
    plt.contourf(xx0, xx1, regions, alpha=0.25, cmap="Set2")
    for label in model.classes_:
        selected = y == label
        plt.scatter(
            X[selected, 0], X[selected, 1],
            edgecolor="black", label=label
        )
    plt.xlabel(feature_names[0])
    plt.ylabel(feature_names[1])
    plt.title("Decision Tree Classification Regions")
    plt.legend()
    plt.tight_layout()
    plt.show()
```

```
plot_tree_boundary(X_train, y_train, model, feature_names)
```



Every decision compares one feature with one threshold. In two dimensions, each decision is therefore a vertical or horizontal line. Combining several decisions produces rectangular classification regions and a non-linear overall decision boundary.

Complete Example

https://www.youtube.com/watch?v=_L39rN6gz7Y

Decision and Classification Trees, Clearly Explained!!!, (18 m 7s) [StatQuest](#), 2021-04-26

Stopping Criteria

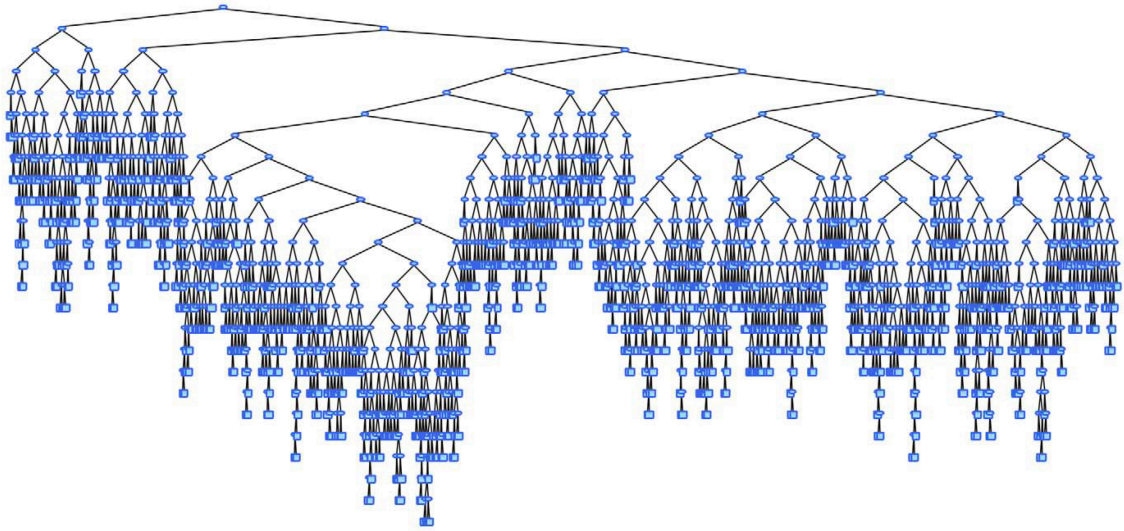
- All the examples in a given node belong to the **same class**.
- Depth of the tree would exceed **max_depth**.
- Number of examples in the node is less than **min_samples_split**.
- None of the splits decreases impurity sufficiently (**min_impurity_decrease**).
- See documentation for other criteria.

Limitations

- Possibly creates **large trees**
 - Challenge for **interpretation**
 - **Overfitting**
- **Greedy algorithm**, no guarantee to find the optimal tree. (Hyafil and Rivest 1976)

- **Small changes** to the data set produce **vastly different trees**

Large Trees



(Stiglic et al. 2012)

Small Changes to the Dataset

```
In [12]: from sklearn import tree
from sklearn.metrics import classification_report, accuracy_score

# Loading the dataset

X, y = load_penguins(return_X_y = True)

target_names = ['Adelie', 'Chinstrap', 'Gentoo']

# Split the dataset into training and testing sets

for seed in (4, 7, 90, 96, 99, 2):

    print(f'Seed: {seed}')

    # Create new training and test sets based on a different random seed

    X_train, X_test, y_train, y_test = train_test_split(
        X, y, test_size=0.2, random_state=seed, stratify=y
    )

