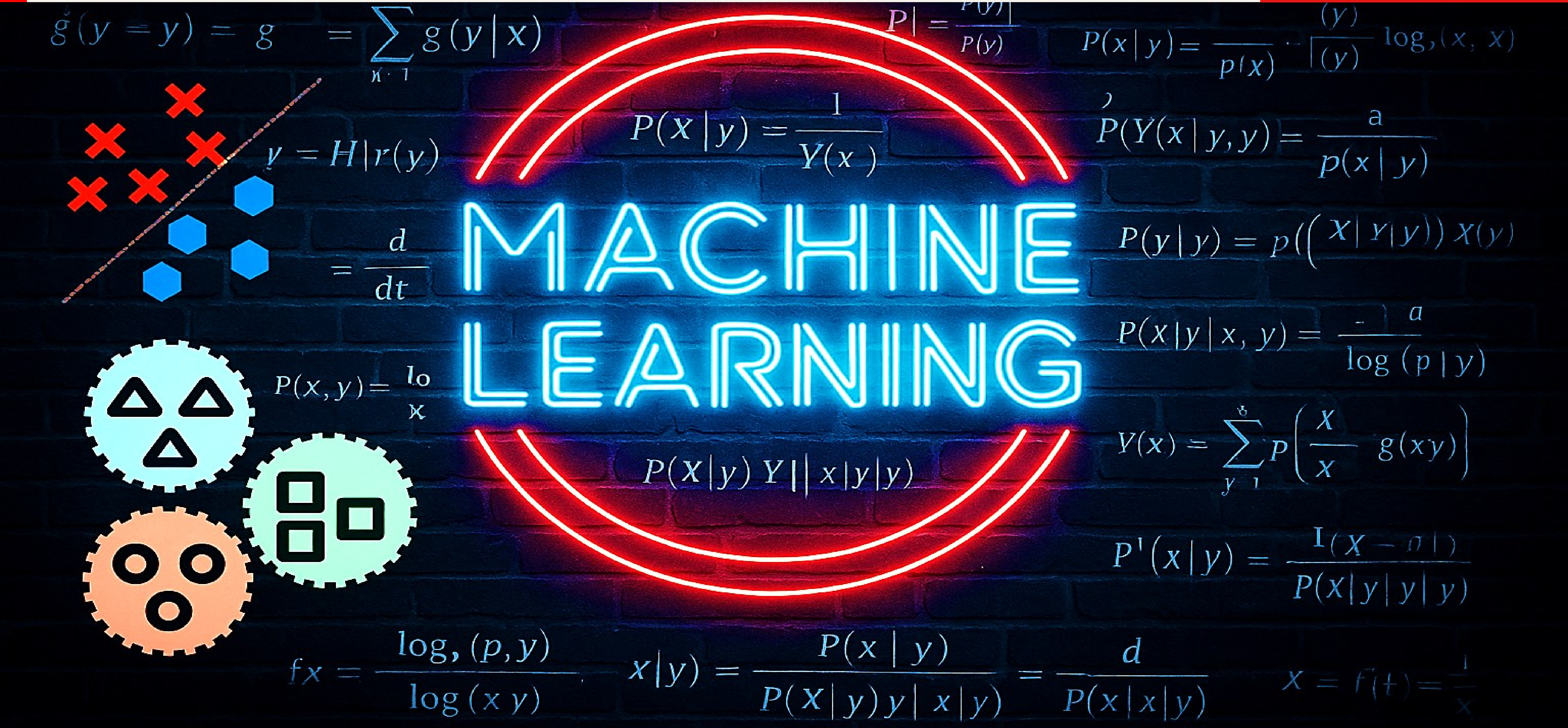
Lecture 3 – Supervised Learning

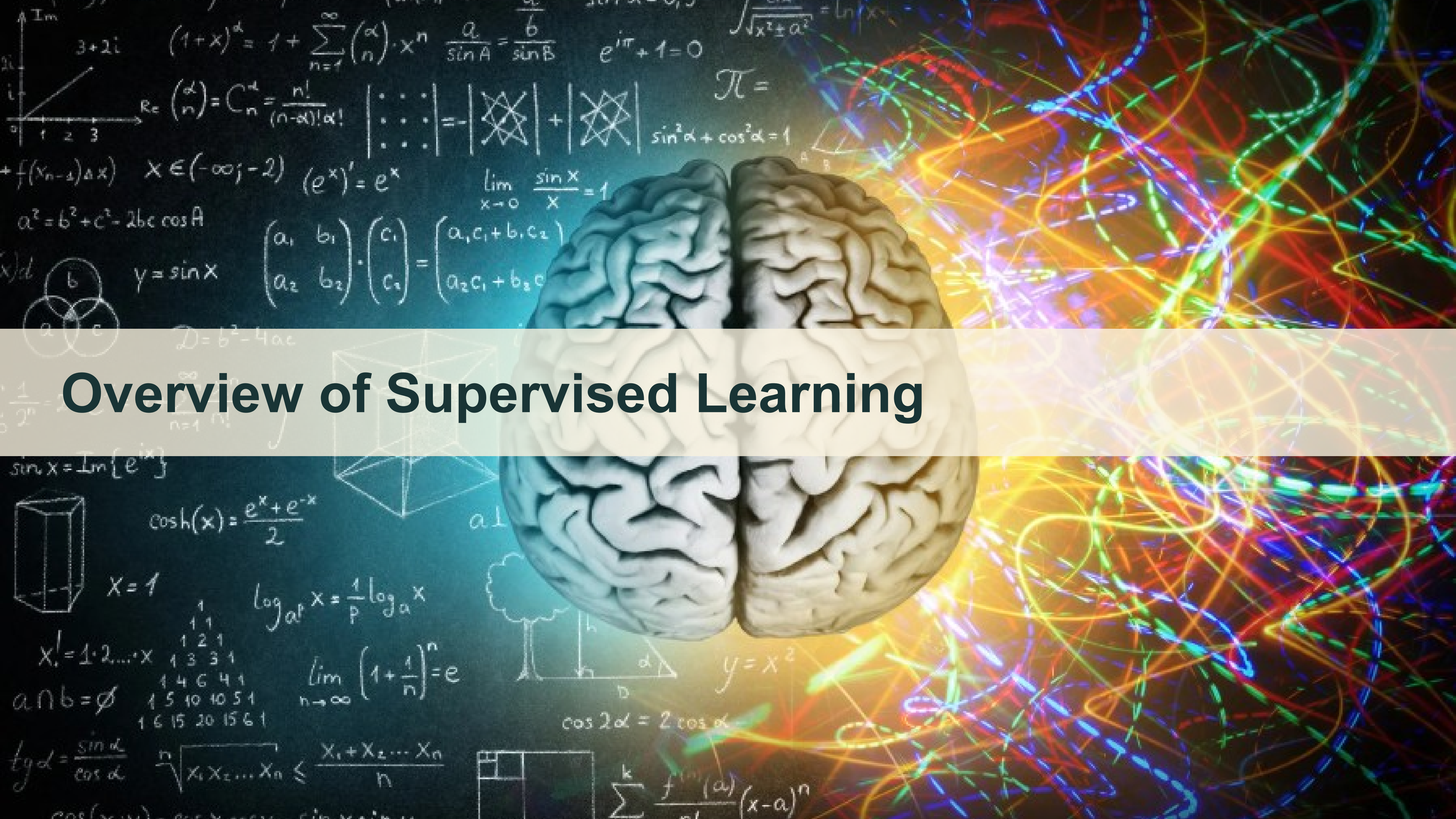
COMPX310| Irfan Ahmad | AI | Data Science | Computer Science

THE UNIVERSITY OF WAIKATO
Te Whare Wānanga o Waikato



Today’s Topics

- Overview of Supervised Learning
- Classification Example 1 (MNIST Data)
- Classification Example 2 (KNN Method)



Overview of Supervised Learning

Supervised Machine Learning

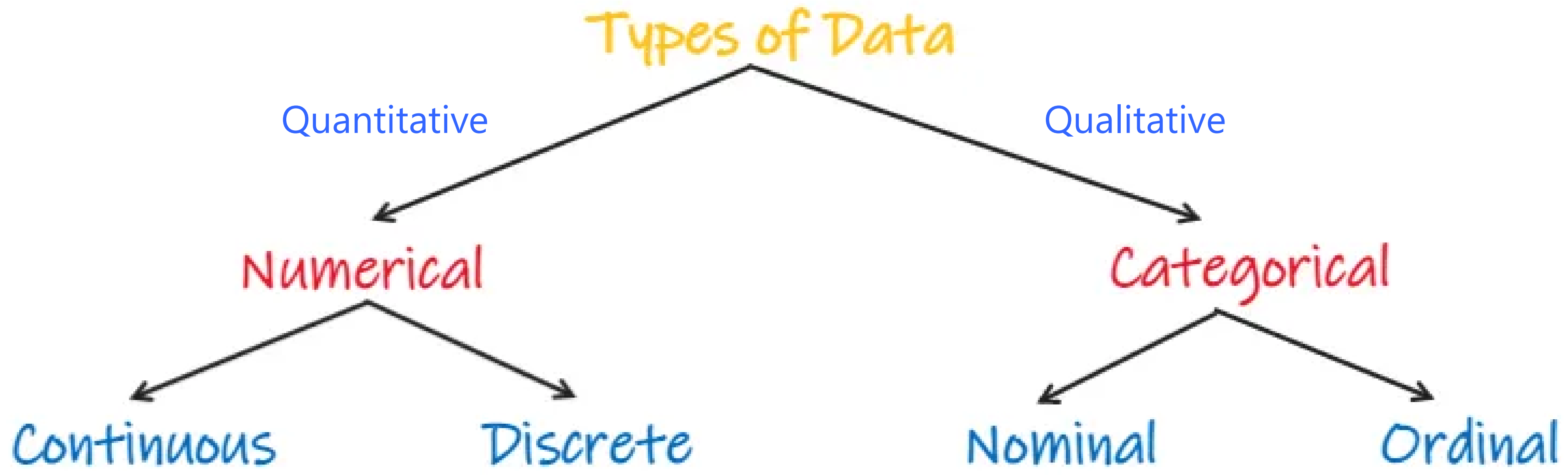
- We have labeled data (input/output pairs)

Input / **Output**

longitude	latitude	median_age	median_income	ocean_proximity	median_house_value
-122.23	37.88	41.0	8.3252	Near Bay	452600.0
-122.22	37.86	21.0	8.3014	Near Bay	358500.0
-122.24	37.85	52.0	7.2574	Near Bay	352100.0
-122.25	37.85	52.0	5.6431	Near Bay	341300.0
-122.25	37.85	52.0	3.8462	Near Bay	342200.0

Numerical and Categorical Variables

- Two different types of variables in our data: numerical or categorical



Quantitative

Qualitative

Continuous
- Can take any value
- Example:
  - Height, weight, price

Discrete
- Countable values only.
- Example:
  - Number of people, test scores

Nominal
- Names, no order.
- Example:
  - Colors, gender, city

Ordinal
- Ordered categories.
- Example:
  - Small/Medium/Large, ranks

Supervised Learning - Classification

- Classifying songs
  - Length and Energy is input, and song class is output.

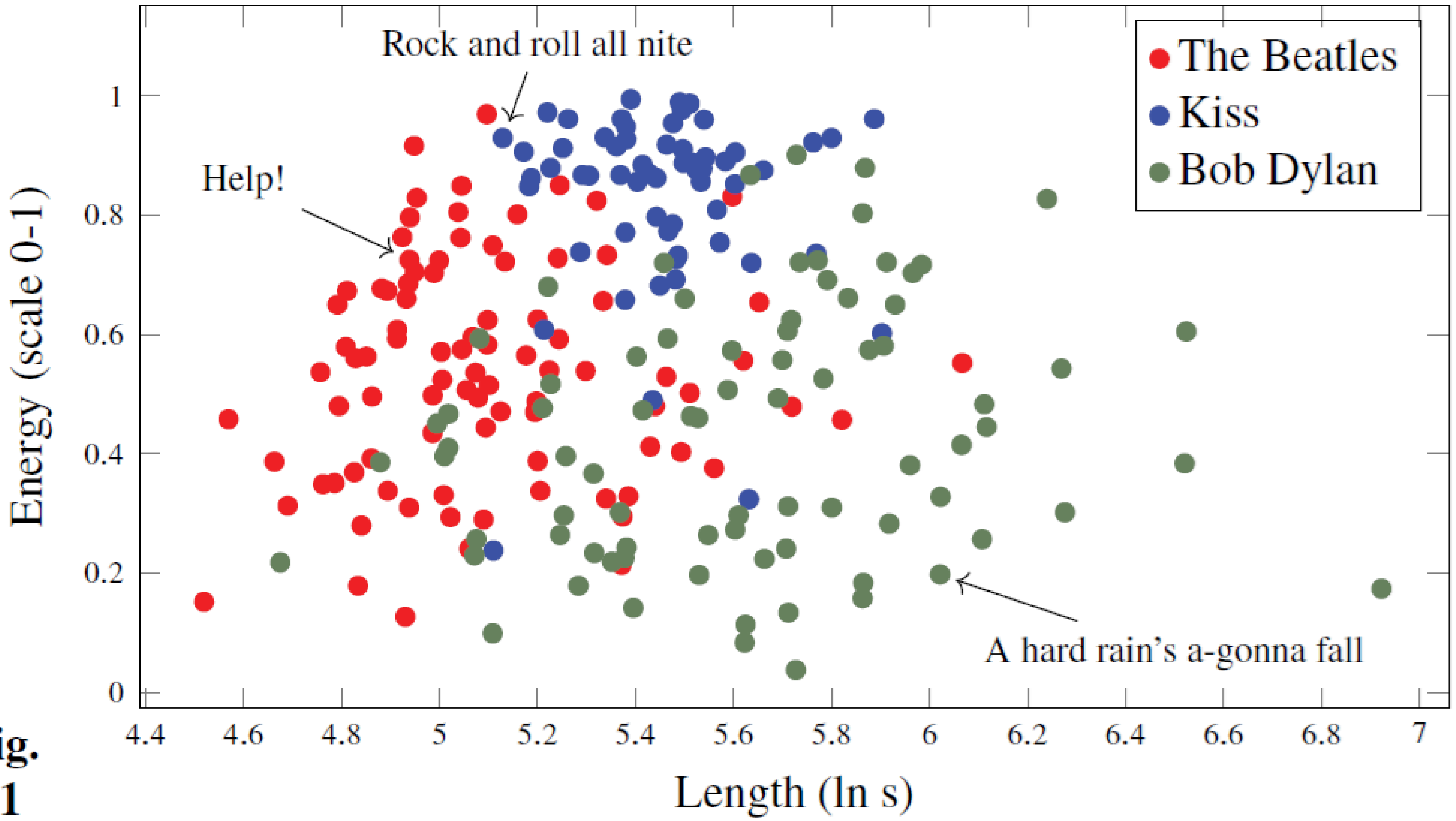
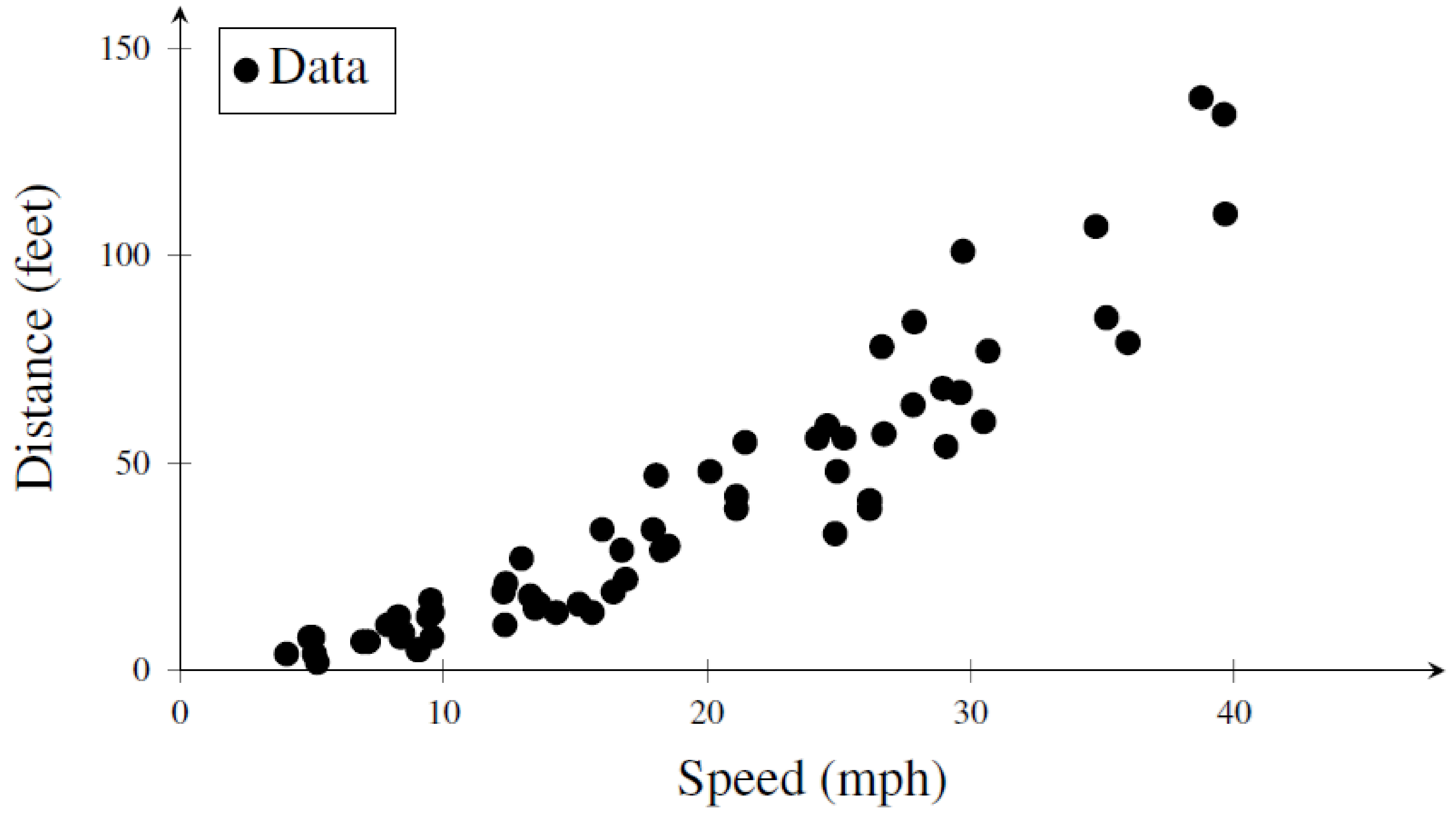
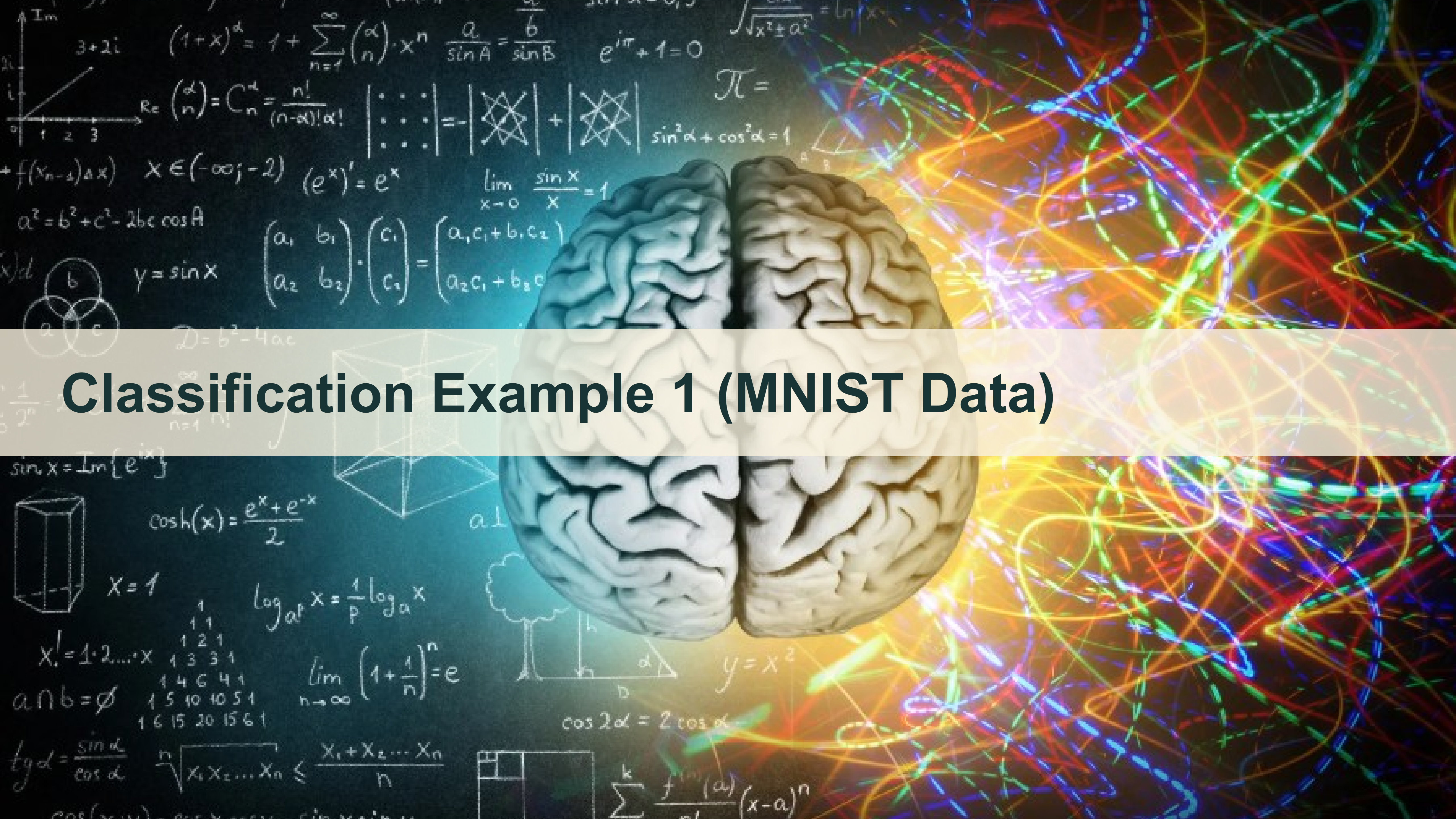


Fig. 2.1

Supervised Learning - Regression

- Car stopping distance
  - Speed is input and Distance is output.





