# RSM338: Machine Learning in Finance

Week 1: Math Bootcamp | January 7–8, 2026

Kevin Mott
Rotman School of Management

Rotman
Commerce

# Welcome to RSM338

Rotman Commerce

# What Is This Course About?

In traditional programming, you write explicit rules for the computer to follow: “if the price drops 10%, sell.” You specify the logic.

**Machine learning** is different. Instead of writing rules, you show the computer examples and let it discover patterns from the statistical properties of the data. The computer learns what predicts what.

**Finance** generates enormous amounts of data: prices, returns, fundamentals, news, filings, transactions. Machine learning gives us tools to extract information from all of it.

Think of ML methods as tools in a toolbox. Just as an experienced contractor knows which tool is right for each job—hammer for nails, wrench for bolts—you’ll learn which ML method is right for each problem:

- **Regression:** Predicting a continuous value (next month’s return)
- **Classification:** Assigning to categories (default vs. no default)
- **Clustering:** Finding natural groupings (asset classes, investor types)
- **Text analysis:** Extracting information from documents (earnings calls, news)

# Why Machine Learning in Finance?

Traditional finance models are elegant but limited:

- CAPM says expected returns depend on one factor (market beta)
  - Fama-French adds size and value
  - But there are hundreds of potential predictors…
- The efficient frontier depends on only risk-return tradeoffs and has few conditions
  - But real investors may have other constraints
  - Do we have good estimates of risk and return?

Machine learning lets us:

- Handle many variables at once without manual selection
- Capture nonlinear relationships
- Let the data tell us what matters

The catch: finance is noisy. Patterns that look predictive often aren't. A major theme of this course is learning to distinguish real signal from noise.

# Course Structure

| Week | Topic |
|---|---|
| 1 | Math Bootcamp (today) |
| 2 | Financial Data |
| 3 | Introduction to Machine Learning |
| 4 | Clustering |
| 5 | Regression |
| 6 | ML & Portfolio Theory |
| 7 | Linear Classification |
| 8 | Nonlinear Classification |
| 9 | Ensemble Methods |
| 10 | Neural Networks |

| | |
|---|---|
| 3 | Introduction to Machine Learning |
| 4 | Clustering |
| 5 | Regression |
| 6 | ML & Portfolio Theory |
| 7 | Linear Classification |
| 8 | Nonlinear Classification |
| 9 | Ensemble Methods |
| 10 | Neural Networks |
| 11 | Text & NLP |
| 12 | Review |

Week 1 builds the mathematical foundation. Everything else builds on it.

**Today's Goal**: Increase your fluency looking at mathematical expressions, recall properties of math that drive intuition. We will NOT need to solve math problems by hand or complete any proofs.

# About Me

**Kevin Mott**

- BS in Mathematics (Northeastern University, Boston, USA)
- PhD in Financial Economics (Carnegie Mellon University, Pittsburgh, USA)
  - Research: Deep learning methods for macro-finance problems
  - I study how neural networks can solve complex economic models
  - In macroeconomics, we can’t run experiments. How to analyze policy? Simulating the macroeconomy with neural networks.
  - In finance, pricing interest rate derivatives has always been hard. But neural networks can solve the pricing equations efficiently.

Email: kevin.mott@rotman.utoronto.ca

Personal Website: kevinpmott.com

Course Website: rsm338.kevinpmott.com

- Here is where we will access lecture notes, which are best opened on a laptop computer in a web browser.
- There will also be PDFs for annotating.

# A Note on Prerequisites

This course assumes you've seen:

- Basic statistics (means, variances, distributions)
- Some calculus (derivatives)
- Introductory finance (returns, portfolios, CAPM)
- A bit of programming

If any of that feels shaky, that's okay. Today's lecture reviews the math and stats foundations. The course website has additional resources.

The goal isn't to filter people out—it's to get everyone to a place where the ML content makes sense.

# Office Hours & Logistics

**TA Office Hours (at the Coding Cafe):** Thursdays 7-8pm

- Location: Management Data Analytics Lab, Rotman South Building, 3rd floor (take the pink stairs)
- There is pizza
- Best for: clarifying assignment questions (with TA), general coding techniques (with Coding Cafe staff)

**Prof. Mott's Office Hours:** Tuesdays 1-2pm (or by appointment)

- Best for: big-picture questions, grading issues, general discussions

This information is on the Quercus homepage (bottom right).

**Coming Up — Week 2 Preassessment:**

- In-class assessment worth 10% of your grade
- Based on Week 1 lecture material (today's content)
- Tests basic math/stats concepts covered in this bootcamp

# Today: Math Bootcamp

Rotman Commerce

# Today’s Goal

This lecture builds the foundation for everything that follows.

**Preliminaries:**

- Notation survival guide (Greek letters, subscripts, summations)
- Key functions (logarithms and exponentials)

**Four main topics:**

1. **Statistics:** Random variables, distributions, expected value, variance
2. **Calculus:** Derivatives and how to find minima
3. **Linear Algebra:** Vectors, matrices, and why they matter
4. **Optimization:** Putting it together—finding the best parameters

If any of this feels new or rusty, that’s fine. The goal is to get everyone on the same page before Week 2.

# Preliminaries: Notation

# (Some) Greek Letters You’ll See

Don’t panic when you see Greek letters. They’re just names for quantities.

| Symbol | Name | Common meaning in this course |
|---|---|---|
| $\mu$ | mu | Mean (expected value) |
| $\sigma$ | sigma | Standard deviation |
| $\rho$ | rho | Correlation |
| $\beta$ | beta | Regression coefficient / market sensitivity |
| $\alpha$ | alpha | Intercept / excess return |
| $\theta$ | theta | Generic parameter |
| $\epsilon$ | epsilon | Error term / noise |
| $\lambda$ | lambda | Regularization parameter |

| Symbol | Name | Common meaning in this course |
|---|---|---|
| $\mu$ | mu | Mean (expected value) |
| $\sigma$ | sigma | Standard deviation |
| $\rho$ | rho | Correlation |
| $\beta$ | beta | Regression coefficient / market sensitivity |
| $\alpha$ | alpha | Intercept / excess return |
| $\theta$ | theta | Generic parameter |
| $\epsilon$ | epsilon | Error term / noise |
| $\lambda$ | lambda | Regularization parameter |
| $\eta$ | eta | Learning rate |

When you see $\mu$, just think “the mean.” When you see $\sigma$, think “the standard deviation.”

# Subscripts: Keeping Track of Things

Subscripts identify *which* observation or *which* variable we're talking about.

**Time subscripts:**

- $r_t$ = return at time $t$
- $P_{t-1}$ = price at time $t-1$ (one period earlier)

**Observation subscripts:**

- $x_i$ = the value for observation $i$
- $y_i$ = the target value for observation $i$

**Variable subscripts:**

- $x_1, x_2, \ldots, x_p$ = features 1 through $p$
- $\beta_j$ = coefficient for feature $j$

**Combining them:**

- $x_{ij}$ = feature $j$ for observation $i$

Rotman Commerce

# Summation and Product Notation

The $\Sigma$ (capital sigma) means “add up”:

$$\sum_{i=1}^{n} x_i = x_1 + x_2 + \cdots + x_n$$

**Read it as:** “Sum of $x_i$ from $i = 1$ to $n$.”

**Examples:**

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i \quad \text{(the sample mean)}$$

$$\text{Var}(X) = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2 \quad \text{(sample variance)}$$

The $\Pi$ (capital pi) means “multiply together”:

$n$

Examples:

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i \quad \text{(the sample mean)}$$

$$\text{Var}(X) = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2 \quad \text{(sample variance)}$$

The $\prod$ (capital pi) means “multiply together”:

$$\prod_{i=1}^{n} x_i = x_1 \times x_2 \times \cdots \times x_n$$

**Why it matters for finance:** Compounding returns multiply. If you earn returns $R_1, R_2, \ldots, R_T$ over $T$ periods, your wealth grows by:

$$\text{Wealth growth} = \prod_{t=1}^{T}(1 + R_t)$$

# Preliminaries: Key Functions

# Logarithms and Exponentials

In finance, we constantly use **log returns**. Understanding logarithms is essential.

**The exponential function:** $e^x$ where $e \approx 2.718$

**The natural logarithm:** $\ln(x)$ is the inverse of $e^x$

They undo each other: $\ln(e^x) = x$ and $e^{\ln(x)} = x$

**Key properties** (these are why logs are so useful):

- Logs turn multiplication into addition: $\ln(a \times b) = \ln(a) + \ln(b)$
- Logs turn division into subtraction: $\ln(a/b) = \ln(a) - \ln(b)$
- Logs turn exponents into multiplication: $\ln(a^b) = b \cdot \ln(a)$

**Log returns:** Instead of the simple return $R_t = \frac{P_t - P_{t-1}}{P_{t-1}}$, we often use:

$$r_t = \ln(1 + R_t) = \ln\left(\frac{P_t}{P_{t-1}}\right) = \ln(P_t) - \ln(P_{t-1})$$

**Why log returns are better:**

Rotman Commerce

$$r_t = \ln(1 + R_t) = \ln\left(\frac{P_t}{P_{t-1}}\right) = \ln(P_t) - \ln(P_{t-1})$$

**Why log returns are better:**

1. **Additivity:** Multi-period log returns just add up: $r_{1 \to T} = r_1 + r_2 + \cdots + r_T$
2. **Symmetry:** A +50% log return followed by -50% gets you back to start
3. **Normality:** Log returns are closer to normally distributed

Common math functions like $e^x$ and $\ln(x)$ are in the numpy library, which by convention we import as np:

```python
import numpy as np

# Computing log returns from prices
prices = np.array([100, 105, 102, 108, 110])
log_returns = np.log(prices[1:]) - np.log(prices[:-1])

print(f"Prices: {prices}")
print(f"Log returns: {log_returns}")
print(f"Sum of log returns: {log_returns.sum():.4f}")
print(f"ln(final/initial): {np.log(prices[-1]/prices[0]):.4f}")  # Same!
```

```
Prices: [100 105 102 108 110]
Log returns: [ 0.04879016 -0.02898754  0.05715841  0.01834914]
Sum of log returns: 0.0953
ln(final/initial): 0.0953
```

# Part I: Statistics

# What Is a Random Variable?

A **random variable** is a quantity whose value is determined by chance.

**Examples:**

- Tomorrow's S&P 500 return
- The outcome of rolling a die
- Whether a borrower defaults on a loan

We don't know the exact value in advance. So how can we make informed guesses about future stock returns, or assess credit risk, or predict anything at all?

**Notation:** We typically use capital letters like $X, Y, R$ for random variables. When we write $X = 3$, we mean "the random variable $X$ takes the value 3."

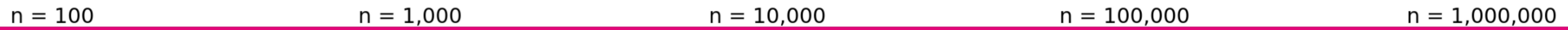
# From Data to Distributions

We can't predict the exact value of a random variable. But we often have **historical data**—past realizations of the same random process.

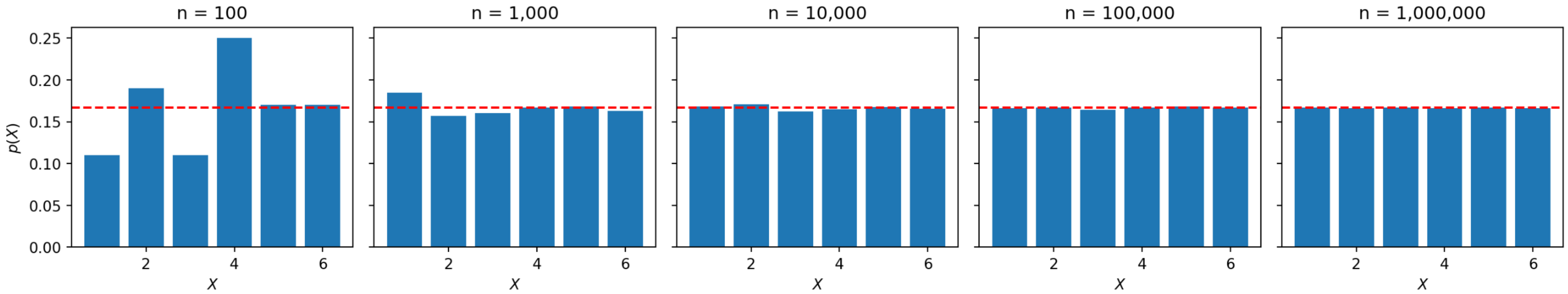
By looking at many past realizations, we can see patterns: some outcomes happen frequently, others are rare. This pattern of "how likely is each outcome?" is called a **distribution**.