    # Creating a new classifier

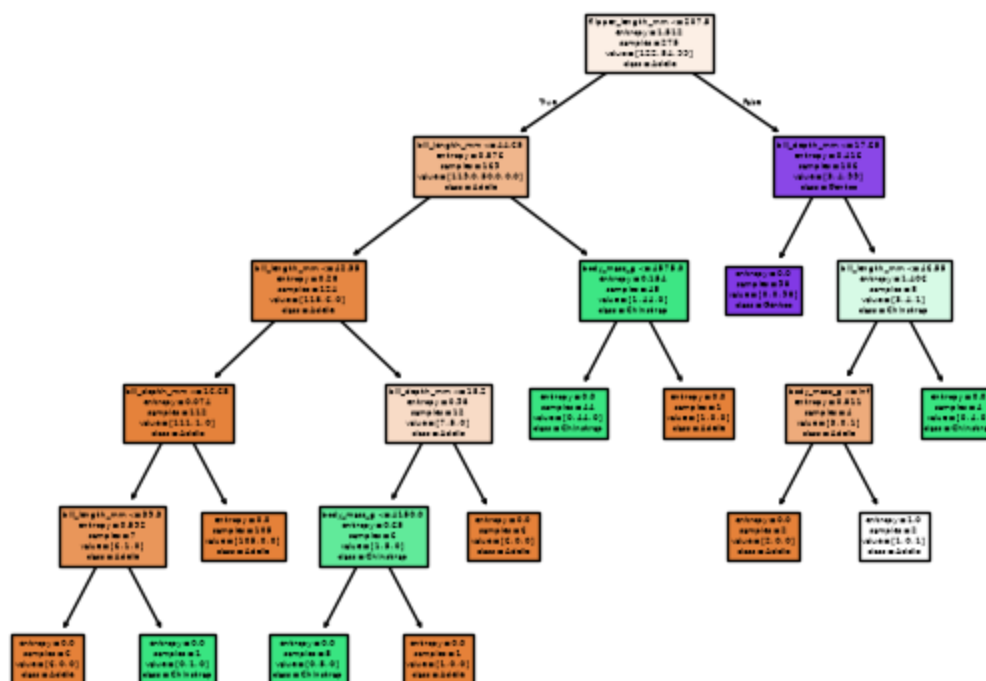
    clf = tree.DecisionTreeClassifier(
        criterion="entropy", random_state=seed
    )
```


Accuracy: 0.96

Classification Report:

	precision	recall	f1-score	support
Adelie	1.00	0.90	0.95	30
Chinstrap	0.88	1.00	0.93	14
Gentoo	0.96	1.00	0.98	25
accuracy			0.96	69
macro avg	0.95	0.97	0.95	69
weighted avg	0.96	0.96	0.96	69

Seed: 7



Accuracy: 0.94

Classification Report:

	precision	recall	f1-score	support
Adelie	0.93	0.93	0.93	30
Chinstrap	0.88	1.00	0.93	14
Gentoo	1.00	0.92	0.96	25
accuracy			0.94	69
macro avg	0.94	0.95	0.94	69
weighted avg	0.95	0.94	0.94	69

Seed: 90



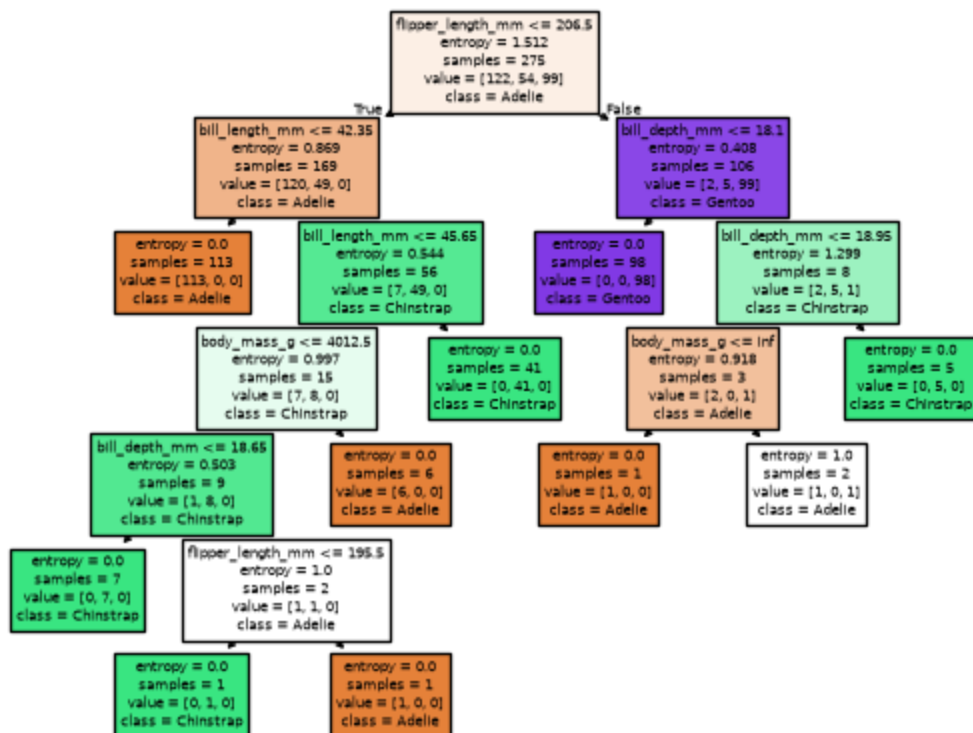
Accuracy: 0.99

Classification Report:

	precision	recall	f1-score	support
Adelie	0.97	1.00	0.98	30
Chinstrap	1.00	1.00	1.00	14
Gentoo	1.00	0.96	0.98	25
accuracy			0.99	69
macro avg	0.99	0.99	0.99	69
weighted avg	0.99	0.99	0.99	69

Seed: 96

Seed: 99



Accuracy: 0.93

Classification Report:

	precision	recall	f1-score	support
Adelie	0.96	0.90	0.93	30
Chinstrap	0.81	0.93	0.87	14
Gentoo	0.96	0.96	0.96	25
accuracy			0.93	69