Classification Example 1 (MNIST Data)

Classification Overview

- Two main tasks in ML:
  - Regression → predict numbers (already covered)
  - Classification → predict categories (cover in this lecture)

- **MNIST** = famous dataset for digit classification (0–9)

- Called the “Hello World” of Machine Learning

MNIST Dataset

- 70,000 handwritten digit images (0–9)
- Collected from US high school students & Census Bureau staff
- Each image: 28 × 28 pixels = 784 features
- Pixel values: 0 = white, 255 = black

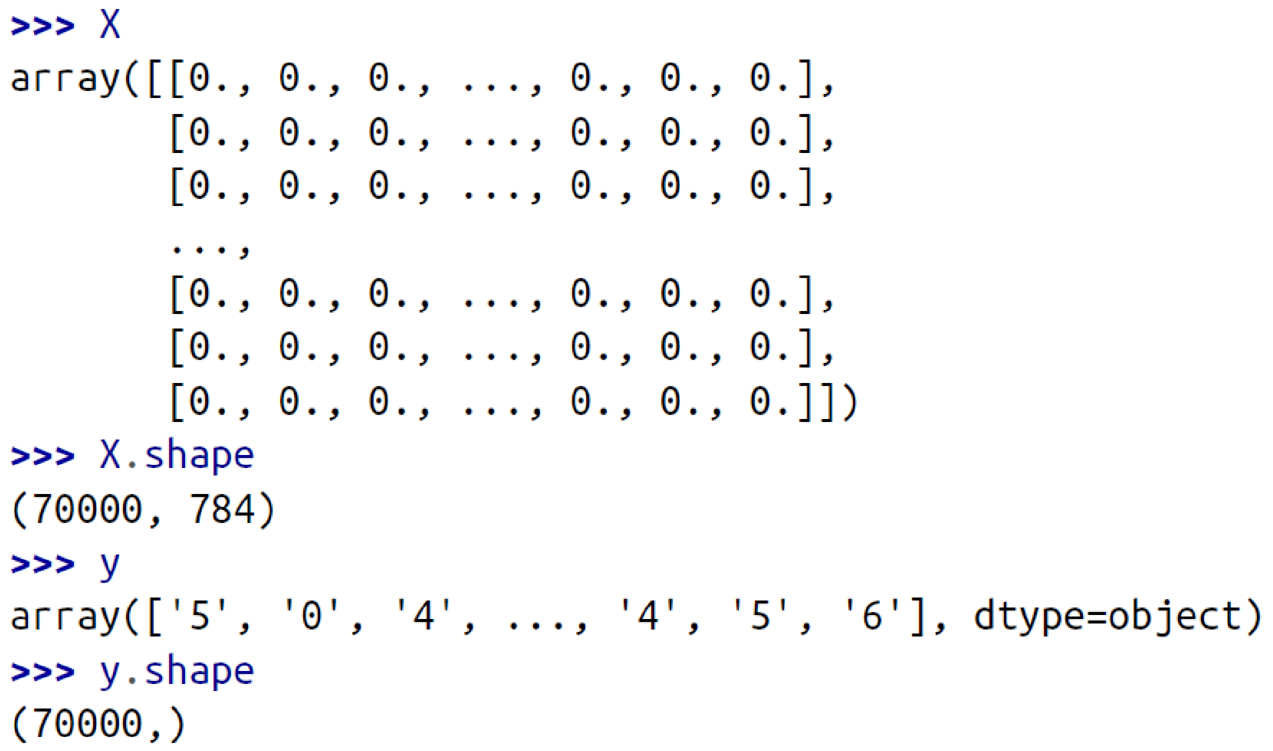
Loading MNIST in Scikit-Learn

- Use **fetch_openml()** from sklearn.datasets
  - fetch_* → download real-life datasets (e.g., fetch_openml())
  - load_* → load small built-in toy datasets
  - make_* → create fake datasets for testing
- Returns data as NumPy arrays (when as_frame=False)
- Data (X) = pixel intensities
- Labels (y) = digit class (0–9)

```python
from sklearn.datasets import fetch_openml

mnist = fetch_openml('mnist_784', as_frame=False)

X, y = mnist.data, mnist.target
```

Dataset Structure

- X.shape → (70000, 784) (70k images × 784 pixels)
- y.shape → (70000,) (label for each image)
- Pixel intensity values: 0–255



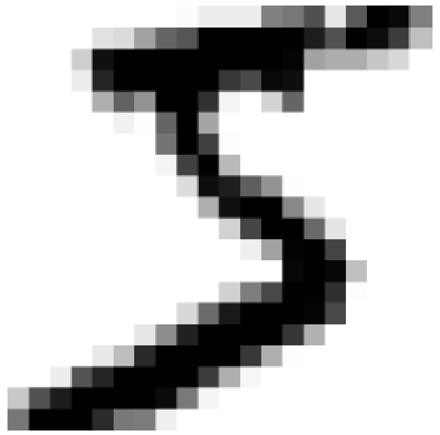
```
>>> X
array([[0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       ...,
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.]])
>>> X.shape
(70000, 784)
>>> y
array(['5', '0', '4', ..., '4', '5', '6'], dtype=object)
>>> y.shape
(70000,)
```

Viewing a Digit

- Reshape feature vector to 28 × 28
- Display using Matplotlib with cmap="binary"

```
import matplotlib.pyplot as plt

def plot_digit(image_data):
    image = image_data.reshape(28, 28)
    plt.imshow(image, cmap="binary")
    plt.axis("off")

some_digit = X[0]
plot_digit(some_digit)
plt.show()
```

```
>>> y[0]
'5'
```



looks like a 5 and label tells us the same

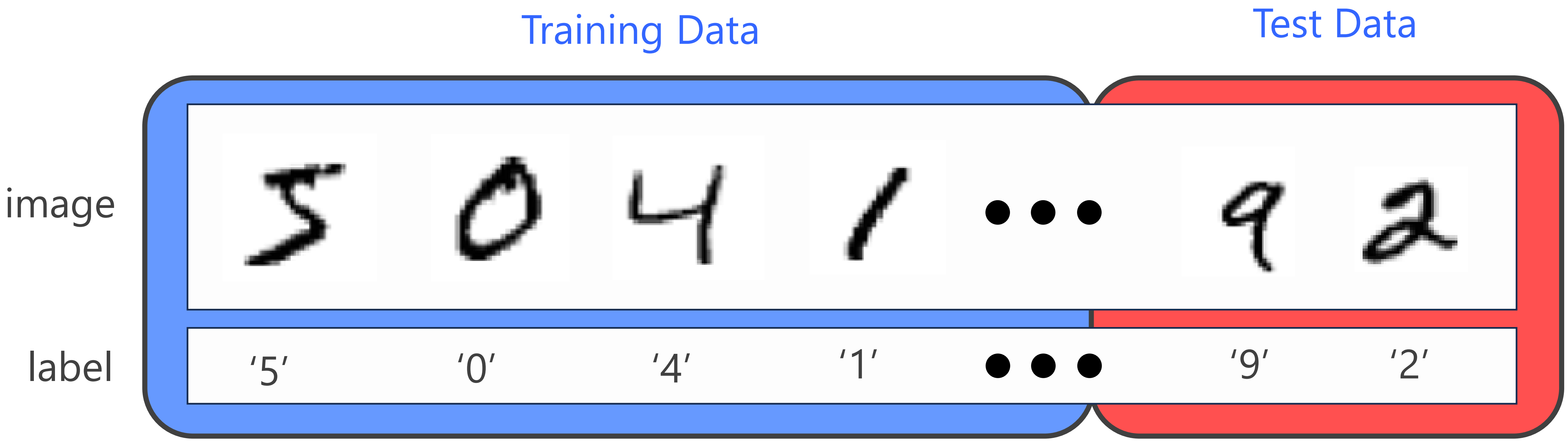
Viewing a Digit

- Figure shows a few more images from the MNIST dataset.

Train Test Split

- MNIST dataset returned by fetch_openml() is **already split** into a **training set** (the first 60,000 images) and a test set (the last 10,000 images)

```
X_train, X_test, y_train, y_test = X[:60000], X[60000:], y[:60000], y[60000:]
```

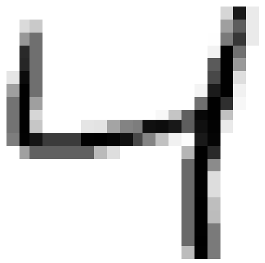


Training a Binary Classifier

- A classifier that **predicts two classes** only.
- Our task: detect digit 5 (class 1) or not 5 (class 0)
- Example: “5-detector” model

✅ (is 5)

❌ (not 5)



```
X_train, X_test, y_train, y_test = X[:60000], X[60000:], y[:60000], y[60000:]
```

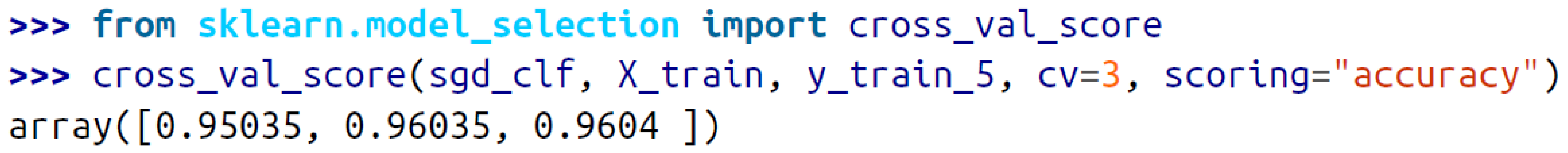
Choose a Classifier (SGDClassifier)

- **SGDClassifier** = Stochastic Gradient Descent Classifier
  - Handles very large datasets well
  - Learns one example at a time → good for online learning

```
from sklearn.linear_model import SGDClassifier

sgd_clf = SGDClassifier(random_state=42)
sgd_clf.fit(X_train, y_train_5)

some_digit = X[0]

>>> sgd_clf.predict([some_digit])
array([ True])
```

Performance Measures

- **Classifier** evaluation is **harder** than regression
- Many metrics to learn (accuracy, confusion matrix, etc.)
- First: measure accuracy with cross-validation
  - Cross-validation = split training set into k folds
  - Train/test k times → more reliable score



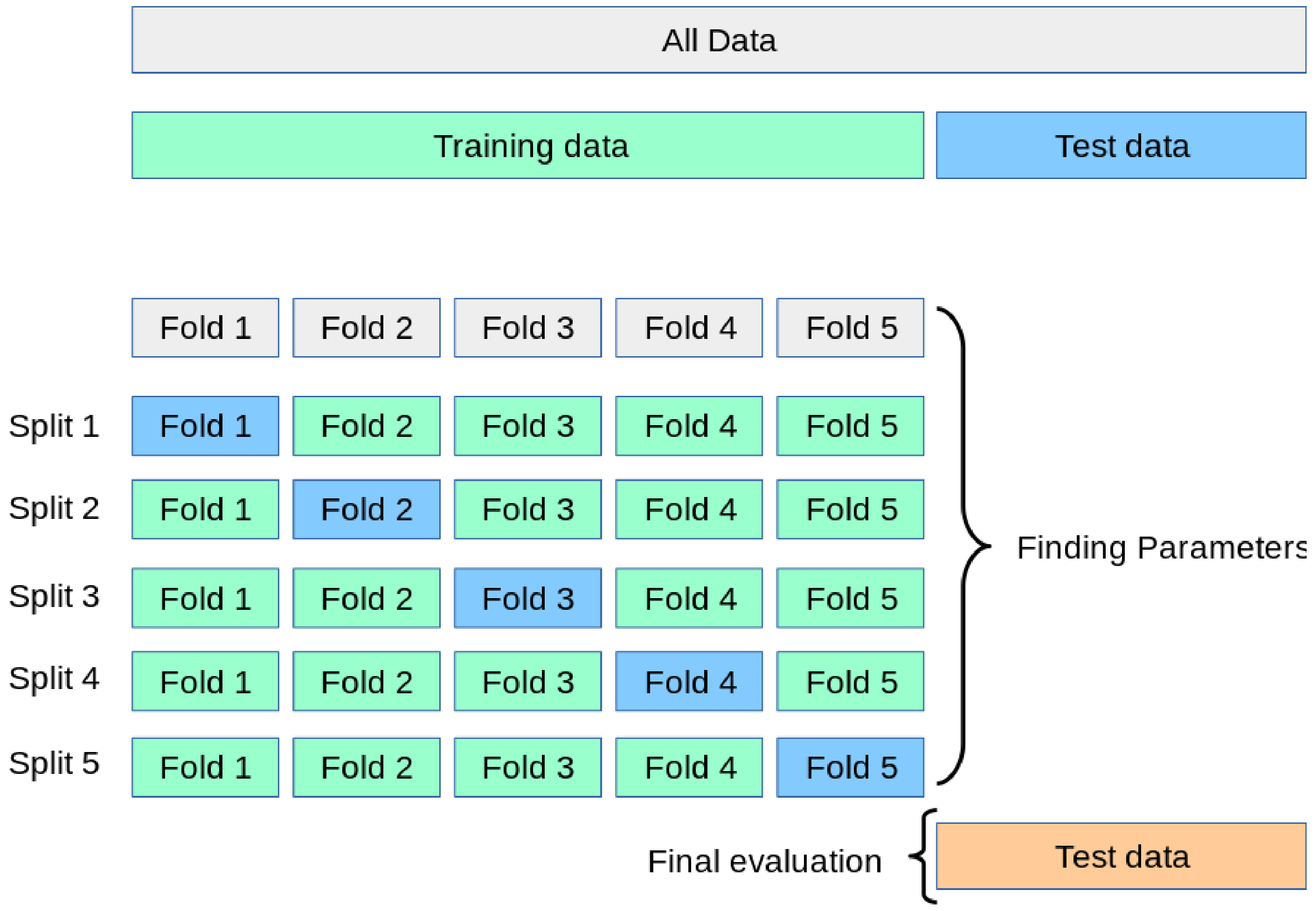
```
>>> from sklearn.model_selection import cross_val_score
>>> cross_val_score(sgd_clf, X_train, y_train_5, cv=3, scoring="accuracy")
array([0.95035, 0.96035, 0.9604 ])
```

Wow! Above 95% accuracy

Performance Measures

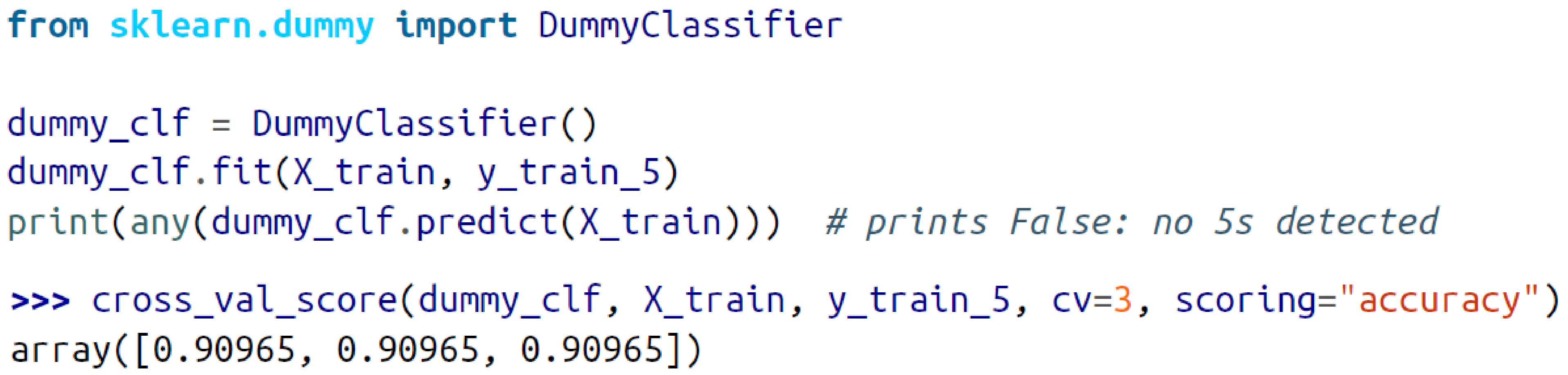
Cross Validation Concept

- In k-fold cross-validation, the data is divided into k subsets, and the model is trained and validated k times.
- Each time, one of the k subsets is used as the test set, and the remaining k-1 subsets form the training set.
- The final performance is the average of the k trials.



