Model Selection & Ensembles

Choosing, Tuning, and Combining Models

Nipun Batra | IIT Gandhinagar

Learning Goals

By the end of this lecture, you will:

Goal	What You'll Learn
Understand	Why models fail (bias vs variance)
Implement	Cross-validation for reliable evaluation
Tune	Hyperparameters with Grid/Random search
Build	Ensemble models (Random Forest, XGBoost)
Apply	Clustering when no labels exist

The Story So Far

We know several algorithms:

Algorithm	Type	When to Use
Linear Regression	Regression	Linear relationships
Logistic Regression	Classification	Binary decisions
Decision Trees	Both	Interpretable rules
K-NN	Both	Similar items

But how do we choose? And can we do better?

Today's Questions

1. Why does my model perform poorly on new data?
2. How do I reliably compare models?
3. How do I find the best settings for a model?
4. Can I combine models to get better results?
5. What if I don't have labels?

Part 1: Bias-Variance Tradeoff

The Fundamental Reason Models Fail

A Story of Two Mistakes

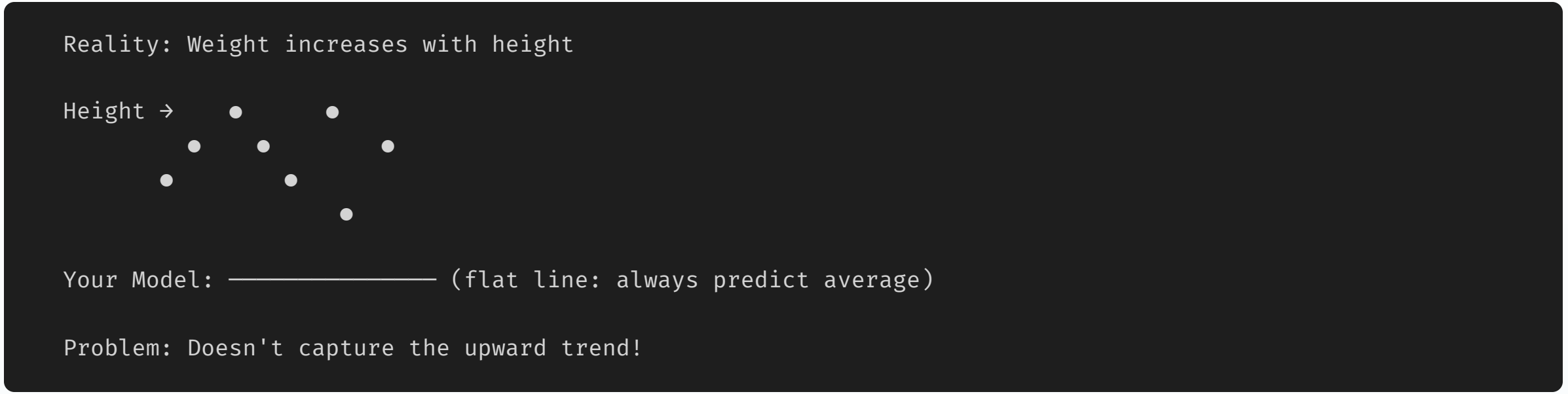
Imagine you're estimating someone's weight from their height.

Mistake Type	What Happened	The Problem
High Bias	Used a flat line (everyone = 70 kg)	Too simple
High Variance	Memorized every person exactly	Too complex

> Every ML mistake is one of these two types!

Underfitting: The Too-Simple Model

Definition: Model is too simple to capture the pattern.



