

Machine Learning

- Many people imagine that data science is mostly machine learning and that data scientists mostly build and train and tweak machine learning models all day long.
- In fact, data science is mostly turning business problems into data problems and collecting data and understanding data and cleaning data and formatting data, after which machine learning is almost an afterthought.

Modeling

- Before we can talk about machine learning, we need to talk about models.
- What is a model? It's simply a specification of a mathematical (or probabilistic) relationship that exists between different variables.
- For instance, if you're trying to raise money for your social networking site, you might build a business model (likely in a spreadsheet) that takes inputs like "number of users," "ad revenue per user," and "number of employees" and outputs your annual profit for the next several years.
- A cookbook recipe entails a model that relates inputs like "number of eaters" and "hungriness" to quantities of ingredients needed.

What Is Machine Learning?

- Machine learning refers to creating and using models that are learned from data.
- Our goal will be to **use existing data to develop models** that we can use to **predict** various outcomes for new data, such as:
 - ❖ Whether an email message is spam or not
 - ❖ Whether a credit card transaction is fraudulent
 - ❖ Which advertisement a shopper is most likely to click on
 - ❖ Which football team is going to win the Super Bowl

What Is Machine Learning?

- **Supervised models** → in which there is a set of data labeled with the correct answers to learn from
- **Unsupervised models** → in which there are no such labels.
- **Semisupervised** → in which only some of the data are labeled
- **Online** → in which the model needs to continuously adjust to newly arriving data
- **Reinforcement** → in which, after making a series of predictions, the model gets a signal indicating how well it did.
- Before we can do that, we need to better understand the fundamentals of machine learning

Overfitting and Underfitting

● Underfitting Example:

- You create a very simple model that **only looks at email length** to decide if it's spam.
- Your model might say:
 - | "If the email is longer than 100 characters, it's spam. Otherwise, it's not."
- **Result:**
 - Many obvious spam emails get missed.
 - Some legitimate long emails (e.g., newsletters) get flagged incorrectly.
 - The model is too simplistic to detect real spam patterns like suspicious links, keywords, or sender behavior.

| **This is underfitting:** the model is too basic and can't capture the complexity of spam detection.

Overfitting and Underfitting

● Overfitting Example:

- You now build a very complex model using:
 - Every single word in the email
 - The number of exclamation marks
 - Whether the email was sent at 3:02 AM
 - The recipient's name
 - ...and 2000 other features
- The model performs perfectly on your training set. But when new emails arrive:
 - It starts misclassifying new spam because it learned specific quirks in your training data rather than general spam patterns.
 - For example, it incorrectly flags an email from "David" because one spam message in training was from a "David."

This is overfitting: the model memorized the training data, including noise, and performs poorly on new, unseen data.

Overfitting and Underfitting

✓ Ideal Scenario:

- You use relevant features like:
 - Presence of common spam keywords ("free", "win", "click")
 - URL patterns
 - Known blacklisted domains
- You train a model that captures true spam patterns and generalizes well.

