Supervised Learning Deep Dive

From Linear Regression to Decision Trees

Nipun Batra | IIT Gandhinagar

Learning Goals

By the end of this lecture, you will:

Goal	What You'll Learn
Understand	How 4 core ML algorithms work
Implement	Linear/Logistic Regression, Trees, K-NN
Evaluate	Accuracy, Precision, Recall, F1, MSE, R^2
Choose	Which algorithm for which problem
Interpret	Model outputs and predictions

Recap: Supervised Learning

Input: Features (X) + Labels (y)

Goal: Learn function f where f(X) ≈ y

```
    Training Data                Model                 Prediction

   ┌──────────────┐          ┌──────────┐          ┌──────────────┐
   │ X (features)│          │          │          │              │
   │ +            │  ──────
 ▶
    │  f(X) = y │  ──────
 ▶
    │  Prediction │
   │ y (labels)  │   train  │          │   infer  │              │
   └──────────────┘          └──────────┘          └──────────────┘
```

Two Types of Supervised Learning

If y is...	Task	Output	Example
Category	Classification	Class label	Spam or Not
Number	Regression	Continuous value	House Price

The **type of label** determines the **type of problem**.

Today's Algorithm Menu

Algorithm	Type	Key Idea	Best For
Linear Regression	Regression	Fit a line	Linear relationships
Logistic Regression	Classification	Probability + threshold	Binary classification
Decision Trees	Both	If-then rules	Interpretable models
K-Nearest Neighbors	Both	Vote by neighbors	Simple patterns

Part 1: Linear Regression

The Line of Best Fit

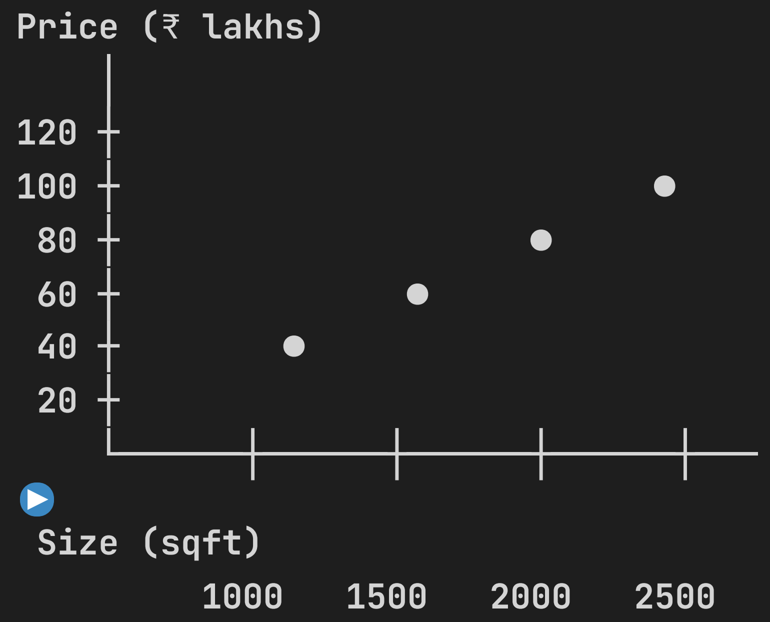
The Simplest ML Model

Question: Given house size, predict price?

Size (sqft)	Price (₹ lakhs)
1000	40
1500	60
2000	80
2500	100

What would a 1750 sqft house cost?

Spotting the Pattern



Observation: Points fall on a line! Price = 0.04 × Size

Linear Regression: The Idea

Find the **best fitting line** through the data.

$$\hat{y} = wx + b$$

Symbol	Name	Meaning	Example
x	Input	Feature value	1500 sqft
$\hat{y}$	Output	Predicted value	₹60 lakhs
w	Weight	Slope (sensitivity)	0.04
b	Bias	Intercept (baseline)	0

The Slope Intuition

What does the slope (w) mean?

w = 0.04	Interpretation
Per unit change in x	y changes by w
+100 sqft	+₹4 lakhs
+500 sqft	+₹20 lakhs

The weight tells you sensitivity: How much does output change when input changes?