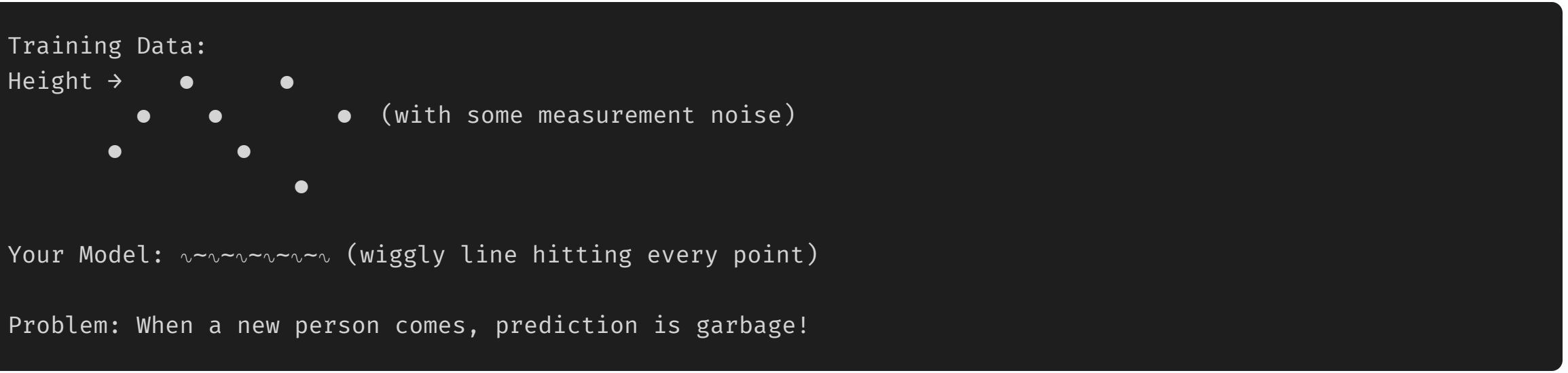
Underfitting: Symptoms

Symptom	Value
Training Error	HIGH
Test Error	HIGH
Model	Too simple

Both errors are high because the model can't even fit the training data!

Overfitting: The Memorizer

Definition: Model is too complex and memorizes noise.



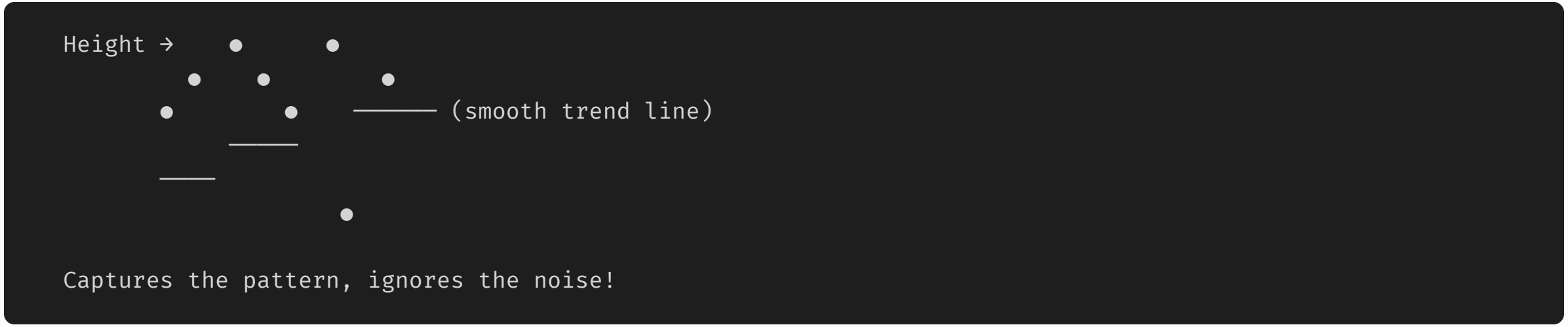
Overfitting: Symptoms

Symptom	Value
Training Error	LOW (near perfect!)
Test Error	HIGH
Model	Too complex

Big gap between training and test error!

The Goldilocks Model

Goal: A model that's "just right"



Symptom	Value
Training Error	Low
Test Error	Low
Gap	Small

Bias-Variance Decomposition

Mathematical view:

$$\text{Total Error} = \text{Bias}^2 + \text{Variance} + \text{Irreducible Noise}$$

Component	What It Means	Source
Bias²	Error from wrong assumptions	Model too simple
Variance	Sensitivity to training data	Model too complex
Noise	Random error in data	Can't reduce

The Tradeoff Visualized

You can't minimize both simultaneously!

Bias-Variance in Different Models

Model	Complexity	Bias	Variance
Linear Regression	Low	High	Low
Polynomial (degree 2)	Medium	Medium	Medium
Polynomial (degree 20)	High	Low	High
Decision Tree (depth 3)	Low	High	Low
Decision Tree (depth 50)	High	Low	High

Diagnosing Your Model

Train Error	Test Error	Diagnosis	Fix
High	High	Underfitting	More complexity
Low	High	Overfitting	Less complexity
Low	Low	Good fit!	Deploy it
High	Low	Something weird	Check for bugs

Learning Curves

Plot error vs training set size:

More data helps reduce overfitting!

Solutions for Overfitting

Technique	How It Helps
More data	Harder to memorize
Simpler model	Less capacity to overfit
Regularization	Penalize complexity
Early stopping	Stop before memorizing
Cross-validation	Better evaluation
Ensemble methods	Average out variance

Solutions for Underfitting

Technique	How It Helps
More features	Give model more information
More complex model	Increase capacity
Less regularization	Let model be more flexible
Feature engineering	Create better features
Polynomial features	Capture non-linear patterns

Part 2: Cross-Validation

Getting Reliable Performance Estimates

The Problem with Train/Test Split

```python
from sklearn.model_selection import train_test_split

# Try different random seeds
for seed in [1, 2, 3, 4, 5]:
    X_train, X_test, y_train, y_test = train_test_split(
        X, y, test_size=0.2, random_state=seed)
    model.fit(X_train, y_train)
    print(f"Seed {seed}: {model.score(X_test, y_test):.2%}")
```

```
Seed 1: 87%
Seed 2: 92%  ← Which one should we believe?
Seed 3: 84%
Seed 4: 90%
Seed 5: 88%
```