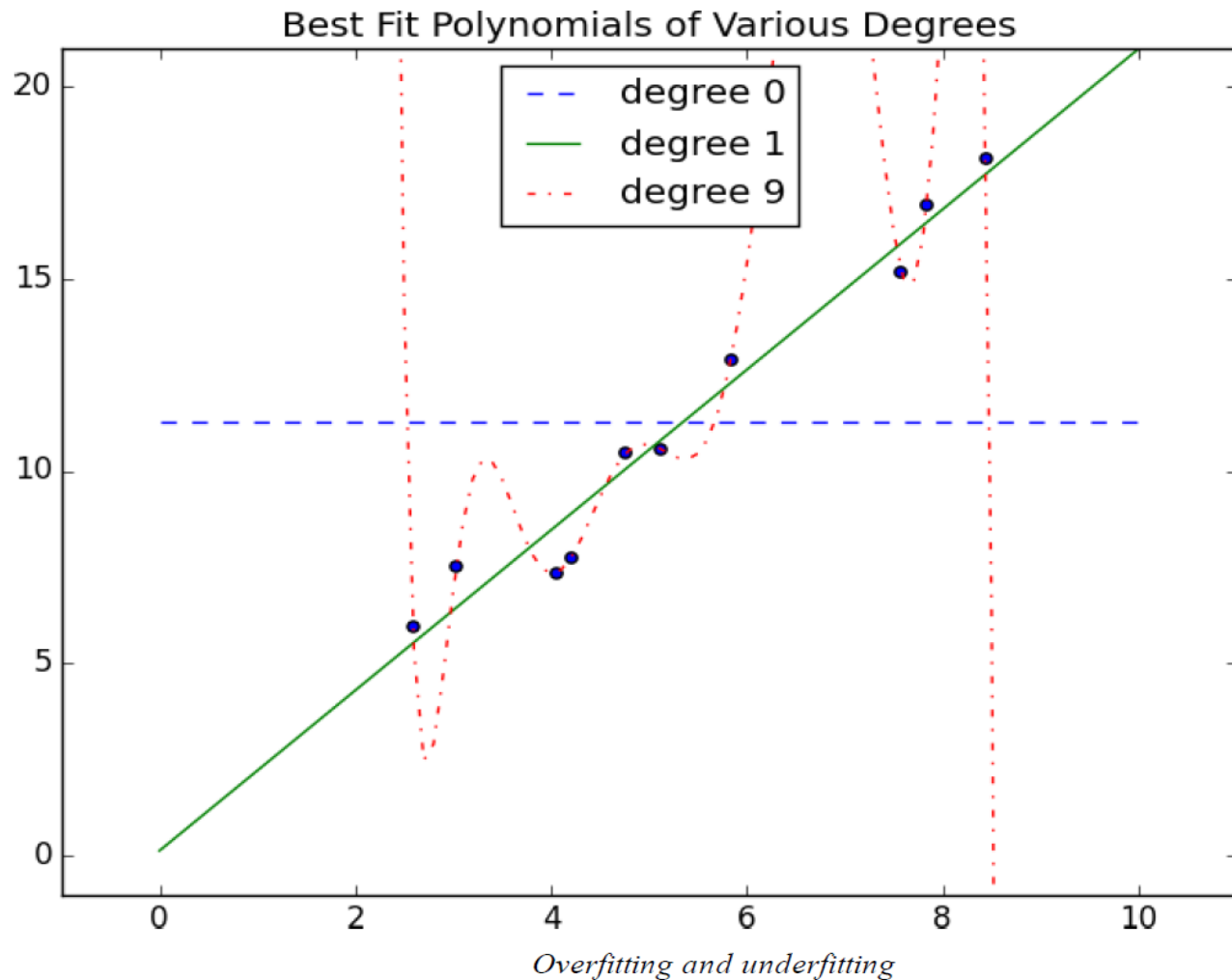
This is a well-fit model — not too simple, not too complex.

Overfitting and Underfitting

- A common danger in machine learning is ***overfitting***—producing a model that performs well on the data you train it on but generalizes poorly to any new data.
- This could involve learning noise in the data. Or it could involve learning to identify specific inputs rather than whatever factors are actually predictive for the desired output.
- The other side of this is ***underfitting***—producing a model that doesn't perform well even on the training data, although typically when this happens you decide your model isn't good enough and keep looking for a better one.

Overfitting and Underfitting

- In the figure, we've fit three polynomials to a sample of data.



Overfitting and Underfitting

- In the figure, we've fit three polynomials to a sample of data.

The chart shows how polynomial regression of different degrees fits the same set of data points (blue dots).

There are three lines:

1. Degree 0 (Blue Dashed Line):

- This is a **constant model** (just the average of all the y-values).
- It **underfits** the data badly — it doesn't capture any trend or structure.
- Too simple to be useful.

