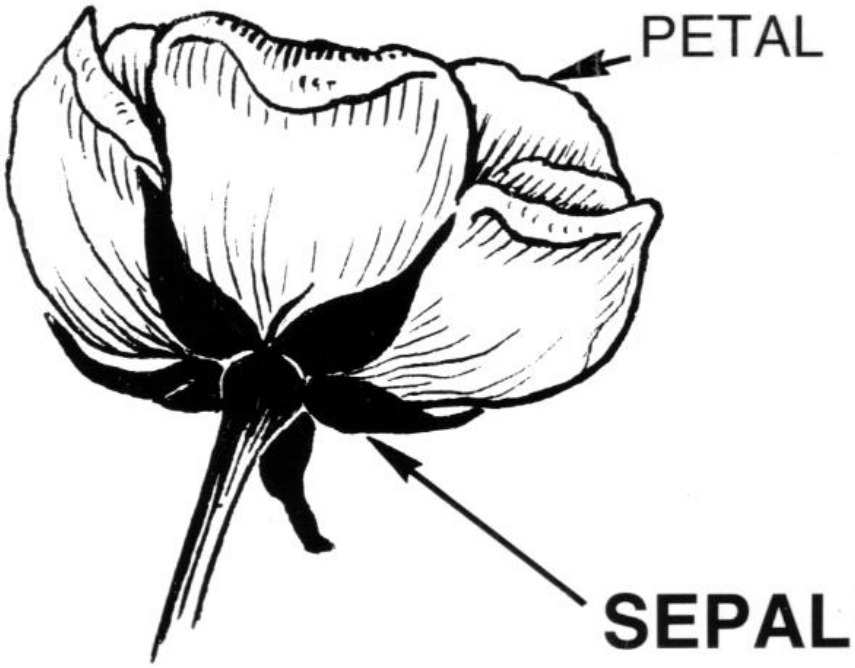
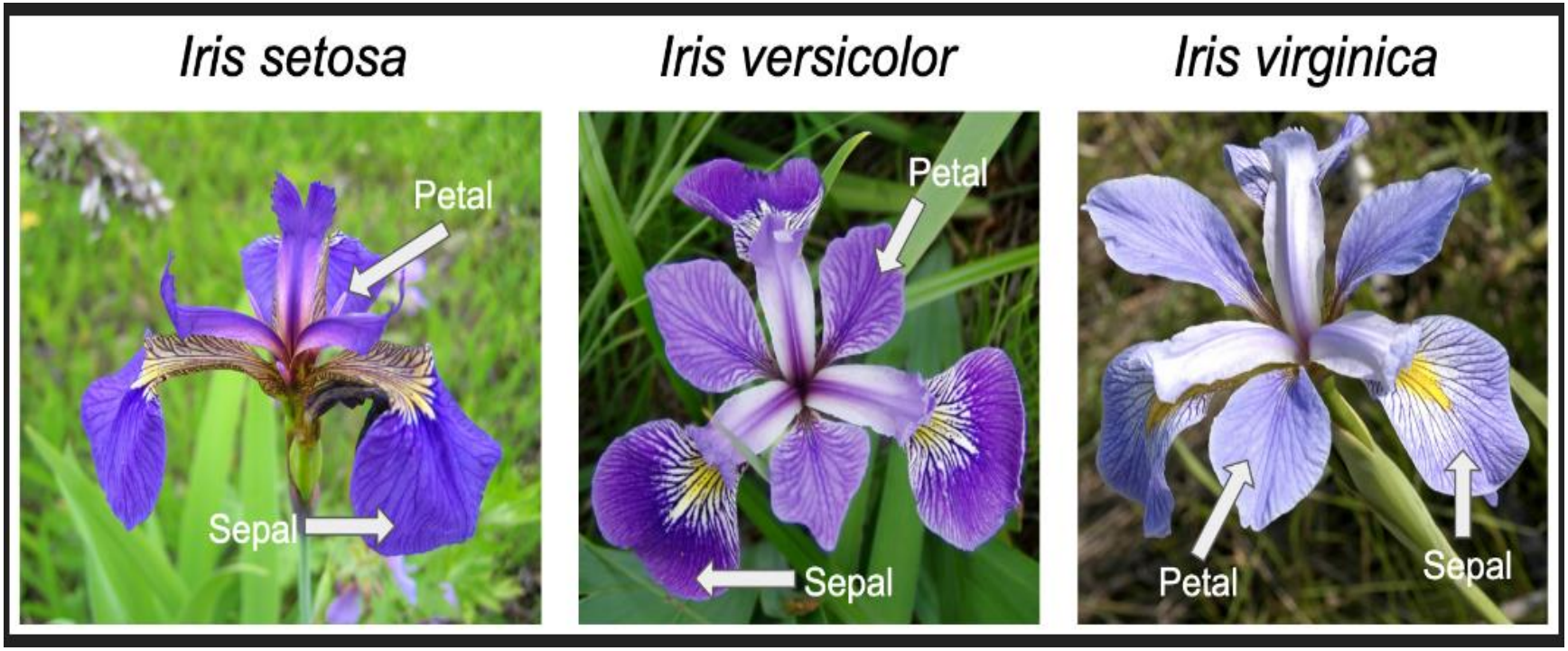
Example Iris Dataset

The Iris dataset is a staple of machine learning. It contains a bunch of measurements for 150 flowers representing three species of iris. For each flower we have its petal length, petal width, sepal length, and sepal width, as well as its species. You can download it from https://archive.ics.uci.edu/ml/datasets/iris:




Iris setosa
Petal
Sepal
Iris versicolor
Petal
Sepal
Iris virginica
Petal
Sepal

Example: The Iris Dataset

```
import requests
data = requests.get( "https://archive.ics.uci.edu/ml/machine-learningdatabases/ iris/iris.data")
with open('iris.dat', 'w') as f:
        f.write(data.txt)
```

The data is comma-separated, with fields:

```
sepal_length, sepal_width, petal_length, petal_width, class
```

For example, the first row looks like:

```
5.1,3.5,1.4,0.2,Iris-setosa
```

Example: The Iris Dataset: Code Snippet

- Build a model that can predict the class (that is, the species) from the first four measurements.

```
from typing import Dict
import csv
from collections import defaultdict
def parse_iris_row(row: List[str]) -> LabeledPoint:
    """“ sepal_length, sepal_width, petal_length, petal_width, class """
    measurements = [float(value) for value in row[:-1]]
    # class is e.g. "Iris-virginica"; we just want "virginica"
    label = row[-1].split("-")[-1]
    return LabeledPoint(measurements, label)
with open('iris.data') as f:
    reader = csv.reader(f)
    iris_data = [parse_iris_row(row) for row in reader]
    # We'll also group just the points by species/label so we can plot them
points_by_species: Dict[str, List[Vector]] = defaultdict(list)
for iris in iris_data:
    points_by_species[iris.label].append(iris.point)
```

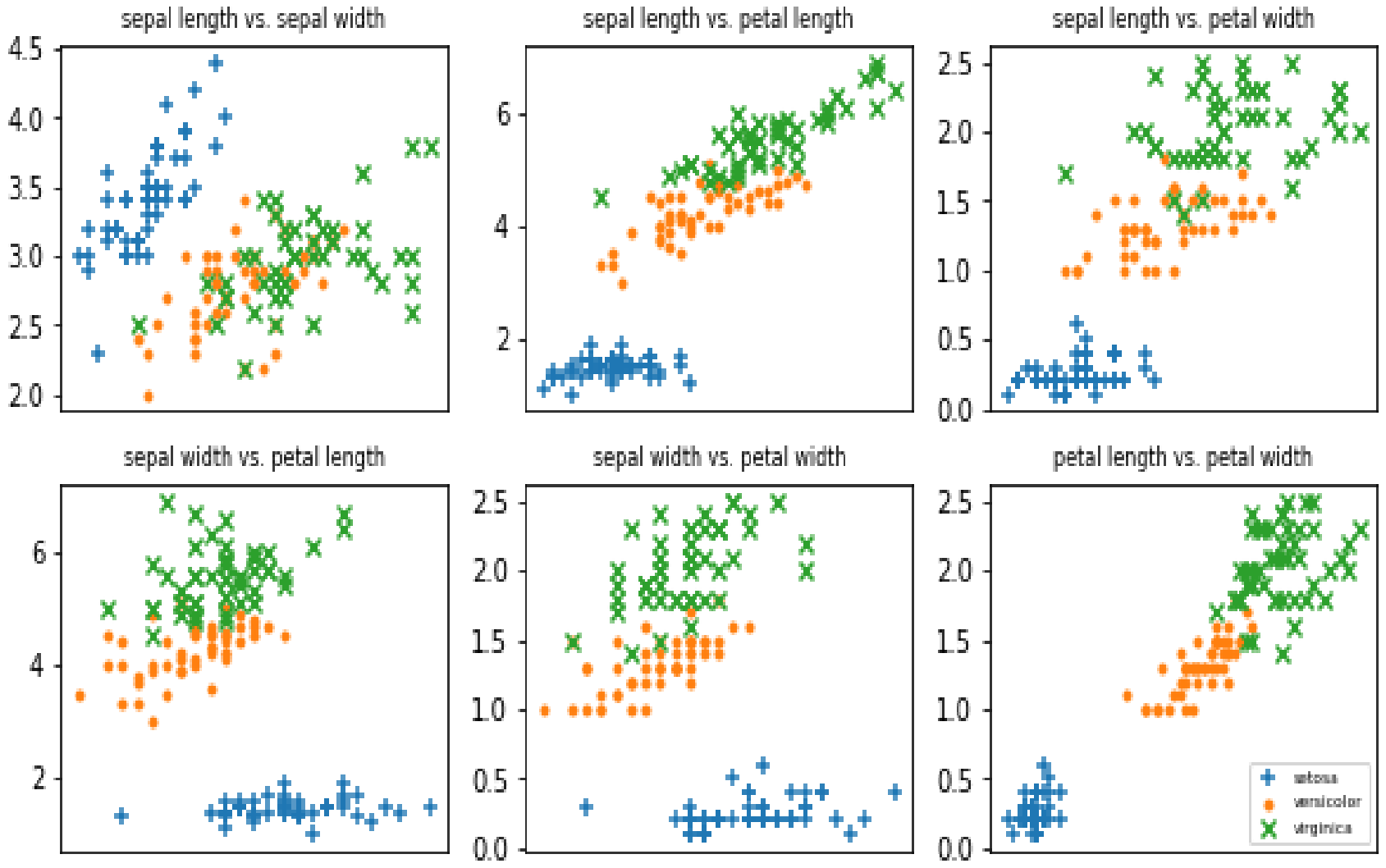
Example: The Iris Dataset Code Snippet

```
import random
from scratch.machine_learning import split_data
random.seed(12)
iris_train, iris_test = split_data(iris_data, 0.70)
assert len(iris_train) == 0.7 * 150
assert len(iris_test) == 0.3 * 150
from typing import Tuple
# track how many times we see (predicted, actual)
confusion_matrix: Dict[Tuple[str, str], int] = defaultdict(int)
num_correct = 0
for iris in iris_test:
    predicted = knn_classify(5, iris_train, iris.point)
    actual = iris.label
    if predicted == actual:
            num_correct += 1
    confusion_matrix[(predicted, actual)] += 1
pct_correct = num_correct / len(iris_test)
print(pct_correct, confusion_matrix)
```