The Lucky/Unlucky Split Problem

What if your test set happened to contain:

Scenario	Accuracy	Reality
Easy examples only	95%	Overly optimistic
Hard examples only	80%	Overly pessimistic
Representative sample	88%	Realistic

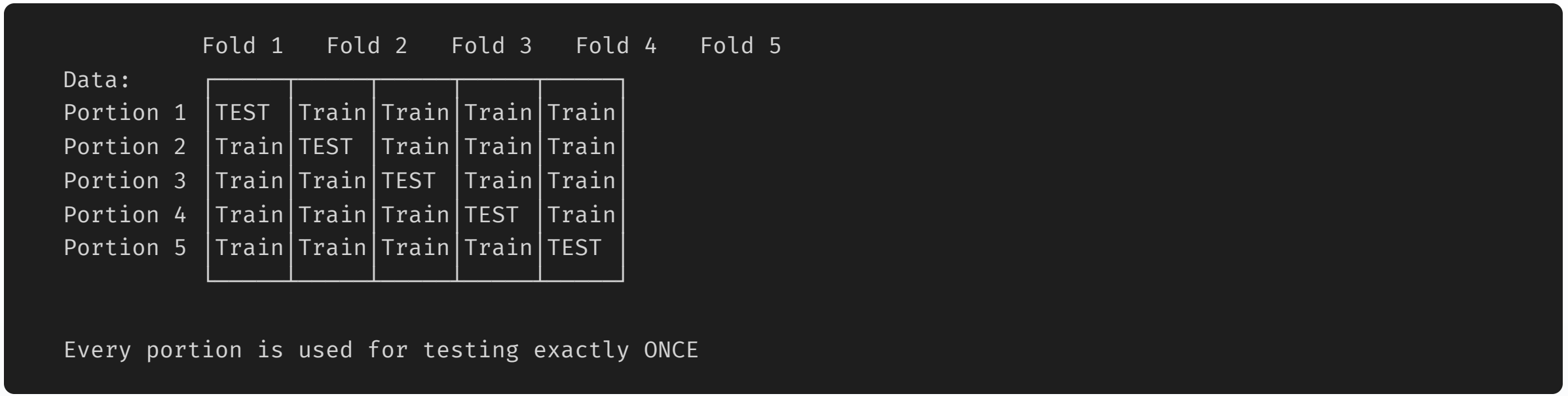
One split = One roll of the dice

K-Fold Cross-Validation

Idea: Use EVERY data point for both training AND testing!

```
Data: [1][2][3][4][5][6][7][8][9][10]

Fold 1: [TEST 1-2] [TRAIN 3-10] → Score: 0.87
Fold 2: [TRAIN 1-2] [TEST 3-4] [TRAIN 5-10] → Score: 0.89
Fold 3: [TRAIN 1-4] [TEST 5-6] [TRAIN 7-10] → Score: 0.91
Fold 4: [TRAIN 1-6] [TEST 7-8] [TRAIN 9-10] → Score: 0.88
Fold 5: [TRAIN 1-8] [TEST 9-10] → Score: 0.90

Final Score: 0.89 ± 0.01
```

Visual: 5-Fold Cross-Validation



K-Fold in sklearn

```python
from sklearn.model_selection import cross_val_score
from sklearn.tree import DecisionTreeClassifier

model = DecisionTreeClassifier(max_depth=5)

# 5-fold cross-validation
scores = cross_val_score(model, X, y, cv=5)

print(f"Fold scores: {scores}")
# [0.87, 0.89, 0.91, 0.88, 0.90]

print(f"Mean: {scores.mean():.3f}")  # 0.890
print(f"Std:  {scores.std():.3f}")   # 0.015
```

Interpreting Cross-Validation Results

```
scores = [0.87, 0.89, 0.91, 0.88, 0.90]
```

Metric	Value	Meaning
Mean	0.890	Expected performance
Std	0.015	Model stability
Min	0.87	Worst case
Max	0.91	Best case

Low std = Stable model

High std = Model performance varies a lot

Choosing K

K	Name	Pros	Cons
5	5-Fold	Fast, standard	Less reliable
10	10-Fold	More reliable	Slower
n	Leave-One-Out	Uses all data	Very slow

Rule of thumb:

- Small dataset (< 1000): Use 10-fold
- Medium dataset: Use 5-fold
- Large dataset (> 10000): Even 3-fold is fine

Stratified K-Fold

Problem: What if your classes are imbalanced?

```
# Original data: 900 class 0, 100 class 1
# Random fold might get: 200 class 0, 0 class 1  ← Bad!
```