The Intercept Intuition

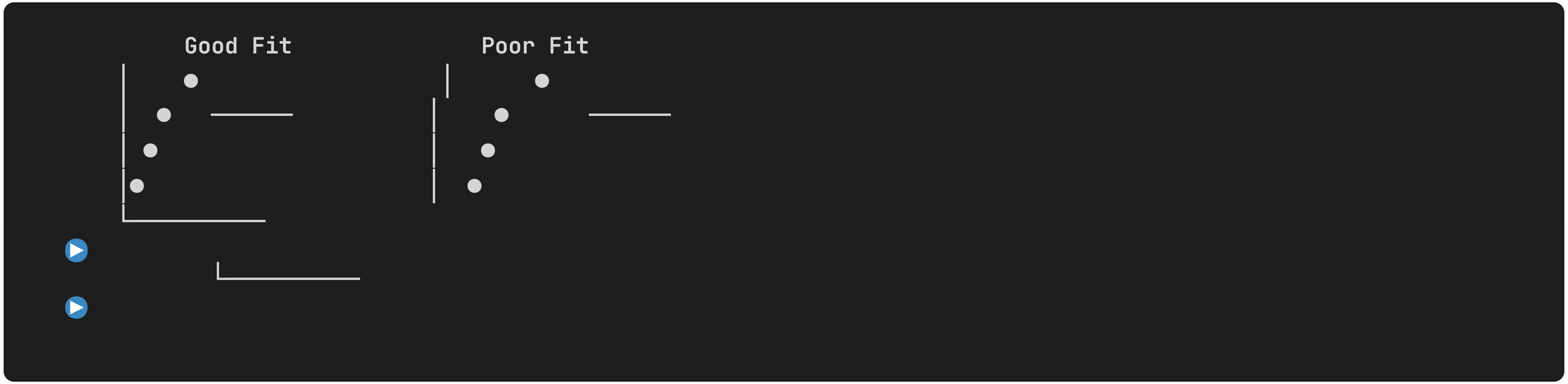
What does the bias (b) mean?

b = 0	b = 10
0 sqft → ₹0	0 sqft → ₹10 lakhs
Line passes through origin	Line shifted up by 10

Real-world: Intercept captures the "baseline" - minimum cost regardless of size (land, permits, etc.)

What is "Best Fitting"?

Problem: Many lines can pass through/near the points.



Question: How do we define "best"?

Measuring Error: Residuals

Residual = Actual - Predicted = $y - \hat{y}$

```
    Price
      │
      │         ● actual
      │         │ ← residual (error)
      │         ▼
      │     ────●──── predicted (on line)
      │    ●
      └────────────────
    ▶️
     Size
```

Goal: Make residuals as small as possible!

Why Squared Errors?

We minimize **Sum of Squared Errors (SSE)**:

$$\text{SSE} = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

Property	Why It Helps
Always positive	Errors don't cancel (+3 and -3)
Penalizes big errors	Error of 10 costs 100, not 10
Differentiable	Can use calculus to find minimum
Closed-form solution	Can solve directly with math

Visualizing SSE

```
             Small SSE                Large SSE
        │   ●                    │         ●
        │  ●  ──────            │    ●    ──────
        │ ●                      │   ●
        │●                       │  ●
        └──────────
    ▶️
                 └───────────
    ▶️


        Squares are small         Squares are big
        (close to line)           (far from line)
```

Best line minimizes total area of squares!

Finding the Best Line

The math (don't memorize!):

$$w = \frac{\sum(x_i - \bar{x})(y_i - \bar{y})}{\sum(x_i - \bar{x})^2}$$

$$b = \bar{y} - w\bar{x}$$

Term	Meaning
$\bar{x}$	Mean of all x values
$\bar{y}$	Mean of all y values

sklearn does this for you automatically!