2. Degree 1 (Green Solid Line):

- This is a **linear regression** line.
- It represents a reasonable balance, capturing the overall trend.
- Likely the best fit for this data in terms of **generalization**.

Overfitting and Underfitting

- In the figure, we've fit three polynomials to a sample of data.

3. Degree 9 (Red Dotted Line):

- This is a **high-degree polynomial** that perfectly fits every point.
- It **overfits** the data — capturing noise and fluctuations rather than true trends.
- While it has zero error on training data, it would likely perform **poorly on unseen data**.

- **Underfitting** happens when the model is too simple to capture the data pattern.
- **Overfitting** occurs when the model is too complex and fits noise.

Overfitting and Underfitting

- In the figure, we've fit three polynomials to a sample of data.

Degree	Example Equation	Behavior	Risk
0	$y = c$	Constant	Ignores all input features
1	$y = aX + b$	Linear	Can't capture curves
2	$y = aX^2 + bX + c$	Parabola	Good balance (if trend is curved)
9	Complex polynomial	Very flexible	Likely overfits

Overfitting and Underfitting


- **Solution:**
- So how do we make sure our models aren't too complex? The most fundamental approach involves **using different data to train the model and to test the model.**
- The simplest way to do this is to **split the dataset**, so that (for example) two-thirds of it is used to train the model, after which we measure the model's performance on the remaining third.

Correctness

- Imagine building a model to make a binary judgment.
- Is this email spam? Should we hire this candidate? Is this air traveler secretly a terrorist?
- Given a set of labeled data and such a predictive model, every data point lies in one of four categories:


- ◆ **1. True Positive (TP)**

- **Definition:** The message is spam, and the model correctly predicted it as spam.
- **Meaning:** Good result — the model did its job correctly.

|  Example: Spam email marked as spam.

- ◆ **2. False Positive (FP) — Type 1 Error**

- **Definition:** The message is not spam, but the model predicted it as spam.
- **Meaning:** Incorrect result — a legitimate message is wrongly flagged.

|  Example: Important email incorrectly sent to spam.

Correctness

◆ 3. False Negative (FN) — Type 2 Error

- **Definition:** The message is spam, but the model predicted it as not spam.
- **Meaning:** Incorrect result — spam sneaks through the filter.

✗ Example: Spam email lands in your inbox.

◆ 4. True Negative (TN)

- **Definition:** The message is not spam, and the model correctly predicted it as not spam.
- **Meaning:** Good result — the system recognized a legitimate message correctly.

✓ Example: Normal email stays in inbox.

Confusion matrix

- We often represent these as counts in a confusion matrix:

	Spam	Not spam
Predict “spam”	True positive	False positive
Predict “not spam”	False negative	True negative

Confusion Matrix

- Let's see how our leukemia test fits into this framework.
- These days approximately 5 babies out of 1,000 are named Luke.
- And the lifetime prevalence of leukemia is about 1.4%, or 14 out of every 1,000 people.
- If we believe these two factors are independent and apply my “Luke is for leukemia” test to 1 million people, we'd expect to see a confusion matrix like:

	Leukemia	No leukemia	Total
“Luke”	70	4,930	5,000
Not “Luke”	13,930	981,070	995,000
Total	14,000	986,000	1,000,000

Accuracy

- We can then use these to compute various statistics about model performance.

- **Accuracy** is defined as the **fraction of correct predictions**:

```
def accuracy(tp: int, fp: int, fn: int, tn: int) -> float:
```

```
    correct = tp + tn
```

```
    total = tp + fp + fn + tn
```

```
    return correct / total
```

```
assert accuracy(70, 4930, 13930, 981070) == 0.98114
```

- That seems like a pretty impressive number.
- But clearly this is not a good test, which means that we probably shouldn't put a lot of credence in raw accuracy.