macro avg	0.91	0.93	0.92	69
weighted avg	0.93	0.93	0.93	69

Seed: 2



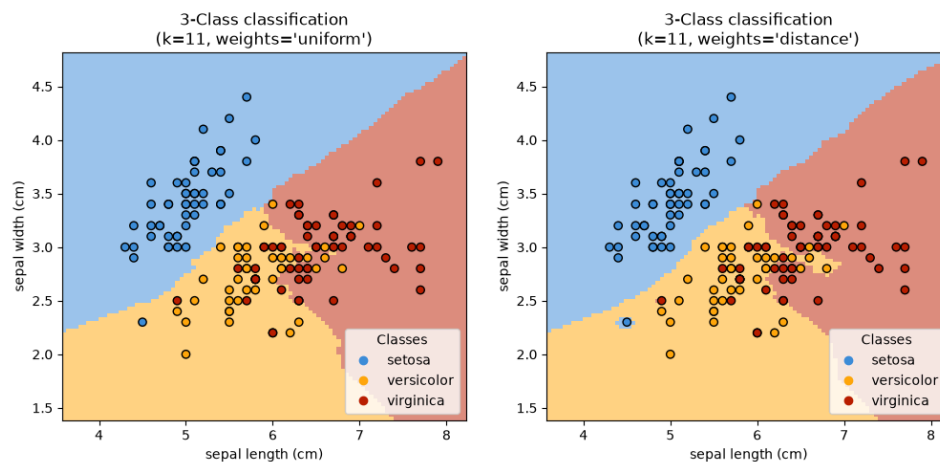
Accuracy: 0.97

Classification Report:

	precision	recall	f1-score	support
Adelie	1.00	0.93	0.97	30
Chinstrap	0.88	1.00	0.93	14
Gentoo	1.00	1.00	1.00	25
accuracy			0.97	69
macro avg	0.96	0.98	0.97	69
weighted avg	0.97	0.97	0.97	69

KNN

k-nearest neighbours (KNN)



- **Instance-based:** the training examples are the model.
- **Non-parametric:** the model does not have a fixed number of learned parameters.
- Prediction assumes that nearby examples tend to have similar targets.

Attribution: [Nearest Neighbors Classification](#), scikit-learn.

KNN provides a useful contrast with decision trees. A decision tree constructs a hierarchy of rules during learning. KNN constructs no rules and estimates no coefficients: it retains the examples and postpones most computation until a prediction is requested.

Non-parametric does not mean assumption-free. KNN assumes that the chosen distance makes proximity meaningful and that nearby examples tend to have similar targets.

Learning: Store the Data