Solution: Stratified K-Fold maintains class proportions

```
from sklearn.model_selection import cross_val_score, StratifiedKFold

skf = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
scores = cross_val_score(model, X, y, cv=skf)
```

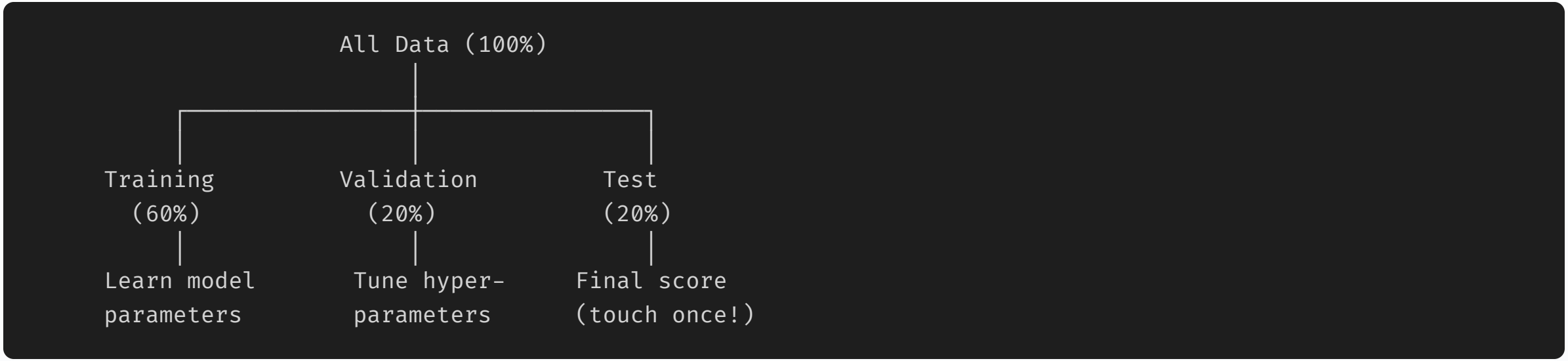
Comparing Models with Cross-Validation

```
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier

models = {
    'Logistic': LogisticRegression(),
    'Tree': DecisionTreeClassifier(max_depth=5),
    'KNN': KNeighborsClassifier(n_neighbors=5)
}

for name, model in models.items():
    scores = cross_val_score(model, X, y, cv=5)
    print(f"{name}: {scores.mean():.3f} ± {scores.std():.3f}")
```

```
Logistic: 0.850 ± 0.020
Tree:     0.890 ± 0.015  ← Winner
KNN:      0.870 ± 0.025
```

The Three-Way Split

For model selection AND final evaluation:



Why Three Sets?

Set	Purpose	When to Use
Training	Learn model parameters	During fit()
Validation	Choose hyperparameters, compare models	During tuning
Test	Final evaluation	Only at the very end!

Never tune on test set! It defeats the purpose of having one.

Part 3: Hyperparameter Tuning

Finding the Best Model Settings

Parameters vs Hyperparameters

Type	Set By	When	Example
Parameters	Algorithm	During training	Weights in regression
Hyperparameters	You	Before training	max_depth, n_neighbors

```
# Hyperparameter (you choose)
model = DecisionTreeClassifier(max_depth=5)

# Parameters (learned automatically)
model.fit(X, y)
print(model.tree_.feature)  # Learned split features
```

Common Hyperparameters

Model	Hyperparameter	What It Controls	Range
Decision Tree	`max_depth`	Tree depth	1 to 20+
Decision Tree	`min_samples_leaf`	Leaf size	1 to 50
K-NN	`n_neighbors`	Neighbors to use	1 to 50
Logistic Reg	`C`	Regularization	0.001 to 100
Random Forest	`n_estimators`	Number of trees	10 to 500

Manual Tuning: Trial and Error

```python
# Try different max_depth values
for depth in [3, 5, 7, 10, 15]:
    model = DecisionTreeClassifier(max_depth=depth)
    scores = cross_val_score(model, X, y, cv=5)
    print(f"depth={depth}: {scores.mean():.3f}")
```

```
depth=3:  0.820
depth=5:  0.870
depth=7:  0.890  ← Best
depth=10: 0.880
depth=15: 0.840  ← Overfitting starts
```

Works but tedious for multiple hyperparameters!

Grid Search: Automated Tuning

Try ALL combinations systematically:

max_depth	min_samples_leaf	Score
3	1	0.82
3	5	0.83
3	10	0.81
5	1	0.87
5	5	0.88
5	**10**	**0.89** ← Best
7	1	0.86
...	...	...