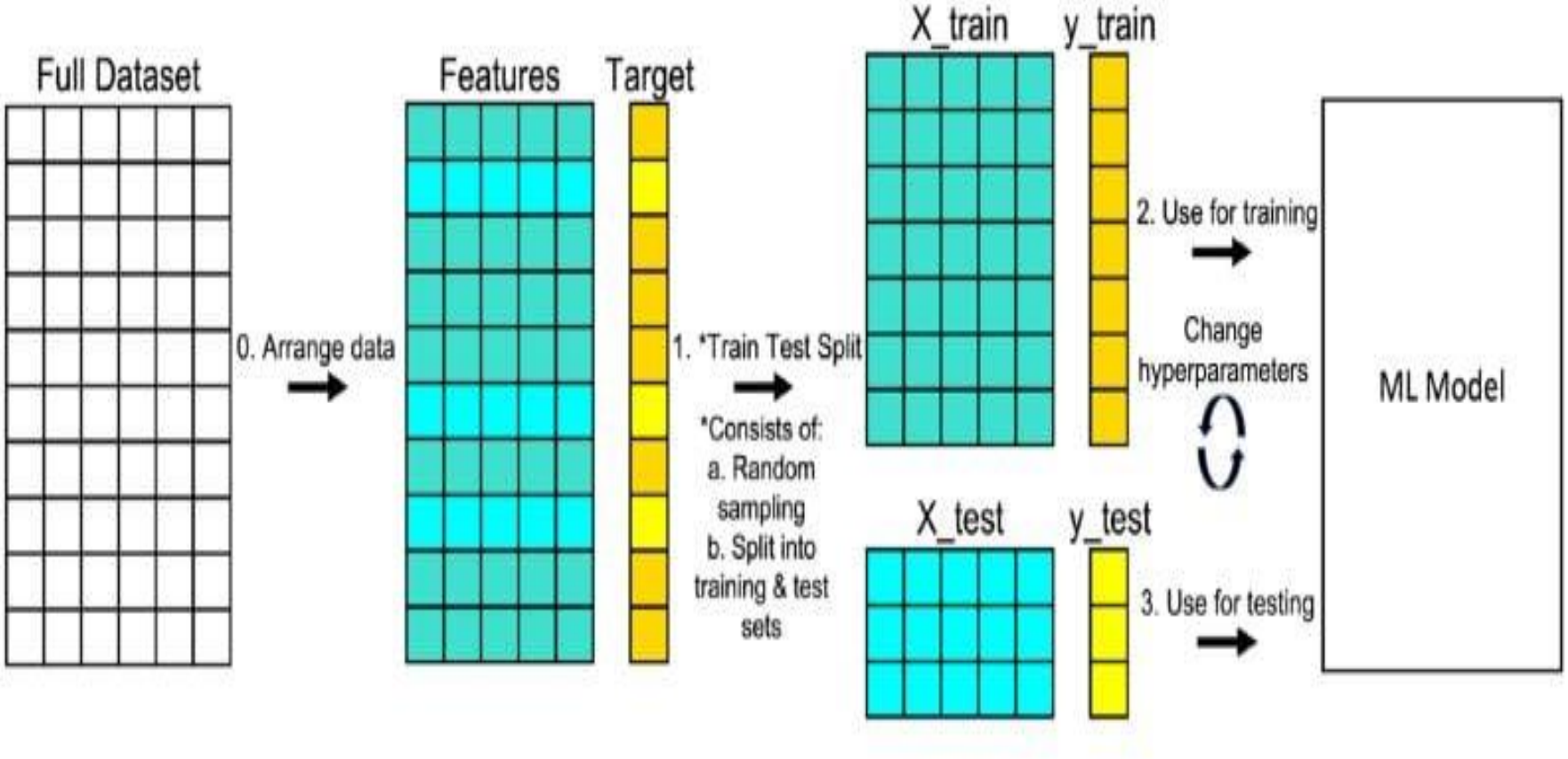
Example: The Iris Dataset: Code Snippet

```
from matplotlib import pyplot as plt
metrics = ['sepal length', 'sepal width', 'petal length', 'petal width']
pairs = [(i, j) for i in range(4) for j in range(4) if i < j]
marks = ['+', '.', 'x'] # we have 3 classes, so 3 markers
fig, ax = plt.subplots(2, 3)
for row in range(2):
    for col in range(3):
        i, j = pairs[3 * row + col]
        ax[row][col].set_title(f"{metrics[i]} vs {metrics[j]}", fontsize=8)
        ax[row][col].set_xticks([])
        ax[row][col].set_yticks([])
        for mark, (species, points) in zip(marks,
        points_by_species.items()):
        xs = [point[i] for point in points]
        ys = [point[j] for point in points]
        ax[row][col].scatter(xs, ys, marker=mark, label=species)
ax[-1][-1].legend(loc='lower right', prop={'size': 6})
plt.show()
```