Let $X$ be the random variable representing which side of the die is revealed.

```python
# Matplotlib gives us plotting capabilities
import matplotlib.pyplot as plt

np.random.seed(42)
sample_sizes = [100, 1000, 10000, 100000, 1000000]

fig, axes = plt.subplots(1, 5, figsize=(15, 3), sharey=True)
for ax, n in zip(axes, sample_sizes):
    rolls = np.random.randint(1, 7, size=n)
    values, counts = np.unique(rolls, return_counts=True)
    ax.bar(values, counts / n)
    ax.axhline(1/6, color='red', linestyle='--')  # True probability
    ax.set_title(f'n = {n:,}')
    ax.set_xlabel('$X$')
axes[0].set_ylabel('$p(X)$')
plt.tight_layout()
plt.show()
```



```
plt.show()
```



With only 100 rolls, the pattern is noisy. With a million rolls, it's nearly perfect—each value of $X$ appears almost exactly 1/6 of the time (red dashed line). **More data gives us a clearer picture of the true distribution.**

This is a **uniform distribution**: each outcome equally likely.

When we write $X \sim \text{Distribution}$, we're saying: "the random variable $X$ follows this pattern."

Once we know (or estimate) a distribution, we can compute useful summary quantities:

- The **expected value** $\mathbb{E}[X]$ — the average outcome if we repeated the process many times
- The **variance** $\text{Var}(X)$ — how spread out the outcomes are around the mean
- The **probability** of specific events — how likely is a 10% loss? A default?

These are the building blocks of statistical estimation and prediction.

# The Normal Distribution

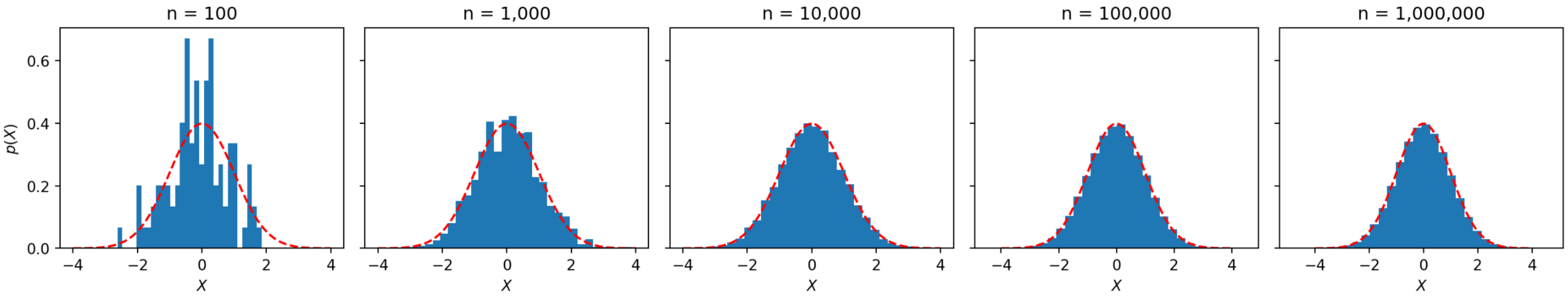
The **normal distribution** (or Gaussian) is the “bell curve.” Most values cluster near the center, with extreme values increasingly rare.

We write: $X \sim \square(\mu, \sigma^2)$

- $\mu$ (mu) = the center (mean)
- $\sigma$ (sigma) = how spread out it is (standard deviation)
- $\sigma^2$ = variance

```python
from scipy.stats import norm

np.random.seed(42)
sample_sizes = [100, 1000, 10000, 100000, 1000000]
x_grid = np.linspace(-4, 4, 100)

fig, axes = plt.subplots(1, 5, figsize=(15, 3), sharey=True)
for ax, n in zip(axes, sample_sizes):
    samples = np.random.normal(loc=0, scale=1, size=n)
    ax.hist(samples, bins=30, density=True)
    ax.plot(x_grid, norm.pdf(x_grid), color='red', linestyle='--')  # True density
    ax.set_title(f'n = {n:,}')
    ax.set_xlabel('$X$')
axes[0].set_ylabel('$p(X)$')
plt.tight_layout()
```

```python
from scipy.stats import norm

np.random.seed(42)
sample_sizes = [100, 1000, 10000, 100000, 1000000]
x_grid = np.linspace(-4, 4, 100)

fig, axes = plt.subplots(1, 5, figsize=(15, 3), sharey=True)
for ax, n in zip(axes, sample_sizes):
    samples = np.random.normal(loc=0, scale=1, size=n)
    ax.hist(samples, bins=30, density=True)
    ax.plot(x_grid, norm.pdf(x_grid), color='red', linestyle='--')  # True density
    ax.set_title(f'n = {n:,}')
    ax.set_xlabel('$X$')
axes[0].set_ylabel('$p(X)$')
plt.tight_layout()
plt.show()
```



Same pattern: with more data, the histogram converges to the true bell curve (red dashed line).

# The Bernoulli Distribution

The **Bernoulli distribution** models yes/no outcomes: something happens (1) or doesn't (0).

We write: $X \sim \text{Bernoulli}(p)$

- $p$ = probability of success (getting a 1)
- $1 - p$ = probability of failure (getting a 0)

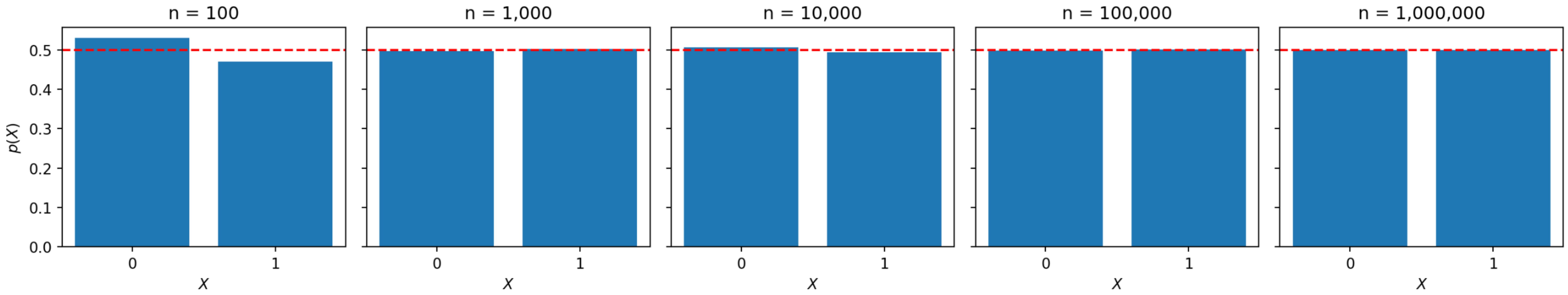
**Finance examples:** Does a borrower default? Does a stock beat the market? Is a transaction fraudulent?

Let $X = 1$ if a coin flip is heads, $X = 0$ if tails, with $p = 0.5$. Same pattern as before: more flips = better estimate of the true probability.

```
np.random.seed(42)
p = 0.5  # Fair coin
sample_sizes = [100, 1000, 10000, 100000, 1000000]

fig, axes = plt.subplots(1, 5, figsize=(15, 3), sharey=True)
for ax, n in zip(axes, sample_sizes):
    flips = np.random.binomial(1, p, size=n)
    values, counts = np.unique(flips, return_counts=True)
    ax.bar(values, counts / n)
    ax.axhline(p, color='red', linestyle='--')  # True probability
    ax.set_title(f'n = {n:,}')
    ax.set_xlabel('$X$')
```

Let $X = 1$ if a coin flip is heads, $X = 0$ if tails, with $p = 0.5$. Same pattern as before: more flips = better estimate of the true probability.

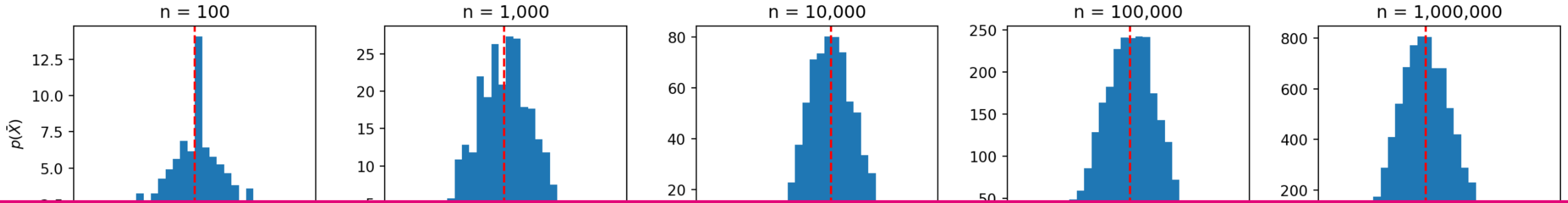
```python
np.random.seed(42)
p = 0.5  # Fair coin
sample_sizes = [100, 1000, 10000, 100000, 1000000]

fig, axes = plt.subplots(1, 5, figsize=(15, 3), sharey=True)
for ax, n in zip(axes, sample_sizes):
    flips = np.random.binomial(1, p, size=n)
    values, counts = np.unique(flips, return_counts=True)
    ax.bar(values, counts / n)
    ax.axhline(p, color='red', linestyle='--')  # True probability
    ax.set_title(f'n = {n:,}')
    ax.set_xlabel('$X$')
    ax.set_xticks([0, 1])
axes[0].set_ylabel('$p(X)$')
plt.tight_layout()
plt.show()
```

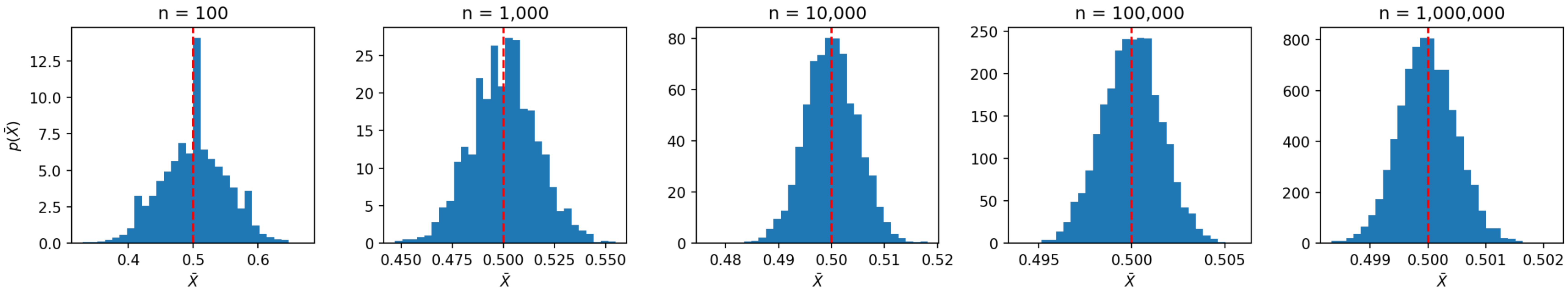


# Why the Normal Distribution Shows Up Everywhere

Here's something remarkable. Suppose we run many experiments, each with $n$ coin flips, and compute the sample mean $\bar{X} = \frac{1}{n} \sum_{i=1}^{n} X_i$ in each experiment. What does the distribution of $\bar{X}$ look like?

```python
np.random.seed(42)
p = 0.5
n_experiments = 5000
sample_sizes = [100, 1000, 10000, 100000, 1000000]

fig, axes = plt.subplots(1, 5, figsize=(15, 3))
for ax, n in zip(axes, sample_sizes):
    # Run 5000 experiments, each with n flips
    sample_means = np.random.binomial(n, p, size=n_experiments) / n
    ax.hist(sample_means, bins=30, density=True)
    ax.axvline(p, color='red', linestyle='--')  # True mean
    ax.set_title(f'n = {n:,}')
    ax.set_xlabel('$\\bar{X}$')
axes[0].set_ylabel('$p(\\bar{X})$')
plt.tight_layout()
plt.show()
```





Two things happen as $n$ increases:

1. **The estimates get better** — the distribution of $\bar{X}$ concentrates around the true mean. This is the **law of large numbers**.
2. **The distribution becomes normal** — no matter what the underlying distribution looks like (here: just 0s and 1s), the distribution of sample means becomes a bell curve. This is the **central limit theorem**.