Linear Regression in sklearn

```python
from sklearn.linear_model import LinearRegression
import numpy as np

# Data: Size (sqft) → Price (lakhs)
X = np.array([[1000], [1500], [2000], [2500]])
y = np.array([40, 60, 80, 100])

# Create and train model
model = LinearRegression()
model.fit(X, y)

# Predict
model.predict([[1750]])  # → 70.0 (₹70 lakhs)
```

Understanding the Learned Model

```
print(f"Weight (w): {model.coef_[0]}")      # 0.04
print(f"Intercept (b): {model.intercept_}")  # 0.0

# The learned equation:
# Price = 0.04 × Size + 0
# For 1750 sqft: 0.04 × 1750 = ₹70 lakhs
```

Attribute	Meaning	Value
`model.coef_`	Weights (one per feature)	[0.04]
`model.intercept_`	Bias term	0.0

Multiple Features

What if we have more than one feature?

$$\hat{y} = w_1x_1 + w_2x_2 + w_3x_3 + b$$

```
# Features: [sqft, bedrooms, bathrooms]
X = [[1500, 3, 2],
     [2000, 4, 3],
     [1200, 2, 1],
     [1800, 3, 2]]
y = [60, 90, 45, 75]

model.fit(X, y)
print(model.coef_)  # → [w_sqft, w_beds, w_baths]
```

Interpreting Multiple Weights

```
# Example output:
# coef_ = [0.03, 5.0, 8.0]
# intercept_ = -10
```

Feature	Weight	Interpretation
sqft	0.03	+100 sqft → +₹3 lakhs
bedrooms	5.0	+1 bedroom → +₹5 lakhs
bathrooms	8.0	+1 bathroom → +₹8 lakhs

Each weight shows feature's contribution to price!

When Linear Regression Works

Scenario	Works?	Why
Linear relationship	✅	Points follow a line
Continuous target	✅	Predicting real numbers
Independent features	✅	No multicollinearity

```
✅
 Works Well
❌
 Doesn't Work
   │     ● ●                  │    ●      ●
   │   ● ●                    │       ●
   │  ● ●                     │   ●     ●
   │ ●                        │ ●         ●
   └──────────
▶
             └──────────

▶


   Linear pattern              Curved pattern
```

When Linear Regression Fails

Problem	Example	Solution
Curved patterns	Diminishing returns	Polynomial features
Outliers	One mansion in data	Robust regression
Discrete targets	Pass/Fail	Use classification
Non-linear relationships	Complex interactions	Trees, Neural Nets

Polynomial Features

For curved relationships:

```
from sklearn.preprocessing import PolynomialFeatures

# Original: [x]
# After poly(degree=2): [1, x, x²]

poly = PolynomialFeatures(degree=2)
X_poly = poly.fit_transform(X)

model = LinearRegression()
model.fit(X_poly, y)  # Now fits curves!
```

Part 2: Logistic Regression

Predicting Categories

The Classification Problem

Question: Given email features, is it spam?

num_exclamations	has_FREE	contains_offer	is_spam
5	Yes	Yes	✅ Spam
0	No	No	❌ Not Spam
3	Yes	Yes	✅ Spam
1	No	No	❌ Not Spam

Output is a category, not a number!

Why Not Linear Regression?

Problem: Linear regression predicts any number (-∞ to +∞)

```
Linear Regression Output:
-2.5, -1.0, 0.3, 0.7, 1.5, 2.8, ...

But we need:
0 (not spam) or 1 (spam)
```

Issue	Example
Predictions outside [0,1]	"Probability = 1.5" ❌
No clear decision boundary	When to say spam?

The Sigmoid Function

Solution: Squash any number to range (0, 1)

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

Input (z)	Output σ(z)	Interpretation
-10	0.00005	Very low probability
-2	0.12	Low probability
0	0.50	50-50
+2	0.88	High probability
+10	0.99995	Very high probability

Sigmoid Visualization

```
P(spam)
  │                          ●●●● 1.0
1.0┼                    ●●●●
   │                ●●●
0.5┼───────────●●●──────────────
   │       ●●●
0.0┼●●●●●●
   └────┼────┼────┼────┼────┼───
▶️
 z = wx + b
       -4   -2    0    2    4

       Definitely      Could be     Definitely
       NOT spam        either       SPAM
```