Precision

- It's common to look at the combination of precision and recall.
- **What is Precision?**
- Precision is defined as the **ratio of correctly classified positive samples (True Positive) to a total number of classified positive samples** (either correctly or incorrectly).
- $\text{Precision} = \text{True Positive} / (\text{True Positive} + \text{False Positive})$
- $\text{Precision} = \text{TP} / (\text{TP} + \text{FP})$

```
def precision(tp: int, fp: int, fn: int, tn: int) -> float:  
    return tp / (tp + fp)  
assert precision(70, 4930, 13930, 981070) == 0.014
```

- Precision helps us to visualize the reliability of the machine learning model in classifying the model as positive.

Recall

- **What is Recall?**
- The recall is calculated as the **ratio between the numbers of Positive samples correctly classified as Positive to the total number of Positive samples.**
- The recall measures the model's ability to detect **positive samples.**
- $\text{Recall} = \text{True Positive} / (\text{True Positive} + \text{False Negative})$
- $\text{Recall} = \text{TP} / (\text{TP} + \text{FN})$

```
def recall(tp: int, fp: int, fn: int, tn: int) -> float:  
    return tp / (tp + fn)
```

```
assert recall(70, 4930, 13930, 981070) == 0.005
```

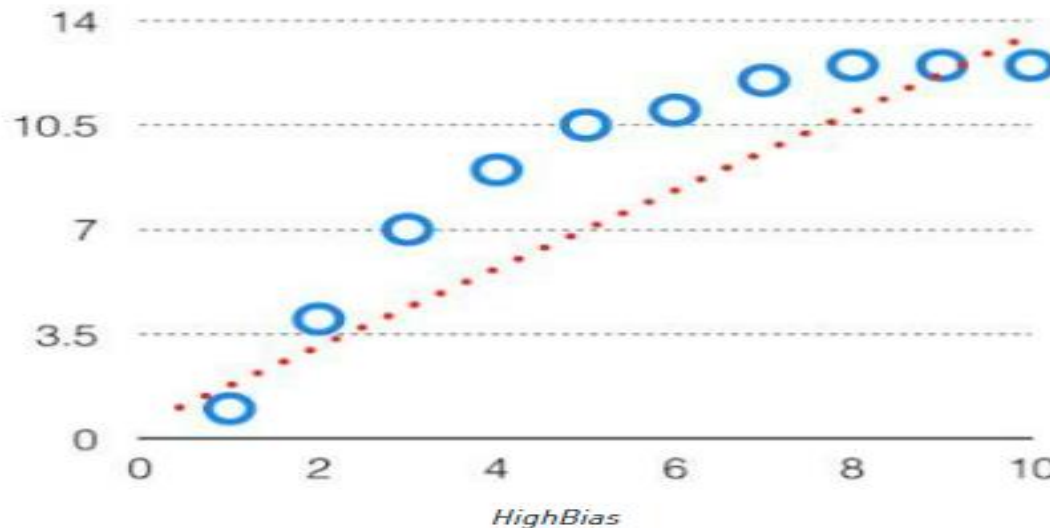
F1-Score

```
def f1_score(tp: int, fp: int, fn: int, tn: int) -> float:  
    p = precision(tp, fp, fn, tn)  
    r = recall(tp, fp, fn, tn)  
    return 2 * p * r / (p + r)
```

- This is the *harmonic mean* of precision and recall and necessarily lies between them.
- Tradeoff between precision and recall. A model that predicts “yes” when it’s even a little bit confident will probably have a high recall but a low precision; a model that predicts “yes” only when it’s extremely confident is likely to have a low recall and a high precision.