GridSearchCV in sklearn

```
from sklearn.model_selection import GridSearchCV
from sklearn.tree import DecisionTreeClassifier

# Define the grid of values to try
param_grid = {
    'max_depth': [3, 5, 7, 10],
    'min_samples_leaf': [1, 5, 10, 20]
}

# Create and run the search
grid_search = GridSearchCV(
    DecisionTreeClassifier(),
    param_grid,
    cv=5,               # 5-fold cross-validation
    scoring='accuracy'  # Metric to optimize
)
grid_search.fit(X_train, y_train)
```

GridSearchCV Results

```python
# Best hyperparameters
print(f"Best params: {grid_search.best_params_}")
# {'max_depth': 5, 'min_samples_leaf': 10}

# Best cross-validation score
print(f"Best CV score: {grid_search.best_score_:.3f}")
# 0.890

# Get the best model (already fitted!)
best_model = grid_search.best_estimator_

# Evaluate on test set
test_score = best_model.score(X_test, y_test)
print(f"Test score: {test_score:.3f}")
# 0.885
```

Viewing All Results

```
import pandas as pd

results = pd.DataFrame(grid_search.cv_results_)
print(results[['param_max_depth', 'param_min_samples_leaf',
               'mean_test_score', 'rank_test_score']])
```

```
   param_max_depth  param_min_samples_leaf  mean_test_score  rank_test_score
0                3                       1            0.820               12
1                3                       5            0.830               10
2                3                      10            0.810               14
3                5                       1            0.870                6
4                5                       5            0.880                3
5                5                      10            0.890                1  ← Best
...
```

The Explosion Problem

Grid search with 5 hyperparameters, 10 values each:

$$10 \times 10 \times 10 \times 10 \times 10 = 100,000 \text{ combinations!}$$

With 5-fold CV: 500,000 model trainings!

Combinations	Time (1 sec/train)
100	8 minutes
1,000	1.4 hours
10,000	14 hours
100,000	6 days

Random Search: Smart Sampling

Idea: Don't try everything, sample randomly!

```
from sklearn.model_selection import RandomizedSearchCV
from scipy.stats import randint

param_dist = {
    'max_depth': randint(1, 20),           # Random 1-20
    'min_samples_leaf': randint(1, 50),    # Random 1-50
    'min_samples_split': randint(2, 50)    # Random 2-50
}

random_search = RandomizedSearchCV(
    DecisionTreeClassifier(),
    param_dist,
    n_iter=50,    # Only try 50 random combinations
    cv=5,
    random_state=42
)
random_search.fit(X_train, y_train)
```

Grid vs Random Search

Aspect	Grid Search	Random Search
Coverage	All combinations	Sample of space
Time	Slow for large grids	Controllable (n_iter)
Continuous params	Must discretize	Can sample directly
Best for	Few params	Many params

> Random search often finds good solutions faster than grid search!

Practical Tuning Strategy

Step	What to Do
1. Baseline	Train with defaults, get baseline score
2. Coarse search	Random search with wide ranges
3. Fine search	Grid search around best area
4. Final eval	Test on held-out test set

```
# Start wide
RandomizedSearchCV(model, wide_params, n_iter=100)

# Then narrow down
GridSearchCV(model, narrow_params, cv=5)
```

Part 4: Ensemble Methods

The Wisdom of Crowds

Why Ensembles Work

One person guessing jar of marbles: Often wrong

1000 people averaging guesses: Usually close!

Individual	Guess
Person 1	520
Person 2	680
Person 3	450
...	...
Average	**503**
Actual	**500**

Errors cancel out!

The Same Idea in ML

One decision tree: May overfit, unstable

100 decision trees averaged: Robust, accurate

Single Model	Ensemble
High variance	Reduced variance
Biased view	Diverse perspectives
One mistake ruins it	Errors average out
Unstable	Stable

Two Ensemble Strategies

Strategy	Key Idea	How It Reduces Error
Bagging	Average many parallel models	Reduces variance
Boosting	Build models sequentially	Reduces bias

Bagging: Bootstrap Aggregating

Steps:

1. **Bootstrap:** Create random subsets of data (with replacement)
2. **Train:** Fit a model on each subset
3. **Aggregate:** Average predictions (regression) or vote (classification)

```
Original Data: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

Bootstrap 1: [1, 1, 3, 4, 6, 6, 7, 9, 9, 10] → Model 1
Bootstrap 2: [2, 2, 3, 5, 5, 6, 8, 8, 9, 10] → Model 2
Bootstrap 3: [1, 3, 4, 4, 5, 7, 7, 8, 10, 10] → Model 3

Final: Average(Model 1, Model 2, Model 3)
```