In fact, the CLT tells us exactly how the sample mean is distributed:

$$\bar{X} \sim \square\left(\mu, \frac{\sigma^2}{n}\right)$$

The spread of our estimates shrinks like $\sigma/\sqrt{n}$. To cut your estimation error in half, you need four times as much data

# Discrete vs. Continuous Distributions

We’ve seen two types of distributions:

**Discrete distributions** — $X$ takes on a finite (or countable) set of values.

- Examples: die rolls (1, 2, 3, 4, 5, 6), loan defaults (0 or 1), number of trades
- We can list each possible value and its probability $p(x)$
- The expected value is a probability-weighted sum:

$$\mathbb{E}[X] = \sum_x x \cdot p(x)$$

**Continuous distributions** — $X$ can take any value in a range.

- Examples: stock returns, interest rates, time until default
- We describe probabilities with a density function $p(x)$

ⓘ **Advanced: Continuous expected value**

For continuous distributions, the expected value is a probability-weighted integral:

**Discrete distributions** — $X$ takes on a finite (or countable) set of values.

- Examples: die rolls (1, 2, 3, 4, 5, 6), loan defaults (0 or 1), number of trades
- We can list each possible value and its probability $p(x)$
- The expected value is a probability-weighted sum:

$$\mathbb{E}[X] = \sum_x x \cdot p(x)$$

**Continuous distributions** — $X$ can take any value in a range.

- Examples: stock returns, interest rates, time until default
- We describe probabilities with a density function $p(x)$

**Advanced: Continuous expected value**

For continuous distributions, the expected value is a probability-weighted integral:

$$\mathbb{E}[X] = \int x \cdot p(x)\, dx$$

An integral is just a continuous sum—same mechanics, different notation.

# Expected Value (Mean)

The **expected value** of a random variable is its mean—the probability-weighted average of all possible outcomes.

**Notation:** $\mathbb{E}[X]$ or $E[X]$ or $\mu$

**Discrete case:** If $X$ can take values $x_1, x_2, \ldots, x_k$ with probabilities $p(x_1), p(x_2), \ldots, p(x_k)$:

$$\mathbb{E}[X] = \sum_{i=1}^{k} x_i \cdot p(x_i) = x_1 \cdot p(x_1) + x_2 \cdot p(x_2) + \cdots + x_k \cdot p(x_k)$$

Each outcome is weighted by how likely it is. Outcomes that happen often contribute more to the average.

**Expected value of functions:** We can also take the expected value of any function $g(X)$. Same idea—probability-weighted sum of the function values:

$$\mathbb{E}[g(X)] = \sum_{i=1}^{k} g(x_i) \cdot p(x_i)$$

This is how we'll define variance: $\mathrm{Var}(X) = \mathbb{E}[(X - \mu)^2]$, the expected squared deviation from the mean.

ⓘ Advanced: Continuous case

Rotman Commerce

$$\mathbb{E}[g(X)] = \sum_{i=1} g(x_i) \cdot p(x_i)$$

This is how we'll define variance: $\text{Var}(X) = \mathbb{E}[(X-\mu)^2]$, the expected squared deviation from the mean.

**Advanced: Continuous case**

If $X$ can take any value in a range, the sum becomes an integral:

$$\mathbb{E}[X] = \int_{-\infty}^{\infty} x \cdot p(x)\, dx$$

Same idea: weight each value $x$ by its density $p(x)$, then "add up" over all values.

**From samples:** When we have data $x_1, x_2, \ldots, x_n$, we estimate the expected value with the sample mean:

$$\bar{x} = \frac{1}{n} \sum_{i=1}^{n} x_i$$

This is equivalent to weighting each observation equally (probability $1/n$ each).

**Why outliers distort the mean:** In the formula $\sum x \cdot p(x)$, even a rare event (low $p(x)$) can contribute heavily if $x$ is extreme. This is why distributions with "fat tails" (rare but extreme events) can have very different means than you'd guess from typical outcomes.

# Variance and Standard Deviation

**Variance** measures how spread out values are around the mean. It's the expected squared deviation from $\mu$:

$$\mathrm{Var}(X) = \mathbb{E}[(X-\mu)^2]$$

**Notation:** $\mathrm{Var}(X)$ or $\sigma^2$

**Discrete case:** The variance is a probability-weighted sum of squared deviations:

$$\mathrm{Var}(X) = \sum_{i=1}^{k} (x_i - \mu)^2 \cdot p(x_i)$$

Outcomes far from the mean (large $(x_i - \mu)^2$) contribute more to variance. This is why distributions with extreme outcomes have high variance.

ⓘ **Advanced: Continuous case**

$$\mathrm{Var}(X) = \int_{-\infty}^{\infty} (x-\mu)^2 \cdot p(x)\,dx$$

$$\mathrm{Var}(X) = \sum_{i=1}^{k} (x_i - \mu)^2 \cdot p(x_i)$$

Outcomes far from the mean (large $(x_i - \mu)^2$) contribute more to variance. This is why distributions with extreme outcomes have high variance.

**Advanced: Continuous case**

$$\mathrm{Var}(X) = \int_{-\infty}^{\infty} (x - \mu)^2 \cdot p(x)\, dx$$

**Standard deviation** is the square root of variance—it's in the same units as $X$:

$$\sigma = \sqrt{\mathrm{Var}(X)}$$

**From samples:** We estimate variance with $s^2 = \frac{1}{n-1} \sum_{i=1}^{n} (x_i - \bar{x})^2$. The $n - 1$ (instead of $n$) corrects for the fact that we estimated $\mu$ with $\bar{x}$.

**In finance:** Standard deviation of returns is called **volatility**. A stock with 20% annualized volatility has much more uncertain returns than one with 10% volatility, even if they have the same expected return.

# Covariance and Correlation

**Covariance** measures whether two variables move together.

$$\text{Cov}(X, Y) = \mathbb{E}[(X - \mu_X)(Y - \mu_Y)]$$

- Positive covariance: when $X$ is high, $Y$ tends to be high
- Negative covariance: when $X$ is high, $Y$ tends to be low

**Correlation** scales covariance to [-1, 1]:

$$\rho_{XY} = \frac{\text{Cov}(X, Y)}{\sigma_X \cdot \sigma_Y}$$

| $\rho$ | Meaning |
|---|---|
| +1 | Perfect positive relationship |
| 0 | No linear relationship |
| -1 | Perfect negative relationship |

$$\text{Cov}(X, Y) = \mathbb{E}[(X - \mu_X)(Y - \mu_Y)]$$

- Positive covariance: when $X$ is high, $Y$ tends to be high
- Negative covariance: when $X$ is high, $Y$ tends to be low

**Correlation** scales covariance to [-1, 1]:

$$\rho_{XY} = \frac{\text{Cov}(X, Y)}{\sigma_X \cdot \sigma_Y}$$

| $\rho$ | Meaning |
|---|---|
| +1 | Perfect positive relationship |
| 0 | No linear relationship |
| -1 | Perfect negative relationship |

**In finance (recall RSM332):** Covariance and correlation are the foundation of portfolio theory. Diversification works because assets with low or negative correlation reduce portfolio variance. The efficient frontier, CAPM, and beta all build on these concepts.

# From Correlation to Linear Regression

Correlation tells us whether two variables move together. **Linear regression** goes further: it finds the best-fitting line.

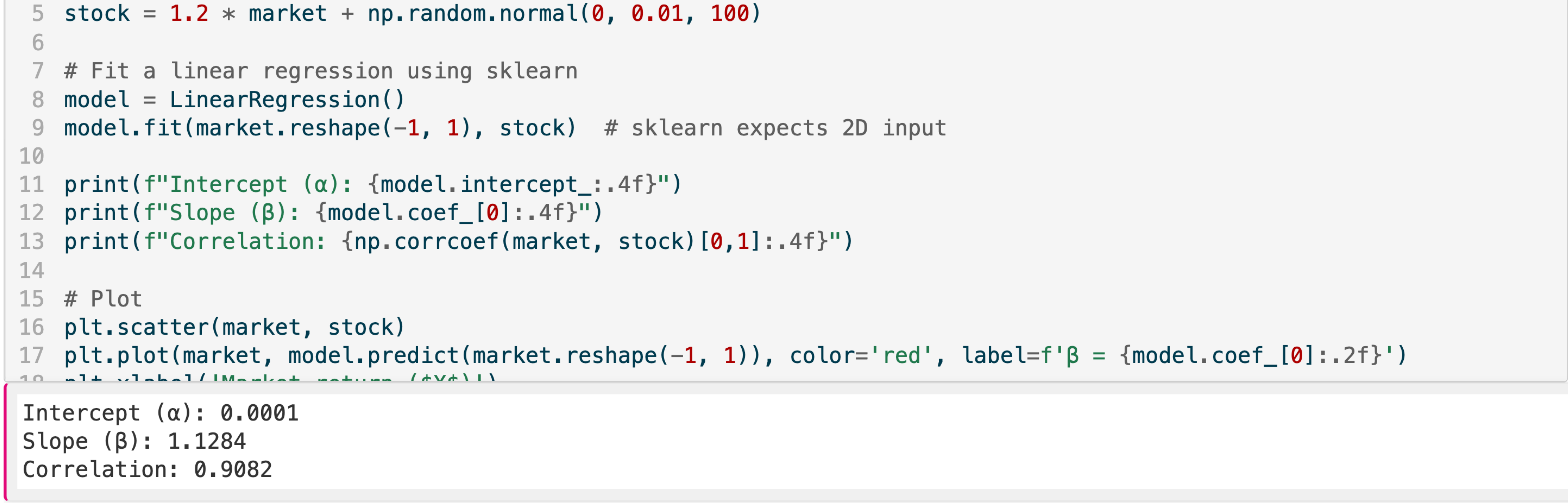
$$Y = \alpha + \beta X + \epsilon$$

- $\alpha$ (alpha) = intercept (where the line crosses the y-axis)
- $\beta$ (beta) = slope (how much $Y$ changes when $X$ increases by 1)
- $\epsilon$ (epsilon) = error term (the part we can't explain)