Bias

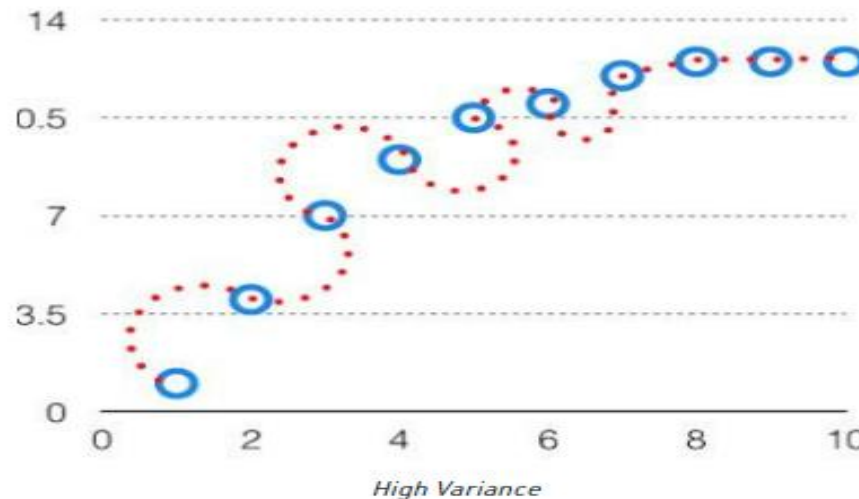
- It is important to understand **prediction errors** (bias and variance) when it comes to accuracy in any machine learning algorithm.
- The bias is known as the **difference between the prediction of the values by the ML model and the correct value.**
- Being high in biasing gives a **large error** in training as well as testing data.



- Its recommended that an algorithm should always **be low biased to avoid the problem of underfitting.**
- By high bias, the data predicted is in a straight line format, thus not fitting accurately in the data in the data set.
- Such fitting is known as **Underfitting of Data.**

Variance

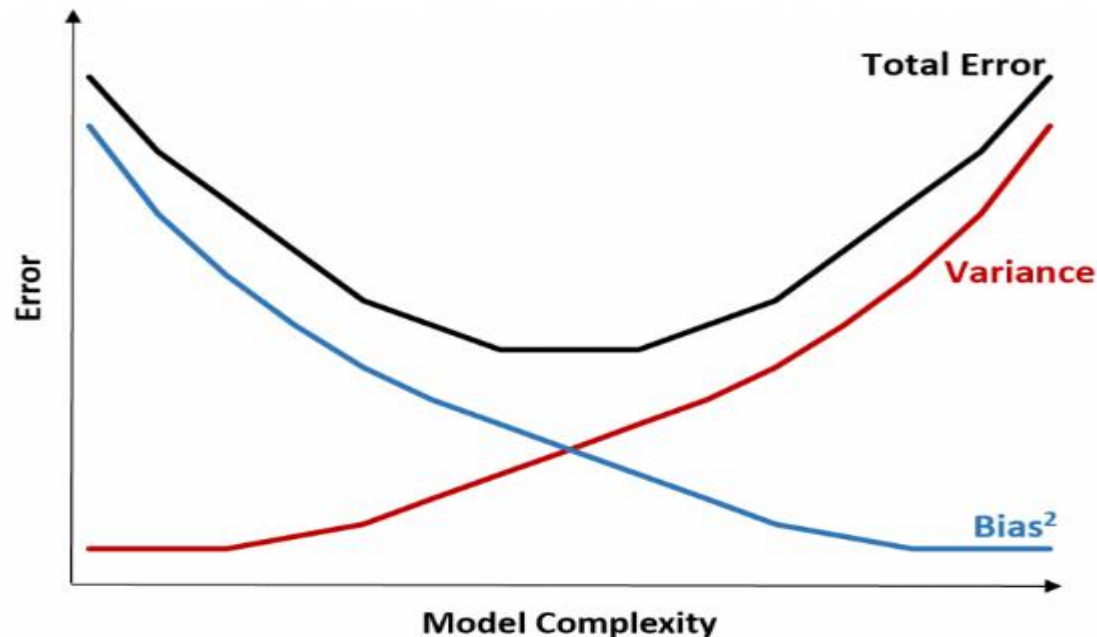
- The variance of the model refers to the **amount by which our model would change if we estimated it using a different training set.**
- The model with high variance has a **very complex fit** to the training data and thus is not able to fit accurately on the data which it hasn't seen before.
- As a result, such models perform very well on training data but has high error rates on test data.
- When a model is high on variance, it is then said to as **Overfitting of Data.**
- Overfitting is fitting the training set accurately via complex curve.
- **While training a data model variance should be kept low.**



- The high variance data looks as above.