Example: The Iris Dataset



Example: The Iris Dataset



KNN Implementation using Libraries

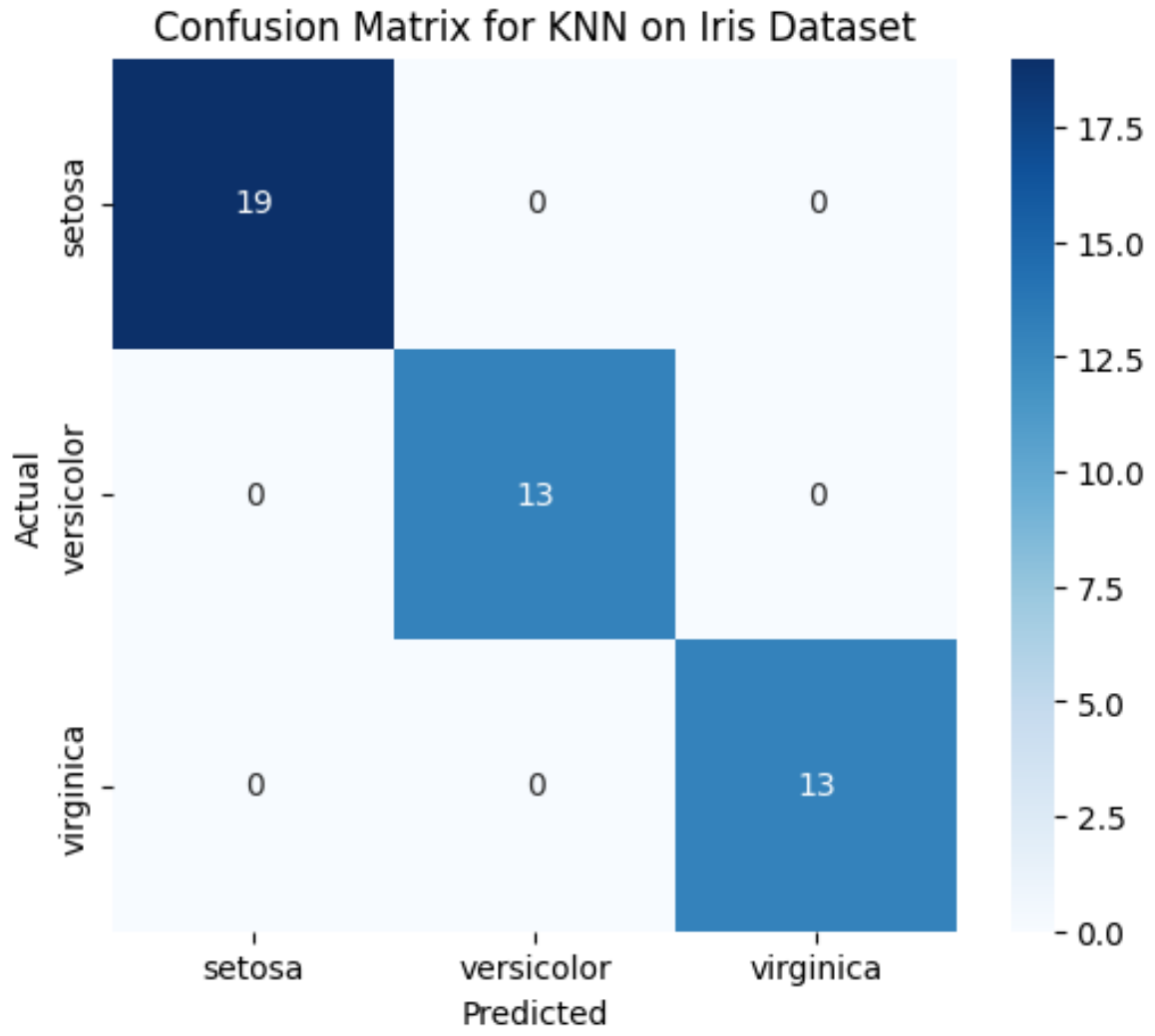
```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score, confusion_matrix
import matplotlib.pyplot as plt
import seaborn as sns

# Step 1: Load the Iris dataset
iris = load_iris()
X = iris.data  # Features: sepal length, sepal width, etc.
y = iris.target  # Labels: 0=setosa, 1=versicolor, 2=virginica
target_names = iris.target_names

# Step 2: Split into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.3, random_state=42)
```

KNN Implementation using Libraries

```
# Step 3: Create and train KNN model
knn = KNeighborsClassifier(n_neighbors=3)
knn.fit(X_train, y_train)

# Step 4: Predict
y_pred = knn.predict(X_test)

# Step 5: Accuracy and confusion matrix
acc = accuracy_score(y_test, y_pred)
print(f"Accuracy: {acc:.2f}")

# Confusion Matrix
cm = confusion_matrix(y_test, y_pred)

# Step 6: Visualize confusion matrix
plt.figure(figsize=(6, 5))
sns.heatmap(cm, annot=True, cmap="Blues", fmt='d',
            xticklabels=target_names, yticklabels=target_names)
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.title("Confusion Matrix for KNN on Iris Dataset")
plt.show()
```

Example: The Iris Dataset

```
Accuracy: 1.00
```



The Curse of Dimensionality

- The **curse of dimensionality** refers to a set of problems and challenges that arise when working with **high-dimensional data** (i.e., data with many features or variables)
- The k-nearest neighbors algorithm runs into trouble in higher dimensions.
  - **Data Sparsity**: In high-dimensional space, data points are very far apart. Models struggle to find patterns, and overfitting becomes likely.
  - **Distance Metrics** Become Useless Algorithms
  - **Computational Cost**: More features = more computations. High-dimensional data increases time complexity and memory usage for many algorithms.
  - **Overfitting** Risk Models can easily find patterns in high dimensions that are just noise.