**The sign of $\beta$ matches the sign of correlation:**

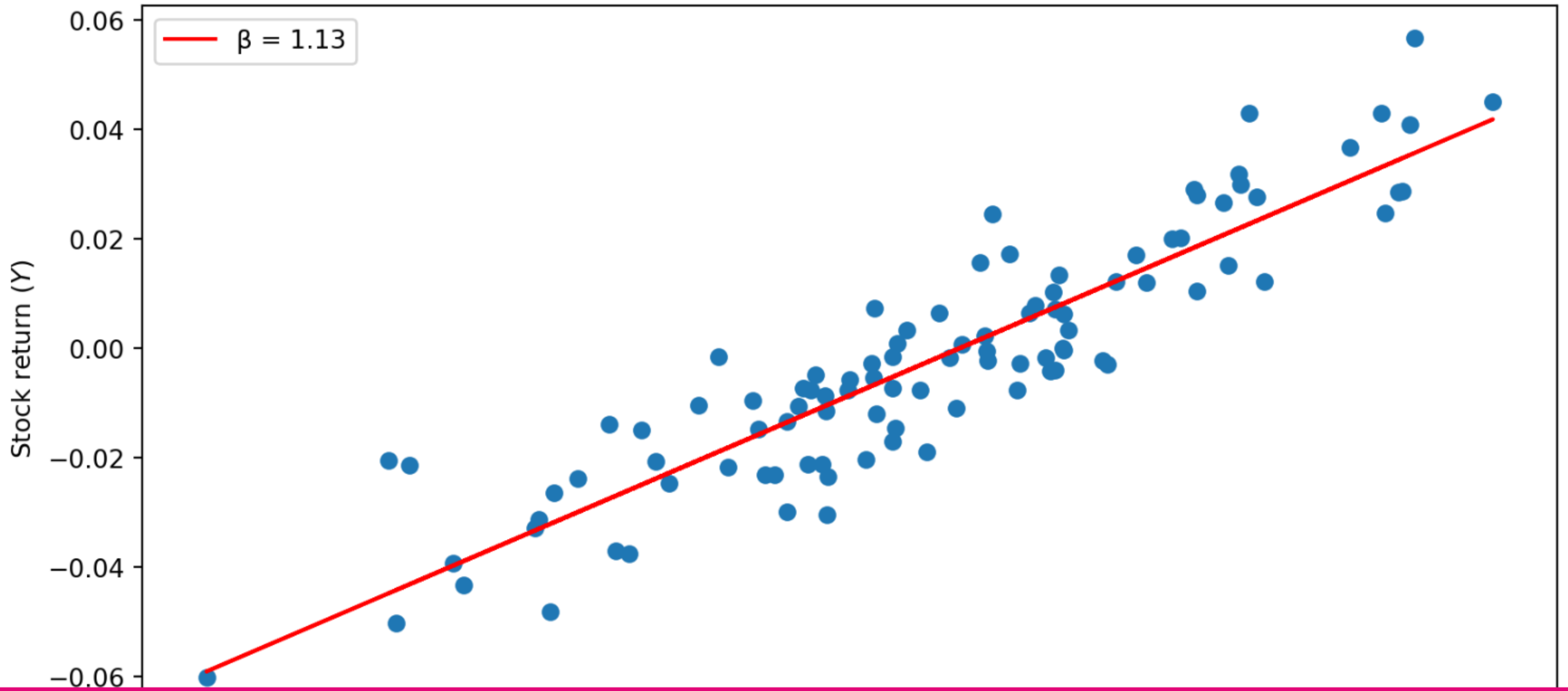
- Positive correlation → positive slope (upward line)
- Negative correlation → negative slope (downward line)
- Zero correlation → zero slope (flat line)

```python
from sklearn.linear_model import LinearRegression

np.random.seed(42)
market = np.random.normal(0, 0.02, 100)
stock = 1.2 * market + np.random.normal(0, 0.01, 100)
```



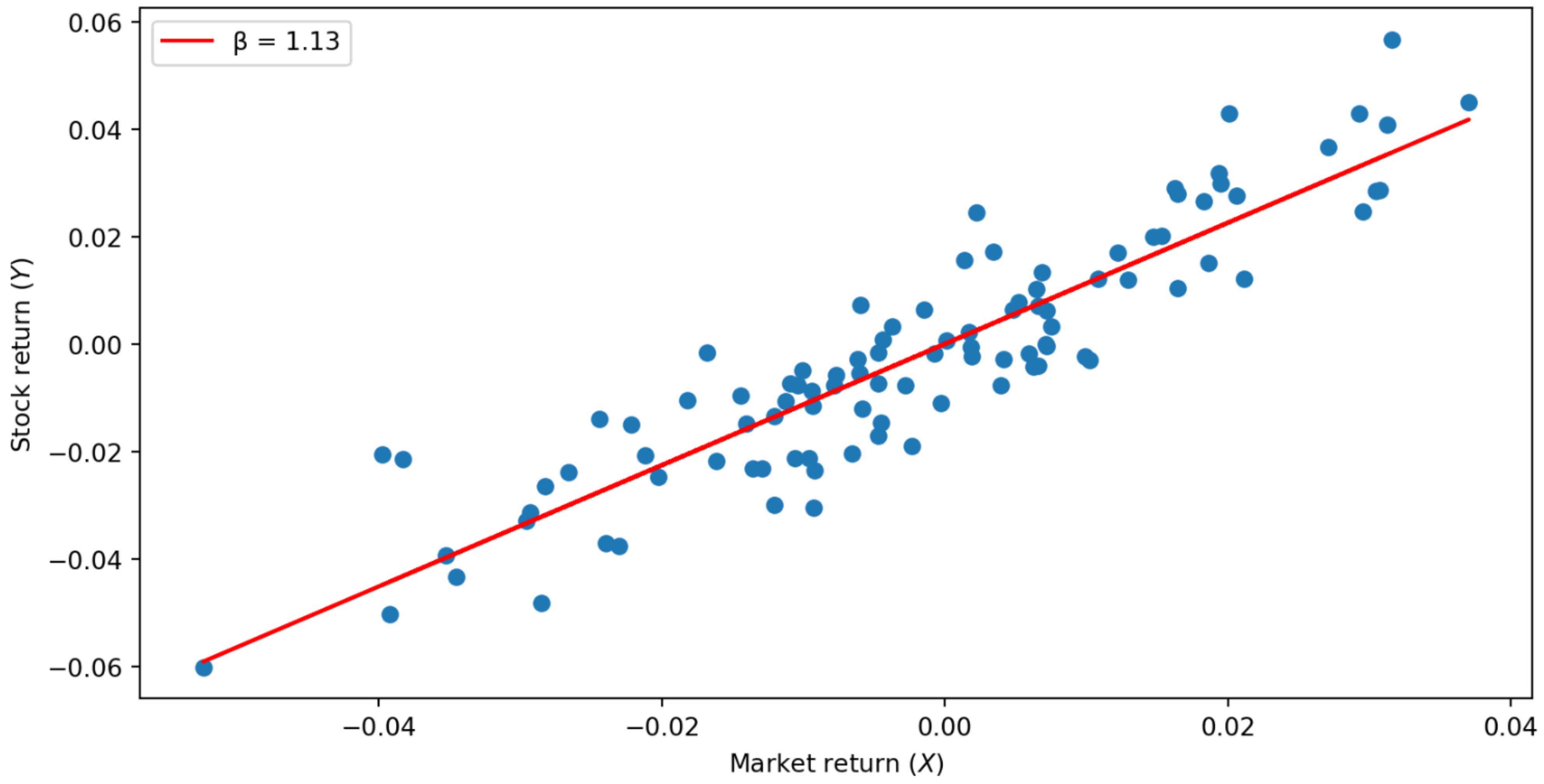
```python
stock = 1.2 * market + np.random.normal(0, 0.01, 100)

# Fit a linear regression using sklearn
model = LinearRegression()
model.fit(market.reshape(-1, 1), stock)  # sklearn expects 2D input

print(f"Intercept (α): {model.intercept_:.4f}")
print(f"Slope (β): {model.coef_[0]:.4f}")
print(f"Correlation: {np.corrcoef(market, stock)[0,1]:.4f}")

# Plot
plt.scatter(market, stock)
plt.plot(market, model.predict(market.reshape(-1, 1)), color='red', label=f'β = {model.coef_[0]:.2f}')
plt.xlabel('Market return ($X$)')
```

```
Intercept (α): 0.0001
Slope (β): 1.1284
Correlation: 0.9082
```



```python
print(f"Intercept (α): {model.intercept_:.4f}")
print(f"Slope (β): {model.coef_[0]:.4f}")
print(f"Correlation: {np.corrcoef(market, stock)[0,1]:.4f}")

# Plot
plt.scatter(market, stock)
plt.plot(market, model.predict(market.reshape(-1, 1)), color='red', label=f'β = {model.coef_[0]:.2f}')
plt.xlabel('Market return ($X$)')
```

```
Intercept (α): 0.0001
Slope (β): 1.1284
Correlation: 0.9082
```



This is the foundation of machine learning: finding relationships in data. sklearn (scikit-learn) is the library we'll use for ML throughout this course.

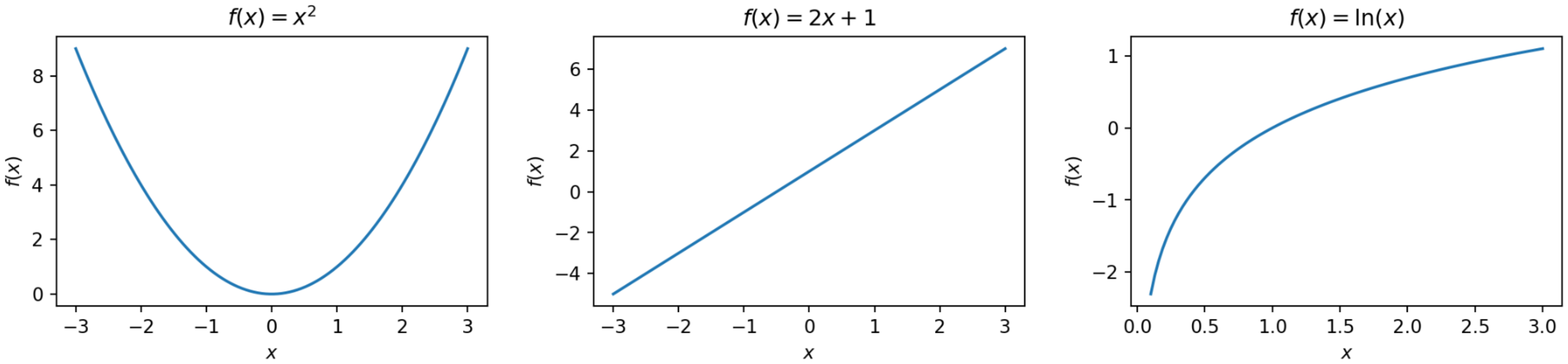
# Part II: Calculus

# Functions: A Quick Reminder

A **function** takes an input and produces an output: $f(x) = x^2$

- Input $x = 3$ → Output $f(3) = 9$
- Input $x = -2$ → Output $f(-2) = 4$

We can visualize functions by plotting input ($x$-axis) against output ($y$-axis):



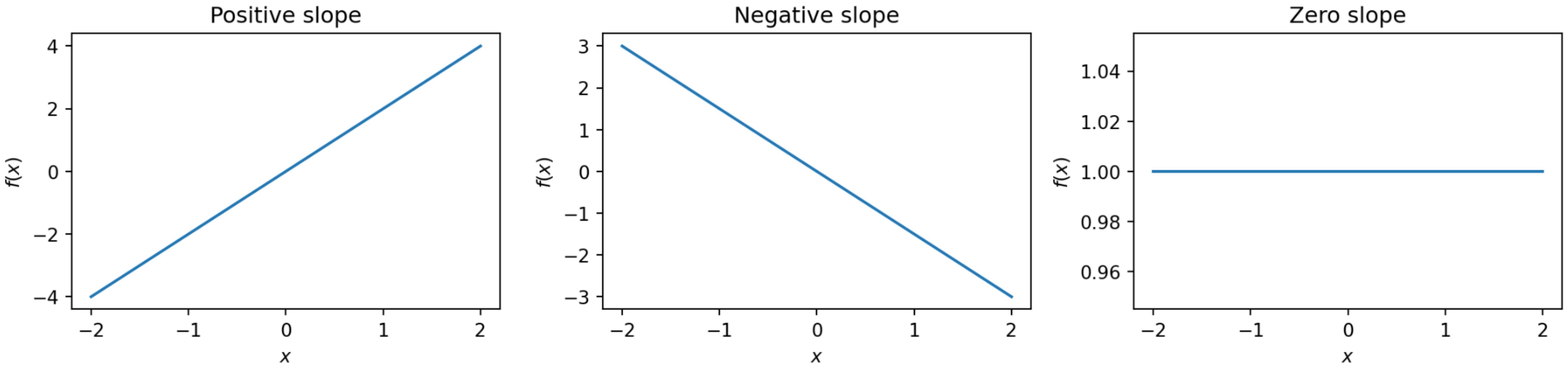
In ML, we'll work with functions that measure error—and we'll want to find the input that makes the error as small as possible.