Logistic Regression Model

$$P(\text{spam}|x) = \sigma(wx + b) = \frac{1}{1 + e^{-(wx+b)}}$$

Two-step process:

Step	Operation	Example
1. Linear	z = wx + b	z = 0.5×5 + (-1) = 1.5
2. Sigmoid	P = σ(z)	P = σ(1.5) = 0.82

Output: 82% probability of spam

The Decision Rule

Threshold: Usually 0.5

If P(spam)	Decision
> 0.5	Predict SPAM
≤ 0.5	Predict NOT SPAM

```
P(spam)
  │
1.0┼  ← Definitely Spam
   │
0.5┼───────────────────────── Threshold
   │
0.0┼  ← Definitely Not Spam
```

Logistic Regression in sklearn

```python
from sklearn.linear_model import LogisticRegression

# Data: [num_exclamations, has_FREE]
X = [[5, 1], [0, 0], [3, 1], [1, 0], [4, 1], [0, 0]]
y = [1, 0, 1, 0, 1, 0]  # 1=spam, 0=not spam

# Train
model = LogisticRegression()
model.fit(X, y)

# Predict class
model.predict([[4, 1]])  # → [1] (spam)

# Predict probability
model.predict_proba([[4, 1]])  # → [[0.12, 0.88]]
#                                    [P(not spam), P(spam)]
```

Understanding predict_proba

```
probs = model.predict_proba([[4, 1]])
# → [[0.12, 0.88]]

print(f"P(not spam) = {probs[0][0]:.2f}")  # 0.12
print(f"P(spam) = {probs[0][1]:.2f}")      # 0.88
```

Class	Index	Probability
Not Spam (0)	probs[0][0]	12%
Spam (1)	probs[0][1]	88%

Always sums to 1.0!

Decision Boundary

Logistic regression learns a **linear decision boundary**:

```
has_FREE
    │
  1 ┼ ● ● ●│       ● = spam
    │      │       × = not spam
    │      │
  0 ┼ × × ×│
    └──────┼─────────
           │
▶
 num_exclamations
           2.5
           ↑
        Decision boundary
```

Where: $wx + b = 0$ ($P = 0.5$)

Interpreting Weights

```python
print(f"Weights: {model.coef_}")      # [[0.8, 2.1]]
print(f"Intercept: {model.intercept_}") # [-1.5]
```

Feature	Weight	Effect on P(spam)
num_exclamations	0.8	More ! → Higher spam prob
has_FREE	2.1	"FREE" present → Much higher spam prob

Positive weight → increases spam probability

Negative weight → decreases spam probability

Multi-class Classification

What if more than 2 classes? (dog, cat, bird)

```
model = LogisticRegression(multi_class='multinomial')
model.fit(X, y)

# predict_proba gives probability for each class
probs = model.predict_proba([[features]])
# → [[0.1, 0.7, 0.2]]  = [P(dog), P(cat), P(bird)]
```

Softmax: Generalizes sigmoid to multiple classes.

Logistic vs Linear Regression

Aspect	Linear Regression	Logistic Regression
Output	Any number	Probability [0, 1]
Task	Regression	Classification
Loss function	MSE	Cross-entropy
Decision	Direct value	Threshold

Despite the name, Logistic Regression is for **classification**, not regression!

Part 3: Decision Trees

Rule-Based Learning

The Most Intuitive Model

How humans make decisions:

```
Should I carry an umbrella?
         │
    Is it cloudy?
    /          \
  Yes            No
   │              │
Is P(rain)>50%?  Don't carry
  /      \
Yes       No
 │         │
Carry    Don't
```

Decision Trees learn these rules from data!