The Bias-Variance Tradeoff

- The bias-variance tradeoff refers to the tradeoff that takes place when we choose to lower bias which typically increases variance, or lower variance which typically increases bias.
- The following chart offers a way to visualize this tradeoff.



The Bias-Variance Tradeoff

- The **total error decreases** as the complexity of a model increases but only up to a certain point.
- **Past a certain point**, variance begins to increase and total error also begins to increase.
- In practice, we only **care about minimizing the total error of a model**, not necessarily minimizing the variance or bias.
- It turns out that the way to minimize the total error is to **strike the right balance between variance and bias**.
- In other words, we want a model that is **complex enough to capture the true relationship** between the explanatory variables and the response variable, but not overly complex such that it finds patterns that don't really exist.
- When a **model is too complex, it overfits the data**.
- But when a model is **too simple, it underfits the data**. This happens because it **assumes the true relationship** between the explanatory variables and the response variable is more simple than it actually is.
- The way to pick optimal models in machine learning is to **strike the balance between bias and variance such that we can minimize the test error of the model on future unseen data**.

Feature Extraction and Selection

- As has been mentioned, when your data doesn't have enough features, your model is likely to underfit.
- And when your data has too many features, it's easy to overfit.
- *But what are features, and where do they come from?*
- Features are whatever inputs we provide to our model.
- If you want to predict someone's salary based on her years of experience, then years of experience is the only feature you have.

Feature Extraction and Selection

- Things become more interesting as your data becomes more complicated.
- Imagine trying to build a spam filter to predict whether an email is junk or not.
- Most models won't know what to do with a raw email, which is just a collection of text.
- You'll have to extract features. For example:
 - ❖ Does the email contain the word Gun?
 - ❖ How many times does the letter d appear?
 - ❖ What was the domain of the sender?
- The answer to a question like the first question here is simply a yes or no, which we typically encode as a 1 or 0.
- The second is a number.
- And the third is a choice from a discrete set of options.

Feature Extraction and Selection

- Pretty much always, we'll extract features from our data that fall into one of these three categories.
- What's more, the types of features we have constrain the types of models we can use.
 - ❖ The Naive Bayes classifier is suited to yes-or-no features, like the first one in the preceding list.
 - ❖ Regression models, require numeric features (which could include dummy variables that are 0s and 1s).
 - ❖ Decision trees, can deal with numeric or categorical data.