# Slope: Rise Over Run

For a straight line, the **slope** tells you how steep it is:

$$\text{slope} = \frac{\text{rise}}{\text{run}} = \frac{\Delta y}{\Delta x} = \frac{y_2 - y_1}{x_2 - x_1}$$

- Positive slope: line goes up as you move right
- Negative slope: line goes down as you move right
- Zero slope: flat line



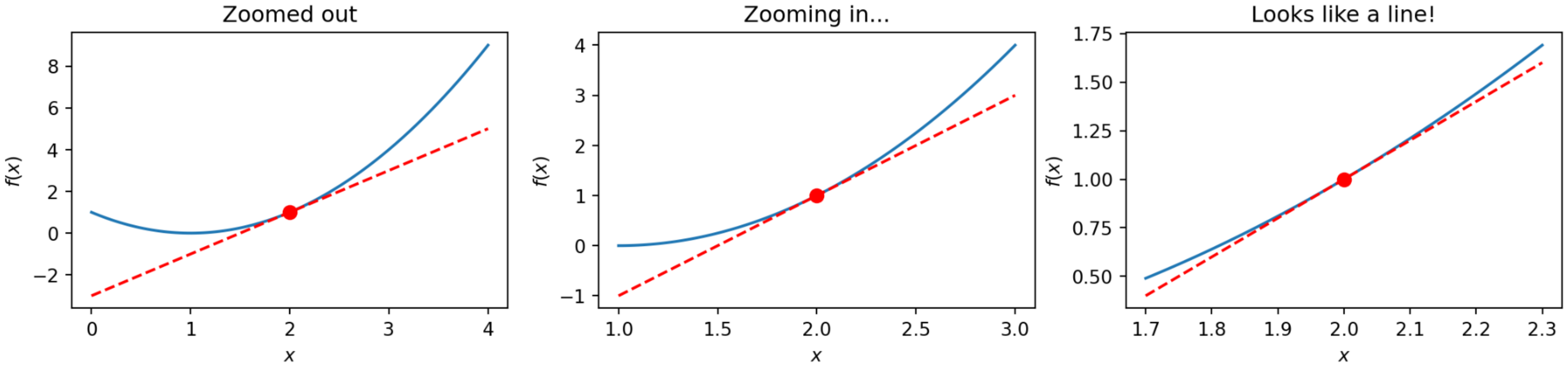
But what about curves? The slope is different at every point…

Rotman Commerce

# Derivatives: Slope at a Point

The **derivative** is the slope of a curve at a specific point—the “instantaneous” rate of change.

Imagine zooming in on a curve until it looks like a straight line. The slope of that line is the derivative.



At $x = 2$, the curve $f(x) = (x - 1)^2$ has slope 2. We write: $f'(2) = 2$.

# Derivative Notation

Several notations mean the same thing—the derivative of $f$ with respect to $x$:

$$f'(x) = \frac{df}{dx} = \frac{d}{dx}f(x)$$

- $f'(x)$ — “f prime of x”
- $\frac{df}{dx}$ — “df dx” (Leibniz notation, emphasizes “change in $f$ per change in $x$”)

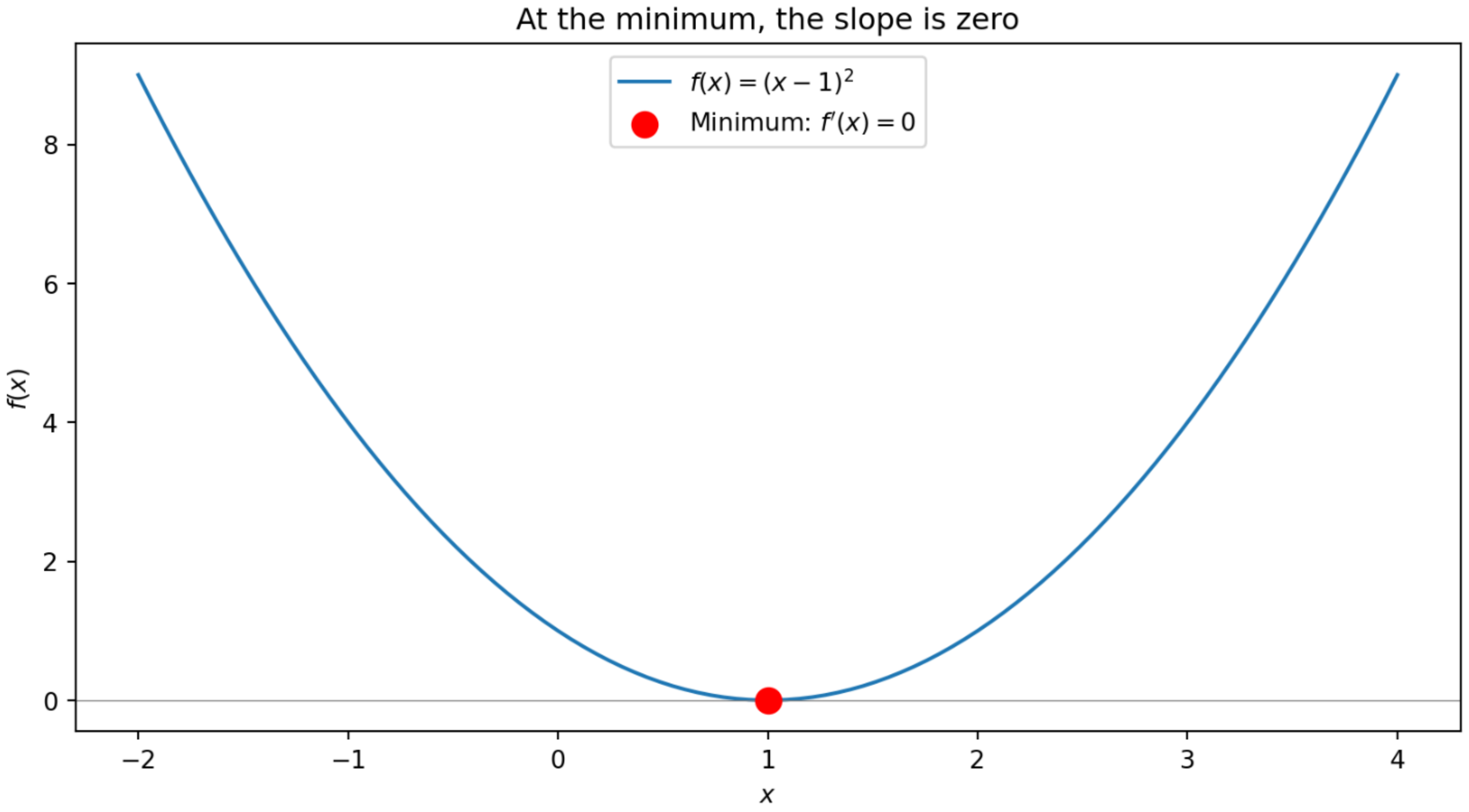
**What the derivative tells you:**

- $f'(x) > 0$: function is increasing at $x$
- $f'(x) < 0$: function is decreasing at $x$
- $f'(x) = 0$: function is flat at $x$

# Why We Care: Finding Extrema

An **extremum** (plural: extrema) is a minimum or maximum of a function.

At an extremum, the function is flat—it's neither increasing nor decreasing. That means the derivative is zero.



**This is the key insight:** To find where a function is minimized (or maximized), find where its derivative equals zero.