The Dummy Classifier Trick

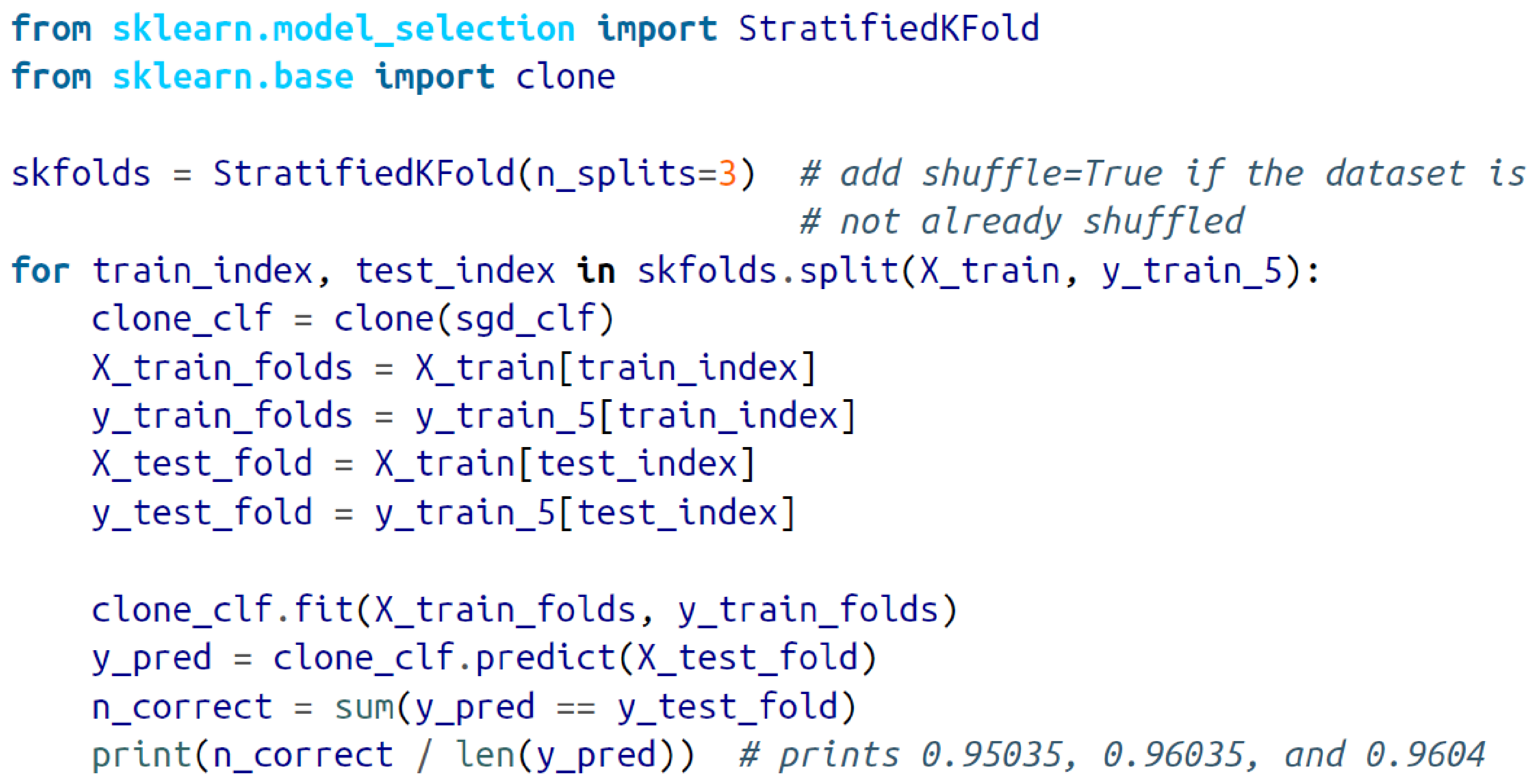
- Always predicts most common class
- Here: always "not 5" → 90% accuracy (because only about 10% of the images are 5s)
- Shows why accuracy can mislead in skewed data.



```
from sklearn.dummy import DummyClassifier

dummy_clf = DummyClassifier()
dummy_clf.fit(X_train, y_train_5)
print(any(dummy_clf.predict(X_train)))  # prints False: no 5s detected

>>> cross_val_score(dummy_clf, X_train, y_train_5, cv=3, scoring="accuracy")
array([0.90965, 0.90965, 0.90965])
```

Manual Cross-Validation (StratifiedKFold)

- StratifiedKFold → keeps class proportions same in each fold
- More control over CV process
- Fresh model each fold (clone())

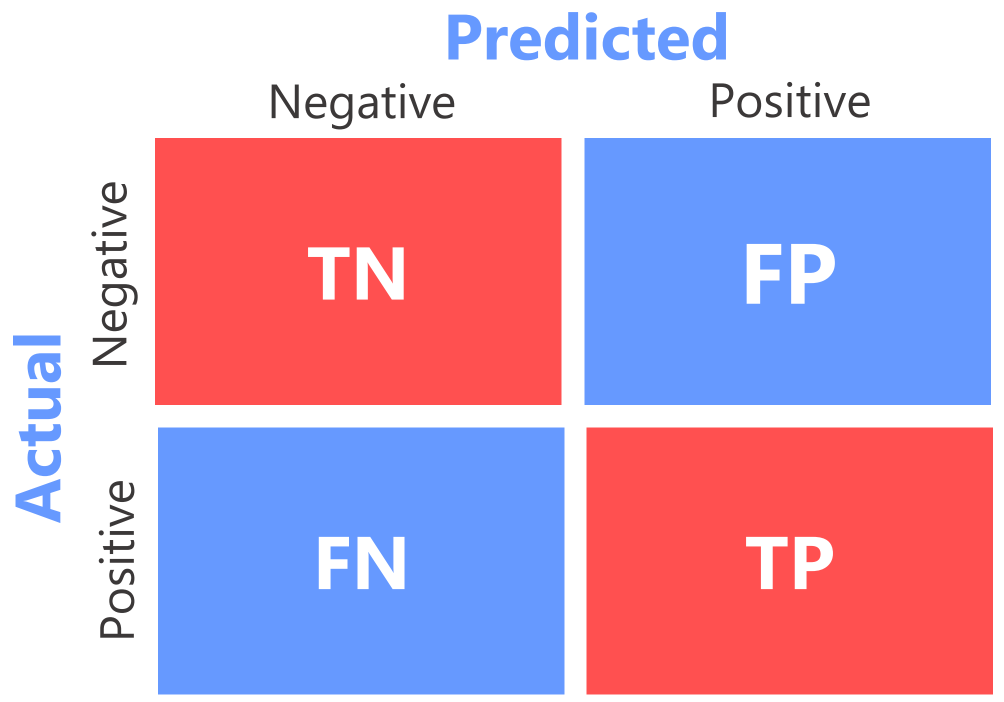


```python
from sklearn.model_selection import StratifiedKFold
from sklearn.base import clone

skfolds = StratifiedKFold(n_splits=3)  # add shuffle=True if the dataset is
                                       # not already shuffled
for train_index, test_index in skfolds.split(X_train, y_train_5):
    clone_clf = clone(sgd_clf)
    X_train_folds = X_train[train_index]
    y_train_folds = y_train_5[train_index]
    X_test_fold = X_train[test_index]
    y_test_fold = y_train_5[test_index]

    clone_clf.fit(X_train_folds, y_train_folds)
    y_pred = clone_clf.predict(X_test_fold)
    n_correct = sum(y_pred == y_test_fold)
    print(n_correct / len(y_pred))  # prints 0.95035, 0.96035, and 0.9604
```

Confusion Matrix

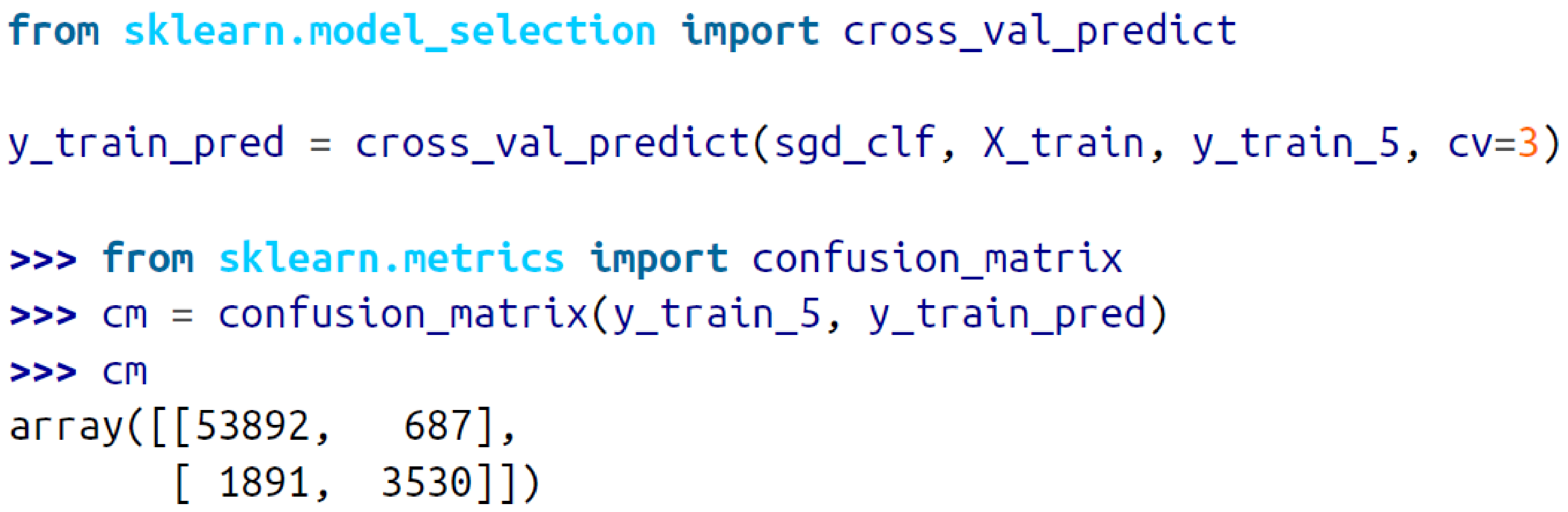
- Confusion matrix summarizes classifier's accuracy.
- Matrix shows correct and incorrect classifications.
- Rows = actual class, columns = predicted class
- Obtain using `confusion_matrix()` in scikit-learn.

		Predicted	
		No diabetes	**Diabetes**
Actual	**No diabetes**	201	85
	Diabetes	25	2689



Making the Confusion Matrix

- We need predictions before making confusion matrix
- Use cross_val_predict() → makes clean predictions (model never sees test fold before predicting)
- Use confusion_matrix() with true labels and predicted labels



```
from sklearn.model_selection import cross_val_predict

y_train_pred = cross_val_predict(sgd_clf, X_train, y_train_5, cv=3)

>>> from sklearn.metrics import confusion_matrix
>>> cm = confusion_matrix(y_train_5, y_train_pred)
>>> cm
array([[53892,   687],
       [ 1891,  3530]])
```

Understanding the Numbers

- True Negatives (TN) → correct “Not 5” predictions (53,892)
- False Positives (FP) → “Not 5” predicted as “5” (687) → Type I error
- False Negatives (FN) → “5” predicted as “Not 5” (1,891) → Type II error
- True Positives (TP) → correct “5” predictions (3,530)

		Predicted	
		Negative	**Positive**
Actual	**Negative**	TN	FP
	Positive	FN	TP

		Predicted	
		Not “5”	**“5”**
Actual	**Not “5”**	53892	687
	“5”	1891	3530

Perfect Classifier Example

- Only TP & TN (no FP, no FN)
- Confusion matrix has numbers only on main diagonal

```
>>> y_train_perfect_predictions = y_train_5  # pretend we reached perfection
>>> confusion_matrix(y_train_5, y_train_perfect_predictions)
array([[54579,     0],
       [    0,  5421]])
```

		Predicted	
		Negative	Positive
Actual	Negative	TN	FP
	Positive	FN	TP

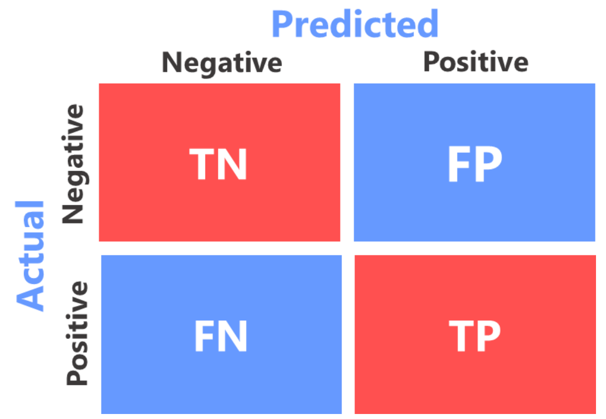
		Predicted	
		Not “5”	“5”
Actual	Not “5”	54579	0
	“5”	0	5421

Accuracy – Proportion of Correct Prediction

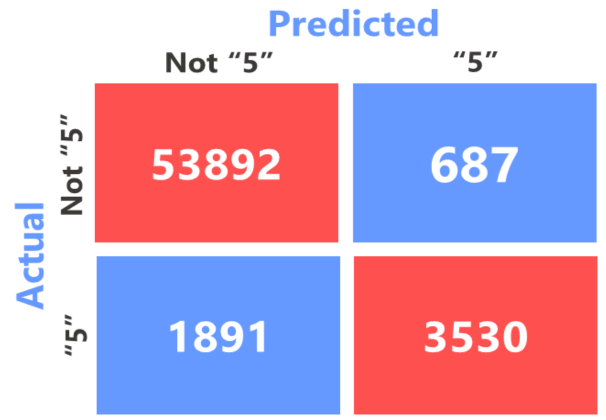
- Proportion of **all correct predictions** in the dataset.
- The overall accuracy of a classifier is estimated by

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

$$Accuracy = \frac{3530 + 53892}{3530 + 53892 + 687 + 1891} = 0.914$$



		Predicted	
		Negative	Positive
Actual	Negative	TN	FP
	Positive	FN	TP



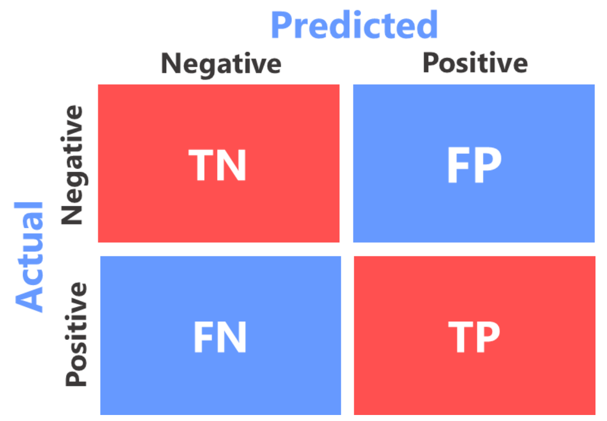
		Predicted	
		Not “5”	“5”
Actual	Not “5”	53892	687
	“5”	1891	3530

Precision – Positive Prediction Accuracy

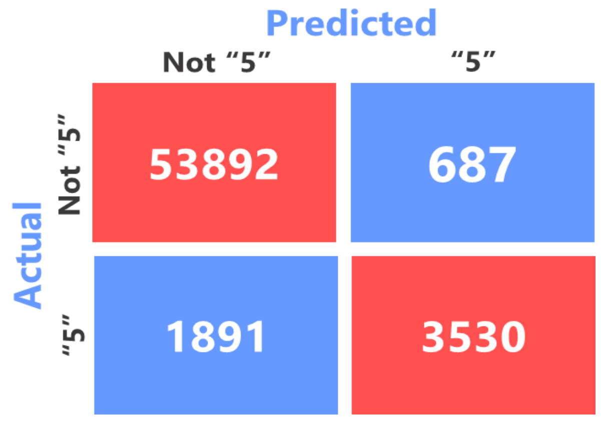
- Measures how many predicted positives are correct.
- Assesses the exactness or quality of the classifier
- Out of 4217 predicted as positive (“5”), 3530 were correct.
- Not useful alone → ignores missed positives

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

$$Precision = \frac{3530}{3530 + 687} = 0.837$$



		Predicted	
		Negative	Positive
Actual	Negative	TN	FP
	Positive	FN	TP



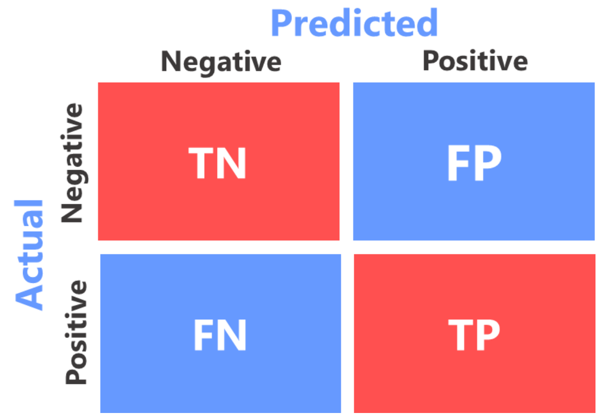
		Predicted	
		Not “5”	“5”
Actual	Not “5”	53892	687
	“5”	1891	3530

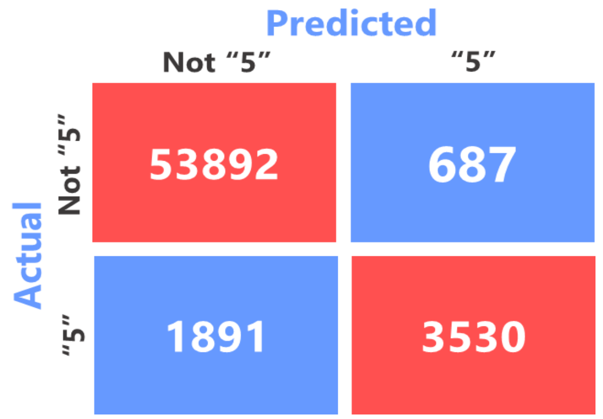
Recall – True Positive Rate

- Measures how many **real positives** were found
- Ratio of true positives to total (actual) positives in the data.
- Also called Sensitivity or True Positive Rate
- Of 5421 actual “5”, 3530 were predicted as “5”.

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

$$Recall = \frac{3530}{3530 + 1891} = 0.651$$




Predicted
Not “5”
“5”
Actual
Not “5”
“5”
53892
687
1891
3530

F1-Score

- Harmonic mean of precision and recall
- Combine precision and recall into a single metric
- Use when we need a single metric to compare two classifiers

$$F1\text{-}Score = \frac{2\times(Precision\times Recall)}{(Precision+Recall)}$$

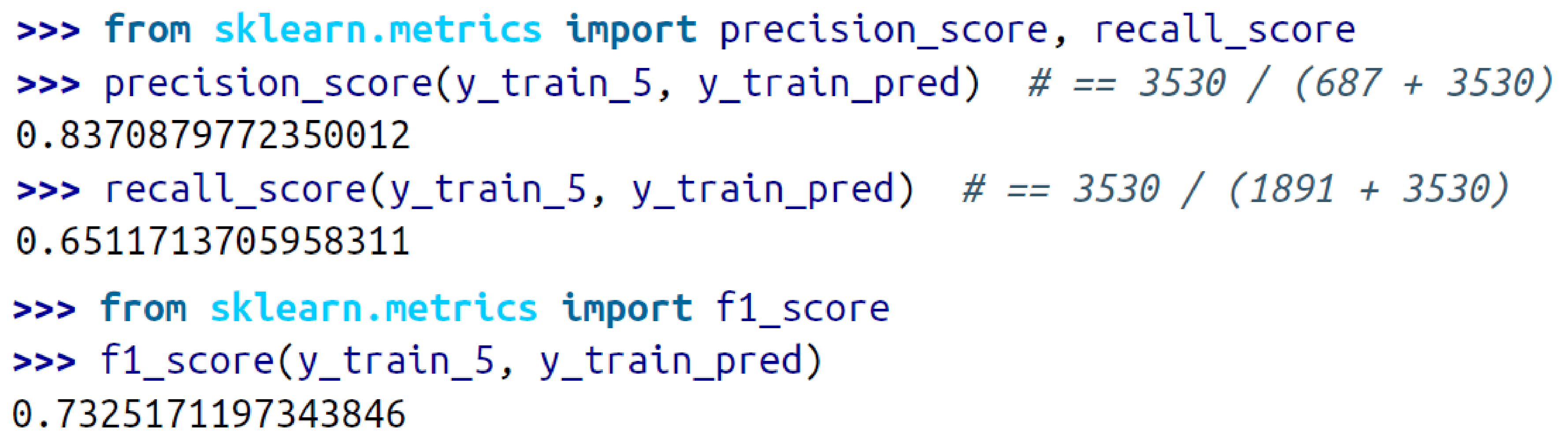
$$F1\text{-}Score = \frac{2\times(0.837\times 0.651)}{(0.837+0.651)} = 0.732$$

		Predicted	
		Negative	Positive
Actual	Negative	TN	FP
	Positive	FN	TP

		Predicted	
		Not “5”	“5”
Actual	Not “5”	53892	687
	“5”	1891	3530

Precision, Recall and F1-Score in Sklearn

- Scikit-Learn provides several functions to compute classifier metrics, including precision, recall and F1-Score.