A Real Example

Task: Predict if someone will buy a product

Age	Income	Student	Buys
Young	High	No	No
Young	High	Yes	Yes
Middle	High	No	Yes
Senior	Medium	No	Yes
Senior	Low	Yes	No

The Learned Tree

```
                  [Income?]
                 /    |    \
             High   Medium   Low
              /       |        \
         [Age?]     [Yes]     [No]
        /     \
    Young   Middle/Senior
      /           \
 [Student?]      [Yes]
   /    \
 Yes     No
  |       |
[Yes]   [No]
```

Decision Tree for Email Spam

```
       [Start]
          │
 "FREE" in subject?
    /         \
  Yes          No
   │            │
[SPAM]    Exclamation marks > 3?
              /      \
            Yes       No
             │         │
       [SPAM]     Has attachment?
                    /      \
                  Yes       No
                   │         │
               [Check]   [Not Spam]
```

How Trees Learn: The Key Question

At each node: Which feature creates the "purest" split?

```
Before Split:           After Split (Good):
┌─────────────┐        ┌───────┐  ┌───────┐
│ ● ● ● ○ ○ ○│  ─
▶
   │ ● ● ●│  │ ○ ○ ○│
│   (mixed)   │        │(pure) │  │(pure) │
└─────────────┘        └───────┘  └───────┘


                       After Split (Bad):
                       ┌───────┐  ┌───────┐
                    ─
▶
  │ ● ● ○│  │ ○ ○ ●│
                       │(mixed)│  │(mixed)│
                       └───────┘  └───────┘
```

Measuring Purity: Gini Impurity

$$\text{Gini} = 1 - \sum_i p_i^2$$

Node Composition	Gini	Purity
All same class (100% A)	0.0	Perfect
90% A, 10% B	0.18	Very pure
50% A, 50% B	0.50	Maximum impurity

Lower Gini = Better split!

Example: Calculating Gini

Node with 8 spam, 2 not-spam:

$$\text{Gini} = 1 - (0.8)^2 - (0.2)^2 = 1 - 0.64 - 0.04 = 0.32$$

Node with 5 spam, 5 not-spam:

$$\text{Gini} = 1 - (0.5)^2 - (0.5)^2 = 1 - 0.25 - 0.25 = 0.50$$

First node is purer!

Information Gain (Alternative)

$$\text{Entropy} = -\sum_i p_i \log_2(p_i)$$

Concept	Intuition
Entropy	Surprise/uncertainty in data
Information Gain	Reduction in entropy after split

Higher Information Gain = Better split!

Decision Tree in sklearn

```python
from sklearn.tree import DecisionTreeClassifier

# Data
X = [[5, 1], [0, 0], [3, 1], [1, 0], [4, 1], [2, 0]]
y = [1, 0, 1, 0, 1, 0]  # spam or not

# Train
model = DecisionTreeClassifier(max_depth=3)
model.fit(X, y)

# Predict
model.predict([[4, 1]])  # → [1] (spam)
```

Visualizing the Tree

```python
from sklearn.tree import plot_tree
import matplotlib.pyplot as plt

plt.figure(figsize=(15, 10))
plot_tree(model,
          feature_names=['num_exclaim', 'has_FREE'],
          class_names=['Not Spam', 'Spam'],
          filled=True,        # Color by class
          rounded=True,       # Rounded boxes
          fontsize=12)
plt.tight_layout()
plt.savefig('tree_visualization.png')
```

Feature Importance

```python
# Which features matter most?
importance = model.feature_importances_

for name, imp in zip(['num_exclaim', 'has_FREE'], importance):
    print(f"{name}: {imp:.3f}")

# Output:
# num_exclaim: 0.35
# has_FREE: 0.65  ← More important!
```

Higher value = Feature used more in splits

Trees for Regression

DecisionTreeRegressor: Predict numbers

```python
from sklearn.tree import DecisionTreeRegressor

# Predict house prices
model = DecisionTreeRegressor(max_depth=4)
model.fit(X_train, y_train)

# Prediction = average of leaf node examples
predictions = model.predict(X_test)
```

Leaf prediction = mean of training samples in that leaf

The Overfitting Problem

```
Shallow Tree (max_depth=2)     Deep Tree (max_depth=10)

┌─────────────────────┐        ┌─────────────────────┐
│   May miss patterns │        │ Memorizes training  │
│   (underfitting)    │        │ data (overfitting)  │
│                     │        │                     │
│     Simple rules    │        │    Complex rules    │
└─────────────────────┘        └─────────────────────┘


Training accuracy: 75%         Training accuracy: 100%
Test accuracy: 73%             Test accuracy: 60%  ← Bad!
```