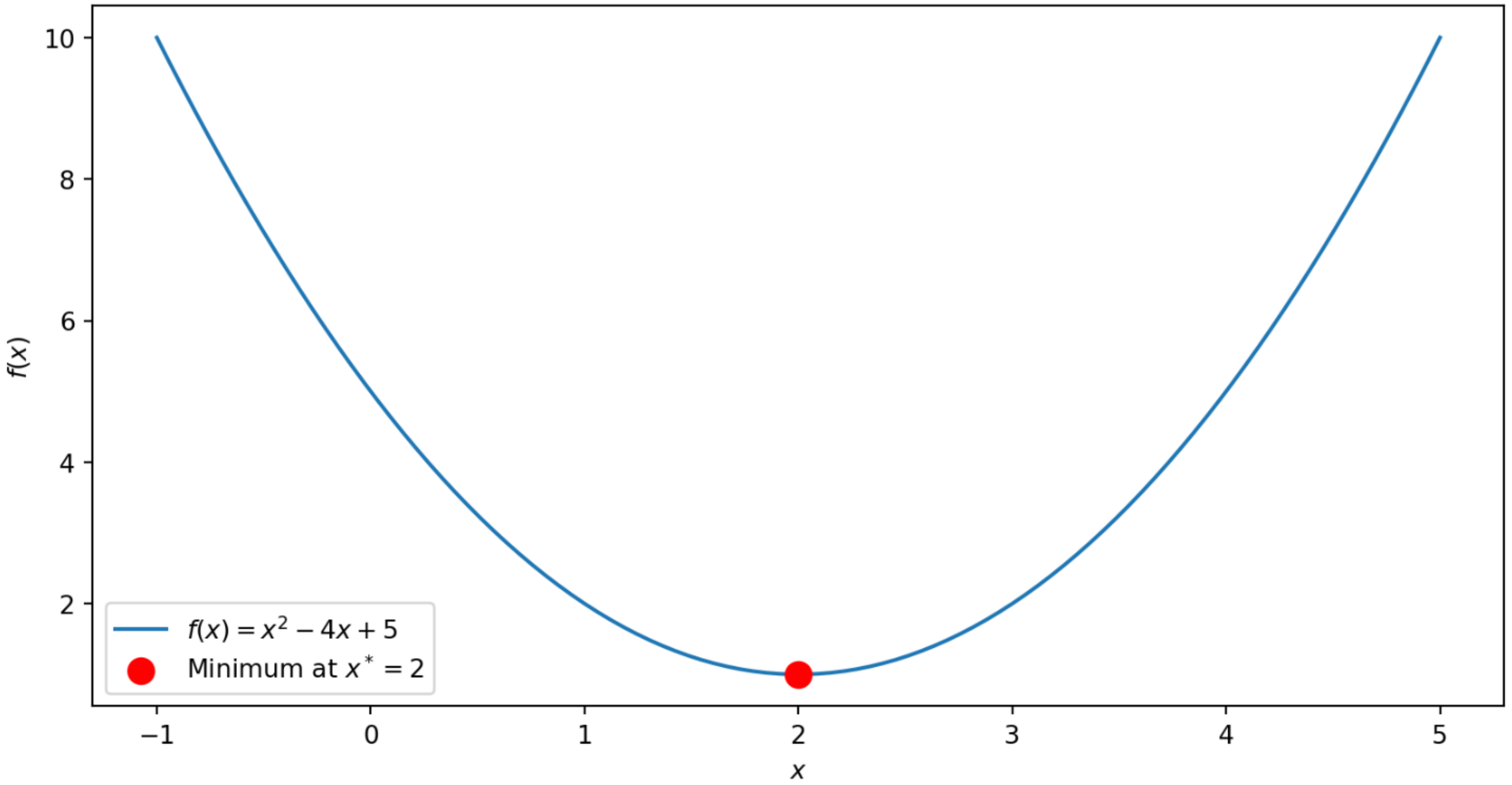
# Finding Minima: The Recipe

**To find the minimum of $f(x)$:**

**1.** Take the derivative $f'(x)$

**2.** Set $f'(x) = 0$ and solve for $x$

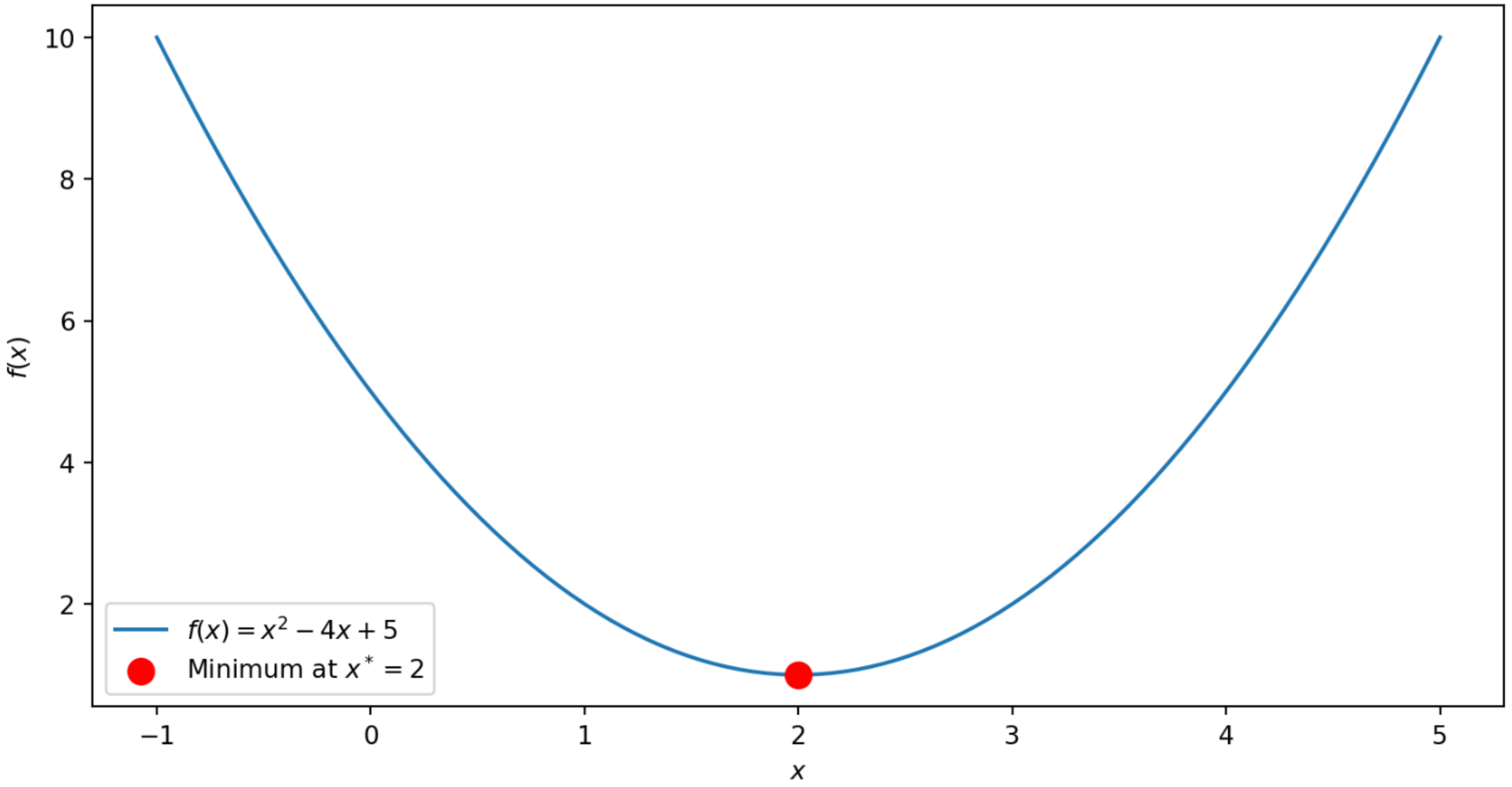
You won't be computing derivatives by hand in this course—computers handle that. But you need to understand the logic: **the minimum is where the slope is zero.**



**To find the minimum of $f(x)$:**

1. Take the derivative $f'(x)$
2. Set $f'(x) = 0$ and solve for $x$

You won't be computing derivatives by hand in this course—computers handle that. But you need to understand the logic: **the minimum is where the slope is zero.**



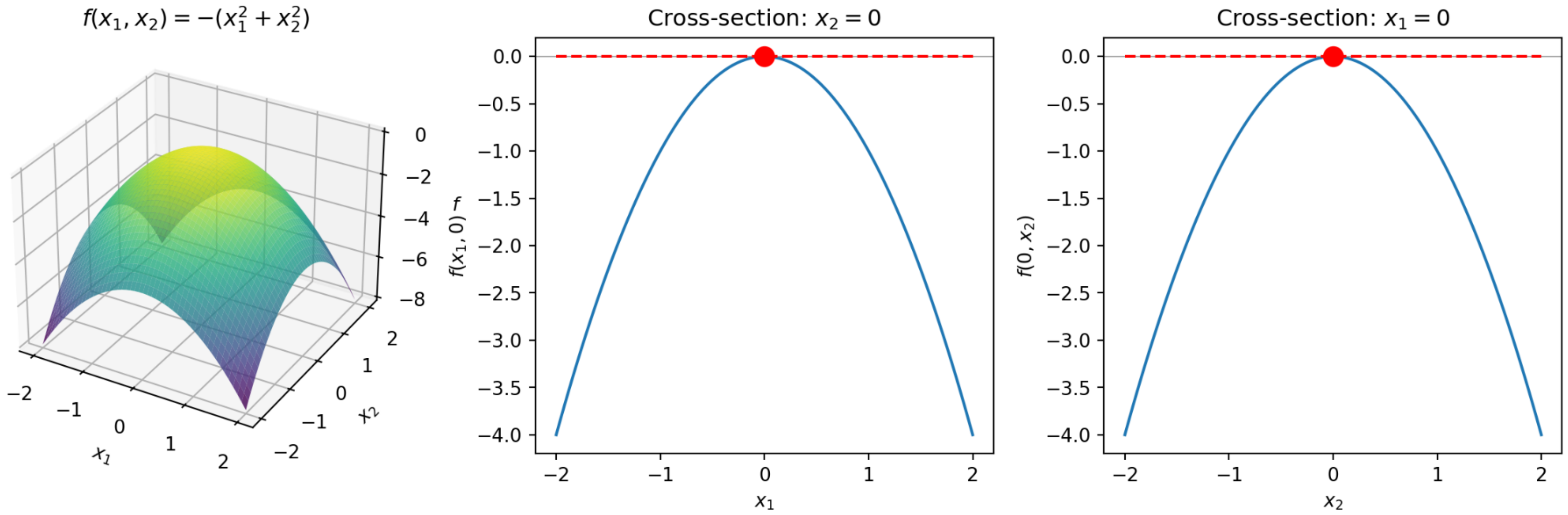
**Why this matters for ML:** In machine learning, we define a **loss function** that measures error. Training a model means finding the parameters that minimize that loss—finding where the slope is zero.

# Functions of Multiple Variables

So far we've looked at functions of one variable: $f(x)$. But what if a function depends on two (or more) variables?

$$f(x_1, x_2) = -(x_1^2 + x_2^2)$$

This function takes two inputs and produces one output. We can visualize it as a surface in 3D:



The maximum is at the origin $(0, 0)$—the top of the "dome." In both cross-sections, the tangent line is flat (slope = 0) at the extremum.

# Partial Derivatives: Slope in One Direction

How do we find the extremum of a function with multiple variables?

We ask: if I move in just the $x_1$ direction (holding $x_2$ fixed), what's the slope? That's the **partial derivative** with respect to $x_1$. We can do the same for $x_2$.

Look back at our cross-sections: each one shows the slope in one direction. At the top of the dome, both cross-sections are flat—the slope is zero in every direction.

**At an extremum, all partial derivatives are zero.** The surface is flat no matter which direction you look. This is what optimization algorithms search for.

# Part III: Linear Algebra

# Why Linear Algebra?

We've seen that **derivatives are slopes**—and slopes are lines. Linear relationships are easy to work with: easy to compute, easy to optimize, easy to interpret.

**Vectors and matrices let us extend this to multiple dimensions:**

| One variable | Multiple variables |
|---|---|
| Derivative $f'(x)$ | Gradient $\nabla f$ (vector of partial derivatives) |
| Slope $\beta$ | Coefficient vector $\boldsymbol{\beta}$ |
| $y = \alpha + \beta x$ | $\mathbf{y} = \mathbf{X}\boldsymbol{\beta} + \boldsymbol{\epsilon}$ |

The structure is the same—we just stack things into vectors and matrices. Instead of writing $n$ separate equations, we write one matrix equation. This is the language of machine learning.

# Vectors

A **vector** is an ordered list of numbers. We write vectors in bold: $\mathbf{x}$

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

We say $\mathbf{x} \in \mathbb{R}^n$ meaning "$\mathbf{x}$ is a vector of $n$ real numbers."

For matrix multiplication, think of a vector as an $(n \times 1)$ matrix—a matrix with $n$ rows and 1 column.

```python
# Vectors in numpy
x = np.array([1, 2, 3, 4, 5])
print(f"x = {x}")
print(f"x has {len(x)} elements")
print(f"x[0] = {x[0]}")  # First element (Python is 0-indexed)
```

```
x = [1 2 3 4 5]
x has 5 elements
x[0] = 1
```

# Matrices

A **matrix** is a 2D array of numbers. We write matrices in bold capitals: $\mathbf{X}$

$$\mathbf{X} = \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1p} \\ x_{21} & x_{22} & \cdots & x_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n1} & x_{n2} & \cdots & x_{np} \end{bmatrix}$$

We say $\mathbf{X} \in \mathbb{R}^{n \times p}$ meaning "$\mathbf{X}$ has $n$ rows and $p$ columns."

- Rows = observations (e.g., different stocks, different days)
- Columns = features (e.g., different variables)

```python
X = np.array([[1, 2, 3],
              [4, 5, 6]])
print(f"X has shape {X.shape}")  # (rows, columns)
print(f"X =\n{X}")
```

```
X has shape (2, 3)
X =
[[1 2 3]
 [4 5 6]]
```

# Transpose

The **transpose** flips rows and columns. We write it as $\mathbf{X}'$ or $\mathbf{X}^T$.

$$\mathbf{X} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \quad \Longrightarrow \quad \mathbf{X}' = \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix}$$

An $(n \times p)$ matrix becomes $(p \times n)$ after transposing.

```python
X = np.array([[1, 2, 3],
              [4, 5, 6]])

print(f"X shape: {X.shape}")
print(f"X' shape: {X.T.shape}")
print(f"X' =\n{X.T}")
```

```
X shape: (2, 3)
X' shape: (3, 2)
X' =
[[1 4]
 [2 5]
 [3 6]]
```

# Matrix Multiplication

To multiply matrices $\mathbf{A}$ and $\mathbf{B}$, the inner dimensions must match:

$$(m \times \underbrace{n) \cdot (n}_{\text{These must be the same!}} \times p) = (m \times p)$$

The result has the same number of rows as $\mathbf{A}$ and the same number of columns as $\mathbf{B}$.

**The dot product** is a special case—two vectors of the same length produce a scalar:

$$\mathbf{x} \cdot \mathbf{y} := \mathbf{x}'\mathbf{y} = \sum_i x_i y_i = x_1 y_1 + x_2 y_2 + \cdots + x_n y_n$$

because $\mathbf{x}'\mathbf{y}$ is $(1 \times n) \cdot (n \times 1) = 1$, a scalar.

```python
x = np.array([1, 2, 3])
y = np.array([4, 5, 6])
# Dot product: 1*4 + 2*5 + 3*6 = 32
print(f"x · y = {np.dot(x, y)}")
```

```
x · y = 32
```

⚠ Warning

⚠ Warning

Matrix multiplication is NOT commutative. With scalars $a, b \in \mathbb{R}$, $a \times b = b \times a$. But with matrices, $\mathbf{AB} \neq \mathbf{BA}$ in general. The order matters!