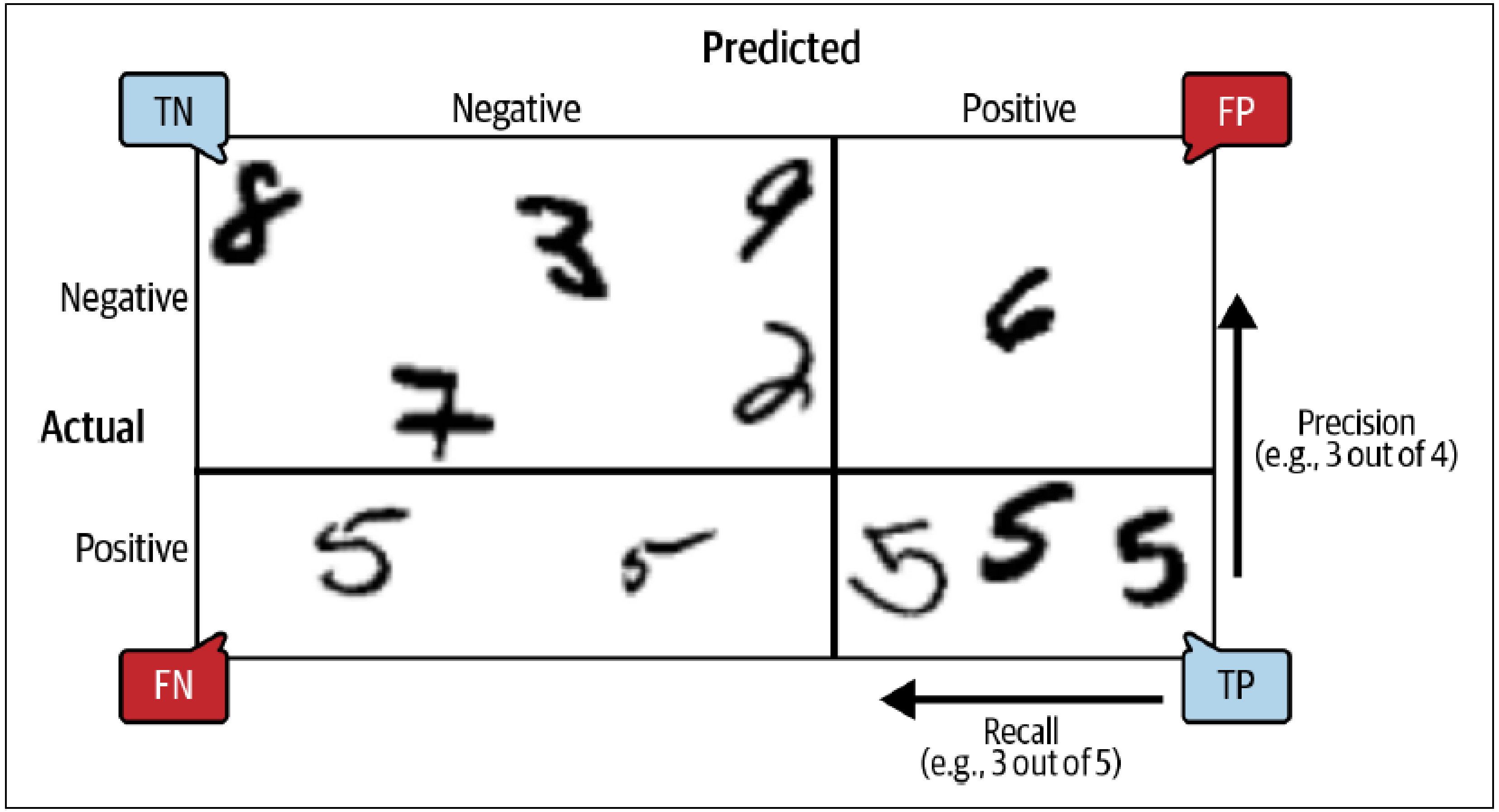


```
>>> from sklearn.metrics import precision_score, recall_score
>>> precision_score(y_train_5, y_train_pred)  # == 3530 / (687 + 3530)
0.8370879772350012
>>> recall_score(y_train_5, y_train_pred)  # == 3530 / (1891 + 3530)
0.6511713705958311
```

```
>>> from sklearn.metrics import f1_score
>>> f1_score(y_train_5, y_train_pred)
0.7325171197343846
```

Confusion Matrix - Summary

- If you are confused about the confusion matrix, Figure below may help.



Precision/Recall Trade-off

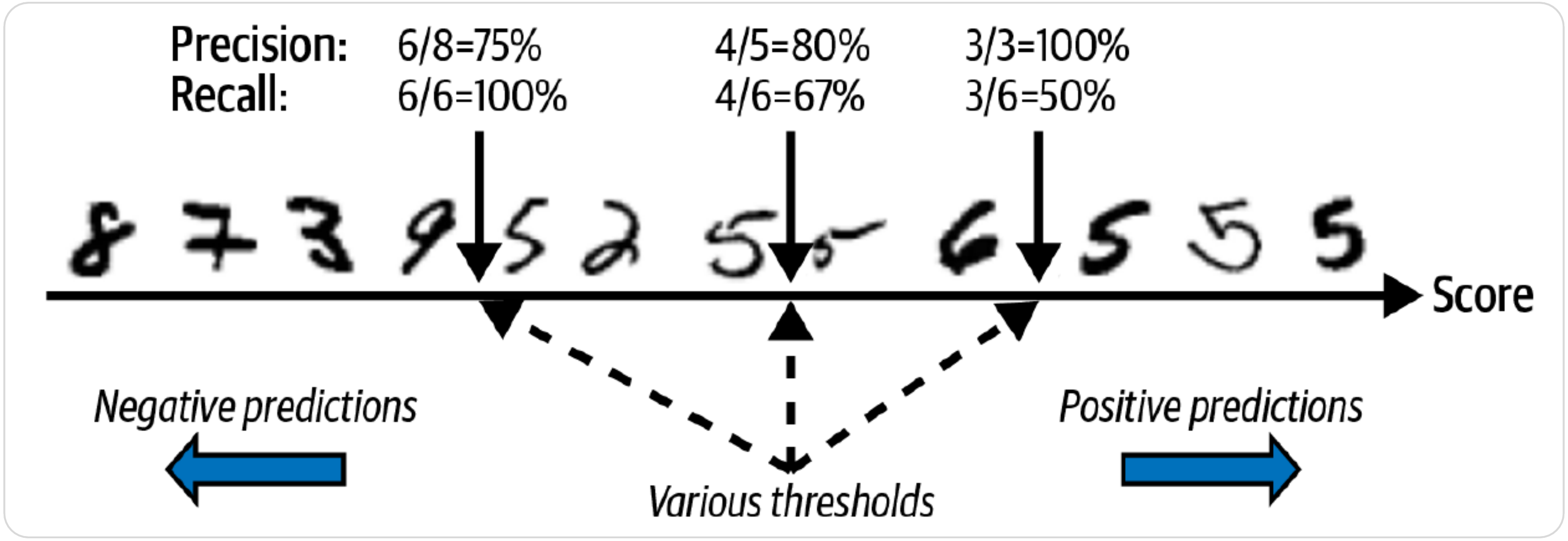
- Some problems need high precision (how many predicted positives are correct)
  - Example: Safe videos for kids
  - Important to avoid bad content, even if we miss good ones

- Some problems need high recall
  - Example: Detecting shoplifters
  - Okay to raise false alarms, but don’t miss real ones

- You can't increase both at the same time. This is called a trade-off
  - ⬆️ High Precision → ⬇️ Low Recall
  - ⬆️ High Recall → ⬇️ Low Precision

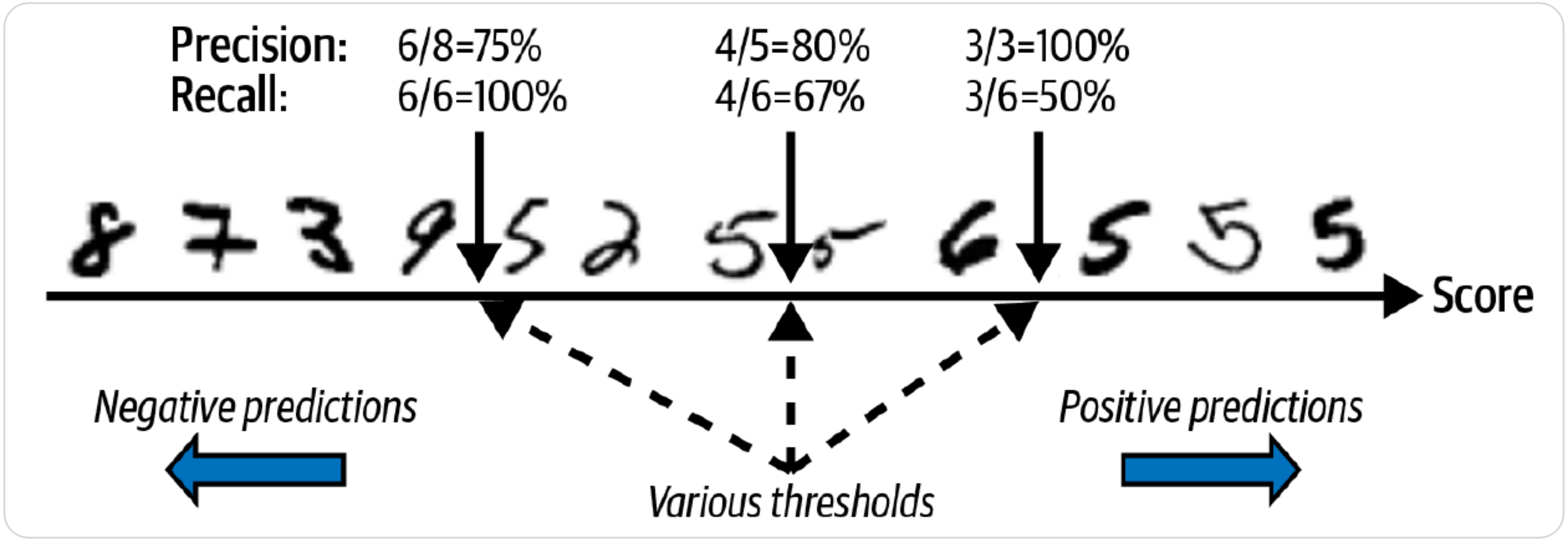
Precision/Recall Trade-off – Example

- Classifier gives each instance a score
- If score > threshold → predict positive (5)
- If score < threshold → predict negative (not-5)
- Moving threshold changes precision and recall.



Precision/Recall Trade-off – Example

- Threshold at center:
  - 4 true positives (actual 5s)
  - 1 false positive (6 wrongly predicted as 5)
  - Precision = 80% (4/5), Recall = 67% (4/6)
- Raise threshold → precision = 100%, recall = 50%
- Lower threshold → recall increases, precision decreases



Controlling Threshold in Code

- By default, SGDClassifier uses threshold = 0
- You can access decision score using:

```
>>> y_scores = sgd_clf.decision_function([some_digit])
>>> y_scores
array([2164.22030239])
```

- Then apply any threshold:

```
>>> threshold = 0
>>> y_some_digit_pred = (y_scores > threshold)
array([ True])
>>> threshold = 3000
>>> y_some_digit_pred = (y_scores > threshold)
>>> y_some_digit_pred
array([False])
```

How do You Decide which Threshold to Use?

- First, get decision scores for all samples:

```
y_scores = cross_val_predict(sgd_clf, X_train, y_train_5, cv=3,
                             method="decision_function")
```

- Then compute precision, recall, and thresholds:

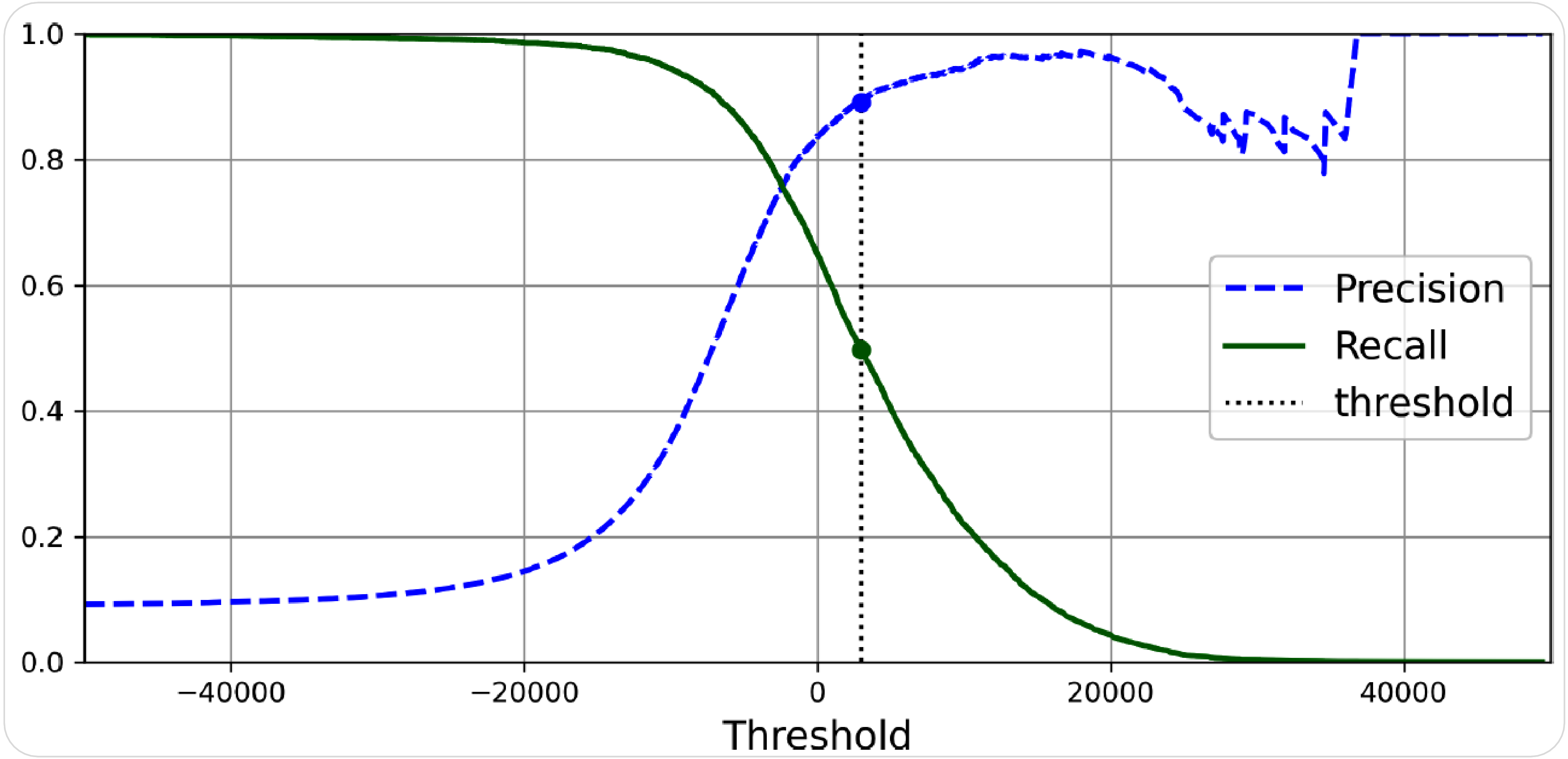
```
from sklearn.metrics import precision_recall_curve

precisions, recalls, thresholds = precision_recall_curve(y_train_5, y_scores)
```

- Plot curves:

```
plt.plot(thresholds, precisions[:-1], "b--", label="Precision", linewidth=2)
plt.plot(thresholds, recalls[:-1], "g-", label="Recall", linewidth=2)
plt.vlines(threshold, 0, 1.0, "k", "dotted", label="threshold")
[...]  # beautify the figure: add grid, legend, axis, labels, and circles
plt.show()
```

How do You Decide which Threshold to Use?