```
def fit(self, X, y):
    self.X_train_ = X.copy()
    self.y_train_ = y.copy()
    return self
```

No tree is constructed and no coefficients are estimated.

KNN is often called a *lazy learning* algorithm. The classifier's learning step is reduced to storing the training set. Retaining a copy requires $\mathcal{O}(ND)$ memory for N examples with D features.

Data preparation and hyperparameter selection can still involve learning. For example, a standardization step estimates means and standard deviations from the training data. The point here is that the KNN classifier itself does not fit model parameters.

Finding the Nearest Neighbours

For a query x and training example x_i , Euclidean distance is

$$d(x, x_i) = \sqrt{\sum_{j=1}^D \left(x^{(j)} - x_i^{(j)}\right)^2}.$$

In [13]: `import numpy as np`

```
def nearest_neighbors(X_train, x, n_neighbors):
    distances = np.sqrt(np.sum((X_train - x) ** 2, axis=1))
    indices = np.argsort(distances, kind="stable")[:n_neighbors]
    return indices, distances[indices]
```

The straightforward full-sort implementation costs $\mathcal{O}(ND + N \log N)$ per query.

Computing all distances costs $\mathcal{O}(ND)$ and sorting them costs $\mathcal{O}(N \log N)$. A production implementation can select the nearest items without a complete sort and can sometimes use structures such as a kd-tree or ball tree.

The stable sort gives deterministic behaviour when two examples are at the same distance. This is a simple implementation choice rather than an essential part of KNN.

Voting and Averaging

After selecting the k nearest examples:

- **Classification:** each neighbour votes for its class.
- **Regression:** average the neighbours' numerical targets.
- **Uniform weights:** every neighbour has the same influence.
- **Distance weights:** closer neighbours receive weight $w_i = 1/d_i$.

For distance-weighted regression,

$$\hat{y}(x) = \frac{\sum_{i=1}^k w_i y_i}{\sum_{i=1}^k w_i}.$$

The neighbourhood search is identical for classification and regression; only the aggregation step changes. Classification totals votes by class, whereas regression averages numerical values.

Distance weighting lets the closest examples have greater influence. If a query exactly matches one or more training examples, the implementation lets only those exact matches vote. This avoids division by zero and has an intuitive interpretation.

Voting in Python

```
In [14]: def voting_weights(distances, mode):
    if mode == "uniform":
        return np.ones(len(distances))

    exact_matches = distances == 0
    if np.any(exact_matches):
        return exact_matches.astype(float)

    return 1 / distances

def vote_probabilities(labels, weights, classes):
    scores = np.array([
        np.sum(weights[labels == label])
        for label in classes
```

```
])  
return scores / np.sum(scores)
```

The comparisons `labels == label` select the neighbours belonging to each class. Their weights are added and then normalized so that the resulting scores sum to one.

With uniform weights, these are simply class proportions among the neighbours. With distance weights, they are normalized weighted votes.

Complete Implementation

[Read the tutorial](#) · [Download the notebook](#) · [Open in Colab](#)

```
In [15]: class SimpleKNeighborsClassifier:  
        """A didactic classifier with a small scikit-learn-like interface."""  
  