**The identity matrix** $\mathbf{I}$ has 1s on the diagonal and 0s everywhere else. It's the matrix analogue of multiplying by 1:

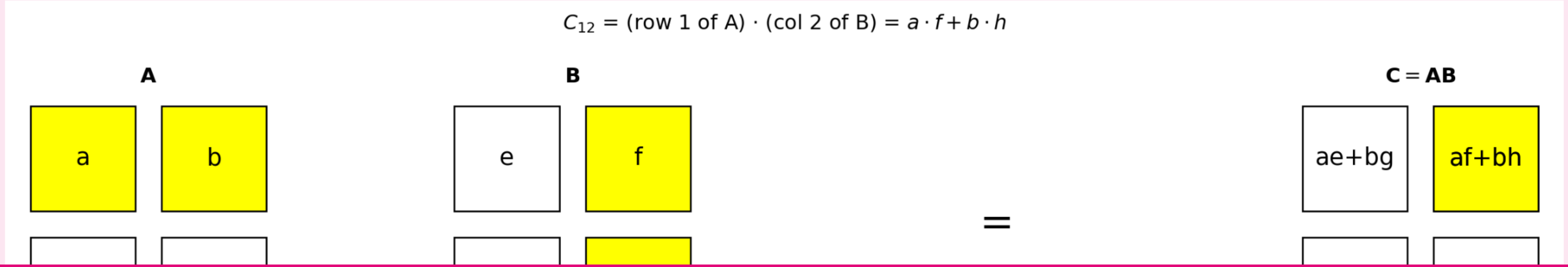
$$\mathbf{AI} = \mathbf{IA} = \mathbf{A}$$

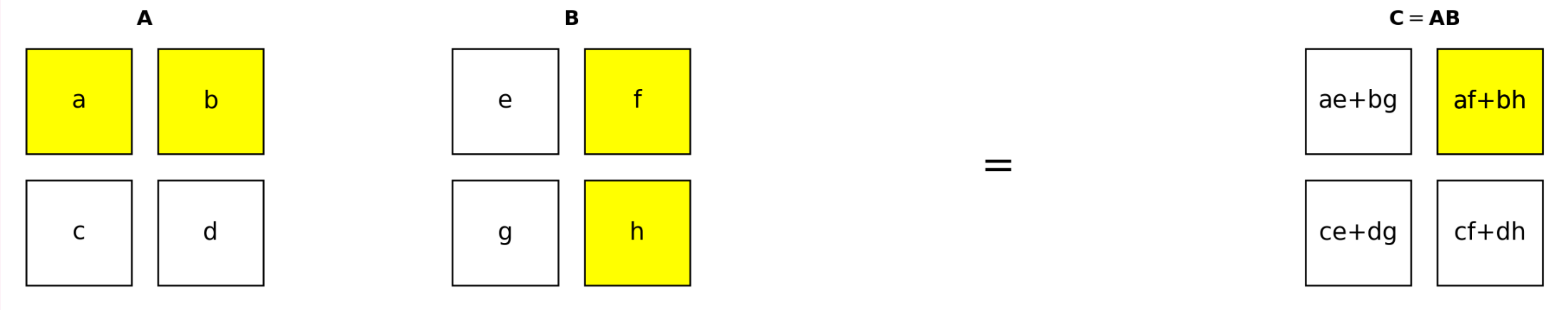
**Matrix inverse:** For square matrices, under the right conditions, we can find an inverse $\mathbf{A}^{-1}$ such that $\mathbf{A}^{-1}\mathbf{A} = \mathbf{AA}^{-1} = \mathbf{I}$. This lets us "undo" multiplication—analogous to division for scalars.

ⓘ **Advanced: How matrix multiplication works**

Each element of the output is a dot product. Specifically, element $(i, j)$ of $\mathbf{C} = \mathbf{AB}$ is the dot product of **row** $i$ **of** $\mathbf{A}$ with **column** $j$ **of** $\mathbf{B}$:

$$C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}$$


$C_{12}$ = (row 1 of A) · (col 2 of B) = $a \cdot f + b \cdot h$
A
a
b
B
e
f
=
C = AB
ae+bg
af+bh



A

a b

c d

B

e f

g h

=

C = AB

ae+bg af+bh

ce+dg cf+dh

In Python, use `@` for matrix multiplication and `.T` for transpose:

```python
A = np.array([[1, 2], [3, 4]])
B = np.array([[5, 6], [7, 8]])
C = A @ B
print(f"A =\n{A}\n")
print(f"B =\n{B}\n")
print(f"A @ B =\n{C}")
print(f"\nC[0,1] = A[0,:] · B[:,1] = 1*6 + 2*8 = {C[0,1]}")
print(f"\nA.T (transpose) =\n{A.T}")
```

```
A =
[[1 2]
 [3 4]]

B =
[[5 6]
 [7 8]]

A @ B =
[[19 22]
 [43 50]]

C[0,1] = A[0,:] · B[:,1] = 1*6 + 2*8 = 22

A.T (transpose) =
[[1 3]
 [2 4]]
```

# Linear Algebra Application: OLS

In linear regression, we want to predict $y$ from multiple variables $x_1, x_2, \ldots, x_p$:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_p x_p + \epsilon$$

But we have $n$ observations, so we need $n$ equations:

$$\begin{aligned}
y_1 &= \beta_0 + \beta_1 x_{11} + \beta_2 x_{12} + \cdots + \beta_p x_{1p} + \epsilon_1 \\
y_2 &= \beta_0 + \beta_1 x_{21} + \beta_2 x_{22} + \cdots + \beta_p x_{2p} + \epsilon_2 \\
&\vdots \\
y_n &= \beta_0 + \beta_1 x_{n1} + \beta_2 x_{n2} + \cdots + \beta_p x_{np} + \epsilon_n
\end{aligned}$$

Stack everything into vectors and matrices:

$$\underbrace{\mathbf{y}}_{n\times 1} = \underbrace{\mathbf{X}}_{n\times p}\underbrace{\boldsymbol{\beta}}_{p\times 1} + \underbrace{\boldsymbol{\epsilon}}_{n\times 1}$$

$$y_n = \beta_0 + \beta_1 x_{n1} + \beta_2 x_{n2} + \cdots + \beta_p x_{np} + \epsilon_n$$

Stack everything into vectors and matrices:

$$\underbrace{\mathbf{y}}_{n\times 1} = \underbrace{\mathbf{X}}_{n\times p}\underbrace{\boldsymbol{\beta}}_{p\times 1} + \underbrace{\boldsymbol{\epsilon}}_{n\times 1}$$

The ordinary least squares (OLS) solution is: $\hat{\boldsymbol{\beta}} = (\mathbf{X}'\mathbf{X})^{-1}\mathbf{X}'\mathbf{y}$

**Advanced: Where does the OLS formula come from?**

Start with $\mathbf{y} = \mathbf{X}\boldsymbol{\beta} + \boldsymbol{\epsilon}$. In expectation, $\boldsymbol{\epsilon}$ averages to zero, so we want to solve $\mathbf{y} = \mathbf{X}\boldsymbol{\beta}$ for $\boldsymbol{\beta}$.

We'd like to "divide by $\mathbf{X}$" but $\mathbf{X}$ is $(n \times p)$—not square, so not invertible!

The trick: premultiply both sides by $\mathbf{X}'$ to make it square:

$$\mathbf{X}'\mathbf{y} = \mathbf{X}'\mathbf{X}\boldsymbol{\beta}$$

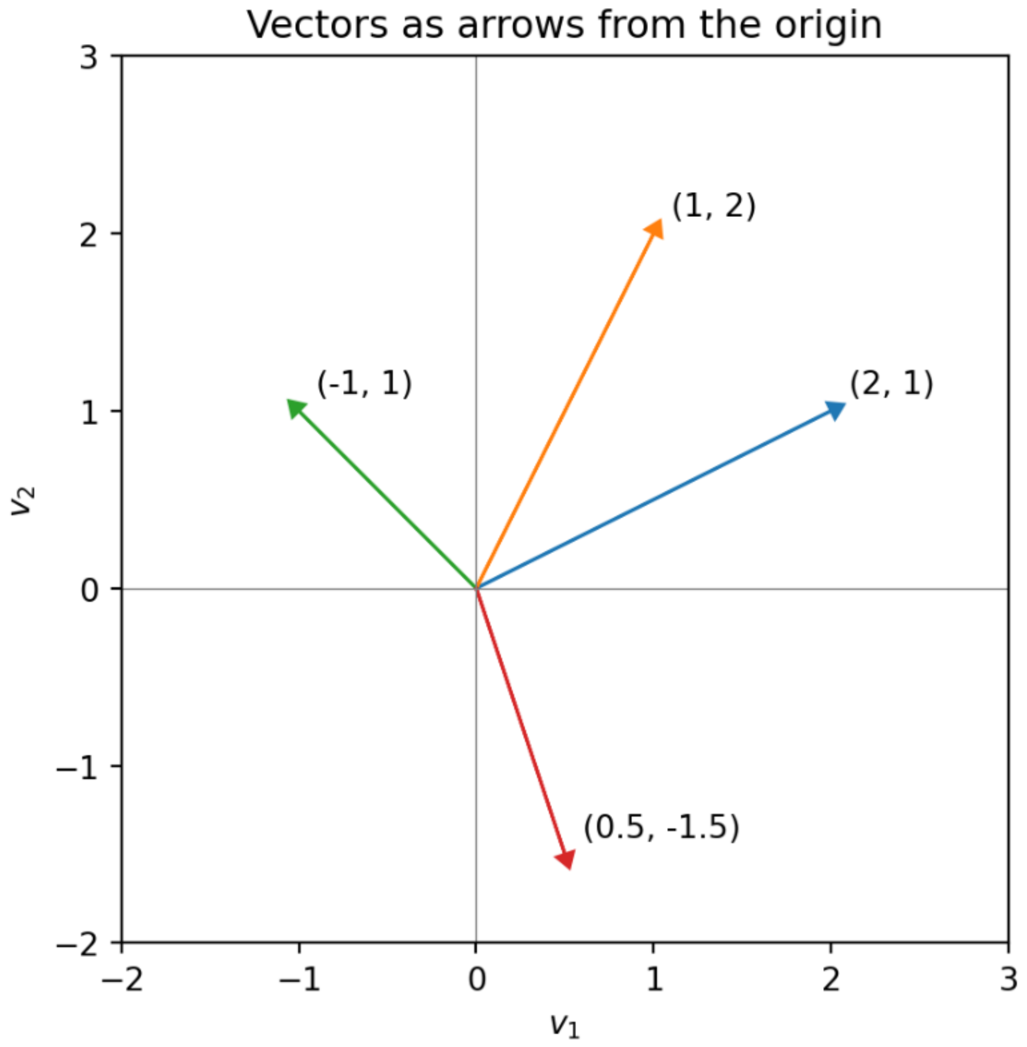
Now $\mathbf{X}'\mathbf{X}$ is $(p \times p)$—square and invertible. Premultiply both sides by $(\mathbf{X}'\mathbf{X})^{-1}$ :

$$\begin{aligned}(\mathbf{X}'\mathbf{X})^{-1}\mathbf{X}'\mathbf{y} &= (\mathbf{X}'\mathbf{X})^{-1}\mathbf{X}'\mathbf{X}\boldsymbol{\beta}\\ (\mathbf{X}'\mathbf{X})^{-1}\mathbf{X}'\mathbf{y} &= \mathbf{I}\boldsymbol{\beta}\\ \boldsymbol{\beta} &= (\mathbf{X}'\mathbf{X})^{-1}\mathbf{X}'\mathbf{y}\end{aligned}$$

# Vectors Have Direction

A vector in $\mathbb{R}^2$ is just two numbers: $\mathbf{v} = \begin{bmatrix} v_1 \\ v_2 \end{bmatrix}$

But we can also think of it as an arrow that **points** somewhere:



The vector $\begin{bmatrix} 2 \\ 1 \end{bmatrix}$ points “2 units right and 1 unit up.”