Controlling Tree Complexity

```python
model = DecisionTreeClassifier(
    max_depth=5,           # Maximum depth of tree
    min_samples_leaf=10,   # Minimum samples per leaf
    min_samples_split=20,  # Minimum samples to split
    max_features='sqrt',   # Features to consider per split
)
```

Parameter	Effect of Increasing
`max_depth`	More complex, more overfitting risk
`min_samples_leaf`	Simpler, less overfitting
`min_samples_split`	Fewer splits, simpler tree

Tree Pros and Cons

Pros

- Easy to understand/explain
- No feature scaling needed
- Handles non-linear patterns
- Works with categorical data
- Shows feature importance

Cons

- Prone to overfitting
- Unstable (small data changes → different tree)
- Not smooth predictions
- Biased toward high-cardinality features
- Greedy (may miss global optimum)

Part 4: K-Nearest Neighbors

Learning by Similarity

The Simplest Idea

"You are the average of your friends"

To predict for a new point:

1. Find the K closest training examples
2. For classification: Vote (majority wins)
3. For regression: Average

No explicit training - just store the data!

K-NN Visualization (K=3)

```
Class A: ●    Class B: ×    New point: ?


    ●   ●
  ●   ●   ●
    ●

         ┌─────┐
      ●  │  ?  │ ← 3 nearest: 2× and 1●
         │ × × │
         └─────┘

    ×   ×
  ×   ×   ×
    ×

Vote: 2 × vs 1 ● → Predict Class B (×)
```

Distance Matters

Euclidean Distance (most common):

$$d(a, b) = \sqrt{\sum_{i=1}^{n} (a_i - b_i)^2}$$

Distance Type	Formula	Best For
Euclidean	$\sqrt{(\Sigma(a_i - b_i)^2)}$	Continuous features
Manhattan	$\Sigma\lvert a_i - b_i\rvert$	Grid-like data
Cosine	$1 - \cos(\theta)$	Text, high-dimensional

K-NN in sklearn

```
from sklearn.neighbors import KNeighborsClassifier

# "Training" (just stores the data!)
model = KNeighborsClassifier(n_neighbors=5)
model.fit(X_train, y_train)

# Prediction (finds 5 nearest, votes)
predictions = model.predict(X_test)

# Can also get probabilities
probs = model.predict_proba(X_test)
```

Choosing K

K Value	Behavior	Trade-off
K=1	Use single nearest	Very sensitive to noise
K=3-5	Small neighborhood	Often works well
K=sqrt(n)	Rule of thumb	Balanced
K=large	Big neighborhood	Might miss local patterns

Use odd K for binary classification to avoid ties!

Effect of K

```
K = 1 (Noisy)           K = 5 (Smooth)          K = 100 (Too Smooth)
┌─────────────┐         ┌─────────────┐         ┌─────────────┐
│ ●│×│●│×│●│× │         │ ●●●│×××│●●● │         │ ●●●●●●●●●●●│
│ ×│●│×│●│×│● │         │ ●●●│×××│●●● │         │ ●●●●●●●●●●●│
│   (complex) │         │  (balanced) │         │  (too simple)│
└─────────────┘         └─────────────┘         └─────────────┘

Follows every point     Smooth boundary         Almost constant
```

K-NN for Regression

Instead of voting, take the average:

```
from sklearn.neighbors import KNeighborsRegressor

model = KNeighborsRegressor(n_neighbors=5)
model.fit(X_train, y_train)

# Prediction = average of 5 nearest neighbors' values
# If neighbors have prices: [50, 55, 60, 52, 58]
# Prediction = mean([50, 55, 60, 52, 58]) = 55
```

Feature Scaling is Critical!

Problem: Features with larger range dominate distance.

```
# Without scaling:
# Feature 1: Age (20-80)
# Feature 2: Income (20000-200000)

# Distance dominated by Income!
# Age difference of 60 vs Income difference of 180000
```

Solution: Always scale features for K-NN!