Random Forest

Bagging + Decision Trees + Feature Randomization

Component	What It Does
Many trees	Reduce variance
Random samples	Different perspectives
Random features	Decorrelate trees
Voting/Averaging	Combine predictions

```
from sklearn.ensemble import RandomForestClassifier

model = RandomForestClassifier(
    n_estimators=100,    # 100 trees
    max_depth=10,        # Limit tree depth
    random_state=42
)
model.fit(X_train, y_train)
```

Why Random Forest Works

```
Tree 1: Uses features [A, B, C] → Predicts: Class 1
Tree 2: Uses features [B, D, E] → Predicts: Class 0
Tree 3: Uses features [A, C, F] → Predicts: Class 1
Tree 4: Uses features [D, E, F] → Predicts: Class 1
 ...
Tree 100: Uses features [A, E, G] → Predicts: Class 1

Vote: 70 trees say Class 1, 30 say Class 0
Final Prediction: Class 1
```

Each tree makes different mistakes → Average is better!

Random Forest: Key Hyperparameters

Parameter	What It Does	Good Values
`n_estimators`	Number of trees	100-500
`max_depth`	Tree depth	10-30 or None
`min_samples_leaf`	Min samples in leaf	1-10
`max_features`	Features per split	'sqrt' or 'log2'

```
model = RandomForestClassifier(
    n_estimators=200,
    max_depth=15,
    max_features='sqrt',
    random_state=42
)
```

Feature Importance from Random Forest

```
import pandas as pd

# Train model
model.fit(X_train, y_train)

# Get importance
importance = pd.DataFrame({
    'feature': feature_names,
    'importance': model.feature_importances_
}).sort_values('importance', ascending=False)

print(importance.head(10))
```

```
       feature  importance
0       income        0.25
1          age        0.18
2   credit_score      0.15
...
```

Boosting: Learning from Mistakes

Key Idea: Each model focuses on what previous models got wrong.

```
Round 1: Train model on all data
         Some examples misclassified (X)

Round 2: Train new model, weight misclassified X higher
         Fixes some errors, might make new ones

Round 3: Focus even more on remaining errors
          ...

Final: Combine all models (weighted vote)
```

Gradient Boosting

```
from sklearn.ensemble import GradientBoostingClassifier

model = GradientBoostingClassifier(
    n_estimators=100,    # Number of boosting rounds
    learning_rate=0.1,   # Step size (smaller = slower but better)
    max_depth=3,         # Shallow trees work best
    random_state=42
)
model.fit(X_train, y_train)
```

XGBoost: The Competition Winner

```
# pip install xgboost
from xgboost import XGBClassifier

model = XGBClassifier(
    n_estimators=100,
    learning_rate=0.1,
    max_depth=5,
    random_state=42
)
model.fit(X_train, y_train)

# Often wins Kaggle competitions!
```

XGBoost = Extreme Gradient Boosting

Faster, more regularized, handles missing values.

Bagging vs Boosting

Aspect	Bagging	Boosting
Goal	Reduce variance	Reduce bias
Training	Parallel (fast)	Sequential (slower)
Trees	Independent	Build on each other
Overfitting risk	Low	Higher
Example	Random Forest	XGBoost, LightGBM
When to use	High variance model	High bias model

Ensemble Comparison

Model	Speed	Accuracy	Interpretable?
Decision Tree	Fast	Medium	Yes
Random Forest	Medium	High	Somewhat
XGBoost	Slow	Very High	No
LightGBM	Fast	Very High	No

Start with Random Forest - great out of the box.
Use XGBoost/LightGBM when you need maximum accuracy.

Practical Example

```
from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier
from xgboost import XGBClassifier

models = {
    'Random Forest': RandomForestClassifier(n_estimators=100),
    'Gradient Boosting': GradientBoostingClassifier(n_estimators=100),
    'XGBoost': XGBClassifier(n_estimators=100)
}

for name, model in models.items():
    scores = cross_val_score(model, X, y, cv=5)
    print(f"{name}: {scores.mean():.3f} ± {scores.std():.3f}")
```

```
Random Forest:     0.925 ± 0.015
Gradient Boosting: 0.932 ± 0.012
XGBoost:           0.938 ± 0.010  ← Best
```

Part 5: Unsupervised Learning

Learning Without Labels

The Unsupervised Problem

Supervised: You have X (features) and y (labels)

Unsupervised: You only have X (no labels!)