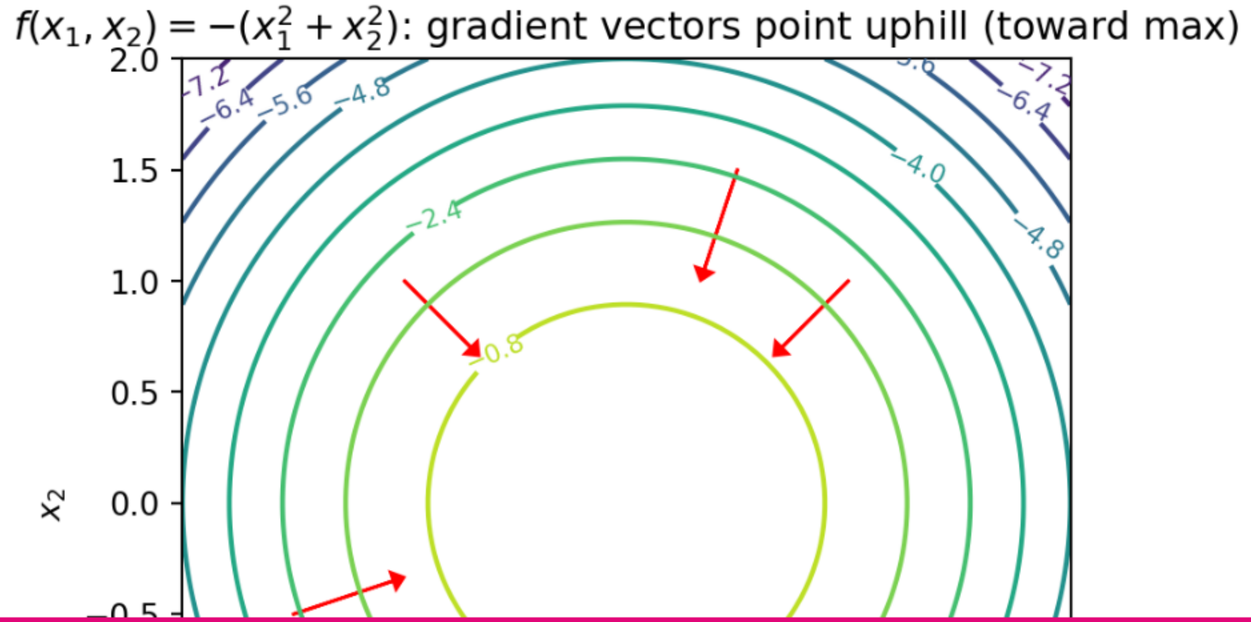
# Linear Algebra Application: The Gradient

The **gradient** is the vector of all partial derivatives:

$$\nabla f = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \\ \vdots \\ \frac{\partial f}{\partial x_p} \end{bmatrix}$$

Since the gradient is a vector, it points in a direction. Which direction? **The direction of steepest increase.**

To find a minimum: follow $-\nabla f$ (the direction of steepest *decrease*).

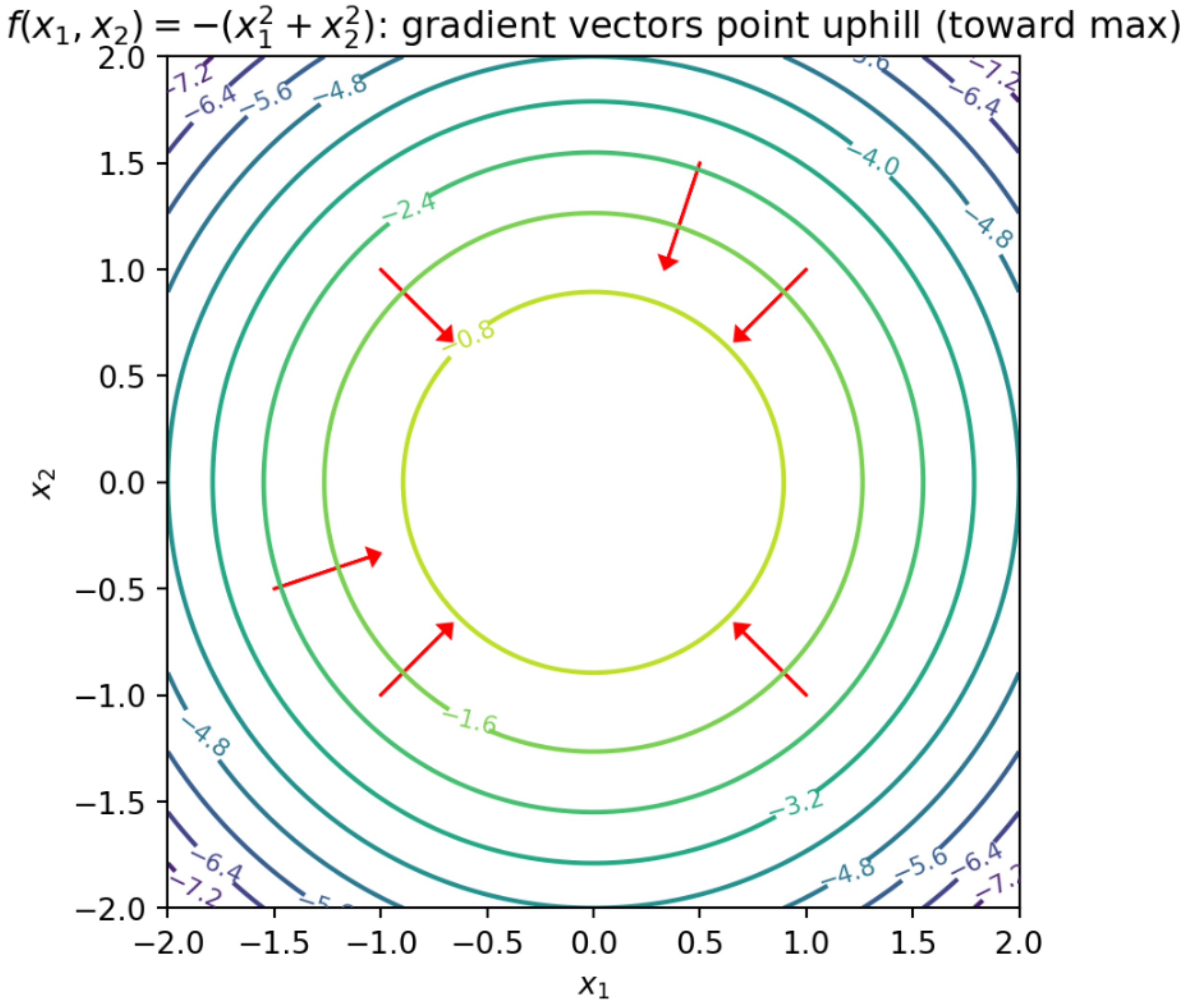


Rotman Commerce

$$\nabla f = \begin{bmatrix} \frac{}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \\ \vdots \\ \frac{\partial f}{\partial x_p} \end{bmatrix}$$

Since the gradient is a vector, it points in a direction. Which direction? **The direction of steepest increase.**

To find a minimum: follow $-\nabla f$ (the direction of steepest *decrease*).


$f(x_1, x_2) = -(x_1^2 + x_2^2)$: gradient vectors point uphill (toward max)

# Part IV: Optimization

# The Argmin Notation

When we write:

$$\theta^* = \arg\min_{\theta} \square(\theta)$$

We mean: “$\theta^*$ is the value of $\theta$ that minimizes $\square$.”

- $\min$ gives you the minimum *value* of the function
- $\arg\min$ gives you the *input* that achieves that minimum

**Example:** If $f(x) = (x - 3)^2$, then:

- $0 = \min_x f(x)$ (the minimum value)
- $3 = \arg\min_x f(x)$ (the $x$ that achieves it)

# Machine Learning Is Optimization

**The ML problem:**

Given data $(\mathbf{X}, \mathbf{y})$, find parameters $\boldsymbol{\theta}$ that minimize a loss function:

$$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} \square(\boldsymbol{\theta}; \mathbf{X}, \mathbf{y})$$

**For linear regression:**

- Loss function: $\square(\boldsymbol{\beta}) = \sum_{i=1}^{n} (y_i - \mathbf{x}_i' \boldsymbol{\beta})^2$
- Solution: $\hat{\boldsymbol{\beta}} = (\mathbf{X}'\mathbf{X})^{-1}\mathbf{X}'\mathbf{y}$

ⓘ **Advanced: Closed-Form Solutions vs. Algorithms**

OLS has a nice **closed-form solution**: we can write down a formula and compute the answer directly.

Most ML methods don't have this luxury. For neural networks, random forests, and many other models, there's no formula—we have to search for the optimum iteratively using **algorithms** like gradient descent. That's why we spend so much time on optimization in this course!

Everything we covered today is a building block for this: - Statistics tells us what we're estimating - Calculus tells us how to find minima - Linear algebra gives us compact notation - Optimization ties it all together

# What’s Next

| Week | Topic |
|---|---|
| 2 | Financial Data: returns, distributions, estimation |
| 3 | Introduction to Machine Learning |
| 4 | Clustering |
| 5 | Regression |
| 6 | ML & Portfolio Theory |
| 7-8 | Classification |
| 9 | Ensemble Methods |
| 10 | Neural Networks |
| 11 | Text & NLP |

# Appendix: Setting Up Your Python Environment

Rotman Commerce

# Step 1: Install VS Code

**Visual Studio Code** is a free, lightweight code editor that works on Windows, Mac, and Linux.

1. Go to https://code.visualstudio.com
2. Download the installer for your operating system
3. Run the installer and follow the prompts

VS Code is not the only option—you could use PyCharm, Jupyter notebooks directly, or even a plain text editor. But VS Code is free, widely used in industry, and works well for this course.

# Step 2: Install Python

**Windows:**

1. Go to https://www.python.org/downloads
2. Download the latest Python 3.x installer
3. **Important:** Check “Add Python to PATH” during installation
4. Run the installer

**Mac:**

Python 3 may already be installed. Open Terminal and type `python3 --version`. If not installed:

1. Install Homebrew (follow instructions at https://brew.sh)
2. Run: `brew install python`

Or download directly from python.org.

# Step 3: Install the Python Extension in VS Code

1. Open VS Code
2. Click the Extensions icon in the left sidebar (or press `Ctrl+Shift+X` / `Cmd+Shift+X`)
3. Search for “Python”
4. Install the extension by Microsoft (it should be the first result)

This extension provides:

- Syntax highlighting for Python files
- Code completion (IntelliSense)
- Ability to run Python code directly in VS Code
- Debugging support

# Step 4: Set Up Your Course Directory

Create a folder structure to keep your work organized:

```
RSM338/
├── Week01/
├── Week02/
├── Week03/
├── ...
├── Assignments/
└── Data/
```

**To create this:**

1. Create a folder called RSM338 somewhere convenient (e.g., Documents)
2. Open VS Code
3. File → Open Folder → select your RSM338 folder
4. In the Explorer sidebar, right-click to create new folders as needed

Each week, create a new `.py` file in the appropriate folder to follow along with lecture examples.

# Step 5: Install Required Packages

Open a terminal in VS Code (Ctrl+`` orCmd+``) and run:

```
pip install numpy pandas matplotlib scikit-learn
```

These are the core packages we'll use:

| Package | Purpose |
|---|---|
| `numpy` | Numerical computing (arrays, linear algebra) |
| `pandas` | Data manipulation (loading CSVs, working with tables) |
| `matplotlib` | Plotting and visualization |
| `scikit-learn` | Machine learning algorithms |

If `pip` doesn't work, try `pip3` instead.

# Verify Your Setup

Create a file called `test_setup.py` and paste this code:

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression

print("All packages imported successfully!")

# Quick test
x = np.array([1, 2, 3, 4, 5])
print(f"numpy works: {x.mean()}")
```

Run it by clicking the play button in the top-right corner of VS Code.

If you see “All packages imported successfully!” you’re ready to go.

# Troubleshooting

**“Python not found” or “pip not found”:**

- Make sure you checked “Add Python to PATH” during installation
- Try restarting VS Code after installing Python
- On Mac, use `python3` and `pip3` instead of `python` and `pip`

**Package installation fails:**

- Try: `python -m pip install package_name`
- On Mac/Linux, you may need: `pip install --user package_name`

**VS Code doesn’t recognize Python:**

- Press `Ctrl+Shift+P` (or `Cmd+Shift+P`), type “Python: Select Interpreter”
- Choose the Python installation you just installed

If you’re still stuck, office hours are a good time to troubleshoot setup issues.

# Reminders

Rotman Commerce

# Before You Go…

**Office Hours:** Thursdays 6–7pm

- Location: Management Data Analytics Lab, Rotman South Building, 3rd floor (take the pink stairs)

**This Week Only — Coding Cafe (Python Refresher):**

- Thursday 5:30–7:30pm (same location)
- First hour: Python basics
- Second hour: More advanced topics, Q&A

**Week 2 Preassessment:**

- In-class assessment worth 10% of your grade
- Based on today's lecture (math bootcamp material)
- Review the slides and course notes before next class