        def __init__(self, n_neighbors=5, weights="uniform"):  
            self.n_neighbors = n_neighbors  
            self.weights = weights  
  
        def fit(self, X, y):  
            X = np.asarray(X, dtype=float)  
            y = np.asarray(y)  
            self._validate_training_data(X, y)  
  
            self.X_train_ = X.copy()  
            self.y_train_ = y.copy()  
            self.classes_ = np.unique(y)  
            self.n_features_in_ = X.shape[1]  
            return self  
  
        def _predict_proba_one(self, x):  
            indices, distances = nearest_neighbors(  
                self.X_train_, x, self.n_neighbors  
            )  
            labels = self.y_train_[indices]  
            weights = voting_weights(distances, self.weights)  
            return vote_probabilities(labels, weights, self.classes_)  
  
        def predict_proba(self, X):  
            X = self._validate_prediction_data(X)  
            return np.vstack([self._predict_proba_one(x) for x in X])  
  
        def predict(self, X):  
            probabilities = self.predict_proba(X)  
            return self.classes_[np.argmax(probabilities, axis=1)]  
  
        def score(self, X, y):  
            return float(np.mean(self.predict(X) == np.asarray(y)))  
  
        def _validate_training_data(self, X, y):  
            if X.ndim != 2 or y.ndim != 1 or len(X) != len(y):  
                raise ValueError()
```



```

        "X must be 2-D and y must be 1-D with matching rows"
    )
    if len(y) == 0:
        raise ValueError("the training set cannot be empty")
    if not np.isfinite(X).all():
        raise ValueError(
            "missing and non-finite feature values are unsupported"
        )
    if not isinstance(self.n_neighbors, (int, np.integer)):
        raise ValueError("n_neighbors must be an integer")
    if not 1 <= self.n_neighbors <= len(y):
        raise ValueError("n_neighbors must be between 1 and len(y)")
    if self.weights not in {"uniform", "distance"}:
        raise ValueError("weights must be 'uniform' or 'distance'")

    def _validate_prediction_data(self, X):
        if not hasattr(self, "X_train_"):
            raise ValueError("call fit before making predictions")
        X = np.asarray(X, dtype=float)
        if X.ndim == 1:
            X = X.reshape(1, -1)
        if X.ndim != 2 or X.shape[1] != self.n_features_in_:
            raise ValueError("X has the wrong number of features")
        if not np.isfinite(X).all():
            raise ValueError(
                "missing and non-finite feature values are unsupported"
            )
        return X

```

When weighted votes are tied, `np.argmax` chooses the first class in the sorted `classes_` array. Production software may offer different tie-breaking rules.

KNN on Palmer Penguins

```

In [16]: from palmerpenguins import load_penguins
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

knn_features = ["bill_length_mm", "bill_depth_mm"]
penguins_knn = load_penguins()
penguins_knn = penguins_knn[
    knn_features + ["species"]
].dropna().copy()

X_knn = penguins_knn[knn_features].to_numpy()
y_knn = penguins_knn["species"].to_numpy()
X_knn_train, X_knn_test, y_knn_train, y_knn_test = train_test_split(
    X_knn, y_knn, test_size=0.2, random_state=42, stratify=y_knn
)

knn_scaler = StandardScaler()
X_knn_train_scaled = knn_scaler.fit_transform(X_knn_train)
X_knn_test_scaled = knn_scaler.transform(X_knn_test)

```

```

for mode in ("uniform", "distance"):
    knn = SimpleKNeighborsClassifier(n_neighbors=5, weights=mode)
    knn.fit(X_knn_train_scaled, y_knn_train)
    print(f"weights='{mode}': {knn.score(X_knn_test_scaled, y_knn_test):.3f}")

```

weights='uniform': 0.957
weights='distance': 0.957

This example retains all three penguin species. Bill length and bill depth make the differences among the decision boundaries easier to see than the earlier, nearly separable Gentoo-versus-non-Gentoo problem.

KNN is sensitive to feature scale because its predictions depend directly on distance. The scaler is fitted using only the training data, preventing information from the test set from influencing preprocessing.

KNN Decision Boundaries