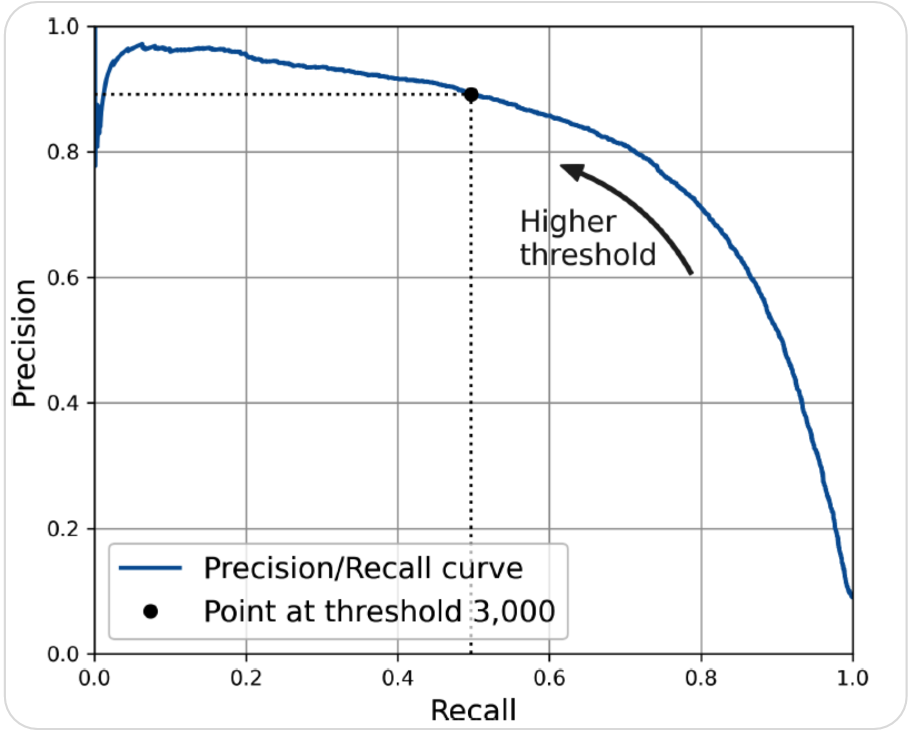
- At threshold ≈ 3000 → precision ≈ 90%, recall ≈ 50%



Precision vs Recall Curve

- Plot precision directly against recall
- Shows trade-off clearly
- Precision drops fast after ~80% recall
- Good idea: pick trade-off before sharp drop (e.g., 60%)

```
plt.plot(recalls, precisions, linewidth=2, label="Precision/Recall curve")
[...]  # beautify the figure: add labels, grid, legend, arrow, and text
plt.show()
```



Choosing a Target Precision

- Can search for threshold giving at least 90% precision
- Use NumPy argmax() to find correct threshold

```
>>> idx_for_90_precision = (precisions >= 0.90).argmax()
>>> threshold_for_90_precision = thresholds[idx_for_90_precision]
>>> threshold_for_90_precision
3370.0194991439557
```

- Then make predictions:

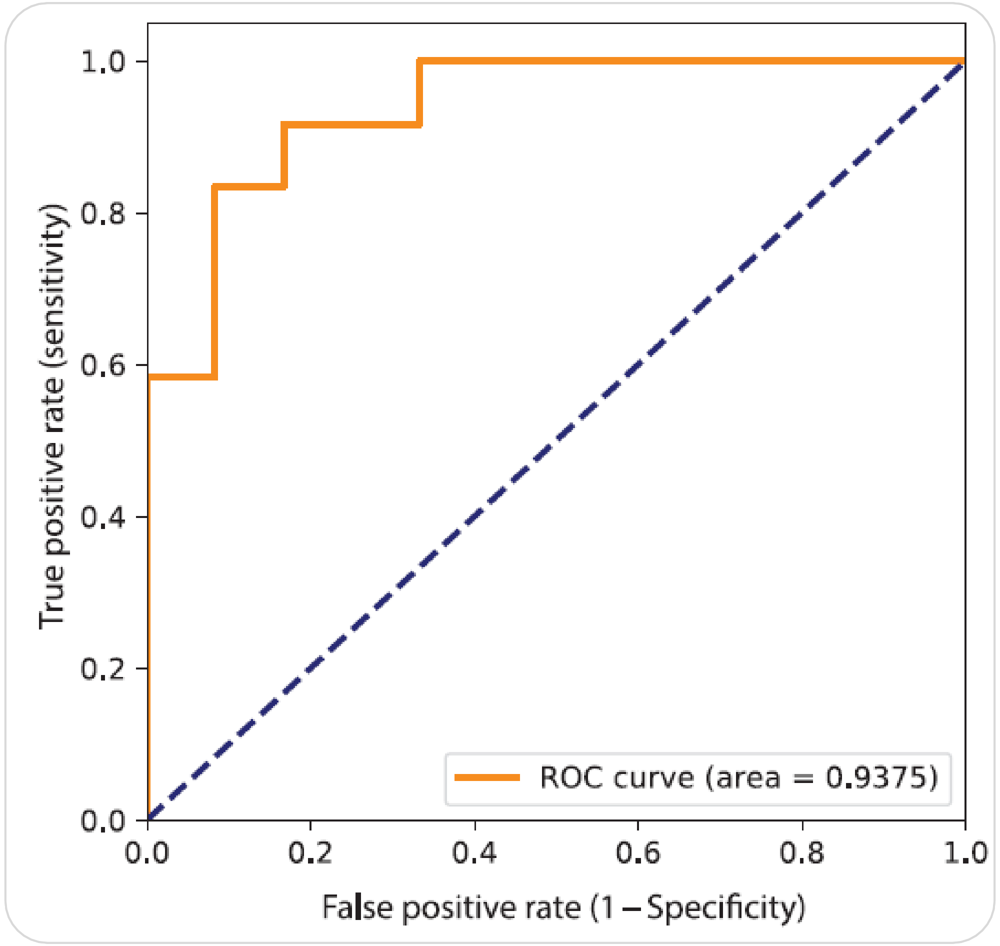
```
y_train_pred_90 = (y_scores >= threshold_for_90_precision)
```

- Result: precision ≈ 90%, recall ≈ 48%

```
>>> precision_score(y_train_5, y_train_pred_90)
0.9000345901072293
>>> recall_at_90_precision = recall_score(y_train_5, y_train_pred_90)
>>> recall_at_90_precision
0.4799852425751706
```

ROC = Receiver Operating Characteristic Curve

- Plots True Positive Rate (Recall) vs False Positive Rate
- FPR = negatives wrongly classified as positives

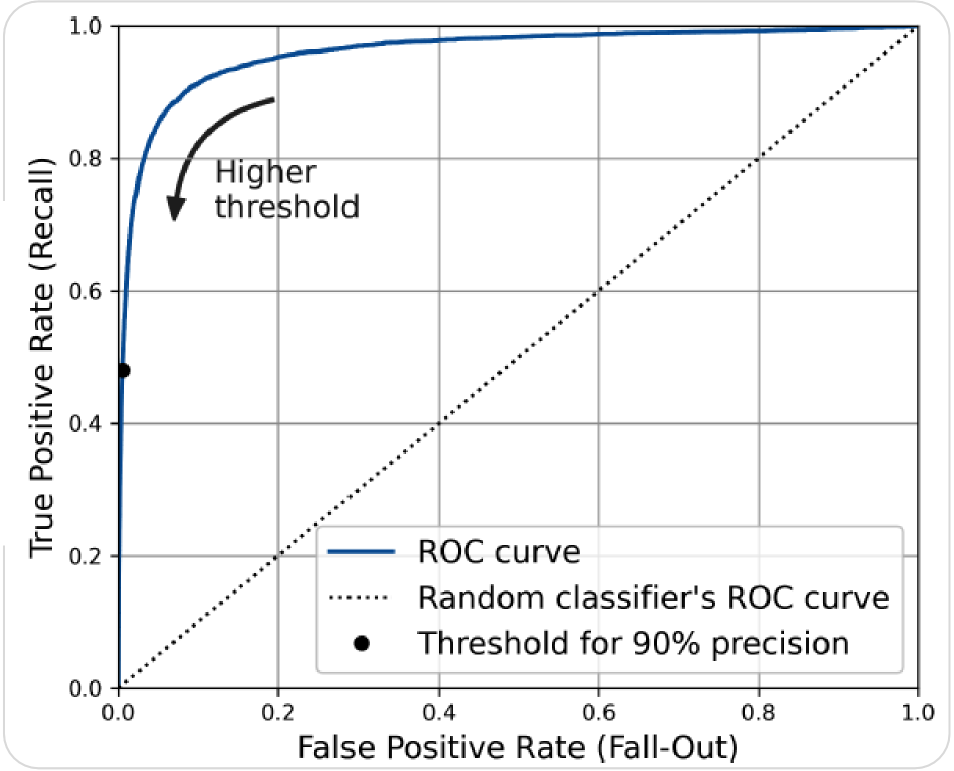


Plotting the ROC Curve

- Use roc_curve() to get FPR, TPR, thresholds
- Then plot FPR (x-axis) vs TPR (y-axis)
- Dotted diagonal = random classifier baseline
- Good models stay near top-left corner

```
from sklearn.metrics import roc_curve

fpr, tpr, thresholds = roc_curve(y_train_5, y_scores)
```

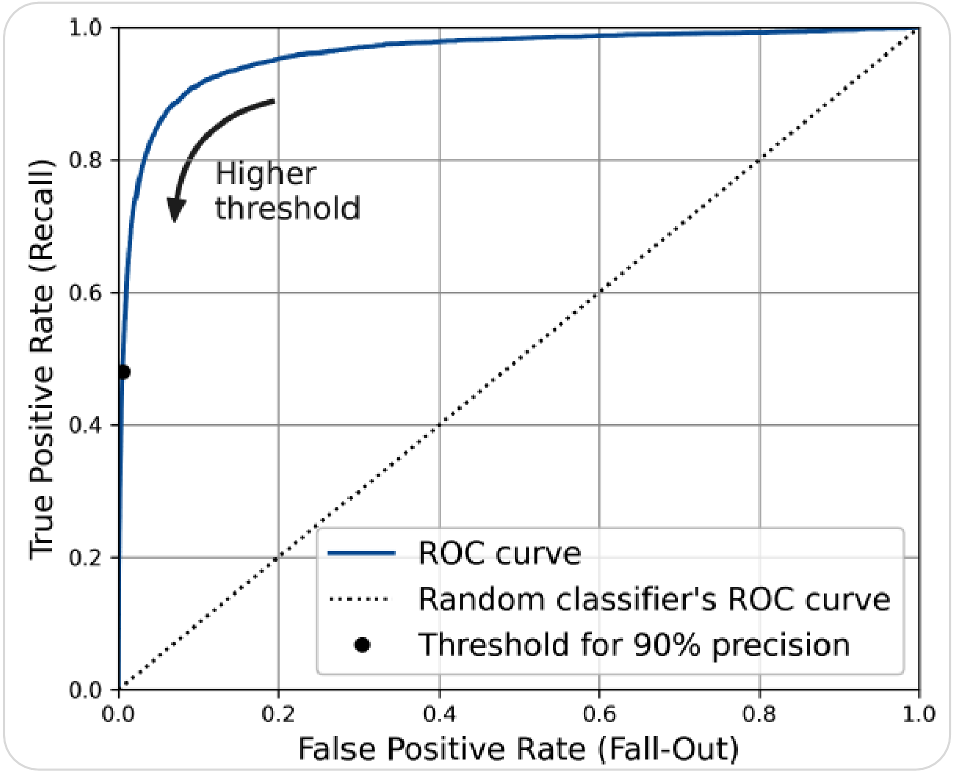
```
idx_for_threshold_at_90 = (thresholds <= threshold_for_90_precision).argmax()
tpr_90, fpr_90 = tpr[idx_for_threshold_at_90], fpr[idx_for_threshold_at_90]

plt.plot(fpr, tpr, linewidth=2, label="ROC curve")
plt.plot([0, 1], [0, 1], 'k:', label="Random classifier's ROC curve")
plt.plot([fpr_90], [tpr_90], "ko", label="Threshold for 90% precision")
[...]  # beautify the figure: add labels, grid, legend, arrow, and text
plt.show()
```

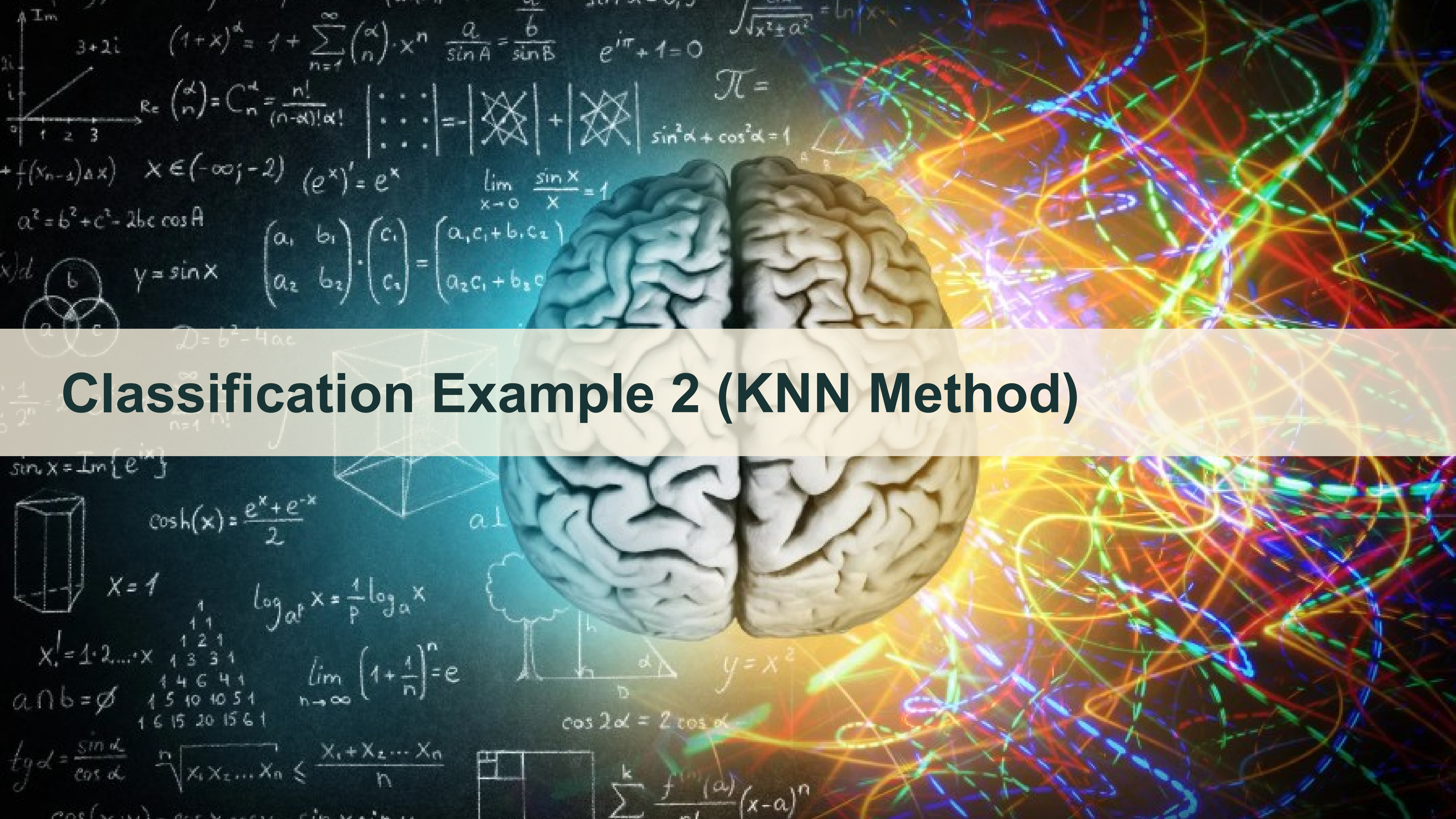


Area Under the Curve (AUC)

- ROC AUC = measure of classifier performance
- Perfect model: AUC = 1.0
- Random model: AUC = 0.5
- Higher AUC → better classifier overall

```
>>> from sklearn.metrics import roc_auc_score
>>> roc_auc_score(y_train_5, y_scores)
0.9604938554008616
```

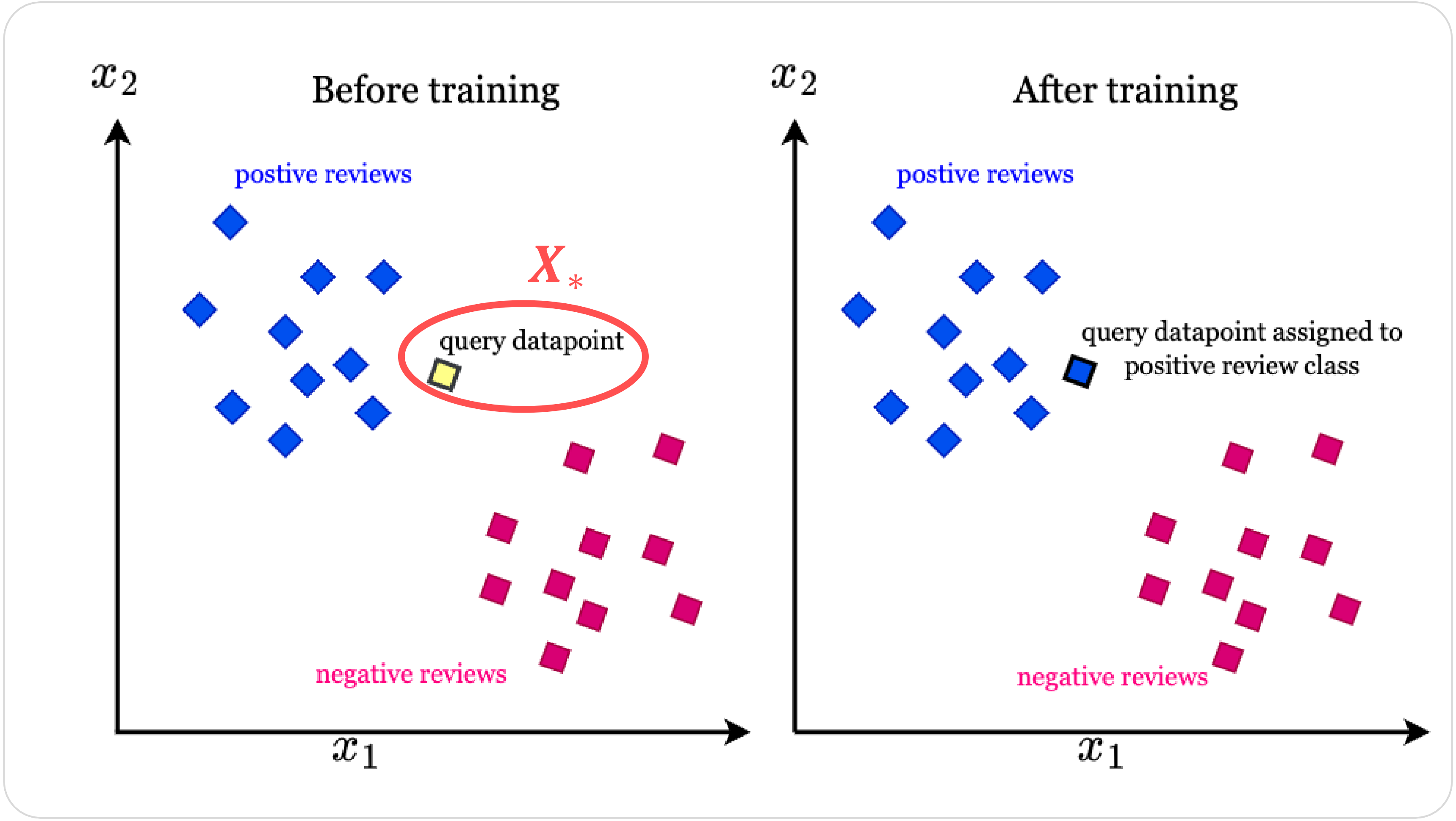




Classification Example 2 (KNN Method)

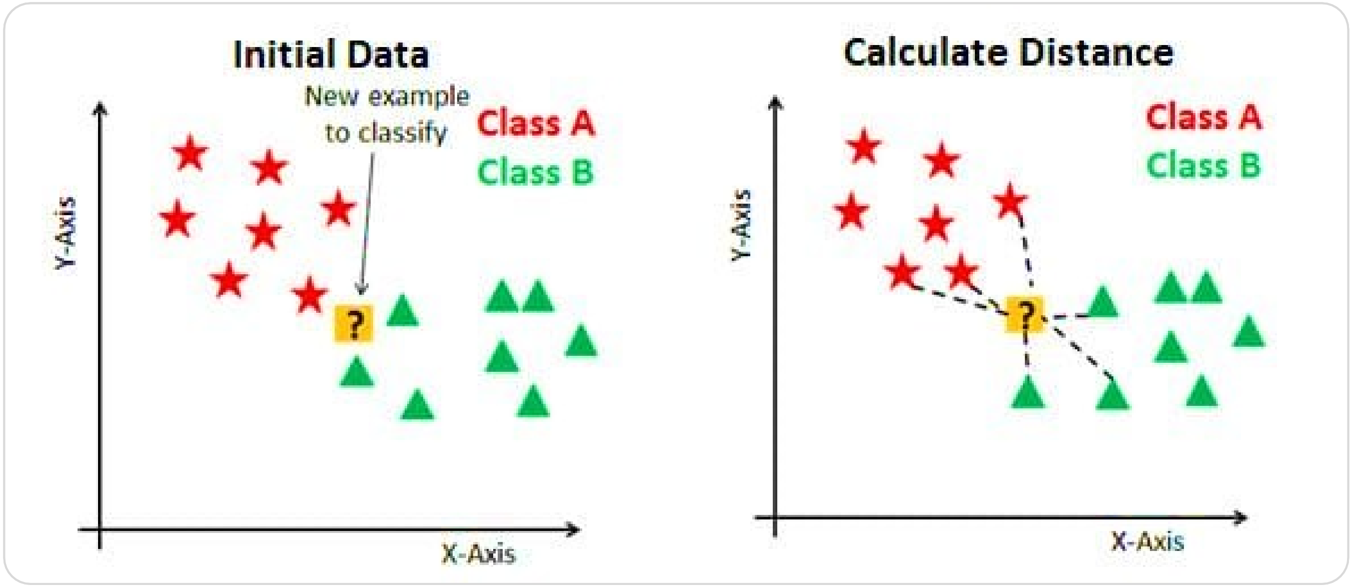
k-Nearest Neighbors (k-NN)