Scaling in Practice

```python
from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import KNeighborsClassifier

# Scale features to mean=0, std=1
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)  # Use same scaler!

# Now use K-NN
model = KNeighborsClassifier(n_neighbors=5)
model.fit(X_train_scaled, y_train)
predictions = model.predict(X_test_scaled)
```

K-NN Pros and Cons

Pros

- No training time (lazy learning)
- Simple to understand
- No assumptions about data
- Works for any number of classes
- Naturally handles multi-class

Cons

- Slow prediction (O(n) per query)
- Memory intensive (store all data)
- Sensitive to feature scaling
- Curse of dimensionality
- Sensitive to irrelevant features

The Curse of Dimensionality

Problem: In high dimensions, "nearest" becomes meaningless.

Dimensions	Volume of unit cube
1D	1
2D	1
10D	1
100D	1

But! Points spread out exponentially. Need exponentially more data!

Part 5: Evaluation Metrics

Measuring Model Performance

The Golden Rule

Always evaluate on data the model has never seen!

```python
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(
    X, y,
    test_size=0.2,      # 20% for testing
    random_state=42     # Reproducibility
)

# Train ONLY on training data
model.fit(X_train, y_train)

# Evaluate ONLY on test data
score = model.score(X_test, y_test)
```

Classification Metric: Accuracy

$$\text{Accuracy} = \frac{\text{Correct Predictions}}{\text{Total Predictions}}$$

```python
from sklearn.metrics import accuracy_score

predictions = model.predict(X_test)
accuracy = accuracy_score(y_test, predictions)
print(f"Accuracy: {accuracy:.1%}")  # e.g., 95.0%
```

Correct	Total	Accuracy
95	100	95%

The Accuracy Trap

Scenario: Detecting rare disease (1% of population)

Model	Strategy	Accuracy
Dumb	Always predict "Healthy"	**99%**
Smart	Tries to detect disease	95%

The "dumb" model has 99% accuracy but **misses ALL sick patients!**

Accuracy is misleading for imbalanced data.

The Confusion Matrix

	Predicted Negative	Predicted Positive
Actual Negative	True Negative (TN)	False Positive (FP)
Actual Positive	False Negative (FN)	True Positive (TP)

```python
from sklearn.metrics import confusion_matrix

cm = confusion_matrix(y_test, predictions)
print(cm)
# [[TN, FP],
#  [FN, TP]]
```

Confusion Matrix Example

Spam Detection (100 emails):

	Pred: Not Spam	Pred: Spam
Actual: Not Spam	85 (TN)	5 (FP)
Actual: Spam	2 (FN)	8 (TP)

Outcome	Meaning	Count
TN	Correctly said "not spam"	85
FP	Wrong! Said spam (was normal)	5
FN	Missed spam!	2
TP	Correctly caught spam	8

Precision: "Of my positive predictions..."

$$\text{Precision} = \frac{TP}{TP + FP}$$

"Of the emails I marked as spam, how many were actually spam?"

Predicted Spam	Actually Spam	Precision
13	8	8/13 = 62%

High precision = Few false alarms

Recall: "Of actual positives..."

$$\text{Recall} = \frac{TP}{TP + FN}$$

"Of all the actual spam, how many did I catch?"

Actual Spam	Caught	Recall
10	8	8/10 = 80%

High recall = Few missed positives

Precision vs Recall Trade-off

Scenario	Priority	Favor
Spam Filter	Don't lose important emails	Precision
Cancer Screening	Don't miss any cancer	Recall
Fraud Detection	Don't miss fraud	Recall
Search Results	Show relevant results	Precision

There's usually a trade-off: Increasing one often decreases the other!

F1 Score: The Balance

Harmonic mean of precision and recall:

$$F1 = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

Precision	Recall	F1 Score
0.90	0.90	0.90
0.95	0.50	0.65
0.50	0.95	0.65
0.00	1.00	0.00

F1 penalizes extreme imbalance!