k-Nearest Neighbors: The Model

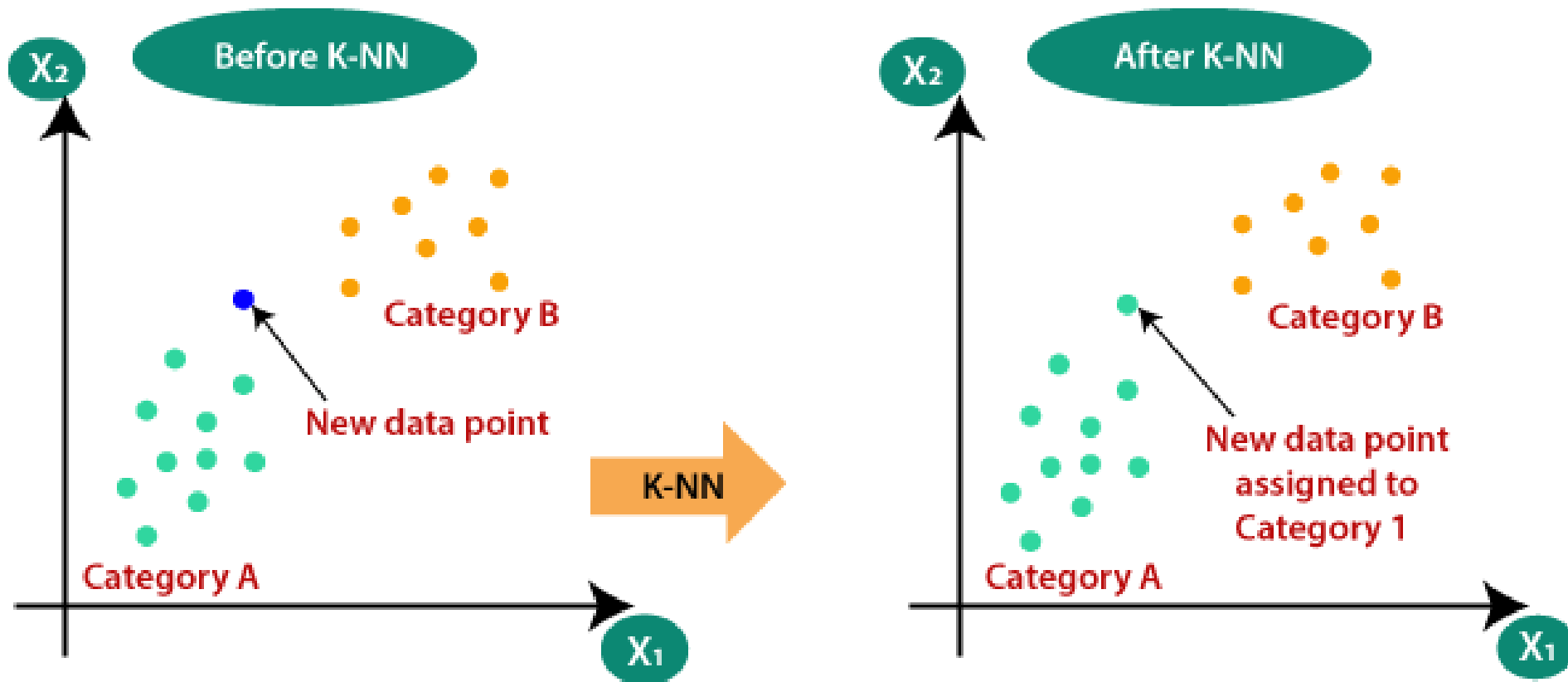
What is k-Nearest Neighbors?

- k-NN is a machine learning algorithm that helps make predictions based on the idea that similar things are close together.
- It is a Supervised Classification or Regression Method

How It Works:

- **Pick a number “k”** (how many neighbors to look at).
- **Measure the distance** between your data point and all others in the dataset (usually with Euclidean distance—think of it as drawing a straight line).
- **Find the k closest points.**
- **Look at their labels** (e.g., "cat," "dog," "spam," "not spam").
- **Predict** the most common label among those neighbors.

k-Nearest Neighbors: The Model



k-Nearest Neighbors: The Model

- **Pros:**

- Very simple and intuitive
- No training phase (it's a lazy learner)

- **Cons:**

- Slow with large datasets
- Doesn't work well with irrelevant features or very different scales
- Neglects a lot of information, since the prediction for each new point depends only on the handful of points closest to it.

k-Nearest Neighbors: The Model



Sample Dataset

Point	Feature 1 (X)	Feature 2 (Y)	Label
A	1	2	Red
B	2	3	Red
C	3	1	Blue
D	6	5	Blue
E	7	7	Blue

k-Nearest Neighbors: The Model

Goal:

Classify a new point $P = (3, 3)$ using $k = 3$ neighbors.