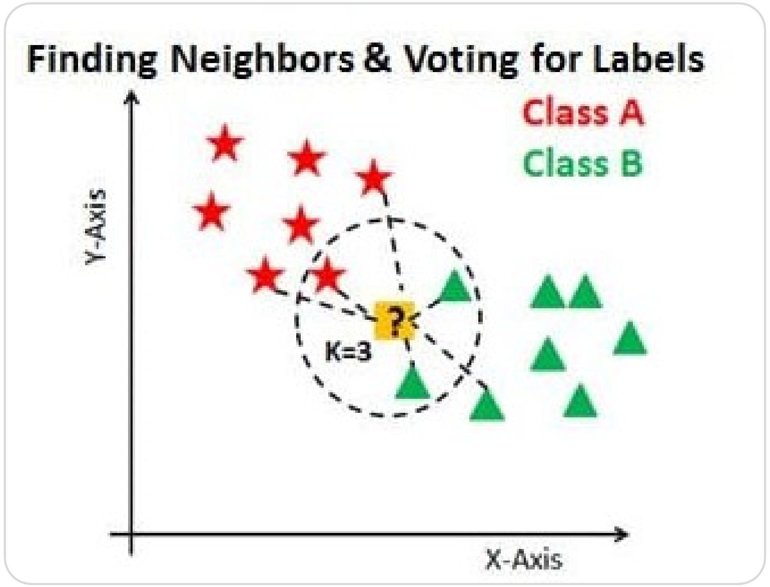
- Predict based on nearby k data points (k-Nearest Neighbors).
  - If a new item is similar to old ones, its output is likely similar too.
  - Non-parametric model



Working of k-NN

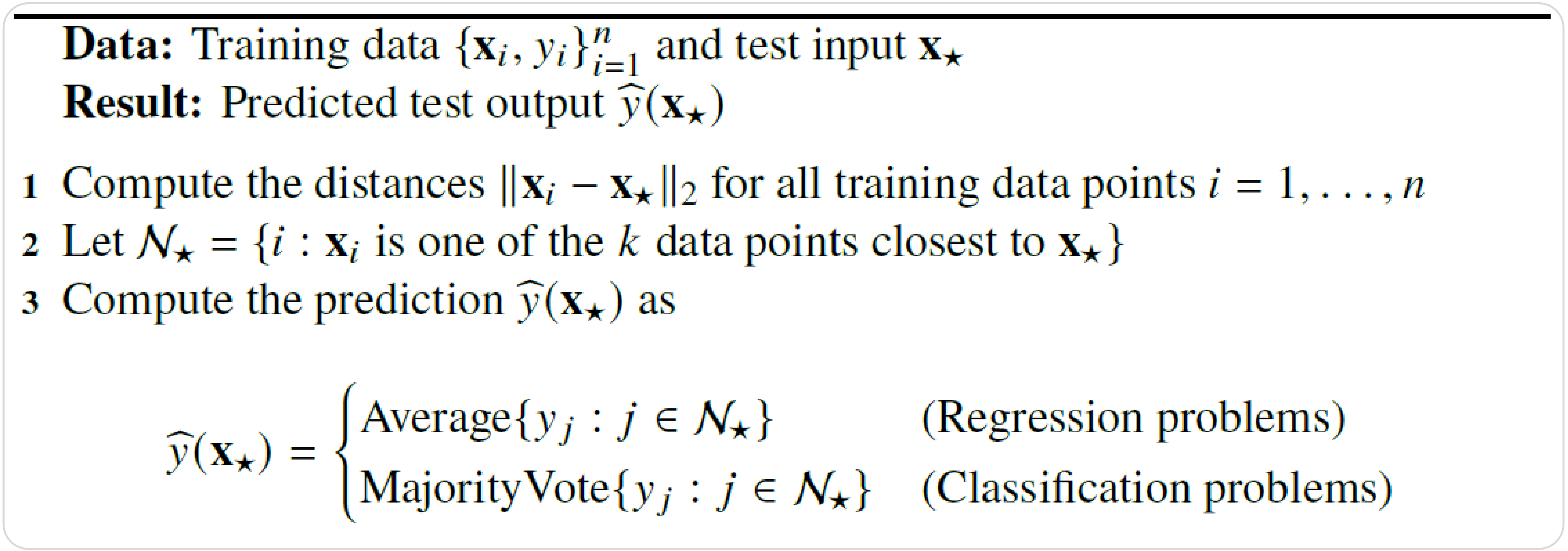
- Measure distance between new point and all old points
- Pick the closest k points (neighbors).
- **For regression:** take the average output
  - A new house price is predicted using prices of 3 nearby houses
- **For classification:** choose the most common output:
  - A fruit is labeled as “apple” if most nearby fruits are apples





Steps of k-NN

- Training data: (X, y) and new data: x★
- Goal: Predict the output (ŷ★) for x★
- Step-1: Measure Distance (use Euclidean distance or another method).
- Step-2: Pick the k closest data points (neighbors)
- Step-3: For regression take the average output and for classification use majority vote.



```
Data: Training data $\{\mathbf{x}_i, y_i\}_{i=1}^{n}$ and test input $\mathbf{x}_\star$
Result: Predicted test output $\widehat{y}(\mathbf{x}_\star)$

1 Compute the distances $\|\mathbf{x}_i - \mathbf{x}_\star\|_2$ for all training data points $i = 1, \ldots, n$
2 Let $\mathcal{N}_\star = \{i : \mathbf{x}_i$ is one of the $k$ data points closest to $\mathbf{x}_\star\}$
3 Compute the prediction $\widehat{y}(\mathbf{x}_\star)$ as
```

$$\widehat{y}(\mathbf{x}_\star) = \begin{cases} \text{Average}\{y_j : j \in \mathcal{N}_\star\} & \text{(Regression problems)} \\ \text{MajorityVote}\{y_j : j \in \mathcal{N}_\star\} & \text{(Classification problems)} \end{cases}$$

Example: Predicting Colors with k-NN

- Training Data (6 Points) and **Test Data Point**: x★ = [1, 2]

Find how far each training point is from x★ = [1, 2]

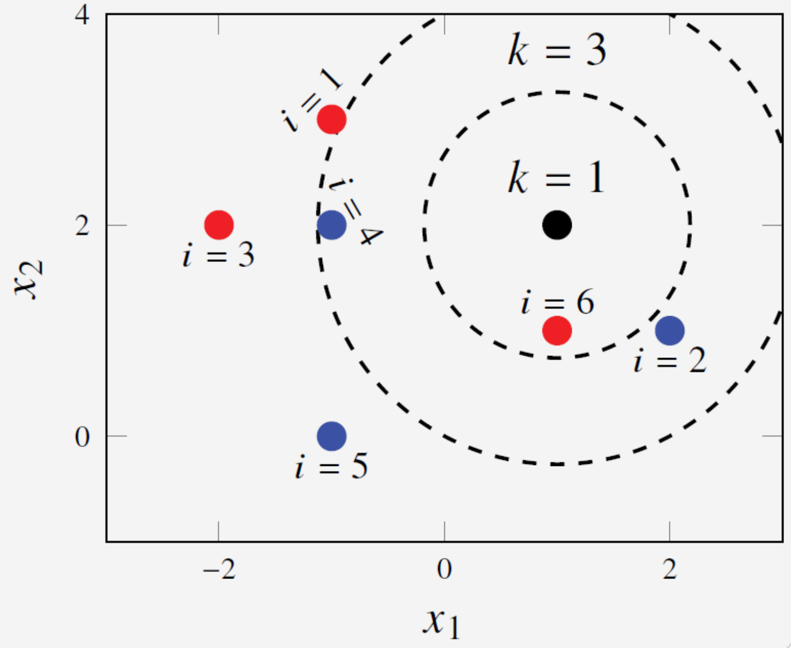
Training Data

i	x_1	x_2	y
1	−1	3	Red
2	2	1	Blue
3	−2	2	Red
4	−1	2	Blue
5	−1	0	Blue
6	1	1	Red

Calculate Distances (Euclidean)

i	$\|\mathbf{x}_i - \mathbf{x}_\star\|_2$	y_i
6	$\sqrt{1}$	Red
2	$\sqrt{2}$	Blue
4	$\sqrt{4}$	Blue
1	$\sqrt{5}$	Red
5	$\sqrt{8}$	Blue
3	$\sqrt{9}$	Red

Fig. 2.3



Prediction with k = 1

- Closest data point: Point 6 → Red
  - Prediction = Red

Find how far each training point is from x★ = [1, 2]

Training Data

i	x_1	x_2	y
1	−1	3	Red
2	2	1	Blue
3	−2	2	Red
4	−1	2	Blue
5	−1	0	Blue
6	1	1	Red

Calculate Distances (Euclidean)

i	$\|\|\mathbf{x}_i - \mathbf{x}_\star\|\|_2$	y_i
6	$\sqrt{1}$	Red
2	$\sqrt{2}$	Blue
4	$\sqrt{4}$	Blue
1	$\sqrt{5}$	Red
5	$\sqrt{8}$	Blue
3	$\sqrt{9}$	Red

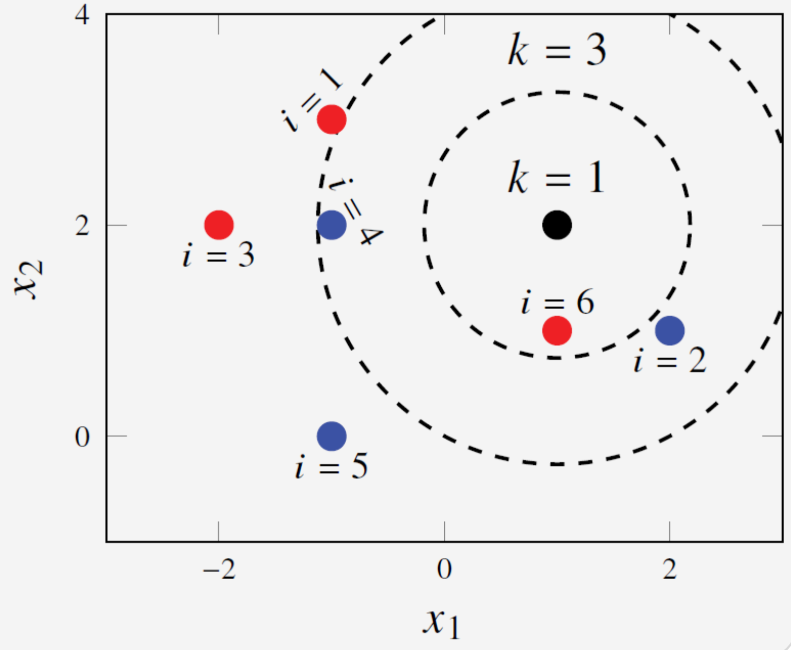


Fig. 2.3

Prediction with k = 3

- Closest data points:
  - Point 6 → Red
  - Point 2 → Blue
  - Point 4 → Blue

- Majority Vote: Blue (2), Red (1)
  - Prediction = Blue

Find how far each training point is from x★ = [1, 2]

Training Data

i	x_1	x_2	y
1	−1	3	Red
2	2	1	Blue
3	−2	2	Red
4	−1	2	Blue
5	−1	0	Blue
6	1	1	Red

Calculate Distances (Euclidean)

i	$\lVert \mathbf{x}_i - \mathbf{x}_\star \rVert_2$	y_i
6	$\sqrt{1}$	Red
2	$\sqrt{2}$	Blue
4	$\sqrt{4}$	Blue
1	$\sqrt{5}$	Red
5	$\sqrt{8}$	Blue
3	$\sqrt{9}$	Red

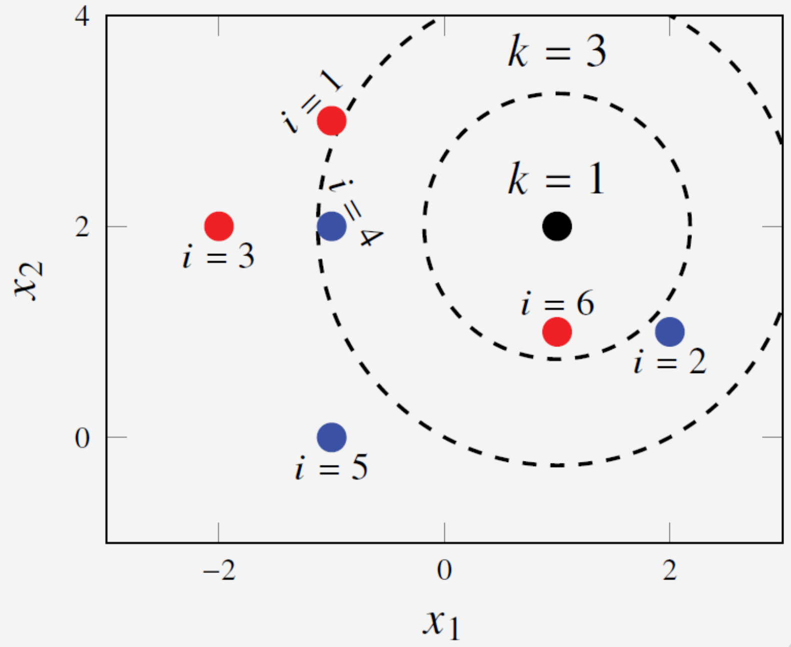
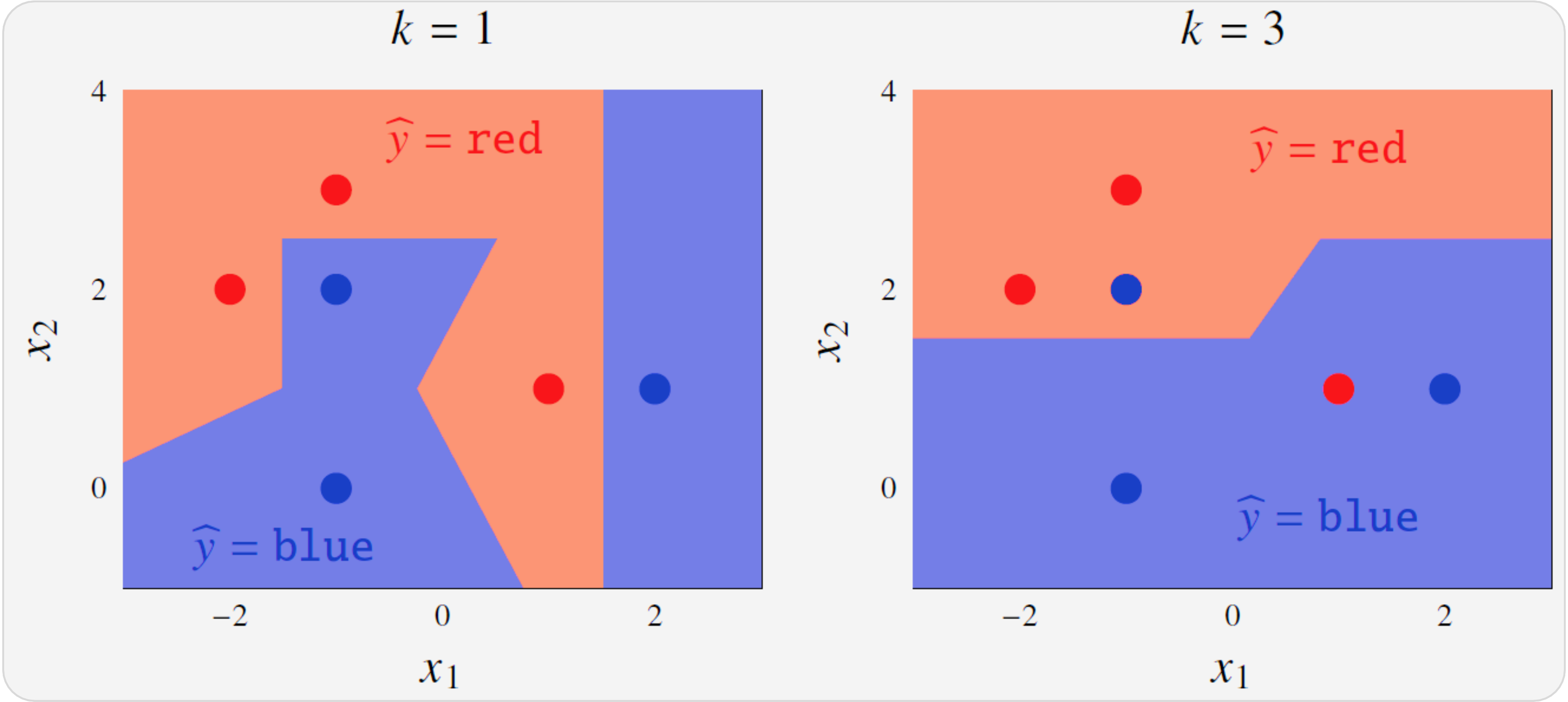


Fig. 2.3

Decision Boundaries for a Classifier

- A decision boundary is the line (or curve) where the predicted class changes.
- It separates regions where the model predicts Red or Blue.
  - For k = 1 → boundaries are more jagged
  - For k = 3 → smoother, more stable regions

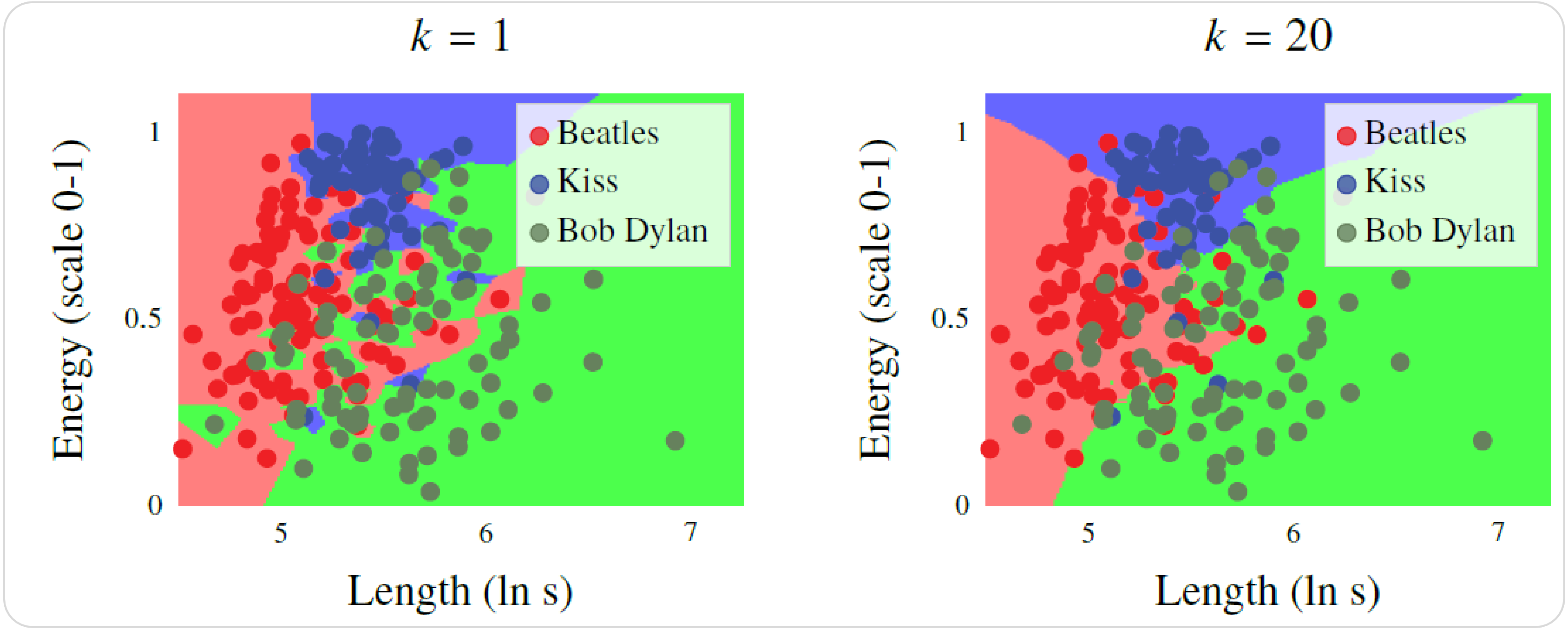


What is a Good Value for K?