All Metrics in Python

```python
from sklearn.metrics import (
    accuracy_score,
    precision_score,
    recall_score,
    f1_score,
    confusion_matrix
)

print(f"Accuracy:  {accuracy_score(y_test, pred):.2f}")
print(f"Precision: {precision_score(y_test, pred):.2f}")
print(f"Recall:    {recall_score(y_test, pred):.2f}")
print(f"F1:        {f1_score(y_test, pred):.2f}")
```

Classification Report

```python
from sklearn.metrics import classification_report

print(classification_report(y_test, predictions))
```

```
              precision    recall  f1-score   support

     class 0       0.95      0.98      0.97        85
     class 1       0.80      0.62      0.70        15

    accuracy                           0.93       100
   macro avg       0.88      0.80      0.83       100
weighted avg       0.93      0.93      0.93       100
```

Regression Metrics: MSE

Mean Squared Error:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

```python
from sklearn.metrics import mean_squared_error

mse = mean_squared_error(y_test, predictions)
print(f"MSE: {mse:.2f}")  # e.g., 2500.0
```

Interpretation	Issue
Average squared error	Units are squared (₹²)

RMSE: Interpretable Error

Root Mean Squared Error:

$$\text{RMSE} = \sqrt{\text{MSE}}$$

```python
import numpy as np

rmse = np.sqrt(mean_squared_error(y_test, predictions))
print(f"RMSE: ₹{rmse:.2f} lakhs")  # e.g., ₹50.0 lakhs
```

RMSE has same units as target - interpretable!

"On average, our predictions are off by ₹50 lakhs"

R^2 Score: Explained Variance

$$R^2 = 1 - \frac{\sum(y_i - \hat{y}_i)^2}{\sum(y_i - \bar{y})^2}$$

R^2 Value	Interpretation
1.0	Perfect predictions
0.8 - 0.9	Good model
0.5 - 0.8	Moderate
0.0	Same as predicting mean
< 0	Worse than predicting mean

R^2 in Python

```python
from sklearn.metrics import r2_score

r2 = r2_score(y_test, predictions)
print(f"R² Score: {r2:.3f}")  # e.g., 0.856

# Interpretation:
# "Our model explains 85.6% of the variance in prices"
```

Metrics Summary Table

Task	Metric	When to Use	Range
Classification	Accuracy	Balanced classes	[0, 1]
Classification	Precision	FP costly	[0, 1]
Classification	Recall	FN costly	[0, 1]
Classification	F1	Balance P/R	[0, 1]
Regression	MSE	General	$[0, \infty)$
Regression	RMSE	Interpretable	$[0, \infty)$
Regression	R^2	Compare models	$(-\infty, 1]$

Complete Workflow

```python
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import classification_report

# 1. Split
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42)

# 2. Train
model = DecisionTreeClassifier(max_depth=5)
model.fit(X_train, y_train)

# 3. Predict
predictions = model.predict(X_test)

# 4. Evaluate
print(classification_report(y_test, predictions))
```

Algorithm Comparison

Choosing the Right Tool

When to Use What?

Scenario	Recommended
Linear relationship, predict number	Linear Regression
Binary classification	Logistic Regression
Need interpretable rules	Decision Tree
Similar items should predict similarly	K-NN
High-dimensional data	Logistic Regression
Non-linear patterns	Decision Tree

Comparison Table

Aspect	Linear Reg	Logistic Reg	Tree	K-NN
Task	Regression	Classification	Both	Both
Scaling needed	No	No	No	Yes
Interpretable	Yes	Somewhat	Very	Somewhat
Training speed	Fast	Fast	Fast	None
Prediction speed	Fast	Fast	Fast	Slow
Handles non-linear	No	No	Yes	Yes

Key Takeaways

1. **Linear Regression:** Fit a line, minimize squared error, predict numbers
2. **Logistic Regression:** Sigmoid for probability, threshold for decision
3. **Decision Trees:** If-then rules, prone to overfitting, very interpretable
4. **K-NN:** Predict by neighbors, no training, scale features!
5. **Metrics:** Accuracy isn't everything - precision/recall matter for imbalanced data

You Now Know 4 Algorithms!

Next: Model Selection & Ensembles

Lab: Implement these algorithms on real datasets

"All models are wrong, but some are useful."
— George Box

Questions?