Step 1: Calculate Euclidean Distance

Euclidean distance formula:

$$\text{distance} = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Point	Coordinates	Distance to P(3,3)	Label
A	(1, 2)	$\sqrt{[(3-1)^2 + (3-2)^2]} = \sqrt{5} \approx 2.24$	Red
B	(2, 3)	$\sqrt{[(3-2)^2 + (3-3)^2]} = \sqrt{1} = 1.00$	Red
C	(3, 1)	$\sqrt{[(3-3)^2 + (3-1)^2]} = \sqrt{4} = 2.00$	Blue
D	(6, 5)	$\sqrt{[(3-6)^2 + (3-5)^2]} = \sqrt{13} \approx 3.61$	Blue
E	(7, 7)	$\sqrt{[(3-7)^2 + (3-7)^2]} = \sqrt{32} \approx 5.66$	Blue

k-Nearest Neighbors: The Model

Step 2: Pick the 3 Nearest Neighbors

Closest distances:

- B (1.00) – Red
 - C (2.00) – Blue
 - A (2.24) – Red
-

Step 3: Majority Vote

- Red: 2 votes (A, B)
- Blue: 1 vote (C)

 Predicted Label: Red

The Model : Code Snippet

- Creating a classifier:

```
from typing import NamedTuple
from scratch.linear_algebra import Vector, distance
class LabeledPoint(NamedTuple):
    point: Vector
    label: str
def knn_classify(k: int, labeled_points: List[LabeledPoint], new_point: Vector) -> str:
    # Order the labeled points from nearest to farthest.
    by_distance = sorted(labeled_points, key=lambda lp:
        distance(lp.point, new_point))
    # Find the labels for the k closest
    k_nearest_labels = [lp.label for lp in by_distance[:k]]
    # and let them vote.
    return majority_vote(k_nearest_labels)
```

The Model : Code Snippet

```
def majority_vote(labels: List[str]) -> str:
    """Assumes that labels are ordered from nearest to farthest."""
    vote_counts = Counter(labels)
    winner, winner_count = vote_counts.most_common(1)[0]
    num_winners = len([count for count in vote_counts.values() if count
                        == winner_count])
    if num_winners == 1:
        return winner # unique winner, so return it
    else:
        return majority_vote(labels[:-1]) # try again without the
        farthest
    # Tie, so look at first 4, then 'b'
assert majority_vote(['a', 'b', 'c', 'b', 'a']) == 'b'
```

The Model : KNN Program

```
from typing import List, NamedTuple
from collections import Counter
import math

# ----- Data Structures -----
class LabeledPoint(NamedTuple):
    point: List[float]
    label: str

# ----- Euclidean Distance Function -----
def distance(p1: List[float], p2: List[float]) -> float:
    return math.sqrt(sum((x - y) ** 2 for x, y in zip(p1, p2)))

# ----- Tie-Breaking Majority Vote -----
def majority_vote(labels: List[str]) -> str:
    """Assumes labels are ordered from nearest to farthest."""
    vote_counts = Counter(labels)
    winner, winner_count = vote_counts.most_common(1)[0]
    num_winners = len([count for count in vote_counts.values() if count ==
winner_count])
    if num_winners == 1:
        return winner # Unique winner
    else:
        return majority_vote(labels[:-1]) # Remove farthest and retry
```

The Model : KNN Program

```
# ----- k-NN Classifier -----

def knn_classify(k: int, labeled_points: List[LabeledPoint], new_point: List[float]) -> str:
    # Sort points by distance from new_point
    by_distance = sorted(labeled_points, key=lambda lp: distance(lp.point, new_point))
    # Get labels of the k closest
    k_nearest_labels = [lp.label for lp in by_distance[:k]]
    # Let them vote
    return majority_vote(k_nearest_labels)

# ----- Example Usage -----

if __name__ == "__main__":
    # Example dataset (2D points)
    data = [
        LabeledPoint([1.0, 2.0], "A"),
        LabeledPoint([2.0, 3.0], "A"),
        LabeledPoint([3.0, 3.0], "B"),
        LabeledPoint([6.0, 5.0], "B"),
        LabeledPoint([7.0, 8.0], "C")
    ]

    # New point to classify
    new_point = [3.5, 3.5]
    # Classify with k=3
    k = 3

    predicted_label = knn_classify(k, data, new_point)

    print(f"Predicted label for {new_point} with k={k} is: {predicted_label}")
```