The Curse of Dimensionality

- The **curse of dimensionality** refers to a set of problems and challenges that arise when working with **high-dimensional data** (i.e., data with many features or variables)
- The k-nearest neighbors algorithm runs into trouble in higher dimensions.
  - **Data Sparsity**: In high-dimensional space, data points are very far apart. Models struggle to find patterns, and overfitting becomes likely.
  - **Distance Metrics** Become Useless Algorithms
  - **Computational Cost**: More features = more computations. High-dimensional data increases time complexity and memory usage for many algorithms.
  - **Overfitting** Risk Models can easily find patterns in high dimensions that are just noise.

The Curse of Dimensionality

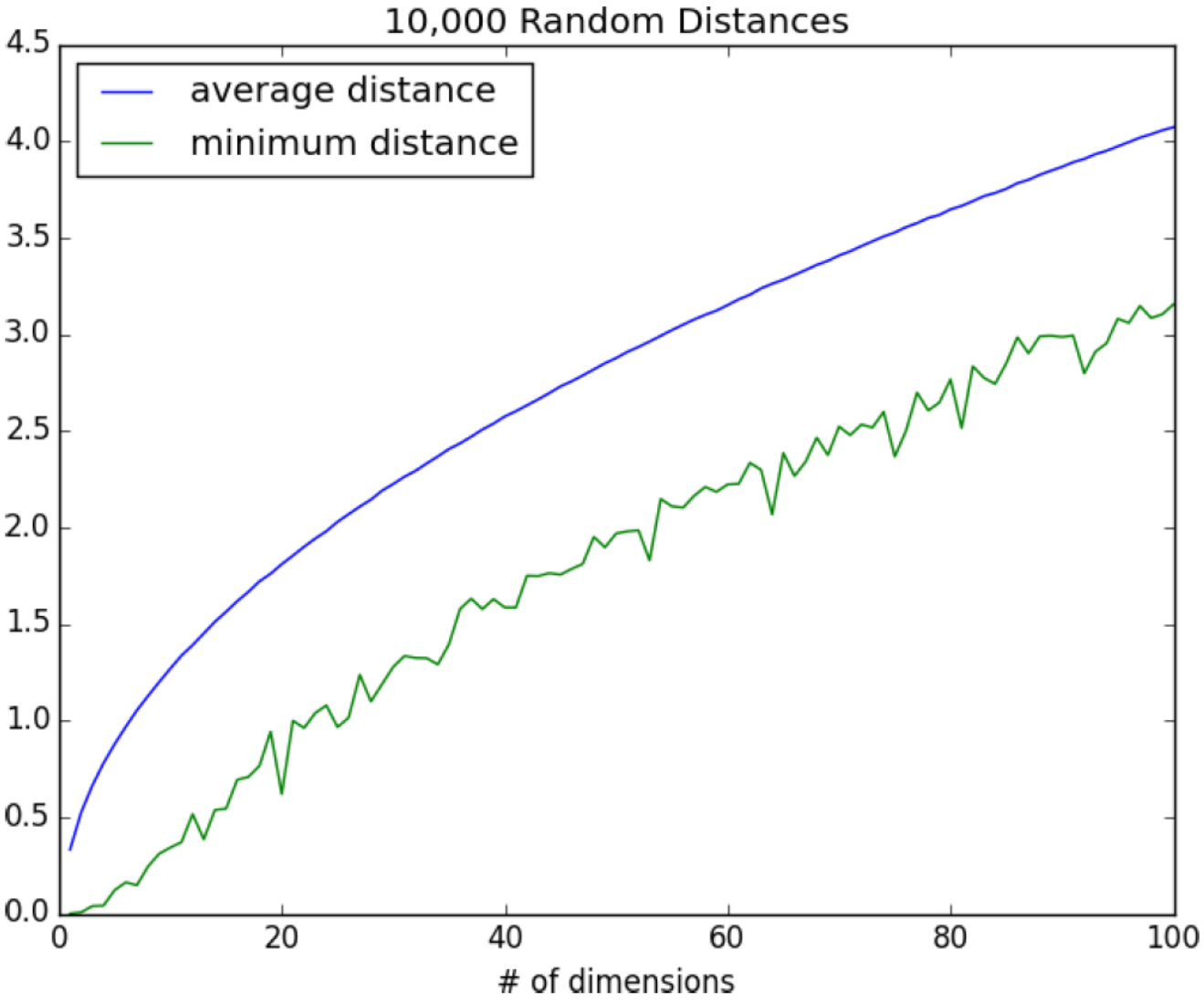
- One way to see this is by rand**omly generating pairs of points in the d-dimensional** "unit cube" in a variety of dimensions, and calculating the distances between them.
- For every dimension from 1 to 100, we'll compute 10,000 distances and use those to compute the average distance between points and the minimum distance between points in each dimension.

The Curse of Dimensionality

- Generating random points should be second nature by now:

```
def random_point(dim: int) -> Vector:
        return [random.random() for _ in range(dim)]

def random_distances(dim: int, num_pairs: int) -> List[float]:
        return [distance(random_point(dim), random_point(dim)) for _ in
range(num_pairs)]

import tqdm
dimensions = range(1, 101)
avg_distances = []
min_distances = []
random.seed(0)
for dim in tqdm.tqdm(dimensions, desc="Curse of Dimensionality"):
    distances = random_distances(dim, 10000) # 10,000 random pairs
    avg_distances.append(sum(distances) / 10000) # track the average
    min_distances.append(min(distances)) # track the minimum
```

The Curse of Dimensionality



The Curse of Dimensionality

- In low-dimensional datasets, the closest points tend to be much closer than average.
- But two points are close only if they're close in every dimension, and every extra dimension—even if just noise—is another opportunity for each point to be farther away from every other point.
- When you have a lot of dimensions, it's likely that the closest points aren't much closer than average, so two points being close doesn't mean very much.