```

In [17]: import matplotlib.pyplot as plt
from sklearn.inspection import DecisionBoundaryDisplay
from sklearn.neighbors import KNeighborsClassifier
from sklearn.pipeline import make_pipeline

configurations = [
    (1, "uniform"),
    (5, "uniform"),
    (25, "uniform"),
    (5, "distance"),
]
classes, y_encoded = np.unique(y_knn_train, return_inverse=True)

fig, axes = plt.subplots(2, 2, figsize=(11, 7), sharex=True, sharey=True)
for ax, (n_neighbors, weights) in zip(axes.ravel(), configurations):
    sklearn_knn = make_pipeline(
        StandardScaler(),
        KNeighborsClassifier(n_neighbors=n_neighbors, weights=weights),
    )
    sklearn_knn.fit(X_knn_train, y_encoded)
    DecisionBoundaryDisplay.from_estimator(
        sklearn_knn,
        X_knn_train,
        response_method="predict",
        multiclass_colors="Set2",
        alpha=0.25,
        ax=ax,
    )

    for label, name in enumerate(classes):
        selected = y_encoded == label
        ax.scatter(
            X_knn_train[selected, 0],
            X_knn_train[selected, 1],
            color=plt.get_cmap("Set2")(label),

```

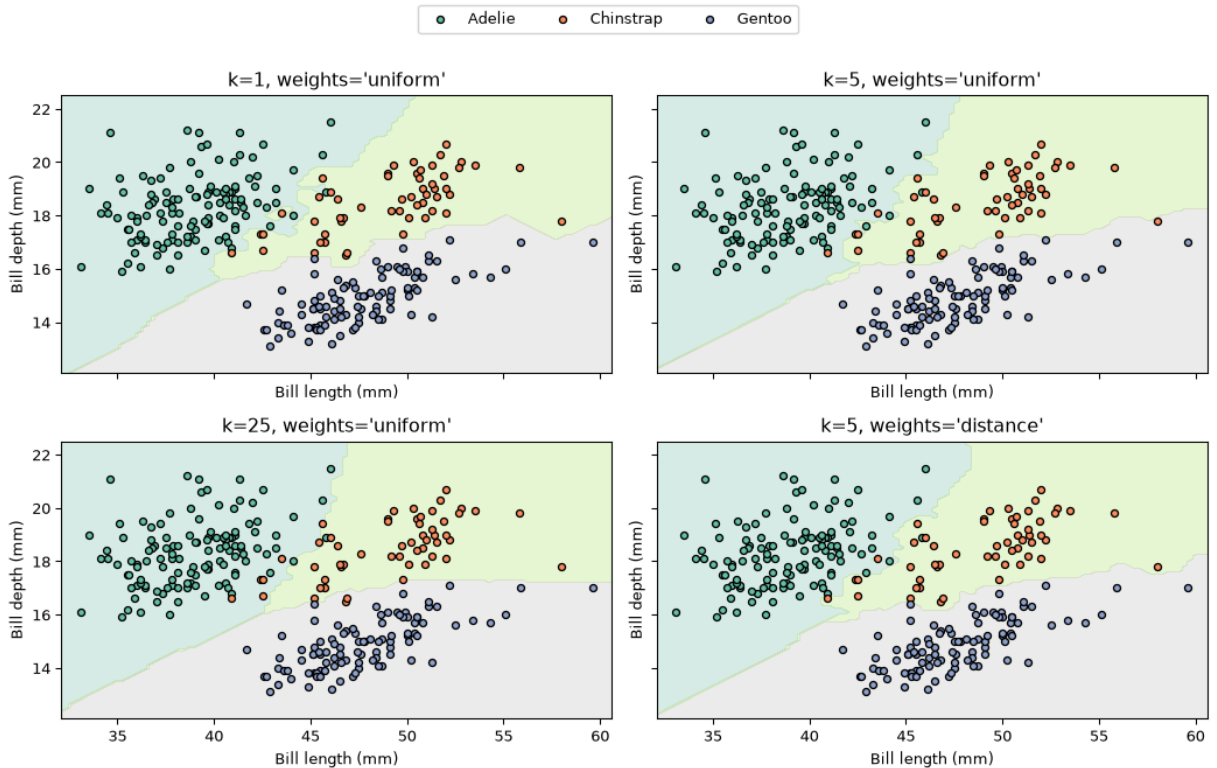
```