- K is a hyperparameter (chosen by user, not learned by model).
- Best k is found by testing different values

Small k (e.g., k = 1): Fits training data too closely (overfitting)

Large k (e.g., k = 20) Fits training data better for generalizing to new data



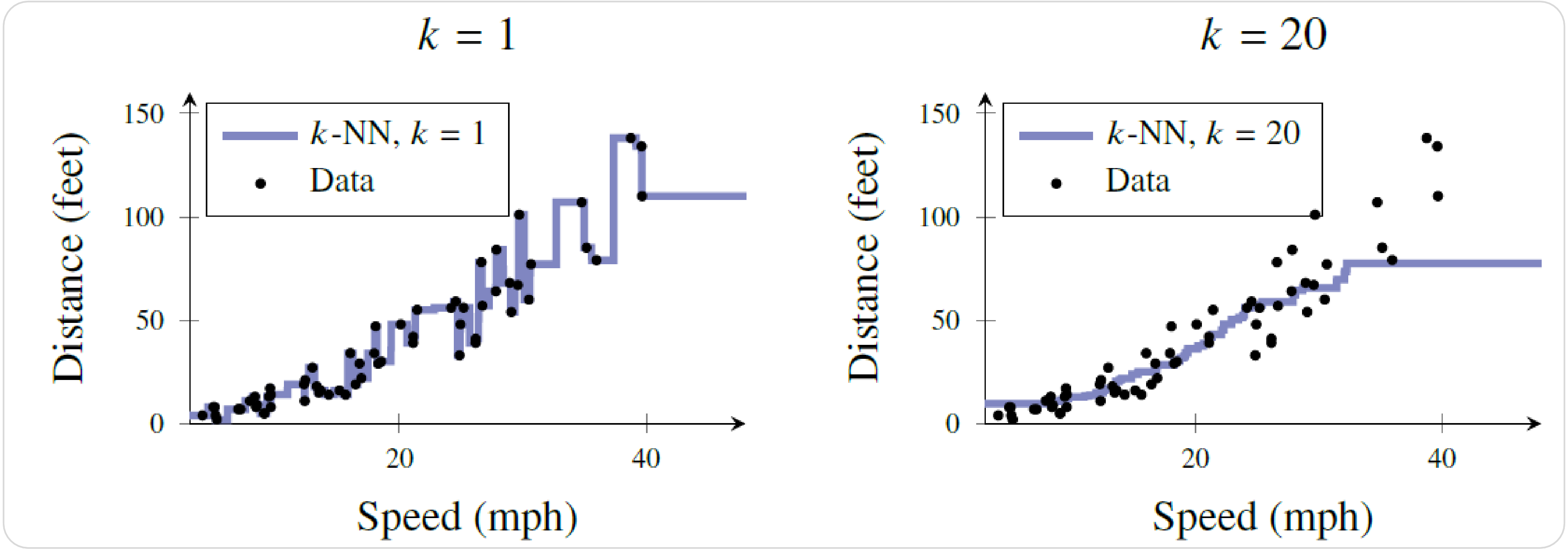
***k*-NN applied to the music classification with K = 1 and K = 20**

What is a Good Value for K?

- K is a hyperparameter (chosen by user, not learned by model).
- Best k is found by testing different values

Small k (e.g., k = 1): Fits training data too closely (overfitting)

Large k (e.g., k = 20) Fits training data better for generalizing to new data



***k*-NN applied to the car stopping distance with K = 1 and K = 20**

Input Normalization in k-NN

- k-NN uses distance to find neighbors

- If one feature has a bigger range, it will dominate the distance:
  - X1: Score [0–1]
  - X2: Height in cm [100–200] → height controls the result

- **Solution:** Normalize your input data

- Two standard ways:
  - **Min–Max scaling:** Squash into [0,1] by **f(x) = (x – min) / (max-min)**
  - **Standardization:** Subtract mean and divide by standard deviation:

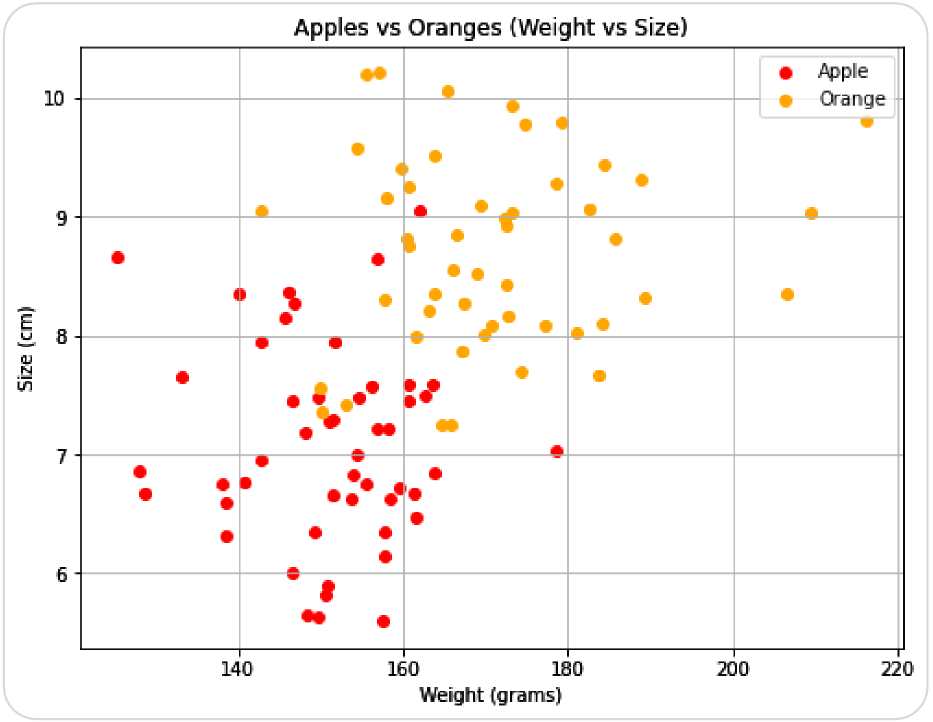
    f(x) = (x-mean)/sdev

KNN in Action — Real Example

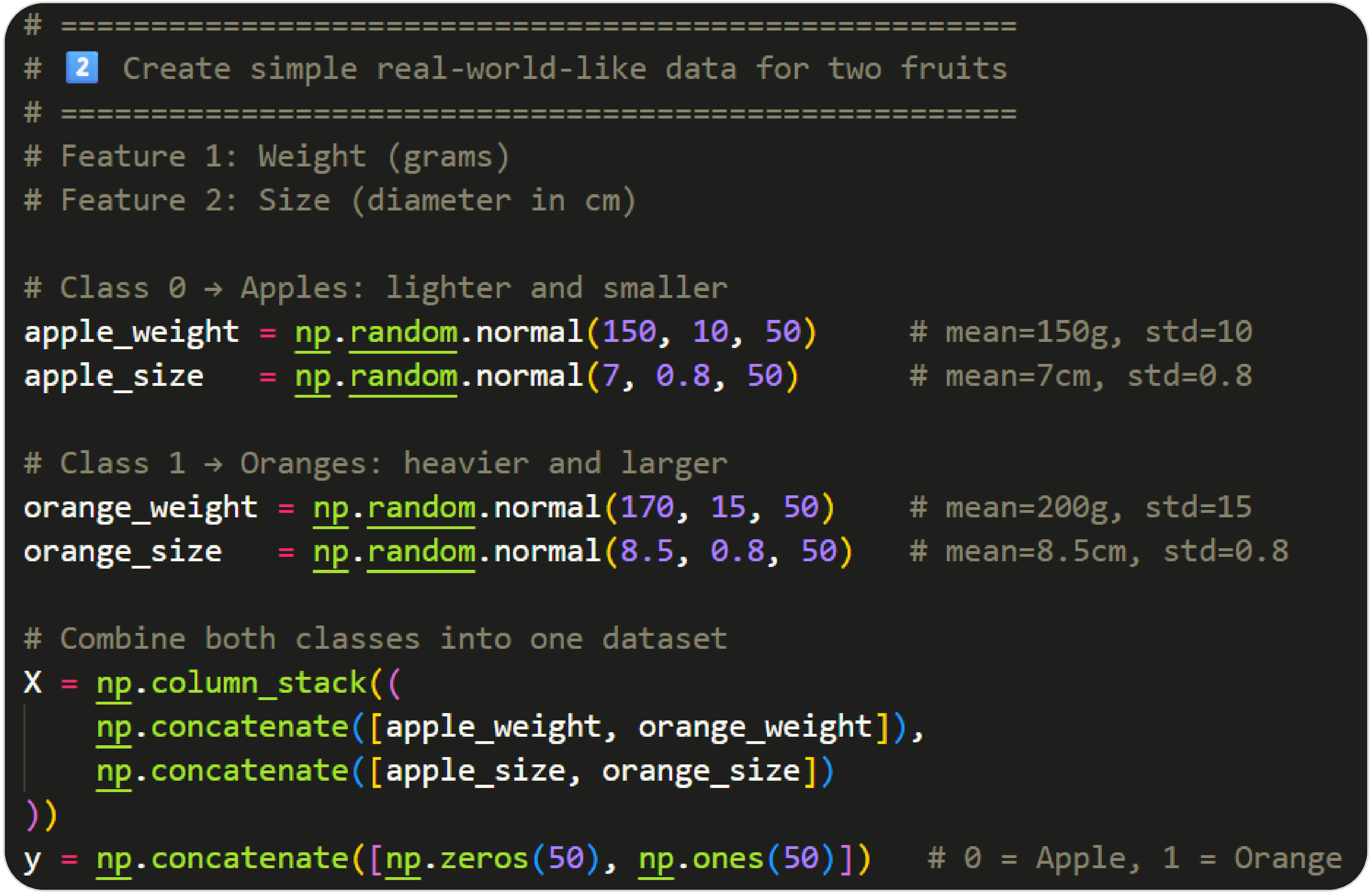
- We classify fruits 🍎 🍊 based on:
  - Weight (grams)
  - Size (diameter in cm)

- **Two classes:**
  - Apple → 0
  - Orange → 1

- **Goal:** Predict fruit type using KNN

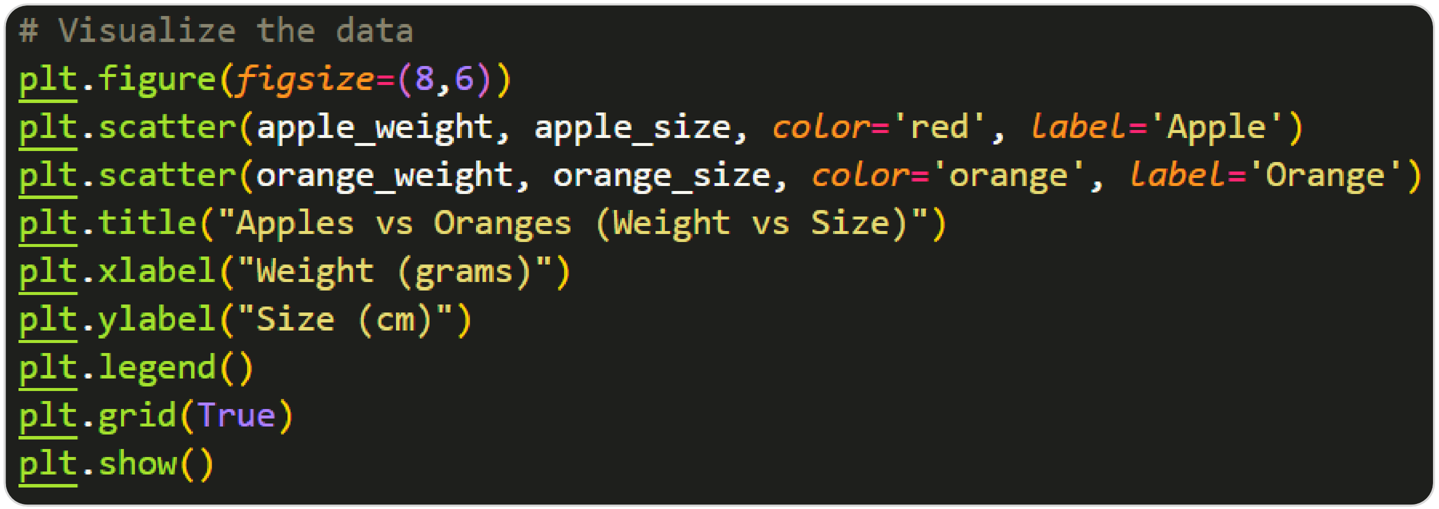


Create simple real-world-like data for two fruits

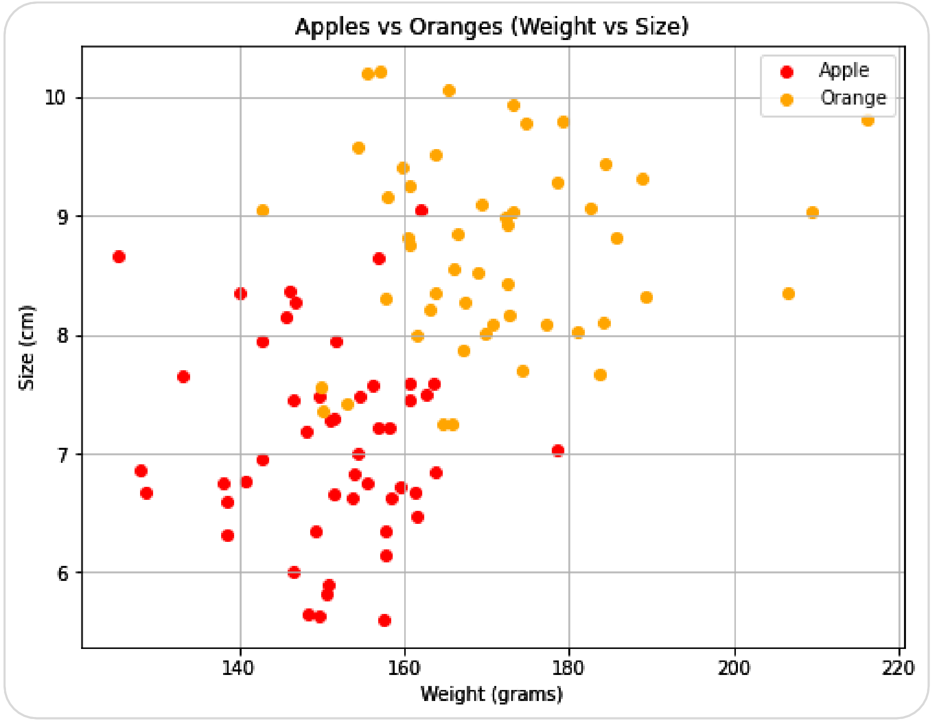


```python
# ====================================================
# 2️⃣ Create simple real-world-like data for two fruits
# ====================================================
# Feature 1: Weight (grams)
# Feature 2: Size (diameter in cm)

# Class 0 → Apples: lighter and smaller
apple_weight = np.random.normal(150, 10, 50)     # mean=150g, std=10
apple_size   = np.random.normal(7, 0.8, 50)      # mean=7cm, std=0.8

# Class 1 → Oranges: heavier and larger
orange_weight = np.random.normal(170, 15, 50)    # mean=200g, std=15
orange_size   = np.random.normal(8.5, 0.8, 50)   # mean=8.5cm, std=0.8

# Combine both classes into one dataset
X = np.column_stack((
    np.concatenate([apple_weight, orange_weight]),
    np.concatenate([apple_size, orange_size])
))
y = np.concatenate([np.zeros(50), np.ones(50)])   # 0 = Apple, 1 = Orange
```

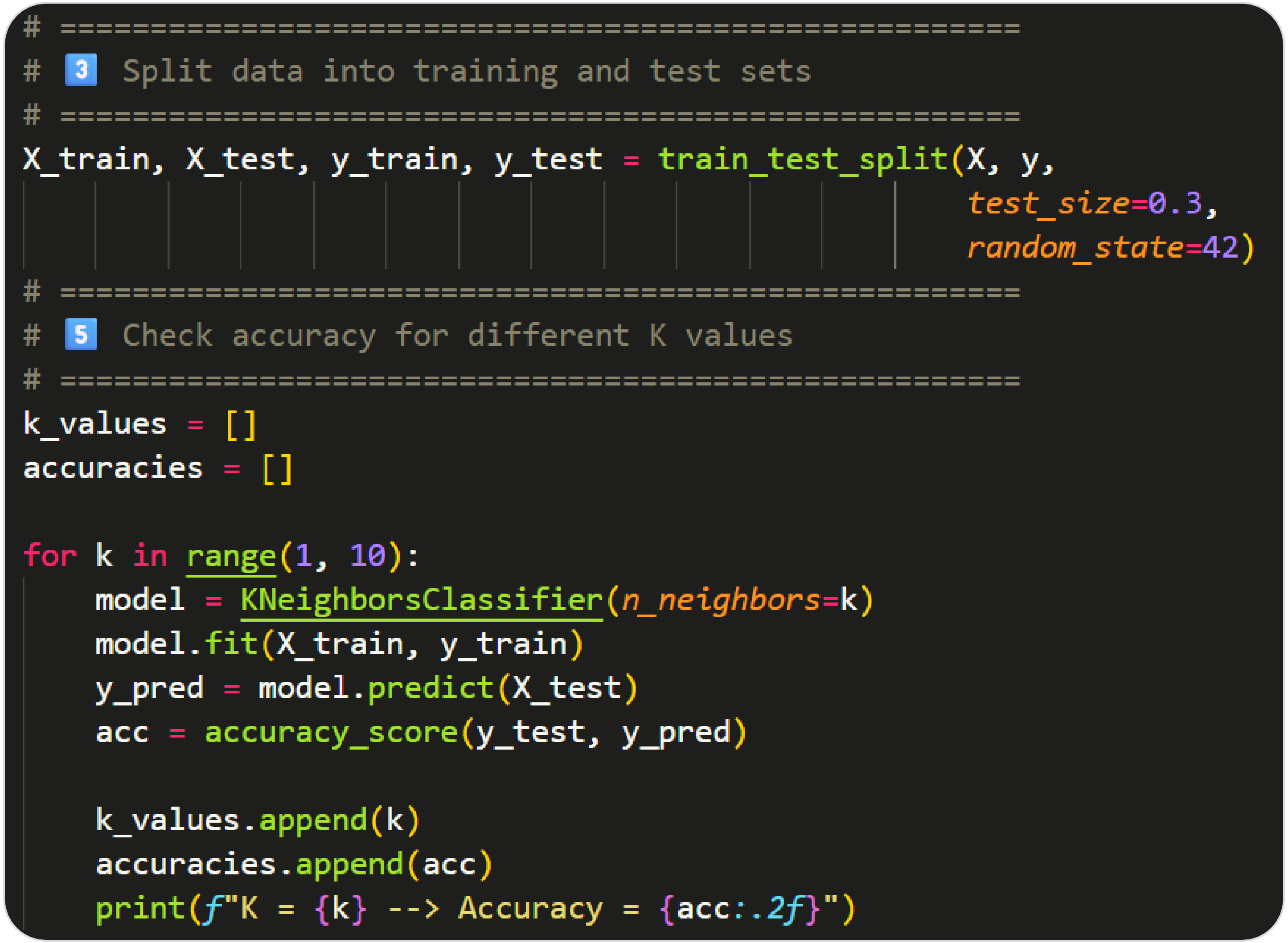
Visualize Data



```python
# Visualize the data
plt.figure(figsize=(8,6))
plt.scatter(apple_weight, apple_size, color='red', label='Apple')
plt.scatter(orange_weight, orange_size, color='orange', label='Orange')
plt.title("Apples vs Oranges (Weight vs Size)")
plt.xlabel("Weight (grams)")
plt.ylabel("Size (cm)")
plt.legend()
plt.grid(True)
plt.show()
```