Naive Bayes

- Naive Bayes is a family of simple probabilistic classifiers based on **Bayes' Theorem**, with a strong (naive) assumption of independence between features.
- Supervised Machine Learning

Bayes' Theorem:

$$P(A|B) = \frac{P(B|A) \cdot P(A)}{P(B)}$$

In the context of classification:

- $P(A|B)$: Posterior probability of class A given features B
- $P(B|A)$: Likelihood of features B given class A
- $P(A)$: Prior probability of class A
- $P(B)$: Evidence (overall probability of features B)

The "naive" assumption is that features are **conditionally independent** given the class.

Naive Bayes

- **Types of Naive Bayes Classifiers**

1. **Gaussian Naive Bayes**: Assumes features follow a normal distribution. Used for continuous data.
2. **Multinomial Naive Bayes**: Best for document classification, especially for discrete features (like word counts).
3. **Bernoulli Naive Bayes**: Assumes binary features (e.g., whether a word exists in a document).

Naive Bayes

Domain	Use Case
Marketing	Customer segmentation, email campaign response prediction
Healthcare	Disease diagnosis based on symptoms
Finance	Credit scoring, fraud detection
E-commerce	Predicting customer churn, product recommendation
Natural Language Processing (NLP)	Spam detection, sentiment analysis, topic categorization

Naive Bayes

Advanages

- Fast and efficient, even on large datasets
- Works well with high-dimensional data (e.g., text)
- Performs well with small datasets and requires less training data
- Easy to interpret and implement

Limitations

- Assumes independence of features, which is rarely true in real data
- Struggles with correlated features (can lead to misleading probabilities)
- Poor performance with continuous variables unless assumptions are met (e.g., Gaussian distribution)

Naive Bayes

```
import math

data = [
    ([1, 1, 1], 'yes'),
    ([1, 1, 0], 'yes'),
    ([0, 1, 1], 'no'),
    ([1, 0, 0], 'no')
]
def train_naive_bayes(data):
    class_counts = {'yes': 0, 'no': 0}
    feature_counts = {'yes': [0]*3, 'no': [0]*3}
    total_docs = len(data)

    for features, label in data:
        class_counts[label] += 1
        for i, val in enumerate(features):
            feature_counts[label][i] += val

    return feature_counts, class_counts, total_docs
```

Naive Bayes

```
def predict(features, feature_counts, class_counts, total_docs):
    results = {}
    for c in class_counts:
        log_prob = math.log(class_counts[c] / total_docs)
        total_in_class = class_counts[c]

        for i, val in enumerate(features):
            p = (feature_counts[c][i] + 1) / (total_in_class + 2)
            if val == 1:
                log_prob += math.log(p)
            else:
                log_prob += math.log(1 - p)

        results[c] = log_prob

    return results, max(results, key=results.get)

#Train and predict
feature_counts, class_counts, total_docs = train_naive_bayes(data)
probs, prediction = predict([1, 1, 0], feature_counts, class_counts,
total_docs)
print("Log probabilities:", probs)
print("Prediction:", prediction)
```

Naive Bayes

```
from sklearn.naive_bayes import BernoulliNB
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, classification_report
# Sample dataset: [Fever, Cough, Fatigue], Label
data = [
    [1, 1, 1],  # yes
    [1, 1, 0],  # yes
    [0, 1, 1],  # no
    [1, 0, 0]   # no
]
labels = ['yes', 'yes', 'no', 'no']
# Split into training and test data (optional, here we'll just train and test on all)
X_train = data
y_train = labels
# Initialize and train the Naive Bayes classifier
model = BernoulliNB()
model.fit(X_train, y_train)
# Predict on the same data (for demonstration)
predictions = model.predict(X_train)
# Output results
print("Predictions:", predictions)
print("Accuracy:", accuracy_score(y_train, predictions))
print("Classification Report:\n", classification_report(y_train, predictions))
```

Simple Linear Regression

- We have used the ***correlation*** function to measure the strength of the linear relationship between two variables.
- For most applications, knowing that such a linear relationship exists isn't enough.
- We'll want to understand the nature of the relationship.
- This is where we'll use simple linear regression.

$$\text{correlation}(x, y) = \frac{1}{n-1} \sum \left(\frac{x_i - \bar{x}}{s_x} \cdot \frac{y_i - \bar{y}}{s_y} \right)$$

The Model

- Recall that we were investigating the relationship between a DataSciencester user's number of friends and the amount of time the user spends on the site each day.
- Let's assume that you've considered that **having more friends causes people to spend more time on the site.**
- The VP of Engagement asks you to **build a model describing this relationship.**

$$y_i = \beta x_i + \alpha + \varepsilon_i$$

Where:

- y_i: Minutes user i spends on the site per day.
- x_i: Number of friends user i has.
- ε_i: Error term.
- α: Intercept — base time spent when number of friends is 0.
- β: Slope — additional time spent for each new friend.

The Model

User	x = Friends	y = Minutes on Site
1	0	23
2	1	24
3	2	26
4	3	28
5	4	30

You can already see there's a clear linear trend.