        edgecolor="black",
        s=20,
        label=name,
    )

    ax.set_title(f"k={n_neighbors}, weights='{weights}'")
    ax.set_xlabel("Bill length (mm)")
    ax.set_ylabel("Bill depth (mm)")

handles, labels = axes[0, 0].get_legend_handles_labels()
fig.legend(handles, labels, loc="upper center", ncols=len(classes))
fig.tight_layout(rect=(0, 0, 1, 0.93))
plt.show()

```



With $k = 1$, every training observation controls a small surrounding region, so the boundary is irregular and sensitive to individual examples. Increasing k smooths the boundary because a larger neighbourhood must agree.

Distance weighting gives the closest observations greater influence. Its boundary with $k = 5$ can therefore retain more local detail than uniform voting with the same k .

The diagram deliberately uses scikit-learn rather than extending the teaching implementation with plotting machinery. The pipeline standardizes the features internally while keeping the axes in their original units.

KNN Limitations

- Prediction retains and searches the training data.

- Distances are sensitive to **feature scaling** and the chosen metric.
- Neighbourhoods become less informative in high-dimensional spaces.
- Small k can be sensitive to noise; large k can hide local structure.
- Class imbalance can dominate a neighbourhood's vote.

The loss of useful distance contrast in high-dimensional spaces is commonly called the *curse of dimensionality*. KNN can also require substantial memory and prediction time because its effective model grows with the training set.

For regression, predictions are averages of observed neighbour targets and therefore do not extrapolate beyond the observed target range.

KNN Exercises and Resources

Experiment with changes in k , voting weights, distance, and feature scaling:

- [Nearest Neighbors Classification](#)
- [Nearest Neighbors Regression](#)
- [Importance of Feature Scaling](#)

The scikit-learn examples are downloadable as Jupyter notebooks. You can also modify the executable notebook produced from this lecture—for example, by changing the four configurations in the decision-boundary cell.

Summary

- The lecture surveyed **decision trees** and **k-nearest neighbours (KNN)**, and used **logistic regression** to illustrate a linear decision boundary.
- We constructed a decision tree that greedily minimizes weighted entropy and a KNN classifier that **stores, searches, and votes**.
- The Palmer Penguins examples illustrated how different learning algorithms produce different **decision boundaries**.

Prologue

Resources

- [Plot the decision surface of decision trees trained on the iris dataset](#) from `sklearn`
- [Decision trees](#) by Jan Kirenz, a Professor at HdM Stuttgart
- [CS 320 Apr12-2021 \(Part 2\) - Decision Boundaries](#) by Tyler Caraza-Harter, an Instructor at UW-Madison

References

Geurts, Pierre, Alexandre Irrthum, and Louis Wehenkel. 2009. "Supervised Learning with Decision Tree-Based Methods in Computational and Systems Biology." *Molecular bioSystems* 5 (12): 1593–605. <https://doi.org/10.1039/b907946g>.

Hyafil, Laurent, and Ronald L. Rivest. 1976. "Constructing Optimal Binary Decision Trees Is NP-Complete." *Inf. Process. Lett.* 5 (1): 15–17. [https://doi.org/10.1016/0020-0190\(76\)90095-8](https://doi.org/10.1016/0020-0190(76)90095-8).

Russell, Stuart, and Peter Norvig. 2020. *Artificial Intelligence: A Modern Approach*. 4th ed. Pearson. <http://aima.cs.berkeley.edu/>.

Stiglic, Gregor, Simon Kocbek, Igor Pernek, and Peter Kokol. 2012. "Comprehensive Decision Tree Models in Bioinformatics." *PLoS ONE* 7 (3): e33812. <https://doi.org/10.1371/journal.pone.0033812>.

Next lecture

- Training a linear model

Marcel **Turcotte**

Marcel.Turcotte@uOttawa.ca

School of Electrical Engineering and **Computer Science** (EECS)

University of Ottawa