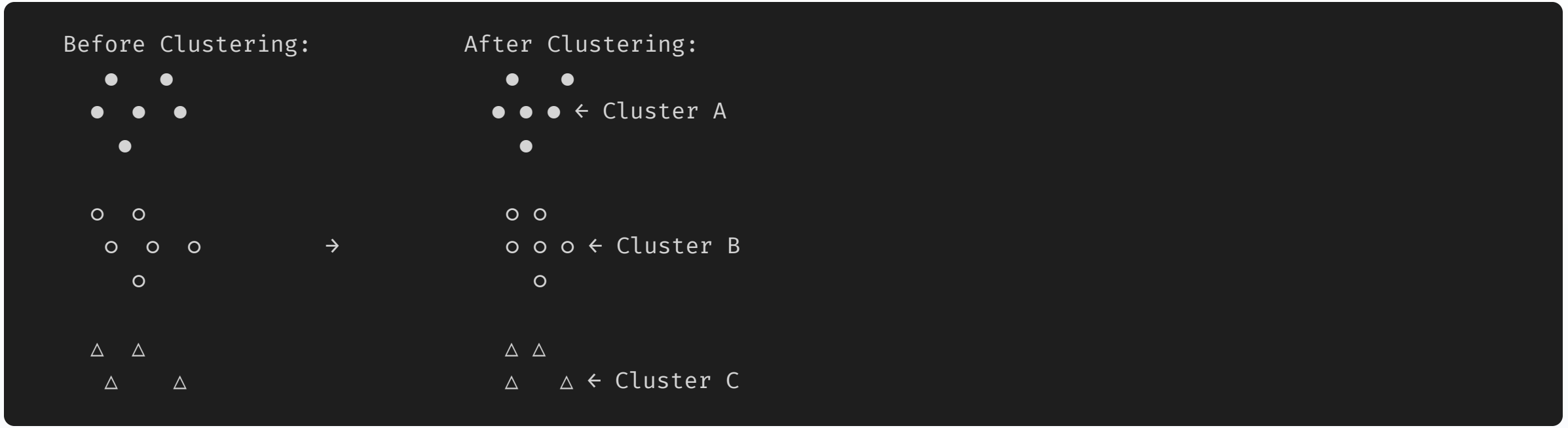
Supervised	Unsupervised
Email → Spam/Not Spam	Email → ???
Image → Cat/Dog	Customers → ???
Price history → Next price	Similar items → ???

Why Unsupervised?

Use Case	What You Want
Customer Segmentation	Group similar customers
Anomaly Detection	Find unusual patterns
Data Compression	Reduce dimensions
Visualization	Plot high-dimensional data
Feature Learning	Discover useful features

Clustering: Finding Groups

Goal: Group similar items together automatically.

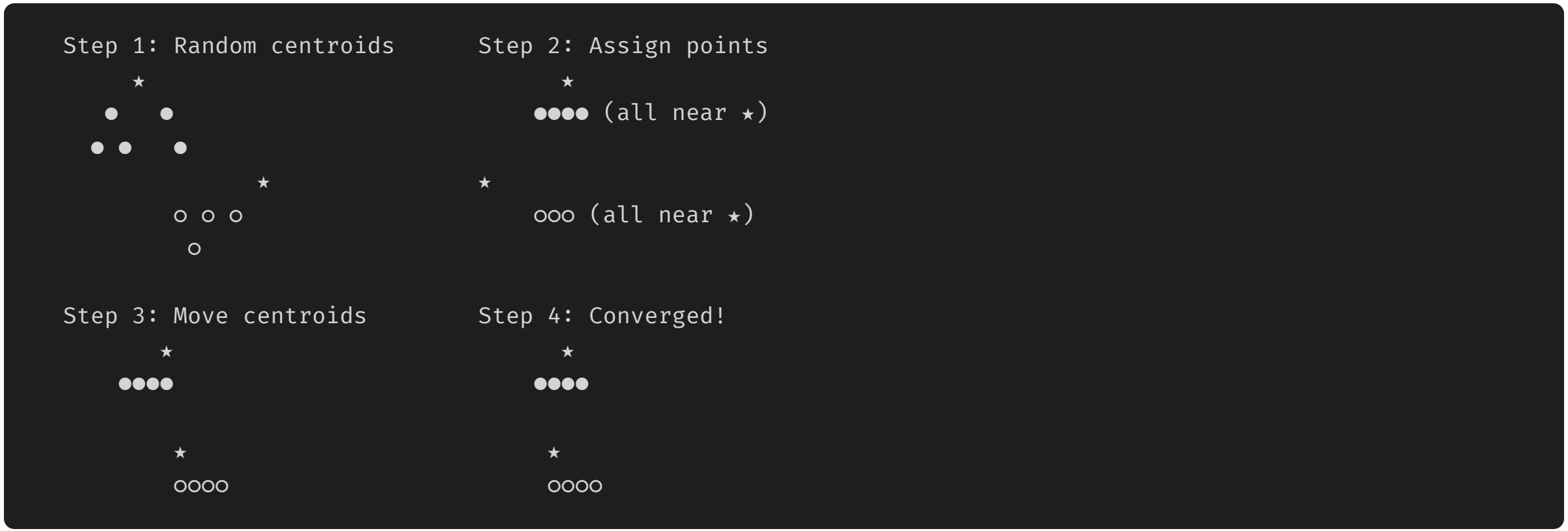


K-Means: The Most Popular

Algorithm:

Step	What Happens
1. Initialize	Place K random centroids
2. Assign	Each point → nearest centroid
3. Update	Move centroids to cluster mean
4. Repeat	Until centroids stop moving

K-Means Visualization



K-Means in sklearn

```
from sklearn.cluster import KMeans

# Create and fit
model = KMeans(
    n_clusters=3,        # How many clusters?
    random_state=42
)
model.fit(X)

# Get results
labels = model.labels_           # Which cluster is each point in?
centers = model.cluster_centers_ # Where are the centers?

# Assign new points
new_labels = model.predict(X_new)
```

Choosing K: The Elbow Method

Plot "inertia" (within-cluster sum of squares) vs K:

```
inertias = []
for k in range(1, 11):
    model = KMeans(n_clusters=k, random_state=42)
    model.fit(X)
    inertias.append(model.inertia_)

plt.plot(range(1, 11), inertias, marker='o')
plt.xlabel('K (number of clusters)')
plt.ylabel('Inertia')
```

Look for the "elbow" - where improvement slows down

Elbow Plot

After the elbow, adding more clusters doesn't help much.

Dimensionality Reduction

Problem: Can't visualize 100 features!

Solution: Reduce to 2-3 dimensions while preserving structure.

Method	Type	Best For
PCA	Linear	General use, fast
t-SNE	Non-linear	Visualization
UMAP	Non-linear	Visualization + structure

PCA: Principal Component Analysis

Idea: Find directions of maximum variance.

```
from sklearn.decomposition import PCA

# Reduce 100 dimensions → 2 dimensions
pca = PCA(n_components=2)
X_2d = pca.fit_transform(X)  # X was 100D, now 2D

# Visualize
plt.scatter(X_2d[:, 0], X_2d[:, 1], c=labels)
plt.xlabel('PC1')
plt.ylabel('PC2')
```

How Much Variance Explained?

```
pca = PCA(n_components=10)
pca.fit(X)

# How much does each component explain?
print(pca.explained_variance_ratio_)
# [0.35, 0.20, 0.12, 0.08, 0.06, 0.05, 0.04, 0.03, 0.02, 0.02]

# Cumulative
import numpy as np
print(np.cumsum(pca.explained_variance_ratio_))
# [0.35, 0.55, 0.67, 0.75, 0.81, 0.86, 0.90, 0.93, 0.95, 0.97]

# 10 components explain 97% of the variance!
```

t-SNE: Better Visualization

```
from sklearn.manifold import TSNE

# Better for visualization (preserves local structure)
tsne = TSNE(
    n_components=2,
    perplexity=30,      # Balance local/global
    random_state=42
)
X_2d = tsne.fit_transform(X)

plt.scatter(X_2d[:, 0], X_2d[:, 1], c=labels)
```

t-SNE often reveals clusters that PCA misses!

PCA vs t-SNE

Aspect	PCA	t-SNE
Type	Linear	Non-linear
Speed	Fast	Slow
Interpretable	Yes (axes have meaning)	No
Best for	Preprocessing, reduction	Visualization
New data	Can transform	Need to refit

Anomaly Detection

Goal: Find the "odd ones out"

```
from sklearn.ensemble import IsolationForest

# Detect anomalies
model = IsolationForest(
    contamination=0.05,  # Expect 5% anomalies
    random_state=42
)
model.fit(X)

# Predict: 1 = normal, -1 = anomaly
predictions = model.predict(X)
anomalies = X[predictions == -1]
```

Summary Table

Task	No Labels	With Labels
Grouping	K-Means, DBSCAN	-
Visualization	PCA, t-SNE	-
Outlier detection	Isolation Forest	-
Prediction	-	All supervised methods

Putting It All Together

Step	What to Do	Tools
1	Start with simple baseline	LogisticRegression, DecisionTree
2	Evaluate with cross-validation	cross_val_score
3	Tune hyperparameters	GridSearchCV, RandomizedSearchCV
4	Try ensembles	RandomForest, XGBoost
5	Final evaluation on test set	accuracy_score, etc.

Key Takeaways

1. **Bias-Variance Tradeoff:** Can't minimize both - find the balance
2. **Cross-Validation:** Always use it - single split is unreliable
3. **Hyperparameter Tuning:** Grid search for small spaces, random for large
4. **Ensembles:** Combine models for better performance
    - Bagging (Random Forest) reduces variance
    - Boosting (XGBoost) reduces bias
5. **Unsupervised:** Clustering and dimensionality reduction when no labels

You Can Now Select & Tune Models!

Next: Neural Networks - Deep Learning Begins

Lab: Cross-validation, hyperparameter tuning, Random Forest

"Ensemble methods: Because two heads are better than one!"

Questions?