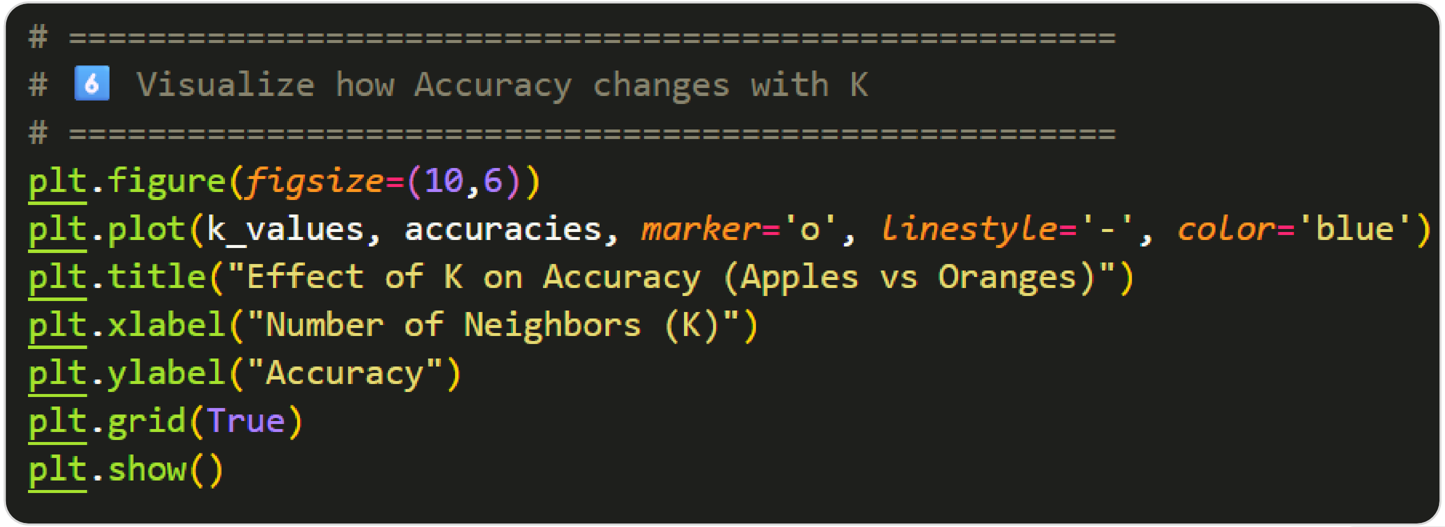


Train KNN Model

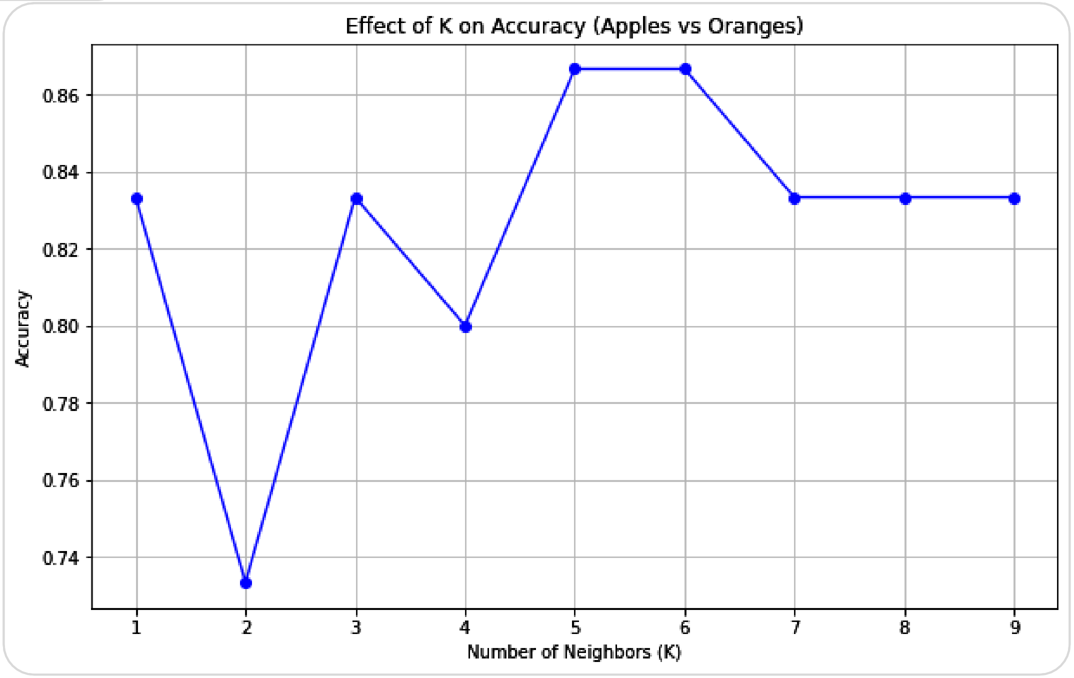


```python
# =====================================================
# 3️⃣ Split data into training and test sets
# =====================================================
X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                    test_size=0.3,
                                                    random_state=42)
# =====================================================
# 5️⃣ Check accuracy for different K values
# =====================================================
k_values = []
accuracies = []

for k in range(1, 10):
    model = KNeighborsClassifier(n_neighbors=k)
    model.fit(X_train, y_train)
    y_pred = model.predict(X_test)
    acc = accuracy_score(y_test, y_pred)

    k_values.append(k)
    accuracies.append(acc)
    print(f"K = {k} --> Accuracy = {acc:.2f}")
```

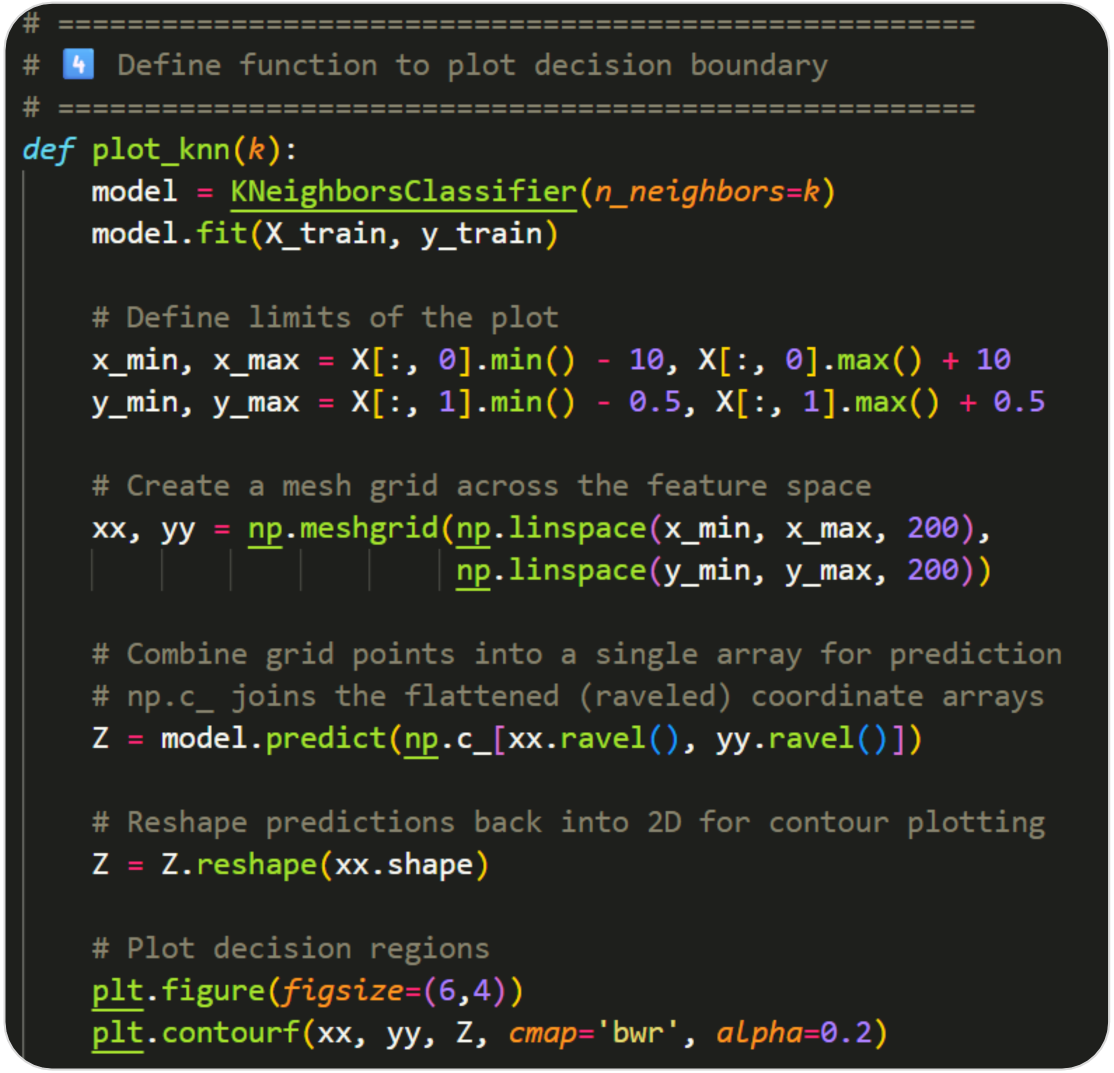
Check Accuracy for Different K



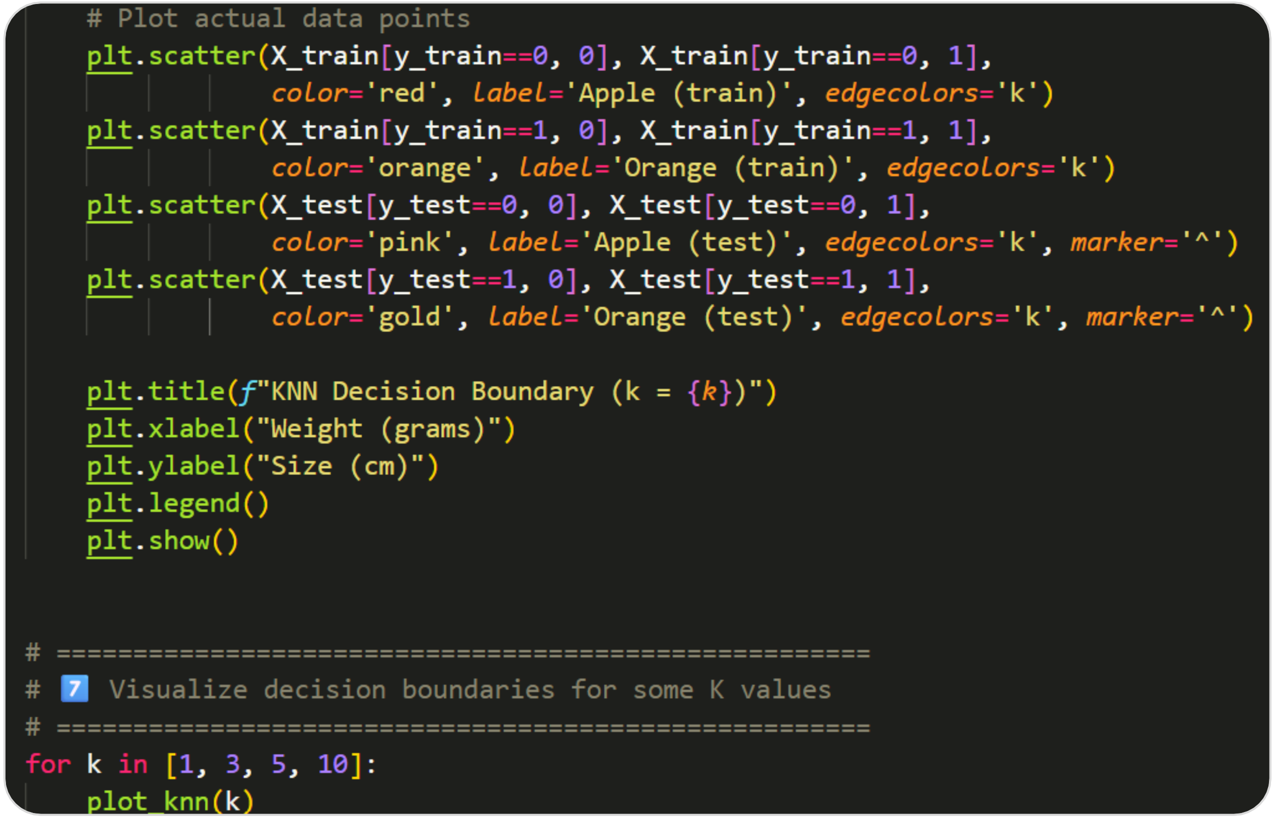
```
# ===========================================================
# 6️⃣ Visualize how Accuracy changes with K
# ===========================================================
plt.figure(figsize=(10,6))
plt.plot(k_values, accuracies, marker='o', linestyle='-', color='blue')
plt.title("Effect of K on Accuracy (Apples vs Oranges)")
plt.xlabel("Number of Neighbors (K)")
plt.ylabel("Accuracy")
plt.grid(True)
plt.show()
```



Visualizing Decision Boundary



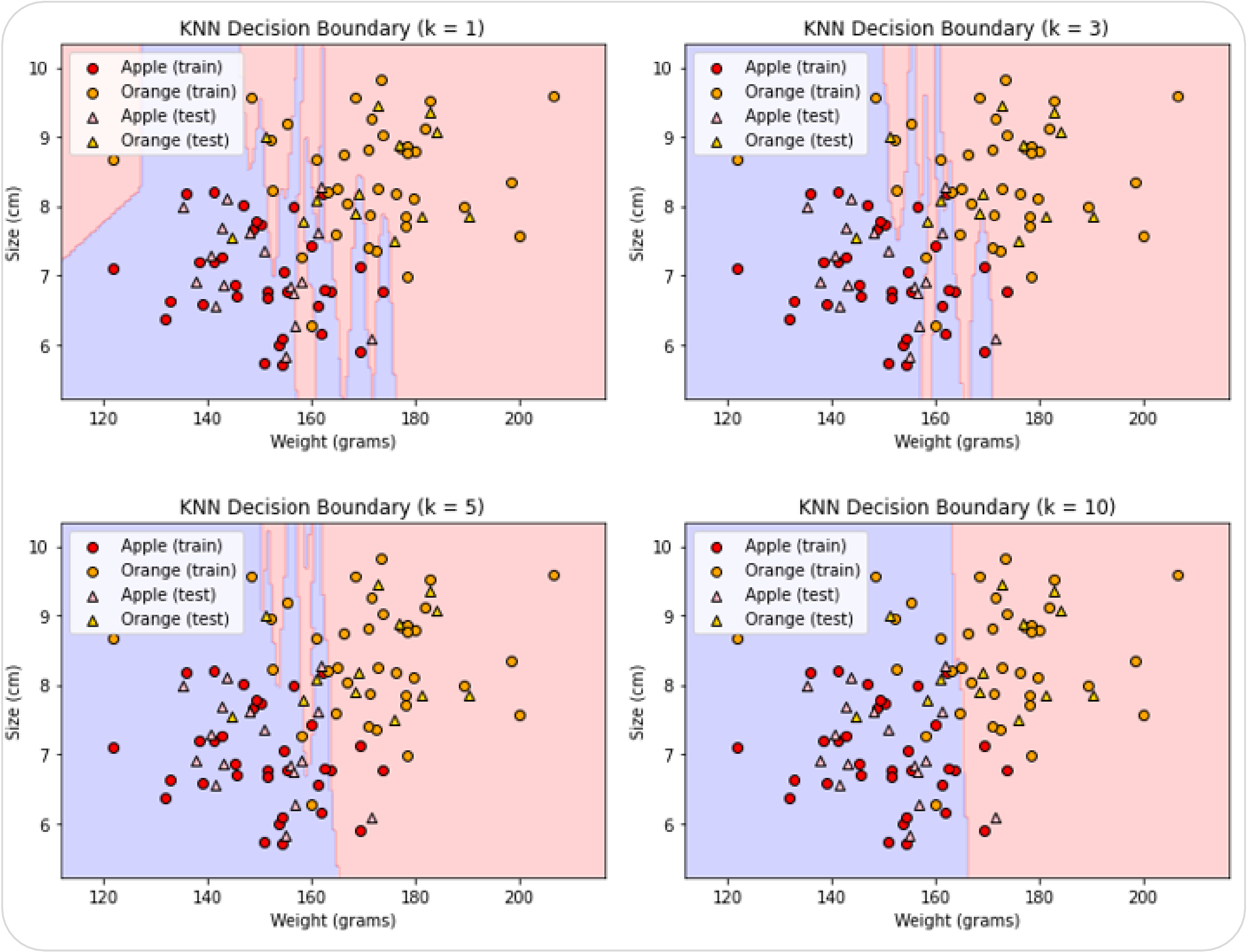
```python
# ======================================================
# 4️⃣ Define function to plot decision boundary
# ======================================================
def plot_knn(k):
    model = KNeighborsClassifier(n_neighbors=k)
    model.fit(X_train, y_train)

    # Define limits of the plot
    x_min, x_max = X[:, 0].min() - 10, X[:, 0].max() + 10
    y_min, y_max = X[:, 1].min() - 0.5, X[:, 1].max() + 0.5

    # Create a mesh grid across the feature space
    xx, yy = np.meshgrid(np.linspace(x_min, x_max, 200),
                         np.linspace(y_min, y_max, 200))

    # Combine grid points into a single array for prediction
    # np.c_ joins the flattened (raveled) coordinate arrays
    Z = model.predict(np.c_[xx.ravel(), yy.ravel()])

    # Reshape predictions back into 2D for contour plotting
    Z = Z.reshape(xx.shape)

    # Plot decision regions
    plt.figure(figsize=(6,4))
    plt.contourf(xx, yy, Z, cmap='bwr', alpha=0.2)
```



```python
    # Plot actual data points
    plt.scatter(X_train[y_train==0, 0], X_train[y_train==0, 1],
                color='red', label='Apple (train)', edgecolors='k')
    plt.scatter(X_train[y_train==1, 0], X_train[y_train==1, 1],
                color='orange', label='Orange (train)', edgecolors='k')
    plt.scatter(X_test[y_test==0, 0], X_test[y_test==0, 1],
                color='pink', label='Apple (test)', edgecolors='k', marker='^')
    plt.scatter(X_test[y_test==1, 0], X_test[y_test==1, 1],
                color='gold', label='Orange (test)', edgecolors='k', marker='^')

    plt.title(f"KNN Decision Boundary (k = {k})")
    plt.xlabel("Weight (grams)")
    plt.ylabel("Size (cm)")
    plt.legend()
    plt.show()


# ======================================================
# 7️⃣ Visualize decision boundaries for some K values
# ======================================================
for k in [1, 3, 5, 10]:
    plot_knn(k)
```

Visualizing Decision Boundary

- Small K → high variance (overfitting)
- Large K → high bias (underfitting)
- Best K → balance between both
- Visual + accuracy helps find optimal K


KNN Decision Boundary (k = 1)
KNN Decision Boundary (k = 3)
KNN Decision Boundary (k = 5)
KNN Decision Boundary (k = 10)
Apple (train)
Orange (train)
Apple (test)
Orange (test)
Size (cm)
Weight (grams)