The Model : Program/Code Snippet

```
from typing import Tuple
from statistics import mean, stdev
from math import sqrt
```

$$\text{correlation}(x, y) = \frac{1}{n-1} \sum \left(\frac{x_i - \bar{x}}{s_x} \cdot \frac{y_i - \bar{y}}{s_y} \right)$$

```
def correlation(x, y):
    n = len(x)
    mean_x, mean_y = mean(x), mean(y)
    std_x, std_y = stdev(x), stdev(y)
     return sum((x_i - mean_x) * (y_i - mean_y) for x_i, y_i in zip(x, y)) / ((n - 1) * std_x * std_y)
```

least_squares_fit function calculates the best-fit line parameters, alpha and beta.

beta (slope): How much y increases for every 1-unit increase in x.

alpha (intercept): The value of y when x = 0.

```
def least_squares_fit(x, y) -> Tuple[float, float]:
    beta = correlation(x, y) * stdev(y) / stdev(x)
    alpha = mean(y) - beta * mean(x)
    return alpha, beta
```

The Model: Program/Code Snippet

```
x = [0, 1, 2, 3, 4]
y = [23, 24, 26, 28, 30]

alpha, beta = least_squares_fit(x, y)
print(f"alpha = {alpha:.2f}, beta = {beta:.2f}")
```

- alpha = 22.60, beta = 1.80

- **What the Model Means**
  - This gives us the model:
  - y=1.80x+22.60
  - A user with **0 friends** is predicted to spend **22 .60minutes/day**.
  - For **each additional friend**, they spend **1.80 more minutes/day**.

The Model: Program/Code Snippet

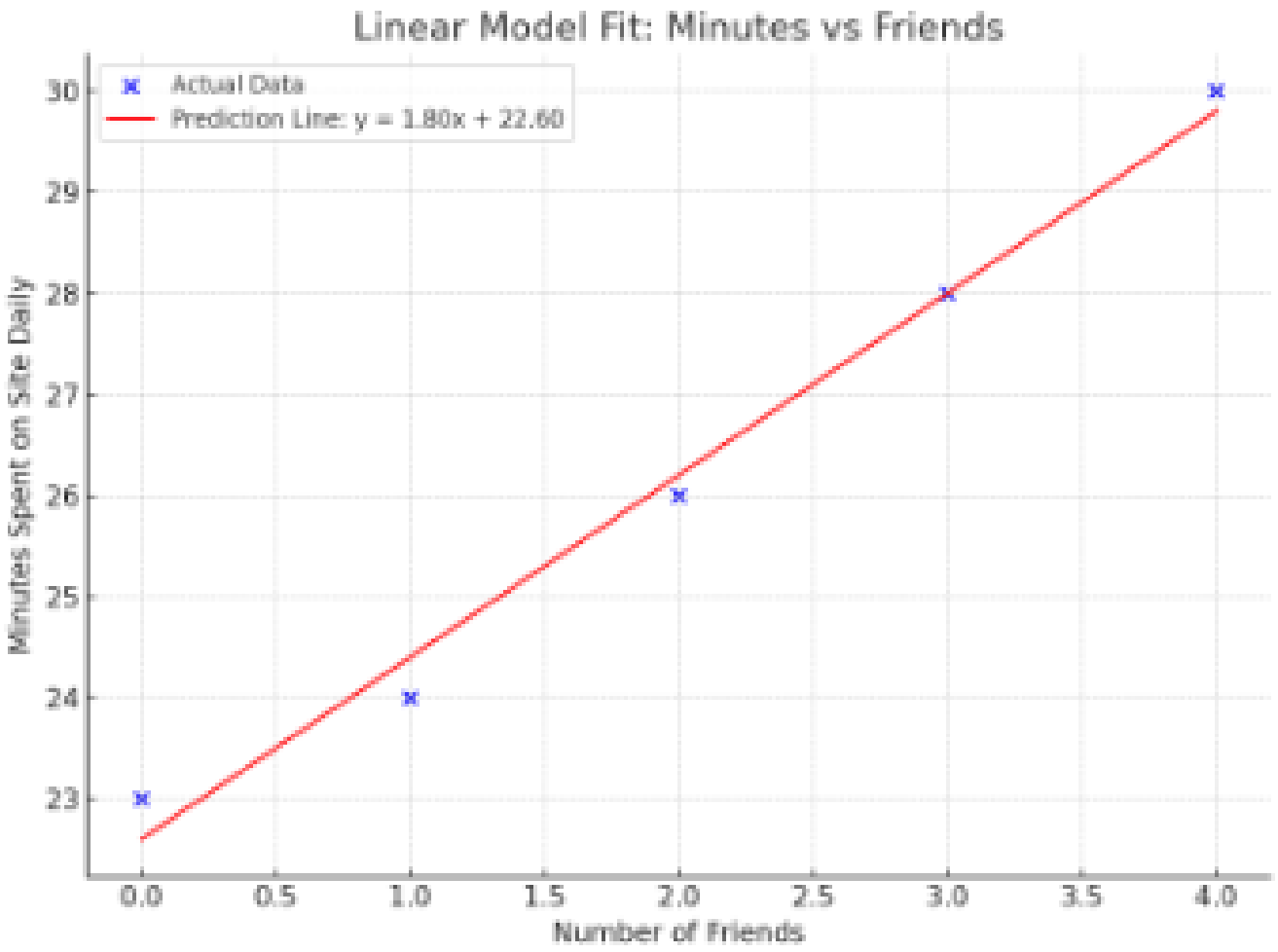
Predicting how many minutes a user with **2 friends** will spend on the site

```
def predict(alpha, beta, x_i):
    return beta * x_i + alpha

prediction = predict(22.60, 1., 2)
print(f"Predicted minutes: {prediction:.2f}")

error = prediction - 26  # = 0.5
```

sum_of_sqerrors function calculates the **total squared error** between your model's predictions and the actual values in your dataset — also known as the **sum of squared errors (SSE)**.

```
def sum_of_sqerrors(alpha, beta, x, y):
    return sum((predict(alpha, beta, x_i) - y_i) ** 2 for x_i, y_i in zip(x, y))

sse = sum_of_sqerrors(alpha, beta, x, y)
print(f"Total Squared Error: {sse:.2f}")
```

The Model


Linear Model Fit: Minutes vs Friends
Actual Data
Prediction Line: y = 1.80x + 22.60
Minutes Spent on Site Daily
Number of Friends
30
29
28
27
26
25
24
23
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0

The Model

SE (Sum of Squared Errors) and **R-squared (R^2)** — two fundamental metrics in linear regression — to understand what each tells us about the quality of a model.

What is SSE (Sum of Squared Errors)?

Definition:

SSE measures the **total squared difference between the actual values** (`y_i`) **and the predicted values** (`ŷ_i`) from the model.

Formula:

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

Where:

- y_i = actual value
- $\hat{y}_i$ = predicted value from the model
- n = number of data points

What it tells us:

- **Lower SSE** → Better model fit (smaller errors).
- **Higher SSE** → Worse model fit (larger prediction errors).

The Model

Definition:

R-squared measures the **proportion of the variance in the dependent variable that is explained by the model**. It's a **relative** measure.

Formula:

$$R^2 = 1 - \frac{\text{SSE}}{\text{TSS}}$$

Where:

- **SSE** = Sum of Squared Errors
- **TSS** = Total Sum of Squares

$$\text{TSS} = \sum_{i=1}^{n} (y_i - \bar{y})^2$$

$\bar{y}$ = mean of all y_i

What it tells us:

- $R^2 = 1$ → Perfect fit: model explains **100%** of the variance.
- $R^2 = 0$ → Model explains **none** of the variance (as good as just predicting the mean).
- Can be **negative** if the model is worse than predicting the mean.

The Model

```
from scratch.statistics import de_mean

def de_mean(data: Vector) -> Vector:
    x_bar = mean(data)
    return [x_i - x_bar for x_i in data]

def total_sum_of_squares(y: Vector) -> float:
    """the total squared variation of y_i's from their mean"""
    return sum(v ** 2 for v in de_mean(y))

def r_squared(alpha: float, beta: float, x: Vector, y: Vector) -> float:
    """
    the fraction of variation in y captured by the model, which equals
    1 - the fraction of variation in y not captured by the model
    """
    return 1.0 - (sum_of_sqerrors(alpha, beta, x, y) /
    total_sum_of_squares(y))
rsq = r_squared(alpha, beta, num_friends_good, daily_minutes_good)
assert 0.328 < rsq < 0.330
```

The Model

Metric	SSE (Sum of Squared Errors)	R-squared
Type	Absolute measure of error	Relative measure of fit
Goal	Minimize	Maximize
Range	0 to $+\infty$	$-\infty$ to 1
Interpretability	Harder (depends on scale of data)	Easier (percentage of variance explained)
Use	Model tuning, optimization	Model evaluation and comparison

- **SSE** tells you **how wrong** your model's predictions are (in total squared units).
- **R-squared** tells you **how well** your model explains the variability in the data (as a percentage).

You typically **minimize SSE** during model training and **report R-squared** to evaluate or compare models.

Maximum Likelihood Estimation (MLE).

- Linear regression model:

  $y_i = \alpha + \beta x_i + \varepsilon_i$, where $\varepsilon_i \sim \mathcal{N}(0, \sigma^2)$

  That is, the errors (differences between actual and predicted values) are normally distributed with mean 0 and constant variance σ^2.

- Goal:

  Estimate α and β based on observed data $(x_1, y_1), \ldots, (x_n, y_n)$.

Maximum Likelihood Estimation (MLE) is used in linear regression to estimate the model parameters (like the intercept α, slope β, and possibly error variance $\sigma 2$) because it provides a principled, probabilistic way to find the values that make the observed data most likely under the assumed model.

Maximum Likelihood Estimation (MLE).

1. Likelihood Function for One Data Point

Given one data point (x_i, y_i), the probability of observing y_i given x_i, α, and β, under the assumption of normally distributed errors, is:

$$L(\alpha, \beta \mid x_i, y_i, \sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(y_i - \alpha - \beta x_i)^2}{2\sigma^2}\right)$$

This expression tells us how *likely* the observed y_i is, given a predicted value $\alpha + \beta x_i$ and a known error variance σ^2.

- $L(\alpha, \beta \mid x_i, y_i, \sigma)$:
  This is the **likelihood function** — the probability (density) of observing the data point (x_i, y_i) given the parameters α, β, and σ.
- α:
  The **intercept** of the linear regression model. It represents the expected value of y when $x = 0$.
- β:
  The **slope** of the regression line. It measures the change in y for a unit change in x.
- x_i:
  The **input (independent variable)** value for the i^{th} observation.
- y_i:
  The **output (dependent variable)** value for the i^{th} observation.
- σ:
  The **standard deviation** of the normally distributed errors. It reflects the variability of the observations around the regression line.

Maximum Likelihood Estimation (MLE).

- $\frac{1}{\sqrt{2\pi\sigma^2}}$:
  This is the **normalization constant** of the normal distribution. It ensures that the total area under the probability density function equals 1.
- $\exp\left(-\frac{(y_i-\alpha-\beta x_i)^2}{2\sigma^2}\right)$:
  This is the **exponential part** of the normal distribution. It decreases as the squared difference between the observed value y_i and the predicted value $\alpha + \beta x_i$ increases.

2. Likelihood of the Whole Dataset

Assuming the data points are independent, the likelihood for the entire dataset is the **product** of the individual likelihoods:

$$L(\alpha, \beta) = \prod_{i=1}^{n} \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(y_i - \alpha - \beta x_i)^2}{2\sigma^2}\right)$$